Sadržaj

Predgovor

Udžbenik pred vama je namenjen studentima druge godine Matematičkog fakulteta Univerziteta u Beogradu i koristi se za predmet "Konstrukcija i analiza algoritama" na drugoj godini studijskog programa Informatika.

Pretpostavlja se da su studenti u ranijem školovanju uspešno savladali osnovne elemente programiranja i da su ovladali osnovama algoritmike (asimptotskom analizom složenosti i O-notacijom, osnovnim tehnikama konstrukcije i analize algoritama i osnovnim strukturama podataka).

Teorijski pregledi algoritama i struktura podataka daju informacije na osnovu kojih bi čitalac trebalo da bude u mogućnosti da samostalno kreira implementaciju u programskom jeziku koji odabere. Ipak, ilustracije radi, algoritmi su implementirani u savremenom jeziku C++ uz intenzivno korišćenje standardne biblioteke tog programskog jezika. Znanje ovog jezika i biblioteke se, stoga, podrazumeva.

Materijal izložen u ovom udžbeniku prevazilazi okvire jednog jednosemestralnog kursa i na nastavniku je da odabere podskup tema koje želi da obradi u toku kursa. Neke teme se mogu i ostaviti studentima za samostalni rad.

Nakon svakog poglavlja dato je nekoliko zadataka koji ilustruju primene opisanih tehnika. Ti zadaci predstavljaju samo ilustraciju, a studentima koji žele da steknu bolje programerske veštine se savetuje da prorade veći broj zadataka koji se mogu naći u namenskim zbirkama zadataka iz oblasti algoritmike, ali i na mnogim onlajn portalima posvećenim učenju programiranja (na primer, `petlja.org`, `codeforces.com`, `leetcode.com`, `geeksforgeeks.org` itd.).

Knjiga ima i prateće online izdanje u kome se nalaze interaktivni apleti koji mogu pomoći studentima u razumevanju nekih tema.

Mole se čitaoci da na sve propuste i eventualne greške ukažu autorima.

Autori

1. Napredne strukture podataka

U ovom poglavlju prikazana je implementacija nekih naprednih struktura podataka. Pretpostavljamo da je čitalac upoznat sa korišćenjem i implementacijom osnovnih struktura podataka: sekvencijalnih struktura podataka (niza, jednostruko i dvostruko povezane liste), steka, reda, reda sa dva kraja, reda sa prioritetom, kao i osnovnih asocijativnih struktura podataka (skupa, multiskupa i mape, tj. rečnika) korišćenjem heš-tabela i balansiranih uređenih binarnih drveta. Za razliku od ovih elementarnijih struktura podataka, napredne strukture po pravilu nisu deo standardnih biblioteka programskih jezika (na primer, nisu uključene u biblioteke jezika C++, C#, Python, Java) i potrebno ih je posebno implementirati (ili preuzeti neku javno dostupnu implementaciju).

U ovom poglavlju ćemo proučiti sledeće strukture podataka:

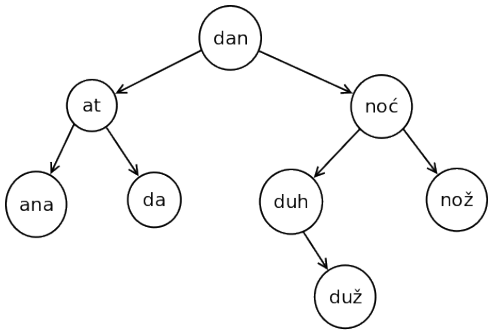
- **Prefiksno drvo** (engl. trie) omogućava da se na još jedan način implementiraju asocijativne strukture podataka (pored uređenih binarnih drveta i heš-tabela), kod kojih se umetanje i pretraga vrši na osnovu ključeva koji su obično ili niske karaktera ili niske dekadnih ili binarnih cifara. Ova drveta se obično koriste u primenama obrade teksta, kao što su provera pravopisa i obrada prirodnog jezika, zahvaljujući svojoj mogućnosti da efikasno skladište velike rečnike koji sadrže mnoštvo sličnih reči, preciznije reči koje dele zajedničke prefikse.
- **Fibonačijev hip** predstavlja još jedan način da se implementira red sa prioritetom (pored korišćenja klasičnog binarnog hipa), kod kojeg se, za razliku od klasičnog hipa, operacije umetanja, ali i smanjivanje prioriteta nekog elementa i spajanja dva hipa vrše u amortizovanom konstantnom vremenu (podsetimo se, ove operacije se kod klasičnih hipova vrše u logaritamskom vremenu).
- **Struktura za predstavljanje disjunktnih skupova** (engl. disjoint set union ili union find) omogućava da se održavaju kolekcije disjunktnih skupova (podskupova nekog skupa), uz mogućnost efikasnog pronalaženja skupa kome dati element pripada i efikasnog spajanja dva skupa u jedan.
- **Strukture za efikasno izvršavanje upita raspona** (engl. range queries) omogućavaju da se nakon određene predobrade (engl. preprocessing) niza elemenata efikasno izvršavaju operacije nad njegovim segmentima (podnizovima uzastopnih elemenata), poput, na primer, izračunavanja zbira elemenata segmenta, minimuma ili maksimuma elemenata segmenta itd. Neke od ovih struktura dopuštaju efikasno kombinovanje ažuriranja podataka (promenu pojedinačnih elemenata ili ažuriranja celih segmenata uvećanjem svih elemenata za neku vrednost) i izračunavanja pomenutih statistika.

1.1 Prefiksno drvo

Struktura podataka sa asocijativnim pristupom (skup, mapa tj. rečnik), kod koje se pristup elementima vrši po ključu, efikasno se implementira korišćenjem uređenog binarnog drveta ili heš-tabele[1]. Ključ ne mora biti celobrojna vrednost, već niska karaktera, niska bitova ili nešto drugo.

Primer 1.1.1

Na slici 1.1 prikazano je uređeno binarno drvo koje sadrži niske ana, at, noć, nož, da, dan, duh *i* duž *kao ključeve.*



Slika 1.1: Uređeno binarno drvo čiji su ključevi niske.

Prilikom svih operacija sa uređenim binarnim drvetom (pretraga, brisanje, umetanje) u svakom čvoru se vrši poređenje ključeva (onog koji je zapisan u čvoru i onog koji se obrađuje) i kada su ključevi niske (ali i neki drugi veći podaci), to poređenje može zahtevati dosta vremena i loše uticati na performanse.

Još jedna struktura podataka u vidu drveta koja omogućava efikasan asocijativni pristup kada su ključevi niske je *prefiksno drvo* (engl. prefix tree), takođe poznato pod engleskim nazivom *trie* (od engleske reči *reTRIEval*).

Problem

Definisati asocijativnu strukturu podataka koja omogućava interfejs skupa/mape (dodavanje, traženje i brisanje elemenata) u kojoj su ključevi niske ili nizovi, a koja podržava i efikasno pronalaženje svih elemenata čiji ključevi imaju zadati prefiks.

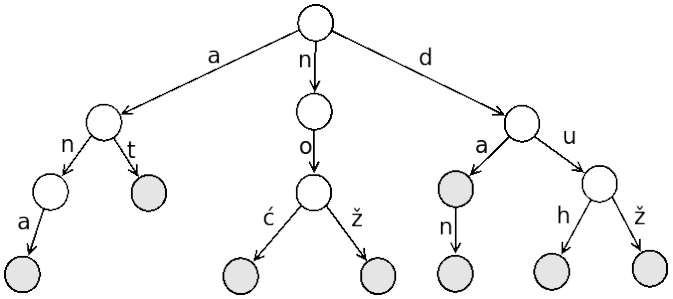
1.1.1 Struktura prefiksnog drveta

Osnovna ideja ove strukture podataka je da se ključ pridružen čvoru dobija nadovezivanjem karaktera koji se nalaze na granama duž putanje od korena do tog čvora. Koren sadrži praznu reč, a prelaskom preko grane se na do tada formiranu reč zdesna nadovezuje još jedan karakter, pa svakom čvoru odgovara neki prefiks ključa. Pritom, zajednički prefiksi različitih ključeva su predstavljeni istim putanjama od korena do tačke razlikovanja (čvora koji odgovara najdužem zajedničkom prefiksu). Čvorovi prefiksnog drveta mogu imati različit broj dece, ali maksimalni broj dece određen je veličinom azbuke koja se koristi za kodiranje ključeva. Ako se pomoću prefiksnog drveta implementira mapa (rečnik), tada se svakom ključu pridružuje vrednost (podatak koji se čuva u čvoru drveta kojim se kompletira taj ključ).

Primer 1.1.2

Jedan primer prefiksnog drveta dat je na slici 1.2.

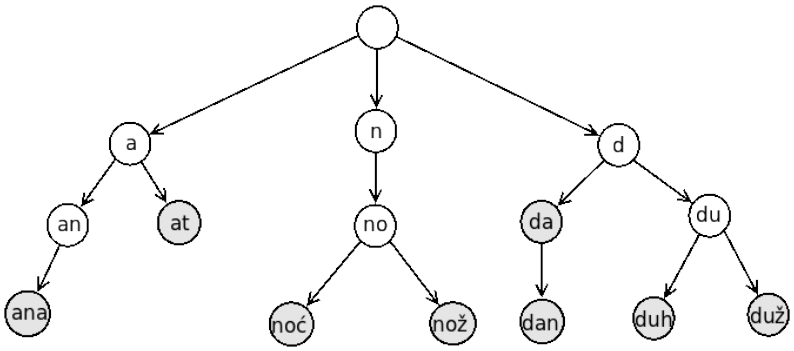
[1] Kod asocijativnih struktura podataka pristup elementima se vrši na osnovu vrednosti ključa, a ne na osnovu indeksa, odnosno pozicije elementa u strukturi podataka.



Slika 1.2: Primer prefiksnog drveta.

Ključevi koje ovo prefiksno drvo čuva su isti oni koje čuva uređeno binarno drvo sa slike 1.1: ana, at, noć, nož, da, dan, duh *i* duž *(bez pridruženih podataka). Većina ključeva koje prefiksno drvo u ovom primeru čuva se završava u nekom od listova. Međutim, to u opštem slučaju ne mora da važi: ključ* da *se ne završava u listu. Stoga je potrebno da svaki čvor prefiksnog drveta čuva i informaciju o tome da li se njime kompletira neki ključ ili ne (što se lako implementira dodavanjem odgovarajuće logičke promenljive u čvor drveta). Na slici 1.2 čvorovi kojima se kompletira neki ključ su obojeni.*

Ilustracije radi, na slici 1.3 uz čvorove prefiksnog drveta predstavljenog na slici 1.2 prikazane su oznake akumulirane do tih čvorova. Treba imati u vidu da se prikazani prefiksi dobijeni nadovezivanjem karaktera duž grana do svakog čvora ne čuvaju eksplicitno u čvoru.

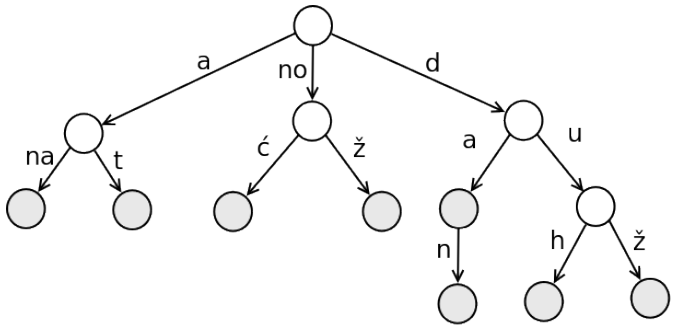


Slika 1.3: Primer prefiksnog drveta sa prikazanim akumuliranim prefiksima za svaki od čvorova.

Mana prefiksnog drveta može biti to što uz podatke koje čuva zauzima i puno dodatne memorije (za čuvanje pokazivača) i stoga je poželjno komprimovati ga. Ako neki čvor ima samo jednog potomka i ne predstavlja kraj nekog ključa, grana do njega i grana od njega se mogu spojiti u jednu, njihovi karakteri nadovezati, a čvor eliminisati. Ovako se dobija kompaktnija reprezentacija prefiksnog drveta kod koje svaki unutrašnji čvor ima bar dva deteta (slika 1.4), pri čemu grane mogu biti označene niskama, a ne samo pojedinačnim karakterima.

Pored toga toga što se prefiksnim drvetom mogu implementirati opšte asocijativne strukture kao skup i mapa, prefiksnim drvetom može se implementirati i konačni rečnik koji omogućava automatsko kompletiranje ili proveru ispravnosti, odnosno automatsko ispravljanje reči koje korisnik unosi na računaru ili mobilnom telefonu.

Ključevi u prefiksnom drvetu ne moraju biti isključivo niske karaktera. Na primer, u prefiksnom drvetu možemo čuvati i ključeve koji su prirodni brojevi, pri čemu se tada koristi niz cifara u njihovom dekadnom ili binarnom zapisu. Pored niski, najčešće se koriste binarne reprezentacije brojeva fiksne širine (zapisanih sa fiksnim brojem binarnih cifara).



Slika 1.4: Primer prefiksnog drveta kod koga su grane označene niskama.

1.1.2 Operacije nad prefiksnim drvetom

Operacije traženja i umetanja elemenata u prefiksno drvo vrše se na prilično očigledan način, dok je u slučaju brisanja nekada potrebno brisati više od jednog čvora.

Ispitivanje da li se neka reč nalazi u prefiksnom drvetu može se realizovati rekurzivnom funkcijom koja kao argumente dobija koren drveta i reč koju traži u drvetu (tokom rekurzivnih poziva u pitanju je koren odgovarajućeg poddrveta prefiksnog drveta i neki sufiks reči koja se pretražuje). Ako je reč prazna, ona se nalazi u drvetu ako i samo ako je koren obeležen kao kraj ključa. Ako reč nije prazna, da bi se ona mogla nalaziti u drvetu potrebno je da postoji dete korena do kog se stiže preko njenog prvog slova i tada pretragu nastavljamo rekurzivno od tog čvora za sufiks reči bez tog prvog slova.

I operaciju umetanja reči u prefiksno drvo možemo implementirati kao rekurzivnu funkciju koja kao argumente dobija koren drveta i reč koju treba ubaciti u to drvo. Ako je reč prazna, obeležavamo da se u korenu nalazi kraj reči. U suprotnom proveravamo da li postoji dete korena do kog se stiže prvim slovom te reči i ako ne postoji dodajemo ga. Zatim rekurzivno nastavljamo umetanje od tog čvora i umećemo sufiks reči bez tog prvog slova.

U nastavku su date implementacije osnovnih operacija nad prefiksnim drvetom na primeru formiranja i pretrage skupa reči na engleskom jeziku. U ovom primeru ključevi su reči (niske karaktera) i nisu im pridružene nikakve vrednosti. Potrebno je podržati operaciju dodavanja ključeva u strukturu podataka i provere da li ključ postoji u strukturi. Čvor prefiksnog drveta može se definisati tako da sadrži niz pokazivača, čijim elementima odgovaraju svi mogući karakteri azbuke iz koje se formiraju ključevi (npr. pošto se u ovom primeru kodiraju samo reči koje se sastoje od malih slova engleske abecede, možemo koristiti niz pokazivača dužine 26). Međutim, efikasnije je u svakom čvoru čuvati informacije samo o onim karakterima za koje postoji grana iz tog čvora, odnosno u ove svrhe iskoristiti mape. Dakle, u svakom čvoru prefiksnog drveta čuvamo neuređenu (heš) mapu koja karakterima pridružuje grane koje kreću iz tog čvora ka njegovoj deci, pri čemu koristimo bibliotečku implementaciju heš-mape (klasu `unordered_map` deklarisanu u istoimenom zaglavlju). Operacije umetanja i pretrage se mogu jednostavno implementirati bilo rekurzivno bilo iterativno (u nastavku je prikazana rekurzivna implementacija).

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

// osnovna struktura čvora prefiksnog drveta - u svakom čvoru čuvamo mapu
// koja karakterima pridružuje pokazivače ka potomcima kao i informaciju
// o tome da li je u čvoru kraj neke reči
struct cvor {
  bool krajKljuca = false;
  unordered_map<char, cvor*> grane;
```

```
};

// sufiks koji počinje na poziciji i reči w tražimo
// u drvetu na čiji koren ukazuje pokazivač drvo
bool nadji(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan, on je u korenu akko je u korenu obeleženo
  // da je tu kraj reči
  if (i == w.size())
    return drvo->krajKljuca;

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako je nađemo, rekurzivno tražimo ostatak sufiksa od pozicije i+1
  if (it != drvo->grane.end())
    return nadji(it->second, w, i+1);

  // nismo našli granu sa w[i], pa reč ne postoji
  return false;
}

// tražimo reč w u drvetu na čiji koren ukazuje pokazivač drvo
bool nadji(cvor *drvo, const string& w) {
  return nadji(drvo, w, 0);
}

// umetanje sufiksa koji počinje na poziciji i reči w
// u drvo na čiji koren ukazuje pokazivač drvo
void umetni(cvor *drvo, const string& w, int i) {
  // ako je sufiks prazan samo u korenu beležimo da je tu kraj reči
  if (i == w.size()) {
    drvo->krajKljuca = true;
    return;
  }

  // tražimo granu na kojoj piše w[i]
  auto it = drvo->grane.find(w[i]);
  // ako takva grana ne postoji, dodajemo je kreirajući novi čvor
  if (it == drvo->grane.end())
    drvo->grane[w[i]] = new cvor();

  // sada znamo da grana sa w[i] sigurno postoji i preko te grane
  // nastavljamo dodavanje sufiksa koji počinje na poziciji i+1
  umetni(drvo->grane[w[i]], w, i+1);
}

// umetanje reči w u drvo na čiji koren ukazuje pokazivač drvo
void umetni(cvor *drvo, string& w) {
  return umetni(drvo, w, 0);
}

// brisanje drveta sa korenom u čvoru drvo
```

```
void obrisi(cvor *drvo) {
  if (drvo != nullptr) {
    for (const auto& p : drvo->grane)
      obrisi(p.second);
    delete drvo;
  }
}
```

Prethodne funkcije možemo testirati sledećim programom: u njemu se najpre neke reči ubacuju u prefiksno drvo, a zatim se za te reči i neke druge reči proverava da li se nalaze u drvetu. Za reči koje se nalaze u drvetu program ispisuje `da`, dok za reči koje se u njemu ne nalaze ispisuje `ne`.

```
// program kojim testiramo gornje funkcije
int main() {
  cvor* drvo = new cvor();
  vector<string> reci
    {"ana", "at", "noc", "noz", "da", "dan", "duh", "duz"};
  vector<string> reci_neg
    {"", "a", "d", "ananas", "marko", "ptica"};
  for (auto w : reci)
    umetni(drvo, w);
  for (const string& w : reci)
    cout << w << ": " << (nadji(drvo, w) ? "da" : "ne") << endl;
  for (const string& w : reci_neg)
    cout << w << ": " << (nadji(drvo, w) ? "da" : "ne") << endl;
  obrisi(drvo);
  return 0;
}
```

Kada azbuka kojom se kodiraju ključevi ima K elemenata i kada se u svakom čvoru čuva neuređena mapa grana, složenost operacija pretraživanja, umetanja i brisanja elementa iz prefiksnog drveta je u najgorem slučaju $O(mK)$, gde je m dužina reči koja se traži, umeće ili briše. Zaista, složenost najgoreg slučaja pretrage neuređene mape je $O(K)$, a prilikom obrade ključa dužine m vrši se m takvih pretraga. Sa druge strane, amortizovana složenost pretrage neuređene mape je $O(1)$, pa je amortizovana složenost ovih operacija $O(m)$. Ako se radi sa niskama karaktera i ako se neuređena mapa implementira pomoću niza od K elemenata, onda je i složenost najgoreg slučaja $O(m)$. Primetimo da je prilikom pretrage broj operacija ograničen i dužinama reči koje se nalaze u drvetu, pa se, preciznije, složenost može ograničiti i sa $O(\min(m, M))$, gde je M dužina najduže reči kojia se nalazi u drvetu.

Dobra strana prefiksnog drveta je to što složenost umetanja i pretrage zavisi od dužine zapisa ključa, a ne od broja elemenata koji se čuvaju u drvetu. Mana je potreba za čuvanjem pokazivača uz svaki čvor u drvetu. Štaviše, prostorna složenost prefiksnog drveta u najgorem slučaju iznosi $O(MNK)$, gde je sa N označen broj ključeva koji se čuvaju u prefiksnom drvetu, a sa M maksimalna dužina ključa. Naime, maksimalni mogući broj čvorova prefiksnog drveta jednak je $O(MN)$ i odgovara slučaju kada nema nikakvog preklapanja karaktera među ključevima, dok je prostorna složenost svakog čvora jednaka $O(K)$ zbog potrebe čuvanja mape u svakom čvoru. Primetimo da je očekivana prostorna složenost manja jer će se u slučaju realne konačne azbuke (na primer, engleske abecede) prvo preklapanje javiti najkasnije nakon 26 ključeva.

Smanjenje memorijske složenosti se može postići i tako što se smanji veličina azbuke (po cenu povećanja dužine ključa). Na primer, umesto 256 različitih 8-bitnih karaktera, možemo svaki karakter podeliti na dve 4-bitne polovine. Na ovaj način dobija se samo 16 različitih 4-bitnih karaktera, ali se dužina svakog ključa dva puta povećava.

Kada bi se umesto prefiksnog drveta koristilo balansirano uređeno binarno drvo koje bi čuvalo kompletne ključeve u čvorovima (slika 1.1), vremenska složenost operacija pretraživanja, umetanja i brisanja bi u najgorem slučaju bila $O(M \cdot \log N)$, gde je sa N označen ukupan broj ključeva koji se čuvaju u drvetu, a sa M maksimalna dužina ključa. Prostorna složenost ove strukture je $O(M \cdot N)$.

Kada bi se koristila heš tabela, prilikom svake operacije umetanja i pretrage bi morala da bude izračunata heš vrednost ključa koji se traži za šta je potrebno vreme $O(M)$. U zavisnosti od broja kolizija, vršila bi se poređenja heš vrednosti (u najgorem slučaju njih $O(N)$, a amortizovano $O(1)$). Na kraju bi ključ koji se traži i ključ sloga koji je pronađen na osnovu jednakosti heš-vrednosti morali da budu eksplicitno upoređeni, za šta je takođe potrebno vreme $O(M)$ (a ako postoje kolizije ovo poređenje bi moralo da se izvrši nekoliko puta). Zato bi složenost najgoreg slučaja operacija bila $O(N+M)$, a amortizovana složenost $O(M)$. Pošto ključ mora biti zapisan eksplicitno u svakom slogu tabele, prostorna složenost pristupa zasnovanog na heš tabelama je $O(M \cdot N)$.

U tabeli 1.1 prikazana je (amortizovana) složenost raznih implementacija skupova/mapa. Pretpostavljamo da je dužina reči koja se obrađuje m, broj reči u kolekciji N, a veličina azbuke K.

Tabela 1.1: Složenost različitih implementacija rečnika

	Heš tabela	Balansirano binarno uređeno drvo	Prefiksno drvo
Pretraga	$O(m)$	$O(m \log N)$	$O(m)$
Umetanje	$O(m)$	$O(m \log N)$	$O(m)$
Brisanje	$O(m)$	$O(m \log N)$	$O(m)$
Prostor	$O(mN)$	$O(mN)$	$O(mNK)$

Dakle, možemo primetiti da prefiksno drvo nema lošiju složenost od heš-tabela, ali podržava novu operaciju (pronalaženje svih reči koje imaju dati prefiks) i stoga se koristi prilikom rešavanja problema u kojima je ta operacija korisna.

Zadatak: Par koji daje najveći XOR

Napiši program koji među unetim neoznačenim brojevima određuje onaj par koji daje najveći rezultat pri operaciji ekskluzivne disjunkcije (XOR) njihovih binarnih zapisa.

Opis ulaza

Sa standardnog ulaza se unosi broj n ($1 \leq n \leq 100000$), a zatim u narednom redu n prirodnih brojeva između 0 i 10^{18}.

Opis izlaza

Na standardni izlaz ispisati maksimalnu vrednost koja se može dobiti kada se ekskluzivna disjunkcija primeni na neka dva uneta broja.

Primer

Ulaz

```
5
1 2 3 4 5
```

Izlaz

```
7
```

Objašnjenje

Najveći rezultat 7 dobija se ekskluzivnom disjunkcijom brojeva 3 i 4 (njihovi binarni zapisi su `00...0000011` i `000...000100`). Isti rezultat dobija se i ekskluzivnom disjunkcijom brojeva 2 i 5 (njihovi binarni zapisi su `0000..0000101` i `0000...00000010`).

Rešenje

Rešenje grubom silom podrazumeva da se na svaki par brojeva primeni operacija XOR i da se ispiše maksimum dobijenih rezultata. Složenost ovog pristupa je $O(n^2)$.

Efikasnije rešenje možemo dobiti primenom naprednih struktura podataka. Pokušajmo da za svaki novi uneti broj efikasno izračunamo najveći broj koji se može dobiti primenom operacije XOR na njega i neki od prethodno unetih brojeva (u rešenju grubom silom, to se dešava u unutrašnjoj petlji). Pokušajmo da taj broj odredimo bit-po-bit i to krenuvši od bitova najveće težine. Na mestu bita najveće težine možemo dobiti 1 ako tekući broj počinje bitom 0 i među ranije učitanim brojevima postoji neki koji počinje bitom 1 ili ako tekući broj počinje bitom 1 i među ranije učitanim brojevima postsoji neki koji počinje bitom 0. U suprotnom, na mestu najveće težine rezultata mora biti bit 0. Nakon određivanja prvog bita, određujemo naredni, ali u slučaju da smo na vodeće mesto rezultata upisali 1, među učitanim niskama zadržavamo samo one koje su na vodećem mestu imali bit suprotan bit vodećem bitu tekućeg broja (u slučaju da smo na vodeće mesto rezultata upisali 0, tada su svi ranije učitani brojevi počinjali istim bitom kojim počinje tekući broj i svi se zadržavaju). Postupak sada ponavljamo za drugi bit, pri čemu razmatramo samo niske koje nisu ranije odbačene.

Primer 1.1.3

Pretpostavimo da su dati brojevi

```
1001
1010
0110
```

i da je tekući broj

```
0010
```

Na vodeće mesto rezultata možemo upisati 1, pri čemu zadržavamo niske

```
1001
1010
```

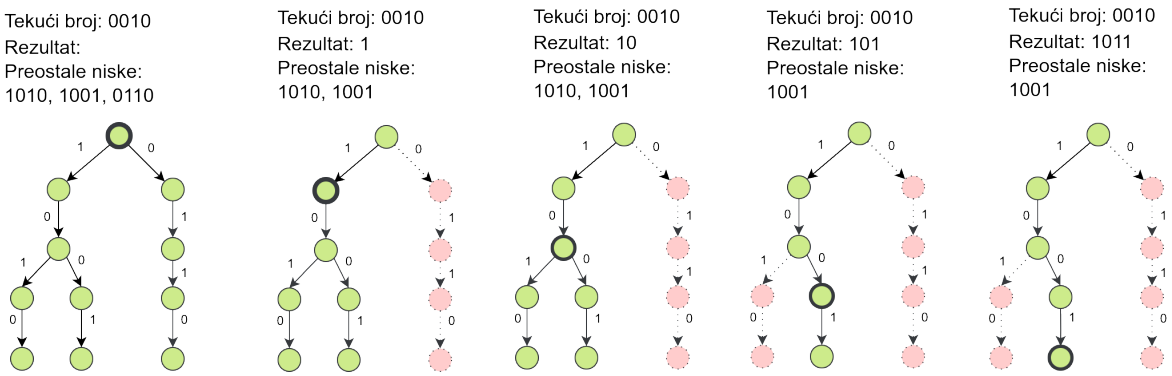
Na drugo mesto rezultata moramo upisati 0, jer sve zadržane niske na mestu drugog bita imaju 0, isto kao i tekući broj. Obe niske se zadržavaju.

Na treće mesto rezultata možemo upisati 1 i tada zadržavamo samo nisku

```
1001
```

Na kraju, na poslednje mesto rezultata možemo upisati 1. Konačan rezultat je, dakle, `1011` *i on se dobija primenom operacije XOR na niske* `0010` *i* `1001`*.*

Pretragu možemo veoma jednostavno organizovati ako sve učitane binarne zapise jedan po jedan upisujemo u prefiksno drvo. Prilikom obrade tekućeg broja, njegove bitove prolazimo sleva nadesno, za svaki bit se spuštajući na naredni nivo drveta. Na svakom nivou drveta, dakle, određujemo jedan bit rezultata. Ako na tekućem nivou postoji grana obeležena bitom suprotnom od tekućeg bita trenutnog broja, spuštamo se njome i u rezultat upisujemo jedinicu, dok se u suprotnom spuštamo granom na kojoj se nalazi bit jednak tekućem bitu trenutnog broja i u rezultat upisujemo nulu. Spuštanjem na naredni nivo drveta ujedno eliminišemo sve one niske koje na odgovarajućem mestu nemaju potreban bit. Ovaj postupak je ilustrovan na slici 1.5.



Slika 1.5: Upotreba prefiksnog drveta

Složenost ovog algoritma je $O(n)$, pri čemu je konstanta jednaka broju bitova kojima se zapisuju brojevi (s obzirom na ograničenja data u zadatku, to je 64).

```
#include <iostream>

using namespace std;

typedef unsigned long long ull;

struct Cvor {
  Cvor* grane[2];
};

Cvor* noviCvor() {
  Cvor* novi = new Cvor();
  novi->grane[0] = novi->grane[1] = nullptr;
  return novi;
}

void ubaci(Cvor* koren, ull broj) {
  Cvor* cvor = koren;
  ull mask = 1ull << (8*sizeof(ull) - 1);
  while (mask != 0) {
    int bit = (broj & mask) != 0;
    if (cvor->grane[bit] == nullptr)
      cvor->grane[bit] = noviCvor();
    cvor = cvor->grane[bit];
    mask >>= 1;
  }
}

void obrisi(Cvor* koren) {
  if (koren != nullptr) {
    obrisi(koren->grane[0]);
    obrisi(koren->grane[1]);
    delete koren;
  }
}
```

```
ull maxXOR(Cvor* koren, ull broj) {
  Cvor* cvor = koren;
  ull rez = 0;
  ull mask = 1ull << (8*sizeof(ull) - 1);
  while (mask != 0) {
    int bit = (broj & mask) != 0;
    if (cvor->grane[!bit] != nullptr) {
      rez = rez | mask;
      cvor = cvor->grane[!bit];
    } else
      cvor = cvor->grane[bit];
    mask >>= 1;
  }
  return rez;
}

int main() {
  int n;
  cin >> n;
  unsigned long long x;
  cin >> x;
  Cvor* koren = noviCvor();
  ubaci(koren, x);
  unsigned long long max = 0;
  for (int i = 1; i < n; i++) {
    cin >> x;
    unsigned long long rez = maxXOR(koren, x);
    if (rez > max)
      max = rez;
    ubaci(koren, x);
  }
  cout << max << endl;
  obrisi(koren);
  return 0;
}
```

1.2 Fibonačijev hip

Redovi sa prioritetom se obično implementiraju korišćenjem strukture podataka hip. Za razliku od binarnog hipa u kom operacije umetanja elementa ima logaritamsku složenost, Fibonačijev hip je struktura podataka kod koje operacije umetanja novog elementa u strukturu ima konstantnu amortizovanu složenost, što je čini bržom. Dodatno, Fibonačijev hip omogućava i spajanje dva hipa u jedan kao i smanjivanje vrednosti ključa u konstantnoj amortizovanoj složenosti, dok složenost izbacivanja minimuma ostaje logaritamska. Dakle, Fibonačijev hip uspešno rešava sledeći problem.

Problem

Definisati strukturu podataka koja podržava efikasno izvršavanje (u amortizovanom konstantnom ili logaritamskom vremenu) sledećih operacija:

- `napravi_hip()` - *pravi novi prazni hip*

- `umetni(H, v)` - *umeće element sa vrednošću ključa* `v` *u hip* `H`
- `minimum(H)` - *vraća element hipa* `H` *sa minimalnom vrednošću ključa*
- `izbaci_minimum(H)` - *briše element hipa* `H` *sa minimalnom vrednošću ključa i vraća ga kao rezultat*
- `unija(H1,H2)` - *od dva hipa* `H1` *i* `H2` *pravi novi hip koji sadrži sve elemente polaznih hipova*
- `smanji_kljuc(H, x, v)` - *elementu* `x` *hipa* `H` *menja vrednost ključa novom vrednošću* `v`*, pri čemu je* `v` *manja ili jednaka od trenutne vrednosti ključa elementa* `x`

Implementacija operacije `smanji_kljuc` zahteva pamćenje dodatnih podataka u strukturi, pa je implementacija hipa koji podržava tu operaciju malo komplikovanija nego implementacija hipa koji podržava sve ostale navedene operacije.

Amortizovana složenost operacija pravljenja hipa, umetanja elementa, određivanja minimuma, pravljenja unije i smanjivanja vrednosti ključa je $O(1)$, dok je amortizovana složenost operacije brisanja minimalnog elementa iz hipa $O(\log n)$. Dakle, operacije umetanja elementa i pravljenja unije se efikasnije izvršavaju nad Fibonačijevim hipom, nego nad klasičnim binarnim hipom (podsetimo se, složenost dodavanja elemenata u binarni hip je $O(\log n)$).

Fibonačijev hip je koristan kada je broj operacija brisanja minimalnog elementa iz hipa mali u odnosu na ostale pomenute operacije. Na primer, u grafovskim algoritmima kao što je određivanje razapinjućeg (povezujućeg) drveta minimalne cene ili najkraćih puteva iz zadatog čvora, ako je graf gust (sadrži veliki broj grana), operacija smanjivanja vrednosti ključa se može često javljati, te ubrzanje sa $O(\log n)$ kod binarnog hipa na $O(1)$ kod Fibonačijevog hipa može doneti osetno ubrzanje. Nedostatak Fibonačijevog hipa je to što je dosta teži za implementaciju od klasičnog, binarnog hipa, to što zahteva više memorije, kao i to što je složenost najgoreg slučaja nekih operacija velika.

Fibonačijev hip je dobio naziv prema Fibonačijevim brojevima na osnovu kojih se formuliše centralna invarijanta koja garantuje dobru složenost operacija. Ovu strukturu podataka osmislili su Fridman i Tardžan sa ciljem da se poboljša vreme izvršavanja Dajkstrinog algoritma za određivanje najkraćih puteva iz zadatog čvora. Međutim, ona ima primene i u drugim grafovskim algoritmima, poput Primovog algoritma za računanje minimalnog povezujućeg drveta, za određivanje maksimalnog toka kroz mrežu, ali i u drugim domenima, poput geometrijskih algoritama, obrade slika i matematičke optimizacije.

1.2.1 Struktura hipa

Fibonačijev min-hip predstavlja kolekciju min-hipova. Svaki mih-hip je drvo u kom čvorovi mogu imati različit broj naslednika, a vrednost u svakom čvoru je manja ili jednaka vrednosti u njegovim naslednicima. Redosled drveta u Fibonačijevom hipu je proizvoljan. Fibonačijevom hipu pristupamo putem pokazivača na koren drveta sa minimalnom vrednošću u celom Fibonačijevom hipu – ovaj čvor zovemo *minimalnim čvorom* Fibonačijevog hipa (ukoliko postoji više čvorova sa minimalnom vrednošću bilo koji od njih se može proglasiti minimalnim čvorom). Jedan primer Fibonačijevog min-hipa prikazan je na slici 1.7. Koreni svih drveta u Fibonačijevom hipu se povezuju u kružnu, dvostruko povezanu listu koju nazivamo *lista korenova* hipa. I sva deca bilo kog čvora su međusobno povezana u kružnu, dvostruko povezanu listu. Svaki čvor (bilo da je koren ili ne) sadrži pokazivač na levog i desnog suseda i na neko dete (a ako je potrebno vršiti i operacije smanjivanja vrednosti ključeva, onda i pokazivač na roditelja i još neke pomoćne podatke koje ćemo opisati u sklopu opisa te operacije). Korišenje kružnih lista omogućava umetanje novih čvorova u te liste i brisanje čvorova iz lista u vremenu $O(1)$. Takođe, dve ovakve liste možemo objediniti u novu listu takođe u vremenu $O(1)$ (videćemo da je ovo bitno za efikasno izvođenje operacije formiranja unije). Ove operacije nad listama su gradivni elementi operacija nad hipom i njihova efikasnost garantuje efikasno izvršavanje operacija nad hipom.

Broj dece čvora naziva se *stepen čvora*. Navedimo sada ključnu invarijantu (uslov koji važi nakon formiranja strukture i nakon primene bilo koje operacije za koji ćemo videti da garantuje složenost operacija).

Invarijanta (broj čvorova poddrveta): Svako poddrvo Fibonačijevog hipa čiji je koren stepena d sadrži bar F_{d+2} čvorova, gde je sa F_k označen k-ti Fibonačijev broj ($F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$).

Dakle, drvo čiji je koren stepena 0 ima bar 1 čvor, stepena 1 ima bar 2 čvora, stepena 2 ima bar 3 čvora, stepena 3 ima bar 5 čvorova, stepena 4 ima bar 8 čvorova itd.). Pošto se lako (indukcijom) dokazuje da je $F_{d+2} \geq \varphi^d$, gde je $\varphi = \frac{1+\sqrt{5}}{2}$, za stepen d svakog čvora koji je koren nekog poddrveta u kom se nalazi m čvorova važi $m \geq F_{d+2} \geq \varphi^d$ tj. $d \leq \log_\varphi m$. Dakle, broj dece svakog čvora koji je koren poddrveta sa m elemenata je $O(\log m)$. Ako u celom Fibonačijevom hipu ima n elemenata, stepen bilo kog čvora je $O(\log n)$. Dakle, pod uslovom da invarijanta važi, liste dece su relativno kratke i prolazak kroz sve elemente takve liste vrši se u logaritamskoj složenosti (u odnosu na ukupan broj elemenata u hipu).

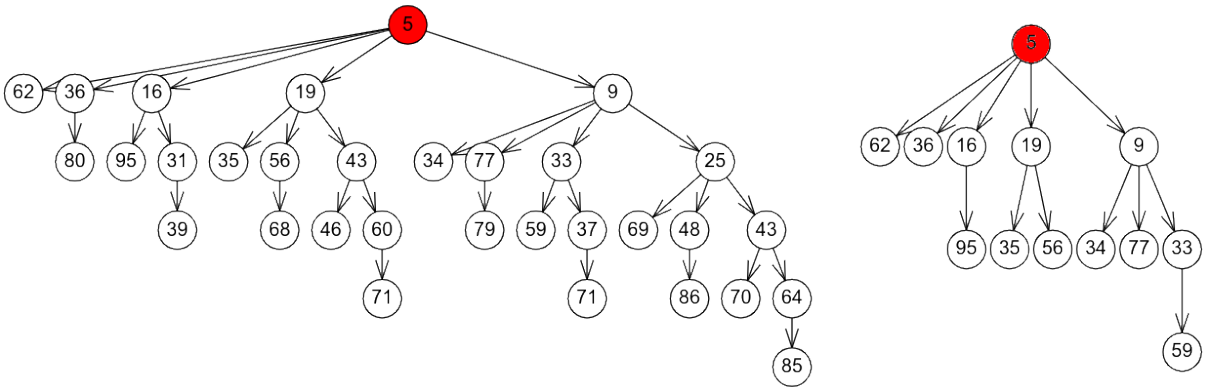
Formulišimo sada drugačiju invarijantu, koju je lakše proveravati, a koja daje garancije da će važiti invarijanta o broju čvorova u poddrvetima.

Invarijanta (stepeni dece): Za svaki čvor x stepena d važi da su stepeni dece bar $0, 0, 1, 2, \ldots, d-2$.

Indukcijom se dokazuje da invarijanta o stepenima dece garantuje invarijantu o broju čvorova poddrveta. Ako je $d = 0$, čvor nema dece, a u njegovom poddrvetu se nalazi $F_2 = 1$ čvor. Ako je invarijanta o stepenima dece ispunjena, na osnovu induktivne hipoteze važi da poddrveta čiji su koreni deca čvora x sadrže redom $F_2, F_2, F_3, \ldots F_d$ čvorova, pa, pošto je $F_2 = F_1 = 1$ i $F_0 = 0$, važi da je ukupan broj čvorova u drvetu čiji je koren x jednak $1 + F_0 + F_1 + F_2 + F_3 \ldots + F_d$ (prva jedinica dolazi od samog korena x). Indukcijom se rutinski može dokazati da je $1 + F_0 + F_1 + F_2 + \ldots + F_d = F_{d+2}$, pa invarijanta o stepenima dece zaista implicira invarijantu o broju čvorova poddrveta.

Ova invarijanta nameće određene stepene dece, ali nije precizirano koja deca treba da imaju tražene stepene. Deca se razmatraju u redosledu njihovog dodavanja roditeljskom čvoru i njihovi stepeni u tom redosledu zadovoljavaju invarijantu (a ne po redosledu kojim su smešteni u listu dece). Dakle, uslov koji je dovoljan da bi hip bio Fibonačijev je da su stepeni dece u redosledu dodavanja $0, 0, 1, 2, 3, \ldots$ tj. da za svaki čvor važi da njegovo dete sa rednim brojem dodavanja i (brojimo od nule) ima stepen $d_i \geq i - 1$ (pri čemu je i $d_i \geq 0$, jer je d_i prirodan broj).

Videćemo uskoro da će procedura formiranja hipa biti takva da su stepeni dece svakog čvora drveta neposredno nakon njegovog prvog formiranja biti $0, 1, 2, 3, \ldots$ (što je i više nego što se zahteva invarijantom o stepenima dece). Na osnovu ovoga se može dokazati da svako poddrvo čiji je stepen korena d ima 2^d čvorova, što takođe garantuje povoljnu složenost operacija. Ipak, invarijanta je oslabljena da bi bilo moguće uklanjati čvorove iz formiranog drveta, što je, videćemo, veoma važno za realizaciju operacije smanjivanja vrednosti ključa (engl. `smanji_kljuc`).



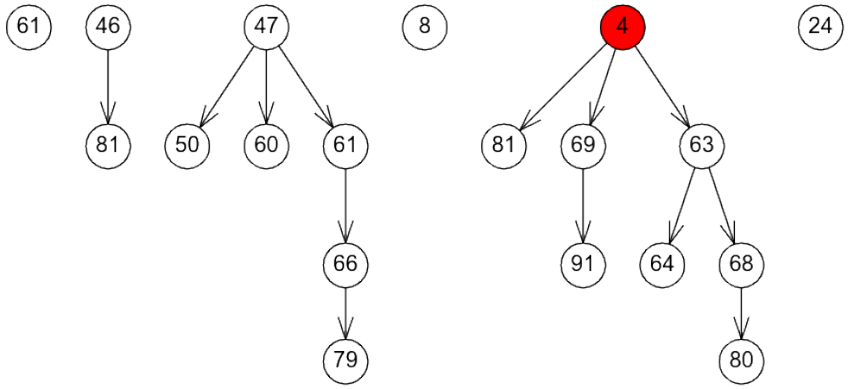
Slika 1.6: Drvo u kom ima $2^5 = 32$ čvora (stepeni čvorova su redom, $0, 1, 2, 3, \ldots$). Drvo u kom ima $F_7 = 13$ čvorova (stepeni čvorova su redom $0, 0, 1, 2, 3, \ldots$)

Na slici 1.6 levo vidi se najgušće moguće popunjeno drvo (koje se dobija odmah nakon formiranja drveta koje sadrži $2^5 = 32$ čvora), dok se desno vidi najređe moguće drvo koje zadovoljava invarijantu i koje sadrži $F_{5+2} = 13$ čvorova. Izbacivanjem bilo kog čvora iz desnog drveta bila bi narušena invarijanta.

Prikažimo, napokon, jedan kompletan primer Fibonačijevog hipa.

Primer 1.2.1

Na slici 1.7 prikazan je Fibonačijev hip koji se sastoji od 6 *min-hipova. Minimalni element u hipu je minimum svih korenova, a to je element sa ključem* 4.



Slika 1.7: Primer Fibonačijevog hipa

Direktnom proverom se lako može pokazati invarijanta o stepenima dece. Ona je uvek trivijalno ispunjena za čvorove čiji je stepen 0, 1 *i* 2 *(jer su stepeni dece uvek veći ili jednaki od 0 što se invarijantom traži). Deca čvora* 47 *imaju redom stepene* 0, 0 *i* 1, *a deca čvora* 4 *imaju redom redom stepene* 0, 1 *i* 2. *Dakle, za svaki čvor jeste zadovoljena invarijanta o stepenima dece (stepeni dece su na nekim mestima veći nego što je potrebno), a ona garantuje i invarijantu o veličinama tj. ukupnom broju čvorova tih drveta. Zaista, drveta koja čine Fibonačijev hip redom imaju* 1, 2, 6, 1, 8, 1 *i* 1 *čvor, što je ponekad i više od odgovarajućih Fibonačijevih brojeva* 1, 2, 5, 1, 5, 1 *i* 1. *Isto važi i za sva njihova poddrveta, pa je centralna invarijanta o veličinama svih drveta zadovoljena.*

1.2.2 Operacije nad hipom

Jedna od osnovnih karakteristika operacija nad Fibonačijevim hipom je lenjo izvršavanje operacija, tj. posao se odlaže za što je moguće kasnije.

1.2.2.1 Umetanje

Umetanje novog elementa u Fibonačijev hip (operacija `insert(H, x)`) se vrši tako što se novi element dodaje u listu korenova, ažurirajući minimalni čvor, ako je potrebno, što je konstantne vremenske složenosti. Dakle, umetanjem k elemenata u prazan Fibonačijev hip dobija se Fibonačijev hip čija lista korenova ima k elemenata.

Primer 1.2.2

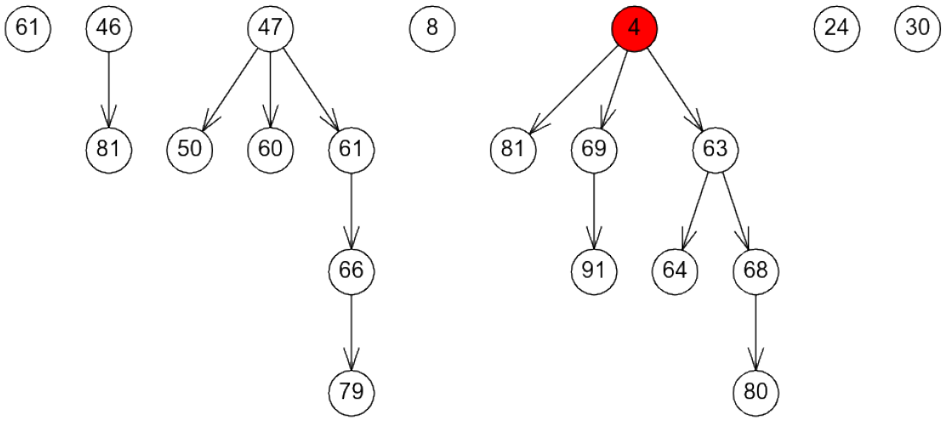
Umetanjem elementa sa ključem 30 *u prethodni Fibonačijev hip dobija se hip prikazan na slici 1.8 (novi element je dodat na kraj liste korenova, ali mogao je biti dodat i na bilo koje drugo mesto, najčešće je to neposredno iza ili ispred minimalnog čvora, na koji, podsetimo se, ukazuje poseban pokazivač).*

1.2.2.2 Unija dva hipa

Unija dva Fibonačijeva hipa H_1 i H_2 (operacija `unija(H1, H2)`) se pravi nadovezivanjem listi korenova ova dva hipa i određivanjem novog minimalnog čvora (to je manji od dva minimalna čvora polaznih hipova). Ovo je takođe operacija konstantne složenosti.

1.2.2.3 Određivanje minimuma

Određivanje minimalnog elementa u Fibonačijevom hipu (operacija `minimum(H)`) je trivijalno, s obzirom na to da se u svakom trenutku čuva pokazivač na minimalni element u strukturi. Ovo je takođe operacija konstantne složenosti.



Slika 1.8: Umetanje elementa 30

1.2.2.4 Brisanje najmanjeg elementa

Operacija brisanja najmanjeg elementa iz Fibonačijevog hipa (operacija `izbaci_minimum(H)`) se izvodi na sledeći način: prvo se sva njegova deca umeću u listu korenova, i on se briše iz liste korenova. Da bi se ažurirao minimum hipa potrebno je potencijalno proći kroz celu listu korenova, koja može biti veoma duga. Zbog toga se pre ažuriranja minimuma vrši jedna posebna pomoćna operacija koja se naziva *konsolidacija* hipa, odnosno hip se dovodi u stanje u kom svi korenovi imaju različit stepen, što dovodi do toga da se njihov broj značajno smanjuje (na osnovu ranije pokazane veze između stepena korenova i veličine drveta, jasno je da su svi stepeni korenova $O(\log n)$, pa ih ne može biti više od $O(\log n)$).

Konsolidacija se vrši na sledeći način: sve dok neka dva čvora iz liste korenova imaju isti stepen vrši se spajanje korenova istog stepena tj. radi se sledeće:

1. pronalaze se dva čvora x i y u listi korenova koja su istog stepena (bez smanjenja opštosti neka važi da je vrednost ključa čvora x manja ili jednaka od vrednosti ključa čvora y)
2. y se izbacuje iz liste korenova i proglašava se detetom čvora x. Na ovaj način, drvo sa korenom x ostaje hip i stepen čvora x se povećava za jedan. Na osnovu invarijante o stepenima dece znamo da važi da je poslednje dodati naslednik čvora x imao bar stepen $d-2$, gde je d stepen čvora x. To znači da bi naredni tj. poslednji dodati naslednik, što je upravo y, morao da ima stepen bar $d-1$. Međutim, mi znamo da on ima stepen d (jer mu je stepen jednak stepenu čvora x), tako da je invarijanta zadovoljena, uz jedan dodatni stepen slobode. Naime, čak i kada uklonimo jednog naslednika čvora y, invarijanta ostaje zadovoljena. Sa druge strane, uklanjanje dva naslednika bi narušilo invarijantu i to ne smemo da radimo. Ovo je značajno za operaciju smanjivanja vrednosti elemenata hipa `smanji_kljuc` i vratićemo se na ovo prilikom opisa te operacije.

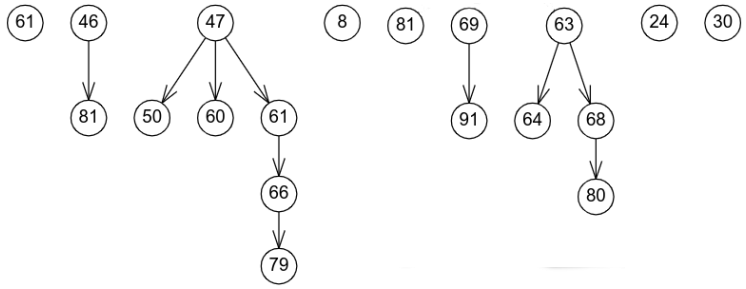
Naglasimo da je izbor para čvorova sa istim stepenima u svakom koraku proizvoljan (konsolidacija se ne mora se vršiti ni u kakvom posebnom redosledu parova čvorova).

Obično se za realizovanje konsolidacije koristi pomoćni niz u koji se smeštaju pokazivači na čvorove iz liste korenova, tako da se na poziciji i čuva pokazivač na neki čvor iz liste korenova koji je stepena i. Dimenzija ovog pomoćnog niza jednaka je maksimalnom stepenu $D(n)$ proizvoljnog čvora u Fibonačijevom hipu – već smo objasnili da je on jednak $O(\log n)$, gde je sa n označen broj elemenata hipa. Korenovi koji su istog stepena se detektuju na sledeći način: prolazi se redom kroz listu korenova i ukoliko je tekući koren stepena i, a i-ti element niza je još uvek prazan, on se inicijalizuje pokazivačem na dati koren, a ukoliko je i-ti element već postavljen, pronađena su dva korena istog stepena i oni se spajaju.

Primer 1.2.3

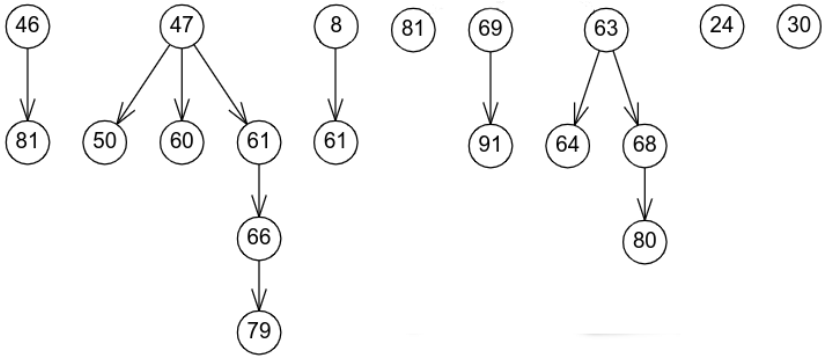
Pogledajmo kako se menja prethodni hip nakon izvođenja operacije brisanja elementa sa minimalnom vrednošću ključa.

U prvom koraku se sva deca uklonjenog minimalnog čvora 4 *umeću u niz korenova.*

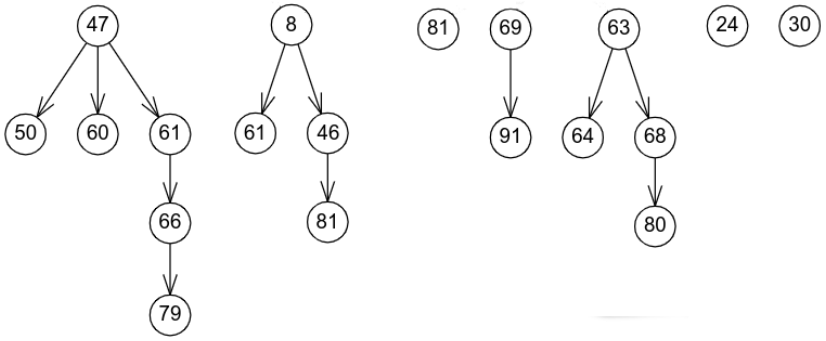


Nakon toga izvršavamo konsolidaciju, tj. spajanje drveta čiji su koreni istog stepena.

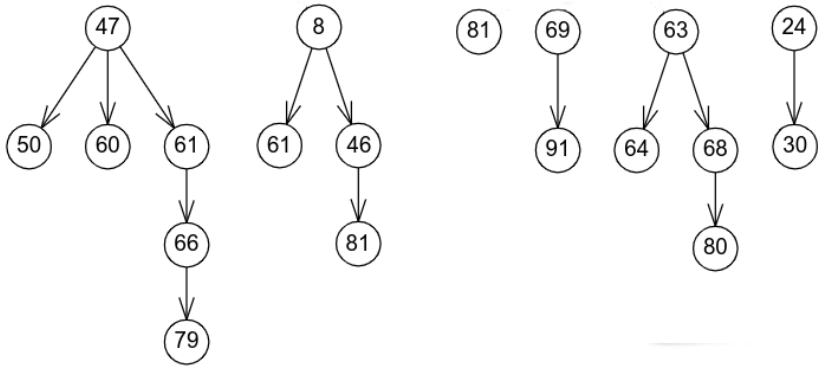
- *Prvo spajamo čvorove* 8 *i* 61*, koji imaju stepen* 0*.*



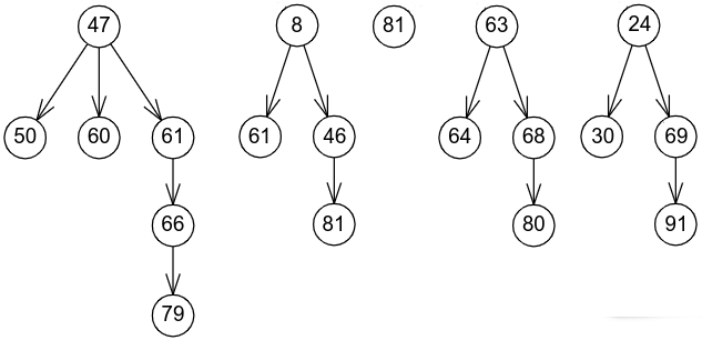
- *Koren dobijenog drveta ima stepen* 1*, ono se spaja sa drvetom čiji je koren* 46*, koji takođe ima stepen* 1*.*



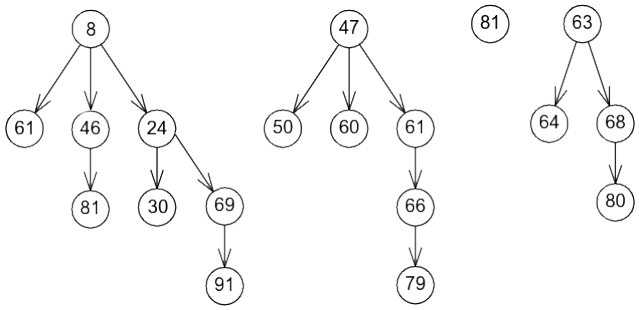
- *Nakon toga spajaju se drveta čiji su koreni* 24 *i* 30 *(oba su stepena* 0*).*



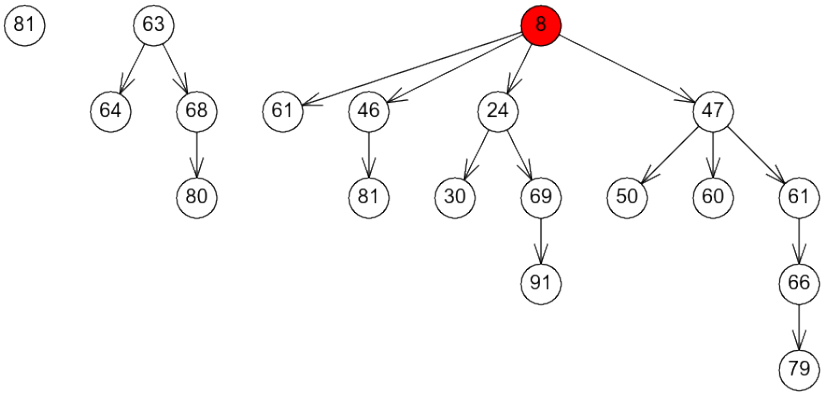
- *Dobijeno drvo se spaja sa drvetom čiji je koren* 69 *(oba korena imaju stepen* 1*).*



- *Dobijeno drvo se spaja sa drvetom čiji je koren* 8 *(oba drveta imaju stepen* 2*).*



- *Na kraju se spaja dobijeno drvo sa korenom* 8 *sa drvetom čiji je koren* 47 *(oba korena imaju stepen* 3*). Nakon tog spajanja, sva drveta imaju različite stepene korenova (stepeni su* 0*,* 2 *i* 4*). Time se konsolidacija završava, garantovano je da je broj drveta* $O(\log n)$ *i zatim se novi minimalni čvor može efikasno odrediti (to je čvor* 8*).*



Analizirajmo složenost prethodno opisanog postupka. Deca se efikasno, algoritmom složenosti $O(1)$ mogu umetnuti u listu korenova (radi se o spajanju dve liste). Nakon toga se vrši konsolidacija, koja u najgorem slučaju može biti i linearne složenosti (najgori slučaj je kada se n jednočlanih drveta spaja). Ipak, može se pokazati da je amortizovana složenost konsolidacije $O(\log n)$ – neformalno, kada se jednom izvrši konsolidacija i dobije se mali broj velikih drveta, svaki naredni korak izbacivanja najmanjeg elementa i konsolidacije biće prilično efikasan. Na kraju se pronalazi novi minimalni čvor, obradom svih korenova. Njih nakon konsolidacije može biti najviše $O(\log n)$, pa je ovo efikasna operacija.

Prikažimo sada jednu jednostavnu implementaciju ovih operacija (jednostavnosti radi, koristimo globalne promenljive i ne vodimo računa o alokaciji i dealokaciji memorije).

```
// Cvor drveta sadrzi podatak i dvostruko povezanu listu dece
struct Cvor {
  int podatak;
```

```
  list<Cvor*> deca;
};

// dvostruko povezana lista korenova
list<Cvor*> hip;

// pokazivac (iterator) koji ukazuje na najmanji cvor
list<Cvor*>::iterator minCvor;

// umetanje novog elementa u hip
void umetni(int podatak) {
  // pravimo novi cvor i upisujemo podatak u njega
  Cvor* novi = new Cvor();
  novi->podatak = podatak;

  if (hip.empty()) {
    // ako je hip prazan ubacujemo novi cvor u njega i taj novi cvor je minimalan
    hip.push_back(novi);
    minCvor = hip.begin();
  } else {
    // ako hip nije prazana ubacujemo novi cvor u njega
    hip.push_front(novi);
    // minimum azuriramo samo ako je novi podatak manji od dotadasnjeg minimalnog
    if (novi->podatak < (*minCvor)->podatak)
      minCvor = hip.begin();
  }
}

// najmanji element hipu
int minimum() {
  return (*minCvor)->podatak;
}

// konsolidacija hipa (pomocna operacija kojom se skracuje lista korenova)
void konsoliduj() {
  // za svaki moguci broj dece pamtimo jedan koren koji ima toliko
  // dece u njemu ce se nalaziti svi koreni novog hipa (i svi koreni
  // ce imati razlicit broj dece)
  const int MAKS_DECE = 64;
  Cvor* pom[MAKS_DECE] = {};
  // obradjujemo jedan po jedan koren hipa
  for (Cvor* x : hip) {
    // dok postoji drugi koren (u novom hipu) sa istim brojem dece kao x
    int brojDece = x->deca.size();
    while (pom[brojDece] != nullptr) {
      // spajamo dva korena tako da je manji podatak iznad
      Cvor* y = pom[brojDece];
      if (x->podatak > y->podatak)
        swap(x, y);
      x->deca.push_back(y);
      // kada mu se pridruzi y, koren x ima jedno dete vise, a u novom hipu vise
```

```
      // ne postoji hip koji ima stari broj dece
      pom[brojDece] = nullptr;
      brojDece++;
    }
    // upisujemo x u novi hip
    pom[brojDece] = x;
  }
  // prebacujemo sve korene iz novog hipa u stari
  hip.clear();
  for (Cvor* cvor : pom)
    if (cvor != nullptr)
      hip.push_back(cvor);
}
```

1.2.2.5 Smanjivanje vrednosti ključa

Još jedna važna operacija nad Fibonačijevim hipom jeste smanjivanje vrednosti ključa čvoru x (operacija `smanji_kljuc(H, x, v)`). Pretpostavljamo da je čvor x kome se vrednost umanjuje već poznat tj. da je poznat pokazivač na njega (jer je to slučaj u mnogim algoritmima koji koriste Fibonačijeve hipove). Ako nije, tj. ako pretpostavljamo da je poznata samo vrednost ključa koja se umanjuje, ali ne i čvor koji sadrži tu vrednost (to je operacija `smanji_vrednost(H, k, v)`) onda se čvor k sa vrednošću ključa k može efikasno pronaći tako što se čuva mapa koja slika ključeve u pokazivače na čvorove (dovoljno je da se vrednost ključa slika u pokazivač na bilo koji čvor koji je sadrži).

Da bi se ova operacija mogla efikasno implementirati, potrebno je čvorove proširiti sa nekoliko dodatnih podataka, koje ćemo u tekstu opisati.

Smanjivanje vrednosti ključa se izvodi na sledeći način: prvo se datom čvoru x promeni vrednost ključa na v, a zatim se, ako x nije koren, ta vrednost poredi sa vrednošću ključa roditelja. Kako bi se moglo pristupiti roditelju, u svim čvorovima se čuva informacija o roditeljskom čvoru (ona ne mora biti definisana jedino u korenima drveta). Ako je x koren ili je vrednost ključa roditelja čvora x manja ili jednaka v, onda uslov hipa nije narušen te nisu potrebne nikakve dalje izmene. Ako to nije slučaj, uslov hipa je narušen te je hip potrebno preurediti. Najpre se vrši *odsecanje* grane između čvora x i njegovog roditelja i poddrvo sa korenom x se dodaje u listu korenova.

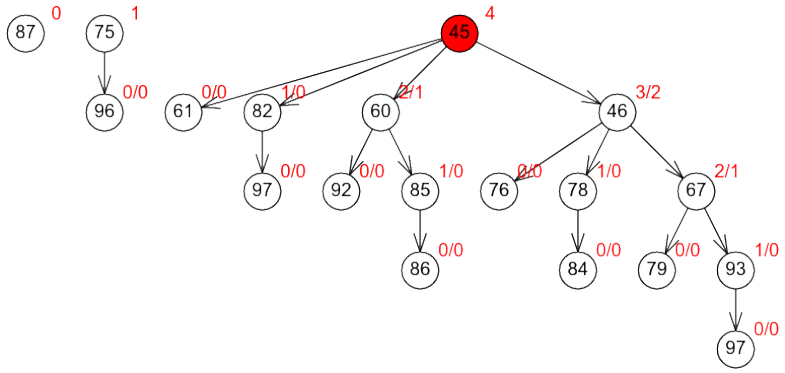
Postavlja se pitanje da li je drvo iz kojeg je isečen čvor x potrebno dalje preuređivati. Odgovor zavisi od toga da li je izbacivanjem poddrveta sa korenom x narušena invarijanta o broju čvorova u tom drvetu, tj. da li je narušena invarijanta o stepenima dece. Podsetimo se da su nakon konsolidacije čvorovi u drvetu obično takvi da imaju jedno dete više od onoga što bi trebalo da imaju na osnovu invarijante tj. da im je moguće ukloniti jedno dete (tj. poddrvo kojem je ono koren), a da invarijanta za njegovog roditelja i dalje bude zadovoljena. Sa druge strane, uklanjanje dva deteta narušava invarijantu. Dakle, postavlja se pitanje da li smo roditelju čvora x već uklonili neko dete od trenutka kada je ubačen u trenutno drvo. Da bismo to znali, svaki čvor pored informacije o svom stepenu (broju svoje dece), sadrži i *oznaku* da li je „izgubio'' dete od poslednjeg trenutka kada je postao dete nekog drugog čvora. Čvorovi se inicijalno ne označavaju i svaki put kada čvor postane dete nekog drugog čvora ili koren, ukoliko je bio označen, oznaka se uklanja. Ako jedan čvor izgubi dva deteta, odsecamo i roditelja i postupak se nastavlja po istom principu, naviše, uz drvo. Preciznije, vrši se sledeći niz koraka:

- smanjuje se vrednost ključa čvora x
- ako x nije koren i ako je nova vrednost v manja od vrednosti roditelja čvora x, odseca se celo drvo čiji je x koren, čvor x se dodaje se u listu korenova (čime to poddrvo postaje samostalno drvo) i uklanja se oznaka (ukoliko je čvor x bio označen)
  - ako je roditelj p čvora x neoznačen (nije mu do sada bilo odsečeno nijedno dete), on se označava

– u protivnom, odseca se celo drvo čiji je koren roditeljski čvor p, čvor p se dodaje u listu korenova (čime i to poddrvo postaje samostalno drvo) i sa njega se uklanja oznaka
– rekurzivno se ponavlja prethodni korak za sve roditelje koji nisu već korenovi i kojima su dva puta odsečena deca.

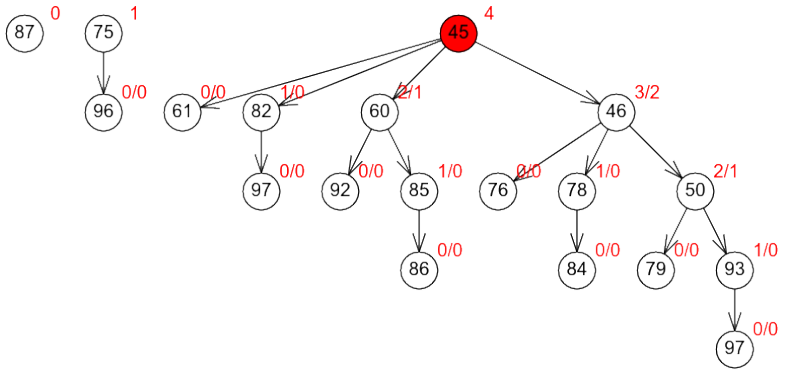
Primer 1.2.4

Razmotrimo hip prikazan na slici 1.9. Gore-desno od svakog čvora napisan je trenutni stepen čvora i minimalni stepen koji čvor mora imati na osnovu invarijante. Na početku skoro svi čvorovi imaju i veći stepen nego što je to potrebno.

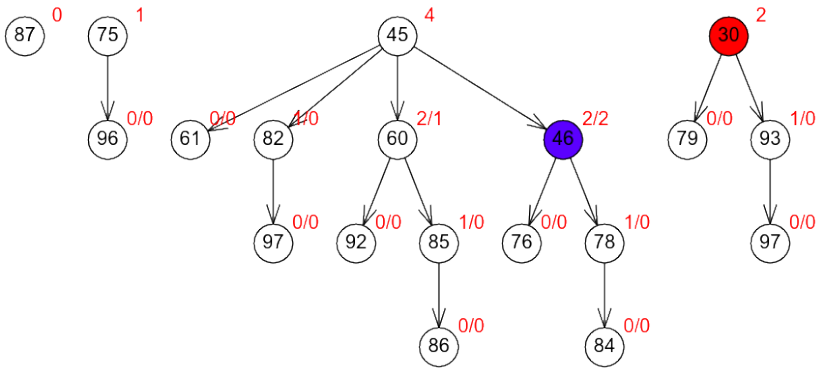


Slika 1.9: Početni hip

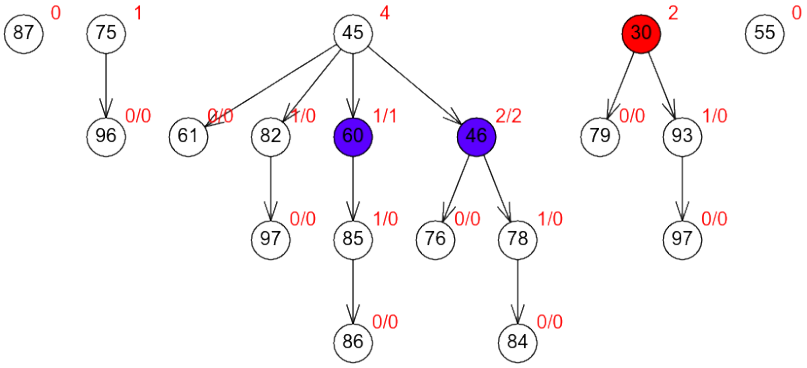
- *Pretpostavimo da želimo da smanjimo vrednost ključa sa* 67 *na* 50. *Pošto je nakon tog smanjivanja vrednost i dalje veća od vrednosti u roditeljskom čvoru, nema potrebe vršiti dalje izmene.*



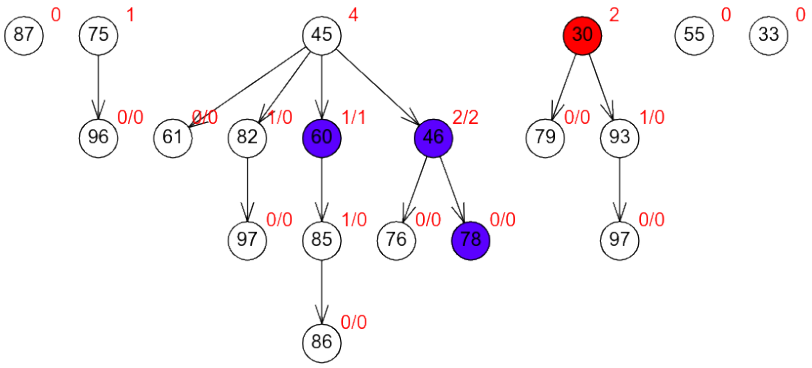
- *Daljim smanjivanjem vrednosti ključa* 50 *na* 30 *narušava se odnos vrednosti u tom i roditeljskom čvoru, pa se čvor odseca i stavlja u niz korenova. Minimalni čvor postaje* 30, *pošto je* 30 *manje od* 45. *Pošto je roditeljski čvor* 46 *neoznačen, nema potrebe dalje popravljati drvo nakon odsecanja, već se samo roditeljski čvor* 46 *označava. Primetimo da je njegov stepen u ovom trenutku jednak minimalnoj vrednosti stepena na osnovu invarijante i njemu nije moguće dalje odsecati decu, a da invarijanta ne bude narušena. Dobija se hip prikazan na sledećoj slici.*



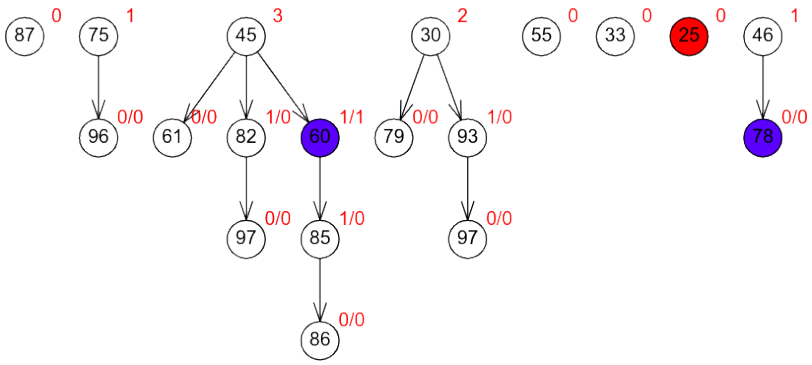
- *Smanjimo sada vrednost ključu* 92 *na* 55. *Čvor se odseca i stavlja u niz korenova. Pošto je kod njemu roditeljskog čvora* 60 *postojao jedan stepen slobode (stepen tog čvora je za* 1 *veći od minimalne vrednosti) tj. pošto* 60 *nije označen čvor, nema potrebe dalje modifikovati drvo.*



- *Smanjimo sada vrednost ključa* 84 *na* 33. *On se odseca i stavlja u niz korenova. Pošto je kod i njemu roditeljskog čvora* 78 *postojao jedan stepen slobode tj. pošto* 78 *nije označen čvor, nema potrebe dalje modifikovati drvo.*



- *Na kraju, smanjimo vrednost ključa* 78 *na* 25. *On se odseca i stavlja u niz korenova. Međutim, ovaj put imamo situaciju da je njegov roditljski čvor* 46 *označen, što znači da je dostigao minimalni stepen, pa se dodatnim odsecanjem njegovog deteta dobija stepen manji od minimalnog. Zato je potrebno odseći i ceo roditeljski čvor* 46. *Njegov roditeljski čvor je koren* 45, *pa nema potrebe za daljim modifikacijama i dobija se hip prikazan na sledećoj slici (korene nema potrebe označavati, jer njihov stepen može biti proizvoljan tj. invarijanta ne zahteva bilo kakvu minimalnu vrednost za stepen korena).*



1.3 Strukture podataka za predstavljanje disjunktnih skupova

Ponekad je u programu potrebno održavati nekoliko disjunktnih skupova (često podskupova nekog skupa), pri čemu je potrebno umeti za dati element efikasno pronaći kom skupu pripada (tu operaciju zovemo `find` tj. `pronadji`) i efikasno spojiti dva zadata skupa u novi, veći skup (tu operaciju zovemo `union` tj. `unija`). Pretpostavićemo da je svaki skup jednoznačno određen nekom oznakom (to može biti redni broj skupa, naziv skupa ili neki kanonski predstavnik skupa). Argumenti operacije `unija` ne moraju biti oznake skupova čiju uniju treba kreirati, već mogu biti proizvoljni elementi tih skupova. Prilikom izvođenja unije polazni skupovi se uklanjaju iz kolekcije, i u kolekciju se dodaje njihova unija.

Problem

Definisati strukture podataka koje omogućavaju sledeći interfejs:

- `pronadji(x)` – *vraća oznaku skupa kom pripada element* `x` *(ta oznaka može biti ili neki brojevni identifikator ili neki kanonski predstavnik tog skupa);*
- `unija(x, y)` – *spaja skup koji sadrži element* `x` *i skup koji sadrži element* `y`.

Pomoću operacije `pronadji` lako možemo za dva elementa proveriti da li pripadaju istom skupu tako što za svaki od njih pronađemo oznaku skupa kom pripada i proverimo da li su ove oznake jednake.

Razmotrimo situaciju u kojoj nekom događaju prisustvuje n osoba. Reći ćemo da se dve osobe poznaju ako se poznaju direktno ili indirektno. Naime, ako se osobe A i B poznaju, i osobe B i C se poznaju, onda zaključujemo da se i osobe A i C poznaju. Želimo da osmislimo strukturu podataka kojom je moguće za proizvoljne dve osobe utvrditi da li se poznaju ili ne, i koja omogućava sklapanje novih poznanstava, tj. kojom se prilikom upoznavanja dve nove osobe, efikasno ažurira relacija poznanstva.

Primetimo da je ovakva struktura podataka korisna kada god radimo sa nekim relacijama ekvivalencije i kada je potrebno predstaviti klase ekvivalencije (koje su disjunktni podskupovi skupa na kom je relacija definisana). Provera da li su dva elementa u relaciji se zasniva na proveri da li pripadaju istoj klasi, a uspostavljanje relacije između bilo koja dva elementa dovodi do spajanja njihovih klasa ekvivalencije. Često se upotrebljava za analizu povezanosti nekih elemenata. Na primer, korisnici društvenih mreža se mogu grupisati u klase ekvivalencije na osnovu svojih poznanstava sa drugim korisnicima i korišćenjem ovakve strukture podataka možemo lako utvrditi da li su svi međusobno povezani (preko zajedničkih poznanika).

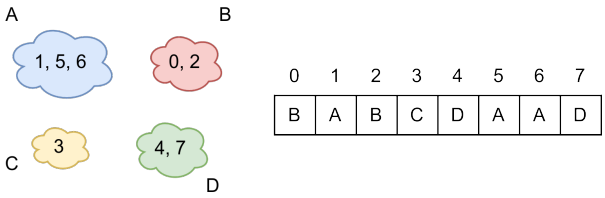
Struktura podataka koja podržava ovakav interfejs se često naziva *union-find* ili *DSU* (engl. disjoint set union), pri čemu se pod tim imenom obično podrazumeva i specifična, efikasna implementacija (pre nje, u nastavku ćemo prikazati i jednostavniju, ali neefikasniju implementaciju).

Strukture podataka za rad sa disjunktnim skupovima imaju različite primene. Jedna od najznačajnijih je održavanje komponenti povezanosti u Kruskalovom algoritmu za konstrukciju minimalnog povezujućeg drveta u grafu. Mogu se koristiti i za segmentaciju slika, tj. particionisanje slike na veći broj disjunktnih

regiona, tako da su svi pikseli istog regiona slični u pogledu nekog svojstva kao što je boja, intenzitet ili tekstura.

1.3.1 Naivna implementacija

Jedna moguća implementacija strukture podataka sa ovakvim interfejsom podrazumeva da se održava preslikavanje svakog elementa u oznaku skupa kojem pripada. Ako pretpostavimo da razmatramo skup od n elemenata i da su svi elementi numerisani brojevima od 0 do $n-1$, onda ovo preslikavanje možemo realizovati pomoću običnog niza gde se na poziciji svakog elementa nalazi oznaka skupa kojem on pripada (ukoliko elementi nisu numerisani brojevima, možemo umesto niza da koristimo mapu kojom se oznake elemenata preslikavaju u oznake skupa kom element pripada). Primer takve reprezentacije prikazan je na slici 1.10.



Slika 1.10: Predstavljanje skupova običnim nizom.

Operacija `pronadji` je tada trivijalna: dovoljno je iz niza pročitati oznaku skupa kom element pripada. Složenost operacije `pronadji` je tada $O(1)$.

Operacija `unija` je mnogo sporija jer podrazumeva da se oznake svih elemenata jednog skupa menjaju u oznake drugog, što zahteva da se prođe kroz ceo niz. Složenost operacije `unija` je tada $\Theta(n)$.

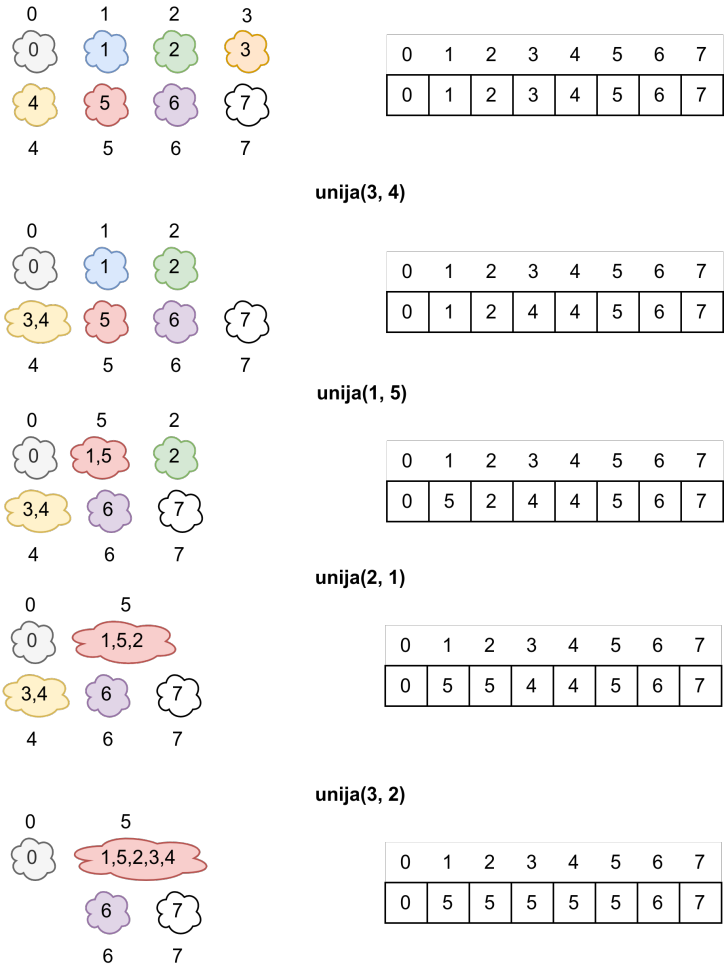
Ukoliko bi se umesto niza koristila neuređena mapa i ako bismo pretpostavili da je složenost operacija sa mapom $O(1)$ (što je zaista amortizovana složenost operacija nad neuređenom mapom), operacija `pronadji` bila bi složenosti $O(1)$ a operacija `unija` složenosti $O(n)$ (jer je ponekad potrebno menjati oznake skupova za sve elemente). Kako bismo odredili amortizovanu složenost (što je relevantno za ocenu složenosti efikasnijih implementacija, koje ćemo kasnije opisati), potrebno je da procenimo potrebno vreme izvršavanja m operacija od kojih je svaka tipa `unija` ili `pronadji`. Najgori slučaj za ukupno vreme izvršavanja ovog niza operacija možemo da ocenimo kao $O(m \cdot n)$, jer iako se operacija `pronadji` izvršava veoma efikasno – u vremenu $O(1)$, složenost operacije `unija` je linearna u odnosu na broj elemenata koji održavamo i najgori slučaj nastupa kada stalno vršimo unije (doduše, maksimalni broj izvršavanja operacije unije različitih skupova je $n-1$, jer će se nakon toga svi elementi objediniti u isti skup, pa je ukupno vreme svih tih operacija reda $O(n^2)$).

Primer 1.3.1

Ilustrujmo sprovođenje operacija `pronadji` i `unija` nad opisanom implementacijom strukture podataka za disjunktne skupove na jednom primeru. Pretpostavimo da je početno stanje takvo da svaki element pripada zasebnom skupu. Razmotrimo na koji način se menja sadržaj odgovarajućeg niza nakon izvršavanja operacija unije. Izvršavanje nekoliko operacija unije prikazano je na slici 1.11. Levo su prikazani skupovi, a desno niz kojim su ti skupovi predstavljeni (iznad svakog niza prikazani su indeksi). Pretpostavljamo da će prilikom izvršavanja operacije `unija(x,y)` oznaka novog skupa biti oznaka skupa kojem pripada element `y`.

Prikažimo sada implementaciju ove tehnike (jednostavnosti radi koristimo globalne promenljive).

```
int id[MAX_N];
int n;
```



Slika 1.11: Primer primena operacije unije.

```
// na pocetku svaki element pripada zasebnom skupu
void inicijalizuj() {
  for (int i = 0; i < n; i++)
    id[i] = i;
}

// oznaku podskupa kome pripada element x citamo sa pozicije x iz niza
int pronadji(int x) {
   return id[x];
}

// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int idx = id[x], idy = id[y];
  // oznake svih elemenata prvog podskupa menjamo u oznaku drugog podskupa
  for (int i = 0; i < n; i++)
    if (id[i] == idx)
      id[i] = idy;
}

// elementi su u istom podskupu ako su im oznake iste
int u_istom_podskupu(int x, int y) {
  return pronadji(x) == pronadji(y);
}
```

Obratimo pažnju da se prilikom izvođenja unije moraju koristiti pomoćne promenljive (razmislite zašto je naredna implementacija neispravna).

```
// pravimo uniju skupova kome pripadaju dati elementi
void unija(int x, int y) {
  // oznake svih elemenata prvog skupa menjamo u oznaku drugog skupa
  for (int i = 0; i < n; i++)
    if (id[i] == id[x])
      id[i] = id[y];
}
```

1.3.2 Efikasna implementacija

Razmotrimo nešto drugačiju implementaciju ove strukture podataka u kojoj je operacija `unija` vremenski efikasnija.

1.3.2.1 Struktura i operacije

Ključna ideja na kojoj se zasniva efikasnije rešenje je da elemente ne preslikavamo u oznake skupova, već da skupove čuvamo u obliku drveta (ne nužno binarnih) tako da svaki element slikamo u njegovog roditelja u drvetu. Svaki koren drveta slikamo samog u sebe i smatramo ga predstavnikom skupa predstavljenog tim drvetom (dakle, predstavnik svakog skupa je koren njegovog drveta). Naglasimo da su u čvorovima ovih drveta pokazivači usmereni od dece ka roditeljima, za razliku od klasičnih drveta gde pokazivači u čvorovima ukazuju od roditelja ka deci.

Da bismo za proizvoljni element saznali oznaku skupa kom pripada tj. da bismo implementirali operaciju `pronadji`, potrebno je da počev od tog elementa prođemo kroz niz roditeljskih čvorova sve dok ne stignemo do korena. Uniju dva skupa (tj. operaciju `unija`) u ovom pristupu možemo jednostavno realizovati tako što

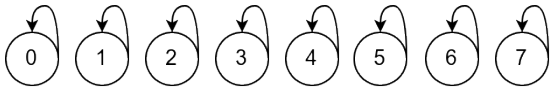
koren jednog drveta usmerimo ka korenu drugog.

Naivna implementacija koja je opisana u prethodnom poglavlju odgovara situaciji u kojoj osoba koja promeni adresu obaveštava sve druge osobe o svojoj novoj adresi, dok implementacija koju trenutno opisujemo odgovara scenariju u kome samo na staroj adresi ostavlja informaciju o svojoj novoj adresi. Ovo, naravno, malo usporava dostavu pošte, jer se mora preći kroz niz preusmeravanja, ali ako taj niz nije predugačak, može biti značajno efikasnije od prvog pristupa.

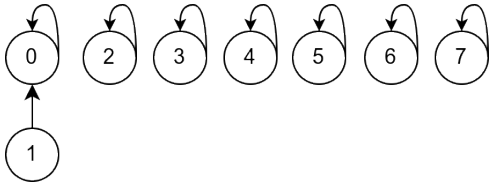
Primer 1.3.2

Prikažimo rad algoritma na jednom primeru. Skupove ćemo predstavljati drvetima.

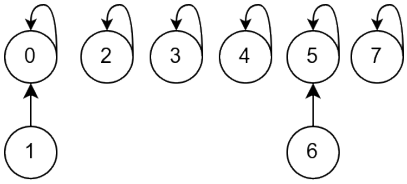
Pretpostavljamo da su na početku svi skupovi jednočlani.



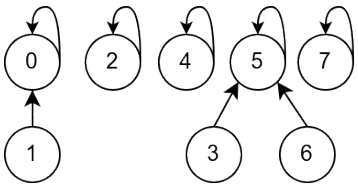
- *Izvršava se operacija* `unija(1, 0)`. *Predstavnik skupa kome pripada* 1 *je* 1 *i čvor koji sadrži* 1 *preusmeravamo tako da ukazuje na čvor predstavnika skupa u kome je* 0, *a to je* 0.



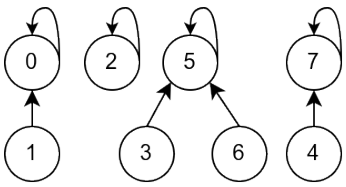
- *Izvršava se operacija* `unija(6, 5)`. *Predstavnik skupa kome pripada* 6 *je* 6 *i čvor koji sadrži* 6 *preusmeravamo tako da ukazuje na čvor koji sadrži predstavnika skupa u kome je* 5, *a to je* 5.



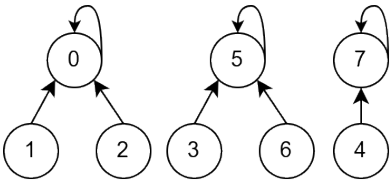
- *Izvršava se operacija* `unija(3, 6)`. *Predstavnik skupa kome pripada* 3 *je* 3 *i čvor koji sadrži* 3 *preusmeravamo tako da ukazuje na čvor koji sadrži predstavnika skupa u kome je* 6, *a to je* 5.



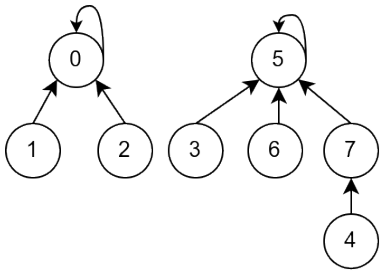
- *Izvršava se operacija* `unija(4, 7)`. *Predstavnik skupa kome pripada* 4 *je* 4 *i čvor koji sadrži* 4 *preusmeravamo tako da ukazuje na čvor koji sadrži predstavnika skupa u kome je* 7, *a to je* 7.



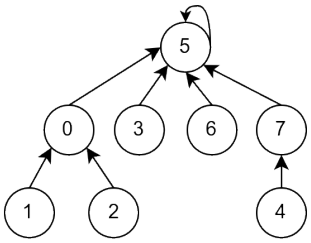
- *Izvršava se operacija* `unija(2, 0)`. *Predstavnik skupa kome pripada* 2 *je* 2 *i čvor koji sadrži* 2 *preusmeravamo tako da ukazuje na čvor koji sadrži predstavnika skupa u kome je* 0, *a to je* 0.



- *Izvršava se operacija* `unija(4, 3)`. *Predstavnik skupa kome pripada* 4 *je* 7 *i čvor koji sadrži* 7 *preusmeravamo tako da ukazuje na čvor koji sadrži predstavnika skupa u kome je* 3, *a to je* 5.



- *Izvršava se operacija* `unija(2, 6)`. *Predstavnik skupa kome pripada* 2 *je* 0 *i čvor koji sadrži* 0 *preusmeravamo tako da ukazuje na čvor koji sadrži predstavnika skupa u kome je* 6, *a to je* 5.



Iako ovako opisana struktura podataka ima drvoliku strukturu, možemo je implementirati korišćenjem statičkog niza. Naime, pošto svaki element u drvetu ima jedinstvenog roditelja, na poziciji nekog elementa u nizu možemo čuvati indeks njegovog roditelja. U slučaju da je element koren nekog drveta, njegov roditelj je on sâm.

Primer 1.3.3

Za drvo iz prethodnog primera, niz roditelja ima sledeći sadržaj:

0	1	2	3	4	5	6	7
5	0	0	5	7	5	5	5

Imajući u vidu ovakvu reprezentaciju, kôd je prilično jednostavno napisati (jednostavnosti radi pretpostavljamo da se podaci smeštaju u globalnim promenljivim).

```
int roditelj[MAX_N];
int n;

// na pocetku svaki element pripada zasebnom skupu
void inicijalizuj() {
  for (int i = 0; i < n; i++)
    roditelj[i] = i;
}

// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
```

```
int pronadji(int x) {
  // sve dok ne stignemo do korena
  while (roditelj[x] != x)
    // penjemo se u roditeljski cvor
    x = roditelj[x];
  return x;
}

// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;

  // postavljamo da je koren prvog podskupa sin korena drugog podskupa
  roditelj[fx] = fy;
}
```

1.3.2.2 Uravnotežavanje drveta

Složenost prethodnog pristupa zavisi od toga koliko su drveta kojima se predstavljaju skupovi uravnotežena[2]. U najgorem slučaju se drveta mogu izdegenerisati u liste i tada je složenost najgoreg slučaja svake od operacija `unija` i `pronadji` $O(n)$.

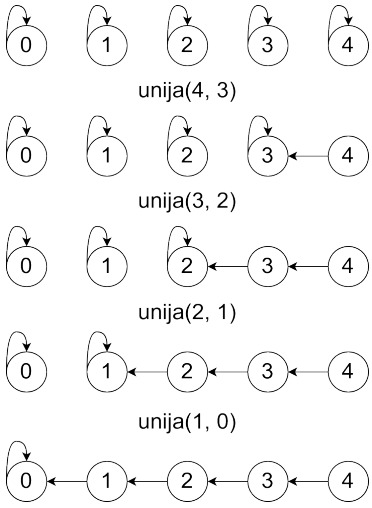
Primer 1.3.4

Ako uvek prilikom izvršavanja operacije `unija(x,y)` usmeravamo pokazivač od predstavnika skupa kojem pripada element x ka predstavniku skupa kojem pripada element y, izvršiće se niz izmena drveta koji je prikazan na slici 1.12. Upit kojim se traži predstavnik skupa kojem pripada element `4` se realizuje nizom koraka kojima se prelazi preko elemenata `3, 2, 1, 0` i u slučaju većeg broja elemenata se dobija veoma neefikasna implementacija pronalaženja predstavnika (a samim tim i unije, čija implementacija podrazumeva pronalaženje predstavnika).

Trenutni algoritam, dakle, nije u najgorem slučaju (ilustrovanom na slici 1.12) efikasniji od naivnog pristupa. Naime, u naivnom pristupu se pronalaženje predstavnika vrši u vremenu $O(1)$, ali unija uvek zahteva vreme $O(n)$, dok ovde i operacija `pronadji` može imati složenost $O(n)$. Ipak, složenost trenutnog algoritma se može popraviti ako se sve vreme drveta održavaju uravnoteženim. Kada su drveta uravnotežena, možemo dokazati da je složenost najgoreg slučaja svake od operacija `unija` i `pronadji` jednaka $O(\log n)$. Na ovaj način se postiže da je vreme izvršavanja niza od m operacija tipa `unija` ili `pronadji` u najgorem slučaju jednako $O(m \log n)$, dok je u naivnom pristupu vreme izvršavanja ovog niza operacija, u slučaju kada je broj operacija tipa `unija` $O(m)$, jednako $O(mn)$.

Prilikom pravljenja unije imamo slobodu izbora korena kog ćemo usmeriti prema drugom korenu i uravnoteženost se postiže time što se ova sloboda na neki način iskoristi. Postoje dva uobičajena načina vršenja unije: *unija na osnovu ranga (visine)* i *unija na osnovu broja elemenata (veličine)*.

[2] Pojam uravnoteženog drveta može se precizno definisati na razne načine. U ovom tekstu ćemo uravnoteženost razmatrati samo neformalno: drvo smatramo uravnoteženijim što je rastojanje listova od korena ujednačenije.



Slika 1.12: Ilustracija izvršavanja niza operacija `unija(x,y)` kada se pokazivač od predstavnika skupa kome pripada element x usmerava ka predstavniku skupa kome pripada element y.

Unija na osnovu ranga (visine)

Osnovna ideja vršenja *unije na osnovu ranga (visine)* je da se prilikom izmena (a one se vrše samo u sklopu operacije unije), ako je moguće, obezbedi da se visina[3] drveta kojim je predstavljena unija ne poveća u odnosu na visine pojedinačnih drveta koja predstavljaju skupove čija se unija pravi. Pretpostavićemo da svakom čvoru drveta pridružujemo broj koji predstavlja visinu tog čvora tj. drveta sa korenom u tom čvoru. Visinu drveta možemo održavati u posebnom nizu (njega ćemo u kodu, nazvati `rang`[4]). Ako se uvek izabere da koren drveta manje visine usmeravamo ka korenu drveta veće ili jednake visine, tada se visina unije povećava samo ako su oba drveta koja uniramo iste visine.

```
int roditelj[MAX_N];
int n;
int rang[MAX_N];

// na pocetku svaki element pripada zasebnom skupu
// i visina svakog drveta je 0
void inicijalizuj() {
  for (int i = 0; i < n; i++) {
    roditelj[i] = i;
    rang[i] = 0;
  }
}

// pravimo uniju podskupova kojima pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;
```

[3] Visinu čvora možemo definisati kao broj grana na putanji od tog čvora do njemu najudaljenijeg lista. Visinu drveta računamo kao visinu njegovog korena.

[4] U poglavlju 1.3.2.3 posvećenom sažimanju puteva, videćemo da kada se vrše optimizacije ovaj niz ne čuva više visinu svakog drveta već rang (engl. rank) koji predstavlja gornju granicu visine tj. broj od kog ta visini sigurno nije veća.

```
  // usmeravamo koren drveta nizeg ka korenu viseg ranga
  if (rang[fx] < rang[fy])
    swap(fx, fy);
  roditelj[fy] = fx;

  // ako su podskupovi istog ranga
  // unija ce biti za jedan veceg ranga
  if (rang[fx] == rang[fy])
    rang[fx]++;
}
```

Primetimo da su nam za izvođenje operacije unije relevantne samo vrednosti ranga predstavnika skupova.

Dokažimo lemu koja garantuje složenost.

Lema 1.3.1 **[Uniranje na osnovu ranga daje uravnotežena drveta]**

U slučaju vršenja unije na osnovu ranga, u svakom drvetu čiji je rang h nalazi se bar 2^h čvorova.

Dokaz. Tvrđenje dokazujemo matematičkom indukcijom.

- Baza indukcije odgovara polaznom stanju u kome je svaki čvor svoj predstavnik. Rangovi svih drveta su tada nula i sva drveta imaju $2^0 = 1$ čvor, pa tvrđenje važi.
- Pokažimo da operacija unije održava ovu invarijantu. Po induktivnoj hipotezi znamo da ako dva drveta koja predstavljaju skupove koji se uniraju imaju rangove r_1 i r_2, onda je broj čvorova u njima redom bar 2^{r_1} i 2^{r_2} čvorova. Ukoliko se uniranjem rang ne poveća, invarijanta je trivijalno očuvana jer se broj čvorova uvećao, a rang je ostao isti. Jedini slučaj kada se uniranjem povećava rang je kada je $r_1 = r_2$ i tada unirano drvo ima rang $r = r_1 + 1 = r_2 + 1$ i bar $2^{r_1} + 2^{r_2} = 2^{r_1} + 2^{r_1} = 2 \cdot 2^{r_1} = 2^{r_1+1} = 2^r$ čvorova.

Time je tvrđenje dokazano. □

Dakle, rang (visina) svakog drveta koje ima n čvorova je $O(\log n)$, pa je složenost operacije pronalaženja predstavnika u skupu od n čvorova reda $O(\log n)$. Pošto uniranje nakon pronalaženja predstavnika vrši još samo $O(1)$ operacija, i složenost uniranja dva skupa je $O(\log n)$.

Dakle, održavanje visina skupova pod kontrolom nam garantuje logaritamsku složenost osnovnih operacija i vreme izvršavanja niza od m operacija od kojih je svaka tipa `unija` ili `pronadji` je u najgorem slučaju jednako $O(m \log n)$.

Unija na osnovu broja elemenata (veličine)

U pristupu *Unija na osnovu broja elemenata (veličine)*, umesto visine se u svakom od skupova održava broj čvorova tog skupa (tj. u svakom čvoru drveta čuva se informacija o broju čvorova drveta kojem je taj čvor koren).

```
// pravimo uniju podskupova kome pripadaju dati elementi
void unija(int x, int y) {
  int fx = pronadji(x), fy = pronadji(y);

  // x i y imaju istog predstavnika, pa su vec u istom podskupu
  if (fx == fy)
    return;
```

```
  // usmeravamo koren manjeg drveta kao korenu veceg drveta
  if (velicina[fx] < velicina[fy])
    swap(fx, fy);
  roditelj[fy] = fx;

  // azuriramo velicinu novog korena
  velicina[fx] += velicina[fy];
}
```

Ako uvek usmeravamo predstavnika skupa sa manjim brojem elemenata ka predstavniku skupa sa većim brojem elemenata, ponovo dobijamo logaritamsku složenost najgoreg slučaja za obe osnovne operacije. Ovo važi zato što i ovaj način pravljenja unije garantuje da ne možemo imati visoko drvo sa malim brojem čvorova. Naime, da bi se dobilo drvo visine 1, potrebna su bar dva drveta visine 0, odnosno bar 2 čvora; da bi se dobilo drvo visine 2 potrebna su bar dva drveta visine 1 koja imaju bar po 2 čvora, odnosno drvo visine 2 ima bar 4 čvora. Analogno prethodnoj, moguće je dokazati i narednu lemu.

Lema 1.3.2 **[Uniranje na osnovu broja elemenata daje uravnotežena drveta]**

U slučaju vršenja unije na osnovu broja elemenata, u svakom drvetu čija je visina h nalazi se bar 2^h čvorova tj. za svaki koren drveta r visine h sa s čvorova važi $s \geq 2^h$.

Dokaz. Dokažimo lemu indukcijom.

- U početku je broj čvorova u svakom drvetu $s = 1$, a visina svakog drveta je $h = 0$, pa važi $s = 2^h$.
- Pretpostavimo da tvrđenje važi pre spajanja dva drveta čiji su koreni r_1 i r_2, koji imaju redom s_1 i s_2 elemenata, visine h_1 i h_2. Na osnovu induktivne hipoteze znamo da je $s_1 \geq 2^{h_1}$ i $s_2 \geq 2^{h_2}$. Pretpostavimo da se drvo r_2 pridružuje drvetu r_1 (što znači da drvo sa korenom r_2 ima manje ili jednako čvorova nego ono sa korenom r_1 tj. da je $s_2 \leq s_1$), čime se dobija drvo sa s čvorova visine h.
  - Ako drvo sa korenom r_1 ima veću visinu od onog sa korenom r_2, tj ako je $h_1 > h_2$, nakon spajanja se visina drveta sa korenom r_1 ne manja, a broj čvorova mu se povećava, pa tvrđenje leme trivijalno nastavlja da važi. Zaista, važi $s = s_1 + s_2 \geq s_1 \geq 2^{h_1} = 2^h$.
  - U suprotnom se visina drveta sa korenom drveta r_1 povećava i postaje za jedan veća od visine drveta sa korenom r_2 tj. važi $h = h_2 + 1$. Tada važi $s = s_1 + s_2 \geq 2 \cdot s_2 \geq 2 \cdot 2^{h_2} = 2^{h_2+1} = 2^h$.

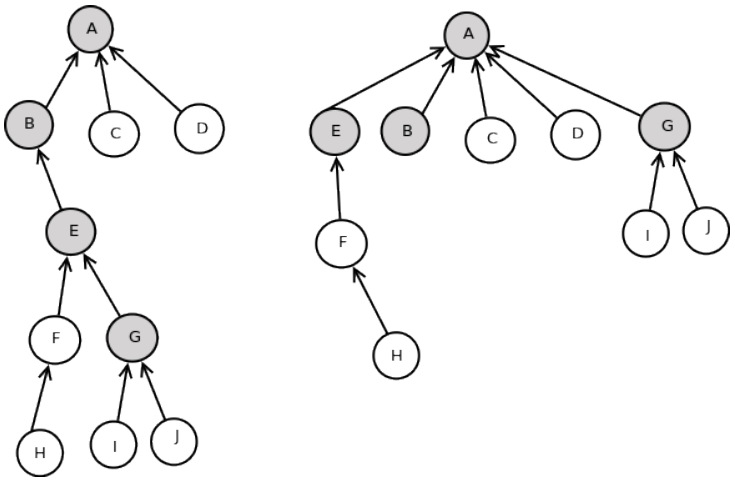
□

Odavde sledi da su visine svih drveta u ovoj strukturi podataka reda $O(\log n)$.

1.3.2.3 Sažimanje puteva

Iako je složenost prethodno opisanih varijanti algoritma sasvim prihvatljiva (složenost izvršavanja m operacija unije je $O(m \log n)$), može se dodatno poboljšati veoma jednostavnom tehnikom poznatom kao *sažimanje puteva* ili *kompresija puteva* (engl. path compression). Naime, prilikom pronalaženja predstavnika možemo sve čvorove kroz koje prolazimo usmeriti ka korenu. Primetimo da tada pored operacije `unija` i operacija `pronadji` menja strukturu drveta kojim je predstavljen taj skup. Jedan način da se to uradi je da se nakon pronalaženja korena, ponovo prođe kroz niz elemenata i svi pokazivači usmere ka korenu (slika 1.13). Na ovaj način se postiže da buduće operacije nad tim skupom budu efikasnije.

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
```



Slika 1.13: Ilustracija postupka sažimanja puteva u dva prolaza nakon traženja predstavnika skupa kome pripada element G.

```
  int koren = x;
  // nalazimo oznaku podskupa kao koreni element podskupa
  while (koren != roditelj[koren])
    koren = roditelj[koren];
  // svim cvorovima na putanji od x do korena
  // postavljamo da je roditeljski cvor koren tog podskupa
  while (x != koren) {
    int tmp = roditelj[x];
    roditelj[x] = koren;
    x = tmp;
  }
  return koren;
}
```

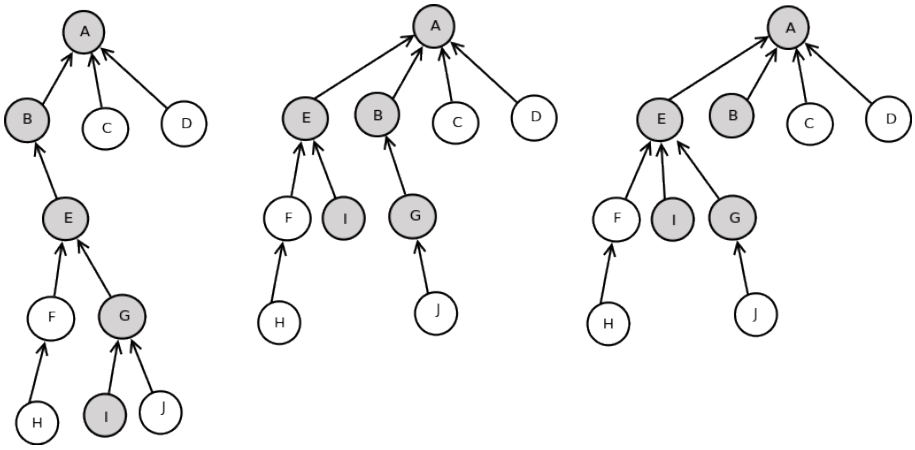
Za sve čvorove koji se obilaze od polaznog čvora do korena, dužine putanja do korena se nakon ovoga smanjuju na 1; slično, skraćuju se i putanje do korena svih čvorova u poddrvetu sa korenom u polaznom čvoru. Ako se vrši uniranje na osnovu broja elemenata, prilikom sažimanja puteva ne menjaju se brojevi elemenata drveta, pa podaci pridruženi čvorovima ostaju korektni i nakon sažimanja puteva. Ako pak vršimo uniju na osnovu ranga, i tumačimo rangove kao visine drveta, jasno je da prilikom sažimanja puteva niz rangova postaje neažuran (jer se neke visine smanjuju, a niz se ne ažurira). Interesantno je da ni u ovom slučaju nema potrebe da se niz ažurira. Naime, rangovi tj. brojevi koji se čuvaju u tom nizu ne predstavljaju više visine čvorova (zato niz nismo ni nazvali `visine`), već gornje granice visina čvorova tj. visine čvorova su manje ili jednake od tih vrednosti. Ovi brojevi nam pomažu da preusmerimo čvorove prilikom uniranja. Lako se pokazuje da se ovim ne narušava složenost najgoreg slučaja i da funkcija nastavlja korektno da radi. Naime, i dalje važi osnovna invarijanta garantovana lemom 1.3.1 da se u svakom drvetu koje ima rang h nalazi bar 2^h čvorova, a pošto je visina manja ili jednaka od ranga, važi da je visina svakog drveta koje sadrži n čvorova najviše $\log n$.

U prethodnoj implementaciji funkcije `pronadji` se dva puta prolazi kroz putanju od čvora x do korena. Slične performanse se mogu dobiti i u samo jednom prolazu. Postoje dva načina na koji se ovo može uraditi: jedan od njih je da se svaki čvor kroz koji se prolazi tokom pronalaženja predstavnika (osim deteta korena) usmeri ka roditelju svog roditelja. Za sve čvorove koji se obilaze (na putu od polaznog čvora do korena), dužine putanja do korena se nakon ovoga smanjuju dvostruko (slika 1.14, sredina), što je dovoljno za odlične performanse.

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
  int tmp;
  // za sve cvorove na putanji od x do korena
  while (x != roditelj[x]) {
    tmp = roditelj[x];
    // novi roditelj od x je roditelj njegovog roditelja
    roditelj[x] = roditelj[roditelj[x]];
    x = tmp;
  }
  return x;
}
```

Drugi način podrazumeva da se, prilikom prolaska od čvora ka korenu, svaki drugi čvor na putanji usmeri ka roditelju svog roditelja (slika 1.14, desno).

```
// naziv podskupa kome element pripada dobijamo kao oznaku korena tog podskupa
int pronadji(int x) {
  // penjemo se do korena tako sto preskacemo po jedan cvor
  while (x != roditelj[x]) {
    // novi roditelj od x je roditelj njegovog roditelja
    roditelj[x] = roditelj[roditelj[x]];
    x = roditelj[x];
  }
  return x;
}
```



Slika 1.14: Ilustracija dva različita načina sažimanja puteva u jednom prolazu tokom traženja predstavnika skupa kojem pripada element I (levo je polazno drvo, u sredini drvo koje se dobija prvim algoritmom, a desno drvo koje se dobija drugim algoritmom).

Primetimo da je na ovaj način dodata samo jedna linija koda u prvobitnu implementaciju operacije `pronadji`.

Kada se primeni bilo koji od tri navedena oblika sažimanja puteva, amortizovana složenost operacija postaje samo $O(\alpha(n))$ (što ovde nećemo dokazivati), gde je $\alpha(n)$ takozvana inverzna Akermanova funkcija koja jako sporo raste[5]. Za bilo koji broj n koji je manji od broja atoma u celom univerzumu (oko 10^{80}) važi da

[5] Akermanova funkcija je poznata po tome što jako brzo raste. Ona je definisana rekurentnim relacijama: $A(0,n) = n+1$, $A(m+1,0) = A(m,1)$, $A(m+1,n+1) = A(m, A(m+1,n))$. Funkcija $\alpha(n)$ je inverzna funkcija funkcije $A(n,n)$ i ona jako sporo raste.

je $\alpha(n) < 5$, tako da je amortizovana vremenska složenost operacija praktično konstantna.

Zadatak: Prvi put kroz matricu

Logička matrica dimenzije $n \times n$ u početku sadrži sve nule. Nakon toga se nasumično dodaje jedna po jedna jedinica. Kretanje po matrici je moguće samo po jedinicama i to samo na dole, na gore, na desno i na levo. Napisati program koji učitava dimenziju matrice, a zatim poziciju jedne po jedne jedinice i određuje nakon koliko njih je prvi put moguće sići od vrha do dna matrice (sa proizvoljnog polja prve vrste koje sadrži jedinicu do proizvoljnog polja poslednje vrste matrice koje sadrži jedinicu).

Opis ulaza

Sa standardnog ulaza se učitava dimenzija matrice $1 \leq n \leq 200$, zatim broj polja m ($1 \leq m \leq n^2$) u koje se upisuje jedinica, a zatim u narednih m redova koordinate tih polja (broj vrste i broj kolone od 0 do $n - 1$, razdvojeni razmakom).

Opis izlaza

Na standardni izlaz ispisati najmanji broj dodatih jedinica nakon kojih je postalo moguće stići od vrha do dna.

Primer

Ulaz

```
4
9
0 0
0 1
1 1
3 3
1 3
2 0
3 0
2 1
2 2
```

Izlaz

```
8
```

Objašnjenje

Posle 8 učitanih polja, matrica postaje

```
1100
0101
1100
1001
```

i vrh i dno postaju spojeni.

Rešenje

Osnovna ideja je da se formiraju svi podskupovi polja matrice između kojih postoji put (oni formiraju klase ekvivalencije tzv. komponente povezanosti). Svaki put kada se uspostavi veza između neka dva polja matrice, podskupovi kojima ona pripadaju se spajaju. Provera da li postoji put između dva polja matrice svodi se onda na proveru da li ona pripadaju istom podskupu.

Putanja od vrha do dna postoji ako i samo ako postoji putanja od bilo kog polja u prvoj vrsti matrice do bilo kog polja u poslednjoj vrsti matrice. To bi dovelo do toga da u svakom koraku moramo da proveravamo sve parove elemenata iz prve i poslednje vrste. Međutim, možemo i bolje. Dodaćemo veštački početni čvor (nazovimo ga izvor) i spojićemo ga sa svim čvorovima u prvoj vrsti matrice i završni čvor (nazovimo ga ušće) koji ćemo spojiti sa svim čvorovima u poslednjoj vrsti matrice. Tada se u svakom koraku može proveriti samo da li su izvor i ušće spojeni tj. da li pripadaju istom podskupu.

Slika 1.15: Komponente povezanosti su obojene različitim bojama. Pošto su izvor i ušće isto obojeni, postoji put od vrha do dna.

Podskupove možemo čuvati pomoću strukture podataka za predstavljanje disjunktnih podskupova.

```
// jednostavnosti radi matricu čuvamo u nizu dimenzije n*n
// redni broj elementa (x, y) u matrici
int kod(int x, int y, int n) {
  return x*n + y;
}

int main() {
  // dimenzija matrice
  int n;
  cin >> n;

  // alociramo matricu n*n tj. niz u kom čuvamo njene elemente
  vector<vector<bool>> a(n);
  for (int i = 0; i < n; i++)
    a[i].resize(n, false);

  // dva dodatna veštačka čvora
  const int izvor = n*n;
  const int usce = n*n+1;

  // inicijalizujemo union-find strukturu za sve elemente matrice
  // (njih n*n), izvor i ušće
  UF_inicijalizacija(n*n + 2);

  // spajamo izvor sa svim elementima u prvoj vrsti matrice
  for (int i = 0; i < n; i++)
```

```
    UF_unija(izvor, kod(0, i, n));

  // spajamo sve elemente u poslednjoj vrsti matrice sa ušćem
  for (int i = 0; i < n; i++)
    UF_unija(kod(n-1, i, n), usce);

  // broj jedinica
  int m;
  cin >> m;

  // korak u kom se spajaju izvor i usce
  int korak = -1;

  // ucitavamo i obrađujemo jednu po jednu jedinicu
  for (int k = 1; k <= m; k++) {
    int x, y;
    cin >> x >> y;
    // ako je u matrici već jedinica, nema šta da se radi
    if (a[x][y]) continue;
    // upisujemo jedinicu u matricu
    a[x][y] = true;
    // povezujemo podskupove u sva četiri smera
    if (x > 0 && a[x-1][y])
      UF_unija(kod(x, y, n), kod(x-1, y, n));
    if (x + 1 < n && a[x+1][y])
      UF_unija(kod(x, y, n), kod(x+1, y, n));
    if (y > 0 && a[x][y-1])
      UF_unija(kod(x, y, n), kod(x, y-1, n));
    if (y + 1 < n && a[x][y+1])
      UF_unija(kod(x, y, n), kod(x, y+1, n));
    // proveravamo da li su izvor i ušće spojeni
    if (UF_pronadji(izvor) == UF_pronadji(usce)) {
      korak = k;
      break;
    }
  }

  cout << korak << endl;

  return 0;
}
```

1.4 Upiti raspona

Neke strukture podataka su posebno pogodne za probleme u kojima se traži da se nad elementima niza izračunavaju statistike (npr. zbir, minimum, maksimum) nekih segmenata tj. raspona uzastopnih elemenata niza. Zahteve tog tipa nazivamo *upiti raspona* (engl. range queries). *Statički upiti raspona* (engl. static range queries), opisani u poglavlju Statički upiti raspona, podrazumevaju da se niz jednom inicijalizuje i nakon toga ne menja, dok *dinamički upiti raspona* (engl. dynamic range queries), opisani u poglavlju Dinamički upiti raspona, omogućavaju da se niz menja između izračunavanja statistika. Ove strukture podataka obično

podržavaju neke od sledećih vrsta upita:

- određivanje elementa niza na datoj poziciji (engl. *point query*),
- određivanje statistike elemenata u datom segmentu niza (engl. *range query*),
- ažuriranje elementa niza na datoj poziciji (engl. *point update*),
- ažuriranje elemenata niza u datom segmentu (engl. *range update*), obično ili tako što se svi elementi postave na neku datu vrednost ili tako što se svi elementi uvećaju za datu vrednost.

Primetimo da mogućnost efikasnog određivanja zbira elemenata proizvoljnog segmenta niza garantuje ujedno i mogućnost određivanja elementa na datoj poziciji (jer se vrednost tog elementa može odrediti kao zbir jednočlanog segmenta koji sadrži samo taj element). Slično, mogućnost efikasnog ažuriranja proizvoljnog segmenta garantuje mogućnost efikasnog ažuriranja pojedinačnih elemenata niza.

- **Prefiksni zbirovi** niza omogućavaju efikasno računanje statistika segmenata i vrednosti pojedinačnog elementa (*range query*, *point query*).
- **Razlike susednih elemenata niza** omogućavaju efikasno ažuriranje segmenata (*range update*).
- **Segmentna drveta** i **Fenvikova drveta** omogućavaju efikasno izvršavanje statistika segmenata i određivanje vrednosti elemenata (*range query*, *point query*), kao i ažuriranje vrednosti pojedinačnog elementa (point update), pri čemu se Fenvikova drveta koriste obično samo za računanje zbira, dok se segmentna drveta koriste za širi spektar statistika.
- **Lenja segmentna drveta** omogućavaju efikasno izvršavanje sve četiri navedene vrste upita.

Iako ćemo sve strukture podataka prikazati u jednodimenzionom obliku, u realnim primenama se veoma često razmatraju dvodimenzionalna, pa i trodimenzionalna uopštenja. Na primer, u dvodimenzionalnom slučaju moguće je efikasno očitavati zbirove proizvoljnih pravougaonih segmenata matrice.

1.4.1 Statički upiti raspona

Za početak ćemo razmatrati statičke upite raspona, kod kojih je zadatak omogućiti efikasno izvršavanje potencijalno velikog broja različitih upita nad segmentima datog niza, pri čemu se vrednosti u nizu ne menjaju.

1.4.1.1 Zbirovi prefiksa

> **Problem**
>
> *Definisati strukturu podataka koja obezbeđuje efikasno izračunavanje zbirova segmenata datog niza (segment određen pozicijama $[a, b]$ se sastoji od uzastopnih elemenata niza od pozicije a do pozicije b, uključujući i njih).*

Rešenje grubom silom bi smestilo sve elemente u niz x i pri svakom upitu iznova računalo zbir elemenata na pozicijama iz intervala $[a, b]$ koji odgovara tom upitu. Ukupno vreme da se svi upiti izvrše je reda $O(mn)$, gde je sa m označen broj upita, a sa n dužina niza, što je u slučaju dugačkih nizova i velikog broja upita nedopustivo neefikasno. I složenost najgoreg slučaja i amortizovana složenost izvršavanja jednog upita su reda $O(n)$.

Jednostavno rešenje je zasnovano na ideji da umesto da čuvamo elemente niza x, čuvamo *niz zbirova prefiksa* (engl. prefix sum array) niza $P_0 = 0$, $P_{i+1} = \sum_{k=0}^{i} x_k$ za $0 \leq i \leq n$. Dakle, zbir P_i čuva zbir prvih i elemenata niza (pošto brojanje pozicija počinje od nule, zbir elemenata $x_0, \ldots, x_i$ sadrži $i + 1$ sabirak i jednak je P_{i+1}). Zbir elemenata svakog segmenta $[a, b]$ onda možemo razložiti na razliku prefiksa niza x do elementa b i prefiksa do elementa $a - 1$:

$$\sum_{k=a}^{b} x_k = \sum_{k=0}^{b} x_k - \sum_{k=0}^{a-1} x_k = P_{b+1} - P_a.$$

Svi zbirovi prefiksa P_i niza x mogu se izračunati algoritmom složenosti $O(n)$ i smestiti u dodatni (a ako je ušteda memorije bitna, onda čak i u originalni) niz. Nakon ovakvog pretprocesiranja, zbir svakog segmenta se može izračunati algoritmom složenosti $O(1)$, pa je ukupna složenost izvršavanja m upita računanja zbira elemenata nekog segmenta niza x jednaka $O(n+m)$.

Ako je poznat niz zbirova prefiksa P, tada se polazni niz x može rekonstruisati računanjem niza razlika susednih elemenata niza prefiksa: $x_i = P_{i+1} - P_i$ za $0 \leq i < n$.

Dvodimenzionalni zbirovi prefiksa

U dvodimenzionalnom slučaju dovoljno je za svaki element matrice na poziciji (v, k) pamtiti zbir $P_{v+1,k+1}$ elemenata pravougaonog segmenta kojem je gornje levo teme polje $(0, 0)$, a donje levo (v, k), pri čemu se u vrsti i koloni 0 nalaze nule. Tada se zbir proizvoljnog segmenta određenog gornjim-levim temenom (v_1, k_1) i donjim-desnim temenom (v_2, k_2) može izraziti kao $P_{v_2+1,k_2+1} - P_{v_2+1,k_1} - P_{v_1,k_2+1} + P_{v_1,k_1}$, kao što je ilustrovano na slici 1.16.

1	3	2	3	2	3	1	1
4	3	1	4	2	4	2	3
1	2	1	3	2	3	2	4
1	4	2	3	2	1	3	1
3	2	4	3	2	2	2	2
4	2	2	1	4	1	3	4
1	3	4	1	2	2	1	3
1	3	2	4	3	3	4	2

0	0	0	0	0	0	0	0	0
0	1	4	6	9	11	14	15	16
0	5	***11***	14	21	25	***32***	35	39
0	6	14	18	28	34	44	49	57
0	7	19	25	38	46	57	65	74
0	10	24	34	50	60	73	83	94
0	14	***30***	42	59	73	***87***	100	115
0	15	34	50	68	84	100	114	132
0	16	38	56	78	97	116	134	154

Slika 1.16: Dvodimenzionalni prefiksni zbirovi za matricu levo su prikazani u matrici desno. Zbir elemenata u crvenom segmentu leve slike se može dobiti izrazom 87 - 30 - 32 + 11. Broj 87 je zbir elemenata matrice od njenog gornjeg levog ugla, do donjeg desnog ugla crvenog pravougaonika, tj. zbir svih elemenata u zelenom, žutom, plavom i crvenom delu matrice (ž+z+p+c). Broj 30 je zbir elemenata matrice od njenog gornjeg levog ugla do donjeg desnog ugla plavog pravougaonika, tj. zbir svih elemenata u zelenom i plavom delu matrice (z+p). Broj 32 je zbir elemenata marice od njenog gornjeg levog ugla do donjeg desnog ugla zelenog pravougaonika tj. zbir svih elemenata u zelenom i žutom pravougaoniku (z+ž). Broj 11 je zbir elemenata matrice od njenog gornjeg levog ugla do donjeg desnog ugla zelenog pravougaonika tj. zbir svih elemenata u zelenom pravougaoniku (z). Izrazom 87 - 32 - 30 + 11, se dakle, računa (z+ž+p+c) - (ž+z) - (p+z) + z, što je jednako c

Niz zbirova prefiksa ima različite primene. Na primer, u obradi slika koristi se za izračunavanje različitih proračuna nad regionima slike (poput zbirova) koji omogućavaju razne operacije nad slikom poput detekcije ivica, zamućenja i drugih. Zbirovi prefiksa imaju primenu i u analizi podataka, obradi podataka, finansijskoj analizi i dr.

1.4.1.2 Razlike susednih elemenata

Donekle srodan problem je i sledeći.

Problem

Definisati strukturu podataka koja omogućava efikasno izvršavanje upita oblika $([a, b], c)$, koji podrazumevaju da se svi elementi niza x na pozicijama iz segmenta $[a, b]$ uvećaju za vrednost c. Potrebno je odrediti sadržaj niza x nakon izvršavanja svih upita.

Direktno rešenje bi za svaki upit u petlji uvećavalo sve elemente niza x na pozicijama iz segmenta $[a, b]$. Složenost tog naivnog pristupa je $O(mn)$, gde je sa m označen broj upita, a sa n dužina niza.

Mnogo efikasnije rešenje se može dobiti ako se umesto elemenata niza pamti *niz razlika svaka dva susedna elementa niza* (engl. difference array): $R_0 = x_0, R_i = x_i - x_{i-1}$ za svako i između 1 i $n-1$. Ključni uvid je da se tokom uvećavanja svih elemenata segmenta $[a, b]$ niza x za vrednost c menjaju samo razlike između elemenata na pozicijama a i $a-1$ (razlika R_a se uvećava za c) kao i između elemenata na pozicijama $b+1$ i b (razlika R_{b+1} se umanjuje za c). Ako znamo sve elemente niza, tada niz razlika susednih elemenata možemo veoma jednostavno izračunati algoritmom složenosti $O(n)$. Sa druge strane, ako znamo niz razlika nekog niza, tada taj niz možemo takođe veoma jednostavno rekonstruisati, izračunavajući zbirove prefiksa niza razlika, čija je vremenska složenost $O(n)$. Na taj način, na osnovu razlika elemenata finalnog niza, možemo lako rekonstruisati taj finalni niz.

Ako ne želimo da u implementaciji prvi i poslednji element niza tretiramo drugačije od ostalih, možemo pretpostaviti da početni niz i niz razlika proširujemo sa po jednom nulom sa leve i desne strane (tada je $R_i = x_{i+1} - x_i$, za svako i od 0 do $n-1$).

Može se primetiti da se rekonstrukcija originalnog niza vrši zapravo izračunavanjem prefiksnih zbirova niza razlika susednih elemenata, što ukazuje na duboku vezu između ove dve tehnike. Zapravo, razlike susednih elemenata predstavljaju određeni diskretni analogon izvoda funkcije, dok prefiksni zbirovi onda predstavljaju analogon određenog integrala. Izračunavanje zbira segmenta kao razlike dva zbira prefiksa odgovara Njutn-Lajbnicovoj formuli.

1.4.2 Dinamički upiti raspona

Za razliku od prethodnih, statičkih upita nad rasponima, ovde ćemo razmatrati tzv. dinamičke upite nad rasponima koji dozvoljavaju da se niz menja tokom vremena, tako da je potrebno razviti naprednije strukture podataka koje omogućavaju izvršavanje oba tipa upita (čitanje i ažuriranje) efikasno.

Naime, niz zbirova prefiksa omogućava efikasno izračunavanje zbirova segmenata niza kod nizova čiji se sadržaj ne menja. Sa druge strane, niz zbirova prefiksa ne omogućava efikasno ažuriranje elemenata niza, jer je pri svakoj izmeni nekog elementa niza potrebno ažurirati i zbirove prefiksa, što je naročito neefikasno kada se ažuriraju elementi blizu početka niza (složenost najgoreg slučaja je $O(n)$), pa se ne mogu primenjivati u situacijama u kojima su upiti izračunavanja zbirova segmenata isprepleteni sa upitima ažuriranja vrednosti elemenata niza.

Slično, niz razlika susednih elemenata dopušta stalna ažuriranja elemenata niza. Međutim, izvršavanje upita kojima se zahteva određivanje elemenata niza podrazumeva rekonstrukciju sadržaja niza na osnovu niza razlika, što je složenosti $O(n)$. Stoga niz razlika nije poželjno upotrebljavati u situacijama kada su upiti uvećavanja segmenata i očitavanja vrednosti elemenata niza ispremletani.

Problemi koje ćemo razmatrati u ovom odeljku su specifični po tome što omogućavaju da se upiti ažuriranja niza i očitavanja njegovih statistika javljaju isprepletano. Za početak razmotrimo malo jednostavniji problem.

> **Problem**
>
> *Definisati strukturu podataka koja obezbeđuje efikasno izračunavanje zbirova segmenata datog niza određenih pozicijama $[a, b]$ (samim tim i pojedinačnih elemenata niza), kao i efikasno menjanje vrednosti pojedinačnih elemenata niza.*

Videćemo da neke strukture podataka dopuštaju da se umesto zbira koriste i neke druge operacije. U narednim poglavljima ćemo razmotriti i složeniji problem u kom će biti dopušteno ažuriranje segmenata niza (a ne samo pojedinačnih elemenata).

Dakle, za početak pretpostavljamo da imamo dat niz od m operacija od kojih je svaka ili izračunavanje zbira nekog segmenta ili ažuriranje pojedinačnog elementa niza i cilj je minimizovati ukupno vreme izvršavanja ovog niza operacija. U ovom slučaju nam nije od koristi struktura podataka kod koje se jedna od ove dve operacije izvršava efikasno, a druga neefikasno jer se može desiti da je većina (ili su sve) od m datih operacija

baš tog drugog tipa. Naravno, treba uzeti u obzir i vreme inicijalizacije strukture podataka, međutim, pošto se inicijalizacija vrši samo jednom, za razliku od upita za koje pretpostavljamo da se izvršavaju puno puta, fokusiraćemo se na složenost vremena izvršavanja upita.

U nastavku ćemo razmotriti dve različite, ali donekle slične strukture podataka koje daju efikasno rešenje prethodnog i njemu sličnih problema: *segmentno drvo* i *Fenvikovo drvo*. Ideja obe ove strukture podataka na neki način nalikuje korišćenju prefiksnih suma: čuvaju se zbirovi nekih unapred pogodno izabranih segmenata i oni se koriste za efikasno računanje zbira proizvoljnog segmenta.

1.4.2.1 Segmentna drveta

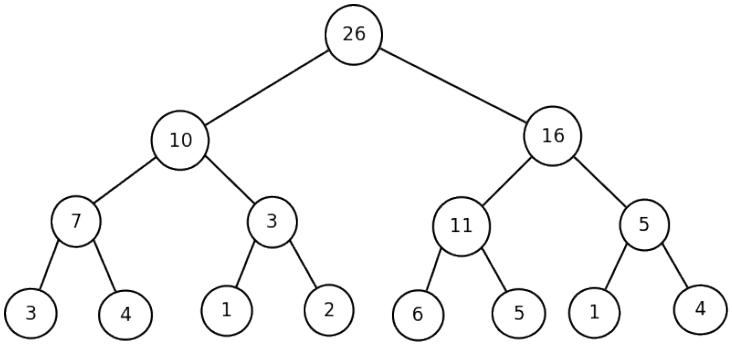
Jedna struktura podataka koja omogućava prilično jednostavno i efikasno rešavanje prethodno opisanog problema je *segmentno drvo* (engl. segment tree). Slično kao kod nizova prefiksa, tokom faze pretprocesiranja izračunavaju se zbirovi segmenata polaznog niza, a onda se zbir elemenata proizvoljnog segmenta polaznog niza izražava putem tih unapred izračunatih zbirova. Razlika je to što se ne izračunavaju zbirovi svih prefiksa, već samo posebno odabranih segmenata, kao i to što se zbir proizvoljnog segmenta u opštem slučaju izračunava obradom nekoliko zbirova segmenata (ne samo dva). Segmentna drveta se ne koriste samo za računanje zbira elemenata, već se mogu koristiti i za druge statistike segmenata koje se izračunavaju asocijativnim operacijama (na primer za određivanje proizvoda, najmanjeg ili najvećeg elementa, nzd-a ili nzs-a svih elemenata i slično). Po istom principu definisane su strukture podataka pod nazivom *kvad-drveta* (engl. quad tree) i *okt-drveta* (engl. oct tree) koja predstavljaju dvodimenzionalna i trodimenzionalna uopštenja segmentnih dreveta.

Segmentna drveta imaju primenu u oblasti računarske geometrije, za probleme koji se rešavaju tehnikom pokretne prave, kao što je pronalaženje presečnih tačaka između duži u ravni, ispitivanje pripadnosti tačke skupu intervala, izračunavanje površine pokrivene skupom pravougaonika u ravni i sl. Koriste se i u drugim oblastima, poput obrade slika i geografskih informacionih sistema.

Opišimo proces konstrukcije segmentnog drveta. Pretpostavimo da je dužina niza stepen broja 2; ako nije, niz se može dopuniti do najbližeg stepena broja 2, u slučaju računanja zbirova nulama[6]. Članovi niza predstavljaju listove drveta. Grupišemo dva po dva susedna čvora i na svakom prethodnom nivou drveta čuvamo roditeljske čvorove koji sadrže zbirove svoja dva deteta.

Primer 1.4.1

Segmentno drvo kojim se efikasno mogu računati zbirovi segmenata niza $3, 4, 1, 2, 6, 5, 1, 4$ *prikazano je na slici 1.17.*



Slika 1.17: Primer segmentnog drveta.

[6]Na dopunjena mesta treba upisati neutralni element za operaciju koja sprovodi, tj. element n koji za svako x zadovoljava da je $f(x, n) = x$. Na primer, ako se segmentnim drvetom računaju proizvodi segmenata, niz se može dopuniti do stepena dvojke jedinicama, ako se računa maksimum, niz se može dopuniti vrednostima $-\infty$, ako se računa minimum niz se može dopuniti vrednostima ∞, ako se računa najveći zajednički delilac, niz se može dopuniti nulama, a ako se računa najmanji zajednički sadržalac, niz se može dopuniti jedinicama.

Pošto je drvo potpuno, najjednostavnija implementacija podrazumeva da se čuva implicitno u nizu (slično kao u slučaju hipa). Pretpostavićemo da elemente drveta smeštamo od pozicije 1, jer je tada aritmetika sa indeksima malo jednostavnija (elementi polaznog niza mogu biti indeksirani i uobičajeno, krenuvši od nule). Koren se smešta na poziciju 1, njegova dva deteta na pozicije 2 i 3, njihova deca na pozicije 4, 5, 6 i 7 itd.

Primer 1.4.2

Drvo iz prethodnog primera se predstavlja sledećim nizom (prvi red sadrži pozicije, a drugi elemente tog niza):

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
-	26	10	16	7	3	11	5	3	4	1	2	6	5	1	4

Uočimo nekoliko karakteristika ovog načina smeštanja elemenata drveta. Koren je smešten na poziciji 1. Ako polazni niz sadrži n elemenata, onda se u segmentnom drvetu elementi polaznog niza nalaze se na pozicijama $[n, 2n-1]$. Element koji se u polaznom nizu nalazi na poziciji p, se u segmentnom drvetu nalazi na poziciji $p+n$. Levo dete čvora na poziciji k nalazi se na poziciji $2k$, a desno na poziciji $2k+1$. Dakle, na parnim pozicijama se nalaze leva deca svojih roditelja, a na neparnim desna. Roditelj čvora k nalazi se na poziciji $\lfloor \frac{k}{2} \rfloor$.

Primer 1.4.3

Na slici 1.17 na kojoj je predstavljeno segmentno drvo za niz od 8 elemenata možemo uočiti da se elementi polaznog niza nalaze na pozicijama $[8, 15]$. *Levo dete čvora sa vrednošću* 10 *koji se nalazi na poziciji* 2 *je čvor sa vrednošću* 7 *koji se nalazi na poziciji* $2 \cdot 2 = 4$, *a desno dete je čvor sa vrednošću* 3 *koji se nalazi na poziciji* $2 \cdot 2 + 1 = 5$. *Roditelj i čvora na poziciji* 4 *i čvora na poziciji* 5 *je čvor koji se nalazi na poziciji* $\lfloor \frac{4}{2} \rfloor = \lfloor \frac{5}{2} \rfloor = 2$.

Formiranje segmentnog drveta

Formiranje segmentnog drveta na osnovu datog niza je veoma jednostavno.

Iterativni postupak

Najpre se elementi polaznog niza prekopiraju u drvo, krenuvši od pozicije n. Zatim se svi unutrašnji čvorovi drveta (od pozicije $n-1$, pa unazad do pozicije 1) popunjavaju kao zbirovi svoje dece (na poziciju k upisujemo zbir elemenata na pozicijama $2k$ i $2k+1$). Korektnost ovog postupka lako se dokazuje indukcijom.

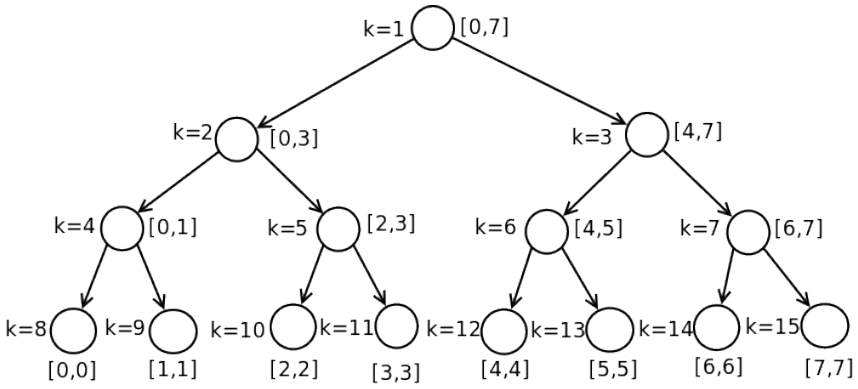
```
// na osnovu datog niza a dužine n u kom su elementi smešteni od
// pozicije 0 formira se segmentno drvo i elementi mu se smeštaju u
// niz drvo krenuvši od pozicije 1
vector<int> formirajSegmentnoDrvo(const vector<int>& a) {
  int n = stepenDvojke(a.size());
  vector<int> drvo(2*n, 0);
  // kopiramo originalni niz u listove (u niz drvo, od pozicije n nadalje)
  copy(begin(a), end(a), next(begin(drvo), n));
  // ažuriramo roditelje već upisanih elemenata
  for (int k = n-1; k >= 1; k--)
    drvo[k] = drvo[2*k] + drvo[2*k+1];
  return drvo;
}
```

Složenost ove operacije je očigledno linearna u odnosu na dužinu niza n.

Prethodni pristup formira drvo odozdo naviše (prvo se popune listovi, pa onda unutrašnji čvorovi sve dok se ne dođe do korena).

Rekurzivni postupak

Još jedan način da se segmentno drvo formira je rekurzivno, odozgo naniže. Iako je ova implementacija komplikovanija i malo neefikasnija (doduše ne asimpotski) usled rekurzivnih poziva, pristup odozgo naniže je u nekim kasnijim operacijama neizbežan, pa ga ilustrujemo na ovom jednostavnom primeru. Svaki čvor drveta predstavlja zbir određenog segmenta pozicija polaznog niza. Segment je jednoznačno određen pozicijom k u nizu koji odgovara segmentnom drvetu, ali da bismo olakšali implementaciju, granice tog segmenta možemo kroz rekurziju prosleđivati kao parametar funkcije, zajedno sa vrednošću k (neka je to segment $[x, y]$). Na slici 1.18 za svaki čvor segmentnog drveta dat je njegov indeks u odgovarajućem nizu `drvo` i granice segmenta koje taj čvor pokriva.



Slika 1.18: Prikaz segmentnog drveta: za svaki čvor drveta prikazan je njegov indeks u nizu kojim je drvo predstavljeno, kao i granice segmenta koji taj čvor pokriva.

Drvo krećemo da gradimo od korena gde je $k = 1$ i $[x, y] = [0, n - 1]$. Ako roditeljski čvor pokriva segment $[x, y]$, tada levo dete pokriva segment $[x, \lfloor \frac{x+y}{2} \rfloor]$, a desno dete pokriva segment $[\lfloor \frac{x+y}{2} \rfloor + 1, y]$. Drvo popunjavamo rekurzivno, tako što najpre popunimo levo poddrvo, zatim desno poddrvo i na kraju vrednost u korenu izračunavamo kao zbir vrednosti u levom i desnom detetu. Izlaz iz rekurzije predstavljaju listovi, koje prepoznajemo po tome što pokrivaju segmente dužine 1, i u njih samo kopiramo elemente sa odgovarajućih pozicija polaznog niza.

```
// od elemenata niza a sa pozicija [x, y] formira se segmentno drvo i
// elementi mu se smeštaju u niz drvo krenuvši od pozicije k
void formirajSegmentnoDrvo(const vector<int>& a, vector<int>& drvo,
                           size_t k, size_t x, size_t y) {
  if (x == y)
    // u listove prepisujemo elemente polaznog niza
    drvo[k] = x < a.size() ? a[x] : 0;
  else {
    // rekurzivno formiramo levo i desno poddrvo
    int s = (x + y) / 2;
    formirajSegmentnoDrvo(a, drvo, 2*k, x, s);
    formirajSegmentnoDrvo(a, drvo, 2*k+1, s+1, y);
    // izračunavamo vrednost u korenu
    drvo[k] = drvo[2*k] + drvo[2*k+1];
  }
}

// na osnovu datog niza a duzine n u kom su elementi smesteni od
```

```
// pozicije 0 formira se segmentno drvo i elementi mu se smeštaju u
// niz drvo krenuvši od pozicije 1
vector<int> formirajSegmentnoDrvo(const vector<int>& a) {
  // niz implicitno dopunjujemo nulama tako da mu duzina postane
  // najblizi stepen dvojke
  int n = stepenDvojke(a.size());
  vector<int> drvo(n * 2);
  // krećemo formiranje od korena, koji se nalazi u nizu drvo
  // na poziciji 1 i pokriva elemente na pozicijama [0, n-1]
  formirajSegmentnoDrvo(a, drvo, 1, 0, n - 1);
  return drvo;
}
```

Vremensku složenost prethodne rekurzivne implementacije možemo opisati narednom rekurentnom jednačinom: $T(n) = 2T(n/2) + O(1), T(1) = O(1)$. Njeno rešenje se dobija direktno iz master teoreme i iznosi $O(n)$, što je isto kao i u slučaju iterativnog formiranja segmentnog drveta.

Računanje zbira elemenata segmenta

I zbir elemenata možemo izračunati bilo iterativnim, bilo rekurzivnim postupkom.

Iterativni postupak

Ilustrujmo iterativni postupak (kažemo i postupak odozdo naviše) pomoću nekoliko primera.

Primer 1.4.4

Razmotrimo kako bismo našli zbir elemenata niza $3, 4, 1, 2, 6, 5, 1, 4$ *na pozicijama iz segmenta* $[2, 6]$*, tj. zbir elemenata* $1, 2, 6, 5$ *i* 1*. U segmentnom drvetu taj segment je smešten na pozicijama iz segmenta* $[2 + 8, 6 + 8] = [10, 14]$ *(slika 1.19). Jasno je da ne treba ići do listova i redom sabirati elemente, već na pametan način iskoristiti već izračunate zbirove segmenata koje se nalaze u drvetu.*

Zbir prva dva elementa $1, 2$ *(koji su na pozicijama* 10 *i* 11*) se nalazi u čvoru iznad njih (na poziciji* 5*), zbir naredna dva elemenata* 6 *i* 5 *(koji su na pozicijama* 12 *i* 13*) se nalazi takođe u čvoru iznad njih (na poziciji* 6*), dok se u roditeljskom čvoru elementa* 1 *(koji je na poziciji* 14*) nalazi njegov zbir sa elementom* 4 *(koji je na poziciji* 15 *i koji ne pripada segmentu koji sabiramo). Zato zbir elemenata na pozicijama iz segmenta* $[10, 14]$ *u segmentnom drvetu možemo razložiti na zbir elemenata na pozicijama iz segmenta* $[5, 6]$ *i elementa na poziciji* 14 *koji odmah zasebno dodajemo na traženi zbir. Na ovaj način penjemo se na nivo iznad tekućeg (dolazimo do roditelja tih čvorova) i nastavljamo postupak računajući zbir elemenata na pozicijama iz segmenta* $[5, 6]$*.*

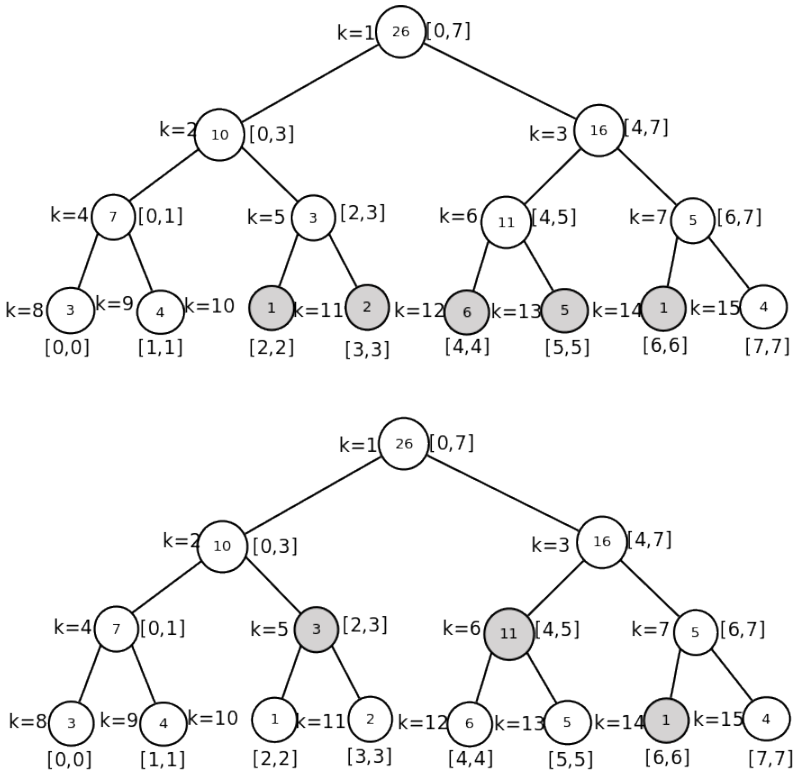
Na poziciji 5 *nalazi se element* 3*, a na poziciji* 6 *element* 11*. U roditeljskom čvoru elementa* 3 *(na poziciji* 5*) nalazi se zbir sa elementom levo od njega, koji ne pripada segmentu koji treba da saberemo. Slično, u roditeljskom čvoru elementa* 11 *(na poziciji* 6*) nalazi se zbir ovog elementa sa elementom desno od njega koji ne pripada segmentu čiji zbir računamo. Stoga oba elementa* 3 *i* 11 *dodajemo zasebno na zbir i ostajemo sa praznim segmentom, čime se postupak računanja zbira datog segmenta završava.*

Primer 1.4.5

Razmotrimo kako bismo računali zbir elemenata na pozicijama iz segmenta $[3, 7]$*, tj. zbir elemenata* $2, 6, 5, 1, 4$*. U segmentnom drvetu taj segment je smešten na pozicijama* $[3 + 8, 7 + 8] = [11, 15]$*.*

U roditeljskom čvoru elementa 2 *(koji je na poziciji* 11*) nalazi se njegov zbir sa elementom* 1*, levo od njega koji ne pripada segmentu koji sabiramo. Stoga element* 2 *odmah dodajemo na zbir.*

Zbirovi elemenata 6 *i* 5 *(na pozicijama* 12 *i* 13*) i elemenata* 1 *i* 4 *(na pozicijama* 14 *i* 15*) se nalaze u čvorovima iznad njih (na pozicijama* 6 *i* 7*), pa se problem svodi na izračunavanje zbira elemenata na pozicijama* 6 *i* 7*.*



Slika 1.19: Računanje zbira elemenata iz segmenta $[2, 6]$ pristupom odozdo naviše. Na prvoj slici su sivom bojom označeni listovi koji odgovaraju traženom segmentu, a na drugoj čvorovi čije se vrednosti sabiraju.

Taj zbir je već izračunat, iznosi 16 *i nalazi se na poziciji* 3*, tako da je samo potrebno da i njega dodamo na zbir.*

Dakle, rezultat se dobija sabiranjem elementa 2 *na poziciji* 11 *i elementa* 16 *na poziciji* 3*.*

Generalno, za sve unutrašnje elemente segmenta čiji zbir računamo smo sigurni da se njihov zbir nalazi u čvorovima iznad njih. Jedini izuzetak mogu da budu elementi na krajevima segmenta.

- Ako je element na levom kraju segmenta levo dete (što je ekvivalentno tome da se nalazi na parnoj poziciji) tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom desno od njega koji takođe pripada segmentu koji treba sabrati (osim eventualno u slučaju jednočlanog segmenta).
- U suprotnom (ako se nalazi na neparnoj poziciji), u njegovom roditeljskom čvoru je njegov zbir sa elementom levo od njega, koji ne pripada segmentu koji sabiramo. U toj situaciji, taj element ćemo posebno dodati na zbir i isključiti iz segmenta koji sabiramo pomoću roditeljskih čvorova.
- Ako je element na desnom kraju segmenta levo dete (ako se nalazi na parnoj poziciji), tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom desno od njega, koji ne pripada segmentu koji sabiramo. I u toj situaciji, taj element ćemo posebno dodati na zbir i isključiti iz segmenta koji sabiramo pomoću roditeljskih čvorova.
- Konačno, ako se krajnji desni element nalazi u desnom čvoru (ako je na neparnoj poziciji), tada se u njegovom roditeljskom čvoru nalazi njegov zbir sa elementom levo od njega, koji pripada segmentu koji sabiramo (osim eventualno u slučaju jednočlanog segmenta).

```
// izračunava se zbir elemenata polaznog niza dužine n koji se
// nalaze na pozicijama iz segmenta [a, b] na osnovu segmentnog drveta
// koje je smešteno u nizu drvo, krenuvši od pozicije 1
```

```
int zbirSegmenta(const vector<int>& drvo, int a, int b) {
  int n = drvo.size() / 2;
  a += n; b += n;
  int zbir = 0;
  // sve dok je segment neprazan
  while (a <= b) {
    // ako je levi kraj segmenta desno dete, dodajemo ga posebno u zbir
    if (a % 2 == 1) zbir += drvo[a++];
    // ako je desni kraj segmenta levo dete, dodajemo ga posebno u zbir
    if (b % 2 == 0) zbir += drvo[b--];
    // penjemo se na nivo iznad, na roditeljske cvorove
    a /= 2;
    b /= 2;
  }
  return zbir;
}
```

Pošto se u svakom koraku dužina segmenta $[a, b]$ polovi, a ona je inicijalno sigurno manja ili jednaka n, složenost ove operacije je $O(\log n)$.

Rekurzivni postupak

Prethodna implementacija vrši izračunavanje odozdo naviše. I za ovu operaciju možemo napraviti rekurzivnu implementaciju koja vrši izračunavanje odozgo naniže. Funkcija kao argument prima čvor drveta (koji je određen pozicijom k i koji sadrži zbir elemenata na pozicijama iz segmenta $[x, y]$). U opštem slučaju, neki elementi na pozicijama segmenta $[a, b]$ čiji zbir elemenata računamo su levo od tekućeg čvora, a neki desno (ili su svi levo ili svi desno). Za svaki čvor u segmentnom drvetu funkcija vraća koliki je doprinos segmenta koji odgovara tom čvoru i njegovim naslednicima traženom zbiru elemenata na pozicijama iz segmenta $[a, b]$ u polaznom nizu. Na početku krećemo od korena i računamo doprinos celog drveta zbiru elemenata iz segmenta $[a, b]$. Postoje tri različita moguća odnosa između segmenta $[x, y]$ koji odgovara tekućem čvoru i segmenta $[a, b]$ čiji zbir elemenata tražimo.

- Ako su segmenti disjunktni, tj. ako je $y < a$ ili je $x > b$, doprinos tekućeg čvora zbiru segmenta $[a, b]$ je nula.
- Ako je segment $[x, y]$ u potpunosti sadržan u segmentu $[a, b]$, tj. ako je $a \leq x$ i $y \leq b$, tada je doprinos potpun, tj. ceo zbir segmenta $[x, y]$ (a to je broj upisan u nizu na poziciji k) doprinosi zbiru elemenata na pozicijama iz segmenta $[a, b]$.
- U suprotnom, segmenti se seku i tada je doprinos tekućeg čvora jednak zbiru doprinosa njegovog levog i desnog deteta.

Iz ovog razmatranja sledi naredna implementacija.

```
// izračunava se zbir onih elemenata polaznog niza, koji se nalaze na
// pozicijama iz segmenta [a, b] i koji se ujedno nalaze u delu
// segmentnog drveta ciji je koren u nizu drvo na poziciji k (taj deo
// drveta pokriva elemente na pozicijama segmenta [x, y] originalnog niza)
int zbirSegmenta(const vector<int>& drvo, int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // segment [x, y] je potpuno sadržan unutar segmenta [a, b]
  if (a <= x && y <= b)
      return drvo[k];
```

```
  // segmenti [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return zbirSegmenta(drvo, 2*k, x, s, a, b) +
         zbirSegmenta(drvo, 2*k+1, s+1, y, a, b);
}

// izračunava se zbir elemenata polaznog niza na pozicijama iz
// segmenta [a, b], na osnovu segmentnog drveta smeštenog u nizu drvo
int zbirSegmenta(const vector<int>& drvo, int a, int b) {
  int n = drvo.size() / 2;
  // krećemo od drveta smeštenog od pozicije 1, koje
  // pokriva elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  return zbirSegmenta(drvo, 1, 0, n-1, a, b);
}
```

Primer 1.4.6

Razmotrimo kako se ovim pristupom određuje zbir elemenata na pozicijama iz segmenta $[2, 6]$ *u originalnom nizu (u segmentnom drvetu ti elementi su na pozicijama* $[2 + 8, 6 + 8] = [10, 14]$*), prikazan na slici 1.20.*

- *Izvršavanje kreće od korena. Segment* $[0, 7]$ *se seče sa segmentom* $[2, 6]$ *te će zbir biti jednak sumi doprinosa segmenata* $[0, 3]$ *i* $[4, 7]$.
- *Segment* $[0, 3]$ *se seče sa segmentom* $[2, 6]$ *te opet pravimo dva rekurzivna poziva za segmente* $[0, 1]$ *i* $[2, 3]$.
- *Segment* $[0, 1]$ *je disjunktan sa segmentom* $[2, 6]$ *pa je njegov doprinos traženoj sumi* 0.
- *Segment* $[2, 3]$ *je sadržan u segmentu* $[2, 6]$ *te je njegov doprinos potpun – jednak je vrednosti* 3 *koju taj čvor čuva.*
- *S druge strane, segment* $[4, 7]$ *se seče sa segmentom* $[2, 6]$ *te opet pravimo dva rekurzivna poziva za segmente* $[4, 5]$ *i* $[6, 7]$.
- *Segment* $[4, 5]$ *je u potpunosti sadržan u segmentu* $[2, 6]$ *te je njegov doprinos potpun i iznosi* 11.
- *Segment* $[6, 7]$ *se seče sa segmentom* $[2, 6]$*, pa iz njega startujemo dva rekurzivna poziva: za segmente* $[6, 6]$ *i* $[7, 7]$.
- *Segment* $[6, 6]$ *je u potpunosti sadržan u segmentu čiji zbir računamo, te je njegov doprinos potpun i iznosi* 1.
- *Segment* $[7, 7]$ *je disjunktan sa segmentom čiji zbir računamo, pa je njegov doprinos jednak nula. Dakle, ukupna vrednost sume segmenta biće jednaka* $3 + 11 + 1 = 15$.

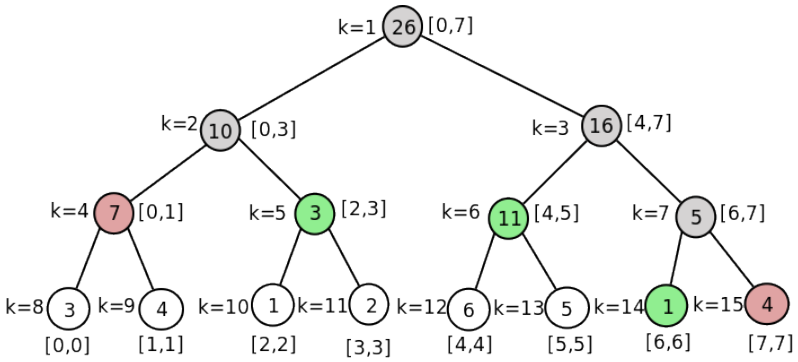
I ova implementacija ima složenost $O(\log n)$. Dokažimo to.

Lema 1.4.1

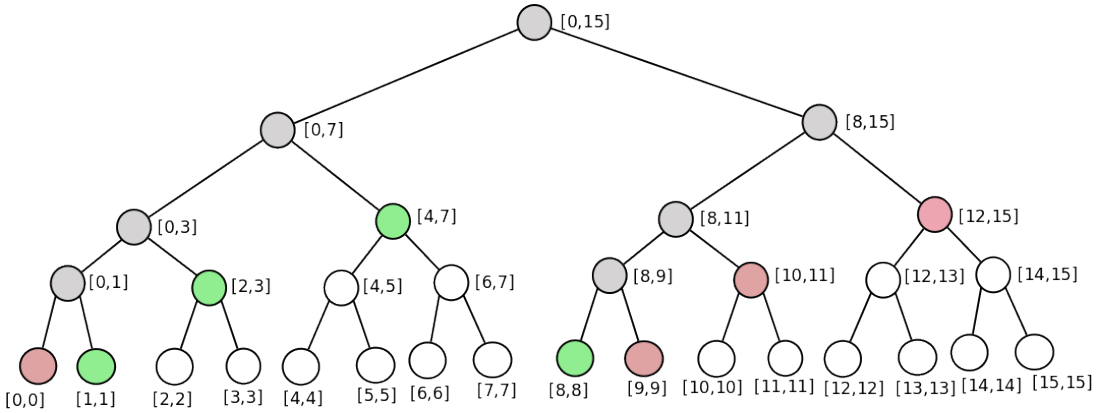
Prilikom rekurzivnog obilaska segmentnog drveta, za svaki nivo segmentnog drveta obilaze se najviše po četiri čvora.

Dokaz. Dokažimo ovo tvrđenje principom matematičke indukcije.

Na prvom nivou se posećuje samo jedan čvor, koren drveta, tako da se na ovom nivou posećuje manje od četiri čvora i baza indukcije važi.



Slika 1.20: Računanje zbira elemenata iz segmenta $[2, 6]$ pristupom odozgo naniže. Sivom bojom su označeni čvorovi koji odgovaraju segmentima koji se seku sa traženim segmentom, roze bojom čvorovi koji su disjunktni, a zelenom bojom čvorovi koji odgovaraju segmentima koji su u potpunosti sadržani u traženom segmentu.



Slika 1.21: Računanje zbira elemenata iz segmenta $[1, 8]$ pristupom odozgo naniže. Sivom bojom su označeni čvorovi koji odgovaraju segmentima koji se seku sa traženim segmentom, roze bojom čvorovi koji su disjunktni, a zelenom bojom čvorovi koji odgovaraju segmentima koji su u potpunosti sadržani u traženom segmentu.

Razmotrimo sada proizvoljni nivo drveta: prema induktivnoj hipotezi na njemu se posećuje najviše četiri čvora.

- Ako se posećuje najviše dva čvora, u narednom nivou se posećuje najviše četiri čvora jer svaki čvor može da proizvede najviše dva rekurzivna poziva.
- Pretpostavimo da se na tekućem nivou posećuju tri ili četiri čvora. Oni mogu biti susedni (slika 1.20, nivo 2), ali i ne moraju (slika 1.21, nivo 3). Pošto čvorovi na jednom nivou segmentnog drveta sadrže sume disjunktnih segmenata, jedini čvorovi iz kojih se mogu pokrenuti rekurzivni pozivi su oni koji sadrže granice segmenta čiju sumu računamo (sivi čvorovi na slici): ostali segmenti će biti ili u potpunosti sadržani ili disjunktni sa segmentom čiju sumu računamo (ili zeleni, ili crveni na slici). Stoga, na svakom nivou postoje samo dva čvora iz kojih se mogu napraviti rekurzivni pozivi, tako da će i naredni nivo zadovoljavati polazno tvrđenje.

□

Pošto je visina segmentnog drveta $O(\log n)$, ukupno se posećuje najviše $4 \log n$ čvorova segmentnog drveta, te je složenost operacije izračunavanja zbira segmenta pristupom odozgo naniže takođe $O(\log n)$.

Ažuriranje vrednosti elementa

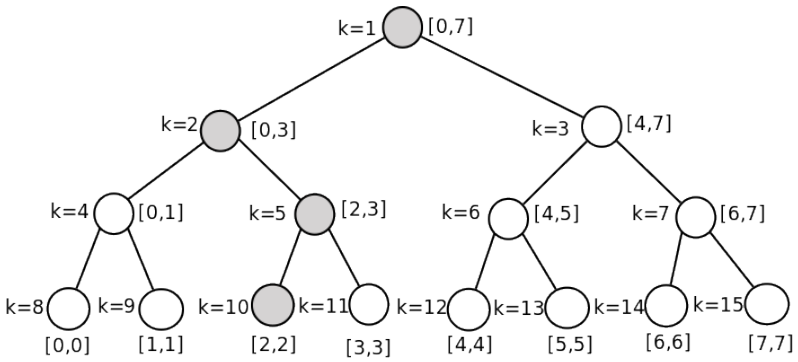
I funkciju za ažuriranje segmentnog drveta pri ažuriranju vrednosti nekog elementa polaznog niza možemo implementirati iterativno i rekurzivno.

Iterativni postupak

Prilikom ažuriranja nekog elementa potrebno je ažurirati sve čvorove na putanji od tog lista do korena. S obzirom na to da znamo poziciju roditelja svakog čvora, ova operacija se može veoma jednostavno iterativno implementirati.

Primer 1.4.7

Na slici 1.22 je na jednom primeru prikazano koje sve čvorove je potrebno ažurirati nakon izmene elementa na poziciji 2 u originalnom nizu.



Slika 1.22: Ažuriranje elementa na poziciji 2 u polaznom nizu. Sivom bojom su označeni čvorovi čije se vrednosti ažuriraju.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije 1
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void upisi(vector<int>& drvo, int i, int v) {
  int n = drvo.size() / 2;
  // prvo ažuriramo odgovarajući list
```

```
  int k = i + n;
  drvo[k] = v;
  // ažuriramo sve roditelje izmenjenih čvorova
  for (k /= 2; k >= 1; k /= 2)
    drvo[k] = drvo[2*k] + drvo[2*k+1];
}
```

Pošto se k polovi u svakom koraku petlje, a kreće od vrednosti najviše $2n - 1$, i složenost ove operacije je $O(\log n)$.

Rekurzivni postupak

I ovu operaciju možemo implementirati odozgo naniže.

```
// ažurira segmentno drvo smešteno u niz drvo od pozicije k,
// koje sadrži elemente polaznog niza a dužine n sa pozicija iz
// segmenta [x, y], nakon što se na poziciju i niza upiše vrednost v
void upisi(vector<int>& drvo, int k, int x, int y, int i, int v) {
   if (x == y)
     // ažuriramo vrednost u listu
     drvo[k] = v;
   else {
     // proveravamo da li se pozicija i nalazi levo ili desno
     // i u zavisnosti od toga ažuriramo odgovarajuće poddrvo
     int s = (x + y) / 2;
     if (x <= i && i <= s)
       upisi(drvo, 2*k, x, s, i, v);
     else
       upisi(drvo, 2*k+1, s+1, y, i, v);
     // pošto se promenila vrednost u nekom od dva poddrveta
     // moramo ažurirati vrednost u korenu
     drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// ažurira segmentno drvo smešteno u niz drvo od pozicije 1,
// koje sadrži elemente polaznog niza a dužine n u kom su elementi
// smešteni od pozicije 0, nakon što se na poziciju i polaznog
// niza upiše vrednost v
void upisi(vector<int>& drvo, int i, int v) {
  int n = drvo.size() / 2;
  // krećemo od drveta smeštenog od pozicije 1 koje
  // sadrži elemente polaznog niza na pozicijama iz segmenta [0, n-1]
  upisi(drvo, 1, 0, n-1, i, v);
}
```

Složenost prethodne implementacije možemo opisati rekurentnom jednačinom: $T(n) = T(n/2) + O(1), T(1) = O(1)$ i njeno rešenje iznosi $O(\log n)$. Naime, dužina intervala $[x, y]$ se u svakom narednom pozivu smanjuje dva puta, a njegova početna dužina je jednaka n.

Umesto funkcije `upisi` često se razmatra funkcija `uvecaj` koja element na poziciji `i` polaznog niza uvećava za datu vrednost `v` i u skladu sa tim ažurira segmentno drvo. Svaka od ove dve funkcije se jednostavno izražava preko one druge.

Druge operacije

Osim za pronalaženje zbira elemenata segmenata, segmentno drvo se može koristiti i za mnoge druge operacije koje imaju svojstvo asocijativnosti. Najčešće su to pronalaženje minimuma ili maksimuma segmenta, NZD ili NZS svih elemenata segmenta i slično. U tim slučajevima se implementacija veoma jednostavno prilagođava tako se na svim mestima u kodu operacija sabiranja menja odgovarajućom binarnom asocijativnom operacijom (`min`, `max`, `nzd`, `nzs` i slično). Pritom, ako se niz dopunjava do dužine koja je stepen broja 2, umesto vrednosti 0, koriste se neutralni elementi odgovarajućih operacija (na primer, za `min` to je vrednost $+\infty$, za `max` to je $-\infty$, za `nzd` to je 0, a za `nzs` to je 1).

Neke malo naprednije modifikacije omogućavaju i izvršavanje složenijih upita. Na primer, možemo da napravimo drvo koje za svaki segment određuje maksimalnu vrednost tog segmenta i ujedno broj pojavljivanja te maksimalne vrednosti u tom segmentu. Tada se u svakom čvoru čuva par (m, n) sačinjen od te dve vrednosti. Parovi se kombinuju na sledeći način. Ako je vrednost u levom detetu čvora (m_l, n_l), a u desnom (m_d, n_d), tada, ako je $m_l > m_d$, važi $(m, n) = (m_l, n_l)$, ako je $m_d > m_l$, važi $(m, n) = (m_d, n_d)$, a ako je $m_l = m_d$, važi $(m, n) = (m_l, n_l + n_d)$.

Navedimo još neke tipične operacije: određivanje broja elemenata u segmentu koji zadovoljavaju dati uslov (na primer, određivanje broja nula u svakom segmentu ili broja parnih brojeva u svakom segmentu), određivanje pozicije k-tog po redu elementa u segmentu koji zadovoljava dati uslov, određivanje pozicije prvog elementa u segmentu koji je veći od date vrednosti, određivanje prve pozicije u segmentu niza koji sadrži samo nenegativne vrednosti takve da je zbir prefiksa segmenta do te pozicije veći ili jednak od date vrednosti i slično.

Napomenimo da se neke od ovih operacija mogu realizovati pomoću običnog segmentnog drveta i binarne pretrage u složenosti $O(\log^2 n)$, a prilagođavanjem segmentnog drveta u složenosti $O(\log n)$. Na primer, u nizu nenegativnih brojeva za dati segment određen pozicijama $[l, d]$, binarnom pretragom intervala $[l, d]$ možemo odrediti najmanju vrednost $k \in [l, d]$ tako da je zbir prefiksa određenog pozicijama $[l, k]$ veći od date vrednosti x. Binarna pretraga zahteva $O(\log (d - l + 1)) = O(\log n)$ koraka, a u svakom koraku zbir segmenta određenog pozicijama $[l, k]$ možemo određivati pomoću segmentnog drveta u vremenu $O(\log n)$ (jer je prefiks određen pozicijama $[l, k]$ segmenta određenog pozicijama $[l, d]$ ujedno segment originalnog niza). Sa druge strane, možemo definisati rekurzivnu funkciju koja u jednom prolasku kroz segmentno drvo pronalazi traženu vrednost k. Naime, ako je vrednost levog deteta (zbira elemenata u levoj polovini trenutnog segmenta) veća ili jednaka od vrednosti x, onda prefiks rekurzivno tražimo u levom detetu, a ako je manja od vrednosti x, onda u desnom detetu tražimo najkraći prefiks čiji je zbir veći ili jednak od razlike vrednosti x i vrednosti levog deteta (što je zbir elemenata u levoj polovini trenutnog segmenta). Time se upit izvršava u vremenu $O(\log n)$.

1.4.2.2 Fenvikova drveta (BIT)

U nastavku ćemo razmotriti još jednu strukturu podataka koja omogućava efikasno računanje statistika nad segmentima niza i ažuriranje pojedinačnih elemenata: to su *Fenvikova drveta* (engl. Fenwick tree), tj. *binarno indeksirana drveta* (engl. binary indexed tree, BIT) koja se malo jednostavnije implementiraju od segmentnih drveta, koriste malo manje memorije i mogu da budu za konstantni faktor brža od njih (iako je složenost operacija asimptotski jednaka). S druge strane, za razliku od segmentnih drveta, koja su pogodna za različite operacije, Fenvikova drveta su specijalizovana samo za asocijativne operacije koje imaju odgovarajuće inverzne (suprotne) operacije (npr. zbir ili proizvod elemenata segmenata se može nalaziti uz pomoć Fenvikovih drveta, jer sabiranju odgovara oduzimanje, a množenju deljenje, ali ne i minimum, najveći zajednički delilac i slično). Potrebu da za razmatranu operaciju postoji inverzna operacija ćemo detaljnije diskutovati kada budemo govorili o realizaciji samih operacija. Segmentna drveta mogu da urade sve što i Fenvikova, dok obratno ne važi.

Slično kao što je slučaj kod segmentnih drveta, iako se naziva drvetom, Fenvikovo drvo zapravo predstavlja niz vrednosti zbirova nekih pametno izabranih segmenata originalnog niza a. Izbor segmenata je u tesnoj vezi sa binarnom reprezentacijom indeksa. Ključna Fenvikova ideja je sledeća:

Kao što se svaki prirodan broj može dobiti sabiranjem nekih stepena dvojke koji su određeni njegovom binarnom reprezentacijom (svaka jedinica i njen položaj u binarnoj reprezentaciji određuje neki stepen dvojke), tako se i svaki prefiks niza može dobiti nadovezivanjem nekih segmenata koji su određeni binarnom reprezentacijom granice tog segmenta (svaka jedinica i njen položaj u binarnoj reprezentaciji određuje jedan takav segment).

Ponovo ćemo, jednostavnosti radi, pretpostaviti da se vrednosti smeštaju od pozicije 1 (vrednost na poziciji 0 je irelevantna) i to i u polaznom nizu a, i u nizu u kom se smešta Fenvikovo drvo. Prilagođavanje koda situaciji u kojoj su u polaznom nizu elementi smešteni od pozicije nula veoma je jednostavno, samo je na početku svake funkcije koja radi sa drvetom indeks polaznog niza potrebno uvećati za jedan pre dalje obrade. Dakle, ako je polazni niz dužine n, elementi drveta se smeštaju u poseban niz na pozicije $[1, n]$.

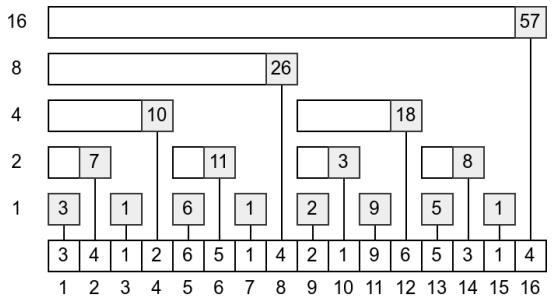
Definicija Fenvikovog drveta je sledeća:

U drvetu se na poziciji k čuva zbir vrednosti polaznog niza sa pozicija iz intervala $(f(k), k]$, gde je $f(k)$ broj koji se dobija od broja k tako što se u binarnom zapisu broja k prva jedinica zdesna zameni nulom.

Primer 1.4.8

- *U Fenvikovom drvetu se na poziciji $k = 21$ zapisuje zbir elemenata polaznog niza sa pozicija iz intervala $(20, 21]$. Naime, broj 21 se binarno zapisuje kao $(10101)_2$ i zamenom prve jedinice zdesna nulom u njegovom binarnom zapisu dobija se binarni zapis $(10100)_2$, tj. broj 20 (važi $f(21) = 20$).*
- *Na poziciji 20 u Fenvikovom drvetu nalazi se zbir elemenata polaznog niza sa pozicija iz intervala $(16, 20]$, jer se brisanjem krajnje desne jedinice iz njegovog binarnog zapisa $(10100)_2$ dobija binarni zapis $(10000)_2$ tj. broj 16 (važi $f(20) = 16$).*
- *Na poziciji 16 u Fenvikovom drvetu čuva se zbir elemenata polaznog niza sa pozicija iz intervala $(0, 16]$, jer se brisanjem kranje desne jedinice iz binarnog zapisa broja 16 dobija 0 (važi $f(16) = 0$).*

Na slici 1.23 je za niz dužine 16 prikazano koji se zbirovi segmenata polaznog niza čuvaju na svakoj od pozicija u Fenvikovom drvetu. Primetimo da neparne pozicije odgovaraju segmentima dužine 1, a da za stepene broja 2 segmenti predstavljaju prefikse polaznog niza do te pozicije.



Slika 1.23: Prikaz segmenata čiji su zbirovi elemenata smešteni u elementima Fenvikovog drveta dužine 16. Na dnu je prikazan polazni niz, segmenti su označeni pravougaonicima, a zbir elemenata svakog segmenta je prikazan u sivom kvadratu koji se nalazi na poziciji na kojoj je taj zbir smešten u Fenvikovom drvetu kada se ono predstavi pomoću niza.

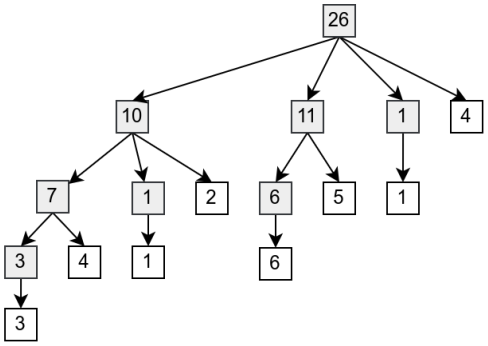
Primer 1.4.9

Razmotrimo niz a sa vrednostima $3, 4, 1, 2, 6, 5, 1, 4$. Odgovarajuće Fenvikovo drvo čuva sledeće vrednosti:

0	1	2	3	4	5	6	7	8	k
	0001	0010	0011	0100	0101	0110	0111	1000	k *binarno*

0	1	2	3	4	5	6	7	8	k
	0000	0000	0010	0000	0100	0100	0110	0000	$f(k)$ *binarno*
	0	0	2	0	4	4	6	0	$f(k)$
	$(0,1]$	$(0,2]$	$(2,3]$	$(0,4]$	$(4,5]$	$(4,6]$	$(6,7]$	$(0,8]$	*interval*
	3	4	1	2	6	5	1	4	*niz* a
	3	7	1	10	6	11	1	26	*Fenvikovo drvo*

Na slici 1.24 prikazana je zavisnost zbirova segmenata od njihovih podsegmenata, odakle se jasno vidi da je zaista u pitanju drvolika struktura (otuda i potiče naziv Fenvikovo drvo). U listovima ove strukture nalaze se elementi polaznog niza, dok se u unutrašnjim čvorovima nalaze elementi Fenvikovog drveta.



Slika 1.24: Drvolika struktura Fenvikovog drveta

Implementacija raznih operacija nad Fenvikovim drvetom je veoma jednostavna, ako se zna način da se iz binarnog zapisa broja ukloni prva jedinica zdesna tj. da se za dati broj k izračuna vrednost $f(k)$. Označimo sa $b(k)$ binarni broj koji sadrži samo jednu jedinicu i to na mestu poslednje jedinice u binarnom zapisu broja k. Važi da je $f(k) = k - b(k)$.

Pod pretpostavkom da su brojevi zapisani u potpunom komplementu, vrednost $b(k)$ se može izračunati izrazom `k & -k`. Oduzimanjem te vrednosti od broja k tj. izrazom `k - (k & -k)` dobijamo efekat brisanja poslednje jedinice u binarnom zapisu broja k i to predstavlja implementaciju funkcije f.

Primer 1.4.10

Broju $k = 20$ *odgovara binarni zapis* $(00010100)_2$*, a broju* $-k$ *binarni zapis* $(11101100)_2$*, te je* `k & -k` *jednako* $(00000100)_2$ *i predstavlja broj koji sadrži samo jednu jedinicu i to na mestu poslednje jedinice u zapisu broja* k*. Oduzimanjem ove vrednosti od broja* k *dobijamo broj* 16 *kome odgovara binarna reprezentacija* $(00010000)_2$.

Dokažimo ispravnost operacije uklanjanje poslednje jedinice iz binarnog zapisa broja.

Lema 1.4.2

Za bilo koji neoznačeni binarni broj k *veći od nule izrazom* `k & -k` *se izračunava broj* $b(k)$ *tj. broj koji se sastoji samo od poslednje jednice u binarnom zapisu broja* k*, a izrazom* `k - (k & -k)` *se izračunava broj* $f(k)$ *dobijen od broja* k *brisanjem poslednje jedinice iz njegovog binarnog zapisa.*

Dokaz. Potpuni komplement broja k se dobija tako što se broju k invertuju svi bitovi i dobijeni rezultat se uveća za 1. Neka broj k ima svoju poslednju jedinicu na poziciji p (ako ima bitova desno od pozicije p, oni su svi nule). Kada se negiraju bitovi dobija se broj koji ima nulu na poziciji p iza koje do kraja slede jedinice (ako ih uopšte ima). Sabiranjem sa 1 dobija se broj koji na poziciji p ima 1 iza čega se nalaze nule. Dakle,

levo od pozicije p bitovi broja k i $-k$ su suprotni, na poziciji p oba imaju jedinice, a desno od pozicije p oba imaju nule. Bitovska konjunkcija brojeva k i $-k$ daje rezultat $b(k)$ koji ima samo jednu jedinicu i to na poziciji p (to je jedino mesto gde i k i $-k$ imaju vrednost 1). □

Drugi način da se izračuna vrednost $f(k)$ jeste računanjem vrednosti izraza `k & (k-1)`. Ispravnost ovog izraza se takođe može jednostavno dokazati. Naime, broj `k-1` ima sve iste bitove od krajnjeg levog do pozicije poslednjog postavljenog bita u broju k, a sve invertovane bitove posle krajnjeg desnog postavljenog bita u broju k.

Primer 1.4.11

Binarni zapis broja $k = 20$ *je* $(00010100)_2$*, a broja* $k - 1 = 19$ *je* $(00010011)_2$ *i izrazu* `k & (k-1)` *odgovara binarni zapis* $(00010000)_2$.

Ipak, izraz `k - (k & -k)` se češće koristi, zbog sličnosti sa izrazom `k + (k & -k)`, koji se koristi prilikom izračunavanja funkcije g koja je definisana u poglavlju 1.4.2.2 posvećenom ažuriranju vrednosti elemenata niza.

Računanje zbira elemenata segmenta

Zbir elemenata u bilo kom prefiksu polaznog niza može da se dobije kao zbir nekoliko elemenata zapisanih u Fenvikovom drvetu.

Primer 1.4.12

Interval pozicija $(0, 21]$ *tj. prefiks niza do pozicije* 21 *se dobija nadovezivanjem intervala* $(0, 16]$, $(16, 20]$ *i* $(20, 21]$.

Ove intervale je jednostavno odrediti. Na poziciji k se nalazi zbir elemenata sa pozicija iz intervala $(f(k), k]$, na poziciji $f(k)$ zbir elemenata sa pozicija iz intervala $(f(f(k)), f(k)]$, itd. Postupak se nastavlja sabirajući elemente sa pozicija k, $f(k)$, $f(f(k))$, $f(f(f(k)))$ itd., sve dok se ne dođe do pozicije nula.

Broj elemenata Fenvikovog drveta čijim se sabiranjem dobija zbir prefiksa polaznog niza je $O(\log n)$. Naime, u svakom koraku se broj jedinica u binarnom zapisu tekućeg indeksa smanjuje, a broj n se binarno zapisuje sa najviše $O(\log n)$ binarnih jedinica.

Zbir elemenata prefiksa na pozicijama iz intervala $(0, k]$ polaznog niza a smeštenog u Fenvikovo drvo `drvo` možemo onda izračunati narednom funkcijom.

```
// na osnovu Fenvikovog drveta smeštenog u niz drvo
// izračunava zbir prefiksa (0, k] polaznog niza
int zbirPrefiksa(const vector<int>& drvo, int k) {
  int zbir = 0;
  while (k > 0) {
     zbir += drvo[k];
     k -= k & -k; // k = f(k)
  }
}
```

Kada umemo da izračunamo zbirove prefiksa polaznog niza, zbir proizvoljnog segmenta određenog pozicijama $[a, b]$ polaznog niza možemo izračunati kao razliku zbira prefiksa određenog pozicijama $(0, b]$ i zbira prefiksa određenog pozicijama $(0, a-1]$. Pošto se oba prefiksa računaju u vremenu $O(\log n)$, i zbir svakog

segmenta možemo izračunati u vremenu $O(\log n)$. Istaknimo na ovom mestu da je zbog ove operacije važno da asocijativna operacija koja se koristi u Fenvikovom drvetu ima inverznu operaciju (u ovom slučaju, pošto se radi o operaciji sabiranja, da bismo mogli da oduzimanjem dve vrednosti prefiksa dobijemo zbir proizvoljnog segmenta[7]).

Ažuriranje vrednosti elementa

Ideju računanja zbira elemenata proizvoljnog segmenta kao razlike dva zbira prefiksa smo videli već ranije, kada smo u te svrhe održavali niz svih zbirova prefiksa. Osnovna prednost Fenvikovih drveta u odnosu na niz svih zbirova prefiksa je to što se mogu efikasno ažurirati. Naime, ažuriranje jednog elementa polaznog niza može rezultovati izmenom vrednosti velikog broja prefiksnih zbirova, naročito ako je element koji menjamo blizu početka niza (u najgorem slučaju broj zbirova koje je potrebno ažurirati je $O(n)$). Stoga je operacija ažuriranja elementa kada se koriste prefiksni zbirovi u najgorem slučaju složenosti $O(n)$.

Razmotrimo funkciju koja ažurira Fenvikovo drvo nakon uvećanja elementa u polaznom nizu na poziciji k za vrednost x. Tada je za x potrebno uvećati sve one zbirove u drvetu u kojima se kao sabirak javlja i element na poziciji k. Potrebno je ažurirati sve one pozicije m čiji pridruženi segment sadrži vrednost k, tj. sve one pozicije m takve da je $k \in (f(m), m]$, tj. pozicije m za koje važi

$$f(m) < k \leq m \tag{1.1}$$

Te pozicije se izračunavaju veoma jednostavno krenuvši od polazne pozicije k, slično kao u prethodnoj funkciji, s tim što se umesto da se od broja k oduzima vrednost $b(k)$, broj k uvećava za vrednost izraza $b(k)$. Obeležimo sa $g(k)$ broj $k + b(k)$, koji se dobija od broja k tako što se k sabere sa brojem koji ima samo jednu jedinicu u svom binarnom zapisu i to na poziciji na kojoj se nalazi poslednja jedinica u binarnom zapisu broja k. U implementaciji se broj $g(k)$ lako može izračunati po formuli `k + (k & -k)`.

Primer 1.4.13

Za broj $k = (101100)_2$ *važi* $g(k) = (101100)_2 + (100)_2 = (110000)_2$.

Dokazaćemo da je potrebno i dovoljno ažurirati vrednosti na pozicijama k, $g(k)$, $g(g(k))$ itd., sve dok se ne dođe do pozicije koja je strogo veća od dužine niza n.

Primer 1.4.14

Ako bi se u prethodnom primeru element na poziciji 3 *u polaznom nizu uvećao za vrednost* 4*, bilo bi potrebno povećati za* 4 *vrednosti elemenata Fenvikovog drveta na pozicijama* 3*,* 4 *i* 8*. Do niza ovih pozicija bismo mogli da dođemo na sledeći način.*

- *Prva pozicija je broj* $k = 3$.
- *Narednu poziciju* $g(3)$ *dobijamo tako što binarnom zapisu broja* 3 *koji iznosi* $(0011)_2$*, dodajemo broj* 1 *(broj* $b(3)$ *koji sadrži tačno jednu jedinicu na poziciji poslednje jedinice u binarnom zapisu datog broja) i dobijamo binarni zapis* $g(3) = (0100)_2$ *koji odgovara broju* 4.
- *Narednu poziciju* $g(g(3)) = g(4)$ *bismo dobili tako što bismo vrednost 4 sabrali sa* $(0100)_2$ *(brojem* $b(4)$ *koji sadrži tačno jednu jedinicu u svom binarnom zapisu i to na poziciji poslednje jedinice), čime bismo dobili* $g(4) = (1000)_2$ *(binarni zapis broja* 8*). Ovde se procedura završava pošto smo stigli do poslednjeg elementa u Fenvikovom drvetu.*

[7] Obratimo pažnju da ako se Fenvikovim drvetom računaju proizvodi segmenata, elementi drveta moraju biti različiti od nule, zbog operacije deljenja.

S obzirom na to da se broj n binarno zapisuje sa $O(\log n)$ bitova i da se u svakom koraku broj krajnjih nula u zapisu broja povećava (iako u svakom koraku vrednost broja raste), ukupan broj koraka je jednak $O(\log n)$. Dakle, broj elemenata Fenvikovog drveta čije je vrednosti potrebno izmeniti iznosi $O(\log n)$.

```
// ažurira Fenvikovo drvo smešteno u niz drvo nakon što se
// u originalnom nizu element na poziciji k uveća za x
void uvecaj(vector<int>& drvo, int n, int k, int x) {
  while (k <= n) {
    drvo[k] += x;
    k += k & -k;
  }
}
```

Dokažimo korektnost opisanog postupka. Krenimo od naredne leme.

Lema 1.4.3

Za svako x između 1 i n, najmanji broj m koji zadovoljava uslov $f(m) < x < m$ je $g(x)$.

Dokaz. Pokažimo najpre da $g(x)$ zadovoljava ovaj uslov. Očigledno važi $x < g(x)$. Pošto $g(x)$ ima sve nule od pozicije poslednje jedinice u binarnom zapisu broja x (uključujući i nju), pa do kraja, brisanjem njegove poslednje jedinice tj. izračunavanjem vrednosti $f(g(x))$ se sigurno dobija broj koji je strogo manji od x.

Pokažimo sada da je $g(x)$ najmanji broj strogo veći od x koji zadovoljava dati uslov, odnosno da nijedan broj m između x i $g(x)$ ne može da zadovolji uslov $f(m) < x$. Naime, svi brojevi iz intervala $(x, g(x))$ se poklapaju sa brojem x na svim pozicijama pre krajnjih nula, a na pozicijama krajnjih nula broja x imaju bar neku jedinicu, čijim se brisanjem dobija broj koji je veći ili jednak x. □

Primer 1.4.15

U prethodno razmatranom primeru za $k = (101100)_2$, kao što smo već videli važi $g(k) = (110000)_2$ i jedini brojevi između k i $g(k)$ su $(101101)_2$, $(101110)_2$ i $(101111)_2$. Brisanjem poslednje jedinice u binarnom zapisu ovih brojeva dobijaju se redom brojevi $(101100)_2$, $(101100)_2$ i $(101110)_2$ i svi oni su veći ili jednaki od k.

Dokažimo i drugu pomoćnu lemu.

Lema 1.4.4

Za svako $1 \leq x \leq n$ važi $f(g(x)) \leq f(x)$.

Dokaz. Važi da je $f(g(x)) = g(x) - b(g(x)) = x + b(x) - b(g(x))$ i $f(x) = x - b(x)$. Zato je dovoljno dokazati da je $x + b(x) - b(g(x)) \leq x - b(x)$ tj. da je $2b(x) \leq b(g(x))$. Neka $b(x)$ ima jedinicu na poziciji p. Broj $2b(x)$ ima jedinicu na poziciji $p + 1$ (brojano zdesna). Broj $g(x)$ sigurno ima sve nule od pozicije p do kraja, pa se njegova poslednja jedinica nalazi ili na poziciji $p + 1$ ili levo od nje, na osnovu čega lako sledi da je on veći ili jednak od $2b(x)$, koji ima jedinicu na poziciji $p + 1$. Ovim je tvrđenje dokazano. □

Dokažimo sada i glavno tvrđenje.

Teorema 1.4.1

Niz vrednosti k, $g(k)$, $g(g(k))$, itd. koje su manje ili jednake od dužine niza n određuje sve vrednosti m takve da važi $k \in (f(m), m]$ tj. takve da važi jednakost (1.1).

Dokaz. Jednakost (1.1) ne može da važi za brojeve $m < k$, a sigurno važi za broj $m = k$, jer je $f(k) < k$ kada je $k > 0$ (a mi pretpostavljamo da je $1 \leq k \leq n$). Dakle prva pozicija u drvetu koju treba ažurirati je pozicija k.

Za sve brojeve $m > k$, sigurno važi desna nejednakost i jedino je potrebno utvrditi da li važi leva tj. da li je $f(m) < k$. Na osnovu primene leme 1.4.3 na broj k znamo da je $g(k)$ najmanji broj m strogo veći od k za koji važi $f(m) < k$.

Dalje, na osnovu primene leme 1.4.3 na broj $g(k)$ znamo da je najmanji broj veći od $g(k)$ za koji važi $f(m) < g(k)$ broj $g(g(k))$. On zadovoljava uslov (1.1). Zaista, važi $k < g(k) < g(g(k))$. Na osnovu leme 1.4.4 primenjene na $g(k)$ važi je $f(g(g(k))) \leq f(g(k)) < k$. Broj $g(g(k))$ je i najmanji broj koji veći od $g(k)$ koji zadovoljava uslov (1.1). Ako bi neki broj m između $g(k)$ i $g(g(k))$ zadovoljio uslov (1.1), važilo bi da je $f(m) < k < g(k)$, što je u suprotnosti sa time da je $g(g(k))$ najmanji broj m veći od $g(k)$ takav da je $f(m) < g(k)$.

Dokaz se dalje nastavlja i završava po potpuno istom principu (što se može formalizovati indukcijom). □

Prethodna teorema garantuje da su jedine pozicije koje treba ažurirati u drvetu prilikom ažuriranja elementa na poziciji k u polaznom nizu upravo pozicije iz serije k, $g(k)$, $g(g(k))$, itd., sve dok su one manje ili jednake n, pa su prethodni postupak i njegova implementacija korektni.

Formiranje Fenvikovog drveta

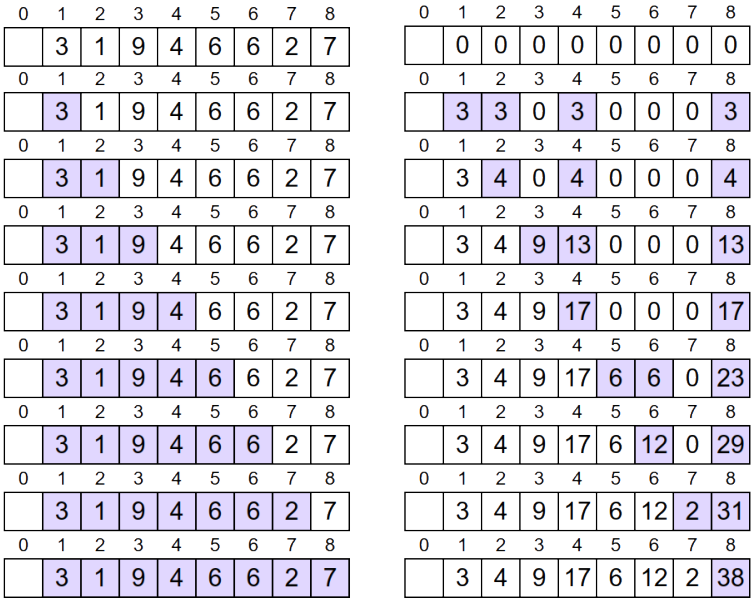
Ostaje još pitanje kako inicijalno formirati Fenvikovo drvo. Formiranje se može svesti na to da se kreira drvo popunjeno samo nulama, a da se zatim uvećava vrednost jednog po jednog elementa niza prethodnom funkcijom.

```
// na osnovu niza a u kom su elementi smešteni
// na pozicijama iz segmenta [1, n] formira Fenvikovo drvo
// i smešta ga u niz drvo (na pozicije iz segmenta [1, n])
vector<int> formirajDrvo(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  fill_n(drvo + 1, n, 0);
  for (int k = 1; k <= n; k++)
     uvecaj(drvo, n, k, a[k]);
  return drvo;
}
```

Primer 1.4.16

Razmotrimo problem formiranja Fenvikovog drveta za niz vrednosti $3, 1, 9, 4, 6, 6, 2, 7$. Krenuvši od niza koji sadrži samo nule, uvećavamo jedan po jedan element niza.

Ova implementacija n puta poziva funkciju `uvecaj` koja je složenosti $O(\log n)$, te je ukupna složenost ovog algoritma $O(n \log n)$. Međutim, pokazuje se da možemo i bolje od ovog. Naime, svaki element Fenvikovog drveta sadrži zbir elemenata nekog segmenta polaznog niza, pa za izračunavanje elemenata Fenvikovog drveta možemo iskoristiti zbirove prefiksa. Naime, element na poziciji k u Fenvikovom drvetu dobijamo kao razliku zbira prefiksa $(0, k]$ i zbira prefiksa $(0, f(k)]$, ako je $f(k) > 0$, dok ako je $f(k) = 0$ element na poziciji k biće jednak baš zbiru prefiksa $(0, k]$.



Slika 1.25: Formiranje Fenvikovog drveta. Originalni niz je predstavljen levo, a Fenvikovo drvo desno.

```
vector<int> formirajDrvoPrefiksneSume(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  vector<int> sume_prefiksa(n+1,0);
  sume_prefiksa[1] = a[1];
  for (int k = 2; k <= n; k++)
    sume_prefiksa[k] = sume_prefiksa[k-1] + a[k];

  fill_n(drvo + 1, n, 0);
  for (int k = 1; k <= n; k++) {
    // racunamo f(k)
    int f_k = k - (k & -k);
    // od sume prefiksa do k oduzimamo sumu prefiksa do f(k)
    // ako je f(k)=0, onda ne vrsimo oduzimanje
    drvo[k] = sume_prefiksa[k] - (f_k > 0 ? sume_prefiksa[f_k] : 0);
  }
  return drvo;
}
```

Izračunavanje zbirova svih prefiksa polaznog niza je vremenske složenosti $O(n)$, ali se nakon toga elementi Fenvikovog drveta računaju u vremenu $O(1)$, te je ukupna vremenska složenost ovog pristupa $O(n)$. Ova implementacija pak zahteva dodatni memorijski prostor veličine $O(n)$ za smeštanje prefiksnih zbirova.

Primetimo da su za izračunavanje k-tog elementa Fenvikovog drveta potrebni samo elementi niza prefiksnih zbirova na pozicijama $j \leq k$. Ovo opažanje omogućava da koristimo samo jedan niz koji ćemo inicijalizovati na niz prefiksnih zbirova, a zatim počev od poslednjeg elementa niza idući ka prvom elementu menjati element po element sa prefiksnog zbira na element Fenvikovog drveta.

```
vector<int> formirajDrvoPrefiksneSumeOpt(int n, const vector<int>& a) {
  vector<int> drvo(n+1);
  fill_n(drvo+1, n, 0);
```

```
  drvo[1] = a[1];
  for (int k = 2; k <= n; k++)
    drvo[k] = drvo[k-1] + a[k];

  for (int k = n; k >= 1; k--){
    int f_k = k - (k & -k);
    if (f_k > 0) drvo[k] -= drvo[f_k];
  }
  return drvo;
}
```

Ova implementacija algoritma za formiranje Fenvikovog drveta je vremenske složenosti $O(n)$, a dodatne prostorne složenosti $O(1)$.

Zadatak: Maksimalni podsegment

Dat je niz celih brojeva. Napisati program koji omogućava izvršavanje sledeće dve vrste upita:

- Izračunavanje maksimalnog zbira nekog podsegmenta datog segmenta određenog pozicijama $[a, b]$ (podsegment je ili prazan ili je određen pozicijama $[a', b']$ takvim da je $a \leq a' \leq b' \leq b$).
- Promena vrednosti elementa niza na nekoj datoj poziciji.

Opis ulaza

Sa standardnog ulaza se učitava vrednost n ($1 \leq n \leq 10^5$), a zatim n celih brojeva iz intervala $[-100, 100]$, koji predstavljaju početne vrednosti niza. Nakon toga se učitava prirodan broj q ($1 \leq q \leq 10^5$) koji predstavlja broj upita koje je potrebno obraditi. Nakon toga se učitava q upita. Upiti mogu biti sledećeg oblika:

- `s a b` – potrebno je odrediti maksimalnu vrednost zbira nekog podsegmenta segmenta određenog pozicijama $[a, b]$, za $0 \leq a \leq b < n$.
- `p k v` – potrebno je elementu niza na poziciji k dodeliti vrednost v, za $0 \leq k < n$ i ceo broj v između -100 i 100.

Opis izlaza

Za svaki upit tipa `s` na standardni izlaz ispisati maksimalnu vrednost zbira datog podsegmenta.

Primer

Ulaz

```
9
-2 1 -3 4 -1 2 1 -5 4
5
s 0 8
s 1 4
p 4 1
s 0 8
s 2 6
```

Izlaz

```
6
4
8
8
```

Rešenje

Kadanov algoritam

Rešenje grubom silom podrazumeva da za svaki segment iz početka računamo vrednost maksimalnog zbira nekog njegovog podsegmenta. Čak i kada se za to koristi neki efikasan algoritam (na primer Kadanov), ovo rešenje je veoma neefikasno. Ažuriranje vrednosti niza vrši se u vremenu $O(1)$, ali se maksimalni podsegment određuje u vremenu $O(n)$, pa je složenost najgoreg slučaja $O(qn)$.

Segmentno drvo

Efikasno rešenje se može dobiti pomoću segmentnog drveta. Ako se neki segment podeli na dva manja podsegmenta, njegov maksimalni podsegment može biti ceo sadržan u levom podsegmentu, ceo sadržan u desnom podsegmentu ili se sastojati od maksimalnog sufiksa levog i maksimalnog prefiksa desnog podsegmenta. Maksimalni prefiks celog segmenta je ili maksimalni prefiks njegovog levog podsegmenta ili sadrži ceo levi podsegment i maksimalni prefiks desnog podsegmenta. Slično, maksimalni sufiks celog segmenta je ili maksimalni sufiks njegovog desnog podsegmenta ili sadrži ceo desni podsegment i maksimalni sufiks levog podsegmenta. Zato segmentno drvo organizujemo tako da svaki čvor sadrži naredna 4 podatka:

- zbir svih elemenata segmenta koji odgovara tom čvoru;
- maksimalni zbir nekog podsegmenta segmenta koji odgovara tom čvoru;
- maksimalni zbir prefiksa segmenta koji odgovara tom čvoru;
- maksimalni zbir sufiksa segmenta koji odgovara tom čvoru.

Formiranje segmentnog drveta vrši se u složenosti $O(n)$. Određivanje maksimalnog zbira podsegmenta za dati segment zahteva jedan prolaz kroz drvo i složenosti je $O(\log n)$. Takođe, i ažuriranje elementa zahteva jedan prolaz kroz drvo (odozdo naviše) i složenosti je $O(\log n)$. Dakle, složenost ovog pristupa je $O(n + q \log n)$.

```
// cvor segmentnog drveta
typedef struct {
  int zbir;        // zbir segmenta
  int maksSegment; // maksimalni zbir podsegmenta
  int maksPrefiks; // maksimalni zbir prefiksa
  int maksSufiks;  // maksimalni zbir sufiksa
} Cvor;

// sve 4 vrednosti su inicijalizovane na nulu
Cvor nula = {};

// odredjuje cvor roditelja na osnovu poznatog levog i desnog deteta
Cvor kombinuj(const Cvor& l, const Cvor& d) {
  Cvor c;
  c.zbir = l.zbir + d.zbir;
  c.maksSegment = max({l.maksSegment, d.maksSegment, l.maksSufiks + d.maksPrefiks});
  c.maksPrefiks = max({l.maksPrefiks, l.zbir + d.maksPrefiks});
  c.maksSufiks  = max({d.maksSufiks, d.zbir + l.maksSufiks});
  return c;
}

// kreira se list drveta na osnovu date vrednosti v
Cvor list(int v) {
  Cvor c;
  c.zbir = v;
  c.maksSegment = max(v, 0);
```

```
  c.maksPrefiks = max(v, 0);
  c.maksSufiks = max(v, 0);
  return c;
}

// formira segmentno drvo na osnovu datog niza
vector<Cvor> formirajDrvo(const vector<int>& a) {
  int n = stepenDvojke(a.size());
  vector<Cvor> drvo(2*n, nula);
  for (int k = 0; k < a.size(); k++)
    drvo[n + k] = list(a[k]);

  for (int k = n-1; k > 0; k--)
    drvo[k] = kombinuj(drvo[2*k], drvo[2*k+1]);
  return drvo;
}

// rekurzivna funkcija koja odredjuje doprinos cvora k u segmentnom drvetu (koji pokriva
// segment [x, y] u originalnom nizu) maksimalnom zbiru podsegmenta segmenta [a, b]
Cvor maksPodsegment(const vector<Cvor>& drvo, int k, int x, int y, int a, int b) {
  // segmenti [x, y] i [a, b] su disjunktni pa ne doprinosi nista
  if (b < x || a > y)
    return nula;
  // segment [x, y] je ceo sadrzan unutar [a, b] pa se ceo uracunava
  if (a <= x && y <= b)
    return drvo[k];
  // segmenti [x, y] i [a, b] se preklapaju, pa se segment [x, y] deli na dve polovine
  int s = (x + y) / 2;
  return kombinuj(maksPodsegment(drvo, 2*k, x, s, a, b),
                  maksPodsegment(drvo, 2*k+1, s+1, y, a, b));
}

// na osnovu segmentnog drveta odre]uje se maksimalni z
int maksPodsegment(const vector<Cvor>& drvo, int a, int b) {
  // pozivamo pomocnu rekurzivnu funkciju krenuvsi od korena drveta
  // koji se nalazi u cvoru 1 i pokriva ceo niz tj. segment [0, n-1]
  int n = drvo.size() / 2;
  Cvor c = maksPodsegment(drvo, 1, 0, n-1, a, b);
  return c.maksSegment;
}

// nakon upisivanja vrednosti v na poziciju k u originalnom nizu
// azurira se segmentno drvo
void azurirajDrvo(vector<Cvor>& drvo, int k, int v) {
  int n = drvo.size() / 2;
  // azuriramo list
  drvo[n+k] = list(v);
  // azuiramo sve njegove pretke
  int r = (n+k) / 2;
  while (r > 0) {
    drvo[r] = kombinuj(drvo[2*r], drvo[2*r+1]);
```

```
    r = r / 2;
  }
}
```

Zadatak: Broj inverzija

Napiši program koji određuje koliko u nizu ima inverzija (pozicija $0 \leq i < j < n$, takvih da je $a_i > a_j$.

Opis ulaza

Sa standardnog ulaza se unosi broj n ($1 \leq n \leq 10^5$) i zatim n celih brojeva, svaki u posebnom redu.

Opis izlaza

Na standardni izlaz ispisati samo traženi broj inverzija.

Primer

Ulaz

```
5
3 1 4 2 5
```

Izlaz

```
3
```

Rešenje

Fenvikovo drvo

Jedan način da se zadatak efikasno reši je da održavamo strukturu podataka u koju se efikasno može umetnuti novi element i kojoj se efikasno može dobiti odgovor na pitanjem koliko je elemenata u strukturi strogo manje od date vrednosti. Ako imamo ovakvu strukturu podataka na raspolaganju, inverzije možemo prebrojati na sledeći način. Niz u kome brojimo inverzije obilazimo sa desnog kraja, dodajući u strukturu jedan po jedan element, nakon što ga obradimo. To znači da će u trenutku obrade bilo kog elementa struktura sadržati tačno one elemente koji su u originalnom nizu desno od njega. Za svaki element proveravamo koliko je elemenata u strukturi manje od njega, uvećavamo brojač inverzija za taj broj i nakon toga umećemo element u drvo.

Jedna takva struktura je Fenvikovo drvo nad nizom koji na poziciji x čuva broj pojavljivanja vrednosti x u strukturi. Brojanje elemenata manjih od x se onda svodi na izračunavanje prefiksne sume do $x - 1$, a dodavanje elementa x se svodi na uvećavanje vrednosti na poziciji x za 1 (i ažuriranje odgovarajućih zbirova u drvetu). Napomenimo da se u opisanom algoritmu niz obilaz zdesna nalevo, jer nam Fenvikovo drvo jednostavnije daje odgovor na to koliko je elemenata manje od date vrednosti, nego koliko je elemenata veće od date vrednosti.

Pošto veličina Fenvikovog drveta (pa i memorijska, ali i vremenska složenost operacija) zavisi od vrednosti najvećeg elementa u njemu, a nama za broj inverzija nisu važne apsolutne vrednosti elemenata, nego samo njihov međusobni odnos, pre primene algoritma možemo svaki element zameniti sa njegovim rangom u sortiranom nizu (isti elementi mogu da imaju isti rang). Ovo je moguće uraditi, na primer, tako što sortiramo niz, a zatim binarnom pretragom za svaki element pronađemo prvu poziciju njegovog pojavljivanja u sortiranom nizu.

```
long long brojInverzija(vector<int>& a) {
  int n = a.size();
  // svaki element u nizu a menjamo rangom tj. prvom pozicijom na
  // kojoj se javlja u sortiranom redosledu (brojimo pozicije od 1)
```

```
  vector<int> b = a;
  sort(begin(b), end(b));
  for (int i = 0; i < n; i++)
    a[i] = distance(begin(b), lower_bound(begin(b), end(b), a[i])) + 1;

  // Fenvikovo drvo
  vector<int> drvo(n+1, 0);
  // broj inverzija
  long long broj = 0;
  for (int i = n-1; i >= 0; i--) {
    // odredjujemo koliko elemenata u drvetu je strogo manje od a[i]
    broj += zbirPrefiksa(drvo, a[i]-1);
    // dodajemo a[i] u drvo
    dodaj(drvo, a[i], 1);
  }
  return broj;
}
```

Zadatak: K-ti parni broj

Napisati program koji omogućava da se u nizu prirodnih brojeva koji je na početku ispunjen nulama, ali čiji se elementi često menjaju tokom izvršavanja programa efikasno pronalazi pozicija k-tog parnog broja po redu.

Opis ulaza

Sa standardnog ulaza se učitava dužina niza n ($1 \leq n \leq 50000$), a zatim i broj upita m ($1 \leq m \leq 50000$). Izvršavanjem upita oblika $u\ p\ x$ se u niz na poziciju $1 \leq p \leq n$ upisuje broj x, dok se upitom $c\ k$ na standardni izlaz ispisuje pozicija k-tog parnog broja u tekućem sadržaju niza (pozicije se broje od 1).

Opis izlaza

Na standardnom izlazu prikazati rezultate izvršavanja upita `c`. Ako u nekom slučaju u nizu ima manje parnih brojeva od vrednosti k, tada umesto pozicije ispisati `-`.

Primer

Ulaz

```
5
8
u 3 1
c 4
u 1 7
u 2 5
c 1
u 1 2
c 2
c 4
```

Izlaz

```
5
4
4
-
```

Objašnjenje

- Niz na početku sadrži 5 nula
- Nakon izvršavanje upita `u 3 1` niz sadrži elemente `0 0 1 0 0`.
- Upitom `c 4` se izračunava pozicija četvrtog parnog broja u nizu i to je pozicija 5.
- Nakon izvršavanja upita `u 1 7`, pa zatim i `u 2 5` sadržaj niza je `7 5 1 0 0`.
- Upitom `c 1` se izračunava pozicija prvog parnog broja u nizu i to je pozicija 4.
- Nakon izvršavanja upita `u 1 2` sadržaj niza je `2 5 1 0 0`.
- Upitom `c 2` se izračunava pozicija drugog parnog broja u nizu i to je pozicija 4.
- Upitom `c 4` se izračunava pozicija četvrtog parnog broja u nizu, međutim on ne postoji (niz sadrži 3 parna broja: 2, 0 i 0).

Rešenje

Direktno rešenje podrazumeva da se elementi upisuju u niz i da se pozicija k-tog parnog određuje primenom linearne pretrage. Ako imamo m_1 operacija ažuriranja i m_2 operacija pretrage složenost takvog rešenja je $O(m_1 + m_2 \cdot n)$, što može biti prilično neefikasno.

Na osnovu niza možemo formirati niz nula i jedinica takav da se jedinice nalaze na mestu parnih elemenata. Tada je pozicija k-tog po redu parnog broja najmanja pozicija takva da je zbir svih jedinica zaključno sa tom pozicijom jednak k. Ako niz nula i jedinica održavamo u Fenvikovom drvetu, vrlo efikasno možemo da izračunavamo zbirove svakog fiksiranog prefiksa. Zahvaljujući činjenici da su zbirovi prefiksa monotono neopadajući (kada se prefiksi produžavaju), traženu poziciju možemo efikasno odrediti algoritmom binarne pretrage. Ako imamo m_1 operacija ažuriranja i m_2 operacija pretrage ukupna složenost će biti $O(m_1 \log n + m_2 \log^2 n)$, tj. $O(m \log^2 n)$.

```
// vraca prvu poziciju p tako da je zbir niza na pozicijama [1, p]
// veci ili jednak k
int prefiksK(const vector<int>& drvo, int k) {
  // poziciju pronalazimo binarnom pretragom po vrednosti zbira prefiksa
  int l = 1, d = drvo.size() - 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (zbirPrefiksa(drvo, s) < k)
      l = s + 1;
    else
      d = s - 1;
  }
  return l;
}

int main() {
  ios_base::sync_with_stdio(false); cin.tie(0);
  int n;
  cin >> n;
  // gradimo Fenvikovo drvo i inicijalizujemo ga jedinicama
  vector<int> drvo(n + 1, 0);
  for (int k = 1; k <= n; k++)
    dodaj(drvo, k, 1);
  // elementi niza
  vector<int> niz(n + 1, 0);

  // broj upita
```

```
  int m;
  cin >> m;
  for (int i = 0; i < m; i++) {
    char c;
    cin >> c;
    if (c == 'u') {
      // upit upisa elementa u niz
      int p, x;
      cin >> p >> x;
      if (x % 2 == 0 && niz[p] % 2 != 0)
        // upisan je novi paran element na poziciju p
        dodaj(drvo, p, 1);
      else if (x % 2 != 0 && niz[p] % 2 == 0)
        // upisan je novi neparan element na poziciju p
        dodaj(drvo, p, -1);
      niz[p] = x;
    } else if (c == 'c') {
      // upit odredjivanja k-tog parnog elementa
      int k;
      cin >> k;
      // trazimo najkraci prefiks ciji je zbir jednak k
      int p = prefiksK(drvo, k);
      // proveravamo da li takav prefiks zaista postoji
      if (p <= n)
        cout << p << "\n";
      else
        cout << "-" << "\n";
    }
  }
  return 0;
}
```

Umesto Fenvikovog, možemo koristiti i segmentno drvo.

Zadatak: Broj različitih elemenata u segmentima

Napiši program koji određuje broj različitih elemenata u svakom od m datih segmenata niza od n elemenata.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 50000$), a zatim n celih brojeva $a_1, \ldots, a_n$, čije su vrednosti između 1 i 100000. Nakon toga se učitava broj m ($1 \leq m \leq 50000$) a zatim u narednih m redova leva i desna granica zatvorenog segmenta $[l_i, d_i]$, razdvojene sa po jednim razmakom ($1 \leq l_i \leq d_i \leq n$).

Opis izlaza

Na standardni izlaz ispisati m brojeva od kojih svaki predstavlja broj različitih elemenata u segmentu $a_{l_i}, \ldots, a_{d_i}$.

Primer

Ulaz

```
5
1 1 2 1 3
```

```
3
1 5
2 4
3 5
```

Izlaz

```
3
2
3
```

Rešenje

Rešenje grubom silom podrazumeva da za svaki segment izbrojimo različite elemente (na primer, korišćenjem strukture podataka za čuvanje skupova). Ono je prilično neefikasno.

Osnovna ideja efikasnog rešenja zadatka je da je broj različitih elemenata niza jednak broju poslednjih pojavljivanja tih elemenata. Možemo kreirati binarni niz koji sadrži jedinice na mestima na kojima se element javlja poslednji put i nule na mestima na kojima se nalaze elementi koji se u tom segmentu javljaju i kasnije i broj različitih elemenata tog niza dobiti sabiranjem tog binarnog niza.

Primer 1.4.17

Na primer, ako je dato 10 *elemenata niza* $3, 4, 1, 3, 2, 5, 4, 7, 2, 2$*, tada možemo napraviti binarni niz* $0, 0, 1, 0, 0, 1, 1, 1, 0, 1$*. Zbir njegovih elemenata je 5, što znači da u originalnom nizu postoji 5 različitih elemenata.*

Dodatno, za svaki segment pozicija oblika $[l, n)$, tj. za svaki sufiks niza, broj različitih elemenata u tom sufiksu možemo dobiti izračunavanjem broja jedinica u odgovarajućem sufiksu binarnog niza. Zaista, za svaku grupu jednakih elemenata postoji samo jedna jedinica i svaki element se broji samo jednom. Za svaki element koji pripada sufiksu određenom pozicijama $[l, n)$ postoji bar jedna jedinica u tom segmentu koja mu odgovara (ako je trenutno pojavljivanje elementa poslednje, onda je jedinica na njegovom mestu, a ako nije, onda sigurno postoji jedinica negde iza njega koja mu odgovara). Primetimo da to ne mora da važi za segmente koji se ne završavaju na poslednjoj poziciji u nizu.

Primer 1.4.18

Zbir poslednja tri elementa u binarnom nizu iz prethodnog primera je 2, što znači da u originalnom nizu postoje 2 različita elementa (to su 7 i 2).

Binarni niz možemo kreirati inkrementalno tokom prolaska polaznog niza sleva nadesno. Naime, za svaki novi element polaznog niza na kraj niza dodajemo jedinicu (ovo njegovo pojavljivanje je sigurno poslednje). Dodatno, proveravamo da li se taj element ranije pojavljivao i ako jeste pronlazimo poziciju njegovog ranijeg poslednjeg pojavljivanja i menjamo je u nulu. Pozicije poslednjih pojavljivanja elemenata možemo čuvati u zasebnoj mapi.

Ovo nam ukazuje na to da bi dobro bilo unete segmente sortirati na osnovu desnih krajeva i obrađivati ih u tom redosledu. Niz nula i jedinica možemo čuvati u Fenvikovom ili segmentnom stablu, što nam omogućava da efikasno ažuriramo pojedinačne vrednosti i određujemo zbirove njegovih sufiksa.

Primer 1.4.19

Prikažimo kako se izvršava primer iz postavke zadatka. Upite obrađujemo u sortiranom redosledu na osnovu desnog kraja.

- *Prvo se obrađuje upit* $[2, 4]$. *To znači da se kreira binarni niz zaključno sa pozicijom 4 iz originalnog niza.*
  - *Dodajemo element 1 (sa pozicije 1) i binarni niz je [1].*
  - *Dodajemo element 1 (sa pozicije 2) i binarni niz postaje [0, 1].*
  - *Dodajemo element 2 (sa pozicije 3) i binarni niz postaje [0, 1, 1].*
  - *Dodajemo element 1 (sa pozicije 4) i binarni niz postaje [0, 0, 1, 1].*

  Upit se izvršava sabirajući elemente sa pozicija $[2, 4]$ *u binarnom nizu, čime se dobija rezultat 2.*
- *Sada se obrađuje upit* $[1, 5]$. *Ovo zahteva da se u binarni niz doda i element sa pozicije 5 iz originalnog niza.*
  - *Dodajemo element 3 (sa pozicije 4) i binarni niz postaje [0, 0, 1, 1, 1].*

  Upit se izršava sabirajući elemente sa pozicija $[1, 5]$ *u binarnom nizu, čime se dobija rezultat 3.*
- *Na kraju se obrađuje upit* $[3, 5]$. *Binarni niz je već proširen do pozicije 5.*

  Upit se izršava sabirajući elemente sa pozicija $[3, 5]$ *u binarnom nizu, čime se dobija rezultat 3.*

Na kraju ispisujemo rezultate upita u redosledu u kom su uneti (a ne u redosledu u kom su obrađeni).

Primetimo da smo odgovaranje na upite odložili za kraj programa, što nam je omogućilo da sve upite učitamo i zatim sortiramo. Ova tehnika se nekada naziva *oflajn obrada upita* i iako se ponekada može upotrebiti za efikasno rešenje zadatka, u realnim situacijama ona nije uvek primenjiva (u nekim primenama se odgovor na svaki upit traži odmah čim se upit postavi).

```
int n;
cin >> n;
vector<int> a(n + 1);
for (int i = 1; i <= n; i++)
  cin >> a[i];

int m;
cin >> m;

struct Upit {
  int l, d, i;
};

vector<Upit> upiti(m);
for (int i = 0; i < m; i++) {
  cin >> upiti[i].l >> upiti[i].d;
  upiti[i].i = i;
}

sort(begin(upiti), end(upiti),
     [](const auto& u1, const auto& u2) {
       return u1.d < u2.d;
     });

unordered_map<int, int> prethodna_pozicija;
vector<int> drvo(n + 1, 0);
vector<int> rezultat(m);
```

```
int tekuci_upit = 0;
for (int i = 1; tekuci_upit < upiti.size() && i <= n; i++) {
  auto it = prethodna_pozicija.find(a[i]);
  if (it != prethodna_pozicija.end()) {
    dodaj(drvo, it->second, -1);
    it->second = i;
  } else {
    prethodna_pozicija[a[i]] = i;
  }
  dodaj(drvo, i, 1);
  while (tekuci_upit < upiti.size() && upiti[tekuci_upit].d == i) {
    rezultat[upiti[tekuci_upit].i] =
      zbirSegmenta(drvo, upiti[tekuci_upit].l, i);
    tekuci_upit++;
  }
}

for (int x : rezultat)
  cout << x << '\n';
```

1.4.3 Ažuriranje segmenata

I segmentna i Fenvikova drveta podržavaju efikasno izračunavanje zbirova određenih segmenata niza (engl. range-query), pa samim tim i određivanje pojedinačnih vrednosti niza (engl. point-query), kao i ažuriranje[8] pojedinačnih elemenata niza (engl. point-update), dok ažuriranje celih segmenata niza odjednom (engl. range-update) nije direktno podržano. Naime, ako se ono svede na pojedinačno ažuriranje svih elemenata unutar segmenta, dobija se loša složenost (u najgorem slučaju vrši se n ažuriranja koja imaju pojedinačnu složenost $O(\log n)$, pa je ukupna složenost $O(n \log n)$). Sa druge strane, čuvanje niza razlika omogućava efikasno ažuriranje celih segmenata (engl. range-update), pa samim tim i pojedinačnih elemenata (engl. point-update), ali ne i efikasno određivanje pojedinačnih elemenata niza (engl. point-query) niti zbirova segmenata (engl. range-query).

	point query	*range query*	*point update*	*range update*
Razlike susednih elemenata	loše / $O(n)$	loše / $O(n)$	dobro / $O(1)$	dobro / $O(1)$
Prefiksni zbirovi	dobro / $O(1)$	dobro / $O(1)$	loše / $O(n)$	loše / $O(n)$
Fenvikovo drvo	dobro / $O(\log n)$	dobro / $O(\log n)$	dobro / $O(\log n)$	loše / $O(n \log n)$
Segmentno drvo	dobro / $O(1)$	dobro / $O(\log n)$	dobro / $O(\log n)$	loše / $O(n \log n)$

U nastavku ćemo videti nekoliko načina da se naprave strukture koje efikasno podržavaju sve navedene operacije.

1.4.3.1 Drvo nad nizom razlika

Problem

Definisati strukturu podataka koja obezbeđuje efikasno ažuriranje segmenata datog niza određenih pozicijama $[a, b]$ *(*range update*), pa samim tim i ažuriranje pojedinačnih elemenata niza (*point update*)*

[8] Pod ažuriranjem se podrazumeva ili postavljanje vrednosti elementa na datu vrednost ili uvećanje trenutne vrednosti elementa za datu vrednost. U nastavku teksta ćemo razmatrati samo uvećanje trenutne za datu vrednost.

*uvećavanjem svih elemenata za datu vrednost, kao i efikasno određivanje vrednosti pojedinačnih elemenata niza (*point query*).*

Primetimo da se ne zahteva mogućnost efikasnog određivanja statistika segmenata (*range query*).

Osnovna ideja rešenja je da se održava niz razlika susednih elemenata polaznog niza i da se taj niz razlika čuva u Fenvikovom drvetu (ili, analogno, u segmentnom drvetu).

- Uvećavanje svih elemenata segmenata polaznog niza za neku vrednost v, svodi se na promenu dva elementa niza razlika, što se uz održavanje drveta može uraditi u vremenu $O(\log n)$.
- Rekonstrukcija elementa polaznog niza na osnovu niza razlika svodi na izračunavanje zbira odgovarajućeg prefiksa, što se pomoću Fenvikovog drveta može uraditi veoma efikasno, u vremenu $O(\log n)$.

Na ovaj način, i ažuriranje celih segmenata niza odjednom i očitavanje pojedinačnih elemenata možemo postići algoritmima složenosti $O(\log n)$, što nije bilo moguće samo uz korišćenje niza razlika (uvećavanja svih elemenata nekog segmenta je tada bilo složenosti $O(1)$, ali je očitavanje vrednosti iz niza bilo složenosti $O(n)$). Dakle, ako Fenvikovo ili segmentno drvo izgradimo nad nizom razlika umesto nad originalnim nizom, gubimo mogućnost efikasnog izračunavanja statistika segmenata (*range query*), ali dobijamo mogućnost ažuriranja segmenata (*range update*).

	point query	*range query*	*point update*	*range update*
Drvo razlika	dobro / $O(\log n)$	loše / $O(n \log n)$	dobro / $O(\log n)$	dobro / $O(\log n)$

1.4.3.2 Dva drveta razlika

Na kraju, opisaćemo nekoliko struktura podataka koje omogućavaju efikasno izvršavanje sve četiri vrste upita.

Problem

Definisati strukturu podataka koja obezbeđuje efikasno ažuriranje segmenata datog niza određenih pozicijama $[a, b]$ (samim tim i ažuriranje pojedinačnih elemenata niza) uvećavanjem svih elemenata za datu vrednost, kao i efikasno određivanje zbirova segmenata određenih pozicijama $[a, b]$ (samim tim i određivanje vrednosti pojedinačnih elemenata niza).

Efikasno uvećanje svih elemenata u datom segmentu za istu vrednost i izračunavanje zbirova segmenata moguće je implementirati održavanjem dva Fenvikova ili segmentna drveta.

Prvo od njih će čuvati razlike originalnog niza i omogući će nam da efikasno uvećavamo segmente i izračunavamo pojedinačne elemente originalnog niza.

Ako je originalni niz dužine inicijalizovan nulama, nakon uvećanja svih elemenata sa pozicija iz segmenta $[a, b]$ za vrednost v dobijamo niz:

			a					b			
0	…	0	v	v	…	v	v	0	…	0

koji se može predstaviti nizom razlika:

			a				b	$b+1$			
0	…	0	v	0	…	0	0	$-v$		…	0

Sabiranjem prefiksa niz razlika možemo dobiti bilo koji pojedinačni element početnog niza (ako je niz razlika u Fenvikovom ili segmentnom drvetu, to možemo uraditi u složenosti $O(\log n)$).

Cilj nam je da izračunamo prefiksne zbirove originalnog niza (njega ne možemo čuvati u Fenvikovom ili segmentnom drvetu, zato što nam je potrebno ažuriranje njegovih segmenata, a ne pojedinačnih elemenata).

Prefiksni zbirovi originalnog niza su:

			a				b			
0	…	0	v	$2v$	…	$(b-a)v$	$(b-a+1)v$	$(b-a+1)v$	…	$(b-a+1)v$

- Prefiksni zbirovi P_k na pozicjama k levo od pozicije a (tj. $k < a$) jednaki su nuli (jer su vrednosti A_k originalnog niza levo od te pozicije jednaki nuli).
- Prefiksni zbirovi P_k na pozicijama k između a i b (tj. $a \leq k \leq b$) jednaki su $(k - a + 1) \cdot v$, gde je $v = A_k$ vrednost koja se nalazi na poziciji k u originalnom nizu.
- Prefiksni zbirovi P_k na pozicijama k desno od pozicije b (tj. $k > b$) jednaki su $(b - a + 1) \cdot v$.

Dakle, važi:

$$P_k = \sum_{i=1}^{k} A_i = \begin{cases} 0 & k < a \\ (k - (a-1)) \cdot v & a \leq k \leq b \\ (b - a + 1) \cdot v & k > b \end{cases}$$

Ključna ideja je da uporedimo ove tražene zbirove sa vrednostima niza koji na svakoj poziciji k sadrži vrednost $k \cdot A_k$, tj. da razmotrimo razlike $X_k = kA_k - P_k$.

- U prvom slučaju (kada je $k < a$) je $A_k = P_k = 0$, pa je i $X_k = 0$.
- U drugom slučaju (kada je $a \leq k \leq b$) je $P_k = (k-a+1) \cdot A_k$, pa je $X_k = kA_k - (k-a+1)A_k = (a-1)A_k$.
- U trećem slučaju (kada je $k > b$) je $P_k = (b-a+1) \cdot v$, dok je $A_k = 0$, pa je $X_k = -(b-a+1) \cdot v$.

Drugim rečima, svaki zbir prefiksa P_k smo razložili na razliku $kA_k - X_k$:

$$P_k = \sum_{i=1}^{k} A_i = k \cdot A_k - X_k = \begin{cases} k \cdot 0 - 0 & k < a \\ k \cdot v - (a-1) \cdot v & a \leq k \leq b \\ k \cdot 0 - (-(b-a+1)) \cdot v & k > b \end{cases}$$

Niz X_k, dakle, nakon ažuriranja originalnog niza ima sledeće vrednosti:

			a				b			
0	…	0	$(a-1)v$	$(a-1)v$	…	$(a-1)v$	$(a-1)v$	$-(b-a+1)v$	…	$-(b-a+1)v$

Ako bismo u svakom trenutku poznavali vrednosti u nizu X_k, tada bismo vrednosti P_k lako mogli izračunati pomoću formule $P_k = kA_k - X_k$. Vrednosti A_k lako izračunavamo pomoću Fenvikovog ili segmentnog drveta u kome čuvamo razlike tog niza. Međutim, i niz X_k je takav da mu se pri svakom ažuriranju povezani segmenti uvećavaju tj. umanjuju za istu vrednost, tako da se i on može lako rekonstruisati ako bi se njegove razlike čuvale u drugom Fenvikovom ili segmentnom drvetu. Naime, niz razlika ovog niza se menja ovako:

			a				b	$b+1$			
0	…	0	$(a-1)v$	0	…	0	0	$-bv$	0	…	0

Dakle, održavamo dva drveta: D_A u kome čuvamo razlike niza A i D_X u kome čuvamo razlike niza X. Prvo inicijalizujemo razlikama niza A, a drugo nulama.

Operacija uvećanja svih elemenata sa pozicija iz $[a, b]$ za vrednost v se svodi na sledeće operacije nad drvetima:

- *uvecaj*(D_A, a, v)
- ako je $b < n$ onda *uvecaj*$(D_A, b + 1, -v)$
- *uvecaj*$(D_X, a, (a - 1)v)$
- ako je $b < n$ onda *uvecaj*$(D_X, b + 1, -bv)$

Operacija izračunavanje vrednosti niza A na poziciji k tj. vrednosti A_k se svodi na izračunavanje prefiksne sume prvih k elemenata Fenvikovog drveta D_A.

- $A_k =$ *zbir_prefiksa*(D_A, k)

Operacija izračunavanja zbira prefiksa elemenata niza A dužine k se svodi na izračunavanje vrednosti

- $P_k = kA_k - X_k = k \cdot$ *zbir_prefiksa*$(D_A, k) -$ *zbir_prefiksa*(D_X, k)

Operacija izračunavanja zira segmenta sa pozicija $[a, b]$ niza A se svodi na izračunavanje vrednosti:

- $S_{ab} = P_b - P_{a-1}$

Pri tom je $P_0 = 0$.

Dakle, na ovaj način možemo postići dobru složenost za sva 4 tipa operacija.

	point query	*range query*	*point update*	*range update*
Dva drveta razlika	dobro / $O(\log n)$	dobro / $O(\log n)$	dobro / $O(\log n)$	dobro / $O(\log n)$

Primer 1.4.20

Prikažimo jedan primer upotrebe ove strukture podataka. Pretpostavimo da niz A ima 10 elemenata i da su u početku svi jednaki nuli. U programu bi se održavala samo drveta nad nizovima D_A i D_X, a ilustracije radi mi ćemo prikazati i sadržaj niza A, njegovih prefiksnih suma P i pomoćnog niza X. Elementi originalnog niza (kao i nizova razlika) smešteni su na pozicijama od 1 do 10.

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	0	0	0	0	0	0	0	0	0	0
D_X	-	0	0	0	0	0	0	0	0	0	0
A	-	0	0	0	0	0	0	0	0	0	0
kA_k	-	0	0	0	0	0	0	0	0	0	0
X	-	0	0	0	0	0	0	0	0	0	0
P	0	0	0	0	0	0	0	0	0	0	0

Uvećajmo element A_5 za 4. To se može svesti na uvećavanje elemenata segmenta $[a, b] = [5, 5]$ za $v = 4$.

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	0	0	0	0	4	-4	0	0	0	0
D_X	-	0	0	0	0	16	-20	0	0	0	0
A	-	0	0	0	0	4	0	0	0	0	0

	0	1	2	3	4	5	6	7	8	9	10
kA_k	-	0	0	0	0	20	0	0	0	0	0
X	-	0	0	0	0	16	-4	-4	-4	-4	-4
P	0	0	0	0	0	4	4	4	4	4	4

Umanjimo sve elemente niza A za 1. To se može svesti na uvećavanje elemenata segmenta $[a, b] = [1, 10]$ za $v = -1$.

	0	1	2	3	4	5	6	7	8	9	10
D_A	-	-1	0	0	0	4	-4	0	0	0	0
D_X	-	0	0	0	0	16	-20	0	0	0	0
A	-	-1	-1	-1	-1	3	-1	-1	-1	-1	-1
kA_k	-	-1	-2	-3	-4	15	-6	-7	-8	-9	-10
X	-	0	0	0	0	16	-4	-4	-4	-4	-4
P	0	-1	-2	-3	-4	-1	-2	-3	-4	-5	-6

Odredimo $S_{[3,7]}$ tj. zbir elemenata niza u segmentu $[3, 7]$. Važi da je $S_{[3,7]} = P_7 - P_2$, pa je potrebno da odredimo P_7 i P_2.

P_7 se određuje tako što se pomoću Fenvikovog drveta efikasno odredi $A_7 = zbir\_prefiksa(D_A, 7) = -1$, zatim $X_7 = zbir\_prefiksa(D_X, 7) = -4$ i na kraju se izračuna $7A_7 - X_7 = 7 \cdot (-1) - (-4) = -3$.

P_2 se određuje tako što se pomoću Fenvikovog drveta efikasno odredi $A_2 = zbir\_prefiksa(D_A, 2) = -1$, zatim $X_2 = zbir\_prefiksa(D_X, 2) = 0$ i na kraju se izračuna $2A_2 - X_2 = 2 \cdot (-1) - 0 = -2$. Traženi zbir segmenta je onda $-3 - (-2) = -1$. Jasno je da je to tačan rezultat, jer se u tom segmentu nalaze elementi $[-1, -1, 3, -1, -1]$.

1.4.3.3 Lenjo ažuriranje segmentnog drveta

Opišimo sada još jedno rešenje prethodnog problema, tj. još jednu strukturu podataka koja omogućava efikasno izvršavanje sve četiri vrste upita. Za razliku od prethodne, ova struktura omogućava i izračunavanje nekih drugih statistika (ne samo zbira elemenata).

Podsetimo se, segmentna drveta omogućavaju efikasno ažuriranje pojedinačnih elemenata i izračunavanje zbirova segmenata (samim tim i određivanje vrednosti pojedinačnih elemenata), međutim, ne omogućavaju efikasno ažuriranje svih elemenata datog segmenta $[a, b]$ (tj. njihovo uvećanje za datu vrednost ili njihovu zamenu datom vrednošću). Ovo se može postići u vremenu $O(\log n)$ ako se na segmentno drvo primeni tehnika *lenje propagacije* (engl. lazy propagation). Ona je primer strategije lenjog izvršavanja kojim se izračunavanja odlažu sve dok odgovarajuće vrednosti nisu neophodne. Tehnika lenje propagacije se dosta oslanja na rad sa segmentnim drvetom odozgo naniže. Jednostavnosti radi, tehniku ćemo predstaviti kroz primer upita uvećavanja segmenata za datu vrednost.

Svaki čvor u segmentnom drvetu čuva zbir nekog segmenta orignalnog niza. Ako se taj segment u celosti sadrži unutar segmenta koji se ažurira, možemo unapred izračunati za koliko se povećava vrednost u tom čvoru. Naime, ako se svaka vrednost u segmentu povećava za v, tada se vrednost zbira tog segmenta povećava za $k \cdot v$, gde je k broj elemenata u tom segmentu. Vrednost zbira u tom čvoru time biva ažurirana u konstantnom vremenu, ali vrednosti zbirova unutar poddrveta kojima je taj čvor koren (uključujući i vrednosti u listovima koje odgovaraju vrednostima polaznog niza) i dalje ostaju neažurne. Njihovo ažuriranje zahtevalo bi linearno vreme, što je nedopustivo skupo. Ključna ideja je da se ažuriranje tih vrednosti odloži i da se one ne ažuriraju odmah, već samo kada zatrebaju tokom nekog naknadnog upita, tj. tokom neke kasnije posete tim čvorovima (koja se inače vrši u sklopu tog kasnijeg upita, a ne posebno u cilju ažuriranja

vrednosti). Naime, ne želimo da te čvorove posećujemo samo zbog ovog ažuriranja, već ćemo ažuriranje uraditi usput, tokom neke druge posete tim čvorovima koja bi se svakako morala desiti.

Postavlja se pitanje kako da signaliziramo da vrednosti zbirova u nekom poddrvetu nisu ažurne i dodatno ostavimo uputstvo na koji način ih treba ažurirati. U tom cilju u svakom od čvorova pored vrednosti zbira segmenta čuvamo i dodatni *koeficijent lenje propagacije*. Ako drvo u svom korenu ima koeficijent lenje propagacije c koji je različit od nule, to znači da vrednosti zbirova u celom tom drvetu nisu ažurne i da je svaki od listova tog drveta potrebno povećati za c i u odnosu na to ažurirati i vrednosti zbirova u svim unutrašnjim čvorovima tog drveta (uključujući i koren). Ažuriranje se može odlagati sve dok vrednost zbira u nekom čvoru ne postane zaista neophodna, a to je tek prilikom upita izračunavanja vrednosti zbira nekog segmenta. Ipak, vrednosti zbirova u čvorovima ćemo ažurirati i češće i to zapravo prilikom svake posete čvoru – bilo u sklopu operacije uvećanja vrednosti iz nekog segmenta pozicija polaznog niza, bilo u sklopu upita izračunavanja zbira nekog segmenta. Naime, na početku obe rekurzivne funkcije možemo proveriti da li je vrednost koeficijenta lenje propagacije tekućeg čvora različita od nule i ako jeste, ažurirati vrednost zbira u tom čvoru tako što ćemo ga uvećati za proizvod tog koeficijenta i broja elemenata koji taj čvor pokriva, zatim koeficijente lenje propagacije oba njegova deteta uvećati za taj koeficijent, a koeficijent lenje propagacije tog čvora postaviti na nulu (time koren drveta koji trenutno posećujemo postaje ažuran, a njegovim poddrvetima se daje uputstvo kako ih u budućnosti ažurirati). Primetimo da se na ovaj način izbegava ažuriranje celog drveta odjednom, već se ažurira samo koren, što je operacija složenosti $O(1)$.

Imajući ovo u vidu, razmotrimo kako se može implementirati funkcija koja vrši uvećanje svih elemenata nekog segmenta. Njena invarijanta će biti da svi čvorovi u drvetu ili sadrže ažurne vrednosti zbirova ili će biti ispravno obeleženi za kasnija ažuriranja (preko koeficijenta lenje propagacije), a da će nakon njenog izvršavanja koren drveta na kom je funkcija pozvana sadržati aktuelnu vrednost zbira. Nakon početnog obezbeđivanja da vrednost u tekućem čvoru postane ažurna, moguća su tri sledeća slučaja.

- Ako je segment u tekućem čvoru disjunktan u odnosu na segment koji se ažurira, tada su svi čvorovi u poddrvetu kojem je on koren ili već ažurni ili ispravno obeleženi za kasnije ažuriranje i nije potrebno ništa uraditi.
- Ako je segment koji odgovara tekućem čvoru potpuno sadržan u segmentu čiji se elementi uvećavaju, tada se njegova vrednost ažurira (uvećavanjem za $k \cdot v$, gde je k broj elemenata segmenta koji odgovara tekućem čvoru, a v vrednost uvećanja), a njegovoj deci se koeficijent lenje propagacije uvećava za vrednost v.
- Na kraju, ako se ova dva segmenta seku, tada se prelazi na rekurzivnu obradu oba deteta. Nakon izvršavanja funkcije nad njima, sigurni smo da će svi čvorovi u levom i desnom poddrvetu zadovoljavati uslov invarijante i da će koreni i levog i desnog poddrveta imati ažurne vrednosti. Ažurnu vrednost u korenu dobijamo sabiranjem vrednosti njegova dva deteta.

Lenjo segmentno drvo čuvaćemo korišćenjem dva niza: `drvo` i `lenjo`: u prvom ćemo čuvati elemente segmentnog drveta, a u drugom koeficijente lenje propagacije svakog od čvorova.

```
// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [x, y]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(vector<int>& drvo, vector<int>& lenjo, int k, int x, int y,
             int a, int b, int v) {
   // ažuriramo vrednost u korenu, ako nije ažurna
   if (lenjo[k] != 0) {
      drvo[k] += (y - x + 1) * lenjo[k];
      // ako nije u pitanju list, propagaciju prenosimo na decu
      if (x != y) {
```

```
        lenjo[2*k] += lenjo[k];
        lenjo[2*k+1] += lenjo[k];
      }
      // vrednost u korenu je sad ažurna
      lenjo[k] = 0;
   }
   // ako su intervali disjunktni, ništa nije potrebno raditi
   if (b < x || y < a) return;
   // ako je interval [x, y] ceo sadržan u intervalu [a, b]
   // vrsimo uvecanje cvora za k * v
   if (a <= x && y <= b) {
     drvo[k] += (y - x + 1) * v;
     // ako nije u pitanju list, propagaciju prenosimo na decu
     if (x != y) {
        lenjo[2*k] += v;
        lenjo[2*k+1] += v;
     }
   } else {
     // u suprotnom se intervali seku,
     // pa rekurzivno obilazimo poddrveta
     int s = (x + y) / 2;
     promeni(drvo, lenjo, 2*k, x, s, a, b, v);
     promeni(drvo, lenjo, 2*k+1, s+1, y, a, b, v);
     // azurnu vrednost u korenu dobijamo kao zbir vrednosti dece
     drvo[k] = drvo[2*k] + drvo[2*k+1];
   }
}

// ažurira elemente lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata originalnog niza sa pozicija iz segmenta [0, n-1]
// nakon što su u originalnom nizu svi elementi sa pozicija iz
// segmenta [a, b] uvećani za vrednost v
void promeni(vector<int>& drvo, vector<int>& lenjo, int n,
             int a, int b, int v) {
   promeni(drvo, lenjo, 1, 0, n-1, a, b, v);
}
```

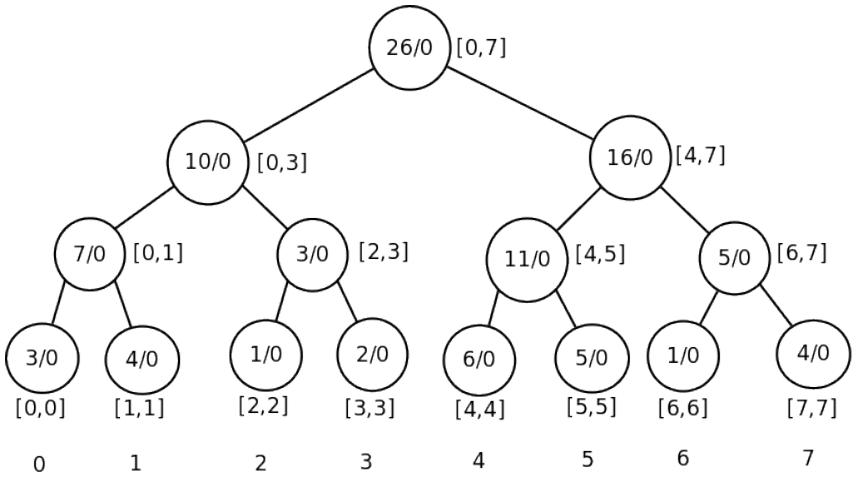
Složenost funkcije ažuriranja svih elemenata datog segmenta u lenjom segmentnom drvetu iznosi $O(\log n)$. Naime, kao i u slučaju operacije računanja sume segmenta u segmentnom drvetu, na svakom nivou se razmatra najviše 4 čvora, te je maksimalni broj čvorova koji se posećuje jednak $4 \log n$.

Primer 1.4.21

Prikažimo rad ove funkcije na primeru lenjog segmentnog drveta sa slike 1.26.

Prikažimo kako bismo sve elemente iz segmenta pozicija $[2, 7]$ *uvećali za* 3.

- *Krećemo od korena koji pokriva segment* $[0, 7]$. *Segmenti* $[0, 7]$ *i* $[2, 7]$ *se seku, pa stoga ažuriranje prepuštamo deci i nakon njihovog ažuriranja, pri povratku iz rekurzije vrednost korena određujemo kao zbir ažuriranih vrednosti njegove dece.*
  - *Na levoj strani se segment* $[0, 3]$ *seče sa segmentom* $[2, 7]$ *pa i on prepušta ažuriranje nasledni-*



Slika 1.26: Lenjo segmentno drvo: u svakom čvoru čuvamo vrednosti zbira odgovarajućeg segmenta i koeficijenta lenje propagacije. Inicijalno su vrednosti svih koeficijenata lenje propagacije jednaki nula.

cama i ažurira se tek pri povratku iz rekurzije.

- *Segment* $[0, 1]$ *je disjunktan u odnosu na segment* $[2, 7]$ *i tu onda nije potrebno ništa raditi.*
- *Segment* $[2, 3]$ *je ceo sadržan u segmentu* $[2, 7]$*, i za njega direktno znamo kako se zbir uvećava: pošto ovaj čvor pokriva dva elementa i svaki se uvećava za* 3*, zbir ovog segmenta se uvećava ukupno za* $2 \cdot 3 = 6$ *i postavlja se na* 9*. U ovom trenutku izbegavamo ažuriranje svih vrednosti u drvetu u kom je taj element koren, već samo naslednicima upisujemo da je potrebno propagirati uvećanje za* 3*, ali samu propagaciju odlažemo za trenutak kada ona postane neophodna.*

– *U povratku iz rekurzije, vrednost* 10 *ažuriramo i nova vrednost je jednaka* $7 + 9 = 16$*.*

– *Što se tiče desnog poddrveta, segment* $[4, 7]$ *je ceo sadržan u segmentu* $[2, 7]$*, pa možemo direktno izračunati novu vrednost zbira u ovom čvoru. Naime, pošto se* 4 *elementa uvećavaju za po* 3*, ukupan zbir se uvećava za* $4 \cdot 3 = 12$*. Zato se vrednost* 16 *menja u* $16 + 12 = 28$*. Propagaciju ažuriranja kroz poddrvo sa korenom u ovom čvoru odlažemo i samo njegovoj deci beležimo da je uvećanje za* 3 *potrebno izvršiti u nekom kasnijem trenutku.*

- *Pri povratku iz rekurzije vrednost u korenu ažuriramo sa* 26 *na* $16 + 28 = 44$*.*

Nakon izvršavanja ovih operacija dobija se drvo prikazano na slici 1.27. Ovo lenjo segmentno drvo odgovara nizu $3, 4, 4, 5, 9, 8, 4, 7$*.*

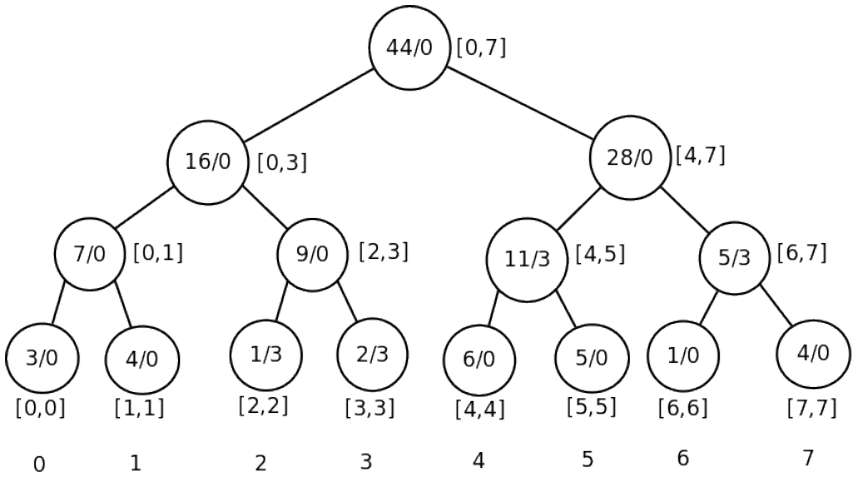
Pretpostavimo da je u dobijenom segmentnom drvetu potrebno elemente iz segmenta $[0, 5]$ *uvećavati za vrednost* 2*.*

- *Ponovo se kreće od korena lenjog segmentnog drveta i kada se ustanovi da se segment* $[0, 7]$ *seče sa segmentom* $[0, 5]$ *ažuriranje se prepušta deci i vrednost u korenu se ažurira tek pri povratku iz rekurzije.*

  – *Segment* $[0, 3]$ *je ceo sadržan u segmentu* $[0, 5]$*, pa se zato vrednost* 16 *uvećava za* $4 \cdot 2 = 8$ *i postavlja na* 24*. Poddrveta se ne ažuriraju odmah, već se samo njihovim korenima upisuje da je sve vrednosti potrebno ažurirati za* 2*.*

  – *U desnom poddrvetu segment* $[4, 7]$ *se seče sa* $[0, 5]$*, pa se rekurzivno obrađuju poddrveta.*

    – *Pri obradi čvora sa vrednošću* 11*, primećuje se da je on trebalo da bude ažuriran jer je njegov koeficijent lenje propagacije različit od nula, međutim još nije, pa se najpre njegova vrednost ažurira i uvećava za* $2 \cdot 3$ *i sa* 11 *menja na* 17*. Njegovi naslednici se ne ažuriraju odmah, već samo ako to bude potrebno i njima se samo upisuje lenja vrednost* 3*.*



Slika 1.27: Lenjo segmentno drvo nakon ažuriranja svih elemenata između pozicija 2 i 7 za vrednost 3.

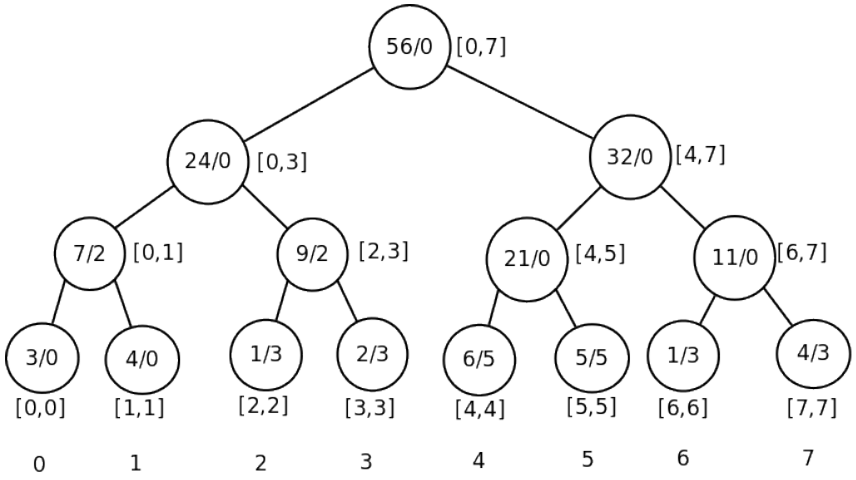
Tek nakon toga se primećuje da se segment $[4, 5]$ *ceo sadrži u segmentu* $[0, 5]$*, pa se vrednost* 17 *uvećava za* $2 \cdot 2 = 4$ *i postavlja na* 21*. Poddrveta se ne ažuriraju odmah, već samo po potrebi tako što se u njihovim korenima postavi vrednost lenjog koeficijenta. Pošto je u njima već upisana vrednost* 3*, ona se sada uvećava za* 2 *i postavlja na* 5*.*

– *Sada se prelazi na obradu poddrveta u čijem je korenu vrednost* 5 *i pošto ono nije ažurno, najpre se vrednost* 5 *uvećava za* $2 \cdot 3 = 6$ *i postavlja na* 11*, a njegovoj deci se lenji koeficijent postavlja na* 3*.*

Nakon toga se primećuje da je segment $[6, 7]$ *disjuktan sa* $[0, 5]$ *i ne radi se ništa.*

- *U povratku kroz rekurziju se ažuriraju vrednosti roditeljskih čvorova.*

Nakon ovih operacija dobija se drvo prikazano na slici 1.28. Ovo lenjo segmentno drvo odgovara nizu $5, 6, 6, 7, 11, 10, 4, 7$.



Slika 1.28: Lenjo segmentno drvo nakon ažuriranja svih elemenata na pozicijama između 0 i 5 za 2.

Funkcija izračunavanja vrednosti zbira segmenta ostaje praktično nepromenjena, osim što se pri ulasku u svaki čvor vrši njegovo ažuriranje, ako je potrebno. Stoga i njena složenost ostaje nepromenjena i iznosi $O(\log n)$.

```
// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije k u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [x, y]
// izračunava se zbir elemenata polaznog niza
// sa pozicija iz segmenta [a, b]
int saberi(vector<int>& drvo, vector<int>& lenjo, int k, int x, int y,
           int a, int b) {
  // ažuriramo vrednost u korenu, ako nije ažurna
  if (lenjo[k] != 0) {
     drvo[k] += (y - x + 1) * lenjo[k];
     if (x != y) {
       lenjo[2*k] += lenjo[k];
       lenjo[2*k+1] += lenjo[k];
     }
     lenjo[k] = 0;
  }

  // intervali [x, y] i [a, b] su disjunktni
  if (b < x || a > y) return 0;
  // interval [x, y] je potpuno sadržan unutar intervala [a, b]
  if (a <= x && y <= b)
      return drvo[k];
  // intervali [x, y] i [a, b] se seku
  int s = (x + y) / 2;
  return saberi(drvo, lenjo, 2*k, x, s, a, b) +
         saberi(drvo, lenjo, 2*k+1, s+1, y, a, b);
}

// na osnovu lenjog segmentnog drveta koje je smešteno
// u nizove drvo i lenjo od pozicije 1 u kome se čuvaju zbirovi
// elemenata polaznog niza sa pozicija iz segmenta [0, n-1]
// izračunava se zbir elemenata polaznog niza sa pozicija
// iz segmenta [a, b]
int saberi(vector<int>& drvo, vector<int>& lenjo, int n, int a, int b) {
  // računamo doprinos celog niza,
  // tj. elemenata iz intervala [0, n-1]
  return saberi(drvo, lenjo, 1, 0, n-1, a, b);
}
```

Primetimo da se tokom izvršavanja ove funkcije drvo može menjati, pa argumenti funkcije ne smeju biti konstantni (iako se suštinski vrednosti u drvetu ne menjaju, njegova interna reprezentacija, tj. raspodela podataka između nizova `drvo` i `lenjo` se može menjati).

Primer 1.4.22

Prikažimo rad prethodne funkcije na tekućem primeru. Razmotrimo kako se za drvo prikazano na slici 1.28 izračunava zbir elemenata iz segmenta $[3, 5]$.

- *Krećemo od korena drveta koje sadrži sumu segmenta* $[0, 7]$. *Segment* $[0, 7]$ *se seče sa* $[3, 5]$, *pa se rekurzivno obrađuju deca.*
  - *U levom poddrvetu segment* $[0, 3]$ *takođe ima presek sa* $[3, 5]$ *pa prelazimo na naredni nivo rekurzije.*

– *Prilikom posete čvora u čijem je korenu vrednost* 7 *primećuje se da njegova vrednost nije ažurna, pa se koristi prilika da se ona ažurira, tako što se uveća za* $2 \cdot 2 = 4$ *i postaje* 11*, a naslednicima se lenji koeficijent postavlja na* 2*.*

Pošto je segment $[0, 1]$ *disjunktan sa* $[3, 5]$*, vraća se vrednost* 0*.*

– *Prilikom posete čvora u čijem je korenu vrednost* 9 *primećuje se da njegova vrednost nije ažurna, pa se koristi prilika da se ona ažurira, tako što se uveća za* $2 \cdot 2 = 4$ *i postaje* 13*, a naslednicima se lenji koeficijent uvećava za* 2*, odnosno postaje* 5*.*

Segment $[2, 3]$ *se seče sa* $[3, 5]$*, pa se rekurzivno vrši obrada poddrveta.*

- *Vrednost* 1 *se prvo ažurira tako što se poveća za* $1 \cdot 5 = 5$ *i postaje* 6*, a onda, pošto je* $[2, 2]$ *disjunktno sa* $[3, 5]$ *vraća se vrednost* 0*.*
- *Vrednost* 2 *se takođe prvo ažurira tako što se poveća za* $1 \cdot 5 = 5$ *i postaje* 7*, a pošto je segment* $[3, 3]$ *potpuno sadržan u* $[3, 5]$ *vraća se vrednost* 7*.*

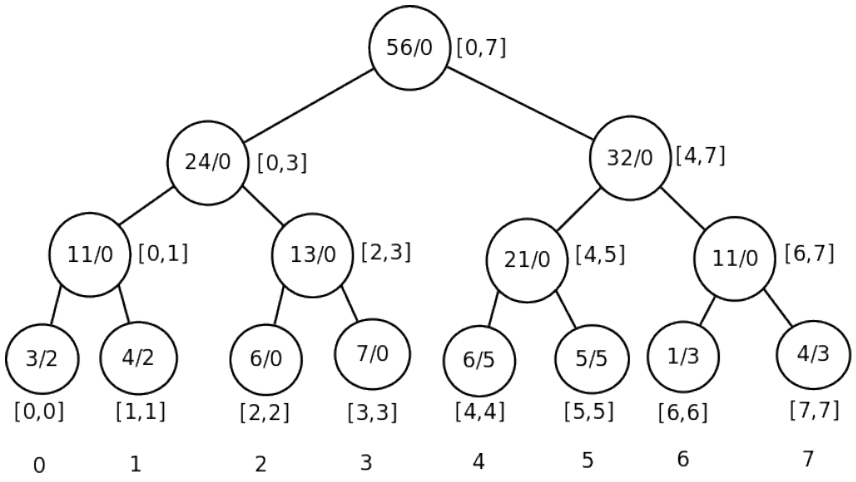
– *U desnom poddrvetu je čvor sa vrednošću* 32 *ažuran, segment* $[4, 7]$ *se seče sa* $[3, 5]$*, pa se prelazi na obradu naslednika.*

– *Čvor sa vrednošću* 21 *je ažuran, segment* $[4, 5]$ *je ceo sadržan u* $[3, 5]$*, pa se vraća vrednost* 21*.*

– *Čvor sa vrednošću* 11 *je takođe ažuran, ali je segment* $[6, 7]$ *disjunktan u odnosu na* $[3, 5]$*, pa se vraća vrednost* 0*.*

• *Dakle, čvorovi* 13 *i* 24 *vraćaju vrednost* 7*, čvor* 32 *vraća vrednost* 21*, pa čvor* 56 *vraća vrednost* $7 + 21 = 28$*.*

Dakle, zbir segmenta $[3, 5]$ *u tekućem drvetu je* 28*. Stanje drveta nakon izvršavanja upita je prikazano na slici 1.29.*



Slika 1.29: Lenjo segmentno drvo nakon računanja zbira elemenata između pozicija 3 i 5

Dakle, u lenjom segmentnom drvetu ne možemo analizom pojedinačnog čvora (npr. lista) zaključiti koja je tačna vrednost tog čvora, međutim, kada nam bude bila potrebna vrednost nekog čvora u drvetu, mi ćemo se od korena spustiti do tog čvora i, idući tom putanjom, sve vrednosti na toj putanji ažurirati. Na taj način, kada budemo stigli do željenog čvora, imaćemo njegovu ažurnu vrednost, što je jedino i važno.

	point query	*range query*	*point update*	*range update*
Lenjo segmentno drvo	dobro / $O(\log n)$	dobro / $O(\log n)$	dobro / $O(\log n)$	dobro / $O(\log n)$

2. Grafovski algoritmi

Grafovi su jedna od najkorisnijih struktura podataka. Još u davna vremena su korišćeni za predstavljanje mreža puteva između gradova i odgovarajućih mapa, koje su putnici nosili sa sobom tokom putovanja. Poznat je slučaj kopije mape iz petog veka koja je sadržala mrežu puteva Rimskog carstva sa nazivima gradova i dužinama puteva između njih, koja je pružala dovoljno informacija potrebnih da se pronađe najkraći put između dva grada. Još jedna od klasičnih primena grafova (specijalno drveta) je predstavljanje genealogija. Naime, porodična stabla su vekovima korišćena u svrhe odgovora na pravna pitanja poput pitanja dozvoljenih brakova, nasledstva i nasleđivanja vlasti. Naravno, postoji mnogo drugih poznatih primera primena grafova, kao što su predstavljanje strana i ivica poliedara, komunikacione mreže, električna kola, strukturne formule molekula, društvene igre, traženje izlaza iz lavirinta, a u današnje vreme veoma popularna primena je za modelovanje odnosa korisnika na društvenim mrežama.

U ovom poglavlju upoznaćemo se sa pojmom grafa, načinima na koje se on može predstaviti u računaru, a zatim i osnovnim grafovskim algoritmima. Najpre ćemo razmotriti dva osnovna algoritma za obilazak grafa: obilazak u dubinu i obilazak u širinu, kao i karakteristike grafova u odnosu na ove dve vrste pretrage. Nakon toga bavićemo se pitanjem povezanosti grafa: razmotrićemo algoritme za određivanje komponenti povezanosti u neusmerenom, odnosno komponenti jakih povezanosti u usmerenom grafu, kao i algoritmima za određivanje mostova i artikulacionih tačaka u grafu.

Razmotrićemo svojstva acikličkih usmerenih grafova, kao i algoritme za konstrukciju topološkog sortiranja grafa. Videćemo na koji način se poznavanje topološkog sortiranja može iskoristiti za efikasno određivanje najkraćih puteva od jednog do svih ostalih čvorova u acikličkom grafu.

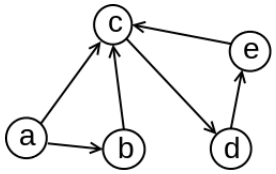
Biće reči i o algoritmima za utvrđivanje da li u datom grafu postoji Ojlerov, odnosno Hamiltonov put (ciklus), tj. put (ciklus) koji kroz svaku granu grafa, odnosno kroz svaki čvor grafa prolazi tačno jednom. Pokazuje se da su ova dva problema iako naizgled jako bliska sasvim različite težine i da se pristupi njihovom rešavanju veoma razlikuju.

Bavićemo se, takođe, različitim problemima koji se bave težinskim grafovima i određivanjem najkraćih puteva u grafu, i to konkretno: Dajkstrinim i Belman-Fordovim algoritmom za određivanje najkraćih puteva od jednog fiksiranog čvora i Flojd-Varšalovim algoritmom za računanje najkraćih čvorova između svaka dva čvora u grafu.

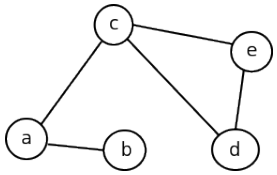
Konačno, prikazaćemo dva različita algoritma za konstrukciju minimalnog povezujućeg drveta datog grafa.

2.1 Osnovni pojmovi

Formalno, *graf* $G = (V, E)$ se sastoji od skupa V *čvorova* i skupa E *grana* (oznake V i E su početna slova engleskih reči za teme – vertex i granu – edge). Grana najčešće odgovara paru različitih čvorova, mada su ponekad dozvoljene i *petlje*, odnosno grane koje vode od čvora ka njemu samom. Graf može biti *neusmeren* tj.
neorijentisan ili *usmeren* tj. *orijentisan*. Grane usmerenog grafa su uređeni parovi čvorova i kod njih je redosled čvorova koje grana povezuje bitan. Ako se graf predstavlja grafički, onda se grane usmerenog grafa crtaju kao strelice usmerene od jednog čvora – početka ka drugom čvoru – kraju grane (slika 2.1). Grane neusmerenog grafa su neuređeni parovi čvorova: one se crtaju kao obične linije, bez usmerenja (slika 2.2).



Slika 2.1: Primer usmerenog grafa.



Slika 2.2: Primer neusmerenog grafa.

Susedom čvora u nazvaćemo svaki čvor v do kog postoji grana iz čvora u.

Primer 2.1.1

Susedi čvora c u neusmerenom grafu sa slike 2.2 su čvorovi a, d i e, dok je u usmerenom grafu sa slike 2.1 jedini sused čvora c čvor d. Primetimo da je u grafu prikazanom na slici 2.1 čvor c povezan i sa čvorovima a, b i e, međutim, oni nisu susedi čvora c jer su odgovarajuće grane usmerene od čvorova a, b i e ka čvoru c.

Stepen $d(v)$ čvora v u neusmerenom grafu je broj grana susednih čvoru v (odnosno broj grana koje čvor v povezuju sa nekim drugim čvorom). U usmerenom grafu razlikujemo *ulazni stepen* čvora v, koji je jednak broju grana za koje je čvor v kraj, i *izlazni stepen* čvora v, koji je jednak broju grana za koje je čvor v početak.

Primer 2.1.2

Stepen čvora c u neusmerenom grafu sa slike 2.2 je 3, dok je ulazni stepen čvora c usmerenog grafa sa slike 2.1 jednak 3, a izlazni stepen 1.

Put od čvora v_1 do čvora v_k u grafu G je niz čvorova grafa $(v_1, v_2, \ldots, v_k)$ povezanih granama grafa (v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k). Put je *prost* ako se svaki čvor u njemu pojavljuje samo jednom. Za čvor v se kaže da je *dostižan* iz čvora u ako postoji put (usmeren, odnosno neusmeren, zavisno od grafa) od čvora u do čvora v. Po definiciji svaki čvor v je dostižan iz čvora v tj. iz sebe. *Ciklus* je put čiji se prvi i poslednji čvor poklapaju. Ciklus je *prost* ako se, sem prvog i poslednjeg čvora, ni jedan drugi čvor u njemu ne javlja dva puta.

Primer 2.1.3

Niz čvorova (a, b, c, d, e) u usmerenom grafu sa slike 2.1 predstavlja jedan prost put u tom grafu, dok put (b, c, d, e, c) nije prost.

Čvor e grafa sa slike 2.1 dostižan je iz čvora a jer postoji put (a, c, d, e) od čvora a do čvora e.

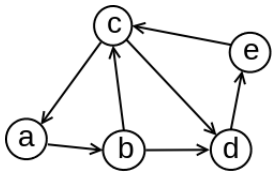
Niz čvorova (c, d, e, c) predstavlja prost ciklus i u datom usmerenom i u datom neusmerenom grafu.

Usmeren graf u kome nema ciklusa naziva se *usmeren aciklički graf* (engl. directed acyclic graph, DAG). *Neusmereni oblik* usmerenog grafa $G = (V, E)$ je isti graf, bez smerova na granama (tako da su parovi čvorova u E neuređeni). Za neusmeren graf se kaže da je *povezan* ako postoji put između proizvoljna dva čvora u grafu. Za usmerene grafove razlikujemo pojam slabe i jake povezanosti: usmereni graf je *slabo povezan* ako u njegovom neusmerenom obliku postoji put između svaka dva čvora u grafu, a *jako povezan* ako za svaka dva čvora u i v u grafu postoji usmeren put od čvora u do čvora v i usmeren put od čvora v do čvora u.

Primer 2.1.4

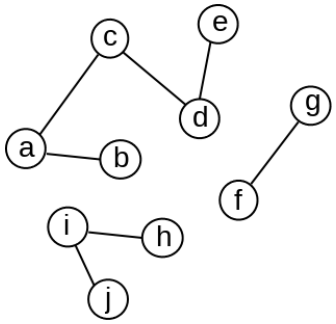
Neusmeren graf sa slike 2.2 je povezan.

Usmeren graf sa slike 2.1 je slabo povezan, ali nije jako povezan: od čvora b, na primer, nije moguće stići do čvora a. S druge strane, graf sa slike 2.3 je jako povezan.



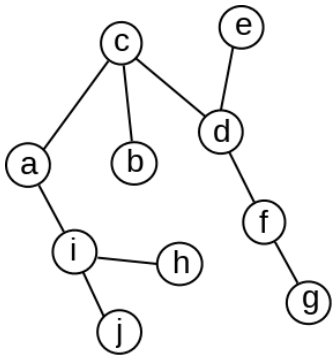
Slika 2.3: Primer usmerenog grafa koji je jako povezan.

Neusmereni graf je *šuma* ako ne sadrži cikluse (slika 2.4). *Drvo* je povezana šuma (slika 2.5). Šuma sadrži jedno ili više drveta.



Slika 2.4: Primer grafa koji je šuma.

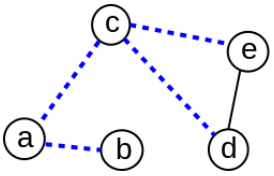
Graf $G' = (V', E')$, $E' \subseteq V' \times V'$, je *podgraf* grafa $G = (V, E)$, $E \subseteq V \times V$ ako istovremeno važi $V' \subseteq V$ i $E' \subseteq E$. Na primer, graf sa skupom čvorova $\{a, b, c, d\}$ i skupom grana $\{(a, b), (a, c), (c, d)\}$ je jedan podgraf usmerenog grafa sa slike 2.1. *Povezujuće drvo* neusmerenog grafa G je njegov podgraf koji je drvo i sadrži sve čvorove grafa G. *Povezujuća šuma* neusmerenog grafa G je njegov podgraf koji je šuma i sadrži sve čvorove grafa G.



Slika 2.5: Primer grafa koji je drvo.

Primer 2.1.5

Grane jednog povezujućeg drveta neusmerenog grafa sa slike 2.2 su prikazane plavom bojom na slici 2.6: ono sadrži sve čvorove grafa i skup grana $\{(a,b),(a,c),(c,d),(c,e)\}$.

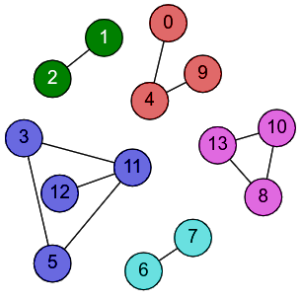


Slika 2.6: Neusmeren graf i jedno njegovo povezujuće drvo (čije su grane prikazane plavom bojom).

Ako neusmereni graf $G = (V, E)$ nije povezan, onda se on može na jedinstven način razložiti u skup povezanih podgrafova, čiji skupovi čvorova predstavljaju klase ekvivalencije za relaciju dostižnosti i koji se nazivaju *komponente povezanosti* grafa G.

Primer 2.1.6

Na slici 2.7 prikazan je graf sa 5 *komponenata povezanosti (čvorovi svake komponente su prikazani zasebnom bojom).*



Slika 2.7: Komponente povezanosti

2.2 Predstavljanje grafa

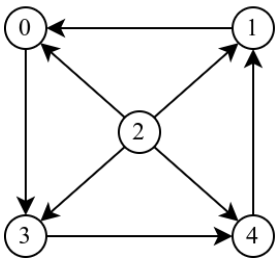
Uobičajena su dva načina predstavljanja grafova: matricom povezanosti i listama povezanosti.

2.2.0.1 Matrica povezanosti

Graf se može predstaviti *matricom povezanosti*, odnosno *matricom susedstva* grafa (engl. adjacency matrix). Neka je $|V| = n$ i $V = \{v_0, v_1, \dots, v_{n-1}\}$. Matrica povezanosti grafa G je kvadratna matrica $A = (a_{ij})$ reda n, sa elementima a_{ij} koji su jednaki 1 (tj. $\top$) ako i samo ako $(v_i, v_j) \in E$, odnosno ako postoji grana od čvora v_i do čvora v_j; ostali elementi matrice A imaju vrednost 0 (tj. $\bot$). Vrsta i ove matrice je, dakle, niz dužine n čija je j-ta koordinata jednaka 1 ako iz čvora v_i vodi grana ka čvoru v_j, odnosno 0 u protivnom.

Primer 2.2.1

Primer reprezentacije jednog usmerenog grafa matricom povezanosti je prikazan na slici 2.8. Čvorovi su numerisani brojevima od 0 *do* 4*. Na primer, na poziciji* $(1, 0)$ *u matrici povezanosti nalazi se vrednost* 1 *jer postoji grana iz čvora* 1 *ka čvoru* 0*, dok se na poziciji* $(0, 1)$ *nalazi* 0 *jer u grafu ne postoji suprotno usmerena grana.*



	0	1	2	3	4
0	0	0	0	1	0
1	1	0	0	0	0
2	1	1	0	1	1
3	0	0	0	0	1
4	0	1	0	0	0

Slika 2.8: Predstavljanje grafa matricom povezanosti.

Ako je graf neusmeren, matrica A je simetrična. Nedostatak predstavljanja grafa matricom povezanosti je to što ona uvek zauzima prostor veličine n^2 tj. $\Theta(|V|^2)$, nezavisno od toga koliko grana ima graf. Ako je broj grana u grafu mali, većina elemenata matrice povezanosti je jednaka nula.

Ako se za predstavljanje grafa koristi matrica povezanosti, složenost operacije dodavanja grane u graf, odnosno operacije uklanjanja grane iz grafa je $O(1)$. Takođe, i ispitivanje da li su dva čvora u grafu povezana granom je složenosti $O(1)$. Prolazak kroz sve čvorove susedne datom čvoru je složenosti $\Theta(|V|)$.

U jeziku C++ graf predstavljen matricom povezanosti možemo deklarisati na sledeći način:

```
bool matricaPov[MAX][MAX];
```

gde je `MAX` maksimalni broj čvorova grafa. Ako broj čvorova saznajemo tek u vreme izvršavanja programa, možemo upotrebiti dinamičke strukture podataka (na primer, `vector`) i upotrebiti sledeću reprezentaciju:

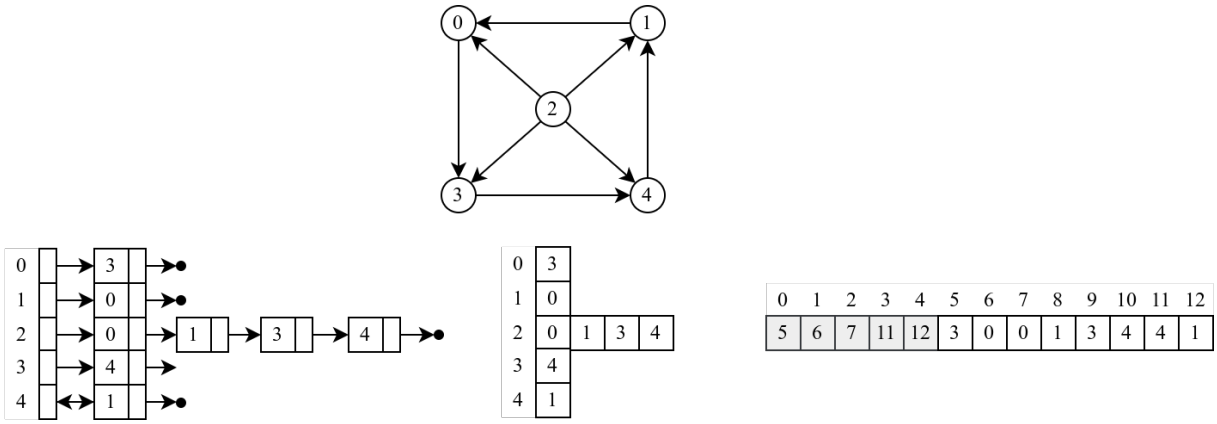
```
vector<vector<bool>> matricaPov(n);
for (int i = 0; i < n; i++)
   matricaPov[i].resize(n);
```

2.2.0.2 Liste povezanosti

Umesto da se i sve nepostojeće grane eksplicitno predstavljaju, kao što je slučaj sa matricom povezanosti grafa, mogu se formirati povezane liste od jedinica iz i-te vrste, $i = 0, 1, \dots, n-1$. Ovaj način predstavljanja grafa naziva se *liste povezanosti*, odnosno *liste susedstva* (engl. adjacency list). Svakom čvoru pridružuje se povezana lista koja sadrži sve čvorove do kojih postoji grana iz tog čvora. Lista može biti uređena prema rednim brojevima čvorova na krajevima njenih grana. Graf je predstavljen nizom lista. Svaki element niza sadrži ime (indeks) čvora i pokazivač na njegovu listu suseda.

Primer 2.2.2

Primer predstavljanja grafa listama povezanosti je prikazan na slici 2.9. Lista povezanosti pridružena čvoru 1 *je dužine* 1 *jer iz čvora* 1 *polazi tačno jedna grana. Slično, lista povezanosti pridružena čvoru* 2 *je dužine* 4 *jer iz čvora* 2 *polaze četiri grane.*



Slika 2.9: Predstavljanje grafa listama povezanosti (levo su prikazane povezane liste, u sredini nizovi, a desno statička implementacija uz pomoć jednog niza).

Iako naziv tako sugeriše, implementacija ovakve reprezentacije grafa ne mora biti zasnovana na povezanim listama, već se umesto povezanih listi može koristiti dinamički proširiv niz (vektor), ili neka reprezentacija skupa (balansirano binarno drvo ili heš tabela).

U jeziku C++ se graf predstavljen listama povezanosti, pri čemu se liste povezanosti implementiraju korišćenjem vektora, deklariše na sledeći način:

```
vector<vector<int>> listaSuseda(n);
```

Na primer, usmereni graf sa slike 2.9 predstavićemo kao:

```
vector<vector<int>> listaSuseda {{3}, {0}, {0, 1, 3, 4}, {4}, {1}};
```

Broj čvorova grafa možemo dobiti kao broj elemenata u spoljašnjem vektoru:

```
int brCvorova = listaSuseda.size();
```

Novu granu `(cvorOd,cvorDo)` možemo dodati u graf na sledeći način:

```
listaSuseda[cvorOd].push_back(cvorDo);
```

dok kroz sve susede čvora možemo iterirati na sledeći način:

```
for (int cvorDo : listaSuseda[cvorOd])
    ...
```

Ako je graf *statički*, odnosno ako nisu dozvoljena umetanja i brisanja čvorova i grana, onda se liste povezanosti mogu predstaviti statičkim nizom a dužine $|V| + |E|$ u slučaju usmerenog grafa (odnosno dužine $|V| + 2|E|$ u slučaju neusmerenog grafa). Prvih $|V|$ članova niza su pridruženi čvorovima. Element niza a na poziciji i, $i < |V|$ je pridružen čvoru v_i i sadrži indeks početka spiska čvorova susednih čvoru v_i, $i = 0, 1, \ldots, n-1$, dok se na pozicijama i, $|V| \leq i < |V| + |E|$ nalaze informacije o susedima čvorova.

Naime, susedi čvora v_i za $0 \leq i < n-1$ nalaze se u nizu a na pozicijama $[a[i], a[i+1])$, dok se susedi čvora v_{n-1} nalaze na pozicijama od $a[n-1]$ do kraja niza (slika 2.9, desno). Primetimo da se na poziciji 0 u nizu a nalazi vrednost koja odgovara broju čvorova u grafu.

Primer 2.2.3

Primer predstavljanja grafa sa slike 2.9(a) statičkim nizom je prikazan na slici 2.9, desno. Na poziciji 0 *nalazi se vrednost* 5*, a na poziciji* 1 *vrednost* 6*, što ukazuje na to da se na pozicijama* $[5, 6)$ *u ovom nizu nalaze susedi čvora* 0 *– u ovom slučaju to je samo čvor* 3*. Susedi čvora* 2 *nalaze se na pozicijama* $[7, 11)$ *i to su redom* $0, 1, 3$ *i* 4*.*

Sa matricama povezanosti je jednostavnije raditi. S druge strane, liste povezanosti su prostorno efikasnije za grafove sa malim brojem grana: njihova memorijska složenost je $O(|V| + |E|)$, za razliku od matrica povezanosti čija je memorijska složenost $O(|V|^2)$. U praksi se često radi sa *retkim* grafovima koji imaju znatno manje grana od maksimalnog mogućeg broja, što je $n(n-1)/2$ za neusmereni, odnosno $n(n-1)$ za usmereni graf ako isključimo petlje (odnosno $n(n-1)/2 + n = n(n+1)/2$ grana za neusmereni, odnosno $n(n-1) + n = n^2$ za usmereni graf ako dozvolimo petlje) i tada je efikasnije koristiti liste povezanosti. U slučaju kada se za implementaciju koriste povezane liste, ispitivanje da li su dva čvora u grafu povezana, kao i uklanjanje grane iz grafa je u najgorem slučaju složenosti $O(|V|)$. U slučaju kada se za implementaciju lista povezanosti koriste heš tabele, očekivano vreme izvršavanja ovih operacija je $O(1)$. Prolazak kroz sve čvorove susedne čvoru v je složenosti $O(d(v))$, gde je $d(v)$ stepen čvora v u slučaju neusmerenog grafa, odnosno izlazni stepen čvora v ako je graf usmeren. Dodavanje novog čvora u graf je jednostavnije nego u slučaju reprezentacije grafa matricom povezanosti.

Treba pomenuti da postoje i drugi načini za predstavljanje grafa: graf se, na primer, može čuvati i kao niz grana, gde se za svaku granu čuva informacija sa kojim čvorovima je incidentna.

U narednim algoritmima smatraćemo da je graf sa kojim radimo dinamički i da je, ako nije drugačije navedeno, zadat listama povezanosti.

2.3 Obilazak grafova

Prvi problem na koji se nailazi pri konstrukciji proizvoljnog algoritma za obradu grafa je kako pregledati graf, tj. njegove čvorove i grane. Za razliku od, na primer, nizova gde je taj problem trivijalan zbog jednodimenzionalnosti ulaza (nizovi se mogu lako pregledati sekvencijalnim prolaskom kroz elemente), pregledanje grafa, odnosno njegov *obilazak*, nije trivijalan problem.

Problem

Definisati algoritam koji krenuvši od zadatog čvora grafa posećuje (na primer, ispisuje, označava ili na neki drugi način obrađuje) sve čvorove koji su dostupni od tog čvora. Svaki čvor treba da bude posećen tačno jednom, a čvorovi mogu da budu posećeni u proizvoljnom redosledu.

Razmotrimo slučaj lavirinta u obliku pravougaone mreže polja, takvih da se sa svakog polja može preći na neko od četiri susedna polja: desno, levo, dole i gore. Međutim, neka od polja lavirinta sadrže zidove i na njih se ne može preći. Lavirint sadrži dva posebna polja koja nazivamo ulaz i izlaz. Potrebno je pronaći put kojim se od ulaznog polja može stići do izlaznog polja u lavirintu.

Ovaj problem možemo predstaviti kao grafovski problem: svakom polju lavirinta koje ne sadrži zid pridružujemo jedan čvor grafa, dok grana između dva čvora postoji ako su odgovarajuća polja pravougaone mreže susedna. Na ovaj način se problem pronalaska puta kroz lavirint svodi na traženje puta kroz graf od ulaznog čvora do izlaznog čvora.

Postoje dva osnovna algoritma za obilazak grafa: *obilazak u dubinu* i *obilazak u širinu*. Umesto termina obilazak često se upotrebljava i termin *pretraga*.

U opštem slučaju, algoritam obilaska tj. pretrage povezanog grafa iz čvora r može imati sledeću strukturu:

Algoritam 1 Obilazak grafa

```
1: procedure ObilazakGrafa(početni čvor r)
2:     dodaj čvor r u kolekciju K
3:     while K nije prazna do
4:         uzmi čvor u iz K
5:         označi čvor u
6:         po potrebi izvrši obradu čvora u
7:         for all grana (u, v) do
8:             if čvor v nije označen then
9:                 ubaci čvor v u K
```

Mehanizam označavanja čvorova nam daje garancije da će se algoritam zaustaviti – svaki čvor biće označen (tj. posećen i obrađen) najviše jednom. Pretpostavlja se da su u početku svi čvorovi neoznačeni. U zavisnosti od toga koja se kolekcija K odabere, dobijaju se različiti algoritmi obilaska koji se mogu primeniti za rešavanje različitih problema (za neke probleme je bitno koji se obilazak primenjuje, dok za neke nije).

- Ukoliko za kolekciju K odaberemo stek, dobijamo neku vrstu algoritma *obilaska u dubinu*[1].
- Ako je kolekcija K red, dobijamo algoritam *obilaska u širinu*.
- Ako je kolekcija K red sa prioritetom dobijamo *obilazak prema prioritetu*. Ako prioritet čvora predstavlja težina grane koja ga spaja sa čvorovima koje smo prethodno obišli dobijamo minimalno povezujuće drvo datog grafa, a ako prioritet čvora predstavlja najkraće do tada određeno njegovo rastojanje od polaznog čvora, dobijamo najkraće puteve od polaznog čvora do svih ostalih čvorova u datom grafu. O težinskim grafovima i ovim algoritmima biće više reči u poglavlju 2.8.

2.3.1 Obilazak u dubinu

Razmotrimo prethodno razmatrani problem traženja puta kroz lavirint. Ovaj problem možemo rešiti tako što izaberemo neku putanju u lavirintu (po nekom unapred definisanom pravilu) i pratimo je sve dok ne stignemo do izlaza ili dok ne dođemo do polja iz koga ne možemo dalje – u tom slučaju se vraćamo unazad i na prvoj prethodnoj raskrsnici idemo nekim alternativnim putem. Ovo je ideja tzv. *obilaska u dubinu*, odnosno *pretrage u dubinu* (DFS, skraćenica od depth–first–search). Obilazak u dubinu je praktično isti za neusmerene i usmerene grafove. Međutim, pošto želimo da ispitamo neke osobine grafova koje nisu iste za neusmerene i usmerene grafove, razmatranje obilaska u dubinu je podeljeno na dva dela: na obilazak neusmerenih i obilazak usmerenih grafova.

2.3.1.1 Neusmereni grafovi

Neka je dat graf $G = (V, E)$. Želimo da izvršimo obilazak grafa tako da uvek kada je to moguće idemo dalje (u dubinu) pre nego što se vratimo u čvor u kom smo već bili. Ovaj pristup zove se *obilazak u dubinu* (DFS). Osnovni razlog korisnosti obilaska u dubinu leži u njenoj jednostavnosti i lakom rekurzivnnom opisu. Implementacija može takođe biti iterativna, a umesto sistemskog steka koji se koristi pri realizaciji rekurzije, može se eliminisati rekurzija tako što se koristi poseban stek – takva implementacija se uklapa u opšti opis algoritama obilaska tj. obilazak grafova dat na početku ovog poglavlja.

Rekurzivna implementacija

Pretraga u dubinu se jednostavno definiše rekurzivno. Razmotrimo problem obilaska grafa u dubinu kada je graf zadat listama povezanosti i kada je dat čvor r grafa iz koga se započinje obilazak. Inicijalno su svi čvorovi *neoznačeni*. Čvorove ćemo *označavati* u trenutku kada ih po prvi put posetimo. Obično se prilikom označavanja čvora vrši neka njegova obrada (mada ćemo videti da obrada može da se vrši i na

[1] Kako će u nastavku biti pokazano, mali detalji u implementaciji mogu uticati na promenu redosleda obilaska čvorova i/ili memorijskih zahteva algoritma, tako da ovako implementiran algoritam ne mora da se poklopi sa osnovnom algoritmom pretrage u dubinu koja se definiše rekurzivno.

drugim mestima). Algoritam funkcioniše tako što označi početni čvor, a zatim se rekurzivno primeni na sve njegove neoznačene susede.

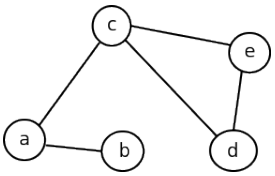
Algoritam 2 Obilazak u dubinu – rekurzivna formulacija

```
1: procedure DFS(čvor u)
2:     označi čvor u
3:     izvrši ulaznu obradu čvora u
4:     for all sused v čvora u do
5:         if čvor v nije označen then
6:             DFS(v)
7:     izvrši izlaznu obradu čvora u
8:
9: DFS(početni čvor r)
```

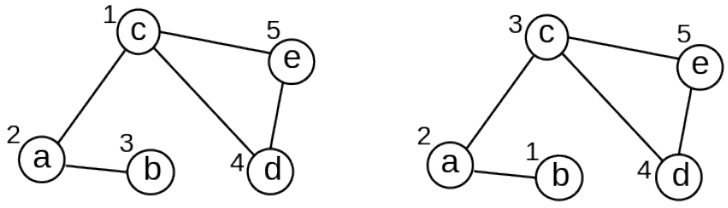
Dakle, svaki čvor u za koga se poziva funkcija DFS se označava kao posećen. Zatim se među susedima čvora u pronalazi prvi neoznačeni sused v_1, pa se iz čvora v_1 rekurzivno pokreće obilazak u dubinu. Zatim se prelazi na sledećeg neoznačenog suseda v_2 i iz njega se rekurzivno pokreće obilazak u dubinu. Postupak (trenutni rekurzivni poziv tj. obrada čvora u) se završava kada su označeni svi susedi čvora u.

Primer 2.3.1

Razmotrimo neusmereni graf prikazan na slici 2.10: neka je on predstavljen listama susedstva tako da su čvorovi u svakoj listi navedeni leksikografski rastuće. Ako se na tom grafu pokrene obilazak u dubinu iz čvora c redom se obilaze čvorovi c, a, b, d, e (slika 2.11 levo), tako što se redom prolazi granama (c, a), (a, b), (c, d) i (d, e). Ako se obilazak u dubinu pokrene iz čvora b, obilaze se redom čvorovi b, a, c, d, e (slika 2.11 desno), tako što se redom prolazi granama (b, a), (a, c), (c, d) i (d, e).



Slika 2.10: Primer neusmerenog grafa.



Slika 2.11: Ilustracija redosleda obilaska čvorova prilikom obilaska grafa u dubinu ukoliko se obilazak pokreće iz čvorova c i b, redom. Uz čvorove su prikazani brojevi koji odgovaraju redosledu kojim se čvorovi po prvi put posećuju.

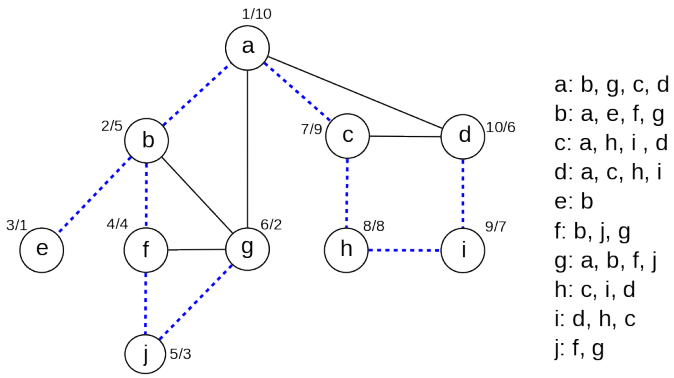
Obilazak grafa se uvek vrši sa nekim ciljem. Kako bi se različite primene uklopile u obilazak u dubinu, poseti čvora ili grane pridružuju se dve vrste obrade, *ulazna obrada* i *izlazna obrada*. Ulazna obrada vrši se u trenutku označavanja čvora. Izlazna obrada vrši se na kraju, kada su obrađeni svi susedi datog čvora. Ulazna i izlazna obrada zavise od konkretne primene algoritma DFS. Na taj način moguće je rešavanje različitih problema jednostavnim definisanjem ulazne i izlazne obrade.

Implementacija algoritma obilaska grafa u dubinu dat je u nastavku. U glavnoj funkciji obilazak se pokreće iz čvora 0. Jednostavnosti radi, graf će u narednim kodovima biti deklarisan kao globalna promenljiva (predstavljen je graf sa slike 2.10 pri čemu su čvorovi od a do e numerisani redom brojevima od 0 do 4).

```
// reprezentacija grafa listama povezanosti
// graf je neusmeren, pa se svaka grana dva puta javlja u listi
vector<vector<Cvor>> listaSuseda;

// pomocna rekurzivna funkcija koja vrši DFS obilazak grafa iz datog cvora
void dfs(int cvor, vector<bool> &posecen) {
  posecen[cvor] = true;
  // ovde ide ulazna obrada

  // rekurzivno prolazimo kroz sve njegove susede
  // koje ranije nismo obisli
  for (int sused : listaSuseda[cvor])
    if (!posecen[sused])
      dfs(sused,posecen);

  // ovde ide izlazna obrada
}

// funkcija koja vrsi DFS obilazak datog grafa iz datog cvora
void dfs(int cvor){
  int brojCvorova = listaSuseda.size();
  vector<bool> posecen(brojCvorova, false);
  dfs(cvor, posecen);
}

int main() {
  // ucitavamo neusmeren graf
  int brojCvorova;
  cin >> brojCvorova;
  listaSuseda.resize(brojCvorova);
  int brojGrana;
  cin >> brojGrana;
  for (int i = 0; i < brojGrana; i++) {
     int cvorOd, cvorDo;
     cin >> cvorOd >> cvorDo;
     listaSuseda[cvorOd].push_back(cvorDo);
     listaSuseda[cvorDo].push_back(cvorOd);
  }

  // vrsimo obilazak od ucitanog pocetnog cvora
  int pocetniCvor;
  cin >> pocetniCvor;
  dfs(pocetniCvor);

  return 0;
}
```

Primer 2.3.2

Primer obilaska grafa u dubinu prikazan je na slici 2.12. Uz svaki čvor su prikazani njegovi redni brojevi u dolaznoj, odnosno odlaznoj DFS numeraciji (dolazna numeracija označava redosled kojim se pokretala ulazna obrada čvorova, a odlazna redosled u kom se završavala obrada čvorova tj. pokretala izlazna obrada čvorova, o čemu će više reči biti u nastavku).



Slika 2.12: Primer obilaska grafa u dubinu. Graf je zadat listama povezanosti prikazanim desno, a uz svaki čvor je prikazan njegov redni broj u dolaznoj i odlaznoj DFS numeraciji.

Teorema 2.3.1

Ako je graf G povezan, onda su po završetku obilaska u dubinu svi čvorovi grafa G označeni, a sve grane grafa G su pregledane bar po jednom (tj. funkcija je pozvana za svaki čvor grafa, a u petlji `for` su analizirani svi njegovi susedi).

Dokaz. Označimo sa U skup neoznačenih čvorova nakon završetka izvršavanja algoritma. Pretpostavimo suprotno tj. pretpostavimo da je U neprazan. Pošto je graf G povezan, a skup označenih čvorova $V \setminus U$ je takođe neprazan (jer sadrži bar polazni čvor r koji se označava na početku algoritma), bar jedan čvor u iz $V \setminus U$ mora biti povezan granom sa bar jednim neoznačenim čvorom v is U. Međutim, ovako nešto je nemoguće, jer kad se poseti čvor u, moraju biti posećeni (pa dakle i označeni) svi njegovi neoznačeni susedi, dakle i čvor v. Dakle, svi čvorovi grafa moraju biti posećeni i označeni. Pošto su svi čvorovi grafa posećeni, a kad se čvor poseti, onda se pregledaju sve grane koje vode iz njega, zaključujemo da su i sve grane grafa pregledane. □

Prilikom izvršavanja algoritma DFS na neusmerenom grafu $G = (V, E)$ koji je zadat listama povezanosti, svaka grana se pregleda tačno dva puta, po jednom sa svakog kraja. Prema tome, ukupan broj izvršavanja tela petlje `for` u svim rekurzivnim pozivima algoritma DFS je $O(|E|)$. S druge strane, broj rekurzivnih poziva je $|V|$, pa se vremenska složenost algoritma DFS može opisati izrazom $O(|V| + |E|)$. Primetimo da složenost obilaska grafa u dubinu zavisi od reprezentacije grafa: naime, ako bi graf bio zadat matricom povezanosti, onda bi prolazak kroz susede jednog fiksiranog čvora podrazumevao prolaz kroz odgovarajuću vrstu matrice, što je složenosti $\Theta(|V|)$. Odatle zaključujemo da je prolazak kroz susede svakog od čvorova složenosti $\Theta(|V|^2)$. Dakle, u slučaju algoritma obilaska u dubinu, obično se efikasnija implementacija dobija korišćenjem reprezentacije grafa listama povezanosti (naročito kada je graf redak, tj. kada svaki čvor ima relativno mali broj suseda).

Prostorna složenost algoritma DFS zavisi od maksimalne dubine rekurzije. U najgorem slučaju graf može imati oblik dugog puta i u tom slučaju je dubina rekurzije proporcionalna broju čvorova u grafu. Dakle, prostorna složenost algoritma DFS je u najgorem slučaju $O(|V|)$.

Komponente povezanosti

Algoritam DFS se mora prilagoditi da bi korektno radio i u slučaju nepovezanih grafova. Naime, ako su posle prvog pokretanja opisanog algoritma svi čvorovi označeni, onda je graf povezan, i obilazak je završen. U protivnom, može se pokrenuti novi obilazak u dubinu polazeći od proizvoljnog neoznačenog čvora u grafu, itd. Prema tome, algoritam obilaska u dubinu se može iskoristiti kako bi se ustanovilo da li je graf povezan, odnosno za pronalaženje svih njegovih komponenti povezanosti.

> **Problem**
>
> *Definisati algoritam koji određuje sve komponente povezanosti neusmerenog grafa tj. koji svim komponentama povezanosti dodeljuje jedinstveni identifikator tako da su svi čvorovi unutar komponente obeleženi identifikatorom te komponente.*

Algoritam se lako implementira ako se pokuša pokretanje obilaska u dubinu redom iz svakog čvora grafa: za svaki čvor se proverava da li je posećen tokom obilaska ranijih čvorova (čime je već pridružen nekoj od ranijih komponenata) i ako jeste, preskače se pokretanje obilaska iz tog čvora.

U ovom i u narednim kodovima, jednostavnosti radi, pretpostavljamo da je graf predstavljen globalnom promenljivom `listaSuseda`.

```
// obilazak u dubinu iz cvora cvor
void dfs(int cvor, int brojKomponente, vector<int>& komponente) {
  // tekuci cvor pridruzujemo tekucoj komponenti
  komponente[cvor] = brojKomponente;

  // rekurzivno prolazimo kroz sve njegove susede
  // koje ranije nismo obisli
  for (int sused : listaSuseda[cvor])
    if (komponente[sused] == -1)
      dfs(sused, brojKomponente, komponente);
}

// za svaki cvor grafa se u vektor komponente upisuje
// redni broj komponente kojoj on pripada;
// funkcija vraca ukupan broj komponenata povezanosti
int komponentePovezanosti(vector<int> &komponente) {
  int brojCvorova = listaSuseda.size();
  komponente.resize(brojCvorova, -1);
  int brojKomponente = 0;
  // pokrecemo novi DFS obilazak iz prvog neposecenog cvora
  for (int cvor = 0; cvor < brojCvorova; cvor++)
    if (komponente[cvor] == -1) {
      dfs(cvor, brojKomponente, komponente);
      brojKomponente++;
    }
  return brojKomponente;
}

int ispisiKomponente() {
  // odredjujemo komponente povezanosti
  int brojCvorova = listaSuseda.size();
  vector<int> komponente;
  int brojKomponenti = komponentePovezanosti(komponente);
```

```
  // ispisujemo rezultat
  cout << "Ukupan broj komponenti povezanosti je "
       << brojKomponenti << endl;
  for (int i = 0; i < brojCvorova; i++)
    cout << "Cvor " << i << " pripada komponenti "
         << komponente[i] << endl;
  return 0;
}
```

Iako se pretraga u dubinu može pozvati i više puta, i ovaj algoritam je vremenske složenosti $O(|V| + |E|)$. Naime, obilazak komponente broj k biće složenosti $O(|V_k| + |E_k|)$, gde je V_k broj čvorova, a E_k broj grana unutar te komponente. Zato se ukupno, tokom svih obilazaka zajedno obiđe $O(|V|)$ čvorova i $O(|E|)$ grana.

Mi ćemo najčešće razmatrati slučaj kada je graf povezan, jer se u opštem slučaju problem svodi na posebnu obradu svake komponente povezanosti.

Iterativne implementacije pomoću steka

Iako je jednostavna za razumevanje, mana rekurzivne implementacije je to što kod nekih grafova može doći do prekoračenja sistemske stek-memorije prilikom izvršavanja programa (to se može desiti ako postoji dugačak niz čvorova kojim se prolazi bez vraćanja unazad). Naime, rekurzivni pozivi koriste sistemski stek računara, na koji se smeštaju parametri aktuelnih rekurzivnih poziva (argumenti prosleđeni prilikom svakog od rekurzivnih poziva i vrednosti lokalnih promenljivih). Umesto sistemskog steka koji i na savremenim sistemima predstavlja veoma ograničen deo memorije (obično je u pitanju tek nekoliko megabajta) možemo u našem programu održavati poseban stek, koji može da zauzme mnogo više memorijskog prostora, jer se alocira u zonama kojima je pridruženo mnogo više memorije (na primer, u zoni hipa). Stoga je ponekad potrebno napraviti implementaciju obilaska u dubinu koja nije rekurzivna, a koja, kao što je to i inače često slučaj kod oslobađanja od rekurzije, zahteva korišćenje strukture podataka stek.

Verna varijanta: oslobađanje od rekurzije

Možemo primetiti da se tokom rekurzivnih poziva na sistemskom steku u svakom trenutku nalaze svi čvorovi na trenutnom putu od početnog do tekućeg čvora. Te informacije su nam potrebne zbog vraćanja unatrag. Naime, kada iz tekućeg čvora ne možemo preći dalje u neki neposećen čvor, čvor koja se na steku nalazi ispod tekućeg je onaj iz kojeg smo došli u tekući čvor i u koji treba da se vratimo (to je njegov roditeljski čvor). Prilikom izvršavanja rekurzivne funkcije, u svakom čvoru se izvršava petlja u kojoj se prolazi kroz neposećene susede tog čvora. Prilikom povratka u prethodni čvor potrebno je izvršiti narednu iteraciju te petlje. Stoga se u sistemskom stek okviru koji odgovara čvoru čuva i trenutna vrednost brojačke promenljive u sklopu te petlje (kako bi se nakon povratka u čvor ta promenljiva mogla uvećati i tako preći na naredni korak iteracije, tj. na narednog suseda). Naravno, sve se ovo dešava automatski (na sistemskom steku se čuvaju vrednosti svih lokalnih promenljivih), pa programer ne mora da vodi računa o tome. U sklopu nerekurzivne implementacije možemo imitirati ovaj postupak, tako što na steku koji ručno održavamo čuvamo uređen par koji sadrži oznaku roditeljskog čvora i brojač u petlji u tom roditeljskom čvoru tj. oznaku deteta tog roditelja koje je trenutno obrađeno.

```
// obilazak grafa u dubinu iz datog pocetnog cvora
void dfs(int pocetniCvor) {
   int brojCvorova = listaSuseda.size();
   vector<bool> posecen(brojCvorova, false);
   // na steku pamtimo roditeljski cvor u koji treba da se vratimo
   // i broj do sada obrađene dece u tom roditeljskom čvoru
   stack<pair<int, int>> stek;
   // prvi roditelj je pocetni cvor i za sada nije obrađeno
```

```
    // ni jedno dete
    stek.emplace(pocetniCvor, 0);
    // ovo je prvi put da smo otkrili pocetni cvor, pa vrsimo njegovu
    // ulaznu obradu
    posecen[pocetniCvor] = true;
    cout << pocetniCvor << endl; // ulazna obrada
    // dok se stek ne isprazni
    while (!stek.empty()) {
        // skidamo roditeljski cvor sa steka
        auto [roditelj, i] = stek.top();
        stek.pop();
        // ako nisu još obrađena sva njegova deca
        if (i < listaSuseda[roditelj].size()) {
            // obradjujemo naredno dete

            // pamtimo na steku da je broj obradjene dece ovog roditelja
            // uvećan za 1, tj. da kada se sledeci put u povratku vratimo
            // do ovog roditelja, da tada preskocimo trenutno dete
            stek.emplace(roditelj, i+1);
            // ako trenutno dete nije vec poseceno, sada ga posecujemo i
            // vrsimo njegovu ulaznu obradu
            int dete = listaSuseda[roditelj][i];
            if (!posecen[dete]) {
                posecen[dete] = true;
                cout << dete << endl; // ulazna obrada
                stek.emplace(dete, 0);
            }
        }
    }
}
```

Implementacija dobijena na ovaj način je jedina koja verno oslikava rekurzivni obilazak u dubinu. Moguće su i drugačije implementacije u kojima je kôd malo jednostavniji, međutim, videćemo da se svaka od njih po nečemu razlikuje od osnovne rekurzivne formulacije.

Pseudo DFS

Razmotrimo sada jednu moguću modifikaciju obilaska grafa implementiranog uz pomoć steka. Da ne bismo morali da prekidamo petlju i nastavljamo njeno izvršavanje posle prekida, možemo program organizovati tako da na stek odmah stavimo sve susede tekućeg čvora i da ih odmah označimo. Oznaka sada znači da je čvor stavljen na stek, tj. da je zakazano da će čvor u nekom narednom trenutku biti posećen, a ne da je već posećen i obrađen. Dakle, na steku čuvamo čvorove koje u budućnosti treba obraditi (kada ih stavimo na stek, smatramo da smo zakazali njihovu obradu). Čvor obično obrađujemo u trenutku kada ga skidamo sa steka. Nakon toga na stek stavljamo njegove susede za koje još nije zakazan obilazak. Opisani postupak izložen je u algoritmu 3.

To što se čvor označava čim se prvi put stavi na stek, a ne tek kada se skida sa steka nam garantuje da će svaki čvor najviše jednom biti upisan na stek. Ipak ovo može uticati na promenu redosleda obilaska čvorova u odnosu na rekurzivnu implementaciju, tako da ovaj algoritam nije u striktnom smislu obilazak grafa u dubinu (zato se ponekad naziva *pseudo DFS*). Ipak, u većini primena dovoljno je samo da se garantuje da će se svaki čvor obići tačno jednom, pa ovaj algoritam često zadovoljava sve naše potrebe.

Čak iako se u ovoj varijanti svaki čvor stavlja samo jednom na stek, dubina steka (pa samim tim i potrošnja

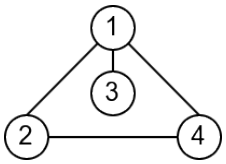
Algoritam 3 Obilazak u dubinu – formulacija sa stekom

```
1: procedure DFS(čvor r)
2:     stavi početni čvor r na stek
3:     while stek nije prazan do
4:         skini čvor u sa vrha steka
5:         izvrši ulaznu obradu čvora u
6:         for all sused v čvora u do
7:             if čvor v nije označen then          ▷ tj. nije mu zakazan obilazak
8:                 stavi čvor v na stek
9:                 označi čvor v                    ▷ tj. zakazuje mu se obilazak
```

memorije) potrebna za izvršavanje algoritma može biti znatno veća nego što je slučaj sa prethodnom, vernom varijantom obilaska u dubinu. Naime, u vernoj varijanti obilaska u dubinu je broj elemenata na steku asimptotski jednak dužini najdužeg puta od početnog do nekog udaljenog čvora iz kog ne možemo dalje da se krećemo, dok se u pseudo DFS obilasku na steku istovremeno nalaze i svi susedi tih čvorova. Na primer, ako razmotrimo graf oblika zvezde u kom je jedan centralni čvor povezan sa N suseda, u vernoj varijanti bi se na stek stavio centralni čvor, a zatim bi se na stek stavljao, pa skidao jedan po jedan njegov sused, tako da je za obilazak dovoljan stek dubine 2. U pseudo DFS algoritmu bi, međutim, nakon centralnog čvora na stek odmah bilo dodato svih N suseda centralnog čvora, te je za obilazak potreban stek dubine N (naravno, u najgorem slučaju oba algoritma zahtevaju prostor $O(N)$).

Primer 2.3.3

Razmotrimo graf na slici 2.13. Neka je zadat listama povezanosti i neka su liste povezanosti sortirane u rastućem redosledu suseda.



Slika 2.13: Graf kod kojeg se razlikuje redosled obilaska čvorova prilikom rekurzivnog obilaska u dubinu i obilaska algoritmom pseudo DFS.

Prilikom rekurzivnog obilaska u dubinu krenuvši od čvora 1*, rekurzivno se prelazi na čvor* 2*, pa zatim na čvor* 4*. Pošto čvor* 4 *nema daljih neposećenih suseda vraćamo se nazad, sve do čvora* 1 *i nakon toga se posećuje čvor* 3*. Redosled obilaska i obrade čvorova je, dakle,* $1, 2, 4, 3$.

Prilikom pseudo DFS obilaska istog grafa iz čvora 1 *na stek se najpre stavlja čvor* 1*. Zatim se taj čvor skida sa steka, obrađuje se i na stek se stavljaju njegovi susedi* 2*,* 3 *i* 4*. Zatim se skida i obrađuje čvor* 2*. On nema neposećenih suseda. Skida se zatim i obrađuje čvor* 3*, koji takođe nema neposećenih suseda. Na kraju se skida i obrađuje čvor* 4*, koji takođe nema neposećenih suseda. Redosled obilaska i obrade je, dakle,* $1, 2, 3, 4$*, što je različito od stvarnog redosleda obilaska u dubinu koji zahteva da se iz čvora* 2 *pređe u do tada neposećeni čvor* 4.

Teorema 2.3.2

Ako je graf povezan, onda su po završetku algoritma 3 svi čvorovi obrađeni (za svaki čvor je izvršena ulazna obrada).

Dokaz. Dokaz teče slično dokazu teoreme 2.3.1. Označimo sa U skup čvorova koji nisu obrađeni nakon završetka izvršavanja algoritma. Pretpostavimo suprotno pretpostavci da postoji neki neobrađeni čvor tj.

pretpostavimo da je skup U neprazan. Pošto je graf G povezan, a skup obrađenih čvorova $V \setminus U$ je takođe neprazan (jer sadrži bar polazni čvor r koji se obrađuje u prvoj iteraciji petlje), bar jedan obrađeni čvor u iz $V \setminus U$ mora biti povezan granom sa bar jednim neobrađenim čvorom v is U. Međutim, ovako nešto je nemoguće, jer nakon što se obradi čvor u, označavaju se svi njegovi do tada neoznačeni susedi, pa i čvor v. Čvor v je, dakle, morao biti označen i stavljen na stek, a svi čvorovi koji su stavljeni na stek se u nekom trenutku obrađuju, pa je i čvor v morao biti obrađen, što je kontradikcija. Dakle, svi čvorovi grafa moraju biti obrađeni. □

Na osnovu algoritma 3 veoma je jednostavno napraviti implementaciju u jeziku C++ (stoga je nećemo prikazivati).

Višestruko stavljanje čvorova na stek

Alternativa prethodnom algoritmu je da se označavanje čvorova vrši u trenutku kada se zaista posete tj. skinu sa steka, kao što je prikazano u opštem algoritmu za obilazak grafova na početku ovog poglavlja. Ovaj pristup opisan je u algoritmu 4.

Algoritam 4 Obilazak u dubinu – formulacija sa stekom

```
1: procedure DFS(čvor r)
2:     stavi početni čvor r na stek
3:     while stek nije prazan do
4:         skini čvor u sa vrha steka
5:         if čvor u nije označen then
6:             označi čvor u
7:             izvrši ulaznu obradu čvora u
8:             for all sused v čvora u do
9:                 stavi čvor v na stek
```

Pod pretpostavkom da se susedi na stek postavljaju u obratnom redosledu od onog u kom su navedeni u listi povezanosti, time se dobija isti redosled obilaska kao u rekurzivnoj implementaciji tj. u pitanju je zaista obilazak u dubinu. Mana je to što se isti elementi mogu više puta naći na steku, tako da je ponekad potrebna značajno veća dubina steka da se ceo graf obiđe. Naime, iako se na stek stavljaju samo neoznačeni čvorovi, oni se ne označavaju prilikom stavljanja već tek prilikom skidanja sa steka. Veoma je važno da se susedi označenih čvorova ne analiziraju kada se ti čvorovi skinu sa steka, jer bi se u suprotnom za isti čvor više puta prolazilo kroz liste suseda.

I ovu implementaciju je veoma jednostavno napraviti na osnovu datog algoritma, pa je nećemo prikazivati.

Primetimo da je izlaznu obradu čvora teže implementirati u (svim) nerekurzivnim implementacijama. Naime, izlaznu obradu tekućeg čvora je potrebno izvršiti tek kada sa steka bude skinut i poslednji njegov sused. Jedan način da se taj trenutak uoči je da se na stek pre suseda tekućeg čvora stavi specijalna oznaka čije skidanje sa steka ukazuje da je potrebno izvršiti izlaznu obradu tekućeg čvora.

Primetimo da je prostorna složenost nerekurzivnih verzija algoritma DFS jednaka maksimalnom broju čvorova koji se tokom algoritma nađu u steku, što je opet u najgorem slučaju $O(|V|)$, kao i u rekurzivnoj implementaciji.

Konstrukcija DFS drveta i DFS numeracija

Prikazaćemo sada dve jednostavne a važne primene algoritma DFS — formiranje specijalnog povezujućeg drveta, takozvanog *DFS drveta* i numeraciju čvorova grafa *DFS brojevima*. Tekst u nastavku se odnosi na rekurzivni obilazak.

Prilikom obilaska grafa G u dubinu, u petlji kojom se prolaze svi susedi čvora u mogu se izdvojiti sve grane ka novooznačenim čvorovima v. Preko izdvojenih grana dostižni su svi čvorovi povezanog neusmerenog grafa, pa je podgraf koga čine izdvojene grane povezan. Taj podgraf nema cikluse, jer se od svih grana

koje vode u neki čvor, izdvaja samo jedna. Prema tome, izdvojene grane su grane podgrafa grafa G koji je u slučaju povezanog grafa povezujuće drvo, koje nazivamo *DFS drvo* grafa G. Čvor iz koga se pokreće obilazak u dubinu je koren DFS drveta. Čak i ako se drvo ne formira eksplicitno, mnoge algoritme je lakše razumeti razmatrajući DFS drvo grafa G.

Problem

Definisati algoritam koji u neusmerenom grafu konstruiše DFS drvo tj. koji određuje sve grane tog drveta.

U nastavku je dat program koji konstruiše DFS drvo datog grafa predstavljenog listama susedstva. DFS drvo biće predstavljeno u vidu vektora grana grafa.

```
// rekurzivna funkcija koja vrsi DFS obilazak grafa od datog cvora  i
// uz to konstruise DFS drvo
void konstruisi_dfs_drvo(int cvor, vector<bool> &posecen,
                         vector<vector<int>> &dfs_drvo) {
  posecen[cvor] = true;

  // rekurzivno prolazimo kroz sve njegove susede
  // koje ranije nismo obisli
  for (int sused : listaSuseda[cvor]) {
    if (!posecen[sused]) {
      // u DFS drvo dodajemo granu iz tekuceg ka novom cvoru
      // i njoj suprotno usmerenu granu (jer je graf neusmeren)
      dfs_drvo[cvor].push_back(sused);
      dfs_drvo[sused].push_back(cvor);
      konstruisi_dfs_drvo(sused, posecen, dfs_drvo);
    }
  }
}

// funkcija koja vrsi DFS obilazak datog grafa iz datog cvora
// i konstruise i ispisuje DFS drvo
void ispisi_dfs_drvo(int cvor) {
  int brojCvorova = listaSuseda.size();
  vector<bool> posecen(brojCvorova, false);
  vector<vector<int>> dfs_drvo(brojCvorova);
  konstruisi_dfs_drvo(cvor, posecen, dfs_drvo);

  // stampamo grane grafa koje pripadaju DFS drvetu
  cout << "Grane DFS drveta su: ";
  for (int i = 0; i < dfs_drvo.size(); i++)
    for (int j = 0; j < dfs_drvo[i].size(); j++)
      cout << "(" << i << "," << dfs_drvo[i][j] << ")" << endl;
}
```

Postoje dve varijante DFS numeracije čvorova:

- *dolazna DFS numeracija*, odnosno *preOrder* numeracija, kojom se čvorovi numerišu prema redosledu označavanja i
- *odlazna DFS numeracija*, odnosno *postOrder* numeracija, kod koje se čvorovi numerišu prema redosledu napuštanja.

Dolazni broj čvora v obeležavaćemo sa $v.Pre$, a odlazni sa $v.Post$. Dolazna i odlazna numeracija čvorova imaju primenu prilikom rešavanja raznih problema nad grafovima.

Primer 2.3.4

Primer grafa sa čvorovima numerisanim na dva načina prikazan je na slici 2.12.

Za primer sa slike 2.12 redosled čvorova u rastućem redosledu dolazne numeracije glasi $a, b, e, f, j, g, c, h, i, d$, *dok je redosled čvorova u rastućem redosledu odlazne numeracije* $e, g, j, f, b, d, i, h, c, a$.

Naredna funkcija ispisuje čvorove u rastućem redosledu njihove dolazne DFS numeracije.

```
void dfs_preorder(int cvor, vector<bool> &posecen) {
  posecen[cvor] = true;
  // stampamo naredni cvor u preOrder numeraciji
  cout << cvor << " ";

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor])
    if (!posecen[sused])
      dfs_preorder(sused,posecen);
}
```

Naredna funkcija ispisuje čvorove u rastućem redosledu njihove odlazne DFS numeracije.

```
void dfs_postorder(int cvor, vector<bool> &posecen) {
  posecen[cvor] = true;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor])
    if (!posecen[sused])
      dfs_postorder(sused,posecen);

  // stampamo naredni cvor u postOrder numeraciji
  cout << cvor << " ";
}
```

Alternativno, možemo implementirati funkcije kojima se za sve čvorove u grafu pamte, a na kraju ispisuju redni brojevi u dolaznoj i odlaznoj DFS numeraciji.

```
// za svaki cvor se pamti dolazni i odlazni redni broj
vector<int> dolazna;
vector<int> odlazna;
// naredni slobodan redni broj (odlazni i dolazni)
int rbr_dolazna = 0;
int rbr_odlazna = 0;

void dfs(int cvor, vector<bool> &posecen) {
  posecen[cvor] = true;

  // cvor dobija naredni dolazni redni broj
  dolazna[cvor] = rbr_dolazna;
```

```
  rbr_dolazna++;

  // rekurzivno prolazimo kroz sve susede koje do sada nismo obisli
  for (int sused : listaSuseda[cvor]) {
    if (!posecen[sused]) {
      dfs(sused,posecen);
    }
  }
  // cvor dobija naredni odlazni redni broj
  odlazna[cvor] = rbr_odlazna;
  rbr_odlazna++;
}

void dfs(int cvor) {
  int brojCvorova = listaSuseda.size();
  dolazna.resize(brojCvorova, -1);
  odlazna.resize(brojCvorova, -1);
  vector<bool> posecen(brojCvorova, false);

  dfs(cvor,posecen);

  cout << "Dolazna i odlazna numeracija cvorova:" << endl;
  for (int i = 0; i < brojCvorova; i++)
    cout << "Cvor " << i << ": " << dolazna[i]
         << "/" << odlazna[i] << endl;
}
```

U odnosu na proizvoljni čvor u drvetu, ostale čvorove možemo podeliti u četiri grupe: čvorovi koji su njegovi preci, čvorove koji su njegovi potomci, čvorove koji su levo od njega i čvorove koji su desno od njega (videti primer na slici 2.14).

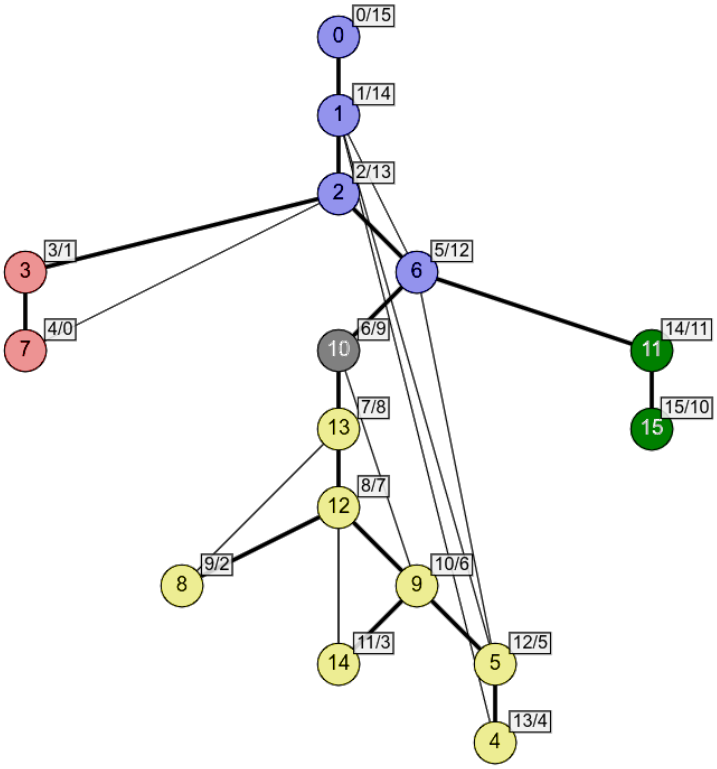
Definišimo precizno ove odnose.

Čvor w nazivamo *pretkom* čvora v u DFS drvetu T sa korenom r (slika 2.15) ako je w na jedinstvenom putu od r do v u T. Ako je čvor w predak čvora v, onda je čvor v *potomak* čvora w.

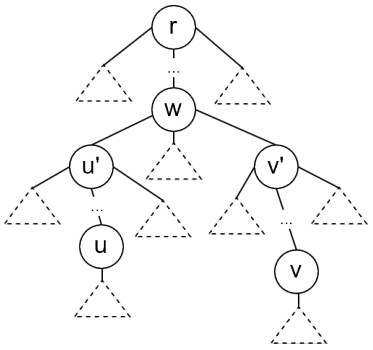
DFS drvo obuhvata sve čvorove povezanog grafa G. Redosled dece svakog čvora u drvetu određen je listama povezanosti kojima se zadaje graf G, pa se za svaka dva deteta istog čvora može reći koji je od njih *levi* (prvi po tom redosledu), a koji *desni*. Relacija levi – desni se prenosi na proizvoljna dva čvora u i v koji nisu u relaciji predak – potomak (slika 2.15). Za čvorove u i v tada postoji "najmlađi" zajednički predak w u DFS drvetu (to je onaj zajednički predak čiji nijedan potomak nije zajednički predak čorova u i v), kao i deca u' i v' čvora w takvi da je čvor u' predak čvora u i čvor v' predak čvora v. Kažemo da je *čvor u levo od čvora v* ako i samo ako je čvor u' levo od čvora v'. Geometrijska interpretacija ove relacije je jasna: DFS drvo ćemo iscrtavati naniže prilikom prelaska u nove – neoznačene čvorove (koraci u dubinu), odnosno sleva udesno prilikom dodavanja novih grana posle povratka u već označene čvorove.

Relacije predak–potomak i levi–desni se mogu okarakterisati i korišćenjem dolazne i odlazne numeracije čvorova.

Redosled obrade čvorova je odozgo-naniže, sleva-nadesno, pa se pre bilo kog čvora obeležavaju čvorovi levo od njega i čvorovi iznad njega (njegovi preci), a posle njega se obeležavaju čvorovi ispod njega (njegovi potomci) kao i čvorovi desno od njega. Dakle, ako su data dva čvora u i v takva da je $u.Pre < v.Pre$, tada je u ili levo od v ili je predak čvora v. Kako bismo znali koji od ta dva slučaja važi, potrebno je da



Slika 2.14: U odnosu na čvor broj 10, crveni čvorovi su levo, zeleni desno, plavi su preci, a žuti su potomci. Čvorovi levo imaju manje i dolazne i odlazne redne brojeve, preci imaju manje dolazne, a veće odlazne redne brojeve, potomci imaju veće dolazne, a manje odlazne redne brojeve, dok čvorovi desno imaju veće i dolazne i odlazne redne brojeve.



Slika 2.15: Ilustracija relacije *levo – desno* na skupu čvorova DFS drveta.

uporedimo odlazne redne brojeve: manji odlazni brojevi ukazuju na čvorove levo, a veći na pretke. Sledeća lema nam daje potpunu karakterizaciju odnosa položaja čvorova na osnovu njihove DFS numeracije (videti primer na slici 2.14).

Lema 2.3.1 **[DFS numeracija i odnos položaja čvorova]**

Ako za dva čvora u i v važi $u.Pre < v.Pre$, tada važi:

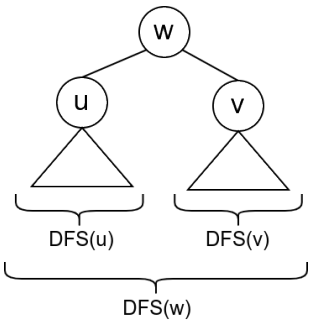
- *ako je $u.Post < v.Post$, tada je u levo od v (obrada čvorova levo ranije počinje i ranije se završava);*
- *ako je $u.Post > v.Post$, tada je u predak od v (obrada predaka ranije počinje, ali se kasnije završava).*

Iz ovoga direktno slede i sledeći (dualni) uslovi.

Ako za dva čvora u i v važi $u.Pre > v.Pre$, tada važi:

- *ako je $u.Post < v.Post$, tada je u potomak čvora v (obrada potomaka kasnije počinje, ali se ranije završava);*
- *ako je $u.Post > v.Post$, tada je u desno od v (obrada čvorova desno kasnije počinje i kasnije se završava).*

Dokaz. Na slici 2.16 prikazana su tri čvora grafa u, v i w u okviru DFS drveta grafa. Čvorovi u i v su deca čvora w, a čvor u je levo od čvora v. Na slici su prikazani i vremenski intervali trajanja rekurzivnih poziva algoritma DFS za svaki od ovih čvorova. Prvo se pokreće DFS obilazak čvora w, nakon toga se pokreće DFS obilazak čvora u, zatim se taj obilazak završava, pa se poziva DFS obilazak čvora v, nakon toga se taj obilazak završava i na kraju se završava DFS obilazak čvora w.



Slika 2.16: Odnos između položaja čvorova u DFS drvetu i trajanja rekurzivnih poziva pokrenutih iz ovih čvorova.

Bilo koji potomak čvora w (recimo w') se nalazi unutar poddrveta sa korenom čiji je koren neko dete čvora w (to su u, v, ...), pa mora da ima veći dolazni DFS broj od dolaznog broja čvora w tj. važi $w'.Pre > w.Pre$. Zapažamo da je DFS algoritam pokrenut iz čvora bilo kog potomka čvora w, aktivan samo u podintervalu vremena za koje je aktivan DFS algoritam iz čvora w. DFS pretraga iz bilo kog čvora w' koji je potomak čvora w završava se pre završetka DFS pretrage iz čvora w. Prema tome, iz činjenice da je neki čvor w' potomak čvora w sledi da je $w'.Post < w.Post$.

Ako je neki čvor u' levo od nekog čvora v', onda u' mora biti u poddrvetu čiji je koren u, a v' u poddrvetu čiji je koren v (za neko u, v koji imaju zajedničkog roditelja w i u je levo od v). DFS obilazak za svaki čvor u poddrvetu sa korenom u i počinje i završava se pre DFS obilaska bilo kog čvora u poddrvetu sa korenom v. Zato je $u'.Pre < v'.Pre$ i $u'.Post < v'.Post$. □

Primetimo da prethodna lema daje potpunu karakterizaciju odnosa položaja čvorova u drvetu (postoji četiri moguća odnosa položaja i četiri odnosa parova rednih brojeva u dolaznoj i odlaznoj numeraciji) – u sva četiri slučaja važe ekvivalencije (odnos položaja važi ako i samo ako važi navedeni odnos rednih brojeva).

Na slikama 2.12 i 2.14 se vidi da sve grane neusmerenog grafa spajaju pretke i potomke tj. da grane ne mogu biti *poprečne* u odnosu na DFS drvo, odnosno ne povezuju čvorove na razdvojenim putevima od korena (tj. dva čvora koja su u odnosu levo – desno). Zaista, ako bi takva grana postojala, DFS algoritam bi "prošao" njome i uključio je u DFS drvo. Naredna lema precizno izražava ovu karakterizaciju grana grafa u odnosu na DFS drvo.

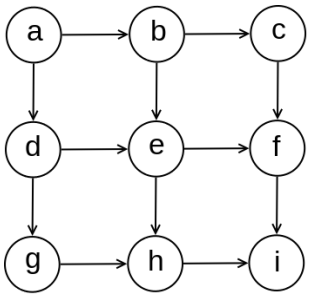
Lema 2.3.2 **[Karakterizacija grana neusmerenog grafa u odnosu na DFS drvo]**
Neka je $G = (V, E)$ povezan neusmeren graf i neka je $T = (V, F)$ DFS drvo grafa G. Svaka grana grafa $e \in E$ pripada drvetu T (tj. $e \in F$) ili spaja dva čvora grafa G, od kojih je jedan predak drugog u drvetu T.

Dokaz. Neka je (u, v) grana neusmerenog grafa G, i pretpostavimo da je u toku DFS pretrage čvor u posećen pre čvora v tj. da je $u.Pre < v.Pre$. Na osnovu leme 2.3.1 važi da je v ili potomak čvora u ili je desno od u. Posle označavanja čvora u, u petlji se rekurzivno pokreće DFS pretraga iz svakog neoznačenog suseda čvora u. U trenutku kad dođe red na čvor v, ako v nije označen, iz v se započinje novi rekurzivni poziv DFS, pa je v dete čvora u u drvetu T i grana (u, v) pripada DFS drvetu T. Ako je v označen, onda je v potomak čvora u u drvetu T. Zaista, pošto je v označen, njegova obrada mora biti započeta i završena tokom trenutnog rekurzivnog poziva u kom se obrađuje čvor u, pa važi $v.Pre > u.Pre$ i $v.Post < u.Post$. Na osnovu leme 2.3.1 važi da čvor u mora biti predak čvora v. □

2.3.1.2 Usmereni grafovi

Procedura pretrage u dubinu usmerenih grafova ista je kao za neusmerene grafove.

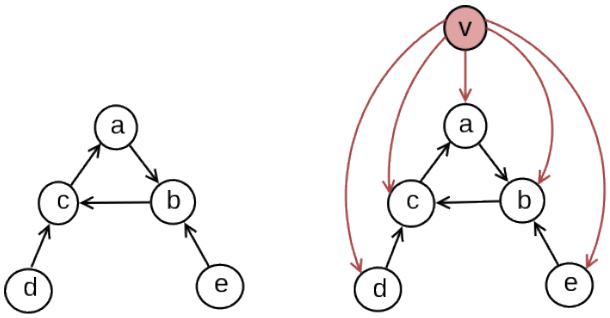
Algoritam DFS za povezan neusmereni graf, započet iz proizvoljnog čvora, obilazi ceo graf. Analogno tvrđenje ne mora biti tačno za usmerene grafove. Preciznije, ovo tvrđenje će važiti samo ako je graf jako povezan (o čemu će biti više reči u poglavlju 2.6). Posmatrajmo usmereni graf na slici 2.17. Ako se DFS pretraga započne iz čvora c, onda će biti dostignuti samo čvorovi u desnoj koloni (c, f i i). DFS pretraga može da dostigne sve čvorove grafa samo ako se započne iz čvora a. Ako se čvor a ukloni iz grafa zajedno sa dve grane koje izlaze iz njega, onda u datom grafu ne postoji čvor iz koga algoritam DFS obilazi ceo graf. Prema tome, uvek kad govorimo o DFS obilasku usmerenog grafa, smatraćemo da je algoritam DFS pokrenut onoliko puta koliko je potrebno da bi svi čvorovi bili označeni i sve grane bile razmotrene. Dakle, u opštem slučaju usmereni graf umesto DFS drveta ima *DFS šumu*.



Slika 2.17: Primer kad pretraga u dubinu usmerenog grafa ako se pokrene iz proizvoljnog čvora ne obilazi sve čvorove grafa.

Jedinstveno DFS drvo se može obezbediti tako što se u graf doda novi čvor v ulaznog stepena 0, koji se poveže granama sa svim čvorovima grafa G (slika 2.18) – tada DFS obilazak pokrenut iz čvora v sigurno obilazi sve čvorove grafa G i postoji jedinstveno DFS drvo.

Čak i kada su jedinstvena za ceo graf, usmerena DFS drveta imaju nešto drugačije osobine od DFS drveta za neusmerene grafove.

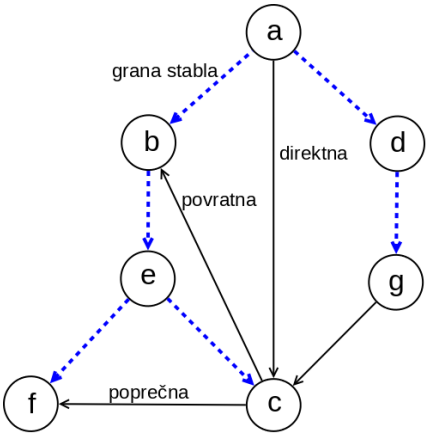


Slika 2.18: Primer dodavanja novog čvora v u graf tako da se iz čvora v graf može u potpunosti obići.

I u DFS šumi kod usmerenih grafova važe iste vrste odnosa između čvorova kao i kod neusmerenih grafova (odnosi predak-potomak i levo-desno) i lema 2.3.1 koja daje karakterizaciju ovih odnosa na osnovu dolazne i odlazne DFS numeracije ostaje na snazi. Naime, odnos trajanja rekurzivnih poziva prikazan na slici 2.16 i dalje važi. Iako to nije pokazano na slici 2.16, isti zaključak je tačan i ako su čvorovi v i w u različitim drvetima DFS šume, pri čemu je drvo čvora v levo od drveta čvora w.

Međutim, karakterizacija grana je drugačija. Na primer, nije više tačno da graf ne može imati poprečne grane, što se može videti iz primera na slici 2.19. U odnosu na DFS drvo grane usmerenog grafa pripadaju jednoj od četiri kategorije:

- *grane DFS drveta*,
- *povratne*,
- *direktne* i
- *poprečne* grane.



Slika 2.19: DFS drvo usmerenog grafa. Klasifikacija grana u odnosu na DFS drvo.

Prve tri vrste grana povezuju dva čvora od kojih je jedan potomak drugog u DFS drvetu (slika 2.19):

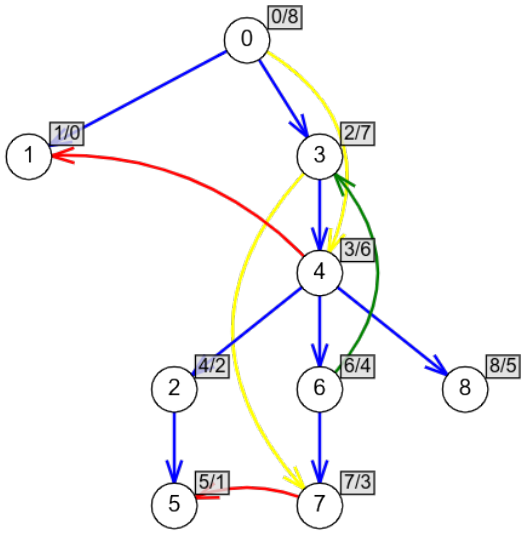
- grana DFS drveta povezuje roditelja sa detetom,
- povratna grana povezuje potomka sa pretkom,
- direktna grana povezuje pretka sa potomkom.

Ove vrste grana postoje i kod neusmerenih grafova, jedino što se ne pravi razlika između direktnih i povratnih grana (one spajaju pretke i potomke, ali nisu usmerene, pa ih ne možemo razlikovati).

Novina su poprečne grane, koje povezuju čvorove koji nisu "srodnici" u drvetu. Poprečne grane, međutim, moraju biti usmerene "zdesna ulevo" (od čvorova sa većim ka čvorova sa manjim dolaznim brojevima), kao što pokazuje sledeća lema. Naime, kod neusmerenog grafa DFS algoritam uvek može "preći" preko poprečne grane, dok kod usmerenog grafa može preći preko grana usmerenih sleva nadesno (zato one postaju

grane grafa i nikad nisu poprečne).

I karakterizaciju grana usmerenog grafa u odnosu na DFS drvo moguće je izvršiti na osnovu dolazne i odlazne numeracije čvorova grafa. U nastavku ćemo dokazati niz lema koje doprinose konačnoj klasifikaciji. Na slici 2.20 ilustrovana je veza vrednosti dolaznih i odlaznih rednih brojeva krajeva grane i njenog tipa.



Slika 2.20: Klasifikacija grana usmerenog grafa pomoću DFS numeracije. Grane od predaka ka potomcima su ili grane grafa (plave) ili direktne grane (žute) i one su usmerene od čvorova sa manjim ka čvorovima sa većim dolaznim rednim brojevima. Povratne grane (zelene) su usmerene ka čvorovima sa manjim dolaznim, ali većim odlaznim rednim brojevima. Poprečne grane (crvene) su usmerene nalevo, ka čvorovima sa manjim i dolaznim i odlaznim rednim brojevima.

Lema 2.3.3 **[Karakterizacija grana od predaka ka potomcima]**

Neka je $G = (V, E)$ usmereni graf i neka je $T = (V, T)$ DFS drvo grafa G. Ako je $(u, v) \in E$ usmerena grana grafa G od u do v za koju važi $u.Pre < v.Pre$, onda je čvor u predak čvora v u drvetu T i grana od u do v je ili grana drveta ili direktna grana.

Dokaz. Pošto prema dolaznoj DFS numeraciji čvor u prethodi čvoru v, čvor v je označen posle čvora u. Grana grafa (u, v) mora biti razmatrana u toku rekurzivnog poziva algoritma DFS iz čvora u. Ako u tom trenutku čvor v nije označen, onda se grana (u, v) mora uključiti u DFS drvo, tj. $(u, v) \in T$, pa je tvrđenje leme tačno. U protivnom, čvor v je označen u toku izvođenja nekog rekurzivnog poziva DFS iz u, pa je v potomak čvora u u DFS drvetu T i grana je direktna. □

Nepostojanje poprečnih grana u odnosu na DFS drvo koje idu sleva udesno pomaže da se numeracija čvorova grafa upotrebi za klasifikaciju četiri vrste grana u odnosu na DFS drvo.

Dokažimo sledeću karakterizaciju povratnih grana.

Lema 2.3.4 **[Karakterizacija povratnih grana]**

Grana (u, v) usmerenog grafa $G = (V, E)$ je povratna u odnosu na DFS drvo grafa ako i samo ako prema odlaznoj numeraciji čvor u prethodi čvoru v, odnosno $u.Post < v.Post$ (specijalno, ako je grana (u, v) petlja, tj. ako je $u = v$, tada važi $u.Post = v.Post$).

Dokaz. Razmotrimo za proizvoljnu granu (u, v) grafa G odnos rednog broja u odlaznoj DFS numeraciji čvorova u i v.

- Ako je (u, v) grana drveta ili direktna grana, onda je čvor v potomak čvora u, pa važi $v.Post < u.Post$.
- Ako je (u, v) poprečna grana, onda zbog toga što je čvor v levo od čvora u, ponovo važi $v.Post < u.Post$.
- Ako je (u, v) povratna grana i $v \neq u$, onda je v pravi predak čvora u i $v.Post > u.Post$. Međutim, pošto je specijalno i petlja jedna vrsta povratne grane, za povratnu granu može da važi i $v = u$, pa samim tim i $v.Post = u.Post$. Prema tome, u opštem slučaju za povratnu granu (u, v) važi $u.Post \leq v.Post$.

□

Na osnovu svega prethodnog sledi sledeća lema.

Lema 2.3.5 **[Karakterizacija grana DFS drveta usmerenog grafa]**

Za usmerenu granu $(u, v) \in E$ važi:

- *ako je $u.Post \leq v.Post$, onda je grana (u, v) povratna,*
- *ako je $u.Post > v.Post$ i $u.Pre > v.Pre$, onda je grana (u, v) poprečna,*
- *ako je $u.Post > v.Post$ i $u.Pre < v.Pre$, onda ako je čvor u roditelj čvora v u DFS drvetu grana (u, v) je grana DFS drveta, a inače je (u, v) direktna grana.*

U nastavku je prikazan program kojim se za svaku granu usmerenog grafa određuje njen tip u odnosu na DFS drvo, na osnovu dolazne i odlazne numeracije čvorova.

```
// vrednosti dolazne i odlazne numeracije
vector<int> dolazna;
vector<int> odlazna;
// naredni slobodni redni broj (dolazni i odlazni)
int rbr_dolazna = 0;
int rbr_odlazna = 0;
// roditelj svakog cvora u DFS drvetu
vector<int> roditelj;

void dfs(int cvor) {
  // cvor dobija naredni dolazni redni broj
  dolazna[cvor] = rbr_dolazna;
  rbr_dolazna++;
  // rekurzivno prolazimo kroz sve susede koje do sada nismo obisli
  for (int sused : listaSuseda[cvor]) {
    if (dolazna[sused] == -1) { // !posecen[sused]
      // 'cvor' je roditelj cvora 'sused' u DFS drvetu
      roditelj[sused] = cvor;
      dfs(sused);
    }
  }
  // cvor dobija naredni odlazni redni broj
  odlazna[cvor] = rbr_odlazna;
  rbr_odlazna++;
}
```

```
// klasifikuju se grane DFS drveta dobijenog obilaskom grafa
// krenuvši od datog čvora
void klasifikuj_grane(int cvor) {
  int brojCvorova = listaSuseda.size();
  dolazna.resize(brojCvorova, -1);
  roditelj.resize(brojCvorova, -1);
  odlazna.resize(brojCvorova, -1);

  dfs(cvor);

  cout << "Dolazna i odlazna numeracija cvorova:" << endl;
  for (int i = 0; i < brojCvorova; i++)
    cout << "Cvor " << i << ": " << dolazna[i]
         << "/" << odlazna[i] << endl;

  cout << "Tipovi grana datog usmerenog grafa: " << endl;
  for (int i = 0; i < listaSuseda.size(); i++)
    for (int j = 0; j < listaSuseda[i].size(); j++) {
      int k = listaSuseda[i][j];

      if (i == roditelj[k])
         cout << i << "-" << k << " je grana DFS drveta" << endl;
      else if (odlazna[i] <= odlazna[k])
         cout << i << "-" << k << " je povratna grana" << endl;
      else // odlazna[i] > odlazna[j]
         if (dolazna[i] < dolazna[k])
            cout << i << "-" << k << " je direktna grana" << endl;
         else
            cout << i << "-" << k << " je poprecna grana" << endl;
    }
}
```

2.3.1.3 Provera postojanja ciklusa

Pokazaćemo sada kako se algoritam DFS pretrage može iskoristiti za utvrđivanje da li je dati graf aciklički.

Problem

Za dati usmereni graf $G = (V, E)$ ustanoviti da li sadrži usmereni ciklus.

Lema 2.3.6

Neka je $G = (V, E)$ usmereni graf, i neka je T DFS drvo grafa G. Tada graf G sadrži usmereni ciklus ako i samo ako graf G sadrži povratnu granu u odnosu na DFS drvo T.

Dokaz. Pretpostavimo da graf G sadrži povratnu granu (u, v). Ona zajedno sa granama DFS drveta na putu od v do u čini ciklus, te implikacija u jednom smeru datog tvrđenja direktno važi. Pokažimo da važi i suprotna implikacija, odnosno da ako u grafu postoji ciklus, tada je nužno jedna od njegovih grana povratna. Zaista, pretpostavimo da u grafu G postoji ciklus koji čine grane $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)$, od kojih niti jedna nije povratna u odnosu na DFS drvo T (čvorove u ciklusu smo označili redom brojevima od 1 do k, krenuvši od proizvoljnog čvora tog ciklusa – ove oznake ne moraju odgovarati oznakama čvorova grafa niti DFS numeraciji čvorova). Ako je $k = 1$,

odnosno ciklus je petlja, onda je sama grana (v_1, v_1) povratna. Ako je pak $k > 1$, pretpostavimo da nijedna od grana $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ nije povratna. Prema lemi 2.3.5 važe nejednakosti $v_1.Post > v_2.Post > \cdots > v_k.Post$, iz kojih sledi da je $v_k.Post < v_1.Post$, pa je grana (v_k, v_1) prema lemi 2.3.5 povratna — suprotno pretpostavci. Time je dokazano da u svakom ciklusu postoji povratna grana u odnosu na DFS drvo. □

Na osnovu prethodne leme zaključuje se da se algoritam za proveru da li graf sadrži ciklus može svesti na DFS pretragu grafa, određivanje odlazne DFS numeracije čvorova datog grafa i proveru postojanja povratne grane. Složenost ovog algoritma je $O(|V| + |E|)$.

2.3.2 Obilazak u širinu

Druga važna tehnika obilaska grafa je *obilazak u širinu* ili *pretraga u širinu* (ili BFS, što je skraćenica od breadth–first–search). Ona podrazumeva obilazak grafa na sistematičan način, redom, po udaljenosti čvorova od početnog (gde pod udaljenošću nekog čvora podrazumevamo broj grana na najkraćem putu od početnog do tog čvora).

Problem

Definisati algoritam koji krenuvši od zadatog čvora grafa posećuje (na primer, ispisuje ili na neki drugi način obrađuje) sve čvorove koji su dostupni od tog čvora, pri čemu se čvorovi obrađuju u neopadajućem redosledu najkraćih rastojanja od zadatog početnog čvora.

Obilazak u širinu se vrši na praktično isti način i kod neusmerenih i kod usmerenih grafova (razlike do kojih može doći ćemo posebno opisati).

Ovaj način obilaska je čini pogodnom za traženje najkraćih puteva (puteva sa najmanjih brojem grana) od čvora iz koga pretraga kreće. Razmotrimo graf poznanstava korisnika neke društvene mreže: pretraga u širinu može se iskoristiti za pronalaženje najmanje udaljenosti (u terminima broja osoba) između dva korisnika u mreži. Slično, u algoritmima rutiranja u mrežama računara pretragom u širinu se može odrediti najkraći put između dva čvora u mreži.

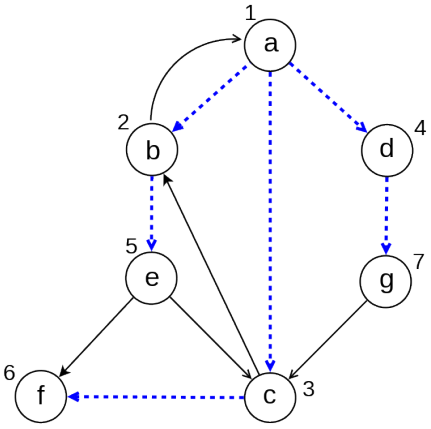
Pretraga grafa u širinu može biti korisna i u drugim kontekstima. Veb, kao složena mreža veb stranica međusobno povezanih vezama (linkovima), se može prirodno razmatrati kao jedan veliki i složen graf čiji čvorovi odgovaraju pojedinačnim veb stranicama, dok grana od jednog čvora do drugog čvora postoji ako i samo ako na prvoj veb stranici postoji link ka drugoj veb stranici. Interesantan problem jeste indeksiranje veb stranica, koje koriste pretraživači za veb kako bi ažurirali svoj sadržaj ili indekse sadržaja drugih veb stranica. Pošto je u ovom problemu cilj da se krećemo sistematično počev od neke veb stranice, to možemo postići pretragom grafa u širinu, koja u prvom koraku obilazi sve stranice ka kojima sa polazne stranice postoji veza, u narednom koraku se posećuju njihove veze, itd.

Kao i u slučaju obilasku u dubinu, obilazak u širinu implicitno određuje drvo koje se u ovom slučaju naziva *drvo pretrage u širinu* (BFS drvo). Pretpostavimo da je graf zadat listama povezanosti. Ako BFS obilazak pokrenemo iz čvora v, onda čvor v postaje koren BFS drveta. Nakon čvora v posećuju se svi susedi čvora v redosledom određenim redosledom u listi povezanosti grafa i oni postaju deca čvora v u BFS drvetu (čvorovi nivoa 1). Zatim se dolazi do svih njihovih neposećenih suseda, odnosno do "unuka" polaznog čvora (čvorovi nivoa 2), i tako dalje. Prilikom obilaska čvorovi se mogu numerisati *BFS brojevima*, slično kao pri obilasku u dubinu. Preciznije, čvor w ima BFS broj k ako je on k-ti po redu čvor označen u toku obilaska algoritmom BFS. BFS drvo grafa formira se uključivanjem samo grana ka novooznačenim čvorovima: lako se pokazuje da je u slučaju polaznog neusmerenog grafa dobijeni podgraf povezan i da nema ciklus, jer od svih grana koje vode nekom čvoru uključujemo tačno jednu. Ako graf nije povezan, obilazak se može pokrenuti zasebno u svakoj komponenti povezanosti (na taj način se dobija BFS šuma).

Zapaža se da izlazna obrada kod obilaska u širinu, za razliku od obilaska u dubinu, nema smisla; obilazak nema povratak "naviše", već se, polazeći od korena, kreće samo naniže.

Primer 2.3.5

Za graf prikazan na slici 2.21 prikazan je postupak obilaska u širinu pokrenut iz čvora a: nakon čvora a posećuju se njegovi susedi – čvorovi b, c i d i oni postaju čvorovi nivoa 1. Nakon njih se obilaze neposećeni susedi čvora b – to je jedino čvor e, pa čvor f kao jedini neposećeni sused čvora c i konačno čvor g kao neposećeni sused čvora d: čvorovi e, f i g postaju čvorovi nivoa 2 u BFS drvetu. Daljom analizom možemo utvditi da nijedan od čvorova e, f i g nema neposećenih suseda, te se obilazak u širinu završava. Grane grafa koje pripadaju BFS drvetu istaknute su isprekidanom linijom, a uz svaki čvor prikazana je njegov redni broj u BFS numeraciji.



Slika 2.21: BFS drvo i BFS numeracija usmerenog grafa.

Kako realizovati pretragu u širinu? Čvorove obilazimo u željenom redosledu korišćenjem pogodne strukture podataka: sve neposećene susede tekućeg čvora smeštamo u datu kolekciju i potrebno ih je obraditi u redosledu dodavanja u tu kolekciju. U ove svrhe možemo iskoristiti red kao strukturu podataka. Čvorovi se obeležavaju čim se postave u red (tj. čim im se zakaže budući obilazak). Ovim se obezbeđuje da se ni jedan čvor neće dva puta staviti u red tj. da će red uvek sadržati različite čvorove. Zato je maksimalni broj elemenata koji mogu biti u redu jednak $|V|$, što omogućava da se red implementira i uz pomoć običnog niza dužine $|V|$.

Prikažimo na koji način se može realizovati pretraga u širinu ako je graf povezan i ako je zadat listama povezanosti (ako graf nije povezan svaka njegova komponenta se obilazi zasebno). Prilikom pretrage štampamo čvorove grafa u redosledu određenom BFS numeracijom čvorova i konstruišemo BFS drvo. BFS drvo ćemo predstaviti listom grana. Pretpostavićemo da je graf usmeren, te da pri dodavanju grane (u, v) u BFS drvo ne treba dodavati i njoj simetričnu granu (v, u).

```
// pretraga grafa u sirinu pokrenuta iz cvora cvor
// pretpostavljamo da je graf usmeren
void bfs(int cvor) {
  int brojCvorova = listaSuseda.size();
  vector<bool> oznacen(brojCvorova, false);
  vector<vector<int>> bfs_drvo(brojCvorova);

  // red u kom cuvamo cvorove u redosledu kojim ih treba posetiti
  queue<int> red;
  red.push(cvor);
  oznacen[cvor] = true;
  // stampamo prvi cvor u redosledu BFS numeracije
  cout << cvor << endl;
  while (!red.empty()) {
```

```
    // dohvatamo element sa pocetka reda i uklanjamo ga iz kolekcije
    int cvor = red.front();
    red.pop();
    for (int sused : listaSuseda[cvor]) {
      // neoznacene susede tekuceg cvora dodajemo u red
      if (!oznacen[sused]) {
        oznacen[sused] = true;
        // stampamo naredni cvor u skladu sa BFS numeracijom
        cout << sused << endl;
        // dodajemo odgovarajucu granu u bfs drvo
        bfs_drvo[cvor].push_back(sused);
        red.push(sused);
      }
    }
  }
  cout << "Grane BFS drveta su: ";
  for (int i = 0; i < bfs_drvo.size(); i++)
    for (int j = 0; j < bfs_drvo[i].size(); j++)
      cout << "(" << i << "," << bfs_drvo[i][j] << ")" << endl;
}
```

Lako je uveriti se da se prilikom BFS obilaska datog povezanog grafa svaki čvor obrađuje po jednom i da se svaka grana pregleda po jednom u slučaju usmerenog grafa, odnosno dva puta ako je graf neusmeren (grana se smatra pregledanom kada se u toku pretrage iz njenog početka naiđe na njen kraj). Stoga je vremenska složenost algoritma pretrage u širinu $O(|V| + |E|)$. Slično kao i kod pretrage u dubinu, ako je graf predstavljen matricom povezanosti, složenost pretrage u širinu jednaka je $\Theta(|V|^2)$, jer je prolaz kroz sve susede nekog čvora složenosti $\Theta(|V|)$. Dakle, obično je pretraga u širinu efikasnija kada je graf predstavljen listama povezanosti, posebno ako je graf redak.

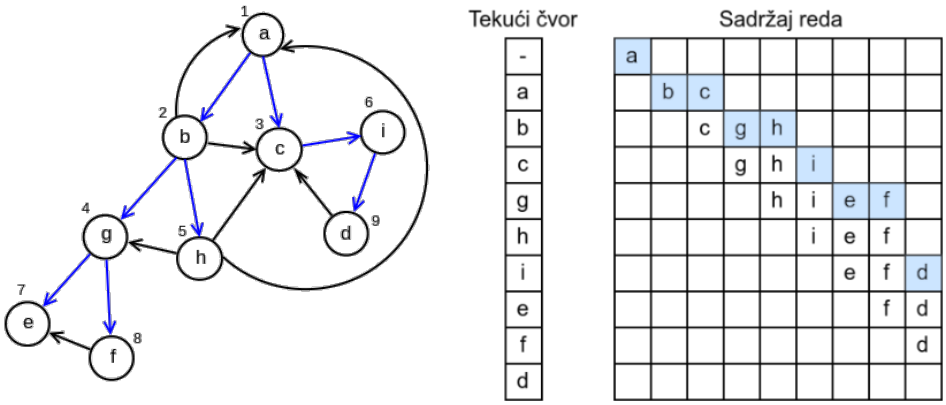
Prostorna složenost algoritma BFS zavisi od maksimalnog broja čvorova koji će red sadržati tokom izvršavanja algoritma. U situaciji kada je polazni čvor povezan sa svim ostalim čvorovima, nakon polaznog čvora će u red biti dodati svi preostali čvorovi. Dakle, prostorna složenost algoritma BFS je u najgorem slučaju $O(|V|)$.

Primer 2.3.6

Razmotrimo primer izvršavanja algoritma obilaska u širinu pokrenutog iz čvora a na primeru grafa sa slike 2.22. Neka redosled suseda čvorova u listi povezanosti kojom je predstavljen graf odgovara leksikografski rastućem poretku oznaka čvorova.

- *Najpre iz reda uzimamo i posećujemo čvor a, a u red dodajemo njegove neposećene susede: čvor b i čvor c.*
- *Iz reda uzimamo i posećujemo čvor b, a dodajemo njegove neposećene susede: čvorove g i h redom. Zatim iz reda uzimamo i posećujemo čvor c, a u red dodajemo njegove neposećene susede: čvor i.*
- *Iz reda uzimamo i posećujemo čvor g, a u red dodajemo njegove neposećene susede: čvorove e i f. Zatim iz reda uzimamo i posećujemo čvor h koji nema neposećenih suseda. Zatim iz reda uzimamo i posećujemo čvor i, a u red dodajemo njegove neposećene susede: čvor d.*
- *Nakon ovoga redom iz reda uzimamo i posećujemo čvorove e, f i d i zaključujemo da nijedan od njih nema neposećenih suseda i pretraga se završava.*

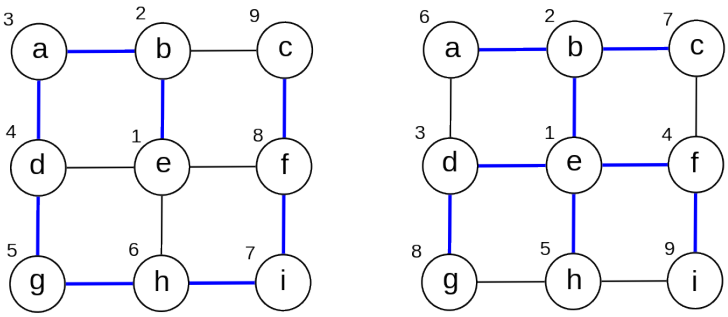
Stanje reda tokom obilaska je prikazano na slici 2.22, desno. Grane BFS drveta su na slici 2.22 levo prikazane plavom bojom, a uz svaki čvor prikazan je njegov broj u okviru BFS numeracije.



Slika 2.22: Primer obilaska u širinu usmerenog grafa.

Primer 2.3.7

Na slici 2.23 prikazani su obilazak u dubinu i obilazak u širinu jednog neusmerenog grafa. Pretpostavljamo da je graf predstavljen listama povezanosti i da su čvorovi u listama povezanosti navedeni u leksikografski rastućem poretku. U levom delu slike prikazan je postupak pretrage u dubinu pokrenut iz čvora a: uz svaki čvor prikazan je njegov redni broj u dolaznoj DFS numeraciji, a plavom bojom su istaknute grane koje pripadaju DFS drvetu grafa. U desnom delu slike ilustrovan je postupak pretrage u širinu pokrenut iz čvora a: uz svaki čvor prikazana je njegov redni broj u BFS numeraciji, a plavom bojom su istaknute grane koje pripadaju BFS drvetu grafa.



Slika 2.23: Razlika između DFS i BFS drveta za obilazak grafa pokrenut iz središnjeg čvora. Plavom bojom označene su grane koje pripadaju DFS i BFS drvetu, redom. Čvorovi su numerisani redosledom kojim se obilaze.

Primetimo da kod neusmerenih grafova u odnosu na BFS drvo mogu postojati jedino grane BFS drveta i poprečne grane (setimo se da u neusmerenom grafu u odnosu na DFS drvo nisu mogle da postoje poprečne grane).

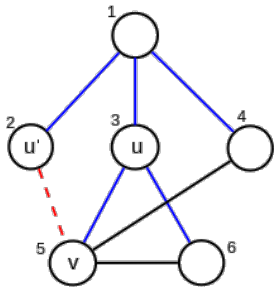
U nastavku ćemo formulisati nekoliko tvrđenja koja važe za obilazak u širinu, odnosno za BFS drvo grafa.

Lema 2.3.7

Ako grana (u, v) pripada BFS drvetu grafa G i čvor u je roditelj čvora v, onda čvor u ima najmanji BFS broj među čvorovima iz kojih postoji grana ka v.

Dokaz. Pretpostavimo suprotno, da u grafu G postoji grana (u', v), takva da čvor u' ima manji BFS broj od čvora u, pri čemu i u i u' oba imaju manje BFS brojeve od čvora v (slika 2.24), kao i da je u' čvor sa najmanjim BFS brojem koji zadovoljava prethodni uslov. U trenutku obrade čvora u' je onda čvor v morao

biti upisan u red, pa je grana (u', v) morala biti uključena u BFS drvo. Međutim, prema pretpostavci, grana (u, v) je uključena u BFS drvo, pa ne može istovremeno biti uključena i grana (u', v), te dobijamo kontradikciju. □



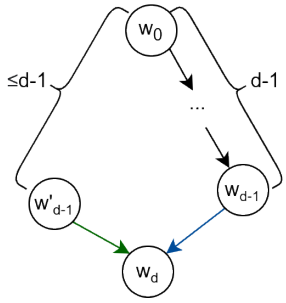
Slika 2.24: Ilustracija uz dokaz leme 2.3.7. Plavom bojom istaknute su grane BFS drveta, a uz svaki čvor prikazan je redni broj u BFS numeraciji čvora.

Definišimo *rastojanje* $\mathrm{d}(u, v)$ između čvorova u i v kao dužinu najkraćeg puta od čvora u do čvora v, s tim da se pod dužinom puta podrazumeva broj grana koje čine taj put. Nivo čvora v u odnosu na BFS drvo sa korenom u jednak je rastojanju $\mathrm{d}(u, v)$.

Lema 2.3.8

Put od korena r BFS drveta do proizvoljnog čvora w kroz BFS drvo najkraći je put od čvora r do čvora w u grafu G.

Dokaz. Indukcijom po d dokazaćemo da do svakog čvora w na rastojanju d od korena r (jedinstveni) put kroz BFS drvo od r do w ima dužinu tačno d. Za $d = 0$ tvrđenje je trivijalno tačno: jedini čvor na rastojanju 0 od korena r je on sâm, a put kroz drvo od r do r takođe ima dužinu 0. Pretpostavimo da je tvrđenje tačno za sve čvorove koji su na rastojanju manjem od d od korena r, i neka je w neki čvor na rastojanju d od korena; drugim rečima, postoji niz čvorova $(w_0 = r, w_1, w_2, \ldots, w_{d-1}, w_d = w)$ koji čine put dužine d od r do w, i ne postoji kraći put od r do w. Pošto je dužina najkraćeg puta od r do w_{d-1} jednaka $d - 1$, prema induktivnoj hipotezi put od r do w_{d-1} kroz BFS drvo ima dužinu $d - 1$. U trenutku obrade čvora w_{d-1}, ako čvor w_d nije označen, pošto u grafu G postoji grana (w_{d-1}, w_d), ta grana se uključuje u BFS drvo, pa do čvora w_d postoji put dužine d kroz BFS drvo (slika 2.25). U protivnom, ako je u tom trenutku čvor w_d već označen, onda do čvora w_d kroz BFS drvo vodi grana iz nekog čvora w'_{d-1}, označenog pre w_{d-1}, iz čega sledi da je nivo čvora w'_{d-1} najviše $d - 1$, što znači da kroz drvo postoji put do čvora w'_{d-1} čija je dužina najviše $d - 1$. Ona ne može biti manja od $d - 1$, jer bi tada u grafu postojao put od r do w_d kraći od d, što je suprotno pretpostavci da je w_d na rastojanju d od korena. Dakle, dužina puta od r do w'_{d-1} u drvetu mora biti tačno $d - 1$, pa pošto grana (w'_{d-1}, w_d) pripada drvetu, rastojanje od r do w_d u drvetu je tačno d. □



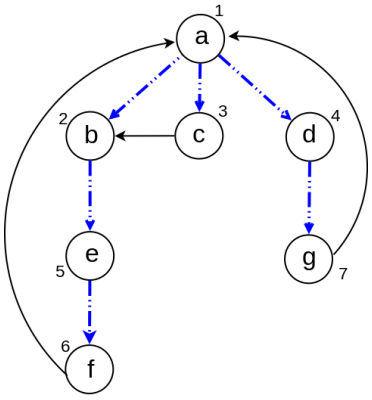
Slika 2.25: Ilustracija uz dokaz leme 2.3.8.

Lema 2.3.9

Ako je graf $G = (V, E)$ neusmeren i $(v, w) \in E$ neka njegova proizvoljna grana, onda se nivoi čvorova v i w razlikuju najviše za jedan.

Dokaz. Pretpostavimo da je od čvorova v i w, čvor v prvi dostignut BFS pretragom i neka je njegov nivo d. Tada je nivo čvora w veći ili jednak od d. S druge strane, nivo čvora w nije veći od $d + 1$, jer do njega vodi grana BFS drveta ili iz čvora v, ili iz nekog čvora koji je označen pre v. Dakle, nivo čvora w je ili d ili $d + 1$, te tvrđenje leme važi. □

Primetimo da analogno tvrđenje ne važi nužno u slučaju da je graf usmeren. Na slici 2.26 je prikazan usmeren graf koji sadrži grane koje povezuju dva čvora čiji se nivoi razlikuju za više od 1: na primer grane (f, a) i (g, a).



Slika 2.26: Ilustracija da lema 2.3.9 ne važi za usmerene grafove (grane (f, a) i (g, a) spajaju čvorove koji nisu na susednim nivoima).

2.3.2.1 Pokretanje obilaska iz više čvorova istovremeno

Uobičajeno je da se obilazak grafa pokreće iz jednog polaznog čvora, međutim, nekada je obilazak u širinu moguće pokrenuti i iz više čvorova istovremeno. Pretpostavimo, recimo, da su u datom grafu neki čvorovi crveni, a neki plavi i da nas zanima najkraće rastojanje (mereno brojem grana jedinične dužine) između nekog crvenog i plavog čvora. Direktan pristup bi podrazumevao da se pokrene obilazak u širinu iz svakog crvenog čvora, da se izmeri rastojanje od njega do njemu najbližeg plavog čvora i da se odredi minimum tako dobijenih vrednosti. To bi zahtevalo da se obilazak u dubinu pokreće iznova za svaki crveni čvor, što može biti neefikasno (složenost može biti čak $O(|V|^3)$). Efikasniji pristup je da se obilazak pokrene istovremeno iz svih crvenih čvorova, što znači da se u početnom koraku svi istovremeno dodaju u red i da se svakom od njih pridruži rastojanje 0. Nakon toga, obilazak u širinu bi se nastavljao na uobičajeni način (uzimanjem elementa sa početka reda i dodavanjem u red njegovih neoznačenih suseda uz rastojanje uvećano za jedan u odnosu na rastojanje elementa sa početka reda), sve dok se ne naiđe na prvi plavi čvor, čime bi se dobilo traženo najkraće rastojanje između crvenih i plavih čvorova. Pošto se vrši samo jedan obilazak, složenost ovog algoritma je $O(|V| + |E|)$, što je u najgorem slučaju $O(|V|^2)$.

Zadatak: Postavljanje računarske mreže

U jednom računarskom kabinetu potrebno je uspostaviti neobičnu računarsku mrežu. Potrebno je postaviti računare na n stolova, pri čemu na nekim stolovima mogu da stoje obični računari (njih imamo na raspolaganju u neograničenom broju), a na nekim posebni serveri (njih posebno moramo kupiti po ceni od c_s dinara). Neke parove stolova tj. računara na njima je moguće povezati direktno mrežnim kablovima, a neke nije. Cena uspostavljanja mrežnog kabla između bilo koja dva stola je c_k dinara. Napisati program koji određuje najmanju cenu koju je potrebno platiti tako da je svaki računar ili server ili je mrežnim kablom povezan sa bar jednim serverom (ne obavezno direktno).

Opis ulaza

Sa standardnog ulaza se u prvom redu unose brojevi c_s ($0 \leq c_s \leq 1000$) i c_k ($0 \leq c_k \leq 1000$), razdvojeni razmakom. U narednom redu se unosi broj stolova n ($2 \leq n \leq 50000$) i m broj parova stolova između kojih je moguće postaviti mrežni kabl $2 \leq m \leq \frac{n(n-1)}{2}$, razdvojeni razmakom. U narednih m redova unose se parovi brojeva između 0 i $n-1$, razdvojenih razmakom koji određuju stolove između kojih je moguće postaviti kabl.

Opis izlaza

Na standardni izlaz ispisati traženu najmanju cenu.

Primer

Ulaz

```
850 350
7 6
0 1
0 4
4 2
1 4
3 5
6 5
```

Izlaz

```
3450
```

Rešenje

Problem modelujemo neusmerenim grafom kom su čvorovi stolovi, a grane postoje između svaka dva čvora (tj. stola) između kojih je moguće uspostaviti kabl.

Ako je cena servera manja ili jednaka ceni jednog kabla (iako ta pretpostavka nije realna), tada je najjeftinije na svaki sto postaviti po jedan server. Naime zamena svakog servera običnim računarom podrazumeva povezivanje tog računara sa nekim serverom pomoću jednog kabla, što povećava ukupnu cenu. Optimalna cena u tom slučaju je jednaka proizvodu broja računara n i cene servera c_s.

U suprotnom je optimalno u svakoj komponenti povezanosti grafa. postaviti po jedan server i sve ostale stolove u toj komponenti popuniti običnim računarima povezanim sa tim serverom. Zaista, ako u nekoj komponenti ne bi postojao bar jedan server, onda računari u toj komponenti ne bi mogli biti povezani ni sa jednim serverom, što je suprotno uslovima zadatka. Ukoliko bi postojala bar dva servera u nekoj komponenti, cena ne bi bila optimalna. Naime jedan od ta dva servera bi se mogao zameniti običnim računarom koji bi se kablom povezao sa drugim serverom u toj komponenti, čime bi se umesto cene servera platila cena kabla koja je manja. Neka je k broj komponenata povezanosti. Potrebno je uspostaviti k servera (po jedan u svakoj komponenti), a ostale računare, njih $n-k$ povezati kablom sa po jednim računarom. Zato je ukupna cena $k \cdot c_s + (n-k) \cdot c_k$.

Broj komponenata povezanosti možemo jednostavno odrediti obilaskom grafa (na primer, rekurzivno implementiranim obilaskom u dubinu).

```
void dfs(const vector<vector<int>>& susedi,
         vector<int>& komponente,
         int cvor, int komponenta) {
  komponente[cvor] = komponenta;
  for (int sused : susedi[cvor])
    if (komponente[sused] == 0)
```

```
      dfs(susedi, komponente, sused, komponenta);
}

int broj_komponenata_povezanosti(const vector<vector<int>>& susedi,
                                 int broj_cvorova) {
  vector<int> komponente(broj_cvorova, 0);
  int komponenta = 0;
  for (int cvor = 0; cvor < broj_cvorova; cvor++)
    if (komponente[cvor] == 0)
      dfs(susedi, komponente, cvor, ++komponenta);
  return komponenta;
}

int main() {
  int broj_racunara, broj_kablova;
  long long cena_servera, cena_kabla;
  cin >> cena_servera >> cena_kabla;
  cin >> broj_racunara >> broj_kablova;
  vector<vector<int>> susedi(broj_racunara);
  for (int i = 0; i < broj_kablova; i++) {
    int racunar1, racunar2;
    cin >> racunar1 >> racunar2;
    susedi[racunar1].push_back(racunar2);
    susedi[racunar2].push_back(racunar1);
  }

  if (cena_servera <= cena_kabla)
    cout << broj_racunara * cena_servera << endl;
  else {
    int broj_komponenata = broj_komponenata_povezanosti(susedi, broj_racunara);
    int broj_servera = broj_komponenata;
    int broj_obicnih = broj_racunara - broj_servera;
    cout << broj_servera * cena_servera + broj_obicnih * cena_kabla << endl;
  }

  return 0;
}
```

Zadatak: Provera da li je graf bipartitan

Grupa studenata se okupila na letnjem kampu. Svako je znao nekoliko drugih studenata. Ispostavilo se da se svaki par studenata poznaje posredno (preko niza zajedničkih poznanika).

Potrebno je da se studenti podele u dve grupe, ali pošto svako želi da upozna što više novih kolega, potrebno je da svaku grupu čine međusobno nepoznate osobe (dve osobe koje se već poznaju ne mogu biti u istoj grupi). Napisati program koji određuje da li je to moguće i ako jeste, koji će sve studenti biti u grupi sa studentom čiji je redni broj 0.

Opis ulaza

Sa standardnog ulaza se učitava broj studenata n ($1 \leq n \leq 10^5$), broj parova m studenata koji se od ranije poznaju ($0 \leq m \leq \frac{n(n-1)}{2}$), a zatim i niz parova brojeva od 0 do $n-1$ koji predstavljaju njihova poznanstva.

Opis izlaza

Na standardni izlaz ispisati redne brojeve studenata koji su u grupi sa studentom koji nosi redni broj 0, u rastućem poretku, ili simbol - ako tražene dve grupe nije moguće formirati.

Primer 1

Ulaz

```
6
6
0 1
1 2
2 3
3 4
4 5
5 0
```

Izlaz

```
0 2 4
```

Objašnjenje

Ako su u jednoj grupi studenti sa brojevima 0, 2 i 4, u drugoj su studenti sa brojevima 1, 3 i 5 i tada se ni u jednoj grupi ne nalaze studenti koji se međusobno poznaju.

Primer 2

Ulaz

```
5
5
0 1
1 2
2 3
3 4
4 0
```

Izlaz

```
-
```

Objašnjenje

Student 0 ne sme da bude u grupi sa studentom 1, koji ne sme da bude u grupi sa studentom 2, što znači da 0 i 2 moraju da budu u istoj grupi. Studenti 2 i 3 ne mogu da budu u istoj grupi, pa su 1 i 3 u istoj grupi. Student 4 ne sme da bude u grupi sa studentom 3, pa on mora biti u grupi sa studentima 0 i 2, međutim, to nije dopušteno, jer se studenti 4 i 0 poznaju.

Rešenje

Studente i njihova poznanstva možemo predstaviti neusmerenim grafom. Postavlja se pitanje da li je taj graf *bipartitan* tj. da li mu se čvorovi mogu podeliti u dve grupe tako da sve grane spajaju čvorove iz dve različite grupe.

Ako neki čvor pripada levoj polovini, tada svi njegovi susedi pripadaju desnoj polovini, njihovi susedi levoj polovini, njihovi susedi desnoj i tako dalje. Zato se zadatak može rešiti obilaskom grafa (na primer u dubinu) obeležavajući čvorove naizmenično (za svaki neposećeni čvor se obeležava da li pripada levoj ili

desnoj polovini). Ako se prilikom obilaska naiđe na čvor čiji je sused već obeležen tako da pripada istoj polovini kao tekući, tada graf nije bipartitan. Ako se na takav čvor ne naiđe, tada graf jeste bipartitan.

Po uslovima zadatka svi studenti se poznaju posredno, što znači da je graf povezan. Ako graf ne bi bio povezan, postupak pretrage u dubinu i označavanja čvorova bi trebalo ponoviti za svaku komponentu povezanosti zasebno.

```
bool moze = true;
// svakom cvoru dodeljujemo jednu od dve boje (tj. oznake grupa)
vector<int> boje(n, -1);
int boja = 0;
stack<int> stek;
stek.push(0);
// prvi cvor bojimo prvom bojom
boje[0] = 0;
while (!stek.empty() && moze) {
  int x = stek.top(); stek.pop();
  // susedi trenutnog cvora x treba da budu obojeni suprotnom bojom od x
  boja = 1 - boje[x];
  for (int sused : susedi[x]) {
    // sused je vec obojen pogresnom bojom, pa graf nije bipartitan
    if (boje[sused] != -1 && boje[sused] != boja) {
      moze = false;
      break;
    }
    // sused nije obojen, pa ga bojimo suprotnom bojom od tekuceg cvora x
    if (boje[sused] == -1) {
      boje[sused] = boja;
      stek.push(sused);
    }
  }
}
```

Zadatak: Avionska presedanja

Jedna avio-kompanija zajedno sa svojim partnerima izvodi letove između poznatih svetskih aerodroma. Napiši program koji određuje da li je moguće da se korišćenjem tih letova stigne sa jednog na drugi dati aerodrom i ako jeste, koliko je najmanje letova potrebno.

Opis ulaza

Sa standardnog ulaza se zadaje broj m ($1 \leq m \leq 100$) letova koje kompanija izvodi, a zatim u narednih m redova opis tih letova (šifra polaznog i šifra dolaznog aerodroma, razdvojeni razmakom). Nakon toga se unosi broj k ($1 \leq k \leq 100$) putnika koji su zainteresovani za letove koje pruža ta kompanija, i u narednih k linija opisi relacija na kojima oni putuju (šifra polaznog i šifra dolaznog aerodroma, razdvojeni razmakom).

Opis izlaza

Za svakog od k putnika na standardni izlaz ispisati najmanji broj letova pomoću kojih mogu da ostvare željeno putovanje ili reč `ne` ako takvo putovanje nije moguće ostvariti pomoću letova koje kompanija izvodi.

Primer

Ulaz

```
BEG FRA
FRA MUC
FRA JFK
BEG MUC
MUC LAX
LAX JFK
LAX ORD
3
BEG JFK
MUC BEG
BEG ORD
```

Izlaz

```
2
ne
3
```

Objašnjenje

Od Beograda (BEG) do Njujorka (JFK) može se stići preko Frankfurta (FRA). Od Minhena (MUC) do Beograda (BEG) nije moguće organizovati putovanje. Od Beograda (BEG) do Čikaga (ORD) moguće je putovati preko Minhena (MUC) i Los Anđelesa (LAX).

Rešenje

Aerodromi i letovi između njih se mogu predstaviti usmerenim grafom. Najkraće puteve u tom grafu možemo najjednostavnije pronaći pretragom u širinu. Nju implementiramo tako što u red stavljamo čvorove u redosledu njihovog rastojanja od polaznog čvora. Na početku stavljamo polazni čvor, a zatim u svakom koraku skidamo čvor sa početka reda i u red dodajemo njegove susede koji ranije nisu posećeni. Uz svaki čvor u red postavljamo i njegovo rastojanje od polaznog čvora. U trenutku kada u red treba staviti dolazni čvor, znamo njegovo najkraće rastojanje. Ako se red isprazni pre nego što se dolazni čvor postavi u njega, onda polazni i dolazni čvor nisu povezani.

```cpp
// pretragom u sirinu odredjujemo najkrace rastojanje izmedju dva aerodro
// graf je predstavljen listama povezanosti, pri čemu su oznake
// cvorova niske, a ne redni brojevi
int brojPresedanjaBFS(const string& aerodromOd, const string& aerodromDo,
                      const map<string, vector<string>>& letovi) {
  // skup posecenih aerodroma
  set<string> posecen;
  // red potreban za implementaciju obilaska u sirinu
  queue<pair<string, int>> red;
  // krecemo od polaznog aerodroma
  red.emplace(aerodromOd, 0);
  while (!red.empty()) {
    // uzimamo tekuci aerodrom iz reda i njegovo rastojanje od polaznog
    string aerodrom;
    int rastojanje;
    tie(aerodrom, rastojanje) = red.front();
    red.pop();
    posecen.insert(aerodrom);

    // lista suseda tekuceg aerdroma
    auto it = letovi.find(aerodrom);
```

```
    if (it != letovi.end()) {
      // prolazimo kroz sve letove sa tekuceg aerodroma
      for (const string& sused : it->second) {
        // ako su poseceni, preskacemo ih
        if (posecen.find(sused) != posecen.end())
          continue;
        // ako smo dostigli dolanzi aerodrom, znamo najkrace
        // rastojanje do njega
        if (sused == aerodromDo)
          return rastojanje+1;
        // dodajemo susedni aerodrom u red
        red.emplace(susedni, rastojanje+1);
      }
    }
  }
  // dolazni aerodrom nije dostizan
  return -1;
}

int main() {
  // graf je predstavljen listama povezanosti, pri čemu su oznake
  // cvorova niske, a ne redni brojevi
  int m;
  cin >> m;
  map<string, vector<string>> letovi;
  for (int i = 0; i < m; i++) {
    string aerodromOd, aerodromDo;
    cin >> aerodromOd >> aerodromDo;
    letovi[aerodromOd].push_back(aerodromDo);
  }

  // odgovaramo na k upita
  int k;
  cin >> k;
  for (int i = 0; i < k; i++) {
    string aerodromOd, aerodromDo;
    cin >> aerodromOd >> aerodromDo;
    int broj = brojPresedanjaBFS(aerodromOd, aerodromDo, letovi);
    if (broj == -1)
      cout << "ne" << endl;
    else
      cout << broj << endl;
  }
  return 0;
}
```

Zadatak: Kose crte

Ispred dvorca, kralj ima vrt pravougaonog oblika koji je podeljen u mrežu $m \times n$ kvadrata. Po obimu pravougaonika, kao i duž dijagonala nekih od tih kvadrata zasadio je živu ogradu i tako je napravio jedan neobičan lavirint. Napisati program koji određuje na koliko oblasti je podeljen taj lavirint (iz jedne oblasti

se ne može doći u drugu ako se ne preskoči živa ograda).

Opis ulaza

Sa standardnog ulaza se učitavaju dimenzije pravougaonika m i n ($1 \leq m, n \leq 50$), a zatim matrica karaktera dimenzije $m \times n$ koja opisuje pojedinačne kvadrate. Karakter \ označava da je ograda postavljena duž glavne, karakter / da je ograda postavljena duž sporedne dijagonale, a razmak da u tom kvadratu nema žive ograde.

Opis izlaza

Na standardni izlaz ispisati traženi broj oblasti.

Primer 1

Ulaz

```
2 2
\/
/\
```

Izlaz

```
4
```

Objašnjenje

Lavirint i njegove četiri oblasti su prikazani na slici.

Primer 2

Ulaz

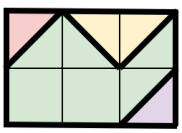
```
2 3
/\/
  /
```

Izlaz

```
4
```

Objašnjenje

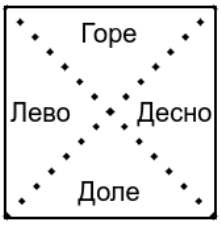
Lavirint i njegove četiri oblasti su prikazani na slici.



Rešenje

Prilično je jasno da se zadatak može rešiti tako što se prebroje komponente povezanosti u grafu koji gradimo na osnovu podataka o lavirintu, međutim, prvo treba na pravi način definisati taj graf. Jedan način je da

svaki kvadrat u lavirintu podelimo na četiri oblasti i da svaka oblast predstavlja jedan čvor grafa (taj graf ima $4mn$ čvorova).



Četiri oblasti u jednom kvadratu

Oblasti u jednom kvadratu su povezane, osim ako je postavljena neka živa ograda.

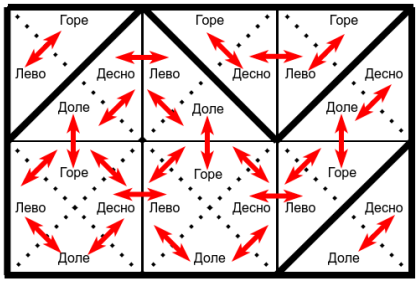
- Iz gornje oblasti možemo doći u levu, osim ako je živa ograda na glavnoj dijagonali i u desnu, osim ako je živa ograda na sporednoj dijagonali.
- Iz donje oblasti možemo doći u levu, osim ako je živa ograda na sporednoj dijagonali i u desnu, osim ako je živa ograda na glavnoj dijagonali.
- Iz leve oblasti možemo doći u gornju, osim ako je živa ograda na glavnoj dijagonali i u donju, osim ako je živa ograda na sporednoj dijagonali.
- Iz desne oblasti možemo doći u gornju, osim ako je živa ograda na sporednoj dijagonali i u donju, osim ako je živa ograda na glavnoj dijagonali.

Mogući su prelasci i iz jednog u drugi kvadrat.

- Iz gornje oblasti bilo kog kvadrata možemo doći u donju oblast kvadrata iznad tog kvadrata (ako takav kvadrat postoji).
- Iz donje oblasti bilo kog kvadrata možemo doći u gornju oblast kvadrata ispod tog kvadrata (ako takav kvadrat postoji).
- Iz leve oblasti bilo kog kvadrata možemo doći u desnu oblast kvadrata levo tog kvadrata (ako takav kvadrat postoji).
- Iz desne oblasti bilo kog kvadrata možemo doći u levu oblast kvadrata desno od tog kvadrata (ako takav kvadrat postoji).

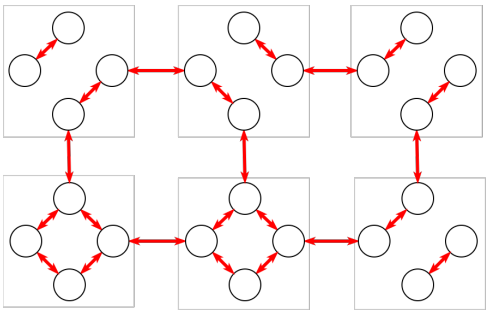
Na narednoj slici prikazana je povezanost oblasti u lavirintu koji se opisuje karakterima:

```
/\/
  /
```



Odgovarajući graf je prikazan na narednoj slici.

Pošto kvadrata ima $m \times n$, čvorova grafa ima $4 \times m \times n$. Svaki čvor je povezan sa najviše 3 druga čvora, pa je ukupna složenost prebrojavanja komponenti $O(mn)$.



```
// svakom polju odgovaraju ima 4 čvora grafa
enum deo {GORE=0, DOLE, LEVO, DESNO};

// obilazak grafa zadatog nizom stringova od cvora sa koordinatama (v0, k0, d0)
// to je cvor u vrsti v0, koloni k0 i delu d0
// poseceni cvorovi su određeni visedimenzionim nizom posecen
void dfs(const vector<string>& linije, int m, int n,
         vector<vector<vector<bool>>>& posecen,
         int v0, int k0, int d0) {
  // na steku cuvamo koordinate cvorova grafa
  stack<tuple<int, int, int>> s;
  s.emplace(v0, k0, d0);
  posecen[v0][k0][d0] = true;

  while (!s.empty()) {
    // skidamo koordinate tekuceg cvora sa steka
    int v, k, d;
    tie(v, k, d) = s.top(); s.pop();

    // odredjujemo susede tekuceg cvora (ima ih najvise 3)
    vector<tuple<int, int, int>> susedi;
    // analiziramo polozaj tekuceg cvora u njegovom polju
    switch(d) {
    case GORE:
      // levi i desni cvor tekuceg polja (ako nisu zaklonjeni preprekama)
      if (linije[v][k] != '\\')
        susedi.emplace_back(v, k, LEVO);
      if (linije[v][k] != '/')
        susedi.emplace_back(v, k, DESNO);
      // donji cvor polja iznad (ako to polje postoji)
      if (v > 0)
        susedi.emplace_back(v-1, k, DOLE);
      break;
    case DOLE:
      // levi i desni cvor tekuceg polja (ako nisu zaklonjeni preprekama)
      if (linije[v][k] != '/')
        susedi.emplace_back(v, k, LEVO);
      if (linije[v][k] != '\\')
        susedi.emplace_back(v, k, DESNO);
```

```
      // gornji cvor polja ispod (ako to polje postoji)
      if (v < m-1)
        susedi.emplace_back(v+1, k, GORE);
      break;
      // ...
    }

    // prolazimo kroz niz suseda tekuceg cvora
    for (const auto& t : susedi) {
      int vv, kk, dd;
      tie(vv, kk, dd) = t;
      // neposecene susede stavljamo na stek
      if (!posecen[vv][kk][dd]) {
        posecen[vv][kk][dd] = true;
        s.push(t);
      }
    }
  }
}

int main() {
  // ucitavamo crtez lavirinta
  int m, n;
  cin >> m >> n;
  string s;
  getline(cin, s);
  vector<string> linije(m);
  string linija;
  for (int i = 0; i < m; i++)
    getline(cin, linije[i]);

  // alociramo vektor posecenosti cvorova
  // na pocetku nije posecen ni jedan cvor
  vector<vector<vector<bool>>> posecen;
  posecen.resize(m);
  for (int i = 0; i < m; i++) {
    posecen[i].resize(n);
    for (int j = 0; j < n; j++)
      posecen[i][j].resize(4, false);
  }

  // odredjujemo komponente povezanosti grafa i ispisujemo njihov broj
  int brojOblasti = 0;
  for (int v = 0; v < m; v++)
    for (int k = 0; k < n; k++)
      for (int d = GORE; d <= DESNO; d++)
        if (!posecen[v][k][d]) {
          dfs(linije, m, n, posecen, v, k, d);
          brojOblasti++;
        }
```

```
  cout << brojOblasti << endl;
  return 0;
}
```

2.4 Topološko sortiranje

Pretpostavimo da je zadat skup poslova u vezi sa čijim redosledom izvršavanja postoje neka ograničenja. Neki poslovi zavise od drugih, odnosno ne mogu se započeti pre nego što se ti drugi poslovi završe. Sve zavisnosti su poznate, a cilj je napraviti takav redosled izvršavanja poslova koji zadovoljava sva zadata ograničenja; drugim rečima, traži se redosled izvršavanja za koji važi da svaki posao započinje tek kad su završeni svi poslovi od kojih on zavisi. Na primer, želimo da sagradimo kuću. Da bismo to uspeli potrebno je da postavimo temelj, da sazidamo zidove, da stavimo krov, da uvedemo struju i vodu. Pritom, naravno, nije moguće, na primer, sazidati zidove dok se ne postavi temelj, niti uvesti vodu dok se ne stavi krov na kuću. Zadatak je odrediti neki ispravan redosled izvršavanja ovih poslova.

Dati problem ima primenu u raznim domenima: na primer, za određivanje redosleda u kom je potrebno izvršiti ponovno izračunavanje vrednosti formula u programima za tabelarna izračunavanja, za utvrđivanje redosleda u kom treba izvršiti zadatke u mejkfajlu, za unapređenje paralelizma instrukcija i slično. Zadatak je osmisliti efikasan algoritam za formiranje takvog redosleda.

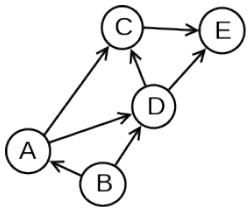
Opisani problem može se formulisati u terminima grafova i naziva se *topološko sortiranje grafa* (engl. topological sort). Naime, zadatim poslovima i njihovim međuzavisnostima može se na prirodan način pridružiti usmereni graf $G = (V, E)$: svakom poslu pridružuje se čvor, a usmerena grana od čvora x do čvora y postoji ako se posao y ne može započeti pre završetka posla x.[2] Zadatak je odrediti *topološki poredak čvorova*, odnosno numeraciju čvorova brojevima od 1 do $n = |V|$ (ili od 0 do $n - 1$) tako da za svaku granu grafa (u, v) važi da je polazni čvor u grane numerisan manjom vrednošću nego završni v čvor te grane. Graf nad kojim razmatramo ovaj problem mora biti bez usmerenih ciklusa, jer se u protivnom neki poslovi nikada ne bi mogli započeti.

Problem

U zadatom usmerenom acikličkom grafu $G = (V, E)$ sa n čvorova numerisati čvorove brojevima od 1 do n, tako da ako je proizvoljan čvor v numerisan brojem k, onda su svi čvorovi do kojih postoji usmerena grana iz čvora v numerisani brojevima većim od k.

Primer 2.4.1

Razmotrimo graf prikazan na slici 2.27.

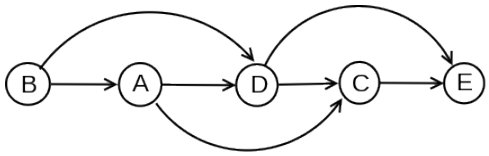


Slika 2.27: Usmereni aciklički graf u kojem postoji tačno jedno topološko uređenje čvorova.

U njemu postoji samo jedno ispravno topološko uređenje čvorova i to je B, A, D, C, E. Čvor D, recimo, mora da bude numerisan većim brojem od čvorova B i A jer postoje grane do D iz čvorova A i B. Slično

[2] Dakle, grane vode od posla koji je nezavisan, ka poslu koji je zavisan. Moguće je formirati graf tako da grane vode od zavisnog ka nezavisnom poslu. Različitim algoritmima odgovaraju različite organizacije grana, pa prilikom konstrukcije algoritma za topološko sortiranje treba obratiti pažnju na to kako su grane usmerene.

čvor D mora biti numerisan manjim brojem od čvorova C i E jer postoje grane od čvora D do čvorova C i E. Dakle, u ovom grafu redni broj čvora D mora biti 3. Graf sa slike 2.27 možemo predstaviti i pogodnije, tako da čvorovi budu poređani duž jedne prave uređeni u odnosu na topološki poredak. Tada su sve grane grafa usmerene udesno (slika 2.28).

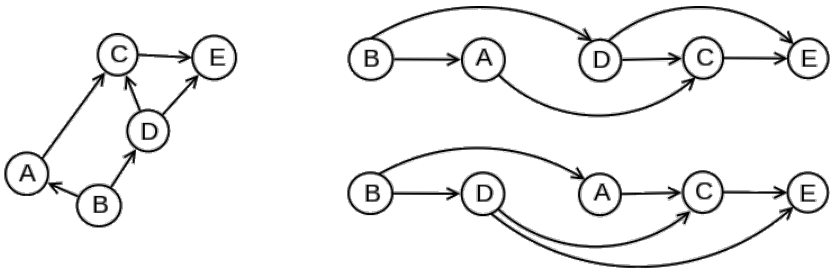


Slika 2.28: Usmereni aciklički graf kod koga su čvorovi poređani u redosledu topološkog uređenja.

U opštem slučaju može postojati veći broj ispravnih topoloških uređenja grafa.

Primer 2.4.2

Ako razmotrimo graf sa slike 2.29, levo, u njemu postoje dva ispravna topološka uređenja: B, A, D, C, E i B, D, A, C, E. Oni su prikazani na slici 2.29, desno.



Slika 2.29: Usmereni aciklički graf u kojem postoje dva različita topološka uređenja čvorova.

Razmotrićemo dva različita algoritma za određivanje topološkog uređenja u acikličkom usmerenom grafu: *Kanov algoritam* (engl. Kahn's algorithm) i *algoritam zasnovan na pretrazi grafa u dubinu*.

2.4.1 Kanov algoritam

Prirodno je započeti algoritam tako što uradimo onaj posao koji ne zavisi ni od jednog drugog posla. Nakon toga nam problem postaje jednostavniji, jer neki poslovi koji sa zavisili od tog prvog posla više ne zavise od njega. Dakle, problem rešavamo svođenjem na manji potproblem istog tipa, tj. induktivno-rekurzivnom konstrukcijom.

Prirodna je, dakle, naredna induktivna hipoteza.

Induktivna hipoteza: Umemo da numerišemo na zahtevani način čvorove svih usmerenih acikličkih grafova sa manje od n čvorova.

Bazni slučaj je slučaj praznog grafa, koji ne sadrži čvorove i on se trivijalno rešava. Kao i obično, posmatrajmo proizvoljni graf sa n čvorova, uklonimo jedan čvor iz njega, primenimo induktivnu hipotezu na preostale čvorove u grafu i pokušajmo da proširimo numeraciju na polazni graf. Ono što je važno primetiti jeste da imamo slobodu izbora čvora koji uklanjamo pa, kao što smo rekli, biramo čvor sa ulaznim stepenom nula; njemu se može dodeliti broj 1. Postavlja se pitanje da li u proizvoljnom usmerenom acikličkom grafu uvek postoji čvor sa ulaznim stepenom nula? Intuitivno se nameće potvrdan odgovor, jer se sa označavanjem negde mora započeti. Sledeća lema potvrđuje ovu činjenicu.

Lema 2.4.1

Usmereni aciklički graf uvek ima čvor ulaznog stepena nula.

Dokaz. Ako bi svi čvorovi grafa imali pozitivne ulazne stepene, mogli bismo da krenemo iz nekog čvora "unazad" prolazeći grane u suprotnom smeru. Međutim, broj čvorova u grafu je konačan, pa se u tom obilasku mora u nekom trenutku naići na neki čvor po drugi put, što znači da u grafu postoji usmereni ciklus. Ovo je suprotno pretpostavci da se radi o acikličkom grafu. Dakle u usmerenom acikličkom grafu uvek postoji čvor čiji je ulazni stepen nula.[3] □

Pretpostavimo da smo pronašli čvor čiji je ulazni stepen nula. Numerišimo ga sa 1, uklonimo sve grane koje vode iz njega, i numerišimo ostatak grafa (koji je takođe aciklički) brojevima od 2 do n: prema induktivnoj hipotezi oni se mogu numerisati brojevima od 1 do $n-1$, a zatim se svaki redni broj može povećati za jedan. Napomenimo još u ovom trenutku da nije neophodno efektivno izbacivati grane iz grafa, jer je to operacija koja se ne izvršava efikasno ako je graf predstavljen listom povezanosti. Naime, odgovarajući efekat moguće je realizovati i efikasnije, smanjivanjem za 1 ulaznog stepena čvora u koji data grana ulazi.

Dakle, problem se može rešiti sukcesivnim pronalaženjem čvorova sa ulaznim stepenom 0. Pri realizaciji ovog algoritma treba pronalaziti čvor sa ulaznim stepenom nula i ažurirati ulazne stepene čvorova posle uklanjanja grana koje polaze iz datog čvora. Drugi problem možemo rešiti tako što alociramo niz `ulazniStepen` dimenzije jednake broju čvorova u grafu i inicijalizujemo ga na vrednosti ulaznih stepena čvorova. Ulazne stepene čvorova u grafu možemo jednostavno odrediti prolaskom kroz skup svih grana u proizvoljnom redosledu i povećavanjem za jedan vrednosti `ulazniStepen[w]` svaki put kad se naiđe na neku granu (v, w). Ukoliko je graf zadat listom povezanosti, sve grane su navedene u listi povezanosti kojom je predstavljen i ovaj korak je linearne složenosti po broju grana u grafu. Prilikom uklanjanja čvora v, svim njegovim susedima w vrednost ulaznog stepena koja se nalazi u nizu na poziciji w smanjujemo za 1 (što je operacija složenosti $O(1)$). Svaka grana će biti uklonjena tačno jednom, tako da će ukupna složenost opet biti linearna po broju grana u grafu.

Pronalaženje čvora sa ulaznim stepenom nula može se sprovesti iteracijom kroz niz ulaznih stepena, međutim, složenost tog koraka je linearna u odnosu na broj čvorova, tako da bi ukupna složenost algoritma bila kvadratna. Problem možemo rešiti i efikasnije tako što tokom izvršavanja algoritma održavamo spisak svih čvorova čiji je ulazni stepen 0 – primetimo da takvih čvorova u svakom koraku može biti više. Dakle, potrebno je čvorove sa ulaznim stepenom nula čuvati u nekoj kolekciji u koju je efikasno umetati i iz koje je efikasno uklanjati elemente. U ove svrhe može se koristiti red (ili stek, što je jednako dobro). Prema prethodnoj lemi u acikličkom grafu postoji bar jedan čvor sa ulaznim stepenom nula, a pošto je graf sve vreme izvršavanja algoritma aciklički, red će u svakom trenutku, do samog kraja, biti neprazan. Svaki put kada uklonimo i numerišemo čvor v iz reda, uklanjamo i sve njegove grane (v, w) tako što vrednost `ulazniStepen[w]` smanjujemo za jedan. Ako vrednost ulaznog stepena čvora w pri tome postane nula, čvor w upisuje se u red, čime se postiže da će se svi čvorovi stepena nula u smanjenom grafu nalaziti u redu (za čvorove koji nisu bili susedni sa v od ranije znamo da su u redu ako i samo ako im je ulazni stepen bio nula, a taj ulazni stepen se nije promenio nakon uklanjanja čvora v, dok smo za čvorove susedene čvoru v upravo efektivno proverili da li im je stepen u smanjenom grafu postao nula i stavili smo ih u red ako i samo ako jeste). Algoritam završava sa radom kada red koji sadrži čvorove stepena nula postane prazan, jer su u tom trenutku svi čvorovi numerisani. Opisani algoritam zove se *Kanov algoritam*, po njegovom autoru Arturu Kanu.

Primer 2.4.3

U tabeli 2.1 ilustrovano je izvršavanje Kanovog algoritma na primeru grafa sa slike 2.29. Pretpostavljamo da je graf zadat listama povezanosti, tako da su za svaki čvor njegovi susedi poređani u leksikografski rastućem poretku. Za svaki od koraka algoritma prikazane su trenutne vrednosti ulaznih stepena čvorova grafa, sadržaj

[3] Analogno se pokazuje da u usmerenom acikličkom grafu uvek postoji čvor izlaznog stepena nula.

reda koji sadrži čvorove ulaznog stepena nula koji još uvek nisu numerisani i poslednji numerisani čvor u grafu. Primetimo da se u drugom koraku moglo desiti da se u red najpre doda čvor D, a zatim čvor A i u tom slučaju bilo bi dobijeno drugačije topološko uređenje: B, D, A, C, E.

Tabela 2.1: Primer izvršavanja Kanovog algoritma za graf sa slike 2.29. Prikazani su ulazni stepeni svih čvorova, trenutni sadržaj reda i određeni redni brojevi čvorova.

$d(A)$	$d(B)$	$d(C)$	$d(D)$	$d(E)$	*Red*	*Naredni numerisani čvor*
1	0	2	1	2	B	
0		2	0	2	A, D	$B:1$
		1		2	D	$A:2$
		0		1	C	$D:3$
				0	E	$C:4$
						$E:5$

Ako nakon završetka rada algoritma za neke čvorove važi da nisu bili dodati u red, to znači da postoji podskup skupa čvorova takav da u odgovarajućem indukovanom podgrafu svi čvorovi imaju ulazni stepen veći od nula. Stoga indukovani podgraf (a time i polazni graf) sadrži usmereni ciklus, suprotno pretpostavci da je graf aciklički.

```
void topolosko_sortiranje() {
  int brojCvorova = listaSuseda.size();
  // niz koji cuva ulazne stepene cvorova
  vector<int> ulazniStepen(brojCvorova,0);
  // niz koji cuva redne brojeve cvorova u topoloskom uredjenju
  vector<int> topoloskoUredjenje;
  // broj posecenih cvorova
  int brojPosecenih = 0;

  // inicijalizujemo niz ulaznih stepena cvorova
  for (int i = 0; i < listaSuseda.size(); i++)
    for (int j = 0; j < listaSuseda[i].size(); j++)
      ulazniStepen[listaSuseda[i][j]]++;

  // red koji cuva cvorove ulaznog stepena nula
  queue<int> cvoroviStepenaNula;

  // cvorove koji su ulaznog stepena 0 dodajemo u red
  for (int i = 0; i < brojCvorova; i++)
    if (ulazniStepen[i] == 0)
       cvoroviStepenaNula.push(i);

  while(!cvoroviStepenaNula.empty()) {
    // cvor sa pocetka reda numerisemo narednim brojem
    int cvor = cvoroviStepenaNula.front();
    cvoroviStepenaNula.pop();
    topoloskoUredjenje.push_back(cvor);

    brojPosecenih++;
```

```
    // za sve susede tog cvora azuriramo ulazne stepene
    for (int i = 0; i < listaSuseda[cvor].size(); i++) {
      int sused = listaSuseda[cvor][i];
      ulazniStepen[sused]--;
      // ako je ulazni stepen suseda postao 0, dodajemo ga u red
      if (ulazniStepen[sused] == 0)
         cvoroviStepenaNula.push(sused);
    }
  }

  // ako smo numerisali sve cvorove u grafu
  if (brojPosecenih == brojCvorova) {
     // stampamo dobijeno topolosko uredjenje
     cout << "Redosled cvorova u topoloskom uredjenju je:" << endl;
     for (int i = 0; i < brojCvorova; i++)
        cout << topoloskoUredjenje[i] << ": " << i+1 << endl;
  }
  else
     // zakljucujemo da graf sadrzi usmereni ciklus
     cout << "Graf nije aciklicki" << endl;
}

int main() {
  topolosko_sortiranje();
  return 0;
}
```

U slučaju kada je graf zadat listama povezanosti vremenska složenost inicijalizacije niza `ulazniStepen` je $O(|V|+|E|)$. U petlji `while` (kroz koju se prolazi $|V|$ puta) za pronalaženje čvora sa ulaznim stepenom nula potrebno je konstantno vreme (pristup redu). Svaka grana (v, w) razmatra se tačno jednom, u petlji kroz koju se prolazi nakon uklanjanja čvora v iz reda. Prema tome, ukupan broj promena vrednosti elemenata niza `ulazniStepen` u svim izvršavanjima spoljašnje petlje `while` jednak je broju grana u grafu. Vremenska složenost Kanovog algoritma je dakle $O(|V| + |E|)$, odnosno linearna je funkcija od veličine grafa.

2.4.2 Algoritam zasnovan na pretrazi u dubinu

Zamislimo na trenutak da je graf zadat tako da su mu grane okrenute od poslova koji zavise ka poslovima od kojih zavise i definišimo rekurzivnu proceduru koja obrađuje posao tj. dodeljuje mu redni broj u topološkom redosledu. Da bi posao za koji je funkcija pozvana mogao da bude urađen, potrebno je da se obezbedi da su svi poslovi od kojih on zavisi završeni pre njega, što znači da je potrebno da se obiđe graf krenuvši od tog posla (čvora) i da se urade svi dostupni poslovi (tj. preskoče ako su već ranije urađeni). Kada se to uradi, tek tada tekućem čvoru može biti dodeljen naredni slobodan broj. Primećujemo, dakle, da ovaj algoritam može da vrši klasičan obilazak grafa u dubinu, dodeljujući traženi broj čvoru u sklopu odlazne numeracije.

Dokažimo da se na ovaj način dobija ispravan topološki redosled.

Lema 2.4.2

Za svaku granu (u, v) acikličnog grafa važi $u.Post > v.Post$.

Dokaz. Kao što smo ranije zaključili u grafu $G = (V, E)$ važi:

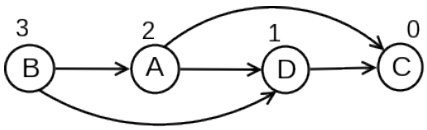
- ako je grana $(u, v) \in E$ grana DFS drveta, direktna ili poprečna grana, za nju važi $u.Post > v.Post$,

- ako je grana $(u, v) \in E$ povratna grana u odnosu na DFS drvo, za nju važi $u.Post \leq v.Post$.

Primetimo da ciklus u usmerenom grafu postoji ako i samo ako u DFS drvetu postoje povratne grane. U usmerenom acikličkom grafu ne postoji ciklus, pa ne postoje povratne grane u odnosu na DFS drvo. Dakle, za svaku granu (u, v) acikličkog grafa važi uslov $u.Post > v.Post$. □

Dakle, ako su grane usmerene od zavisnog ka nezavisnom čvoru, odlazna numeracija daje ispravan topološki redosled. Implementacija je sasvim jednostavna: u klasičnom DFS obilasku prilikom izlazne obrade čvoru treba dodati redni broj u topološkom redosledu tj. dodati ga na kraj niza u kom se čvorovi čuvaju u skladu sa topološkim redosledom. U glavnoj funkciji pokrećemo DFS obilazak svakog čvora (u proizvoljnom redosledu), preskačući čvorove koji su već ranije obrađeni (slično kao kod određivanja komponenata povezanosti).

Vratimo se na standardnu postavku problema u kom su grane grafa orijentisane od nezavisnog ka zavisnom čvoru. Ako sa $t(x)$ označimo redni broj čvora x u topološkom poretku grafa G, za svaku granu (u, v) potrebno je da važi $t(u) < t(v)$. Pošto za svaku granu (u, v) važi $u.Post > v.Post$, ako čvorove grafa uredimo u opadajućem redosledu u odnosu na odlaznu numeraciju čvorova, dobićemo jedno topološko uređenje grafa. Implementaciju je potrebno izmeniti samo tako što se u odlaznoj obradi čvorovima dodeljuju brojevi unazad, od n do 1. Ako čvorove ne numerišemo, već ih stavljamo u niz u skladu sa topološkim redosledom, oni će u nizu biti složeni naopako i niz je nakon obrade potrebno obrnuti tj. čvorove obrađivati od kraja ka početku tog niza. Alternativno, čvorove možemo postavljati na namenski stek i obrađivati ih u topološkom redosledu tako što ćemo ih jedan po jedan skidati sa tog steka. Moguće je koristiti i neku strukturu podataka koja dopušta efikasno dodavanje elemenata na početak (npr. listu ili red sa dva kraja).



Slika 2.30: Usmereni aciklički graf: uz svaki čvor prikazana je vrednost njegove odlazne numeracije u odnosu na DFS pretragu pokrenutu iz čvora B.

Primer 2.4.4

Razmotrimo graf sa slike 2.30: on je usmeren i aciklički. Ako pokrenemo DFS pretragu iz čvora B redosled čvorova u kojima napuštamo čvorove je C, D, A, B. Dakle, topološko uređenje grafa dobijamo obrtanjem ovog redosleda, odnosno redosled čvorova u topološkom poretku biće B, A, D, C. Primetimo da to odgovara i redosledu čvorova sleva nadesno u prikazu grafa kod koga su sve grane usmerene sleva udesno.

```
void dfs(int cvor, vector<bool> &posecen, vector<int> &odlazna) {
  posecen[cvor] = true;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor]) {
    if (!posecen[sused])
       dfs(sused,posecen,odlazna);
  }

  // u vektor odlazna dodajemo na kraj naredni cvor
  // koji napustamo pri DFS obilasku
  odlazna.push_back(cvor);
}
```

```
void topolosko_sortiranje() {
  int brojCvorova = listaSuseda.size();
  vector<bool> posecen(brojCvorova);
  // niz koji sadrzi redom cvorove prema redosledu napustanja
  vector<int> odlazna;

  for (int cvor = 0; cvor < brojCvorova; cvor++)
    if (!posecen[cvor])
       dfs(cvor,posecen,odlazna);

  // cvorove ispisujemo u opadajućem redosledu odlazne numeracije
  cout << "Redosled cvorova u topoloskom uredjenju je:" << endl;
  for (int i = brojCvorova - 1; i >= 0; i--)
    cout << odlazna[i] << ": " << brojCvorova - i << endl;

}

int main() {
  topolosko_sortiranje();
  return 0;
}
```

S obzirom na to da se prikazani algoritam svodi na DFS pretragu i određivanje odlazne numeracije čvorova, njegova vremenska složenost iznosi $O(|E| + |V|)$.

2.5 Mostovi i artikulacione tačke u neusmerenom grafu

U ovom poglavlju bavićemo se pitanjem koliko je dati neusmereni graf dobro povezan. Drugim rečima interesuje nas da li u grafu postoji slaba tačka, odnosno usko grlo, čijim bi uklanjanjem graf prestao da bude povezan. U raznim praktičnim primenama ovaj problem je veoma značajan i potrebno je omogućiti da se uska grla u grafu brzo detektuju. Slična pitanja se mogu postaviti i za usmerene grafove, međutim, mi ćemo se u ovom poglavlju ograničiti na slučaj neusmerenih grafova.

Granu neusmerenog grafa $G = (V, E)$ čijim se uklanjanjem iz grafa broj komponenti povezanosti grafa povećava nazivamo *most* (engl. bridge, cut edge). Specijalno, povezani graf nakon uklanjanja mosta prestaje da bude povezan.

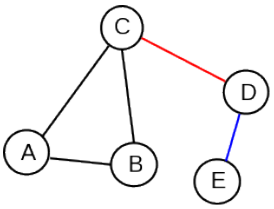
U računarskim mrežama pronalaženje mostova može pomoći u identifikovanju kritičnih veza čiji kvar može dovesti do raspada mreže. U transportnim sistemima mostovi predstavljaju ključne rute koje povezuju različite oblasti, a čijim bi uklanjanjem transport bio onemogućen.

Primer 2.5.1

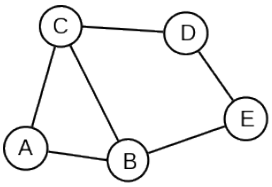
Ukoliko u grafu prikazanom na slici 2.31 uklonimo granu (C, D) ili granu (D, E) graf prestaje da bude povezan, te ove dve grane, svaka za sebe, čine most u datom grafu.

Postoje grafovi u kojima nema mostova (slika 2.32).

Ukoliko u neusmerenom grafu $G = (V, E)$ postoji čvor $v \in V$ takav da se njegovim uklanjanjem iz grafa (zajedno sa granama koje su mu susedne) broj komponenti povezanosti grafa povećava, onda takav čvor nazivamo *artikulacionom tačkom* (engl. articulation point, cut vertex). Specijalno, povezani graf nakon uklanjanja artikulacione tačke prestaje da bude povezan.



Slika 2.31: Primer grafa koji sadrži dva mosta: jedan je označen crvenom, a drugi plavom bojom.

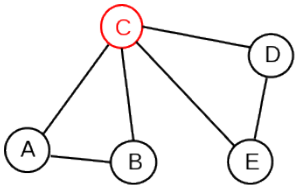


Slika 2.32: Primer grafa koji ne sadrži ni most ni artikulacionu tačku.

U transportnim sistemima artikulacione tačke ukazuju na kritične raskrsnice čija bi eventualna nedostupnost (npr. u slučaju neke nesreće) imala značajan uticaj na tok saobraćaja. Na ovaj način mogu se planirati prioritetne investicije u saobraćajnu infrastrukturu. Slično, artikulacione tačke u električnim kolima predstavljaju komponente čijim bi otkazivanjem rada došlo do gubitka veze između ostalih komponenti. Identifikovanje ovih komponenti može poboljšati pouzdanost dizajna električnog kola. U kontekstu društvenih mreža artikulacione tačke predstavljaju osobe koje povezuju dve ili više različitih zajednica ljudi i čije bi uklanjanje razbilo mrežu na nepovezane grupe.

Primer 2.5.2

Ukoliko u povezanom grafu prikazanom na slici 2.33 uklonimo čvor C, graf prestaje da bude povezan, te je čvor C jedna artikulaciona tačka ovog grafa. Štaviše, neposredno se proverava da je čvor C jedina artikulaciona tačka u ovom grafu.



Slika 2.33: Graf koji sadrži jednu artikulacionu tačku: čvor C.

Graf može da ne sadrži artikulacione tačke (slika 2.32), a može i da sadrži veći broj artikulacionih tačaka (slika 2.34).

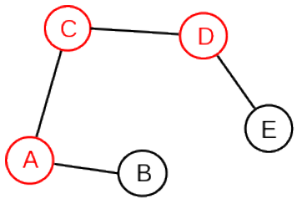
U nastavku ćemo razmotriti probleme pronalaženja svih mostova i artikulacionih tačaka u datom neusmerenom grafu. Bez smanjenja opštosti može se pretpostaviti da je dati graf povezan. Ako graf nije povezan, onda se mostovi i artikulacione tačke mogu tražiti nezavisno u svakoj komponenti povezanosti grafa.

2.5.1 Određivanje mostova

Pozabavimo se najpre problemom određivanja mostova, za koji se pokazuje da je donekle jednostavniji.

Problem

Definisati algoritam koji u datom neusmerenom povezanom grafu određuje sve mostove.



Slika 2.34: Graf koji sadrži veći broj artikulacionih tačaka: to su čvorovi A, C i D. Svaka grana ovog grafa je most.

Direktan način da se u datom neusmerenom povezanom grafu $G = (V, E)$ pronađu svi mostovi podrazumeva da za svaku granu $e \in E$ grafa G proverimo da li je graf bez grane e povezan (npr. korišćenjem algoritma DFS). Složenost ovog algoritma iznosi $O(|E| \cdot (|V| + |E|))$.[4]

Postoji efikasniji algoritam za određivanje mostova u grafu. Mi ćemo u nastavku razmotriti algoritam koji je osmislio Robert Tardžan (engl. Robert Tarjan) i koji je linearne vremenske složenosti u funkciji veličini grafa.

2.5.1.1 Tardžanov algoritam za određivanje mostova

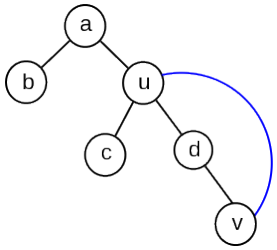
Mostovi ne pripadaju ciklusima. Zaista, važi naredno trivijalno tvrđenje (koje navodimo bez dokaza).

Lema 2.5.1 **[Veza mostova i ciklusa]**

Grana (u, v) je most u grafu G ako i samo ako ne pripada nijednom ciklusu u grafu G.

Specijalno, pošto u drvetu ne postoje ciklusi, ako je graf drvo, onda je svaka grana u tom grafu most (na primer, takav je graf na slici 2.34).

Razmotrimo DFS drvo dobijeno DFS obilaskom datog grafa $G = (V, E)$. S obzirom na to da je polazni graf neusmeren, postoje dve vrste grana grafa u odnosu na DFS drvo: grane DFS drveta i grane koje povezuju potomka sa pretkom u odnosu na DFS drvo (u nastavku ćemo ih, jednostavnosti radi, zvati povratne grane, mada one nisu usmerene, pa se mogu istovremeno smatrati i za direktne i za povratne grane). Ako je grana (u, v) povratna, ona ne može biti most u grafu, jer je deo ciklusa koji ta grana čini sa granama DFS drveta (videti primer na slici 2.35).



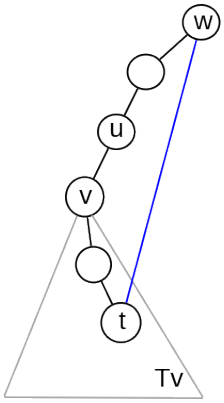
Slika 2.35: Grana (u, v) koja povezuje potomka i pretka u DFS drvetu ne može biti most.

Dakle, mostovi mogu biti samo grane DFS drveta, te je dovoljno da algoritam razmatra samo njih kao kandidate, čime se može značajno smanjiti broj grana koje je potrebno proveravati. Međutim, ni ove grane ne moramo proveravati tehnikom grube sile. Neka je (u, v) grana DFS drveta. Pretpostavimo da je DFS pretraga najpre posetila čvor u pa čvor v, tj. da je čvor u roditelj čvora v u DFS drvetu grafa.

Za granu (u, v) DFS drveta važi da je most ako i samo ako ne postoji ni jedna druga grana između nekog pretka čvora u (uključujući i čvor u) i nekog potomka čvora v (uključujući i čvor v).

[4] Primetimo da je uklanjanje grane iz grafa u slučaju kada je graf zadat listama povezanosti neefikasno, ali pošto se nakon uklanjanja svake grane izvršava algoritam pretrage u dubinu, ova operacija ne utiče na složenost kompletnog algoritma.

Dakle, grana (u, v) je most ako i samo ako poddrvo DFS drveta čiji je koren čvor v (koje je "iznad"[5] čvora v) ostaje nepovezano sa delom grafa "ispod" ove grane tj. čvora u. Drugim rečima tada ne postoji način da se stigne iz poddrveta T_v čiji je koren čvor v do čvora u ili nekog pretka čvora u.



Slika 2.36: Ilustracija definicije vrednosti $L(v)$.

Potrebno je za svaki čvor $v \in V$ izračunati koliko se "nisko" možemo vratiti nekom povratnom granom iz proizvoljnog čvora poddrveta T_v čiji je koren čvor v. To možemo kvantifikovati na osnovu dolaznih DFS brojeva čvorova do kojih vode povratne grane iz T_v ("niži" čvorovi tj. čvorovi bliži korenu imaju i niže DFS brojeve). Kao što je ranije opisano, za svaki čvor $v \in V$ može se odrediti vrednost $v.Pre$ koja označava redni broj čvora v pri dolaznoj DFS numeraciji – tu vrednost ćemo u nastavku kraće zvati *rednim brojem čvora* v. Cilj nam je da za svaki čvor v odredimo i redni broj *"najnižeg" pretka dostižnog povratnom granom* (engl. lowlink) do kog se možemo popeti nekom (najviše jednom) povratnom granom iz poddrveta T_v (slika 2.36). Naglasimo da ovo ne mora biti "najniži" dostižni predak, jer se uz korišćenje više povratnih grana možda može stići do nekog "nižeg" pretka tj. pretka koji ima manji DFS broj i bliže je korenu drveta.

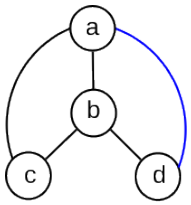
Definišimo sada precizno pojam "najnižeg" pretka dostižnog povratnom granom[6]. Označimo sa $L(v)$ (engl. lowlink) manju od vrednosti rednog broja čvora v i najmanje među vrednostima rednih brojeva čvorova do kojih se može stići povratnom granom iz proizvoljnog čvora poddrveta T_v (koje uključuje i čvor v), koja nije grana DFS drveta koja vodi od v ka njegovom roditeljskom čvoru u DFS drvetu. Moguće je da iz poddrveta T_v ne postoji nijedna grana ka pretku čvora v i tada je vrednost $L(v)$ jednaka rednom broju čvora v. Dakle važi sledeće:

$$L(v) = \min\{v.Pre, \min_{\substack{w \text{ je predak } v \\ \text{postoji grana } (t,w),\ t \in T_v, \\ \text{koja ne spaja } v \text{ sa njegovim roditeljem}}} w.Pre\}.$$

Vrednost $L(v)$ je definisana kao redni broj čvora, međutim, pošto postoji jasna bijekcija između čvorova i njihovih rednih brojeva možemo slobodno reći da je "najniži" predak čvora v dostižan povratnom granom onaj čvor čiji je redni broj $L(v)$. Putanja od čvora v do njegovog "najnižeg" pretka dostižnog povratnom granom se uvek sastoji od niza grana DFS drveta za kojima sledi najviše jedna povratna grana. "Najniži" predak dostižan povratnom granom je uvek jedinstveno određen, ali putanja do njega ne mora biti (primer je dat na slici 2.37).

[5] Pošto se drvo u ovom tekstu crta naopako, tako da mu je koren na vrhu crteža, a listovi na dnu, deo grafa "iznad" nekog čvora je nacrtan ispod tog čvora. U nastavku će biti korišćena terminologija u kojoj su termini iznad/ispod, niži/viži, gore/dole definisani u skladu sa položajem korena i listova, a ne u skladu sa naopakim crtanjem drveta. Ovo je u skladu i sa tim da je visina korena 0 i da visina čvorova raste kako se udaljavamo od korena. Da bismo smanjili mogućnost zabune, ove odrednice će biti pisane pod navodnicima.

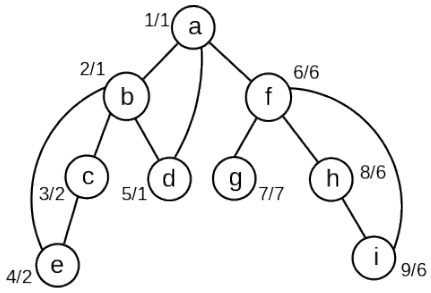
[6] Naglasimo da se pretpostavlja da se radi o neusmerenom grafu. U usmerenim grafovima je moguće definisati srodan pojam, ali je definicija komplikovanija i biće izložena u poglavlju 2.6.



Slika 2.37: Najniži dostižni čvor povratnom granom čvora b je čvor a. Do njega se može stići putevima (b, c, a) i (b, d, a).

Primer 2.5.3

Na slici 2.38 dat je primer grafa gde je uz svaki čvor v prikazan njegov redni broj v.Pre i vrednost $L(v)$. Na primer, važi $L(c) = 2$ i $L(e) = 2$ jer iz čvora e granom (e, b) možemo stići do čvora b čija je redni broj jednak 2. Slično, važi $L(b) = 1$ jer iz čvora d koji je potomak čvora b možemo stići granom (d, a) do čvora a čiji je redni broj 1.



Slika 2.38: Primer grafa i odgovarajućeg DFS drveta. Uz svaki čvor v prikazan je njegov redni broj i vrednost $L(v)$.

Sada možemo dati karakterizaciju mostova pomoću funkcije L. Poddrvo T_v DFS drveta čiji je koren u čvoru v ostaće nakon izbacivanja grane (u, v) nepovezano sa delom grafa "ispod" ove grane ako i samo ako važi $L(v) > u$.*Pre*. Dakle, važi naredna teorema.

Teorema 2.5.1 **[Karakterizacija mosta u grafu pomoću funkcije L]**

Grana (u, v) je most u grafu G ako i samo ako važi $L(v) > u$.Pre.

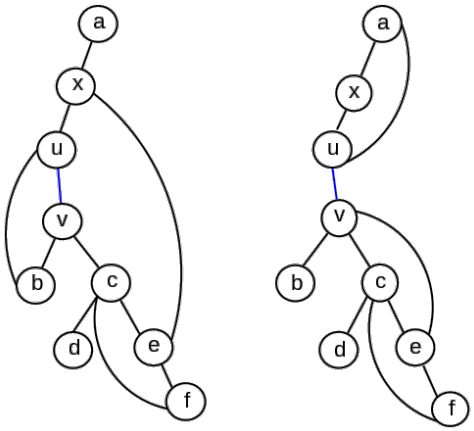
Primer 2.5.4

Razmotrimo graf prikazan na slici 2.39, levo.

Grana (u, v) nije most u grafu jer se iz poddrveta T_v možemo vratiti u deo DFS drveta "ispod" ove grane. Preciznije, iz čvora b možemo se vratiti u čvor u, a iz čvora e u čvor x. Pošto je čvor x "ispod" čvora u, važi x.Pre $< u$.Pre i vrednost $L(v)$ jednaka je rednom broju x.Pre čvora x. Stoga nije zadovoljen uslov $L(v) > u$.Pre, pa grana (u, v) nije most. Čak i kad graf ne bi sadržao granu (x, e), nakon izbacivanja grane (u, v) mogli bismo se granom (b, u) iz poddrveta T_v vratiti do čvora u, a time i do proizvoljnog čvora "ispod" njega u DFS drvetu, te i u tom slučaju grana (u, v) ne bi bila most u grafu (važilo bi da je $L(v) = u$.Pre, pa ni tada ne bi važilo $L(v) > u$.Pre).

Razmotrimo sada graf sa slike 2.39, desno. Nakon izbacivanja grane (u, v) iz grafa, iz poddrveta T_v možemo se vratiti "najniže" do čvora v, tj. važi uslov $L(v) = v$.Pre $> u$.Pre i u ovom grafu grana (u, v) jeste most.

Primetimo da, takođe, važi da je grana (u, v) most ako i samo ako je $L(v) = v$.



Slika 2.39: Levo: primer grafa u kome grana (u, v) nije most, desno: primer grafa u kome grana (u, v) jeste most.

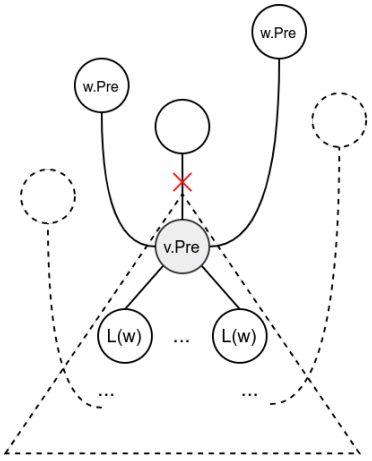
Ostaje pitanje kako efikasno izračunati vrednosti $L(v)$ za sve čvorove v u grafu. Važi sledeća lema.

Lema 2.5.2 **[Rekurzivna veza za L]**

Za svaki čvor $v \in V$ neusmerenog grafa $G = (V, E)$ važi:

$$L(v) = \min\{v.Pre, \min_{\substack{(v,w)\in E \\ w \text{ je predak cvora } v \\ w \text{ nije roditelj cvora } v}} w.Pre, \min_{w \text{ je dete od } v} L(w)\} \tag{2.1}$$

Dokaz. Razmatrajmo proizvoljni čvor v i putanju od njega do "najnižeg" dostižnog čvora povratnom granom. Ta putanja je ili prazna ili se sastoji od tačno jedne povratne grane ili se sastoji od nekoliko grana DFS drveta za kojima sledi tačno jedna povratna grana. U jednakosti (2.1) se analiziraju sve moguće takve putanje i traži se ona najbolja.



Slika 2.40: Ilustracija jednakosti (2.1)

Zaista, prazna putanja je obuhvaćena izrazom $v.Pre$, a putanje koje se sastoje tačno od jedne povratne grane izrazom

$$\min_{\substack{(v,w)\in E \\ w \text{ je predak od } v \\ w \text{ nije roditelj od } v}} w.Pre.$$

Ako putanja sadrži grane DFS drveta, nakon prelaska prve grane te putanje stiže se do nekog čvora w (koji nije roditelj čvora v), a zatim se nastavlja putanjom od čvora w do "najnižeg" dostižnog čvora povratnom granom krenuvši od čvora w. Zaista, do svakog čvora dostižnog povratnom granom iz w, može se stići putanjom i iz čvora v koja se završava povratnom granom, a ako bi se od čvora w moglo doći do nekog "nižeg" čvora putanjom sa povratnom granom, do njega bi se moglo stići putanjom sa povratnom granom i iz v (slika 2.40). Putanje od v koje sadrže bar jednu granu drveta se, dakle, analiziraju izrazom:

$$\min_{w \text{ je dete od } v} L(w).$$

□

Na osnovu ovoga, vrednosti funkcije L se mogu odrediti tokom DFS obilaska grafa. Definišemo rekurzivnu funkciju koja vrši DFS obilazak takvu da će nakon završetka njenog rekurzivnog poziva za proizvoljni čvor v grafa biti određene vrednosti $v.Pre$ i $L(v)$. Prilikom ulazne obrade čvora v dodeljuje mu se ulazni broj $v.Pre$ i vrednost $L(v)$ se inicijalizuje na tu vrednost. Nakon toga se obrađuju sve grane (v, w). Prilikom obrade grane (v, w), radi se sledeće:

- Ako čvor w nije posećen, vrši se rekurzivno njegov obilazak i grana (v, w) postaje grana DFS drveta. Nakon rekurzivnog poziva (u kom se vrši obrada kompletnog poddrveta čiji je koren w), izračunata je vrednost $L(w)$, pa vrednost $L(v)$ ažuriramo na vrednost $\min\{L(v), L(w)\}$.
- Ako je čvor w ranije posećen i nije roditelj čvora v, ažuriramo vrednost $L(v)$ na vrednost $\min\{L(v), w.Pre\}$ (ovim se obrađuje svaka grana ka pretku w čvora v, a ako je w posećeni potomak čvora v, važi $w.Pre > v.Pre \geq L(v)$, pa do ažuriranja neće doći).

Indukcijom, uz korišćenje leme 2.5.2, lako se može dokazati korektnost opisanog algoritma.

Teorema 2.5.2 **[Korektnost algoritma izračunavanja funkcije L]**

Prethodnim algoritmom se ispravno izračunavaju vrednosti $L(v)$ za sve čvorove v.

Implementacija u nastavku određuje dolazne redne brojeve svih čvorova, redne brojeve njihovih "najnižih" predaka dostižnih povratnom granom tj. funkcije L i sve mostove u jednom DFS obilasku.

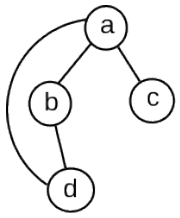
```
int vreme_dolazna = 0;
vector<bool> posecen;
vector<int> dolazna;
vector<int> lowlink;
vector<int> roditelj;
// niz grana koje su mostovi u grafu
vector<pair<int,int>> mostovi;

void dfs_mostovi(int cvor) {
  // dodeljujemo redni broj cvoru 'cvor'
  posecen[cvor] = true;
  dolazna[cvor] = vreme_dolazna++;
  // inicijalizujemo vrednost funkcije L cvora 'cvor' na njegov redni broj
  lowlink[cvor] = dolazna[cvor];
```

```
  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor]) {
    // ukoliko je sused vec posecen
    if (posecen[sused]) {
      // ako grana ne vodi ka roditelju datog cvora
      if (sused != roditelj[cvor])
         // po potrebi azuriramo vrednost L cvora na redni broj suseda
         if (dolazna[sused] < lowlink[cvor])
             lowlink[cvor] = dolazna[sused];
    } else {
    // ukoliko sused nije posecen

      // pamtimo granu DFS drveta
      roditelj[sused] = cvor;
      // pokrecemo pretragu iz cvora 'sused'
      dfs_mostovi(sused);

      // nakon obrade poddrveta čiji je koren cvor 'sused' vrednost L cvora 'sused' je odredjena;
      // po potrebi azuriramo vrednost L za cvor 'cvor'
      if (lowlink[sused] < lowlink[cvor])
         lowlink[cvor] = lowlink[sused];

      // proveravamo da li je grana (cvor, sused) most
      // u ovom trenutku su vrednosti L cvora 'sused' i rednog broja cvora 'cvor' odredjene
      if (lowlink[sused] > dolazna[cvor])
         mostovi.emplace_back(cvor, sused);
    }
  }
}

void ispisi_mostove(int cvor) {
  int brojCvorova = listaSuseda.size();
  posecen.resize(brojCvorova, false);
  dolazna.resize(brojCvorova);
  lowlink.resize(brojCvorova);
  roditelj.resize(brojCvorova, -1);

  dfs_mostovi(cvor);

  cout << "Mostovi u grafu su: ";
  for (int i = 0; i < mostovi.size(); i++)
   cout << "(" << mostovi[i].first << ", " << mostovi[i].second << ") " ;
  cout << endl;
}
```

Primer 2.5.5

U nastavku je ilustrovano izvršavanje opisanog algoritma na primeru jednostavnog neusmerenog grafa sa slike 2.41. Pretpostavlja se da je graf zadat listama povezanosti tako da su susedi svakog čvora uređeni leksikografski rastuće i da DFS pretraga započinje iz čvora a.

- *Pokrećemo DFS iz čvora a, postavljamo $a.Pre = 1$ i $L(a) = 1$*



Slika 2.41: Primer grafa koji sadrži jedan most: granu (a, c)

- *Razmatramo suseda b čvora a.*
  Pokrećemo DFS iz čvora b, postavljamo b.Pre $= 2$ i $L(b) = 2$
  - *Razmatramo suseda a čvora b.*
    To je grana ka roditelju koju dalje ne obrađujemo.
  - *Razmatramo suseda d čvora b.*
    Pokrećemo DFS iz čvora d, postavljamo d.Pre $= 3$ i $L(d) = 3$
    - *Razmatramo suseda a čvora d*
      Grana (d, a) je grana od potomka ka pretku, pa postavljamo $L(d) = a$.Pre, i dobijamo $L(d) = 1$.
    - *Razmatramo suseda b čvora d.*
      To je grana ka roditelju, koju dalje ne obrađujemo.
  - *Vraćamo se u čvor b.*
    Posto važi $L(d) < L(b)$ postavljamo $L(b) = L(d)$, pa važi i $L(b) = 1$
    Pošto je $L(d) < b$.Pre, grana (b, d) nije most.
- *Vraćamo se u čvor a.*
  Pošto važi $L(b) = L(a)$, ne radimo ništa.
  Pošto je $L(b) = a$.Pre, grana (a, b) nije most.
- *Razmatramo suseda c čvora a.*
  Pokrećemo DFS iz čvora c, postavljamo c.Pre $= 4$ i $L(c) = 4$
  - *Razmatramo suseda a čvora c*
    To je grana ka roditelju koju dalje ne obrađujemo.
- *Vraćamo se u čvor a.*
  Pošto važi $L(c) > L(a)$, ne radimo ništa.
  Pošto je $L(c) > a$.Pre, grana (a, c) jeste most.
- *Razmatramo suseda d čvora a.*
  Pošto važi $L(d) = L(a)$, ne radimo ništa.

Tardžanov algoritam za određivanje svih mostova u grafu se zasniva na DFS pretrazi, sa odgovarajućom dolaznom i odlaznom obradom koja je složenosti $O(1)$, pa je vremenska složenost ovog algoritma $O(|V| + |E|)$.

2.5.2 Određivanje artikulacionih tačaka

Pozabavimo se sada problemom određivanja artikulacionih tačaka.

Problem

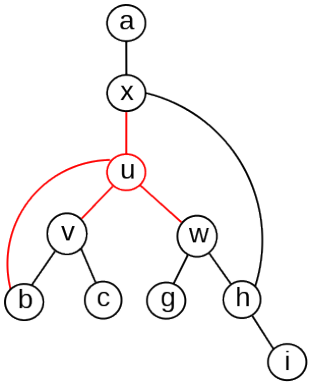
Definisati algoritam koji u datom neusmerenom povezanom grafu određuje sve artikulacione tačke.

Artikulacione tačke u datom neusmerenom povezanom grafu možemo da odredimo tako što za svaki čvor $v \in V$ grafa G izvršimo proveru da li je graf bez čvora v povezan. Složenost ovog direktnog algoritma je $O(|V| \cdot (|V| + |E|))$. Umesto njega, razmotrićemo Tardžanov algoritam, koji je dosta efikasniji.

2.5.2.1 Tardžanov algoritam za određivanje artikulacionih tačaka

Veoma nalik Tardžanovom algoritmu za određivanje mostova je i algoritam za traženje artikulacionih tačaka u grafu. Pokušajmo da formulišemo kriterijum da je čvor artikulaciona tačka kroz nekoliko primera.

Primer 2.5.6

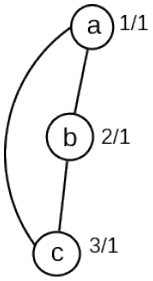


Slika 2.42: Primer grafa u kome je čvor u artikulaciona tačka.

Razmotrimo graf sa slike 2.42. Čvor u ima dva deteta u DFS drvetu: v i w. Nakon izbacivanja čvora u i njemu susednih grana iz grafa, iz poddrveta T_w možemo se vratiti u deo grafa "ispod" čvora u (jer je $L(w) = x.Pre$), međutim, iz poddrveta T_v možemo se vratiti "najniže" do čvora u (jer važi $L(v) = u.Pre$). S obzirom na to da čvor u ima dete v tako da nijedan čvor iz poddrveta T_v nije povezan sa nekim pretkom čvora u, čvor u jeste artikulaciona tačka u grafu.

Primer 2.5.7

Primetimo da za koren a DFS drveta grafa sa slike 2.43 i njegovo dete b važi uslov $L(b) = a.Pre$, tj. čvor a ima dete iz kojeg se ne možemo povratnom granom vratiti "niže" od čvora a, a pritom čvor a nije artikulaciona tačka.



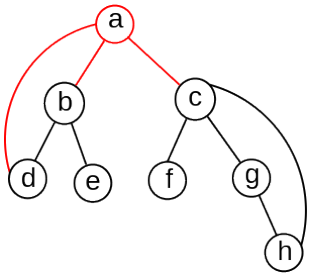
Slika 2.43: Ilustracija grafa u kome za koren DFS drveta a i njegovo dete b važi uslov $L(b) \geq a.Pre$, a koren a DFS drveta nije artikulaciona tačka.

Razlog je to što je a koren DFS drveta, pa ne postoji deo DFS drveta "ispod" njega. Dakle, slučaj čvora koji je koren DFS drveta mora se zasebno razmatrati.

Primer 2.5.8

Razmotrimo graf sa slike 2.44.

Čvor a kao koren DFS drveta ima dva deteta i nakon njegovog izbacivanja graf postaje nepovezan. Dakle, čvor a je artikulaciona tačka u grafu.



Slika 2.44: Primer grafa u kome je čvor a kao koren DFS drveta artikulaciona tačka.

Karakterizacija artikulacionih tačaka data je narednom lemom.

Lema 2.5.3 **[Karakterizacija artikulacionih tačaka]**

Čvor u je artikulaciona tačka grafa ako i samo ako je ispunjen jedan od naredna dva uslova:

- *u nije koren DFS drveta i ima dete v u DFS drvetu takvo da nijedan čvor u poddrvetu T_v nije povezan sa nekim pretkom čvora u.*
- *u je koren DFS drveta i ima bar dva deteta;*

Prvi uslov odgovara situaciji kada nakon izbacivanja čvora u iz grafa više nije moguće doći iz poddrveta T_v čiji je koren v dete čvora u, do nekog pretka čvora u. Ako je zadovoljen drugi uslov, s obzirom na to da u neusmerenim grafovima ne postoje poprečne grane, izbacivanje korena DFS drveta dovelo bi do "razbijanja" grafa na veći broj komponenti povezanosti (po jednu za svako dete korena DFS drveta).

Iz primera je jasno da se, kao i u slučaju mostova, za ispitivanje povezanosti elemenata poddrveta sa precima čvora mogu koristiti vrednosti $L(v)$, čime se dobija naredna teorema.

Teorema 2.5.3 **[Karakterizacija artikulacionih tačaka pomoću vrednosti funkcije L]**

Čvor u je artikulaciona tačka u grafu ako i samo ako važi jedan od naredna dva uslova:

- *u nije koren DFS drveta i za neko dete v čvora u važi uslov $L(v) \geq u$.Pre.*
- *u je koren DFS drveta i ima bar dva deteta;*

U nastavku je data implementacija Tardžanovog algoritma kojim se u datom grafu određuju sve artikulacione tačke.

```
int vreme_dolazna = 0;
vector<bool> posecen;
vector<int> dolazna;
vector<int> lowlink;
vector<int> roditelj;
vector<bool> artikulacioneTacke;

void dfs_artikulacione(int cvor) {
  // dodeljujemo redni broj cvoru 'cvor'
  posecen[cvor] = true;
  dolazna[cvor] = vreme_dolazna ++;

  // broj dece cvora cvor
  int broj_dece = 0;

  // vrednost funkcije L za cvor 'cvor' inicijalizujemo na njegov redni broj
```

```
  lowlink[cvor] = dolazna[cvor];

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor]) {
    // ako je grana (cvor, sused) povratna i ne vodi ka roditelju cvora 'cvor'
    if (posecen[sused]) {
       if (sused != roditelj[cvor])
          // po potrebi azuriramo vrednost L za cvor 'cvor'
          if (dolazna[sused] < lowlink[cvor])
             lowlink[cvor] = dolazna[sused];
    } else {
      // 'sused' je novo dete cvora 'cvor' u DFS drvetu
      broj_dece++;
      // dodajemo granu u DFS drvo
      roditelj[sused] = cvor;

      // pokrecemo pretragu iz cvora 'sused'
      dfs_artikulacione(sused);

      // nakon obrade poddrveta čiji je koren cvor 'sused' vrednost funkcije L
      // cvora 'sused' je odredjena, pa po potrebi azuriramo vrednost L cvora 'cvor'
      if (lowlink[sused] < lowlink[cvor])
         lowlink[cvor] = lowlink[sused];

      // ako 'cvor' nije koren DFS drveta i ako cvor nijedan cvor u poddrvetu cvora 'sused'
      // nije povezan sa nekim pretkom cvora 'cvor' onda je 'cvor' artikulaciona tacka
      if (roditelj[cvor] != -1 && lowlink[sused] >= dolazna[cvor])
         artikulacioneTacke[cvor] = true;
    }
  }
  // obradjena su sva deca cvora 'cvor'
  // proveravamo da li je cvor 'cvor' koren DFS drveta i da li ima vise od jednog deteta
  if (roditelj[cvor] == -1 && broj_dece > 1)
    // ako je uslov ispunjen, cvor je artikulaciona tacka
    artikulacioneTacke[cvor] = true;
}

void ispisi_artikulacione_tacke(int cvor) {
  int brojCvorova = listaSuseda.size();
  posecen.resize(brojCvorova, false);
  dolazna.resize(brojCvorova);
  lowlink.resize(brojCvorova);
  roditelj.resize(brojCvorova, -1);
  // inicijalno nijedan cvor nije artikulaciona tacka
  artikulacioneTacke.resize(brojCvorova, false);

  dfs_artikulacione(cvor);

  cout << "Artikulacione tacke u grafu su: ";
  for (int i = 0; i < artikulacioneTacke.size(); i++) {
    if (artikulacioneTacke[i])
```

```
      cout << i << " ";
 }
 cout << endl;
}
```

Zadatak: Usmeravanje puteva

U jednom gradu su ulice uske i stvara se gužva u saobraćaju. Gradske vlasti su odlučile da sve ulice postanu jednosmerne, u nadi da će se time povećati protočnost, međutim, nisu sigurni da li je moguće da usmere puteve tako da se i dalje može stići od bilo koje, do bilo koje druge tačke u gradu.

Opis ulaza

Sa standardnog ulaza se učitava broj tačaka u gradu n ($1 \leq n \leq 10^5$), a zatim broj dvosmernih puteva između njih m ($1 \leq m \leq 10^5$). U narednih m linija nalaze se po dva različita broja a_i i b_i ($1 \leq a_i, b_i \leq n$, $a_i \neq b_i$), koji predstavljaju redne brojeve tačaka spojenih ulicom.

Opis izlaza

Na standardni izlaz ispisati 0 ako nije moguće usmeriti ulice ili m parova tačaka koji predstavljaju usmerene ulice.

Primer 1

Ulaz

```
3
3
1 2
2 3
1 3
```

Izlaz

```
1 2
2 3
3 1
```

Primer 2

Ulaz

```
4
4
1 2
2 3
3 1
1 4
```

Izlaz

```
0
```

Rešenje

Problem se sasvim direktno modeluje neusmerenim grafom u kom su čvorovi tačke u gradu, a grane dvosmerni putevi između njih. Po pretpostavkama zadatka polazni graf je povezan.

Ključni uvid za rešenje zadatka je da je usmeravanje puteva moguće ako i samo ako u grafu ne postoji most. Naime, ako u grafu postoji most, kako god da ga usmerimo, neće postojati drugi put između dela drveta

iznad i ispod tog mosta. Ako u grafu ne postoji most, tada je moguće usmeriti sve grane drveta naniže, a sve povratne grane naviše, čime bi se dobila veza između svih čvorova grafa. Naime, pošto je polazni graf povezan, od korena je moguće kroz drvo stići do bilo kog čvora drveta (tj. grafa). Od svakog čvora je moguće povratnom granom vratiti se u neki deo drveta "ispod" tog čvora, pa se krećući se na taj način može stići i do korena. Dakle, bilo koji čvor i koren su obostrano dostižni, pa su i svi čvorovi međusobno obostrano dostižni.

Pošto se tokom određivanja mostova pamte roditelji svih čvorova, grane drveta možemo, na primer, prepoznati korišćenjem niza roditelja.

```
// određujemo sve mostove grafa
dfs_mostovi(0);

if (mostovi.size() > 0)
  // ako u grafu ima mostova, ulice se ne mogu usmeriti
  cout << 0 << endl;
else {
  // u grafu nema mostova, pa usmeravamo grane drveta "naviše", a
  // povratne grane "naniže"
  for (int cvor = 0; cvor < n; cvor++)
    for (int sused : listaSuseda[cvor]) {
      // u neusmerenom grafu je svaka grana predstavljena dva puta
      // ovim se obezbeđuje da će se svaka grana razmatrati samo jednom
      if (cvor > sused) continue;
      // (u, v) je ta grana usmerena od pretka ka potomku
      int u = cvor, v = sused;
      if (dolazna[u] > dolazna[v])
        swap(u, v);

      // usmeravamo granu na pravi način
      if (roditelj[v] == u)
        // grana drveta
        cout << u+1 << " " << v+1 << endl;
      else
        // povratna grana
        cout << v+1 << " " << u+1 << endl;
    }
}
```

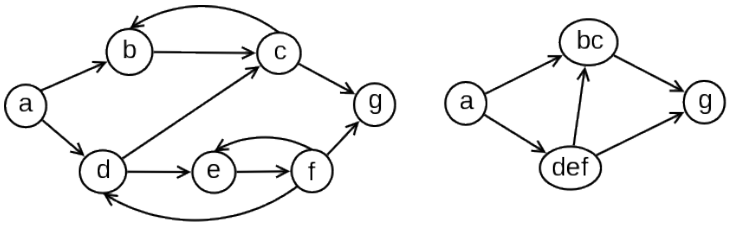
2.6 Komponente jake povezanosti grafa

Za usmereni graf kažemo da je *jako povezan* (engl. strongly connected) ako je svaki čvor grafa dostižan iz svakog drugog čvora u grafu.

Na skupu čvorova usmerenog grafa $G = (V, E)$ može se definisati relacija $\sim$ *obostrane dostižnosti*: $u \sim v$ ako je čvor u dostižan iz čvora v i čvor v je dostižan iz čvora u. Po definiciji za svaki čvor u važi $u \sim u$ (napomenimo da to ne znači da u grafu postoje petlje). Primetimo da su dva čvora obostrano dostižna ako i samo ako pripadaju nekom zajedničkom usmerenom ciklusu. Za relaciju obostrane dostižnosti važi da je:

- refleksivna – za svaki čvor $u \in V$ važi $u \sim u$,
- simetrična – za svaka dva čvora $u, v \in V$ važi $u \sim v$ ako i samo ako $v \sim u$,
- tranzitivna – za svaka tri čvora $u, v, w \in V$ iz $u \sim v$ i $v \sim w$ sledi i $u \sim w$.

Relacija obostrane dostižnosti je stoga relacija ekvivalencije. Ona razlaže skup čvorova V u klase ekvivalencije koje nazivamo *komponentama jake povezanosti* grafa G (engl. strongly connected components). Na slici 2.45 (levo) prikazan je usmereni graf G koji ima četiri komponente jake povezanosti, koje se sastoje redom od čvorova $\{a\}$, $\{b, c\}$, $\{d, e, f\}$ i $\{g\}$.



Slika 2.45: Graf G i odgovarajući kondezovani graf G^C čiji čvorovi odgovaraju komponentama jake povezanosti grafa G.

Komponente jake povezanosti u grafu imaju razne primene. Razmotrimo primer društvene mreže u kojoj korisnici mogu pratiti akcije drugih korisnika. Potrebno je identifikovati grupe korisnika koji su tesno povezani jedni sa drugima: svako od njih prati svakog drugog, direktno ili indirektno. Ovaj problem odgovara problemu određivanja komponenti jake povezanosti u grafu u kome su korisnici čvorovi, a usmerene grane odgovaraju relaciji praćenja korisnika. U softverskim sistemima moduli se mogu predstaviti čvorovima, a zavisnosti među njima usmerenim granama grafa. Identifikovanje komponenti jake povezanosti pomaže u pronalaženju ciklusa ili potencijalnih problema u strukturi međuzavisnosti modula, koji mogu biti od pomoći u unapređenju dizajna softvera.

Od grafa G može se formirati *kondenzovani* ili *komprimovani* graf G^C: to je usmereni aciklički graf koji sadrži informacije o komponentama jake povezanosti grafa G (slika 2.45, desno). Naime, svaki čvor u grafu G^C odgovara jednoj komponenti jake povezanosti grafa G, a dva čvora u grafu G^C su povezana granom ako i samo ako u grafu G postoji bar jedna grana od nekog čvora prve komponente do nekog čvora druge komponente jake povezanosti. Jasno je da je graf G^C aciklički: ako bi u njemu postojao ciklus, to bi značilo da se sve komponente jake povezanosti koje pripadaju ciklusu mogu spojiti u jednu, veću komponentu jake povezanosti. U nastavku teksta ćemo komponente jake povezanosti zvati kraće samo komponente.

Rešavamo sledeći problem.

Problem

Za dati usmereni graf $G = (V, E)$ odrediti sve komponente jake povezanosti.

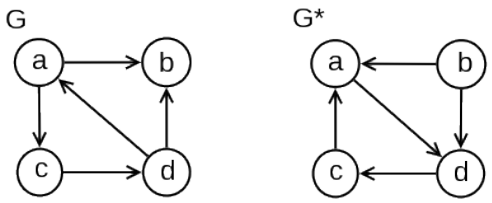
U nastavku teksta razmotrićemo nekoliko varijanti algoritma za rešavanje ovog problema zasnovanih na obilasku iz svakog čvora, a zatim i Tardžanov algoritam i Kosaradžuov algoritam.

2.6.1 Algoritam zasnovan na obilasku iz svih čvorova

Komponenta jake povezanosti kojoj pripada čvor $v \in V$ može da se odredi tako što se za svaki čvor u dostižan iz v pokrene DFS obilazak sa ciljem da se ustanovi da li je v dostižan iz u. Postupak se zatim može ponoviti za čvorove koji ne pripadaju prethodno izdvojenim komponentama (ako takvi čvorovi postoje). Postupak je neefikasan jer zahteva veliki broj pokretanja algoritma DFS.

Opisani postupak može se usavršiti. Neka je G^* graf koji sadrži iste čvorove kao graf G, ali suprotno usmerene grane: grana (v, u) pripada grafu G^* ako i samo ako grana (u, v) pripada grafu G (videti sliku 2.46). Ovaj graf zvaćemo *transponovanim grafom* (engl. transpose graph) grafa G. Komponente možemo da odredimo i na sledeći način: pokrenemo DFS obilazak iz čvora v_0 u oba grafa, G i G^*. Prvi DFS obilazak pronalazi skup čvorova A koji su dostižni iz čvora v_0, a drugi DFS obilazak skup čvorova B iz kojih je dostižan čvor v_0. Komponenta jake povezanosti kojoj pripada čvor v_0 jednaka je $A \cap B$. Ovaj postupak ponavljamo za proizvoljni čvor koji ne pripada do sada određenim komponentama povezanosti,

ukoliko takav čvor postoji. Najgori scenario je kada su svi čvorovi zasebne komponente i tada je složenost $O(|V|(|V|+|E|))$.



Slika 2.46: Graf G i njemu transponovani graf G^*.

Primer 2.6.1

Razmotrimo grafove G i G^* prikazane na slici 2.46: DFS obilazak grafa G pokrenut iz čvora d obilazi skup čvorova $A = \{d, a, b, c\}$, dok DFS obilazak grafa G^* pokrenut iz čvora d obilazi skup čvorova $B = \{d, c, a\}$. Važi $A \cap B = \{d, a, c\}$, te čvorovi d, a i c pripadaju istoj komponenti jake povezanosti. Primetimo da jedino čvor b ne pripada istoj komponenti povezanosti kao čvor d, a pošto DFS obilazak iz čvora b u grafu G obilazi samo čvor b (jer je njegov izlazni stepen 0), to čvor b čini sam za sebe drugu (preostalu) komponentu jake povezanosti grafa G.

Postoji nekoliko različitih algoritama linearne vremenske složenosti za određivanje komponenti jake povezanosti u usmerenom grafu, a najpoznatiji među njima su Tardžanov algoritam i Kosaradžuov algoritam. Oba algoritma su zasnovana na DFS obilasku grafa, samo se kod Tardžanovog algoritma sve radi u jednom prolazu kroz graf, dok se u Kosaradžuovom algoritmu dva puta poziva algoritam DFS.

2.6.2 Tardžanov algoritam

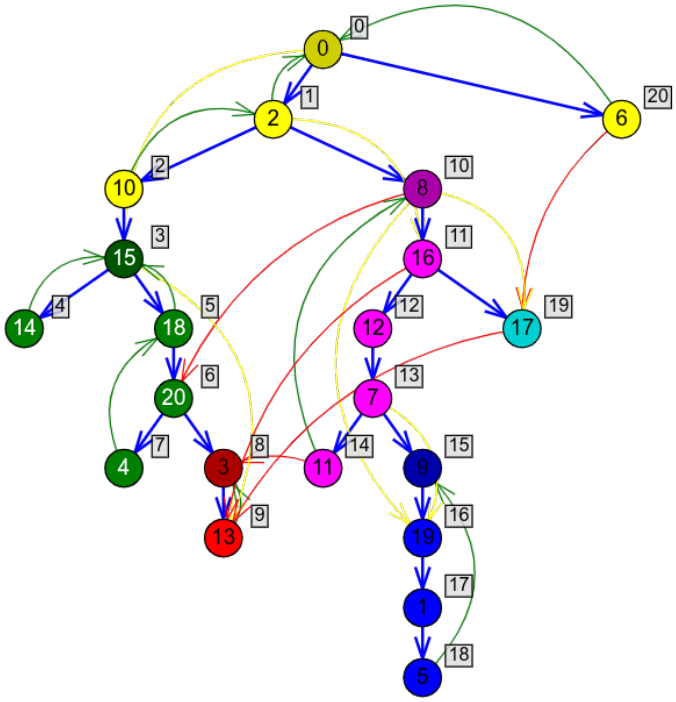
Kao što je to često slučaj kod grafovskih algoritama i izdvajanje komponenata jake povezanosti se zasniva na pažljivoj analizi DFS drveta i njegovih osobina. Prilikom DFS obilaska datog usmerenog grafa G implicitno se formira DFS drvo, odnosno DFS šuma. Jednostavnosti radi, možemo pretpostaviti da se DFS šuma sastoji od jednog drveta. Naime, svako drvo DFS šume se može analizirati zasebno, jer je jasno da su čvorovi svake komponente jake povezanosti podskup čvorova nekog pojedinačnog drveta u DFS šumi. Alternativno, kao što je objašnjeno u poglavlju 2.3.1.2, graf možemo proširiti novim čvorom v ulaznog stepena 0 koji je povezan granama sa svim ostalim čvorovima. Prošireni graf sem komponenti grafa G ima samo još jednu dodatnu jednočlanu komponentu jake povezanosti $\{v\}$. Naime, nijedan drugi čvor ne može biti sa njim u komponenti jake povezanosti, jer je ulazni stepen čvora v jednak 0.

U narednom razmatranju ćemo u potpunosti zanemariti direktne grane (one grane koje spajaju pretke sa potomcima). Naime, od svakog pretka do potomka se već može stići granama DFS drveta, pa se uklanjanjem direktnih grana ne menja dostižnost čvorova i komponente jake povezanosti ostaju nepromenjene.

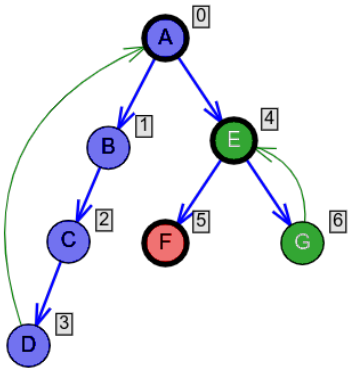
2.6.2.1 Raspored komponenti u DFS drvetu

Razmotrimo graf prikazan na slici 2.47. Komponente jake povezanosti su obeležene raznim bojama. Zapaža se da su čvorovi svake komponente jake povezanosti grupisani u DFS drvetu, tj. da je svaka komponenta deo nekog poddrveta DFS drveta. U dosta slučajeva čvorovi čak imaju i uzastopne redne brojeve, mada to ne mora biti uvek slučaj (na primer, poslednji čvor žute komponente se otkriva tek nakon što se otkriju i obrade sve ostale komponente). Ovaj odnos položaja komponenti unutar DFS drveta je ključan element Tardžanovog algoritma i u nastavku ćemo se potruditi da ga preciznije ispitamo i formalno dokažemo njegova svojstva.

Prilikom DFS obilaska važno nam je da znamo kada smo prešli iz jedne komponente u drugu. U svaku komponentu ulazimo tako što najpre posetimo njen čvor koji ima najmanji redni broj (dolazni DFS broj). Nazovimo *baznim čvorom* b (engl. base vertex) komponente X onaj čvor te komponente koji ima najmanji redni broj pri dolaznoj DFS numeraciji, tj. onaj čvor $b \in X$ za koji važi $b.Pre = \min_{v \in X} v.Pre$.



Slika 2.47: Komponente jake povezanosti obeležene bojama na DFS drvetu. Pored svakog čvora naveden je njegov redni broj u dolaznoj DFS numeraciji. Bazni čvor svake komponente prikazan je malo tamnijom bojom.



Slika 2.48: Graf koji sadrži tri komponente jake povezanosti grafa. Uz svaki čvor dat je njegov redni broj u dolaznoj DFS numeraciji. Bazni čvorovi svake komponente su podebljani.

Primer 2.6.2

Razmotrimo graf sa slike 2.48: on sadrži tri komponente jake povezanosti: $\{A, B, C, D\}$, $\{E, G\}$ *i* $\{F\}$. *Bazni čvor prve komponente je čvor* A, *druge* E, *a treće* F.

Bazni čvorovi grafa sa slike 2.47 su $0, 15, 3, 8, 9$ *i* 17.

Naredni primer će nam ukazati na važne odnose koji u svakoj komponenti važe između njenog baznog čvora i ostalih čvorova. Te odnose ćemo zatim formalizovati i dokazati u lemi 2.6.1.

Primer 2.6.3

U primerima na slikama 2.47 i 2.48 zapaža se da su svi čvorovi svake komponente potomci njenog baznog čvora u DFS drvetu. Dodatno, na putu od baznog čvora do bilo kog drugog čvora u toj komponenti kroz grane DFS drveta ne može da bude prekida, tj. put sadrži samo čvorove te komponente. Na primer, u grafu prikazanom na slici 2.48 jednoj komponenti povezanosti pripadaju čvorovi $\{A, B, C, D\}$ *i čvorovi* B, C *i* D *jesu potomci baznog čvora te komponente – čvora* A *u DFS drvetu. Slično važi i za čvor* G *koji je potomak baznog čvora* E *svoje komponente. Posebno, s obzirom da čvorovi* A *i* D *pripadaju istoj, prvoj komponenti jake povezanosti grafa, to važi i za čvorove* B *i* C *koji se nalaze na putu od čvora* A *do čvora* D *kroz grane DFS drveta. Naredna lema dokazuje da to nije slučajno. Istaknimo i da obratno ne mora da važi, tj. jasno je da mogu postojati potomci baznog čvora u DFS drvetu koji ne pripadaju njegovoj komponenti: na primer, čvorovi* E, F *i* G *grafa sa slike 2.48 su potomci čvora* A, *ali se ne nalaze sa njim u istoj komponenti.*

Naredna lema pokazuje da zapažanja iz primera 2.6.3 važe i u opštem slučaju.

Lema 2.6.1 **[Odnos baznih i ostalih čvorova komponente]**

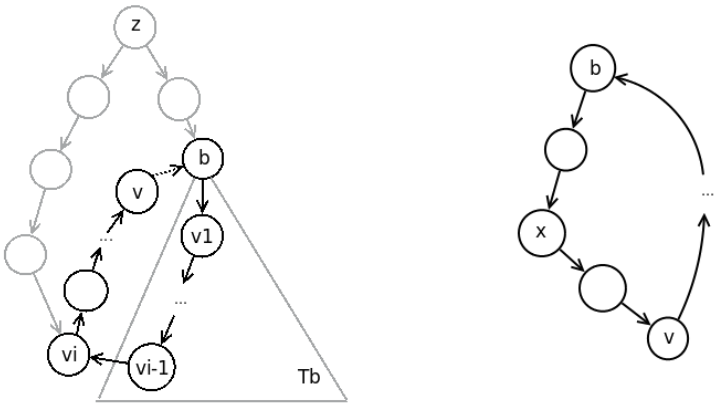
Neka je b *bazni čvor komponente* X. *Tada:*

a) svaki čvor $v \in X$ *je potomak čvora* b *u DFS drvetu,*
b) svi čvorovi na putu od b *do* $v \in X$ *preko grana DFS drveta pripadaju komponenti* X.

Dokaz. Pretpostavimo suprotno, odnosno da u komponenti X postoje čvorovi koji nisu u poddrvetu T_b DFS drveta sa korenom u b, i neka je v jedan od takvih čvorova. Neka je $(v_0 = b, v_1, \dots, v_k = v)$ put od b do v kroz čvorove komponente X. Neka je v_i prvi čvor na tom putu koji nije u poddrvetu T_b (dakle, čvor b je predak čvorova v_j za $j = 1, \dots, i-1$). Tada je grana (v_{i-1}, v_i) poprečna u odnosu na DFS drvo. Naime, pošto v_i nije u poddrvetu T_b, grana (v_{i-1}, v_i) nije grana DFS drveta, a ne može biti ni povratna, jer bi onda čvor v_i bio predak čvora b i b ne bi bio bazni čvor te komponente. Sve poprečne grane u odnosu na DFS drvo su usmerene ulevo, pa i grana (v_{i-1}, v_i). Dakle, čvor v_i je levo od čvora v_{i-1}, pa važi $v_i.Pre < v_{i-1}.Pre$. Dodatno, oni imaju zajedničkog pretka z u DFS drvetu (slika 2.49, levo).

Čvor z ne može biti jedan od čvorova $v_0 = b, v_1, v_2, \dots, v_{i-1}$, jer bi inače čvor v_i bio u poddrvetu sa korenom u čvoru b. Pošto je b predak čvora v_{i-1}, onda je z i zajednički predak čvorova v_i i b, te je čvor v_i levo i od čvora b, pa važi $v_i.Pre < b.Pre$. Pritom, čvor v_i pripada komponenti u kojoj je bazni čvor b, jer je obostrano dostižan sa čvorom b: b je dostižan iz čvora v_i putem od v_i do $v_k = v$ i dalje putem od v_k do b koji postoji jer je v, po pretpostavci, u ovoj komponenti. Ovo je u suprotnosti sa pretpostavkom da je b bazni čvor ove komponente. Dakle, svi čvorovi komponente X se nalaze u poddrvetu T_b DFS drveta.

Dokažimo drugi deo leme. Neka je x proizvoljni čvor na putu od b do v (slika 2.49, desno). Postoji put od čvora b do čvora x kroz grane DFS drveta, a takođe i put od čvora x do čvora b: taj put dobija se nadovezivanjem puta od x do v kroz grane DFS drveta i puta od v do b (ovaj put postoji jer čvorovi v i b pripadaju istoj komponenti). Stoga je x u istoj komponenti kao i čvor b. Zaključujemo da svi čvorovi na putu od čvora b do čvora v kroz grane DFS drveta pripadaju komponenti čiji je bazni čvor b. □



Slika 2.49: Ilustracije uz dokaz leme 2.6.1

Naredni primer sugeriše činjenicu da se čvorovi svake komponente dobijaju tako što se iz poddrveta čiji je koren bazni čvor te komponente uklone druge komponente. Ovo ćemo formalizovati i dokazati u lemi 2.6.2.

Primer 2.6.4

Na slikama 2.47 i 2.48 zapaža se da se svi čvorovi koji pripadaju bilo kojoj komponenti jake povezanosti dobijaju tako što se iz drveta čiji je koren bazni čvor te komponente uklone poddrveta čiji su korenovi bazni čvorovi drugih komponenti. Na primer, ako na slici 2.47 iz drveta čiji je koren čvor 0*, odsečemo poddrveta čiji su koreni čvorovi* 15 *i* 8 *ostaju samo žuti čvorovi koji čine jednu komponentu. Proverite da ovo važi i za sve ostale komponente.*

U grafu prikazanom na slici 2.48 čvor A je bazni čvor. Čvorovi E i F su takođe bazni čvorovi i pritom su potomci baznog čvora A. Primetimo da su čvorovi koji pripadaju komponenti čiji je bazni čvor A oni koji su potomci od A, a nisu potomci ni čvora E ni čvora F: to su čvorovi B, C i D. Slično, čvorovi koji pripadaju komponenti sa baznim čvorom E su oni koji su potomci od E, a nisu potomci od F, što je samo čvor G.

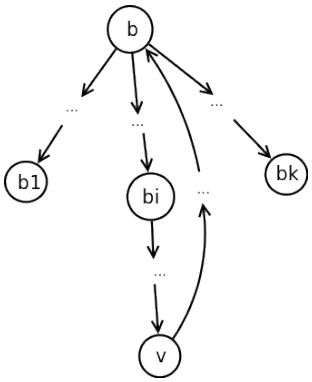
Naredna lema dokazuje da zapažanja iz primera 2.6.4 važe u opštem slučaju.

Lema 2.6.2 **[Odsecanje drugih komponenti]**

Neka je b bazni čvor neke komponente i neka su $b_1, b_2, \ldots, b_k$ bazni čvorovi nekih drugih komponenti koji su potomci čvora b u odnosu na DFS drvo. Tada se komponenta kojoj pripada čvor b sastoji od svih potomaka čvora b koji nisu potomci nekog od čvorova $b_1, b_2, \ldots, b_k$.

Dokaz. Neka je čvor v u istoj komponenti kao i čvor b. Na osnovu leme 2.6.1 on mora biti potomak u odnosu na DFS drvo čvora b. Pretpostavimo suprotno tvrđenju, da je on potomak i čvora čvora b_i za neko i, $1 \leq i \leq k$ (slika 2.50).

Pošto je b_i potomak čvora b, onda postoji put kroz grane DFS drveta od čvora b do čvora b_i. Slično, pošto je čvor v potomak čvora b_i postoji put od čvora b_i do čvora v, a zbog toga što su v i b u istoj komponenti, postoji i put od čvora v do čvora b. Nadovezivanjem ova dva puta dobija se put od čvora b_i do b. Na osnovu toga što je čvor b dostižan iz čvora b_i i čvor b_i dostižan iz čvora b sledi da su čvorovi b i b_i u istoj komponenti što je u suprotnosti sa pretpostavkom leme jer su b i b_i bazni čvorovi različitih komponenti. Dakle, čvorovi koji su u istoj komponenti kao i čvor b ne mogu biti potomci i nekog drugog baznog čvora b_i, koji je potomak čvora b. □



Slika 2.50: Ilustracija uz dokaz leme 2.6.2.

2.6.2.2 Izdvajanje komponenti uz pomoć steka

Pod pretpostavkom da nekako umemo da odredimo bazne čvorove svih komponenti jake povezanosti (što je problem kojim ćemo se pozabaviti u poglavlju 2.6.2.3), leme 2.6.1 i 2.6.2 nam daju mogućnost da odredimo koji su čvorovi zajedno sa nekim baznim čvorom u istoj komponenti jake povezanosti. I više od toga, ovakav raspored komponenata nam daje mogućnost da ih prilično jednostavno nabrojimo tokom DFS obilaska.

Potrebno je da u neku pogodnu strukturu podataka smeštamo čvorove u redosledu DFS obilaska, a da neposredno pre nego što napustimo neki čvor proverimo da li je on bazni i ako jeste ispišemo (i uklonimo) njega i sve njegove potomke koji se i dalje nalaze u toj strukturi podataka: to će biti čvorovi koji su nakon njega posećeni, i nakon njega dodati u tu strukturu. Dakle, elemente treba dodavati na kraj ove strukture podataka i uklanjati ih sa kraja ove strukture. Struktura podataka koju je pogodno koristiti u ove svrhe je stek. Naime, prilikom označavanja čvora v u toku DFS obilaska, čvor v se upisuje na zaseban namenski stek namenjen nabrajanju komponenata jake povezanosti. Kada se završi poziv algoritma DFS iz čvora v, u sklopu izlazne obrade, ako je v bazni čvor neke komponente, sa steka se uklanja čvor v i svi čvorovi iznad njega na steku (naravno, unazad: od elementa na vrhu steka pa sve do čvora v). Na taj način ako je čvor v bazni, izdvaja se komponenta koja sadrži čvor v i svi njeni čvorovi uklanjaju se sa steka. Redosled uklanjanja baznih čvorova nam garantuje da ćemo sa baznim čvorovima neke komponente ukloniti samo one potomke koji nisu potomci nekog drugog baznog čvora. Opisani postupak prikazan je u algoritmu 5.

Algoritam 5 Izdvajanje komponenti jake povezanosti uz pomoć steka

```
1:  procedure DFS(početni čvor u)
2:      označi čvor u
3:      stavi čvor u na stek
4:      for all čvor v koji je sused čvora u do
5:          if čvor v nije označen then
6:              DFS(v)
7:      if čvor u je bazni čvor komponente then
8:          skidaj elemente sa steka sve dok se ne skine u
9:          skinuti čvorovi su svi čvorovi komponente čiji je bazni čvor u
10:
11: for all neoznačen čvor u do
12:     DFS(u)
```

Dokažimo korektnost ovog algoritma.

Teorema 2.6.1 **[Korektnost nabrajanja komponenti pomoću steka]**

Za svaki bazni čvor b komponente jake povezanosti grafa G važi sledeće:

- *Pri ulaznoj obradi čvora b, on se postavlja na stek.*
- *Pre njegove izlazne obrade ispravno su obrađene sve komponente jake povezanosti u poddrvetu čiji je on koren. Na vrhu steka, iznad čvora b, nalaze se svi čvorovi njegove komponente povezanosti.*
- *Nakon izlazne obrade čvora b on se uklanja sa steka, zajedno sa svim čvorovima njegove komponente, nakon čega je sadržaj steka isti kao pri ulazu u čvor b.*

Dokaz. Dokaz se može izvesti indukcijom po broju m komponenti jake povezanosti koje čine graf.

Za $m = 1$, postoji samo jedan bazni čvor b i on je koren celog DFS drveta. U prvom koraku DFS obrade on se postavlja na prazan stek, zatim se svi ostali čvorovi, jedan po jedan stavljaju na stek, sve dok se na samom kraju DFS obilaska ne dođe do izlazne obrade čvora b. U tom trenutku (pošto je on jedini bazni čvor), svi čvorovi se skidaju sa steka, obrađuje se ispravno jedina komponenta grafa i stek na kraju ostaje prazan, kao što je i bio pre ulaska u čvor b.

Neka je tvrđenje tačno za grafove sa manje od m komponenti i neka je G graf sa m komponenti. Neka je b prvi bazni čvor na koji se nailazi pri DFS obilasku (to je čvor iz kog pokrećemo DFS obilazak), a neka su $b_1, b_2, \dots, b_{m-1}$ njegovi potomci, koji su bazni čvorovi ostalih komponenti. Svaki od njih će tokom DFS obilaska iz čvora b u nekom trenutku biti stavljen na stek. Prema induktivnoj hipotezi, posle završetka DFS obilaska iz čvora b_i, $1 \leq i \leq m-1$, izdvojene su sve komponente dostižne iz čvora b_i i svi njihovi čvorovi su uklonjeni su sa steka, ostavljajući stek svaki put u stanju kao pre ulaza u čvor b_i. To znači da se u trenutku izlazne obrade čvora b na steku nalaze svi čvorovi drveta sa korenom b (potomci čvora b), koji ne pripadaju ni jednom drvetu čiji je koren neki od čvorova $b_1, \dots, b_{m-1}$. Međutim, na osnovu leme 2.6.2 znamo da su to tačno čvorovi komponente čiji je bazni čvor b. Prilikom izlazne obrade čvora b ta komponenta se ispravno izdvaja, njeni čvorovi se skidaju sa steka i stek ostaje prazan. □

2.6.2.3 Određivanje baznih čvorova komponenti

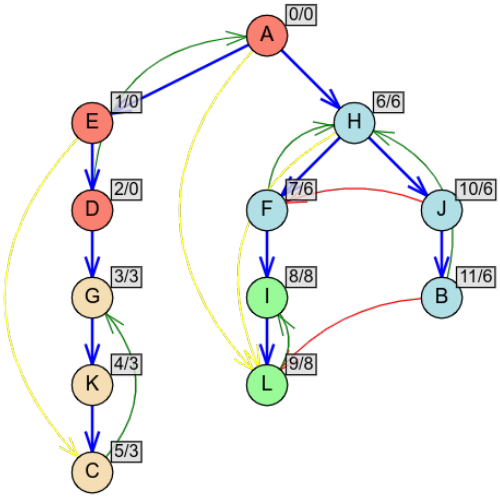
Da bi Tardžanov algoritam bio kompletan, nedostaje još samo da formulišemo neki efektivni test kojim bismo mogli da za dati čvor ispitamo da li je on bazni čvor svoje komponente.

U svakoj komponenti jake povezanosti moguće je stići od bilo kog čvora do bilo kog drugog čvora. Zato se od svakog čvora komponente može stići do njenog baznog čvora, što je "najniži" tj. čvor sa najmanjim rednim brojem u toj komponenti. Poželjno je zato da za svaki čvor ispitamo koji je "najniži" čvor do kog se može stići u drvetu, u nadi da će to uvek biti bazni čvor komponente.

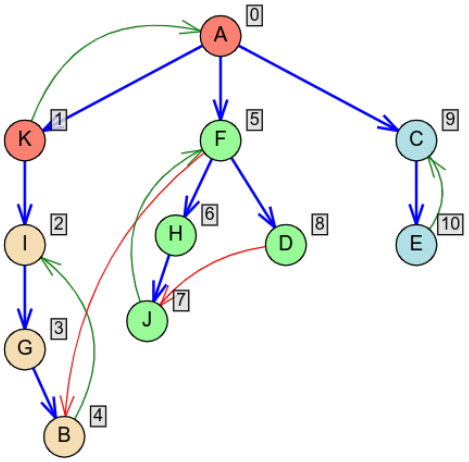
Razmotrimo graf na slici 2.51. Vidimo da je na ovom grafu nastupila idealna situacija, jer je "najniži" čvor do kog je moguće vratiti se iz proizvoljnog čvora grafa uvek bazni čvor komponente kojoj taj čvor pripada. Bazni čvorovi se onda lako prepoznaju tako što su to oni čvorovi kojima se redni broj poklapa sa najmanjim brojem čvora do kog se iz tog čvora možemo vratiti.

Ipak, problem nije tako jednostavan, jer ponekad poprečne grane mogu da nas odvedu do čvora koji ima manji redni broj od rednog broja baznog čvora komponente. Razmotrimo sada graf na slici 2.52. Zbog poprečne grane (F, B) se iz čvora F (redni broj 5) možemo vratiti do čvora B (redni broj 4), pa zatim do čvora I (redni broj 2). Dakle, najmanji redni broj čvora do koga možemo da se vratimo iz čvora F (redni broj 5) je 2, pa, pošto taj broj nije jednak rednom broju čvora F, čvor F ne prepoznajemo kao bazni čvor komponente, a on to jeste. Dakle, razmatranje poprečne grane (F, B) pokazuje da jednostavan kriterijum koji smo formulisali nije sasvim korektan. S druge strane, ako se poprečna grana (D, J) ne bi gledala, ne bi bilo moguće ustanoviti da se iz čvora D (redni broj 8) možemo vratiti do čvora J (redni broj 7), pa zatim unazad do čvora F (redni broj 5), i ne bi bilo jasno da čvor D pripada komponenti čiji je bazni čvor F.

Jasno je da moramo modifikovati kriterijum tako da uključi razmatranje nekih, a isključi razmatranje nekih drugih poprečnih grana. Vidimo da ne treba određivati najmanji redni broj *proizvoljnog* čvora do kog se možemo "spustiti" iz datog čvora v, jer taj čvor može da bude takav da ne pripada istoj komponenti kojoj pripada i čvor v. Nas zapravo zanima da za svaki čvor v odredimo najmanji broj čvora *u njegovoj komponenti*



Slika 2.51: Uz svaki čvor grafa označen je njegov redni DFS broj i najmanji redni broj čvora do kog je moguće vratiti se iz tog čvora. Situacija izgleda idealno – iz svakog čvora je moguće vratiti se tačno do baznog čvora njegove komponente.



Slika 2.52: Grana (D, J) mora biti razmatrana u okviru puteva ka čvorovima sa manjim rednim brojem jer vodi ka čvoru iste komponente (inače se ne bi moglo videti da se od D možemo stići do F), a grana (F, B) ne sme biti razmatrana, jer vodi ka čvoru različite komponente (inače bi delovalo da se od F može stići do nižeg čvora).

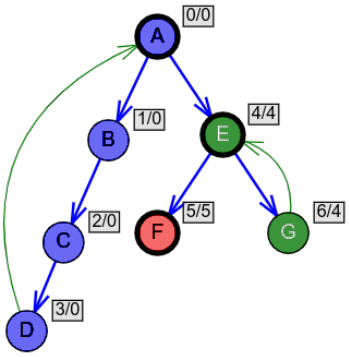
do kog je moguće vratiti se. Lako se može dokazati da je čvor bazni čvor komponente ako i samo ako se taj broj poklapa sa njegovim rednim brojem. Direktne grane, već smo konstatovali, nemaju nikakvog uticaja na određivanje ovog broja. Dopušteno je "podizati se" granama drveta, a "spuštati se" povratnim granama (jer povratne grane uvek spajaju dva čvora koja su u istoj komponenti, pošto kreiraju ciklus sa granama drveta između ta dva čvora) i "spuštati se" poprečnim granama (ka čvorovima sa manjim rednim brojevima), ali ne svim poprečnim granama, već samo onim koje spajaju čvorove koji su u istoj komponenti (poprečna grana može povezivati dva čvora iz iste ili iz različite komponente, slika 2.52).

Iako deluje da smo problem određivanja svih čvorova neke komponente sveli na problem određivanja njenog baznog čvora komponente, a problem određivanja baznog čvora komponente na poznavanje čvorova komponente, videćemo uskoro da nismo napravili cirkularnu definiciju i da smo se približili rešenju problema.

Opisani brojevi (najmanji redni broj čvora unutar komponente do kog se možemo popeti) daju jasan kriterijum za određivanje baznih čvorova, ali se ne mogu jednostavno odrediti korišćenjem jednog DFS obilaska (što je ilustrovano primerom 2.6.5). Umesto njih, koriste se redni brojevi čvorova unutar komponente, ali do kojih se može stići samo kretanjem niz grane drveta, a zatim povratkom uz najviše jednu poprečnu ili povratnu granu. Za svaki čvor određujemo *"najniži" čvor njegove komponente dostižan jednom povratnom ili poprečnom granom*. Redni broj tog čvora ponovo obeležavamo sa $L(v)$ (engl. lowlink), kao u poglavlju 2.5, ali jasno je da $L(v)$ mora biti definisan nešto drugačije nego u slučaju neusmerenih grafova. Neka je X komponenta koja sadrži čvor v. Označimo sa M najmanji redni broj čvora komponente X do koga se iz čvora v može stići putem koji se sastoji od grana DFS drveta i koji se završava najviše jednom povratnom ili poprečnom granom. Vrednost $L(v)$ definišemo kao $\min\{v.Pre, M\}$, odnosno:

$$L(v) = \min\{v.Pre, \min_{\substack{w \in X \\ \text{postoji grana } (t,w),\ t \in T_v \\ (t,w) \text{ je poprecna ili povratna}}} w.Pre\}.$$

Na slici 2.53 prikazan je graf sa slike 2.48, pri čemu je uz svaki čvor v, pored rednog broja u dolaznoj numeraciji, prikazana i vrednost $L(v)$ čvora. Bazni čvorovi su tačno oni čvorovi za koje važi $v.Pre = L(v)$.



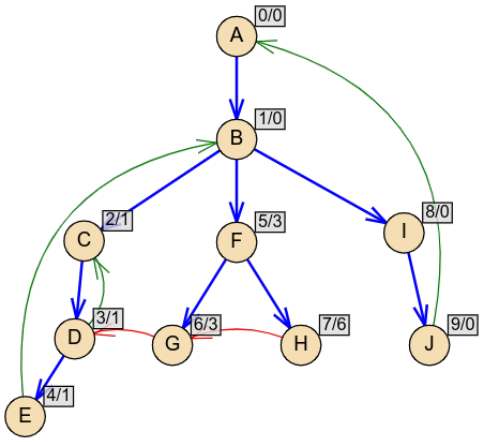
Slika 2.53: Graf kod koga su uz svaki čvor v prikazane vrednosti $v.Pre$ i $L(v)$. Čvorovi za koje važi $L(v) = v.Pre$ su bazni čvorovi komponenti povezanosti.

Primer 2.6.5

Graf na slici 2.54 ima jednu komponentu i na njemu se može videti kako se "najniži" čvor u komponenti do kog se može doći može odrediti praćenjem lanca putanja određenog vrednostima funkcije L.

Na primer,

- $L(F) = 3$, *jer se od čvora F granom drveta stiže do G, pa zatim poprečnom granom do čvora D koji ima redni broj 3. Dakle, iz drveta sa korenom F se jednom poprečnom granom vraćamo u drvo sa korenom D.*



Slika 2.54: Graf sa jednom komponentom jake povezanosti. Uz svaki čvor v je napisan njegov redni broj $v.Pre$ i vrednost $L(v)$.

- $L(D) = 1$, *jer se od čvora* D *granom drveta stiže do čvora* E*, pa zatim povratnom granom do čvora* B *koji ima redni broj 1. Dakle iz drveta sa korenom* D *se jednom povratnom granom vraćamo u drvo sa korenom* B*.*
- $L(B) = 0$*, jer se od čvora* B *granama drveta redom stiže do čvorova* I*, pa zatim* J*, a onda se povratnom granom od* J *vraćamo u čvor* A *koji ima redni broj* 0*. Dakle, iz drveta sa korenom* B *se jednom povratnom granom možemo vratiti u bazni čvor* A *sa rednim brojem* 0*.*

Da bi se za čvor F *ustanovilo da se može vratiti u čvor* A *sa rednim brojem* 0 *potrebno je, između ostalog, da vidimo granu* (J, A)*, što se dešava tek pred kraj DFS obilaska, nakon što je obilazak čvora* F *završen. Ovo ukazuje na to da je za određivanje najnižeg čvora u komponenti do kog se može stići (što je uvek bazni čvor) potrebno više puta obilaziti graf. Videćemo da to nije slučaj sa vrednostima* $L(v)$ *i da se one mogu lako odrediti tokom jedinog obilaska grafa koji vrši Tardžanov algoritam.*

Analogno lemi 2.5.2, moguće je formulisati lemu koja opisuje rekurzivne veze između vrednosti funkcije L između čvorova DFS drveta (na osnovu koje sledi postupak za određivanje vrednosti funkcije L tokom DFS obilaska).

Bilo koji čvor koji nije bazni ima manju vrednost $L(v)$ od svog rednog broja $v.Pre$. Zaista, postoji put od njega do baznog čvora, koji u nekom trenutku mora da "pobegne" iz poddrveta sa korenom v, bilo povratnom granom bilo poprečnom granom ka čvoru u istoj komponenti i da nas dovede do čvora sa manjim rednim brojem. Prateći lanac ovakvih putanja, doći ćemo u jednom trenutku i do baznog čvora za koji je $L(v) = v.Pre$. Dokažimo ovo i formalno.

Lema 2.6.3

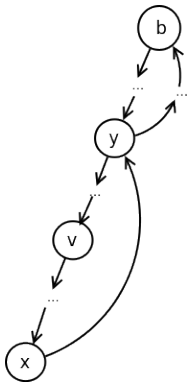
Čvor v *je bazni čvor ako i samo ako važi* $L(v) = v.Pre$*.*

Dokaz. Pokažimo prvi smer tvrđenja: ako je čvor v bazni onda važi $L(v) = v.Pre$. Pretpostavimo suprotno, odnosno da za bazni čvor v važi $L(v) < v.Pre$ i pokažimo da onda čvor v nije bazni čvor. Prema definiciji vrednosti L, postoji čvor w u istoj komponenti kao i v takav da je $L(v) = w.Pre$. Stoga je $w.Pre < v.Pre$ te čvor v nije bazni čvor.

Dokažimo sada suprotni smer implikacije: ako za čvor v važi uslov $L(v) = v.Pre$, onda je čvor v bazni. Pretpostavimo suprotno: da važi uslov $L(v) = v.Pre$, a da čvor v nije bazni čvor. Neka je b bazni čvor komponente koja sadrži čvor v. Prema lemi 2.6.1 čvor b je predak čvora v. Pošto su b i v u istoj komponenti,

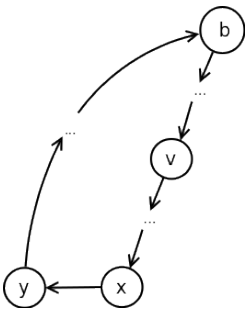
postoji prosti put p od v do b. Neka je y prvi čvor na putu p koji nije u poddrvetu sa korenom u v i neka je x čvor koji mu prethodi na putu p:

- ako je grana (x, y) povratna (slika 2.55), onda je y predak čvora v i $L(v) \leq y.Pre < v.Pre$, suprotno pretpostavci (čvor y nije na putu od v do x, jer je put p po pretpostavci prost);



Slika 2.55: Slučaj kada je grana (x, y) povratna u dokazu leme 2.6.3.

- ako je grana (x, y) poprečna (slika 2.56), onda je ona usmerena ulevo i $y.Pre < x.Pre$. S obzirom na to da su poddrvo sa korenom u y i poddrvo sa korenom u v disjunktni i da postoji grana od potomka čvora v (odnosno x) do y, na osnovu svojstava poprečnih grana možemo zaključiti da važi $y.Pre < v.Pre$. Odavde sledi da je $L(v) \leq y.Pre$, jer je y u istoj komponenti jake povezanosti kao i v i dostižan je preko niza grana DFS drveta nakon koga sledi jedna povratna ili poprečna grana. Na osnovu ovoga i činjenice da je $y.Pre < v.Pre$, dobijamo da važi $L(v) < v.Pre$, suprotno pretpostavci.



Slika 2.56: Slučaj kada je grana (x, y) poprečna u dokazu leme 2.6.3.

□

Finalizujmo opis algoritma tako što ćemo prikazati kako se određuju vrednosti $L(v)$. Prilikom ulaska u čvor u tokom DFS obilaska dodeljujemo mu redni broj $v.Pre$ i inicijalizujemo $L(v) = v.Pre$. Zatim obrađujemo sve grane (v, w) koje vode iz čvora v do nekog čvora w.

- Ako čvor w nije posećen, rekurzivno vršimo njegov obilazak i grana (v, w) postaje grana DFS drveta. Nakon rekurzivnog poziva poznata je vrednost $L(w)$, pa vrednost $L(v)$ ažuriramo na vrednost $\min\{L(v), L(w)\}$.
- Ako je čvor w posećen, a grana (v, w) je povratna ili poprečna, a čvor v je u istoj komponenti, analiziramo i ažuriramo vrednost $L(v)$ na vrednost $\min\{L(v), w.Pre\}$ – ovo je u skladu sa definicijom vrednosti L, i ne koristi se izraz $\min\{L(v), L(w)\}$ jer je dopušten prelaz preko proizvoljno mnogo grana drveta, ali samo preko jedne povratne ili poprečne grane.
- Ako je čvor w posećen, a grana (v, w) je direktna ili je poprečna, a čvor w je u nekoj drugoj komponenti, ovu granu prosto preskačemo.

Ostaje pitanje kako efikasno utvrditi da li poprečna grana (v, w) grana vodi ka čvoru iste ili druge komponente povezanosti. Jedno rešenje je da se primeti da se ono može svesti na pitanje da li se u trenutku obrade ove grane njen krajnji čvor w nalazi u namenskom steku namenjenom nabrajanju komponenti ili ne. Dajmo prvo malo precizniju karakterizaciju elemenata koji se nalaze na steku.

Lema 2.6.4 **[Čvorovi koji ostaju na steku nakon izlazne obrade]**

Čvor se nalazi na steku nakon njegove izlazne obrade ako i samo ako postoji put u grafu od njega do nekog čvora koji se trenutno nalazi ispod njega na steku

Dokaz. Ako bi se na steku nalazio čvor tokom čije je izlazne obrade utvrđeno da se u grafu ne može od tog čvora doći do nekog čvora ispod njega na steku, taj čvor bi bio bazni čvor komponente i morao bi prilikom svoje izlazne obrade biti uklonjen sa steka zajedno sa ostalim čvorovima njegove komponente. □

Primer 2.6.6

U grafu na slici 2.52 se u trenutku ulazne obrade čvora F na steku nalaze čvorovi A i K pri čemu je završena izlazna obrada samo čvora K. On se nalazi na steku jer postoji put od njega do čvora A koji je ispod njega na steku.

Sada možemo dokazati i karakterizaciju poprečnih grana.

Lema 2.6.5 **[Poprečne grane i stek]**

Poprečna grana (v, w) vodi iz čvora v ka čvoru w koji je u trenutku obrade čvora v na steku ako i samo ako su v i w u istoj komponenti povezanosti.

Dokaz. Ako je grana (v, w) poprečna, čvor w je levo od v. On će kao takav biti posećen pre čvora v i pritom dodat na stek, a u trenutku posete čvora v njegova izlazna obrada će već biti završena.

Ako čvor w nije više na steku prilikom obrade čvora v, on je morao biti uklonjen sa steka zajedno sa svim čvorovima svoje komponente, što znači da nije u istoj komponenti sa v.

Ako je čvor w i dalje na steku prilikom obrade čvora v, na osnovu leme 2.6.4, pratećи lanac putanja ka sve nižim i nižim čvorovima na steku, iz njega ćemo se "spustiti" do nekog zajedničkog pretka čvorova v i w, čime se obezbeđuje da su čvorovi v i w uzajamno dostižni i nalaze se u istoj komponenti. □

Primer 2.6.7

U grafu na slici 2.52 poprečna grana FB vodi ka čvoru B koji se ne nalazi više na steku u trenutku obrade čvora F, pa F i B nisu u istoj komponenti. Poprečna grana DJ vodi ka čvoru J koji je i dalje na steku u trenutku obrade čvora D, pa su čvorovi D i J u istoj komponenti.

Dakle, važi sledeće:

- poprečna grana (v, w) vodi ka čvoru w koji je u trenutku obrade čvora v na steku ako i samo su v i w u istoj komponenti povezanosti;
- povratna grana (v, w), jasno, uvek vodi ka čvoru w koji je u trenutku obrade čvora v na steku (jer je w predak od v, pa je završena njegova ulazna obrada tokom koje je je stavljen na stek, ali nije završena njegova izlazna obrada niti izlazna obrada baznog čvora njegove komponente, pa w nije još mogao biti skinut sa steka);
- ako direktna grana (v, w) vodi ka čvoru w na steku, važi da je $w.Pre > v.Pre$, pa ta grana ne može uticati na vrednost $L(v)$.

Zato u kodu možemo proveravati one grane (v, w) koje vode od čvora v do čvora w koji se nalazi na steku, ne analizirajući posebno njihovu vrstu.

Ispitivanje da li je čvor na steku prolaskom kroz stek nije efikasno, pa dodatno koristimo namenski niz u kome ćemo tokom izvršavanja algoritma pamtiti za svaki čvor da li se trenutno nalazi na steku ili ne. Alternativno rešenje je da se prilikom skidanja čvora v sa steka njegove vrednosti $v.Pre$ i $L(v)$ postave na $+\infty$. Kada se to uradi sve poprečne grane mogu biti posećene, ali do smanjenja vrednosti $L(v)$ može doći samo kod grana ka čvorovima koji su još na steku i nalaze se u istoj komponenti.

Sada možemo dati i zaokruženi pseudokod Tardžanovog algoritma.

Algoritam 6 Tardžanov algoritam za određivanje komponenata jake povezanosti

```
1:  procedure DFS(čvor v)
2:      dodeli redni broj v.Pre
3:      L(v) = v.Pre
4:      stavi v na stek
5:      for all sused w čvora v do
6:          if čvor w nije označen then
7:              DFS(w)
8:              L(v) = min{L(v), L(w)}
9:          else if w je na steku then
10:             L(v) = min{L(v), w.Pre}
11:     if L(v) = v.Pre then
12:         skidaj elemente sa steka sve dok se ne skine v
13:         skinuti čvorovi su svi čvorovi komponente sa korenom v
14: for all neoznačen čvor v do
15:     DFS(v)
```

Tardžanov algoritam za određivanje komponenti jake povezanosti oslanja se na DFS obilazak grafa, te je složenosti $O(|V| + |E|)$.

Algoritam se može implementirati u jeziku C++ na sledeći način.

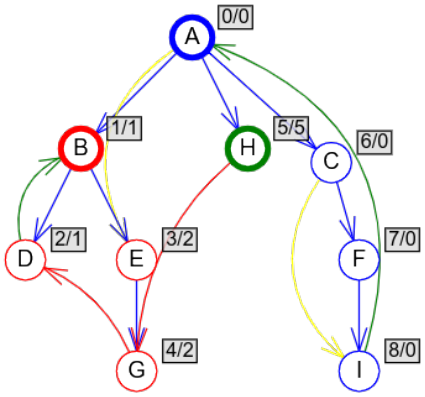
```
int vreme_dolazna = 1;

void dfs(int cvor, vector<int> &dolazna, vector<int> &lowlink,
         stack<int> &redosledUObilasku, vector<int> &naSteku) {
  dolazna[cvor] = lowlink[cvor] = vreme_dolazna;
  vreme_dolazna++;
  redosledUObilasku.push(cvor);
  naSteku[cvor] = true;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (int sused : listaSuseda[cvor]) {
    // ako cvor 'sused' do sada nismo posetili
    if (dolazna[sused] == -1) {
       // pokrecemo DFS obilazak iz cvora 'sused'
       dfs(sused, dolazna, lowlink, redosledUObilasku, naSteku);
       // ako je potrebno azuriramo vrednost L cvora 'cvor'
       if (lowlink[sused] < lowlink[cvor])
          lowlink[cvor] = lowlink[sused];
    }
    // ako je u pitanju povratna ili poprecna grana,
```

```
    // azuriramo vrednost L za cvor 'cvor'
    // samo ako se sused nalazi u steku
    // to znaci da sused pripada istoj komponenti povezanosti
    else if (naSteku[sused])
      if (dolazna[sused] < lowlink[cvor])
         lowlink[cvor] = dolazna[sused];
  }

  // ako je u pitanju bazni cvor komponente
  // stampamo sve cvorove te komponente
  if (dolazna[cvor] == lowlink[cvor]) {
    while(1) {
      // ispisujemo element sa vrha steka i uklanjamo ga
      int cvor_komponente = redosledUObilasku.top();
      cout << cvor_komponente << " ";
      naSteku[cvor_komponente] = false;
      redosledUObilasku.pop();
      // ako smo stigli do baznog cvora prekidamo petlju
      if (cvor_komponente == cvor) {
         cout << "\n";
         break;
      }
    }
  }
}

void ispisi_komponente(int cvor) {
  int brojCvorova = listaSuseda.size();
  vector<int> dolazna(brojCvorova,-1);
  vector<int> lowlink(brojCvorova);
  // stek na koji smestamo cvorove u redosledu DFS obilaska
  stack<int> redosledUObilasku;
  // vektor koji omogucava brzu proveru da li se cvor nalazi na steku
  vector<bool> naSteku(brojCvorova,false);

  cout << "Komponente jake povezanosti su: " << endl;
  dfs(cvor, dolazna, lowlink, redosledUObilasku, naSteku);
}
```

Primer 2.6.8

Razmotrimo izvršavanje ovog algoritma na primeru grafa prikazanog na slici 2.57.

- *dfs(*A*)*
  stek: A
  inicijalizujemo $A.Pre = L(A) = 0$
  - *grana* (A, B) *postaje grana drveta*
    dfs(B*)*
    stek: A, B
    inicijalizujemo $B.Pre = L(B) = 1$
    - *grana* (B, D) *postaje grana drveta*



Slika 2.57: Primer grafa koji ima tri komponente jake povezanosti: $\{B, D, E, G\}$, $\{H\}$ i $\{A, C, F, I\}$. Uz svaki čvor prikazan je redni broj u dolaznoj numeraciji i vrednost funkcije L.

dfs(D)
stek: A, B, D
inicijalizujemo $D.Pre = L(D) = 2$

- *grana* (D, B) *je povratna grana*
  smanjujemo $L(D) = 1$

izlaz iz D*:* $L(D) = 1$*, a* $D.Pre = 2$*, pa on nije bazni čvor komponente*
$L(B)$ *nije veće od* $L(D)$*, pa ostaje* 1

– *grana* (B, E) *postaje grana drveta*
dfs(E)
stek: A, B, D, E
inicijalizujemo $E.Pre = L(E) = 3$

- *grana* (E, G) *postaje grana drveta*
  dfs(G)
  stek: A, B, D, E, G
  $E.Pre = L(E) = 4$

  - *grana* (G, D) *je poprečna grana*
    D *je na steku, pa smanjujemo* $L(G) = 2$

  izlaz iz G*:* $L(G) = 2$*, a* $G.Pre = 4$*, pa on nije bazni čvor komponente*
  $L(E)$ *smanjujemo na vrednost* $L(G) = 2$

izlaz iz E*:* $L(E) = 2$*, a* $E.Pre = 3$*, pa on nije bazni čvor komponente*
$L(B)$ *nije veće od* $L(E)$*, pa ostaje* 1

izlaz iz B*:* $L(B) = B.Pre = 1$*, on jeste bazni čvor komponente*
sa steka se skidaju G*,* E*,* D*,* B *koji čine komponentu*
stek: A
$L(A)$ *nije veće od* $L(B)$*, pa ostaje* 0

– *direktna grana* (A, E) *se preskače jer* E *nije na steku*

– *grana* (A, H) *postaje grana drveta*
dfs(H)
stek: A, H
inicijalizujemo $H.Pre = L(H) = 5$

– *poprečna grana* (H, G) *se preskače jer G nije na steku*

izlaz iz H*:* $L(H) = H.Pre = 5$*, pa je on bazni čvor komponente*
sa steka se skida: H *koji čini komponentu*
stek: A
$L(A)$ *nije veće od* $L(H)$*, pa ostaje* 0

– *grana* (A, C) *postaje grana drveta*
dfs(C*)*
stek: A*,* C
inicijalizujemo $C.Pre = L(C) = 6$

– *grana* (C, F) *postaje grana drveta*
dfs(F*)*
stek: A*,* C*,* F
inicijalizujemo $F.Pre = L(F) = 7$

- *grana* (F, I) *postaje grana drveta*
dfs(I*)*
stek: A*,* C*,* F*,* I
inicijalizujemo $I.Pre = L(I) = 8$

- *povratna grana* (I, A)
smanjujemo $L(I) = 0$

izlaz iz I*:* $L(I) = 0$*, a* $I.Pre = 8$*, pa on nije bazni čvor komponente*
$L(F)$ *smanjujemo na vrednost* $L(I) = 0$

izlaz iz F*:* $L(F) = 0$*, a* $F.Pre = 7$*, pa on nije bazni čvor komponente*
$L(C)$ *smanjujemo na vrednost* $L(F) = 0$

– *direktna grana* (C, I)
$L(C)$ *nije veće od* $L(I)$*, pa ostaje* 0

izlaz iz C*:* $L(C) = 0$*, a* $C.Pre = 6$*, pa on nije bazni čvor komponente*
$L(A)$ *nije veće od* $L(C)$*, pa ostaje* 0

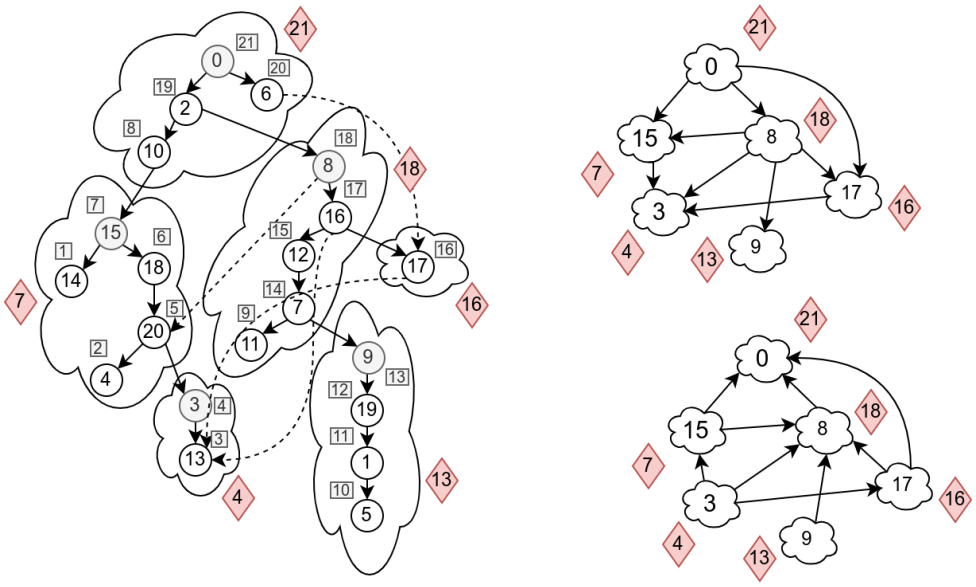
izlaz iz A*:* $L(A) = A.Pre = 0$*, pa je on bazni čvor komponente*
sa steka se skidaju: I*,* F*,* C*,* A*, koji čine komponentu*
stek:

2.6.3 Kosaradžuov algoritam

Drugi algoritam linearne vremenske složenosti za određivanje komponenti jake povezanosti u usmerenom grafu je Kosaradžuov algoritam. Ovaj algoritam je osmislio (međutim nije publikovao) Kosaradžu (Rao Kosaraju) 1978. godine, a zatim je do njega kasnije nezavisno došao i Šarir (Micha Sharir) 1981. godine. On koristi dva DFS obilaska, pa je malo sporiji od Tardžanovog algoritma, ali je jednostavniji za razumevanje.

Možemo primetiti da se prilikom izvršavanja Tardžanovog algoritma bazni čvorovi skidaju sa steka (zajedno sa ostalim čvorovima iz njihovih komponenata) u rastućem redosledu njihove odlazne numeracije. Pri tome, bazni čvor ima maksimalni redni broj odlazne DFS numeracije od svih čvorova svoje komponente. Obilazak baznih čvorova teče u obratnom topološkom redosledu čvorova kondenzovanog grafa (grafa dobijenog tako što se svaka komponenta predstavi samo njenim baznim čvorom), tj. obilazak tog acikličkog grafa se vrši od njegovih “listova” ka “korenu”.

Primer 2.6.9



Slika 2.58: Na slici levo je uz svaki čvor prikazan odlazni redni broj DFS obilasku grafa sa slike 2.47 (preglednosti radi, grane unutar iste komponente nisu prikazane), dok je uz svaku komponentu prikazan najveći redni broj odlazne DFS numeracije čvora te komponente (to je redni broj baznog čvora). Desno je prikazan kondenzovani graf originalnog grafa i kondenzovani graf njegovog transponovanog grafa.

Na slici 2.58 prikazani su odlazni redni brojevi čvorova u DFS obilasku grafa sa slike 2.47. Jasno se vidi da bazni čvorovi imaju najveći odlazni broj od svih čvorova u njihovoj komponenti. Taržanov algoritam bi redom pronalazio komponente čiji su bazni čvorovi 3, 15, 9, 17, 8, *i* 0 *(a njhovi odlazni redni brojevi su redom* 4, 7, 13, 16, 18, 21*).*

Možemo primetiti i da sve grane u kondenzovanom grafu vode od čvorova sa većim ka čvorovima sa manjim odlaznim brojevima. Dokažimo ovo i formalno. Označimo sa $v.Post$ odlazni redni broj čvora v u nekom DFS obilasku koji po jednom posećuje sve čvorove grafa, a sa $C.Post$ maksimalni odlazni redni broj svih čvorova komponente C, tj. $C.Post = \max_{v \in C} v.Post$.

Teorema 2.6.2

Neka su C i C' dve različite komponente jake povezanosti grafa G i neka u kondenzovanom grafu grafa G postoji grana (C, C'). Tada važi $C.Post > C'.Post$.

Dokaz. Razlikujemo dva slučaja u odnosu na to koju od komponenti C i C' je algoritam DFS prvu posetio.

- Pretpostavimo da DFS obilazak najpre posećuje neki čvor v komponente C, i da u tom trenutku nijedan drugi čvor iz C i C' nije posećen. Svi čvorovi komponente C dostižni su iz čvora v. Dodatno, s obzirom na to da u kondenzovanom grafu grafa G postoji grana (C, C'), iz čvora v su dostižni i svi čvorovi komponente C'. To znači da će DFS obilazak pokrenut iz čvora v posetiti sve čvorove u iz $C \cup C'$, te će oni biti njegovi potomci u DFS drvetu. Odatle sledi da je $v.Post > u.Post$ za svako $u \in C \cup C'$, $u \neq v$, odakle važi $C.Post > C'.Post$.
- Pretpostavimo da DFS obilazak najpre posećuje neki čvor v komponente C', i da u tom trenutku nije posećen nijedan drugi čvor iz C' i C. S obzirom na to da u kondenzovanom grafu grafa G postoji grana (C, C') i da je kondenzovani graf aciklički, u kondenzovanom grafu ne sme postojati grana iz nekog čvora komponente C' ka nekom čvoru komponente C. Stoga DFS obilazak pokrenut iz

čvora v ne stiže ni do jednog čvora komponente C. Odavde sledi da će čvorovi komponente C biti posećeni kasnije, odnosno važi $C.Post > C'.Post$.

□

Zato, ako bismo DFS obilazak pokrenuli iz onog baznog čvora koji ima najmanji odlazni broj, sigurni bismo bili da bi taj DFS obilazak obišao sve čvorove u njegovoj komponenti i nijedan drugi čvor (jer u kondenzovanom grafu ne postoji nijedna grana koja bi mogla da "pobegne" iz te komponente). Međutim, mi taj bazni čvor ne znamo i nije nam lako da ga odredimo. Dakle, ako bismo puštali DFS obilazak od "listova" ka "korenu" kondenzovanog grafa, dobili bismo tačno jednu po jednu komponentu povezanosti, međutim, problem je što mi ne znamo čvorove koji pripadaju komponentama koje su "listovi" u kondenzovanom grafu i nije lako odrediti ih. Sa druge strane, sigurni smo da čvor sa najvećim odlaznim rednim brojem pripada "korenu", tj. komponenti kondenzovanog grafa iz koje ne izlazi ni jedna grana. Ako bismo pustili DFS iz tog čvora, nabrojali bismo i čvorove van te komponente, međutim, ne i ako *grafu obrnemo grane*!

Razmotrimo transponovani graf G^T grafa G, dobijen promenom usmerenja svih grana u grafu: on ima iste komponente jake povezanosti kao i graf G. Drugim rečima, dva čvora su uzajamno dostižna u polaznom grafu ako i samo ako su uzajamno dostižna u njemu transponovanom grafu. Zato, ako pokrenemo DFS obilazak iz bilo kog čvora u transponovanom grafu, sasvim smo sigurni da će u tom obilasku biti dostignuti svi čvorovi iz komponente jake povezanosti kojoj pripada polazni čvor (a možda i neki drugi). Primetimo da će kondenzovani grafovi grafova G i G^T biti međusobno transponovani (slika 2.58, desno). Drugim rečima, u transponovanom kondenzovanom grafu neće postojati grana iz korene komponente ka drugim komponentama. Stoga je za određivanje korene komponente, koja sadrži neki čvor v, dovoljno pokrenuti algoritam DFS iz čvora v u grafu G^T. Na ovaj način se obilaze svi čvorovi komponente čvora v i ništa više od toga. Algoritam se dalje nastavlja po istom principu. Možemo iz grafa ukloniti sve ove čvorove (zapravo, označiti da su posećeni), pronaći čvor sa najvećom vrednošću odlazne numeracije u ostatku grafa, pokrenuti novi DFS obilazak u grafu G^T i tako dalje.

Odavde direktno sledi Kosaradžuov algoritam, koji ima dve faze:

- u prvoj fazi se pokreće odgovarajući DFS obilazak grafa G, sve dok se ne posete svi čvorovi grafa G (obilazak pokrećemo redom iz svakog čvora, pri čemu već posećene čvorove preskačemo); pritom se čvorovi grafa sortiraju rastuće prema vrednosti odlazne numeracije čvorova;
- u drugoj fazi se konstruiše transponovani graf G^T grafa G i pokreneće niz DFS (ili BFS) obilazaka u redosledu dobijenom u prethodnom koraku, odnosno u opadajućem redosledu odlazne numeracije. Svaki skup čvorova, koji je dostižan u narednom obilasku daje novu komponentu jake povezanosti grafa G.

Opisani algoritam se sastoji u pokretanju dva obilaska grafa, pa mu je vremenska složenost $O(|V| + |E|)$.

Primer 2.6.10

U primeru na slici 2.58 pokretanjem DFS obilaska iz čvora 0 *dobijaju se odlazni redni brojevi koji su prikazani na slici.*

- *Pokrećemo u transponovanom grafu DFS obilazak iz čvora sa najvećim odlaznim rednim brojem. To je čvor* 0 *čiji je odlazni redni broj* 21. *DFS obilazak redom nabraja čvorove* 0, 2, 10 *i* 6.
- *Čvorovi sa odlaznim brojevima* 20 *i* 19 *su već posećeni, pa naredni DFS obilazak pokrećemo iz čvora sa odlaznim rednim brojem* 18. *To je čvor* 8 *i DFS obilazak nabraja čvorove* 8, 16, 12, 7 *i* 11.
- *Čvor sa odlaznim rednim brojem* 17 *je već posećen, pa naredni DFS obilazak pokrećemo iz čvora sa odlaznim rednim brojem* 16. *To je čvor* 17 *i DFS obilazak iz njega nabraja samo njega.*
- *Čvorovi sa odlaznim brojevima* 15 *i* 14 *su već posećeni, pa naredni DFS obilazak pokrećemo iz čvora sa odlaznim rednim brojem* 13. *To je čvor* 9 *i DFS obilazak nabraja čvorove* 9, 19, 1 *i* 5.

- *Čvorovi sa odlaznim brojevima* 12, 11, 10, 9 *i* 8 *su već posećeni, pa naredni DFS obilazak pokrećemo iz čvora sa odlaznim rednim brojem* 7. *To je čvor* 15 *i DFS obilazak nabraja čvorove* 15, 14, 18, 20 *i* 4.
- *Čvorovi sa odlaznim brojevima* 6 *i* 5 *su već posećeni, pa naredni DFS obilazak pokrećemo iz čvora sa odlaznim rednim brojem* 4. *To je čvor* 3 *i DFS obilazak nabraja čvorove* 3 *i* 13.
- *Čvorovi sa odlaznim brojevima* 3, 2 *i* 1 *su već posećeni, pa se algoritam završava.*

Primetimo da se u prvom koraku algoritma čvorovi uređuju u obrnutom topološkom redosledu grafa G. Dodatno, algoritam generiše komponente jake povezanosti u opadajućem redosledu odlazne numeracije, odnosno čvorovi kondenzovanog grafa se dobijaju u topološkom redosledu.

Zadatak: Dokaži sve formule!

Potrebno je dokazati sva zadata tvrđenja. Neka tvrđenja su elementarna (data kao pojedinačni iskazi), a neka su implikacije (između iskaza). Za implikacije smatramo da su unapred dokazane, dok elementarni iskazi mogu biti dokazani ili direktno ili primenom pravila *modus ponens* iz ranije dokazanog iskaza i implikacije:

$$\frac{A \qquad A \Rightarrow B}{B}$$

Potrebno je odrediti najmanji mogući broj iskaza koje je potrebno dokazati direktno, tako da se zatim iz njih i datih implikacija mogu primenom pravila *modus ponens* dokazati svi ostali iskazi.

Opis ulaza

Sa standardnog ulaza se unosi broj elementarnih iskaza m ($1 \leq m \leq 10^4$), a zatim broj implikacija $n (1 \leq n \leq 10^4)$. Nakon toga se unosi niz parova brojeva i i j ($0 \leq i < m$), koji predstavljaju implikacije $p_i \Rightarrow p_j$.

Opis izlaza

Na stanadardni izlaz ispisati najmanji broj iskaza koje treba dokazati.

Primer

Ulaz

```
6
7
0 1
1 3
1 2
3 0
2 4
4 2
5 2
```

Izlaz

```
2
```

Objašnjenje

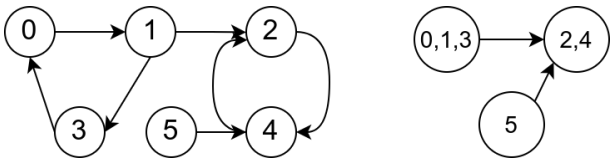
Dovoljno je, na primer, dokazati formule 0 i 5. Nije moguće dokazati samo jednu formulu.

Rešenje

Elementarne iskaze možemo predstaviti čvorovima, a implikacije granama grafa. Svi čvorovi dostižni iz nekog čvora koji odgovara dokazanom tvrđenju odgovaraju tvrđenjima koja mogu biti dokazana primenom implikacija.

Jednostavan slučaj je kada u grafu nema ciklusa, tj. kada je u pitanju usmereni aciklički graf (engl. directed acyclic graph, DAG). Potrebno je i dovoljno dokazati samo ona tvrđenja koja se ne javljaju na desnoj strani nijedne implikacije, tj. samo ona tvrđenja kojima odgovaraju čvorovi grafa ulaznog stepena 0. Zaista, ta tvrđenja ne mogu biti dokazana primenom implikacija (jer nijedna implikacija ne vodi do njih), dok sva ostala tvrđenja mogu biti dokazana primenom implikacija (pošto nema ciklusa, prateći grane unazad, doći će se do nekog čvora čiji je ulazni stepen 0).

Slučaj grafa koji sadrži cikluse je donekle komplikovaniji. Primetimo da za svaku komponentu jake povezanosti grafa važi da je dovoljno dokazati bilo koji njen čvor i tada će sigurno biti moguće dokazati i sve ostale njene čvorove (jer su svi čvorovi u komponenti uzajamno dostižni). Zato je moguće izvršiti redukciju grafa, tj. napraviti kondenzovani graf u kom je svaka komponenta povezanosti predstavljena pojedinačnim čvorom. Kondenzovan graf je uvek aciklički i na njega možemo primeniti rešenje za aciklički graf. Na narednoj slici je prikazan graf iz primera ovog zadatka, kao i njegov kondenzovani graf, koji za čvorove ima tri komponente jake povezanosti polaznog grafa.



Ilustracije radi, u narednoj implementaciji eksplicitno gradimo kondenzovani graf (doduše, za rešenje ovog zadatka to nije neophodno, već je dovoljno samo prebrojati ulazne stepene njegovih čvorova).

```
// graf implikacija
vector<vector<int>> susedi;

// redni broj trenutne komponente jake povezanosti
int brojKomponenata = 0;
// preslikavanje čvorova u redne brojeve njihovih komponenata
vector<int> komponente;


// liste suseda u kondenzovanom grafu u kome su čvorovi komponente jake povezanosti
// polaznog grafa
vector<vector<int>> susedneKomponente;

// određuju se komponente jake povezanosti i gradi se kondenzovani graf
// liste povezanosti ovog grafa se smeštaju u globalnu promenljivu susedneKomponente
void napraviKondenzovaniGraf() {
  // određuju se komponente jake povezanosti
  // broj komponenata je sadržan globalnoj promenljivoj brojKomponenata
  // komponenta svakog čvora je određena globalnim nizom komponente
  odrediKomponente();
  // broj čvorova kondenzovanog grafa je broj komponenata jake povezanosti
  susedneKomponente.resize(brojKomponenata);
  // grane koje su već dodate u kondenzovani graf
  set<pair<int, int>> dodateGrane;
  // prolazimo kroz sve grane originalnog grafa
  for (int cvor = 0; cvor < susedi.size(); cvor++)
```

```
    for (int sused : susedi[cvor]) {
      // u kondenzovani graf dodajemo granu između komponente čvora i
      // komponente suseda (ako nije već dodata)
      if (komponente[cvor] != komponente[sused] &&
          dodateGrane.count({komponente[cvor], komponente[sused]}) == 0) {
        susedneKomponente[komponente[cvor]].push_back(komponente[sused]);
        dodateGrane.emplace(komponente[cvor], komponente[sused]);
      }
    }
}

int main() {
  // ucitavamo podatke o iskazima i implikacijama
  // ...

  // pravimo kondenzovani graf u kome su cvorovi komponente povezanosti
  napraviKondenzovaniGraf();

  // određujemo ulazne stepene čvorova kondenzovanog grafa
  vector<int> ulazniStepen(susedneKomponente.size(), 0);
  for (int cvor = 0; cvor < susedneKomponente.size(); cvor++)
    for (int sused : susedneKomponente[cvor])
      ulazniStepen[sused]++;

  // brojimo čvorove čiji je ulazni stepen nula
  int broj = 0;
  for (int cvor = 0; cvor < susedneKomponente.size(); cvor++)
    if (ulazniStepen[cvor] == 0)
      broj++;

  cout << broj << endl;

  return 0;
}
```

2.7 Ojlerovi i Hamiltonovi putevi

U nekim grafovskim problemima potrebno je pronaći put između dva čvora koji posećuje svaku granu grafa tačno jednom ili, dualno, put koji posećuje svaki čvor grafa tačno jednom. U ovom poglavlju bavićemo se problemima ispitivanja da li u grafu postoje ovakve vrste puteva. Iako ova dva problema na prvi pogled nalikuju, pokazaće se da se tehnike za njihovo rešavanje značajno razlikuju.

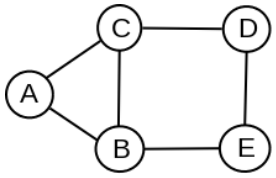
2.7.1 Ojlerovi putevi i ciklusi

Razmotrimo naredni problem: poštar treba da dostavi određen broj pisama na različite adrese u nekoliko susednih ulica. Poštar gleda u mapu i uočava na koji način su te ulice međusobno povezane. Crta pojednostavljenu mapu i pritom svaku od raskrsnica označava nekim brojem. Poštar želi da dostavi sva pisma, a da pritom svakim putem prođe tačno jednom. Postavlja se pitanje da li je to moguće uraditi. Ovaj problem moguće je modelovai u terminima grafova: svaka raskrsnica predstavlja čvor grafa, a put koji vodi od jedne raskrsnice do druge granu grafa. Dati problem se onda svodi na pitanje da li u grafu postoji put koji sadrži svaku granu grafa tačno jednom.

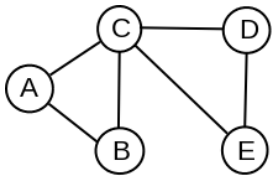
Ojlerov put (engl. Eulerian trail, Eulerian path) je put u grafu koji prolazi kroz svaku granu grafa tačno jednom (pri čemu neke čvorove može posetiti i više puta). *Ojlerov ciklus* (engl. Eulerian circuit, Eulerian cycle) je Ojlerov put čiji se početni i krajnji čvor poklapaju.

Primer 2.7.1

U neusmerenom grafu prikazanom na slici 2.59 postoji Ojlerov put (C, D, E, B, C, A, B)*, ali ne postoji Ojlerov ciklus. U grafu prikazanom na slici 2.60 postoji Ojlerov ciklus* (C, D, E, C, A, B, C)*.*



Slika 2.59: Neusmeren graf koji sadrži Ojlerov put (na primer (C, D, E, B, C, A, B), ali ne sadrži Ojlerov ciklus.



Slika 2.60: Neusmeren graf koji sadrži Ojlerov ciklus. Jedan Ojlerov ciklus u ovom grafu je (C, D, E, C, A, B, C).

U nastavku ćemo se baviti određivanjem Ojlerovih puteva i ciklusa.

Problem

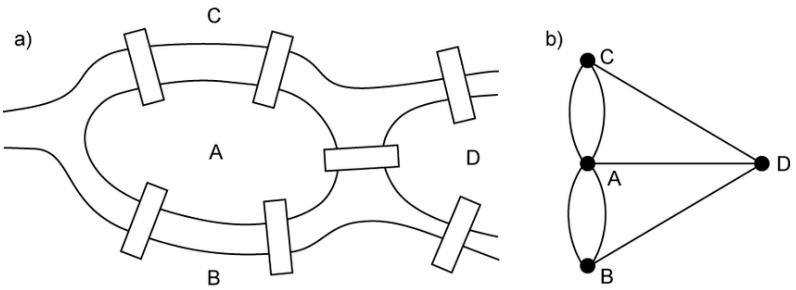
Za dati graf (bilo usmeren, bio neusmeren) utvrditi da li ima Ojlerov put ili Ojlerov ciklus i ako ima odrediti ga.

Pojam Ojlerovih grafova u vezi je sa, kako se smatra, prvim rešenim problemom teorije grafova. Švajcarski matematičar Leonard Ojler naišao je 1736. godine na sledeći zadatak. Grad Kenigsberg, danas Kalinjingrad, leži na obalama i na dva ostrva na reci Pregel, kao što je prikazano na slici 2.61, levo. Grad je povezan putem sedam mostova. Pitanje koje je mučilo mnoge tadašnje građane Kenigsberga bilo je da li je moguće početi šetnju iz bilo koje tačke u gradu i vratiti se u polaznu tačku, prelazeći pri tome svaki most tačno jednom. Ovaj problem se može formulisati kao sledeći problem iz teorije grafova: da li je moguće u neusmerenom povezanom grafu pronaći ciklus, koji svaku granu grafa sadrži tačno jednom, tj. Ojlerov ciklus. Drugim rečima, da li je moguće nacrtati graf sa slike 2.61, desno, ne dižući olovku sa papira, tako da olovka svoj put završi na mestu sa koga je i krenula. Napomenimo da ovaj graf ima višestruke grane između parova čvorova, pa strogo gledano po definiciji nije graf, već multigraf. Ojler je dokazao da je ovakav obilazak moguć ako i samo ako je graf povezan i svi njegovi čvorovi imaju paran stepen (teorema 2.7.1 u nastavku). Grafovi koji sadrže Ojlerov ciklus zovu se *Ojlerovi grafovi*. Pošto "graf" na slici 2.61, desno, ima čvorove neparnog stepena, zaključujemo da problem Kenigsberških mostova nema rešenje.

Pre nego što dokažemo karakterizaciju Ojlerovih grafova preko stepena čvorova, dokažimo jednu lemu.

Lema 2.7.1 [Ciklus u Ojlerovom neusmerenom grafu]

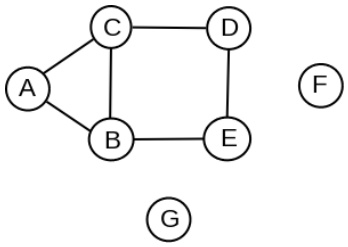
Ako u neusmerenom grafu G svaki čvor ima paran stepen, onda u tom grafu postoji ciklus.



Slika 2.61: Problem Kenigsberških mostova, i odgovarajući multigraf.

Dokaz. Pretpostavimo da smo započeli obilazak grafa iz proizvoljnog čvora u proizvoljnim redosledom. Sigurno je da ćemo se tokom obilaska vratiti u čvor u, jer kad god uđemo u neki čvor, smanjujemo njegov stepen za jedan, činimo ga neparnim, pa ga uvek možemo i napustiti. Naravno, ovakav obilazak ne mora da sadrži sve grane grafa. Dakle, u svakom grafu u kom je stepen svih čvorova paran, za svaki čvor u mora da postoji ciklus koji počinje i završava se u u. □

Primetimo da Ojlerov put i Ojlerov ciklus može postojati i u nepovezanom grafu, ukoliko graf, pored jedne povezane komponente, sadrži samo izolovane čvorove (slika 2.62).



Slika 2.62: Nepovezani graf koji sadrži Ojlerov put (na primer (C, D, E, B, C, A, B)).

Teorema 2.7.1 **[Karakterizacija Ojlerovih neusmerenih grafova]**

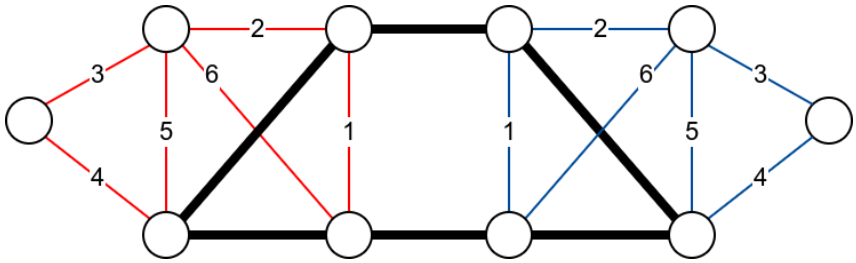
U neusmerenom grafu $G = (V, E)$ postoji Ojlerov ciklus ako i samo u grafu G svi čvorovi imaju parni stepen, i svi čvorovi stepena različitog od nula pripadaju istoj komponenti povezanosti.

Dokaz. Lako je pokazati da ako u grafu postoji Ojlerov ciklus, onda svi čvorovi grafa moraju imati paran stepen. Naime, za vreme obilaska ciklusa, u svaki čvor se ulazi isto toliko puta koliko puta se iz njega izlazi. Pošto se svaka grana prolazi tačno jednom, broj grana susednih proizvoljnom čvoru mora biti paran.

Da bismo indukcijom dokazali da je ovaj uslov i dovoljan da bi graf imao Ojlerov ciklus, moramo najpre da izaberemo parametar po kome će biti izvedena indukcija. Taj izbor treba da omogući smanjivanje problema, bez njegove promene. Ako uklonimo čvor ili granu iz grafa, stepeni čvorova u dobijenom grafu nisu više svi parni. Treba da uklonimo takav skup grana S, da za svaki čvor u grafa G broj grana iz skupa S susednih sa u ostane paran (makar i 0). Proizvoljan ciklus zadovoljava ovaj uslov, a na osnovu leme 2.7.1 svaki graf čiji su svi čvorovi parnog stepena sadrži neki ciklus. Sada možemo da formulišemo induktivnu hipotezu i dokažemo teoremu.

Induktivna hipoteza: Povezani neusmereni graf sa manje od m grana čiji svi čvorovi imaju paran stepen sadrži Ojlerov ciklus, koji se može efektivno pronaći.

Posmatrajmo graf $G = (V, E)$ koji sadrži m grana u kom svi čvorovi imaju paran stepen. Neka je P neki ciklus u grafu G (koji postoji, na osnovu leme 2.7.1), i neka je G' graf dobijen uklanjanjem grana ciklusa P iz grafa G. Stepeni svih čvorova u grafu G' su parni, jer je broj uklonjenih grana susednih bilo kom čvoru paran. Ipak se induktivna hipoteza ne može primeniti na graf G', jer on ne mora biti povezan (slika 2.63).



Slika 2.63: Primer konstrukcije Ojlerovog ciklusa indukcijom. Punom linijom izvučene su grane pomoćnog ciklusa. Izbacivanjem grana ovog ciklusa iz grafa, dobija se graf sa dve komponente povezanosti.

Neka su $G'_1, G'_2, \ldots, G'_k$ komponente povezanosti grafa G'. U svakoj komponenti povezanosti stepeni svih čvorova su parni. Pored toga, broj grana u svakoj komponenti je manji od m jer je ukupan broj grana u svim komponentama manji od m. Prema tome, induktivna hipoteza se može primeniti na svaku od komponenti posebno: u svakoj komponenti G'_i postoji Ojlerov ciklus P'_i, i mi znamo da ga pronađemo. Potrebno je sada sve ove cikluse objediniti sa pomoćnim ciklusom P u jedan Ojlerov ciklus za graf G. Polazimo iz proizvoljnog čvora ciklusa P ("magistralnog puta") sve dok ne dođemo do nekog čvora v_j koji pripada nekoj komponenti G'_j. Tada obilazimo komponentu G'_j ciklusom P'_j ("lokalnim putem") i vraćamo se u čvor v_j. Nastavljamo obilazak ciklusa P na taj način, obilazeći cikluse komponenti u trenutku nailaska na njih. Na kraju obilaska ćemo se vratiti u polazni čvor. U tom trenutku sve grane grafa G smo prošli tačno jednom, što znači da je konstruisan Ojlerov ciklus.

Ovaj dokaz sugeriše efikasan rekurzivni algoritam za konstrukciju Ojlerovog ciklusa u grafu. □

Videli smo da neusmereni graf može imati Ojlerov put, a da istovremeno nema Ojlerov ciklus (slika 2.59). Naredna lema (koja se dokazuje veoma jednostavno) daje potreban i dovoljan uslov da neusmeren graf ima Ojlerov put.

Teorema 2.7.2 [Postojanje Ojlerovog puta i ciklusa u neusmerenom grafu]

Neusmereni graf ima Ojlerov put ako i samo ako svi čvorovi stepena različitog od nula pripadaju jednoj komponenti povezanosti i važi jedan od narednih uslova:

- *stepen svakog čvora je paran ili*
- *stepen tačno dva čvora je neparan, a ostalih čvorova je paran.*

Ako važi prva pretpostavka onda je svaki Ojlerov put istovremeno i Ojlerov ciklus. Ukoliko važi druga pretpostavka, čvorovi neparnog stepena su početni i krajnji čvor Ojlerovog puta i u ovakvom grafu ne postoji Ojlerov ciklus.

U grafu sa slike 2.59 čvorovi B i C su neparnog stepena, dok su ostali čvorovi parnog stepena, te čvorovi B i C predstavljaju početni i krajnji čvor Ojlerovog puta. Dodatno, u ovom grafu ne postoji Ojlerov ciklus.

Situacija je slična i u usmerenim grafovima. Naredna lema daje potreban i dovoljan uslov da u usmerenom grafu postoji Ojlerov put.

Teorema 2.7.3 [Postojanje Ojlerovog puta i ciklusa u usmerenom grafu]

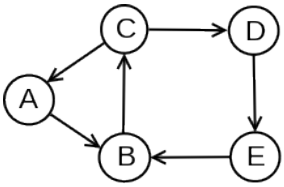
U usmerenom grafu postoji Ojlerov put ako i samo ako svi čvorovi koji nisu izolovani pripadaju istoj slabo povezanoj komponenti povezanosti (istoj komponenti povezanosti odgovarajućeg neusmerenog grafa) i važi jedan od narednih uslova:

- *ulazni i izlazni stepeni svih čvorova su međusobno jednaki ili*
- *ulazni stepen veći je za jedan od izlaznog stepena tačno za jedan čvor, izlazni stepen veći je za jedan od ulaznog stepena tačno za jedan čvor, dok su za sve ostale čvorove ulazni i izlazni stepen međusobno jednaki.*

U prvom slučaju, svaki Ojlerov put je i Ojlerov ciklus, a u drugom slučaju Ojlerov put počinje u čvoru čiji je izlazni stepen veći za jedan, a završava se u čvoru čiji je ulazni stepen veći za jedan.

Primer 2.7.2

U grafu prikazanom na slici 2.64 su ulazni i izlazni stepeni čvorova A, E i D međusobno jednaki, ulazni stepen čvora B je veći za jedan od izlaznog, dok je ulazni stepen čvora C za jedan manji od izlaznog stepena. U ovom grafu postoji Ojlerov put koji počinje u čvoru C, a završava se u čvoru B (npr. (C, D, E, B, C, A, B)), dok Ojlerov ciklus ne postoji.



Slika 2.64: Usmereni graf koji sadrži Ojlerov put (C, A, B, C, D, E, B), ali ne sadrži Ojlerov ciklus.

2.7.1.1 Hirholcerov algoritam

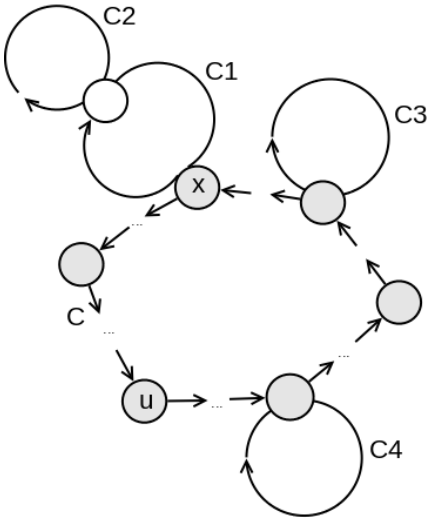
Iako se iz dokaza teoreme 2.7.1 može izdvojiti ideja za konstrukciju Ojlerovog ciklusa u povezanom neusmerenom grafu kod koga su stepeni svih čvorova parni, potrebno je precizirati detalje implementacije da bi se umesto rekurzivne dobila efikasna iterativna implementacija. Jedan efikasan način za konstruisanje Ojlerovog ciklusa u Ojlerovom grafu predstavlja *Hirholcerov algoritam*, zasnovan na ideji prikazanoj u dokazu teoreme 2.7.1. Algoritam se može primeniti i na neusmerene i na usmerene grafove. On se sastoji od nekoliko etapa, pri čemu se u svakoj etapi prethodno pronađeni ciklus proširuje narednim ciklusom u novi, veći ciklus. Postupak se zaustavlja kada se sve grane grafa dodaju u ciklus (i to je konačan, Ojlerov ciklus).

Polazi se iz proizvoljnog čvora u u grafu. U svakom koraku prelazi se proizvoljnom neposećenom granom do nekog suseda trenutnog čvora. Ovaj korak se ponavlja sve dok se ne vratimo u polazni čvor u. U njega se moramo vratiti u nekom momentu, jer je stepen svakog čvora paran. Na ovaj način konstruiše se inicijalni ciklus. Ukoliko on prolazi skupom svih grana, Ojlerov ciklus je konstruisan. Inače, ciklus se proširuje na sledeći način: polazeći iz čvora u duž ciklusa pronalazi se prvi čvor v koji pripada tekućem ciklusu i koji ima granu koja nije uključena u ciklus (u slučaju usmerenog grafa vraćamo se unazad duž ciklusa i tražimo prvi čvor koji ima izlaznu granu koja nije uključena u ciklus). Tom granom se kreće u otkrivanje novog ciklusa koji se sastoji isključivo od grana koje još uvek nisu u prethodnom ciklusu. Put se pre ili kasnije vraća u čvor v i time se pronalazi novi ciklus koji dodajemo u tekući ciklus. Na ovaj način se od ova dva ciklusa konstruiše novi ciklus. Postupak se nastavlja dok pronađeni ciklus ne obuhvati sve grane grafa.

Primer 2.7.3

Razmotrimo graf sa slike 2.65. Obilazak kreće iz čvora u i inicijalno se konstruiše ciklus C čiji su čvorovi označeni sivom bojom. Nakon toga se unazad vraćamo čvorovima ciklusa C počev od čvora u i redom se sa glavnim ciklusom objedinjavaju ciklusi C_1, C_2, C_3 i C_4.

Čvorove inicijalnog ciklusa stavljamo na stek u redosledu obilaska. Naime, stek nam omogućava da se efikasno vraćamo unazad kroz ciklus tražeći čvor iz kog nisu sve grane posećene i iz kojeg možemo da započnemo novi ciklus. Ako je to čvor na vrhu steka, krećemo obilazak novog ciklusa dodajući njegove čvorove na vrh steka. U suprotnom, ako su sve grane čvora na vrhu steka posećene, završili smo obradu tog čvora i njega dodajemo na početak niza čvorova koji određuje Ojlerov ciklus. Primetimo da ni u jednom trenutku ne moramo eksplicitno da proveravamo da li smo se vratili u početni čvor tekućeg ciklusa. Kada stek postane prazan, to znači da smo za sve čvorove razmotrili sve grane koje polaze iz njih, tj. da su sve grane obrađene i da je Ojlerov ciklus konstruisan.



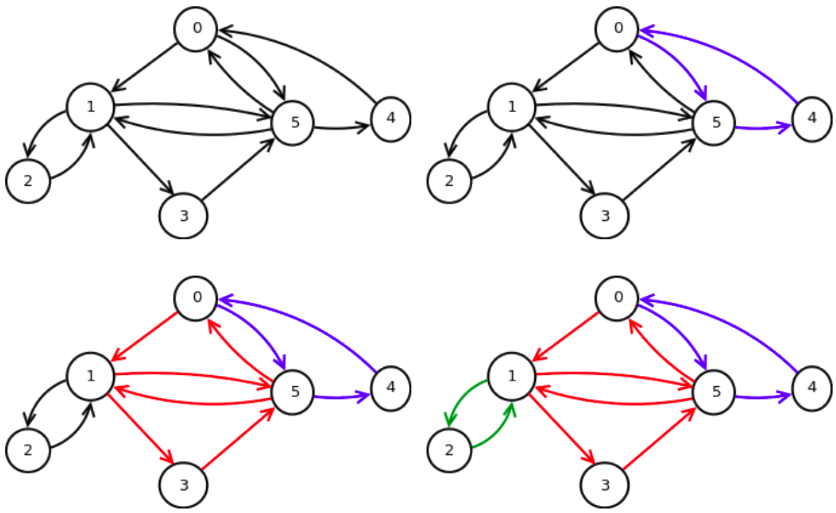
Slika 2.65: Objedinjavanje ciklusa u novi ciklus.

Ako graf G sadrži samo Ojlerov put, a ne i Ojlerov ciklus, algoritam započinje svoj rad iz čvora čiji je izlazni stepen za jedan veći od ulaznog. Alternativno, u graf G se može dodati grana kojom se postiže uslov da graf ima Ojlerov ciklus i zatim iskoristiti prethodni algoritam. Nakon pronalaska Ojlerovog ciklusa u dopunjenom grafu, ta grana se uklanja iz ciklusa i na taj način ostajemo sa Ojlerovim putem.

U nastavku je prikazana implementacija Hirholcerovog algoritma koji pronalazi Ojlerov ciklus u usmerenom grafu. Pretpostavlja se da su svi čvorovi parnog stepena, pa obilazak može da krene iz proizvoljnog čvora (u implementaciji obilazak uvek kreće od čvora 0).

```
vector<int> pronadjiOjlerovCiklus() {
  // rezultujuci ciklus, odredjen cvorovima kroz koje se prolazi
  vector<int> ojlerovCiklus;

  // stek na koji stavljamo cvorove u redosledu obilaska
  stack<int> tekuciPut;

  // na stek dodajemo polazni cvor
  tekuciPut.push(0);

  // sve dok postoji neki cvor na tekucem putu
  while (!tekuciPut.empty()) {
    int tekuciCvor = tekuciPut.top();

    // ako iz tekuceg cvora postoji jos neka grana koju nismo posetili
    if (listaSuseda[tekuciCvor].size() > 0) {
      // pronalazimo cvor do koga postoji grana iz tekuceg cvora;
      // uzimamo poslednji iz liste povezanosti jer je njega lako
      // ukloniti iz liste povezanosti
      int naredniCvor = listaSuseda[tekuciCvor].back();
      // brisemo granu iz tekuceg ka narednom cvoru
      listaSuseda[tekuciCvor].pop_back();

      // napredujemo ka narednom cvoru
      tekuciCvor = naredniCvor;
      tekuciPut.push(naredniCvor);
```

```
    }
    // ako iz tog cvora ne postoji neposecena grana
    else {
      // cvor dodajemo u Ojlerov ciklus
      ojlerovCiklus.push_back(tekuciCvor);
      // vracamo se unazad da bismo nasli preostale cikluse
      tekuciPut.pop();
    }
  }
  // obrcemo cvorove u ciklusu
  reverse(begin(ojlerovCiklus), end(ojlerovCiklus));

  return ojlerovCiklus;
}
```



Slika 2.66: Ilustracija izvršavanja Hirholcerovog algoritma u slučaju datog usmerenog grafa.

Primer 2.7.4

Na slici 2.66 ilustrovano je izvršavanje Hirholcerovog algoritma na datom usmerenom Ojlerovom grafu. Algoritam startuje iz čvora 0 *i pronalazi, recimo, ciklus* $0, 5, 4, 0$*. Grane ovog ciklusa se tokom obilaska uklanjaju iz grafa. Prilikom povratka u čvor* 0 *proverava se da li postoji neka neposećena grana u grafu koja polazi iz čvora* 0 *i pošto postoji grana* $(0, 1)$ *nastavljamo granom ka čvoru* 1 *i pronalazimo novi ciklus* $(0, 1, 5, 1, 3, 5, 0)$*. Grane i ovog ciklusa se tokom obilaska uklanjaju iz grafa. Put kojim smo se do sad kretali sadrži redom čvorove* $(0, 5, 4, 0, 1, 5, 1, 3, 5, 0)$*. U ovom trenutku konstatujemo da iz čvora* 0 *ne postoji nijedna neposećena grana u grafu, te jedan po jedan čvor tekućeg puta prebacujemo s kraja puta u Ojlerov ciklus; pre nego što čvor prebacimo u Ojlerov ciklus proveravamo da li postoji još neka neposećena grana u grafu koja kreće iz tog čvora, ako postoji krećemo u potragu za novim ciklusom iz tog čvora, a ako ne postoji, onda čvor prebacujemo u Ojlerov ciklus. Nakon prebacivanja čvorova* 0*,* 5 *i* 3 *utvrđuje se da iz čvora* 1 *postoji grana ka čvoru* 2 *koja do sada nije bila posećena te otkrivamo novi ciklus* $1, 2, 1$ *i čvorove ovog ciklusa dodajemo na kraj tekućeg puta. Nakon ovog ciklusa, sve dok ne iscrpemo sve čvorove iz tekućeg puta nećemo pronaći nijednu novu granu. Konačno, zaključujemo da je u ovom grafu jedan od Ojlerovih ciklusa* $(0, 5, 4, 0, 1, 5, 1, 2, 1, 3, 5, 0)$*. Sadržaj tekućeg puta i dela konstruisanog Ojlerovog ciklusa korak po korak prikazani su u tabeli 2.2.*

Tabela 2.2: Primer izvršavanja Hirholcerovog algoritma za graf sa slike 2.66.

tekuciCvor	*tekuciPut*	*ojlerovCiklus*
0	(0)	
5	$(0, 5)$	
4	$(0, 5, 4)$	
0	$(0, 5, 4, 0)$	
1	$(0, 5, 4, 0, 1)$	
5	$(0, 5, 4, 0, 1, 5)$	
1	$(0, 5, 4, 0, 1, 5, 1)$	
3	$(0, 5, 4, 0, 1, 5, 1, 3)$	
5	$(0, 5, 4, 0, 1, 5, 1, 3, 5)$	
0	$(0, 5, 4, 0, 1, 5, 1, 3, 5, 0)$	
5	$(0, 5, 4, 0, 1, 5, 1, 3, 5)$	(0)
3	$(0, 5, 4, 0, 1, 5, 1, 3)$	$(0, 5)$
1	$(0, 5, 4, 0, 1, 5, 1)$	$(0, 5, 3)$
2	$(0, 5, 4, 0, 1, 5, 1, 2)$	$(0, 5, 3)$
1	$(0, 5, 4, 0, 1, 5, 1, 2, 1)$	$(0, 5, 3)$
2	$(0, 5, 4, 0, 1, 5, 1, 2)$	$(0, 5, 3, 1)$
1	$(0, 5, 4, 0, 1, 5, 1)$	$(0, 5, 3, 1, 2)$
5	$(0, 5, 4, 0, 1, 5)$	$(0, 5, 3, 1, 2, 1)$
1	$(0, 5, 4, 0, 1)$	$(0, 5, 3, 1, 2, 1, 5)$
0	$(0, 5, 4, 0)$	$(0, 5, 3, 1, 2, 1, 5, 1)$
4	$(0, 5, 4)$	$(0, 5, 3, 1, 2, 1, 5, 1, 0)$
5	$(0, 5)$	$(0, 5, 3, 1, 2, 1, 5, 1, 0, 4)$
0	(0)	$(0, 5, 3, 1, 2, 1, 5, 1, 0, 4, 5)$
$-$		$(0, 5, 3, 1, 2, 1, 5, 1, 0, 4, 5, 0)$

Vreme izvršavanja Hirholcerovog algoritma u usmerenom grafu koji je predstavljen listama povezanosti iznosi $O(|E|)$. Naime, ukoliko je $|V| > |E|$, onda graf ima izolovane čvorove koji se tokom algoritma ne razmatraju. Istaknimo i to da ukoliko je graf neusmeren, onda prilikom brisanja neke grane, treba obrisati obe njene kopije: (u, v) i (v, u), što je operacija koja se ne izvršava efikasno u slučaju reprezentacije grafa listama povezanosti. Stoga je Hirholcerov algoritam u slučaju neusmerenog grafa manje efikasan.

Ako se traži Ojlerov put (a ne ciklus), algoritam se primenjuje u istom obliku, jedino što izvršavanje mora da počne od čvora čiji je izlazni stepen za jedan veći od ulaznog.

2.7.1.2 Flerijev algoritam

Drugi poznati algoritam za konstrukciju Ojlerovih ciklusa ili puteva u grafu, pored Hirholcerovog, je i *Flerijev algoritam* (engl. Fleury's algorithm). On kreće od proizvoljnog čvora i u svakom koraku tekući put proširuje nekom granom iz tekućeg čvora, koju zatim uklanja iz grafa. Ako je moguće, bira se grana koja nije most tj. grana koja ne razbija graf na više komponenata povezanosti. Ako takva grana ne postoji, onda se put proširuje granom koja jeste most. Naime, ako bismo izabrali granu koja je most kada iz tog čvora postoji još neka grana, prešli bismo u komponentu povezanosti kojoj ne pripada taj čvor i ne bismo mogli da se vratimo do njega i da prođemo tom granom. Naglasimo da iako polazni Ojlerov graf sigurno nema mostova, oni se mogu pojaviti tokom izvršavanja algoritma, jer se grafu izbacuju grane.

Ovaj algoritam se zasniva na detekciji mostova u grafu u svakom koraku i manje je efikasan – njegova složenost iznosi $O(|E|^2)$. Postoje inkrementalni aloritmi za detekciju mostova čijom se upotrebom složenost algoritma može popraviti, ali Flerijev algoritam i dalje ostaje složeniji i neefikasniji od Hirholcerovog, pa ga nećemo dalje analizirati.

2.7.2 Hamiltonovi putevi i ciklusi

Jedan od važnih problema u bioinformatici jeste mapiranje genoma, gde je zadatak iskombinovati veliki broj malih fragmenata genetskog koda u jedinstvenu genomsku sekvencu. Ovaj problem se može razmatrati kao grafovski: svaki od fragmenata odgovara čvoru grafa, a svako preklapanje (poklapanje kraja nekog fragmenta sa početkom nekog drugog) grani grafa. Pitanje koje se postavlja jeste da li je moguće napraviti prost put u grafu koji prolazi kroz svaki čvor grafa tačno jednom.

Hamiltonov put (engl. Hamiltonian path) je put koji posećuje svaki čvor grafa tačno jednom. Ako Hamiltonov put počinje i završava se u istom čvoru on se naziva *Hamiltonov ciklus* (engl. Hamiltonian cycle, Hamiltonian circuit).

Primer 2.7.5

Graf prikazan na slici 2.59 sadrži Hamiltonov put A, B, C, D, E. Graf sa slike 2.59 ima i Hamiltonov ciklus koji počinje i završava se u čvoru A: to je ciklus (A, B, E, D, C, A).

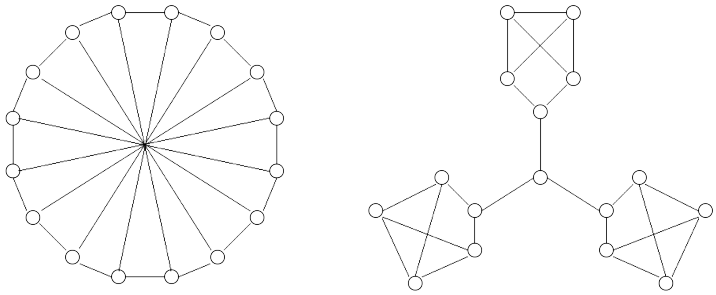
Graf koji sadrži Hamiltonov ciklus zove se *Hamiltonov graf* (engl. Hamiltonian graph). Hamiltonovi ciklusi dobili su ime u čast Vilijama Hamiltona koji je 1857. godine izumeo slagalicu koja uključuje potragu za ovom vrstom ciklusa u grafu sačinjenom od ivica dodekaedra.

Prirodno se razmatra problem određivanja Hamiltonovih ciklusa.

Problem

Za dati graf (bilo usmeren, bilo neusmeren) utvrditi da li ima Hamiltonov ciklus i ako ima odrediti ga.

Za razliku od problema Ojlerovih ciklusa, problem nalaženja Hamiltonovih ciklusa (odnosno karakterizacije Hamiltonovih grafova) je vrlo težak. Da bi se proverilo da li je graf Ojlerov, dovoljno je znati stepene njegovih čvorova. Za utvrđivanje da li je graf Hamiltonov to nije dovoljno. Zaista, dva grafa prikazana na slici 2.67 imaju po 16 čvorova stepena 3 (dakle imaju isti broj čvorova i iste stepene čvorova), ali je prvi Hamiltonov, a drugi očigledno nije. Problem ispitivanja da li je dati graf Hamiltonov spada u klasu NP-kompletnih problema, pa nije poznato da li postoji subeksponencijalni algoritam za njegovo rešavanje.



Slika 2.67: Dva grafa sa po 16 čvorova stepena 3, od kojih je prvi Hamiltonov, a drugi nije.

Algoritam grube sile podrazumeva proveru svih permutacija čvorova i njegova složenosti je $O(|V|!)$, što je izrazito neefikasno.

Moguće je dizajnirati algoritam zasnovan na dinamičkom programiranju čija bi složenost bila donekle bolja $O(|V| \cdot 2^{|V|})$. Osnovna ideja je da se rešavaju potproblemi oblika: "Za fiksirani početni čvor u, skup čvorova S i završni čvor v, da li postoji put koji počinje iz čvora u, prolazi kroz sve čvorove iz S i završava se čvorom v". Obeležimo ovu logičku vrednost sa $DP(S, v)$. Bazu čini prazan skup S i tada $DP(\emptyset, v)$ važi tačno za čvorove v za koje postoji grana (u, v). Za neprazne skupove S važi $DP(S, v)$ ako i samo ako postoji $v' \in S$ tako da važi $DP(S \setminus \{v\}, v')$ i postoji grana (v', v). Na kraju, Hamiltonov ciklus postoji ako postoji neki čvor v takav da važi $DP(V \setminus \{u\}, v)$ i postoji grana (v, u).

Mi se u ovom udžbeniku nećemo detaljnije baviti Hamiltonovim putevima i ciklusima.

Zadatak: De Brujnov niz

De Brujnov niz je ciklični niz brojeva iz intervala $[0, k)$, koji kao svoje segmente sadrži sve moguće n-točlane niske čiji su elementi brojevi iz intervala $[0, k)$. Na primer, niz `0110` je de Brujnov za $k = 2$ i $n = 2$, jer sadrži redom `01`, `11`, `10` i, na kraju `00` (jer je dopušteno da se elementi čitaju i ciklično, deo sa kraja niza, pa zatim sa početka). Najkraći de Brujnov niz sadrži tačno k^n elemenata. Zaista, konstruktivno ćemo pokazati da postoji de Brujnov niz dužine k^n. Sa druge strane, postoji tačno k^n različitih n-točlanih nizova čiji su elementi brojevi iz intervala $[0, k)$, svaki od njih se javlja u de Brujnovom nizu i počinje na različitoj poziciji, pa zato dužina de Brujnovog niza mora biti bar k^n. Napisati program koji za dato n i k ispisuje neki najkraći de Brujnov niz.

Opis ulaza

Broj n ($1 \leq n \leq 10$), a zatim i broj k ($2 \leq k \leq 4$).

Opis izlaza

Na standardni izlaz ispisati bilo koji najkraći de Brujnov niz.

Primer 1

Ulaz

```
2 2
```

Izlaz

```
0110
```

Primer 2

Ulaz

```
3 3
```

Izlaz

```
002221211122012021011020010
```

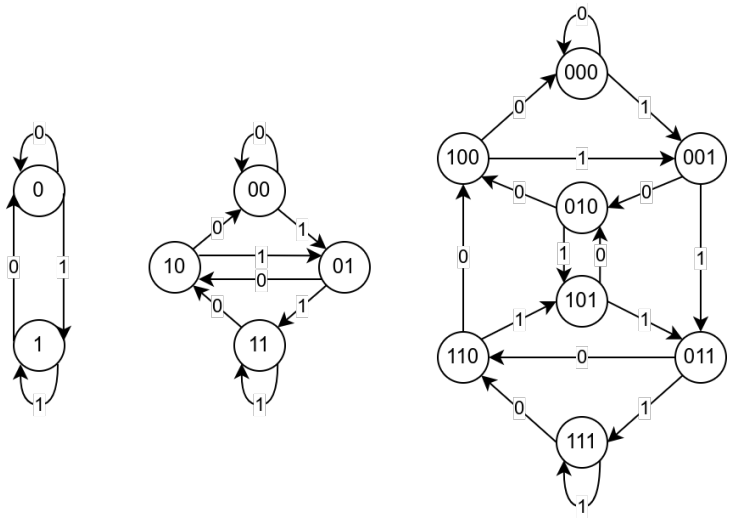
Rešenje

Svaka od k^n nizova treba da se javi tačno jednom kao segmenta traženog niza, pri čemu se svaka dva uzastopna segmenta poklapaju na $n - 1$ pozicija. Dva segmenta mogu biti uzastpna ako i samo ako se poslednjih $n-1$ brojeva prvog poklapa sa $n-1$ prvih brojeva drugog segmenta. Ovo je moguće modelovati usmerenim grafom. Svaki niz brojeva iz intervala $[0, k)$ dužine n možemo predstaviti jednim čvorom grafa (graf, dakle, ima n^k čvorova), a grana od čvora A do čvora B postoji ako i samo ako niz A i niz B mogu da budu uzastopni segmenti (koji se preklapaju na $n - 1$ pozicija). Primeri takvih grafova za $k = 2$ i $n = 1$, $n = 2$ i $n = 3$, su prikazani na slici.

Svaki put u grafu odgovara binarnom nizu i obratno. Na primer, de Brujnom nizu `00111010` za $n = 3$ i $k = 2$, odgovara put kroz čvorove `001`, `011`, `111`, `110`, `101`, `010`.

DeBrujnov niz se može dobiti pronalaskom Hamiltonovog ciklusa u grafu (jer Hamiltonov ciklus obilazi svaki čvor tačno jednom, pre nego što se vrati u početni čvor). Dužina niza koja odgovara Hamiltonovom ciklusu je $k^n + n - 1$, pri čemu se poslednjih $n - 1$ elemenata niza poklapa sa prvih $n - 1$ elemenata, pa se oni mogu izostaviti (što odgovara izostavljanju poslednjih $n - 1$ elemenata Hamiltonovog ciklusa.

Međutim, de Brujinov niz za dato n i k se može dobiti i od grafa za to k i $n - 1$. Svakom segmentu $a_1, a_2, \ldots, a_{n-1}, a_n$ odgovara grana od čvora $a_1, \ldots, a_{n-1}$ do čvora $a_2, \ldots a_n$. U tom slučaju se kroz svaku granu prolazi samo jednom, što znači da de Brujnovim nizovima odgovaraju Ojlerovi ciklusi i obratno.



Slika 2.68: De Brujnov graf

Zadatak ze zato može rešiti pronalaženjem Ojlerovog ciklusa u grafu za dato k i $n-1$. Ojlerov ciklus možemo pronaći Hirholcerovim algoritmom. Ako znamo sve čvorove Ojlerovog ciklusa (pri čemu su prvi i poslednji jednaki), niz možemo dobiti čitajući redom njihove poslednje brojeve, i izostavljajući poslednji čvor.

U narednoj implementaciji rekurzivno nabrajamo sve varijacije i tako gradimo sve čvorove grafa kao eksplicitne niske cifara (implementacija bi se mogla unaprediti korišćenjem neke efikasnije reprezentacije grafa).

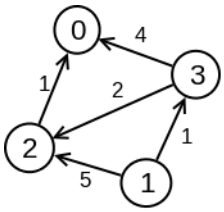
```
// pomocna funkcija za generisanje varijacija
void sviCvorovi(int m, int k, string& s, vector<string>& rezultat) {
  if (m == 0) {
    rezultat.push_back(s);
    return;
  }
  for (int i = 0; i < k; i++) {
    s[m-1] = '0' + i;
    sviCvorovi(m-1, k, s, rezultat);
  }
}

// cvorovi grafa su sve varijacije duzine m od azbuke {0, 1, .., k-1}
vector<string> sviCvorovi(int m, int k) {
  vector<string> rezultat;
  string s(m, ' ');
  sviCvorovi(m, k, s, rezultat);
  return rezultat;
}

int main() {
  int n;
  cin >> n;
  int k;
  cin >> k;
```

```
  // generisemo vektor svih cvorova grafa
  // cvorovi su varijacije duzine m od azbuke {0, 1, .., k-1}
  vector<string> cvorovi = sviCvorovi(n-1, k);

  // za svaki cvor pamtimo koja grana je na redu za obilazak
  unordered_map<string, int> tekucaGrana;
  // na pocetku nismo obisli nijednu granu
  for (const string& cvor : cvorovi)
    tekucaGrana[cvor] = 0;

  // Ojlerov put određen čvorovima kroz koje se prolazi
  vector<string> ojlerovPut;

  stack<string> tekuciPut;
  // krecemo, na primer, od cvora 00..00
  tekuciPut.push(string(n-1, '0'));
  while (!tekuciPut.empty()) {
    string tekuciCvor = tekuciPut.top();
    // ako postoji neobrađena grana iz tekućeg čvora
    if (tekucaGrana[tekuciCvor] < k) {
      // oznaka na toj grani
      char grana = '0' + tekucaGrana[tekuciCvor]++;
      // čvor do koga vodi ta grana
      string sledeciCvor = tekuciCvor.substr(1) + string(1, grana);
      tekuciPut.push(sledeciCvor);
    } else {
      ojlerovPut.push_back(tekuciCvor);
      tekuciPut.pop();
    }
  }

  // ispisujemo rezultujuci niz
  // (dobijen od poslednjih brojki na cvorovima kojima prolazi Ojlerov put)
  for (int i = 0; i < ojlerovPut.size() - 1; i++)
    cout << ojlerovPut[i].back();
  cout << endl;

  return 0;
}
```

2.8 Težinski grafovi

U problemima koji slede bavićemo se *težinskim grafovima* (engl. weighted graph), odnosno grafovima u kojima je svakoj grani pridružena njena *težina* (engl. weight). Umesto težina grane često ćemo koristiti i termin dužina grane i pod terminom *dužina puta* smatraćemo zbir dužina grana na tom putu (a ne broj grana na tom putu). Na slici 2.69 prikazan je jedan usmereni težinski graf. U ovom grafu dužina puta $(1, 2, 0)$ iznosi $5 + 1 = 6$, dok dužina puta $(1, 3, 2, 0)$ iznosi $1 + 2 + 1 = 4$.

U programskom jeziku C++ težinski graf možemo predstaviti ili matricom povezanosti u kojoj se umesto logičkih vrednosti čuvaju težine grana (uz neku specijalnu numeričku vrednost koja označava da čvorovi nisu povezani) ili listama povezanosti, gde se u svakom elementu liste povezanosti čuva indeks krajnjeg čvora



Slika 2.69: Usmereni težinski graf.

grane i težina grane. Ako težinski graf predstavljamo listama povezanosti i ako su dužine grana celobrojne, graf možemo deklarisati na sledeći način:

```
typedef int Cvor;
typedef int Tezina;
vector<vector<pair<Cvor, Tezina>>> listaSuseda(n);
```

Novu granu $(cvorOd, cvorDo)$ težine t dodajemo u graf na sledeći način:

```
listaSuseda[cvorOd].emplace_back(cvorDo, t);
```

Podsetimo se da se primenom funkcije `emplace_back` najpre konstruiše uređeni par čvora i njegove težine, a zatim taj uređeni par dodaje na kraj vektora.

Primer 2.8.1

Usmereni težinski graf sa slike 2.69 zadajemo listama povezanosti na sledeći način:

```
vector<vector<pair<Cvor, Tezina>>> listaSuseda
    {{},  {{2,5}, {3,1}},  {{0,1}},  {{0,4}, {2,2}}};
```

Ako je graf neusmeren, možemo ga smatrati usmerenim, pri čemu svakoj njegovoj neusmerenoj grani odgovaraju dve usmerene grane iste dužine, u oba smera. Algoritmi koje ćemo razmatrati nad težinskim grafovima odnose se i na usmerene i na neusmerene grafove.

2.9 Najkraći putevi iz zadatog čvora

Postoji mnogo situacija u kojima se javlja potreba da se izračunaju najkraći putevi kroz graf. Na primer, graf može odgovarati auto–karti: čvorovi su gradovi, a dužine grana dužine direktnih puteva između gradova (ili vreme potrebno da se taj put pređe, ili troškovi da se izgradi). Zadatak je pronaći najkraći put (u smislu dužine, proteklog vremena ili potrebnog ulaganja) od jednog grada do drugog.

Algoritmi za određivanje najkraćih puteva igraju važnu ulogu i u rutiranju podataka kroz komunikacionu mrežu. Naime, određivanje najefikasnijih putanja (u kontekstu minimizovanja kašnjenja ili zagušenja) kroz mrežu od izvora do odredišta je od suštinskog značaja.

Algoritmi za računanje najkraćih puteva nalaze primenu i u optimizovanju lanca snabdevanja, odnosno u određivanju najkraćih puteva za transport robe od proizvođača do distributivnih centara, a zatim i krajnjih potrošača.

Iako je često potrebno da se nađe najkraći put od konkretnog početnog do konkretnog završnog čvora, nalaženje tog najkraćeg puta nije moguće bez nalaženja najkraćih puteva od početnog do mnogih drugih čvorova grafa. Zato se obično razmatra sledeći problem.

Problem

Za dati usmereni graf $G = (V, E)$ i jedan njegov čvor s pronaći najkraće puteve od čvora v do svih ostalih čvorova u grafu G.

Primer 2.9.1

Razmotrimo primer određivanja najkraćih puteva od čvora 1 do svih ostalih čvorova u grafu sa slike 2.69:

- *najkraći put do čvora 3 je direktna grana $(1, 3)$ dužine 1,*
- *najkraći put do čvora 2 je put $(1, 3, 2)$ ukupne dužine $1 + 2 = 3$, a ne direktna grana $(1, 2)$ dužine 5,*
- *najkraći put do čvora 0 je put $(1, 3, 2, 0)$ ukupne dužine $1 + 2 + 1 = 4$.*

2.9.1 Aciklički grafovi

Pretpostavimo najpre da je graf $G = (V, E)$ aciklički. U tom slučaju problem je lakši i njegovo rešenje pomoći će nam da problem rešimo i u opštem slučaju. Težine grana mogu biti proizvoljne (pozitivne, nula, pa čak i negativne).

Na primer, ako je graf drvo i tražimo najkraće puteve od korena do svih ostalih čvorova, tada do svakog čvora postoji jedinstven put i on je ujedno i najkraći. Dužine tih puteva se mogu odrediti bilo rekurzivno (tako što se za svaki čvor rekurzivno odredi najkraći put od korena do njegovog roditelja, koji se zatim produžava granom od roditelja do čvora), bilo iterativno (tako što će se najkraći put od svakog čvora produžiti granama do njegovih naslednika). Primetimo da se u osnovi zapravo krije algoritam zasnovan na dinamičkom programiranju (koji je u slučaju rekurzivne implementacije odozgo-naniže, a u slučaju iterativne odozdo-naviše).

I za acikličke grafove koje nisu drveta do rešenja se može doći na sličan način.

2.9.1.1 Rekurzivni pristup

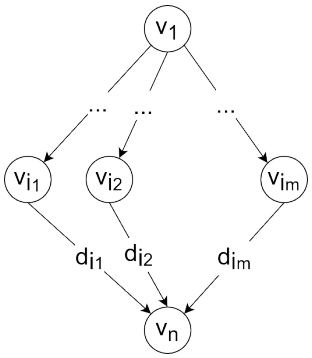
Neka je $|V| = n$. Pošto je graf aciklički, možemo da iskoristimo topološko sortiranje grafa i da čvorove u topološkom redosledu označimo sa $v_1, v_2, \ldots, v_n$. Ako je redni broj čvora s od koga treba odrediti najkraće puteve u topološkom redosledu jednak k (tj. ako je $s \equiv v_k$), onda se čvorovi sa rednim brojevima manjim od k ne moraju ni razmatrati, jer ne postoji način da se iz čvora s do njih dođe. Zato, jednostavnosti radi, pretpostavimo da je redni broj čvora s od koga treba izračunati najkraće puteve u topološkom redosledu 1, tj. da je $s \equiv v_1$. Ako dodatno pretpostavimo da osim čvora s u grafu nema drugih čvorova čiji je ulazni stepen nula, tada možemo biti sigurni da će do svakog čvora postojati put od polaznog (u suprotnom, do nekih čvorova neće postojati put, pa za njih možemo pretpostaviti da je dužina najkraćeg puta jednaka $+\infty$). Redosled čvorova dobijen topološkim sortiranjem je pogodan za primenu indukcije, tj. otkriva jednostavno rekurzivno rešenje.

Izlaz iz rekurzije nastupa kada se obrađuje čvor v_0 i najkraći put od tog čvora do njega samog je dužine 0.

Posmatrajmo poslednji čvor u topološkom redosledu čvorova, odnosno čvor v_n. Najkraće puteve od čvora s do svih ostalih čvorova, sem do v_n, možemo da odredimo rekurzivno. Označimo dužinu najkraćeg puta od čvora s do proizvoljnog čvora v_i sa $v_i.SP$ (engl. shortest path), a dužinu grane (x, y) između proizvoljna dva čvora x i y sa $d(x, y)$. Kako bismo odredili vrednost $v_n.SP$, dovoljno je da proverimo samo one čvorove v_i iz kojih postoji grana do čvora v_n (slika 2.70).

Pošto se prema pretpostavci najkraći putevi do svih ostalih čvorova već znaju, vrednost $v_n.SP$ biće jednaka minimumu zbira $v_i.SP + d(v_i, v_n)$, po svim čvorovima v_i iz kojih vodi grana do čvora v_n:

$$v_n.SP = \min_{(v_i, v_n) \in E} \{v_i.SP + d(v_i, v_n)\}.$$



Slika 2.70: Računanje najkraćeg puta od čvora $s \equiv v_1$ do poslednjeg čvora v_n u topološkom poretku.

Da li je time problem rešen? Najkraći putevi do čvorova $v_1, \ldots v_{n-1}$ su određeni bez razmatranja čvora v_n. Važno pitanje je da li dodavanje čvora v_n u skup čvorova do kojih smo odredili najkraće puteve može da skrati najkraći put od v_1 do nekog od njih, tj. da li je moguće da je najkraći put preko v_n kraći od najkraćeg puta koji ne vodi preko čvora v_n. Međutim, pošto je v_n poslednji čvor u topološkom redosledu, ni jedan drugi čvor nije dostižan iz v_n, pa se dužine ostalih najkraćih puteva ne menjaju. Dakle, uklanjanje čvora v_n iz grafa, pronalaženje najkraćih puteva u preostalom grafu i vraćanje čvora v_n nazad u graf su osnovni delovi algoritma.

Iz ovog razmatranja direktno sledi odgovarajući rekurzivni algoritam. Primetimo da on može biti veoma neefikasan, usled ponovljenih rekurzivnih poziva (jer iz čvora može voditi više grana). Ovo se, naravno, optimizuje nekim oblikom dinamičkog programiranja.

2.9.1.2 Induktivni pristup: istovremeno topološko sortiranje i određivanje najkraćih puteva

U prethodno opisanom postupku je pre početka traženja najkraćih puteva potrebno izvrši topološko sortiranje grafa. Takođe, potrebno je za svaki čvor odrediti grane koje vode do njega, što nije efikasna operacija kada se koriste liste povezanosti (grane su usmerene od predaka ka potomcima u topološkom redosledu, a ne obratno). Pokušajmo sada da unapredimo algoritam za traženje najkraćih puteva u acikličkom grafu, tako da se algoritam za topološko sortiranje obavlja istovremeno sa pronalaženjem najkraćih puteva. Drugim rečima, cilj je objediniti dva prolaza kroz graf (jedan za topološko sortiranje i drugi za nalaženje najkraćih puteva) u jedan, pri čemu ćemo sve vreme izračunavanja vršiti induktivno, od predaka ka potomcima u topološkom redosledu.

Razmotrimo način na koji se algoritam rekurzivno izvršava (posle nalaženja topološkog redosleda). Prvi korak je poziv rekurzivne procedure za čvor v_n. Procedura zatim poziva rekurzivno samu sebe, sve dok se ne dođe do čvora $s \equiv v_1$. U tom trenutku se dužina najkraćeg puta od čvora s do čvora s postavlja na 0, i rekurzija počinje da se "razmotava". Razmatra se čvor v_2; dužina najkraćeg puta do njega izjednačuje se sa dužinom grane (v_1, v_2), ako ona postoji; u protivnom, ne postoji put od v_1 do v_2. Sledeći korak je provera čvora v_3. U čvor v_3 ulaze najviše dve grane — od čvorova v_1 i/ili v_2, pa se upoređuju dužine odgovarajućih puteva. Umesto ovakvog izvršavanja rekurzije unazad, pokušaćemo da iste korake izvršimo unapred.

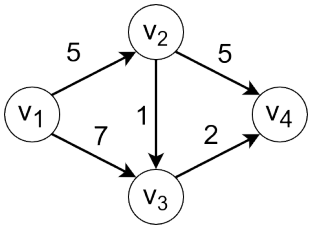
Indukcija se primenjuje prema rastućim rednim brojevima u topološkom redosledu počevši od $s \equiv v_1$. Ovaj redosled oslobađa nas potrebe da redne brojeve unapred znamo, pa ćemo biti u stanju da izvršavamo istovremeno i algoritam za određivanje topološkog uređenja i algoritam za određivanje najkraćih puteva od čvora s. Dakle, možemo razmotriti narednu induktivnu hipotezu.

Induktivna hipoteza: Znamo prvih $k-1$ čvorova u topološkom redosledu čvorova i dužine najkraćih puteva od čvora s do njih.

Potrebno je da odredimo čvor v_k sa rednim brojem k. To može da bude proizvoljni čvor u koji ne vodi ni jedna grana iz čvora sa rednim brojem većim od k (tj. bilo koji čvor stepena 0 kada se izbace sve grane iz prethodnih čvorova). Da bismo pronašli najkraći put do v_k, moramo da proverimo sve grane koje vode

u njega. Topološki redosled garantuje da sve takve grane polaze iz čvorova sa manjim rednim brojevima. Prema induktivnoj hipotezi ti čvorovi su poznati, kao i dužine najkraćih puteva do njih. Za svaku granu (v_i, v_k) znamo dužinu $v_i.SP$ najkraćeg puta od s do v_i, pa je dužina najkraćeg puta od s do v_k preko v_i jednaka $v_i.SP + d(v_i, v_k)$. Pored toga, kao i ranije, ne moramo da vodimo računa o eventualnim promenama najkraćih puteva ka čvorovima sa manjim rednim brojevima od k, jer se do njih ne može doći iz čvora v_k.

Da ne bismo za svaki čvor određivali iz kojih čvorova postoji grana ka njemu, jer to nije efikasna operacija u reprezentaciji grafa listama povezanosti, tokom izvršavanja algoritma možemo pamtiti dužine $v.\overline{SP}$ trenutno poznatih najkraćih puteva do svih čvorova v, uključujući i one sa većim rednim brojem od tekućeg čvora v_k. Prilikom razmatranja čvora v_k potrebno je jedino razmotriti grane (v_k, v_j) koje polaze iz njega, za čvor v_j proveriti da li je vrednost $v_k.\overline{SP} + d(v_k, v_j)$ manja od $v_j.\overline{SP}$ i ako jeste ažurirati vrednost $v_j.\overline{SP}$. Dakle umesto da granu grafa obrađujemo "unazad", odnosno u trenutku kada obrađujemo njen završni čvor, mi je obrađujemo "unapred", u trenutku kada obrađujemo njen polazni čvor. Primetimo da se te grane već obrađuju tokom Kanovog algoritma prilikom izbacivanja čvora v_k, kada se smanjuju ulazni stepeni čvorova v_j, pa ažuriranje najkraćih rastojanja možemo uraditi u istoj petlji. Pošto su pre obrade čvora v_k već obrađene sve grane do v_k iz prethodnih čvorova v_i i pošto je svaka od njih obrađivana u trenutku kada je već bilo poznato najkraće rastojanje do v_i, prilikom obrade čvora v_k niz najkraćih rastojanja sadrži najkraće rastojanje do v_k, tj. važi $v_k.SP = v_k.\overline{SP}$. Naime, umesto da se algoritam određivanja minimuma pokrene u trenutku obrade čvora v_k, on se malo po malo izvršavao tokom obrade prethodnih čvorova v_j.



Slika 2.71: Računanje najkraćeg puta u acikličkom grafu: čvorovi su označeni redom koji odgovara njihovoj poziciji u topološkom poretku.

Primer 2.9.2

Razmotrimo izvršavanje algoritma na primeru acikličkog grafa sa slike 2.71. Redosled čvorova u topološkom sortiranju grafa je v_1, v_2, v_3, v_4. Neka je zadatak odrediti najkraće puteve iz čvora $s \equiv v_1$.

- *Na početku postavljamo vrednost $v_1.\overline{SP}$ najkraćeg puta do čvora v_1 na 0 (i to proglašavamo konačnom vrednošću $v_1.SP$ najkraćeg puta do čvora v_1), a vrednosti najkraćih puteva $v_j.\overline{SP}$ do svih ostalih čvorova v_j na $+\infty$.*

  Razmatramo grane koje polaze iz čvora v_1: to su (v_1, v_2) i (v_1, v_3) i ažuriramo najkraće puteve do čvora v_2 i do čvora v_3 na $v_2.\overline{SP} = \min\{+\infty, 0 + 5\} = 5$, a $v_3.\overline{SP} = \min\{+\infty, 0 + 7\} = 7$.

- *Drugi čvor u topološkom redosledu je čvor v_2: tekuću vrednost $v_2.\overline{SP}$ najkraćeg puta do njega proglašavamo za konačnu vrednost najkraćeg puta, odnosno proglašavamo $v_2.SP = 5$.*

  Zatim razmatramo grane koje izlaze iz čvora v_2: to su grane (v_2, v_3) i (v_2, v_4) i ažuriramo najkraće puteve do čvorova v_3 i v_4: $v_3.\overline{SP} = \min\{7, 5 + 1\} = 6$ i $v_4.\overline{SP} = \min\{+\infty, 5 + 5\} = 10$.

- *Treći po redu čvor u topološkom poretku je čvor v_3 te tekuću vrednost $v_3.\overline{SP}$ najkraćeg puta do njega proglašavamo za konačnu, odnosno proglašavamo $v_3.SP = 6$.*

  Razmatramo jedinu granu (v_3, v_4) koja polazi iz čvora v_3 i vršimo ažuriranje vrednosti $v_4.\overline{SP}$ najkraćeg puta do čvora v_4: $v_4.\overline{SP} = \min\{10, 6 + 2\} = 8$.

- *U ovom trenutku proglašavamo kao konačnu i dužinu $v_4.\overline{SP}$ najkraćeg puta do čvora v_4, odnosno proglašavamo $v_4.SP = 8$, čime se izvršavanje algoritma završava.*

Iz prethodnog razmatranja sledi naredna teorema o korektnosti ovog algoritma.

Teorema 2.9.1

Neka je $G = (V, E)$ usmeren aciklički težinski graf i v_1 neki njegov čvor čiji je ulazni stepen 0. Ako se najkraća rastojanja inicijalizuju tako da je $v_1.\overline{SP} = 0$ i $v_i.\overline{SP} = +\infty$, za svako $1 < i \leq n = |V|$, tada se nakon sprovođenja algoritma dobija topološki redosled čvorova $v_1, \ldots, v_n$ i za svaki čvor $v_i \in V$ važi $v_i.\overline{SP} = v_i.SP$, tj. niz sadrži najkraće rastojanje od čvora v_1 do svih ostalih čvorova u grafu.

Do sada smo razmatrali izračunavanje dužina najkraćih puteva, a ne i računanje samih najkraćih puteva. Primetimo da smo najkraće puteve određivali jedan po jedan: svaki novi najkraći put se dobija produživanjem nekog prethodno određenog najkraćeg puta poslednjom granom na putu. Grane kojima se vrši produživanje formiraju tzv. *drvo najkraćih puteva* (engl. shortest-path tree). Kako bismo mogli da rekonstruišemo najkraće puteve do svakog čvora, za svaki čvor pamtimo njegovog prethodnika (roditelja) na najkraćem putu od čvora s. Jedino čvor $s \equiv v_1$ neće imati roditelja na najkraćem putu (ako čvor s nije prvi u topološkom redosledu, roditelje neće imati ni čvorovi koji su ispred njega u topološkom redosledu). Na taj način možemo jednostavno da rekonstruišemo same najkraće puteve.

Razmotrimo implementaciju algoritma: pretpostavićemo da je potrebno odrediti najkraće puteve od čvora 0 do svih ostalih čvorova. Pretpostavimo, jednostavnosti radi, i da je polazni čvor jedini čvor ulaznog stepena 0. Vrednosti $+\infty$ ćemo u implementaciji predstavljati najvećim celim brojem koji može da se zapiše u tipu `int`. To olakšava implementaciju jer se izbegavaju granaja, ali treba biti obazriv da se taj broj tokom algoritma ne sabira sa drugim brojevima (inače bi došlo do prekoračenja).

```
// umesto vrednosti beskonacno koja oznacava da put nije pronađen
// koristimo najveći broj koji se može zapisati u tipu težine grana
const Tezina INF = numeric_limits<Tezina>::max();

// funkcija koja stampa put od polaznog cvora v do datog cvora
// kroz grane drveta najkracih puteva
void odstampajPutDoCvora(Cvor cvor, vector<Cvor> roditelji) {
  // ako postoji put do roditeljskog cvora, stampamo ga
  if (roditelji[cvor] != -1)
     odstampajPutDoCvora(roditelji[cvor], roditelji);
  // stampamo tekuci cvor
  cout << " " << cvor;
}

// funkcija koja stampa najkraci put do datog cvora i njegovu duzinu
void odstampajNajkraciPut(Cvor cvor, vector<Cvor> roditelji,
                          vector<Tezina> minRastojanja) {
  cout << "Najkraci put do cvora " << cvor << " je:";
  odstampajPutDoCvora(cvor, roditelji);
  cout << " i duzine je " << minRastojanja[cvor] << endl;
}

// funkcija koja racuna najkrace puteve od cvora 0 u aciklickom grafu
void aciklicki_najkraci_putevi() {
  int brojCvorova = listaSuseda.size();
  // niz koji za svaki cvor cuva njegov ulazni stepen
```

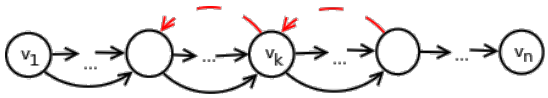
```
  vector<int> ulazniStepeni(brojCvorova, 0);
  // niz koji za svaki cvor cuva duzinu
  // trenutno poznatog najkraceg puta do njega
  vector<Tezina> minRastojanja(brojCvorova, INF);
  // niz koji za svaki cvor cuva roditelja u najkracem putu
  vector<Cvor> roditelji(brojCvorova, -1);

  // najkraci put od cvora 0 do njega samog postavljamo na 0
  minRastojanja[0] = 0;

  // inicijalizujemo niz ulaznih stepena cvorova
  // potreban za izvršavanje Kanovog algoritma
  for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
    for (const auto& [sused, tezina] : listaSuseda[cvor])
      ulazniStepeni[sused]++;

  // red koji cuva cvorove ulaznog stepena nula
  queue<Cvor> cvoroviStepenaNula;

  // pretpostavljeno je da je polazni cvor 0 jedini stepena 0
  cvoroviStepenaNula.push(0);

  while(!cvoroviStepenaNula.empty()) {
    // cvor sa pocetka reda je naredni u topoloskom redosledu
    Cvor cvor = cvoroviStepenaNula.front();
    cvoroviStepenaNula.pop();
    // do njega je odredjen najkraci put i stampamo ga
    odstampajNajkraciPut(cvor, roditelji, minRastojanja);

    for (const auto& [sused, tezina] : listaSuseda[cvor]) {
      // ukoliko je do nekog cvora kraci put preko upravo razmatranog cvora
      // vrsimo azuriranje najkraceg rastojanja do tog cvora
      if (minRastojanja[cvor] + tezina < minRastojanja[sused]) {
         minRastojanja[sused] = minRastojanja[cvor] + tezina;
         // azuriramo koji je roditeljski cvor na najkracem putu
         roditelji[sused] = cvor;
      }
      ulazniStepeni[sused]--;
      // ukoliko je stepen nekog od suseda pao na 0, dodajemo ga u red
      if (ulazniStepeni[sused] == 0)
         cvoroviStepenaNula.push(sused);
    }
  }
}
```

U algoritmu se svaka grana razmatra po jednom u toku inicijalizacije ulaznih stepena čvorova, i po jednom u trenutku kad se njen polazni čvor uklanja iz reda. Pristup redu zahteva konstantno vreme. Svaki čvor se razmatra tačno jednom. Prema tome, vremenska složenost algoritma za određivanje najkraćih puteva u acikličkom grafu je u najgorem slučaju $O(|V| + |E|)$.

2.9.2 Grafovi sa nenegativnim granama: Dajkstrin algoritam

Kad graf nije aciklički, ne postoji topološki redosled, pa se razmatrani algoritam ne može direktno primeniti. Međutim, osnovne ideje se mogu iskoristiti i u opštem slučaju, doduše, samo kod grafova kod kojih su dužine grana nenegativne. Naime, jednostavnost algoritma za određivanje najkraćih puteva iz zadatog čvora u acikličkom grafu posledica je sledeće osobine topološkog redosleda: ako je v_k čvor sa rednim brojem k u topološkom poretku, onda:

- ne postoje putevi od čvorova sa rednim brojevima većim od k do čvora v_k,
- dualno, ne postoje putevi od čvora v_k do čvorova sa rednim brojevima manjim od k (slika 2.72).



Slika 2.72: Kada čvorove poređamo slevo nadesno u topološkom redosledu, u acikličkom grafu mogu postojati samo grane sleva nadesno, koje vode od čvorova sa manjim rednim brojem ka čvorovima sa većim rednim brojem (označene crnom bojom), dok grane koje vode zdesna ulevo, od čvorova sa većim rednim brojem ka čvorovima sa manjim rednim brojem, ne mogu postojati (označene crvenom bojom).

Ova osobina topološkog redosleda omogućuje nam da organizujemo redosled rekurzivnih tj. induktivnih izračunavanja tj. da nađemo najkraći put od čvora $s \equiv v_1$ do čvora v_k, ne vodeći računa o čvorovima koji su posle v_k u topološkom redosledu (jer od njih sigurno ne postoje putevi do v_k). Može li se nekako definisati redosled čvorova proizvoljnog grafa (koji nije nužno aciklički) koji bi omogućio nešto slično?

Ključna ideja je *razmatrati čvorove grafa redom prema dužinama najkraćih puteva od čvora s do njih*. Naime, ako se čvorovi urede u niz $v_1, v_2, \ldots, v_n$ neopadajuće na osnovu dužina tih puteva, i ako u grafu nema grana negativne težine, važe veoma slična svojstva kao kod topološkog poretka:

- najkraći put do čvora v_k ne prolazi preko čvorova sa rednim brojevima većim od k,
- dualno, najkraći put do nekog čvora sa rednim brojem manjim od k ne prolazi preko čvora v_k.

Dakle, prilikom određivanja najkraćeg puta do v_k mogu se zanemariti putevi koji vode preko čvorova sa rednim brojevima većim od k, tj. treba razmotriti samo puteve preko čvorova sa rednih brojeva manjih od k, koji mogu mogu biti određeni rekurzivno tj. induktivno. Ako bismo unapred mogli da odredimo ovaj redosled čvorova, rekurzivni algoritam bi mogao da funkcioniše po istom principu kao i u slučaju acikličkih grafova. Izbacili bismo čvor v_n koji je najudaljeniji od čvora $s \equiv v_1$, izračunali rastojanja od s do svih ostalih čvorova, a zatim bismo analizirali grane (v_i, v_n) i rastojanje do v_n odredili kao $\min_{(v_i,v_n)\in E}\{v_i.SP + d(v_i, v_n)\}$. Zahvaljujući prethodnim uslovima znamo da grane koje vode iz v_n, nazad ka prethodnim čvorovima ne treba razmatrati, tj. da određivanje najkraćeg rastojanja do v_n ne može ni na koji način ažurirati najkraća rastojanja prethodnih čvorova (ona će biti ispravno određena i pre razmatranja čvora v_n). Izlaz iz ovog rekurzivnog postupka je slučaj $v_1 \equiv s$ (jer je najbliži čvor početnom čvoru s upravo čvor s) kada je najkraće rastojanje jednako 0.

Međutim, algoritam moramo organizovati drugačije, jer se, naravno, dužine puteva, pa ni ovaj poredak na početku ne znaju, već se izračunavaju u toku izvršavanja algoritma.

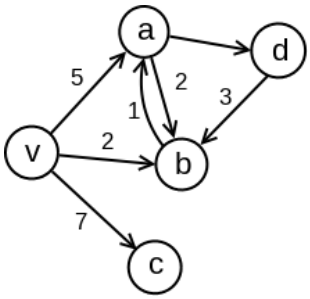
- Pošto su sve grane nenegativne dužine, najkraće rastojanje od čvora s do čvora s je 0 i čvor s je prvi u ovom redosledu. Dakle, ponovo važi $s \equiv v_1$.
- Nakon toga proveravamo sve grane koje izlaze iz čvora v_1. Neka je (v_1, v_2) najkraća među njima. Pošto su, po pretpostavci, sve dužine grana nenegativne, najkraći put od čvora s do čvora v_2 je grana (v_1, v_2). Dužine svih drugih puteva od čvora s do čvora v_2 su veće ili jednake od dužine ove grane. Čvor v_2 je, dodatno, najbliži od svih čvorova čvoru s. Prema tome, znamo najkraći put do čvora najbližeg čvoru s.
- Pokušajmo da napravimo sledeći korak. Kako možemo da pronađemo čvor do koga vodi naredni najkraći put, odnosno da odredimo čvor v_3 koji je treći najbliži čvoru s ($v_1 \equiv s$ i v_2 su prva dva

najbliža). Jedini putevi koje treba uzeti u obzir su preostale grane iz čvora v_1 (koje ne vode do v_2) ili putevi koji se sastoje od dve grane, tako da je prva grana (v_1, v_2), a druga je neka grana koja polazi iz čvora v_2. Ostali putevi mogu biti samo duži od ovih. Čvor v_3, dakle, određujemo kao onaj čvor v_i različit od v_1 i v_2 za koji je najmanja vrednost $\min\{d(v_1, v_i), d(v_1, v_2) + d(v_2, v_i)\}$.

- Sličan se postupak koristi i za određivanje narednih čvorova.

Primer 2.9.3

Razmotrimo graf sa slike 2.73 i problem traženja najkraćih puteva od čvora v.



Slika 2.73: Usmereni težinski graf koji sadrži cikluse.

- *Prvi najbliži čvor čvoru v je on sâm.*
- *Drugi najbliži čvor je jedan od suseda čvora v i to onaj do koga vodi najkraća grana – to je čvor b i najkraći put do b je dužine 2.*
- *Treći najbliži čvor čvoru v je ili neki drugi sused čvora v ili neki čvor do koga vodi grana iz čvora b (kao prvog najbližeg čvora čvoru s) – to je čvor a, do koga vodi put (v, b, a) dužine $2 + 1 = 3$.*
- *Četvrti najbliži čvor čvoru v je čvor do koga se stiže granom iz nekog od čvorova v, b ili a – to je čvor d, do koga vodi put (s, b, a, d) dužine $2 + 1 + 1 = 4$.*
- *Peti najbliži čvor je čvor c do koga se stiže granom iz v, b, a ili d (jedina grana koja iz tih čvorova vodi ka nekom novom čvoru je grana iz v do c dužine 7).*

Može se formulisati sledeća induktivna hipoteza.

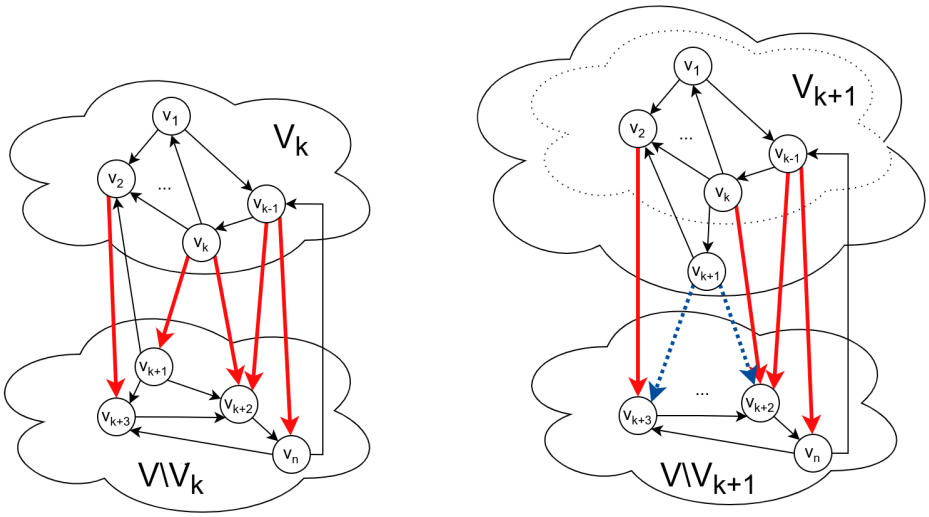
Induktivna hipoteza: Za zadati graf $G = (V, E)$ i njegov čvor s, umemo da odredimo k čvorova najbližih čvoru s, kao i dužine najkraćih puteva do njih.

Zapazimo da se indukcija primenjuje po broju čvorova do kojih su dužine najkraćih puteva već izračunate, a ne po veličini grafa. Pored toga, pretpostavlja se da su to čvorovi najbliži čvoru s, i da umemo da ih odredimo. Čvor s je najbliži sam sebi i na rastojanju je 0, pa je baza (slučaj $k = 1$) rešena. Kad k dostigne vrednost $|V|$, rešen je kompletan problem. Ako se ne traže rastojanja do svih čvorova, nego samo do nekog istaknutog čvora v', postupak se može prekinuti i ranije, čim se izračuna rastojanje do čvora v'.

Označimo sa $V_k = \{v_1, \ldots, v_k\}$ skup koji se sastoji od k najbližih čvorova čvoru s (uključujući i $s \equiv v_1$). Problem je pronaći čvor v_{k+1} koji je najbliži čvoru s među čvorovima van V_k, i pronaći najkraći put od s do v_{k+1}.

Najkraći put od s do v_{k+1} može da sadrži samo čvorove iz V_k. On ne može da sadrži neki čvor v van V_k, jer bi u tom slučaju čvor v bio bliži čvoru s od čvora v_{k+1}, ako su težine grana pozitivne. Ako eventualno postoje grane težine nula, za čvor v_{k+1} ćemo odabrati neki od onih čvorova van V_k do kojih najkraći put prolazi samo kroz čvorove iz V_k (bar jedan takav čvor mora da postoji). Prema tome, da bismo pronašli čvor v_{k+1}, dovoljno je da proverimo grane koje spajaju čvorove iz V_k sa čvorovima koji nisu u V_k; sve druge grane se za sada mogu ignorisati. Neka je (v_i, v) proizvoljna grana grafa G takva da je $v_i \in V_k$ i $v \notin V_k$. Takva grana određuje put od s do v koji se sastoji od najkraćeg puta od s do v_i (koji je prema

induktivnoj hipotezi već poznat) i grane (v_i, v). Dovoljno je uporediti dužine svih takvih puteva i izabrati najkraći među njima (slika 2.74, levo).



Slika 2.74: Nalaženje sledećeg najbližeg čvora zadatom čvoru $s \equiv v_1$. Levo je prikazana neoptimizovana varijanta u kojoj se u svakom koraku razmatraju sve grane između skupova V_k i $V \setminus V_k$. Desno je prikazana optimizovana varijanta. U trenutku proširivanja skupa V_k čvorom v_{k+1} dovoljno je razmotriti samo grane koje idu iz v_{k+1} ka čvorovima van V_k.

Algoritam određen ovom induktivnom hipotezom izvršava se na sledeći način. U svakoj iteraciji u skup V_k se dodaje novi čvor. U prvoj je to čvor s. U svakoj narednoj iteraciji je to onaj čvor $v \notin V_k$ koji je najbliži čvoru s, tj. za koji je najmanja vrednost izraza:

$$\min \{v_i.SP + d(v_i, v) | \, v_i \in V_k\}. \tag{2.2}$$

Iz već iznetih razloga, tako odabrani čvor v je zaista $(k+1)$-vi (sledeći) najbliži čvor čvoru s, pa ga možemo obeležiti sa v_{k+1}. Formulom (2.2) određena je i stvarna dužina najkraćeg puta od čvora s do čvora v_{k+1} (ona se neće dalje smanjivati). Prema tome, dodavanje čvora v_{k+1} skupu V_k proširuje induktivnu hipotezu.

Algoritam je sada u potpunosti preciziran, ali mu se efikasnost može poboljšati. Osnovni korak algoritma je pronalaženje sledećeg najbližeg čvora v_{k+1} čvoru s. To se ostvaruje izračunavanjem dužine najkraćeg puta (preko čvorova iz V_k) do svakog čvora $v \notin V_k$ prema formuli (2.2). Možemo da, kao i u slučaju acikličkog grafa, u nizu pamtimo dužine $v.\overline{SP}$ trenutno poznatih najkraćih puteva do svih čvorova $v \in V$, a da im pri proširivanju skupa V_k ažuriramo vrednosti. Za čvorove van V_k to će biti najkraći putevi koji vode isključivo preko čvorova u V_k, tj. preko svih grana koje polaze iz čvorova u V_k, a ne i globalno najkraći, jer možda do njih postoji i neki kraći put koji vodi preko čvorova van V_k, pa uključuje i neke do sada nepregledane grane.

Vrednosti $v.\overline{SP}$ za čvorove van V_k se menjaju prilikom proširivanja skupa V_k (koji figuriše u formuli (2.2)). Ipak, realno je očekivati da proširivanje skupa V_k čvorom v_{k+1} ne utiče na vrednost rastojanja mnogih čvorova $v \notin V_k$. Zaista, mogu se promeniti samo vrednosti za one čvorove v do kojih postoji grana (v_{k+1}, v), tj. samo one vrednosti koje odgovaraju putevima kroz novododati čvor v_{k+1}. Prema tome, prilikom dodavanja čvora v_{k+1} u skup V_k treba proveriti sve grane od čvora v_{k+1} ka čvorovima van V_k (slika 2.74, desno). Za svaku takvu granu (v_{k+1}, v) upoređujemo zbir $v_{k+1}.\overline{SP} + d(v_{k+1}, v)$ sa trenutnom vrednošću $v.\overline{SP}$, i po potrebi ažuriramo (smanjujemo) vrednost $v.\overline{SP}$. Primetimo da smo isti ovaj način ažuriranja koristili i za izračunavanje najkraćih puteva u acikličkom grafu. Svaka iteracija algoritma, dakle, obuhvata nalaženje čvora $v \equiv v_{k+1}$ van V_k sa najmanjom trenutnom vrednošću $v.\overline{SP}$, i eventualno ažuriranje vrednosti $v.\overline{SP}$ za

njegove susede koji nisu u V_k. Pošto su prilikom izbora tog čvora v_{k+1} već obrađene sve grane koje vode do njega od čvorova v_i koji su bliži čvoru s od njega i pošto se na najkraćem putu do njega ne može naći nijedan drugi čvor van V_k, tekuća vrednost $v_{k+1}.\overline{SP}$ se poklapa sa stvarnom vrednošću $v_{k+1}.SP$ najkraćeg rastojanja do njega.

Ovaj algoritam dobio je naziv *Dajkstrin algoritam* (engl. Dijkstra's algorithm) po Edzgaru Dajkstri koji ga je osmislio 1956. godine. On pripada grupi pohlepnih algoritama, jer se u svakom koraku bira lokalno optimalno rešenje – najbliži čvor i nakon obrade trenutno najbližeg čvora rastojanje do njega se više nikada ne razmatra.

Najkraće puteve od čvora s do svih ostalih čvorova našli smo tako što smo pronalazili jedan po jedan najkraći put. Svaki novi put je određen jednom granom, koja produžuje prethodno poznati najkraći put do novog čvora. Sve te grane – produženi puteva formiraju drvo sa korenom u čvoru s. Ovo drvo naziva se *drvo najkraćih puteva* i važno je za rešavanje mnogih problema sa putevima. Primetimo da ako bi dužine svih grana u grafu bile međusobno jednake, onda bi drvo najkraćih puteva u stvari bilo BFS drvo sa korenom u čvoru s.

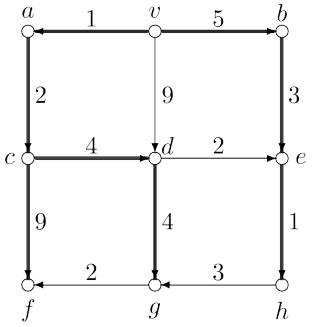
Prethodnim razmatranjem smo dokazali narednu teoremu.

Teorema 2.9.2

Neka je dat težinski graf $G = (V, E)$ u kom su težine svih grana nenegativne i neka je $s \in V$. Neka su u startu čvorovima pridružene vrednosti tako da je $s.\overline{SP} = 0$ i $v.\overline{SP} = +\infty$, za svako $v \in V$, $v \neq s$. Nakon što se ponovi n koraka Dajkstrinog algoritma, tj. nakon što se svaki čvor grafa ubaci u skup V_k, za svaki čvor $v \in V$ biće ispravno određen najkraći put od s, tj. važiće $v.SP = v.\overline{SP}$.

Primer 2.9.4

Izvršavanje Dajkstrinog algoritma za nalaženje najkraćih puteva od čvora v ilustrovano je na slici 2.75.



	v	a	b	c	d	e	f	g	h
a	0	1	5	∞	9	∞	∞	∞	∞
c	0	**1**	5	3	9	∞	∞	∞	∞
b	0	**1**	5	**3**	7	∞	12	∞	∞
d	0	**1**	**5**	**3**	7	8	12	∞	∞
e	0	**1**	**5**	**3**	**7**	8	12	11	∞
h	0	**1**	**5**	**3**	**7**	**8**	12	11	9
g	0	**1**	**5**	**3**	**7**	**8**	12	11	**9**
f	0	**1**	**5**	**3**	**7**	**8**	12	**11**	**9**

Slika 2.75: Primer izvršavanja Dajkstrinog algoritma.

U primeru na slici 2.75 podebljane su grane koje pripadaju drvetu najkraćih puteva od čvora $s \equiv v$, sa korenom u čvoru v. Primetimo da, recimo, najkraći put do čvora h dobijamo tako što granom (e, h) produžimo prethodno pronađeni najkraći put do čvora e: put (v, b, e).

Tabela registruje promene u nizu $\overline{SP}$ u kom se čuvaju dužine najkraćih puteva od čvora v do svih ostalih čvorova preko čvorova koji pripadaju skupu V_k. Elementi skupa V_k (pored čvora v) su oni čije su dužine najkraćih puteva ispisane podebljano (to su konačne dužine najkraćih puteva).

- *Prva vrsta tabele odnosi se samo na puteve koji se sastoje od jedne grane koja polazi iz čvora v. Bira se najkraći od tih puteva (grana) i u ovom slučaju on vodi ka čvoru a.*
- *Druga vrsta tabele pokazuje ažuriranje dužina najkraćih puteva uključujući sada sve puteve koji se sastoje od jedne grane koja polazi iz čvora v ili od dve grane preko čvora a. Drugi najkraći put od čvora v vodi do čvora c.*

- *U svakom koraku algoritma bira se novi, naredni najbliži čvor čvoru v, dodaje se u skup V_k i u narednoj vrsti tabele se prikazuju dužine trenutnih najkraćih puteva od čvora v do svih čvorova.*

Na osnovu tabele moguće je rekonstruisati i same najkraće puteve. Na primer, ako želimo da saznamo koji je najkraći put od čvora v do čvora h dužine 9, razmatramo kolonu koja odgovara čvoru h i tražimo poslednju promenu vrednosti u toj koloni – ona odgovara vrsti označenoj čvorom e i to znači da je čvor e roditelj čvora h u drvetu najkraćih puteva. Zatim, na osnovu tabele određujemo roditelja čvora e, gledajući kolonu tabele koja odgovara čvoru e i utvrđivanjem na kom čvoru se desila poslednja promena vrednosti u ovoj tabeli – to je čvor b te je on roditelj čvora e, dok je roditelj čvora b polazni čvor v. Konačno, najkraći put rekonstruišemo čitajući dobijeni niz roditeljskih čvorova unazad i kao najkraći put od čvora v do čvora h dobijamo put (v, b, e, h). Naravno, umesto da pamtimo celu tabelu, efikasnije je da roditeljske čvorove održavamo u zasebnom nizu.

U Dajkstrinom algoritmu potrebno je da $O(|V|)$ puta pronalazimo dužine najkraćih puteva do čvorova van skupa V_k i da često ažuriramo dužine puteva. Struktura podataka pogodna za efikasno nalaženje minimalnih elemenata je red sa prioritetom. Pretpostavićemo da je on realizovan kao min-hip (mada može biti realizovan i pomoću balansiranog binarnog drveta, kao uređen skup). Pošto je potrebno da pronađemo čvor do koga je dužina puta najmanja, sve čvorove v van skupa V_k čuvamo u redu sa prioritetom, sa ključevima $v.\overline{SP}$ jednakim dužinama trenutno najkraćih puteva od čvora s do njih. Na početku su sve dužine puteva osim one do čvora s jednake $+\infty$, pa redosled elemenata u hipu nije bitan, sem što je čvor s na vrhu. Nalaženje čvora v_{k+1} koji je među čvorovima van V_k najbliži čvoru s je jednostavno: on se uzima sa vrha hipa. Posle toga se za svaku granu (v_{k+1}, v) proverava da li je v van V_k i da li najkraći put do čvora v_{k+1} produžen granom (v_{k+1}, v) skraćuje trenutno poznati najkraći put do čvora v.

Međutim, kad se promeni dužina puta $v.\overline{SP}$ do nekog čvora v, potrebno je da se promeni i položaj čvora v u hipu. Prema tome, potrebno je na odgovarajući način popravljati hip. S obzirom na to da se putevi do čvora mogu samo skraćivati, to znači da se vrednost ključa u hipu može samo smanjiti i eventualno pri tom postati manja od vrednosti ključa svog roditelja (pošto je prethodna vrednost elementa bila manja od vrednosti njegove dece u hipu, to će važiti i za smanjenu vrednost ključa). Operacija smanjivanja vrednosti ključa u min-hipu nije direktno podržana u standardnoj biblioteci jezika C++, ali se odgovarajuća popravka hipa može ručno implementirati razmenom vrednosti elementa i njegovog roditelja, sve dok uslov hipa ne bude zadovoljen. Ova operacija se dakle može izvesti algoritmom složenosti $O(\log n)$, gde je n broj elemenata u hipu. Međutim, problem sa ovim popravkama je u tome što hip kao struktura podataka ne podržava efikasno pronalaženje zadatog elementa. Naime, traženje elementa u hipu je operacija linearne vremenske složenosti u odnosu na broj elemenata u hipu. U ručnoj implementaciji bismo mogli u posebnom nizu čuvati položaj svakog čvora u hipu. Ipak, postoji jednostavan način da se ovo izbegne i da se upotrebi samo osnovni interfejs reda sa prioritetom (dodavanje elemenata, pronalaženje i brisanje najmanjeg elementa). Naime, umesto da se vrednost rastojanja do nekog čvora u hipu zameni novom (manjom) vrednošću, vrši se umetanje novog čvora sa istom oznakom čvora i novom (manjom) vrednošću rastojanja (ova tehnika se naziva *tehnikom lenjog brisanja* (engl. lazy deletion technique)). S obzirom na to da je nova vrednost rastojanja manja od stare, novi čvor će sigurno biti skinut iz hipa pre starog. Ostaje problem što će se u nekom kasnijem trenutku iz hipa skinuti i stari podatak o čvoru v sa većom vrednošću najkraćeg puta. Taj podatak treba da se zanemari, pa je prilikom uzimanja elementa sa vrha hipa potrebno prosto preskočiti čvorove koji su ranije bili obrađeni tj.
koji se već nalaze u skupu V_k.

Ostaje bojazan da ovakva implementacija može da poveća vremenska složenost algoritma. Ukoliko bismo vršili popravke ključeva u hipu, u hipu bismo čuvali u svakom trenutku najviše $O(|V|)$ elemenata. U implementaciji koja čuva kopije čvorova prilikom obrade svake (usmerene) grane može se dodati maksimalno jedan novi element u hip. Dakle ukupan broj elemenata u hipu biće sigurno manji ili jednak od $O(|V|+|E|)$, te će svaka od operacija umetanja elementa u hip i brisanja minimalnog elementa iz hipa biti složenosti $O(\log(|V| + |E|))$. S obzirom na to da je $|E| \leq |V|^2$, i stoga $\log |E| \leq 2 \cdot \log |V|$, složenosti $O(\log |E|)$ i $O(\log |V|)$ se asimptotski ne razlikuju, te će operacije nad hipom koji čuva kopije čvorova

biti iste asimptotske složenosti kao i u slučaju kada nema kopija. Naravno, memorijsko zauzeće može biti veće.

Dajkstrin algoritam za nalaženje najkraćih puteva od datog čvora prikazan je u nastavku.

```
// Dajkstrin algoritam za odredjivanje najkracih puteva
// iz datog polaznog cvora start do svih cvorova u grafu
void najkraciPuteviDajkstra(int start) {
   int brojCvorova = listaSuseda.size();
  // da li je cvoru odredjeno najkrace rastojanje
  vector<bool> resen(brojCvorova, false);
  // duzina trenutno poznatog najkraceg puta do cvora
  vector<Tezina> minRastojanja(brojCvorova, INF);
  // roditeljski cvor svakog cvora u drvetu najkracih puteva
  vector<Cvor> roditelji(brojCvorova, -1);

  // uredjen par koji čine rastojanje do cvora i broj cvora;
  // redosled elemenata mora biti ovakav zbog operacije poredjenja parova
  typedef pair<Tezina, Cvor> RastojanjeCvora;
  // min-hip u koji smestamo poznata rastojanja do svih cvorova
  priority_queue<RastojanjeCvora,
                 vector<RastojanjeCvora>,
                 greater<RastojanjeCvora>> rastojanja;

  // ubacujemo polazni cvor u hip i postavljamo rastojanje do njega na 0
  rastojanja.emplace(0, start);
  minRastojanja[start] = 0;

  // broj cvorova do kojih je konacno odredjeno najkrace rastojanje
  int brojResenih = 0;

  // dok se ne odredi najkrace rastojanje do svih cvorova
  while (brojResenih < brojCvorova) {
    // izdvajamo naredni najblizi cvor
    auto [rastojanje, cvor] = rastojanja.top();
    rastojanja.pop();

    // ako je taj cvor ranije resen, preskacemo ga
    // ovo se moze desiti zbog lenjog azuriranja hipa
    if (resen[cvor])
       continue;

    // inace pamtimo da smo sada odredili najkrace rastojanje do njega
    brojResenih++;
    resen[cvor] = true;

    // stampamo informaciju o najkracem putu do tog cvora;
    // najkraci putevi se ispisuju u redosledu njihovog "otkrivanja"
    odstampajNajkraciPut(cvor, roditelji, minRastojanja);

    // prolazimo kroz sve grane iz tekuceg cvora
    for (const auto& [sused, tezina] : listaSuseda[cvor]) {
```

```
      // koje vode ka susedima kojima jos nije odredjeno najkrace rastojanje
      if (!resen[sused]) {
         // ukoliko je put kroz tekuci cvor do suseda kraci od poznatog
         // najkraceg puta, azuriramo vrednost najkraceg puta do suseda
         // i vrednost njegovog roditeljskog cvora
         if (minRastojanja[cvor] + tezina < minRastojanja[sused]) {
            minRastojanja[sused] = minRastojanja[cvor] + tezina;
            roditelji[sused] = cvor;
            // ubacujemo element u hip;
            // ako je postojala prethodna vrednost, ne brisemo je:
            // nova vrednost ce se naci u hipu iznad stare
            rastojanja.emplace(minRastojanja[sused], sused);
          }
       }
    }
  }
}
```

Kao što smo već pomenuli, operacije umetanja i brisanja iz hipa su složenosti $O(\log |V|)$. Možemo imati najviše $O(|V|+|E|)$ umetanja u hip (u varijanti sa čuvanjem kopija čvorova) i najviše $O(|V|+|E|)$ brisanja iz hipa. Prema tome, vremenska složenost algoritma je $O((|V| + |E|) \log |V|)$. Zapaža se da je Dajkstrin algoritam sporiji nego algoritam koji određuje najkraće puteve od zadatog čvora u acikličkom grafu. Ukoliko bi se umesto binarnog hipa koristio Fibonačijev hip, vremenska složenost Dajkstrinog algoritma bila bi jednaka $O(|E| + |V| \log |V|)$. Napomenimo da je kod gustih grafova, kod kojih je broj grana asimptotski jednak $|V|^2$, umesto hipa efikasnije koristiti običan niz i minimum tražiti linearnom pretragom (složenost je tada $O(|V|^2)$, dok je složenost varijante sa hipom $O(|V|^2 \log |V|)$.

Tip pretrage koji se javlja u Dajkstrinom algoritmu se naziva i *pretraga sa prioritetom* — svakom čvoru dodeljuje se prioritet (u ovom slučaju trenutno najmanje poznato rastojanje od polaznog čvora), pa se čvorovi obilaze redosledom koji je određen prioritetom. Kad se završi razmatranje tekućeg čvora, razmatraju se sve njemu susedne grane i pritom se mogu promeniti prioriteti nekih drugih čvorova. Način izvođenja tih promena je detalj po kome se jedna pretraga sa prioritetom razlikuje od druge. Pretraga sa prioritetom je karakteristična za težinske grafove i složenija je od obične pretrage za faktor $\log |V|$ ako se koristi hip.

2.9.3 Grafovi sa negativnim granama

Razmotrimo sada problem određivanja najkraćih puteva iz zadatog čvora u opštem slučaju, kada težine grana mogu biti i negativne. Prirodno je zapitati se da li negativne težine grana imaju ikakvog smisla. Naime, u kontekstu rastojanja na mapama nemaju, međutim pokazuje se da težinski grafovi čije težine mogu biti i pozitivne i negativne nisu neuobičajeni u modelovanju nekih problema iz realnog sveta.

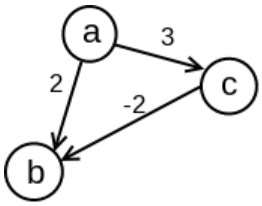
- Na ovaj način možemo modelovati problem održavanja salda na računu: čvorovi grafa odgovaraju računima, grane transakcijama nad računima, pri čemu izvršavanje transakcije može dovesti do zarade – koju ćemo modelovati granom pozitivne težine ili do gubitka – koji ćemo modelovati granom negativne težine.
- Razmotrimo, takođe, problem putovanja iz jednog grada u drugi, pri čemu je moguće na nekom od delova puta pokupiti nekog putnika i zaraditi određenu količinu novca na tom delu puta. Cilj nam je da dođemo iz jednog grada u drugi sa najviše novca na kraju. Ovaj problem možemo modelovati u vidu problema traženja najkraćeg puta u grafu čije težine grana mogu biti pozitivne i odgovaraju troškovima putovanja između ta dva grada (ako na tom delu puta putujemo sami) ili negativne (ako na nekom delu puta pokupimo jednog ili više putnika i od njih zaradimo više novca nego što su troškovi tog dela puta).

- Nalaženje puta sa najmanjim proizvodom težina grana se može svesti na nalaženje puta sa najmanjim zbirom težina grana u grafu u kom je težina svake grane zamenjena njenim logaritmom (jer je logaritam monotona funkcija, a logaritam proizvoda jednak je zbiru logaritama). Primetimo da nakon logaritmovanja neke težine grana mogu postati negativne, pa se ponovo susrećemo sa problemom najkraćeg puta u grafu sa negativnim težinama grana.

Ukoliko graf sadrži neku granu negativne težine, Dajkstrin algoritam ne garantuje da ćemo do svakog čvora odrediti zaista najkraće puteve.

Primer 2.9.5

Razmotrimo problem određivanja najkraćih puteva iz čvora a do svih ostalih čvorova u grafu prikazanom na slici 2.76. Ako bismo primenili Dajkstrin algoritam, on bi najpre odredio najkraći put do čvora b (kao čvora koji je od svih čvorova do kojih postoji grana iz a najbliži čvoru a) i proglasio bi da je on dužine 2, a nakon toga bi odredio najkraći put do čvora c kao drugog najbližeg čvora čvoru a i proglasio bi da je dužina najkraćeg puta do njega 3 čime bi se se izvršavanje algoritma završilo. Ipak, jasno je da najkraći put od čvora a do čvora b vodi preko čvora c i dužine je 1, međutim Dajkstrin algoritam ne bi razmatrao ovaj put.

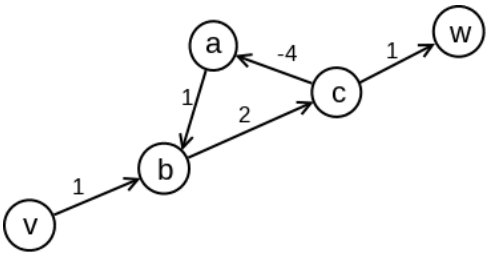


Slika 2.76: Primer grafa koji sadrži granu negativne težine i za koji Dajkstrin algoritam pokrenut iz čvora a ne računa dobro najkraća rastojanja.

U nastavku teksta ćemo često pretpostavljati da graf ne sadrži ciklus negativne težine. Ovaj uslov je prirodan jer inače do nekih čvorova ne mora da postoji najkraći put (jer put možemo uvek dodatno skratiti još jednim prolaskom kroz ciklus). Možemo definisati da je najkraće rastojanje do takvih čvorova $-\infty$ (iako je put uvek konačan, pa mu dužina ne može biti $-\infty$).

Primer 2.9.6

Razmotrimo graf sa slike 2.77: najkraći put od čvora v do čvora w se ne može odrediti jer svaki prolazak kroz ciklus (b, c, a, b) smanjuje dužinu puta za 1 (dužina najkraćeg puta je $-\infty$).



Slika 2.77: Primer grafa koji sadrži ciklus negativne težine.

2.9.3.1 Opšti algoritam zasnovan na relaksaciji grana

Osnovni korak u algoritmu zasnovanom na relaksacijama grana je isti kao i u Dajkstrinom algoritmu i u algoritmu za određivanje najkraćih puteva u acikličkim grafovima, a to je tzv. *relaksacija grane* (u, v), odnosno provera da li je vrednost $v.\overline{SP}$ trenutno najkraćeg utvrđenog rastojanja od početnog čvora s do

čvora v veća od vrednosti $u.\overline{SP} + d(u, v)$; ako je to tačno, kažemo da je relaksacija uspešna, ažurira se vrednost $v.\overline{SP}$ i pamti da najkraći put do čvora v vodi kroz čvor u.

Zapravo, svi algoritmi za rešavanje ovog problema mogu se podvesti pod sledeću opštu shemu prikazanu u algoritmu 7.

Algoritam 7 Opšti algoritam relaksacije grana

1: $s.\overline{SP} = 0$, $v.\overline{SP} = +\infty$, za sve $v \neq s$
2: **while** postoji grana (u, v) takva da je $v.\overline{SP} > u.\overline{SP} + d(u, v)$ **do**
3: $\quad v.\overline{SP} = u.\overline{SP} + d(u, v)$

Ako u grafu postoji negativan ciklus, ovaj opšti algoritam se ne zaustavlja, a ako nema negativnih ciklusa algoritam se zaustavlja i ispravno izračunava najkraća rastojanja do svih čvorova (što ćemo uskoro i dokazati). Primetimo da nije preciziran redosled obrade grana i konkretni algoritmi se razlikuju po tome kojim redom i koliko puta obrađuju grane (na primer, Dajkstrin algoritam obrađuje čvorove u redosledu neopadajućeg rastojanja i redom relaksira grane koje iz njih izlaze). Obično je redosled obrade takav da se pod određenim pretpostavkama o grafu unapred zna da je dostignuto konačno rešenje, tj. takav da se ne vrši efektivna provera da li postoje grane koje se mogu relaksirati.

Dokažimo zajednička svojstva algoritama zasnovanih na relaksaciji grana. Iako neke od narednih lema važe i u slučaju kada u grafu postoji negativni ciklus, to nam neće biti potrebno u analizi algoritama, tako da ćemo u svim narednim lemama razmatrati samo grafove bez negativnih ciklusa (osim ako je naglašeno drugačije).

Lema 2.9.1 **[Nejednakost trougla]**

Neka je dat težinski graf $G = (V, E)$ bez negativnih ciklusa i neka $x.SP$ označava najkraće rastojanje od čvora $s \in V$ do čvora $x \in V$. Za svaku granu $(u, v) \in E$ važi $u.SP + d(u, v) \geq v.SP$.

Dokaz. Ne može da važi $v.SP > u.SP + d(u, v)$, jer bi tada postojao put od s do u na koji bi se mogla nadovezati grana (u, v), čime bi se dobio put od s do v koji bi bio kraći od najkraćeg puta od s do v, što je kontradikcija. □

Jasno je da se relaksacijom samo može smanjiti procena rastojanja $v.\overline{SP}$. Naredna lema tvrdi da ta procena nikada ne može biti manja od stvarne vrednosti najkraćeg rastojanja $v.SP$.

Lema 2.9.2 **[Donja granica procene rastojanja]**

Neka je dat težinski graf $G = (V, E)$ bez negativnih ciklusa i neka za svaki čvor $v \in V$, $v.SP$ označava najkraće rastojanje od čvora s do čvora v, a $v.\overline{SP}$ tekuću procenu tog rastojanja tokom izvršavanja postupka relaksacije grana (u proizvoljnom redosledu). Tokom čitavog tog procesa za svaki čvor v važi $v.\overline{SP} \geq v.SP$.

Dokaz. Tvrđenje dokazujemo indukcijom po broju relaksiranih grana. Bazu indukcije čini slučaj inicijalizacije. Za sve čvorove $v \neq s$ mora da važi $v.SP \leq v.\overline{SP} = +\infty$. Pošto graf nema negativni ciklus koji sadrži s, važi $s.\overline{SP} = s.SP = 0$.

Pretpostavimo da tvrđenje važi nakon k relaksiranih grana i dokažimo da važi i nakon relaksiranja dodatne grane (u, v). Za sve čvorove $x \neq v$ ne menja se vrednost $x.\overline{SP}$, pa na osnovu induktivne hipoteze važi $x.\overline{SP} \geq x.SP$. Za čvor v nakon relaksacije važi $v.\overline{SP} = u.\overline{SP} + d(u, v)$. Na osnovu induktivne hipoteze važi $u.\overline{SP} \geq u.SP$, pa je $v.\overline{SP} \geq u.SP + d(u, v)$, međutim, na osnovu nejednakosti trougla (leme 2.9.1) važi $u.SP + d(u, v) \geq v.SP$, pa važi $v.\overline{SP} \geq v.SP$. □

Jasno je da kada jednom procena $v.\overline{SP}$ dostigne vrednost $v.SP$ ona sve vreme nadalje ostaje jednaka toj vrednosti (jer relaksacija samo može smanjiti vrednost $v.\overline{SP}$, a na osnovu prethodne leme znamo da ona ne može biti manja od $v.SP$.

Naredna lema nam garantuje da će nakon relaksacija grana iz čvora u, za koji je već izračunata vrednost najkraćeg rastojanja od početnog čvora s (tj. važi $u.\overline{SP} = u.SP$), biti ispravno izračunate vrednosti najkraćeg rastojanja onih njegovih suseda v za koje se najkraći put završava granom (u, v).

Lema 2.9.3 **[Relaksacija poslednje grane na najkraćem putu]**

Neka je dat težinski graf $G = (V, E)$ bez negativnih ciklusa i neka je (u, v) poslednja grana na najkraćem putu od $s \in V$ do $v \in V$. Nakon što se u trenutku kada važi $u.\overline{SP} = u.SP$ ta grana relaksira ili se ustanovi da se ne može relaksirati, važi $v.\overline{SP} = v.SP$.

Dokaz. Pošto je (u, v) poslednja grana na najkraćem putu do v, važi da je $v.SP = u.SP + d(u, v)$. Zato je $v.\overline{SP} \geq v.SP = u.SP + d(u, v) = u.\overline{SP} + d(u, v)$. Nakon što se grane (u, v) relaksira ili se ustanovi da to nije moguće, važi da je vrednost $v.\overline{SP}$ minimum stare vrednosti $v.\overline{SP}$ i vrednosti $u.\overline{SP} + d(u, v)$. Zato važi $v.\overline{SP} = u.\overline{SP} + d(u, v) = u.SP + d(u, v) = v.SP$. □

Naredna lema nam garantuje da ako ne postoji grana koja se može relaksirati, tada su sva najkraća rastojanja ispravno izračunata.

Lema 2.9.4 **[Uslov za postojanje grane koja se može relaksirati]**

Neka je dat težinski graf $G = (V, E)$ bez negativnih ciklusa i neka je za neki čvor $v \in V$ važi $v.\overline{SP} \neq v.SP$. Tada postoji grana $e \in E$ koja se može relaksirati.

Dokaz. Neka je $(s \equiv v_1, v_2, \ldots, v_n = v)$ najkraći put od početnog čvora do čvora v i neka je v_i prvi čvor na tom putu za koji je $v_i.\overline{SP} \neq v_i.SP$. Pošto za čvor $v_1 \equiv s$ sve vreme važi $s.\overline{SP} = s.SP = 0$, mora važiti $i > 0$. Ako se grana (v_{i-1}, v_i) ne bi mogla relaksirati, na osnovu leme 2.9.3 moralo bi da važi $v_i.\overline{SP} = v_i.SP$, što je kontradikcija. Dakle, postoji grana koja se može relaksirati. □

Sada je jednostavno dokazati korektnost opšte sheme relaksacije grana.

Teorema 2.9.3 **[Zaustavljanje i korektnost opšteg algoritma]**

Neka je dat težinski graf $G = (V, E)$ bez negativnih ciklusa. Tada se algoritam 7 zaustavlja i nakon zaustavljanja za svaki čvor $v \in V$ važi $v.\overline{SP} = v.SP$.

Dokaz. Pošto se sve vrednosti $v.\overline{SP}$ smanjuju tokom izvršavanja algoritma (u svakoj relaksaciji neke grane strogo se smanji vrednost $v.\overline{SP}$ za neki čvor v), a na osnovu leme 2.9.2 su one ograničene odozdo, u jednom trenutku sve vrednosti $v.\overline{SP}$ će dostići svoje granice $v.SP$. To je sasvim ako su u pitanju grane celobrojne težine, jer je zbir svih vrednosti $v.\overline{SP}$ opadajući niz celih brojeva koji je odozdo ograničen zbirom svih vrednosti $v.SP$, pa mora biti konačan. Ako su grane racionalni brojevi, teorijski bi mogao da postoji beskonačno dugačak opadajući niz racionalnih brojeva koji je odozdo ograničen. Međutim, množenjem svih težina nekim pogodno odabranim stepenom broja 10 može se lako postići da težine svih grana postanu celobrojne, bez suštinske izmene problema, tako da je i u tom slučaju jasno da se algoritam mora zaustaviti nakon konačnog broja koraka.

Ako bi se tada mogla relaksirati neka grana (u, v), važilo bi da je $v.\overline{SP} > u.\overline{SP} + d(u, v)$, tj. $v.SP > u.SP + d(u, v)$, što je nemoguće na osnovu nejednakosti trougla (lema 2.9.1). Dakle, tada nema grana koje se mogu relaksirati, pa se algoritam zaustavlja.

Pošto se algoritam zaustavi, ne postoji grana koja se može relaksirati. Ako bi za neki čvor $v \in V$ važilo $v.\overline{SP} \neq v.SP$, na osnovu leme 2.9.4 postojala bi grana koja bi se mogla relaksirati, što je kontradikcija. □

2.9.3.2 Belman-Fordov algoritam

Ukoliko graf ima negativne težine grana, ali ne sadrži ciklus negativne težine, najkraći putevi iz datog čvora s mogu se odrediti *Belman-Fordovim algoritmom*[7]. Važi i više od toga: ako u grafu postoji ciklus negativne težine, Belman-Fordov algoritam to detektuje.

U izvođenju ovog algoritma biće nam značajna naredna lema, koja garantuje da će nakon što se *redom* relaksira jedna po jedna grana najkraćeg puta do nekog čvora uspešno biti izračunato najkraće rastojanje do tog čvora (kao i do svih ostalih čvorova na tom putu).

Lema 2.9.5 **[Relaksacija najkraćeg puta]**

Neka je dat težinski graf $G = (V, E)$ bez negativnih ciklusa i neka je $p = (s \equiv v_1, \dots, v_j)$ najkraći put od s do v_j. Proizvoljni niz relaksacija grana koji uključuje i redom relaksacije grana (v_1, v_2), (v_2, v_3), ..., (v_{j-1}, v_j) dovodi do toga da važi $v_j.\overline{SP} = v_j.SP$. Pre, posle i između ovih relaksacija je dopušteno vršiti bilo koje druge relaksacije grana.

Dokaz. Tvrđenje dokazujemo indukcijom po čvorovima u nizu p tj. dokazujemo da za svako $1 \leq k \leq j$ nakon relaksacija grana (v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k) važi $v_k.\overline{SP} = v_k.SP$.

Baza indukcije je čvor s. Pošto graf nema negativnih ciklusa, važi $s.SP = 0$. Vrednost $s.\overline{SP}$ mora sve vreme biti 0, jer je, znamo na osnovu leme 2.9.2, uvek veća ili jednaka od $s.SP$, tokom relaksacije se samo može smanjiti, a inicijalizuje se na 0.

Pretpostavimo da smo redom relaksirali grane (v_1, v_2), (v_2, v_3), ..., (v_{k-2}, v_{k-1}) i da je $v_{k-1}.\overline{SP} = v_{k-1}.SP$. U nekom trenutku nakon toga se pokušava relaksacija grane (v_{k-1}, v_k). Na osnovu leme 2.9.3 sledi da nakon toga sigurno važi $v_k.\overline{SP} = v_k.SP$. Na osnovu leme 2.9.2 važi $v_k.\overline{SP} \geq v_k.SP$, pa se ova vrednost ne može dalje smanjivati i neće se menjati. □

Dakle, dovoljno je da naš algoritam obezbedi da se *sve grane svih najkraćih puteva relaksiraju redom od prve do poslednje*. Belman-Fordov algoritam to postiže tako što ukupno $|V| - 1$ puta proizvoljnim redosledom izvršava relaksaciju svih grana u grafu.

Algoritam 8 Belman-Fordov algoritam

```
1: s.SP̄ = 0, v.SP̄ = +∞, za sve v ≠ s
2: for i = 1..|V| − 1 do
3:     for all (u, v) ∈ E do
4:         if v.SP̄ > u.SP̄ + d(u, v) then
5:             v.SP̄ = u.SP̄ + d(u, v)
```

Tvrdimo da su, ako graf ne sadrži negativan ciklus, nakon toga korektno izračunati najkraći putevi od čvora v do svih čvorova u grafu. Za čvorove koji su nedostižni iz početnog čvora, dužina najkraćeg puta ostaje $+\infty$. Ako graf sadrži negativan ciklus, nakon svih ovih relaksacija i dalje postoje grane koje se mogu relaksirati (provera postojanja takve grane je ujedno i način da se proveri da li postoji negativan ciklus, tj. da li su pronađena rastojanja korektna).

Dokažimo ovo i formalno.

Lema 2.9.6

Neka je dat težinski graf $G = (V, E)$ bez negativnih ciklusa. Kada se Belman-Fordov algoritam (algoritam 8) primeni na graf G, za svaki čvor v važi $v.\overline{SP} = v.SP$.

[7] Ričard Belman i Lester Ford su prvi objavili ovaj algoritam 1958. i 1959. godine, međutim, smatra se da je njega ipak prvi objavio Alfonso Simbel 1955. godine.

Dokaz. Ako je v dostižan iz s u grafu bez negativnih ciklusa mora da postoji prost put od s do v. U tom putu ne može da bude više od $|V|$ čvorova, pa ni više od $|V|-1$ grana. Pošto se u svakoj iteraciji relaksiraju sve grane, u prvoj iteraciji je sigurno relaksirana prva grana na tom putu, u drugoj druga itd. (naravno moguće je da se u nekoj iteraciji redom relaksira i više grana najkraćeg puta). Pošto se vrši $|V|-1$ iteracija, sve grane su redom relaksirane, pa na osnovu leme 2.9.5 važi $v.\overline{SP} = v.SP$.

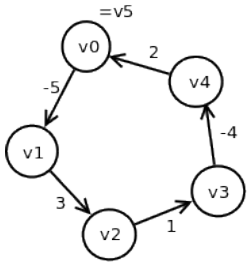
Ako v nije dostižan iz s tada je $v.\overline{SP} = v.SP = +\infty$. Zaista, vrednost $v.\overline{SP}$ se inicijalizuje na $+\infty$, a ne može da se smanji ispod stvarne vrednosti $v.SP$ koja je jednaka $+\infty$. □

Teorema 2.9.4 **[Zaustavljanje i korektnost Belman-Fordovog algoritma]**

Ako u grafu nema ciklusa negativne težine, Belman-Fordov algoritam se zaustavlja i za sve čvorove v grafa važi $v.\overline{SP} = v.SP$ i nijedna grana se ne može relaksirati. Ako u grafu postoji ciklus negativne težine, po završetku algoritma postoji bar jedna grana koja se može relaksirati.

Dokaz. Prvi deo tvrđenja direktno sledi iz leme 2.9.6. Za svaku granu (u,v) na osnovu nejednakosti trougla važi $v.\overline{SP} = v.SP \leq u.SP + d(u,v) = u.\overline{SP} + d(u,v)$, pa se grana (u,v) ne može relaksirati.

Pretpostavimo da G sadrži ciklus negativne težine $c = (v_0, v_1, \ldots, v_k)$, $v_k = v_0$, dostižan iz čvora s (slika 2.78).



Slika 2.78: Ciklus negativne težine.

Tada je:

$$\sum_{i=1}^{k} d(v_{i-1}, v_i) < 0 \tag{2.3}$$

Pretpostavimo suprotno, da se nijedna grana ne može relaksirati, tj. da za svaku granu $(u,v) \in E$ važi $v.\overline{SP} \leq u.\overline{SP} + d(u,v)$. Tada i za svaku granu ciklusa $(v_{i-1}, v_i) \in E, i = 1, \ldots, k$, važi: $v_i.\overline{SP} \leq v_{i-1}.\overline{SP} + d(v_{i-1}, v_i)$. Sabiranjem ovih nejednakosti za sve grane ciklusa dobija se:

$$\sum_{i=1}^{k} v_i.\overline{SP} \leq \sum_{i=1}^{k} v_{i-1}.\overline{SP} + \sum_{i=1}^{k} d(v_{i-1}, v_i)$$

Pošto je $v_0 = v_k$, važi:

$$\sum_{i=1}^{k} v_i.\overline{SP} = \sum_{i=1}^{k} v_{i-1}.\overline{SP}$$

te dalje važi:

$$\sum_{i=1}^{k} d(v_{i-1}, v_i) \geq 0$$

suprotno pretpostavci (2.3). □

Primetimo da se relaksacija grana ne mora vršiti istim redosledom u svakoj od iteracija – samo je bitno da se u svakoj iteraciji relaksiraju sve grane grafa. Time se postiže da se na svakom najkraćem putu sigurno relaksira po jedna dodatna grana.

Primetimo da se nakon izvršavanja i-te iteracije petlje, pronalaze najkraći putevi čiji broj grana ne prevazilazi vrednost i.

S obzirom da se često može desiti da se najkraća rastojanja izračunaju i pre sprovođenja svih iteracija, algoritam je moguće prekinuti ranije ako se desi da se u nekoj iteraciji ne uspe relaksacija nijedne grane.

Implementacija ovog algoritma je veoma jednostavna.

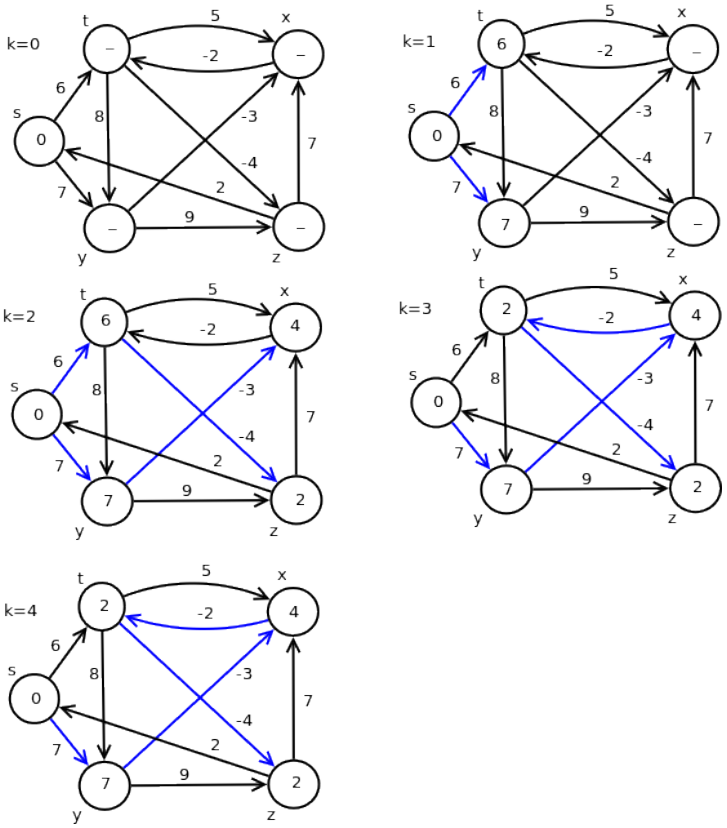
```
// funkcija koja koriscenjem Belman-Fordovog algoritma
// racuna najkrace puteve do svih cvorova
void najkraciPuteviBelmanFord(int start) {
  int brojCvorova = listaSuseda.size();
  // duzina trenutno poznatog najkraceg puta do cvora
  vector<Tezina> minRastojanja(brojCvorova, INF);
  // prethodnik cvora na najkracem putu
  vector<Cvor> roditelji(brojCvorova, -1);

  // postavljamo rastojanje do polaznog cvora
  minRastojanja[start] = 0;

  // |V|-1 put prolazimo kroz skup svih grana
  for (int k = 0; k < brojCvorova - 1; k++) {
    bool biloRelaksacija = false;
    for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
      for (const auto& [sused, tezina] : listaSuseda[cvor]) {
         // ukoliko je potrebno vrsimo relaksaciju grane
         if (minRastojanja[cvor] + tezina < minRastojanja[sused]) {
             minRastojanja[sused] = minRastojanja[cvor] + tezina;
             // i postavljamo koji cvor mu prethodi na najkracem putu
             roditelji[sused] = cvor;
             biloRelaksacija = true;
         }
      }
    if (!biloRelaksacija) break;
  }

  // ukoliko i dalje postoji put koji je moguce skratiti
  // onda graf sadrzi ciklus negativne duzine
  for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
    for (const auto& [sused, tezina] : listaSuseda[cvor])
      if (minRastojanja[cvor] + tezina < minRastojanja[sused]) {
         cout << "Graf sadrzi ciklus negativne duzine" << endl;
         return;
```

```
    }

  // stampamo najkrace puteve do svih cvorova u grafu
  for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
    odstampajNajkraciPut(cvor, roditelji, minRastojanja);
}
```

Složenost Belman-Fordovog algoritma iznosi $O(|V| \cdot |E|)$, jer spoljašnjom `for` petljom prolazimo $O(|V|)$ puta, a u svakoj iteraciji ove petlje prolazimo skupom svih grana u grafu.

Primer 2.9.7

Na slici 2.79 ilustrovano je izvršavanje Belman-Fordovog algoritma.



Slika 2.79: Primer izvršavanja Belman-Fordovog algoritma. Graf ima 5 čvorova, te se algoritam sastoji od 4 iteracije, za $k = 1, 2, 3, 4$. Prva slika odgovara inicijalizaciji, a naredne slike pojedinačnim prolazima kroz sve grane. Vrednosti trenutno najkraćih rastojanja prikazane su unutar čvorova, a grane plave boje vode od prethodnika čvorova na (trenutno) najkraćim putevima: ako je grana (u, v) plave boje, onda se do čvora v najkraćim putem dolazi preko čvora u. U svakom prolazu grane se relaksiraju u narednom redosledu: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$.

Napomenimo još to da se Belman-Fordov algoritam može unaprediti tako što se smanji broj relaksacija koje se razmatraju u svakoj iteraciji algoritma. Naime, ako se trenutno najkraće rastojanje do čvora v nije izmenilo od trenutka kada su poslednji put relaksirane grane koje polaze iz čvora v, onda nema potrebe ponovo vršiti relaksaciju grana koje polaze iz čvora v. Na ovaj način se broj grana koje treba relaksirati u svakoj iteraciji potencijalno smanjuje. Ovaj algoritam poznat je pod nazivom *brži algoritam najkraćeg puta* (engl. Shortest Path Faster Algorithm – SPFA).

Zadatak: Najmanje naporna staza

Mapa jednog predela je predstavljena pravougaonom matricom polja tako je za svako polje poznata nadmorska visina tog dela predela. Planinari tokom kretanja sa svakog polja mogu da prelaze samo na neko od najviše četiri njemu susednih polja i, pošto nose puno tereta, na putu ne mogu da savladaju velike visinske razlike. Težina nekog puta se definiše kao najveća apsolutna vrednost visinske razlike između neka dva susedna polja na tom putu. Napisati program koji određuje najmanju težinu nekog puta koji povezuje gornje levo i donje desno polje na mapi.

Opis ulaza

Sa standardnog ulaza se unose dimenzije matrice m i n, a zatim i matrica dimenzije $m \times n$ koja sadrži nadmorske visine (prirodne brojeve između 0 i 1000).

Opis izlaza

Na standardni izlaz ispisati najmanju težinu puta.

Primer 1

Ulaz

```
3 3
1 2 2
3 8 2
5 4 5
```

Izlaz

```
2
```

Objašnjenje

Najpovoljniji put je onaj gde planinari prelaze preko polja visine 1, 3, 5, 4, 5. Na tom putu savladavaju redom razlike od 2, 2, 1 i 1 metar, pa je težina puta jednaka 2.

Primer 2

Ulaz

```
5 5
1 2 1 1 1
1 2 1 2 1
1 2 1 2 1
1 2 1 2 1
1 1 1 2 1
```

Izlaz

```
0
```

Objašnjenje

Najlakši put je onaj gde se sve vreme kreću poljima čija je visina 1.

Rešenje

Dajkstrin algoritam

Zadatak se može rešiti jednostavnom modifikacijom Dajkstrinog algoritma za pronalaženje najkraćeg puta od gornjeg levog polja, pa do svih ostalih polja, uključujući i poslednje, donje desno polje. Jedina razlika u odnosu na uobičajeni Dajkstrin algoritam se u slučaju da je poznato najmajnje rastojanje r_{uv} od čvora u do čvora v (najveća razlika visina na najboljem putu od u do v) i rastojanje r_{vw} od čvora v do njemu susednog

čvora w (razlika visina), tada je najmanje rastojanje od čvora u do čvora v jednako $\max(r_{uv}, r_{vw})$. Tada se vrednost rastojanja ažurira na osnovu dodele $r_{uw} := \min(r_{uw}, \max(r_{uv}, r_{vw}))$.

"Težina" grane (u, v) je u ovom primeru definisana kao razlika visina dva čvora koje ta grana spaja. "Dužina" puta $(v_0, v_1, \ldots, v_k)$ je ovde definisana kao maksimalna razlika visina čvorova na putu. "Zbir" dužine dva puta jednak je njihovom maksimumu. Ova metrika zadovoljava sledeća svojstva:

- *Nenegativnost:* težina svake grane je nenegativna (jer se gleda apsolutna razlika visina).
- *Usmerenost:* "dužina" puta koji se sastoji od jedne usmerene grane jednaka je "težini" te grane (što direktno sledi iz definicija).
- *Nejednakost trougla:* Za svaka tri čvora u, v, w, "dužina" najkraćeg puta od u do w je manja ili jednaka "zbiru" "dužine" najkraćeg puta od u do v i "težine" grane v do w. Zaista, "dužina" puta od u do w preko čvora v je maksimum najveće razlike visina na putu od u do v i razlike visina čvorova v i w. Nemoguće je da je najmanja moguća razlika visina na putu od u do w veća od ovog broja (jer uvek možemo proći putem preko v).

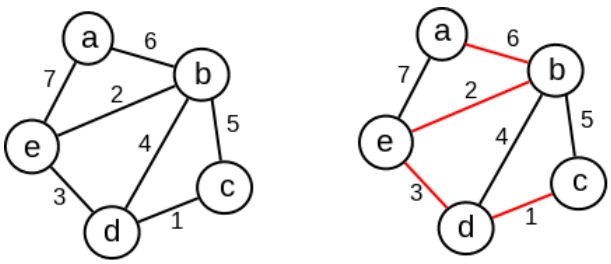
Ova svojstva su dovoljna da osiguraju korektnost Dajkstrinog algoritma.

```
int najmanjeNapornaStaza(vector<vector<int>>& visine) {
  // dimenzije matrice
  int m = visine.size(), n = visine[0].size();
  // rastojanje od početnog do svakog drugog čvora pamtimo u matrici
  vector<vector<int>> rastojanje(m, vector<int>(n, INF));
  // podatak da li je do čvora određeno rastojanje pamtimo u matrici
  vector<vector<bool>> resen(m, vector<bool>(n, false));
  // red sa prioritetom koji koji sadrži trojke na čijem je prvom mestu
  // dužina grane, a zatim čvor iz kog ta grana polazi
  // (njegove koordinate)
  priority_queue<tuple<int, int, int>,
                 vector<tuple<int, int, int>>,
                 greater<tuple<int, int, int>>> pq;
  // rastojanje do početnog čvora je 0
  rastojanje[0][0] = 0;

  // krećemo od početnog čvora
  pq.emplace(0, 0, 0);
  while (!pq.empty()) {
    // skidamo element sa vrha reda sa prioritetom
    auto [tezina, v, k] = pq.top();
    pq.pop();
    // ako je do čvora na poziciji (v, k) ranije određen najkraći put,
    // taj čvor preskačemo
    if (resen[v][k])
      continue;
    // pamtimo da smo upravo odredili najkraći put do čvora na
    // poziciji (v, k)
    resen[v][k] = true;

    // prolazimo kroz sve susede ovog čvora
    int dv[] = {-1, 1, 0, 0};
    int dk[] = {0, 0, -1, 1};
    for (int i = 0; i < 4; i++) {
      int vv = v + dv[i];
```

```
      int kk = k + dk[i];
      if (!(0 <= vv && vv < m && 0 <= kk && kk < n))
        continue;

      // do kojih je najkraće rastonja još nepoznato
      if (resen[vv][kk])
        continue;

      // težina grane od trenutnog do susednog čvora
      int razlika = abs(visine[vv][kk] - visine[v][k]);
      // ažuriramo rastojanje do suseda ako je manje od dosadašnjeg
      int novoRastojanje = max(rastojanje[v][k], razlika);
      if (novoRastojanje < rastojanje[vv][kk]) {
        rastojanje[vv][kk] = novoRastojanje;
        pq.emplace(rastojanje[vv][kk], vv, kk);
      }
    }
  }
  // vraćamo najkraće rastojanje do ciljnog polja
  return rastojanje[m-1][n-1];
}
```

2.10 Minimalno povezujuće drvo

Razmotrimo sistem računara koje treba povezati optičkim kablovima tako da postoji veza između svaka dva računara. Poznati su troškovi postavljanja kabla između svaka dva računara. Cilj je projektovati mrežu optičkih kablova tako da ukupna cena mreže bude minimalna. Sistem računara može biti predstavljen neusmerenim grafom čiji čvorovi odgovaraju računarima, a grane – potencijalnim vezama između računara, sa odgovarajućim (pozitivnim) cenama. Tada se problem svodi na pronalaženje povezanog podgrafa (sa granama koje odgovaraju postavljenim optičkim kablovima) koji sadrži sve čvorove, takav da mu ukupna suma težina grana bude minimalna (slika 2.80). Nije teško videti da taj podgraf mora da bude drvo. Naime, ako bi podgraf imao ciklus, onda bi se iz ciklusa mogla ukloniti proizvoljna grana, čime bi se dobio podgraf koji je i dalje povezan, a ima manju ukupnu težinu. Traženi podgraf zove se *minimalno povezujuće (razapinjuće) drvo* (MCST, skraćenica od engl. minimum–cost spanning tree) i ima mnogo primena. Sličan prethodnom problemu je i problem povezivanja N gradova autoputevima, tako da postoji put između svaka dva grada, a da ukupna cena izgradnje autoputa bude minimalna moguća. Minimalno povezujuće drvo ima primenu i u konstrukciji približnih algoritama, na primer, za rešavanje problema trgovačkog putnika.



Slika 2.80: Neusmereni povezani težinski graf i njegovo minimalno povezujuće drvo.

Dakle, problem kojim se bavimo je konstrukcija efikasnog algoritma za nalaženje minimalnog povezujućeg drveta u datom neusmerenom težinskom grafu.

Problem

Za zadati neusmereni povezani težinski graf $G = (V, E)$ konstruisati neko povezujuće drvo T minimalne težine.

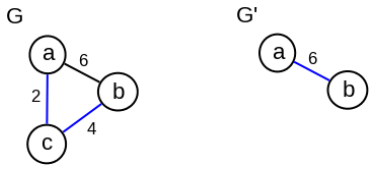
Ako su sve grane grafa međusobno različite težine, minimalno povezujuće drvo je jedinstveno određeno (što ne mora biti slučaj kada postoje grane iste težine).

Postoji više različitih algoritama za određivanje minimalnog povezujućeg drveta datog grafa: mi ćemo razmotriti dva – Primov i Kraskelov algoritam.

2.10.1 Indukcija po broju čvorova i grana

Prilikom konstrukcije grafovskih algoritama prirodno je pokušati indukcijom po broju čvorova ili po broju grana. Na primer, pronalaženje najkraćih puteva iz jednog čvora u acikličnom grafu kao i Dajkstrin algoritam se mogu opisati tako što se iz grafa izbaci jedan čvor (poslednji u odgovarajućem poretku), rekurzivno reši smanjeni problem i na kraju doda izbačeni čvor. Naravno, implementacija teče iterativno dodajući jedan po jedan čvor, pri čemu se čvorovi moraju obrađivati u odgovarajućem redosledu.

Nažalost, takav pristup se ne može jednostavno prilagoditi rešavanju problema minimalnog povezujućeg drveta. Naime, u opštem slučaju ne postoji neki prirodan redosled u kom bi se čvorovi obrađivali. Izbačeni čvor može biti artikulaciona tačka i razbiti graf na nepovezane komponente tako da se ne dobije problem istog oblika. Možemo odrediti minimalnu povezujuću šumu za posebne komponente, ali to komplikuje algoritam. Čak i kada se izbacivanjem čvora dobije povezan graf, nije jasno kako bi ubacivanje novog čvora uticalo na pronađeno minimalno povezujuće drvo manjeg grafa. Naime, nemamo garanciju da je minimalno povezujuće drvo podgrafa poddrvo minimalnog povezujućeg drveta celog grafa (slika 2.81). Da bi ubačeni čvor bio povezan sa ostalima, sigurno je potrebno da neka grana koja ga spaja sa susedima bude uključena u krajnje drvo. Međutim, moguće je da umesto samo jedne takve grane u drvo treba ubaciti i više njih, a izbaciti neke grane ranije konstruisanog poddrveta. Analiza ovog pristupa postaje komplikovana i odustaćemo od nje, jer, kao što ćemo videti postoje jednostavniji, veoma efikasni algoritmi.



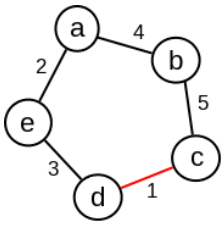
Slika 2.81: Primer kada minimalno povezujuće drvo podgrafa G' grafa dobijenog izbacivanjem jednog čvora iz grafa G nije podgraf minimalnog povezujućeg drveta grafa G.

Do sličnog zaključka se dolazi i ako se pokuša indukcija po broju grana. Za razliku od čvorova među kojima ne možemo unapred uspostaviti neki prirodan redosled, grane se prirodno mogu urediti po težini i možemo uticati na to koju granu ćemo izbaciti, rekurzivno rešiti potproblem i na kraju je ponovo vratiti u drvo. Važi (što ćemo kasnije i formalno dokazati) da najkraća grana grafa mora biti uključena u minimalno povezujuće drvo. Zato je prirodno nju ukloniti iz grafa, rešiti rekurzivno potproblem (primeniti induktivnu hipotezu) i na kraju je uključiti. Da li je ovo regularna primena indukcije?

Prvi problem u ovakvoj primeni indukcije je u tome što posle uklanjanja grane, dobijeni problem nije ekvivalentan polaznom. Naime, izbor jedne grane ograničava mogućnosti izbora drugih grana.

Primer 2.10.1

Razmotrimo primer grafa sa slike 2.82. Grana (c, d) je grana minimalne težine u grafu, i ako nju uklonimo i problem svedemo na traženje minimalnog povezujućeg drveta u grafu G bez grane (c, d), dobijamo da njega čine sve ostale grane grafa: grane (d, e), (e, a), (a, b) i (b, c). Dodavanjem grane (c, d) u ovo minimalno povezujuće drvo dobijamo ciklus, a znamo da ciklus ne može biti deo minimalnog povezujućeg drveta.

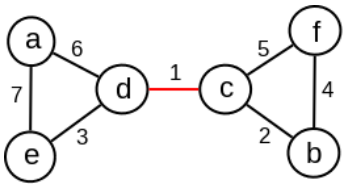


Slika 2.82: Graf koji je u obliku ciklusa.

Drugi problem sa primenom ovakve induktivne hipoteze je taj što uklonjena grana može biti most i nakon njenog uklanjanja graf ne mora da ostane povezan.

Primer 2.10.2

Razmotrimo graf sa slike 2.83: nakon uklanjanja grane (c, d) minimalne težine, graf se raspada na dve komponente povezanosti.



Slika 2.83: Graf koji se nakon uklanjanja grane minimalne težine raspada na dve komponente povezanosti.

Možemo rekurzivno konstruisati minimalnu povezujuću šumu, ali ponovo, kao i u slučaju indukcije po broju čvorova, odustajemo od ovog pristupa, jer postoje jednostavniji.

2.10.2 Primov algoritam

Jedan mogući pristup rešavanju ovog problema je da se tokom izvršavanja algoritma minimalno povezujuće postupno gradi, dodajući u svakom koraku trenutnom poddrvetu jednu novu granu i jedan novi čvor. Prema tome, indukcija se ovde ne izvodi po veličini grafa (broju grana ili čvorova grafa), već prema broju *grana u poddrvetu minimalnog povezujućeg drveta koje se gradi*.

Induktivna hipoteza: Za zadati povezan graf $G = (V, E)$ umemo da pronađemo drvo T_k sa k grana ($k < |V| - 1$), tako da je drvo T_k poddrvo nekog minimalnog povezujućeg drveta T grafa G.

Bazni slučaj je $k = 0$. Pošto svaki čvor grafa pripada njegovom minimalnom povezujućem drvetu T (koje postoji, jer je graf povezan), drvo T_0 može da sadrži proizvoljni čvor grafa i nijednu granu.

Pretpostavimo da smo pronašli drvo T_k koje zadovoljava induktivnu hipotezu i da je potrebno da T_k proširimo narednom granom. Kako da pronađemo novu granu za koju ćemo biti sigurni da pripada nekom minimalnom povezujućem drvetu? U svakom povezujućem drvetu koje proširuje drvo T_k mora da postoji bar jedna grana koja povezuje neki čvor iz T_k sa nekim čvorom u ostatku grafa. Neka je E_k skup svih takvih grana. Pošto je graf povezan, taj skup mora biti neprazan. Za narednu granu uzimamo proizvoljnu granu sa najmanjom težinom iz E_k. Tvrdimo da svaka takva grana pripada nekom minimalnom povezujućem drvetu T.

Lema 2.10.1

Neka je T_k poddrvo nekog minimalnog povezujućeg drveta T težinskog grafa $G = (V, E)$ i neka je E_k skup grana koje povezuju čvorove iz T_k sa čvorovima van T_k. Neka je $e = (u, v)$ neka od grana najmanje težine među granama iz E_k. Tada postoji minimalno povezujuće drvo T' grafa G koje sadrži sve grane iz T_k i granu e.

Dokaz. Ako drvo T sadrži granu e, onda je ono traženo drvo $T' = T$.

Ako T ne sadrži e, ono sadrži tačno jedan put od u do v. Pošto grana (u, v) ne pripada minimalnom povezujućem drvetu T, onda ona ne pripada ni putu od u do v kroz grane drveta T. Međutim, pošto u pripada, a v ne pripada drvetu T_k, na tom putu mora da postoji bar još jedna grana (u', v') takva da $u' \in T_k$ i $v' \notin T_k$. Razmotrimo odnos težina grana (u, v) i (u', v'):

- Težina (u, v) je najmanja među težinama grana koje povezuju T_k sa ostatkom grafa, pa težina grane (u', v') ne može biti manja od nje.
- Težina grane (u', v') ne može biti ni veća od težine grane (u, v). Zaista, ako granu (u, v) dodamo minimalnom povezujućem drvetu T, a iz njega izbacimo granu (u', v'), dobijamo povezujuće drvo manje težine, što je kontradikcija.
- Ako je težina grane (u', v') jednaka težini grane (u, v), tada se zamenom grane (u', v') granom (u, v) dobija povezujuće drvo T' iste težine, pa je i ono minimalno. Dakle, postoji minimalno povezujuće drvo koje sadrži granu (u, v).

□

Algoritam se, dakle, sprovodi na sledeći način. Drvo T se inicijalizuje drvetom T_0 koje sadrži samo jedan (proizvoljno odabrani) čvor. U svakoj iteraciji se vrši proširivanje trenutno konstruisanog drveta T_k jednom granom i čvorom, tako što se pronalazi neka grana iz skupa E_k koji sadrži sve grane koje povezuju T_k sa nekim čvorom van T_k, a koja ima najmanju težinu od svih grana u tom skupu (ako ima više takvih, bira se proizvoljna).

Umesto čuvanja i ažuriranja celog skupa E_k, za svaki čvor v van T_k možemo u nizu pamtiti rastojanje od drveta T_k, tj. minimalnu težinu grane od v do nekog čvora iz T_k (ako takva grana ne postoji, vrednost postavljamo na $+\infty$) i u svakom koraku možemo određivati minimum tog niza. Kada se odabere grana najmanje težine iz E_k i njen čvor $v \notin T_k$, analiziramo sve grane koje vode od v do nekog čvora w van T_k i ako je težina neke takve grane (v, w) manja od trenutnog rastojanja w od drveta T_k, ažuriramo to rastojanje (i granu koja od drveta vodi do njega).

Radi efikasnog nalaženja minimuma, sve grane skupa E_k možemo čuvati u redu sa prioritetom, uređenom po dužinama grana. Nakon dodavanja novog čvora v u T_k, u red se dodaju grane koje ga spajaju sa čvorovima van T_k. Međutim, neke grane koje su bile u E_k ne treba više da budu, jer sada spajaju dva čvora u drvetu. Slično kao kod Dajkstrinog algoritma i ovde možemo koristiti lenjo ažuriranje reda sa prioritetom i te grane ne moramo brisati iz reda. Zato, kada se grana minimalne težine izvadi iz reda sa prioretom, treba proveriti da li ona spaja čvor u T_k sa čvorom van njega. Ako spaja dva čvora unutar T_k, treba je prosto preskočiti.

Postupak se završava kada se konstruiše drvo $T_{|V|-1} \equiv T$, jer ono sadrži sve čvorove grafa (pošto je broj čvorova uvek za jedan veći od broja grana, tada drvo povezuje svih $|V|$ čvorova). Pošto je na osnovu leme 2.10.1 i ono poddrvo nekog minimalnog povezujućeg drveta, ono mora biti minimalno povezujuće drvo.

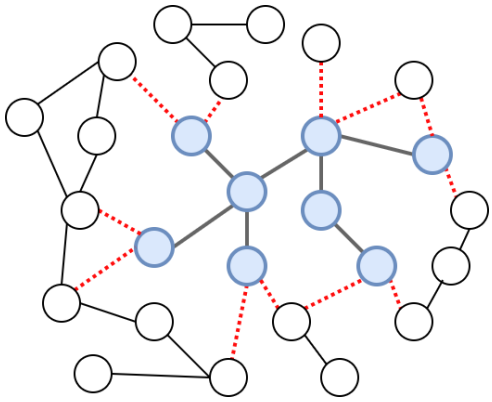
Opisani algoritam poznat je pod nazivom *Primov algoritam* i sličan je Dajkstrinom algoritmu za nalaženje najkraćih puteva od zadatog čvora. Interesantno je da je ovaj algoritam prvi osmislio Vojteh Jarnik 1930. godine, a da su tek kasnije nezavisno do njega došli Robert Prim 1957. godine i Edzger Dajkstra 1959. godine.

Iz opisa induktivne konstrukcije i leme 2.10.1 sledi sledeća teorema.

Teorema 2.10.1 **[Korektnost Primovog algoritma]**

Za proizvoljan neusmeren povezan težinski graf $G = (V, E)$ Primov algoritam konstruiše jedno minimalno povezujuće drvo.

Jedina razlika između Primovog i Dajkstrinog algoritma je u tome što se ne traži čvor van trenutnog skupa koji ima minimalno rastojanje od početnog čvora, već se traži čvor van trenutnog skupa koji ima minimalno rastojanje od trenutno konstruisanog drveta. Ostatak algoritma prenosi se praktično bez promene.



Slika 2.84: Nalaženje sledećeg čvora i grane minimalnog povezujućeg drveta. Podebljano je drvo T_k, a grane skupa E_k su prikazane isprekidano. Drvetu se dodaje neka grana tog skupa koja ima najmanju težinu.

```
// Primov algoritam za odredjivanje minimalnog povezujuceg drveta
void minimalnoPovezujuceDrvoPrim() {
   int brojCvorova = listaSuseda.size();
   // da li je cvor ukljucen u trenutno drvo
   vector<bool> uDrvetu(brojCvorova, false);
   // najkrace rastojanje od cvora do trenutnog drveta tj.
   // duzina najkrace grane koja povezuje trenutno drvo i cvor
   vector<Tezina> minRastojanja(brojCvorova, INF);
   // roditeljski cvor svakog cvora u minimalnom povezujucem drvetu
   vector<Cvor> roditelji(brojCvorova, -1);

   // uredjen par koji čine rastojanje do cvora i broj cvora;
   // redosled elemenata mora biti ovakav zbog operacije poredjenja parova
   typedef pair<Tezina, Cvor> RastojanjeCvora;
   // min-hip u koji smestamo duzine najkracih grana od drveta do svih cvorova
   priority_queue<RastojanjeCvora,
                  vector<RastojanjeCvora>,
                  greater<RastojanjeCvora>> rastojanja;

   // pocetni cvor moze biti bilo koji
   Cvor start = 0;
   // ubacujemo pocetni cvor u hip i postavljamo duzinu grane do njega na 0
   rastojanja.emplace(0, start);
   minRastojanja[start] = 0;

   // broj cvorova trenutno ukljucenih u minimalno povezujuce drvo
   int brojCvorovaUDrvetu = 0;

   // dok se svi cvorovi ne ukljuce u drvo
   while (brojCvorovaUDrvetu < brojCvorova) {
      // izdvajamo cvor cvor najblizi drvetu (u prvom koraku izdvajamo pocetni cvor)
      auto [rastojanje, cvor] = rastojanja.top();
      rastojanja.pop();

      // ako je taj cvor vec u drvetu, treba ga preskociti
      // ovo se moze desiti jer vrsimo lenjo azuriranje hipa
```

```
      if (uDrvetu[cvor])
        continue;

      // dodajemo cvor u drvo
      uDrvetu[cvor] = true;
      brojCvorovaUDrvetu++;

      // za sve susede tekuceg cvora
      for (const auto& [sused, tezina] : listaSuseda[cvor]) {
         if (!uDrvetu[sused]) {
            // ako je grana iz tekuceg cvora do sused kraca od
            // prethodno najkrace grane iz drveta do tog suseda,
            // azuriramo vrednost najkrace grane i roditeljskog cvora
            if (tezina < minRastojanja[sused]) {
               minRastojanja[sused] = tezina;
               roditelji[sused] = cvor;
               // ubacujemo element u hip;
               // ako je postojala prethodna vrednost, ne brisemo je:
               // nova vrednost ce se naci u hipu iznad stare
               rastojanja.emplace(minRastojanja[sused], sused);
            }
          }
       }
   }

   cout << "Minimalno povezujuce drvo se sastoji od grana: " << endl;
   for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
       if (roditelji[cvor] != -1)
          cout << "(" << roditelj[cvor] << "," << cvor << ") težine "
          << minRastojanja[cvor] << endl;
}
```

Kada se koristi red sa prioritetom, složenost Primovog algoritma identična je složenosti Dajkstrinog algoritma za nalaženje najkraćih rastojanja od zadatog čvora i iznosi $O((|E| + |V|) \log |V|)$.

Ukoliko su težine svih grana u grafu međusobno različite, minimalno povezujuće drvo grafa biće jedinstveno.

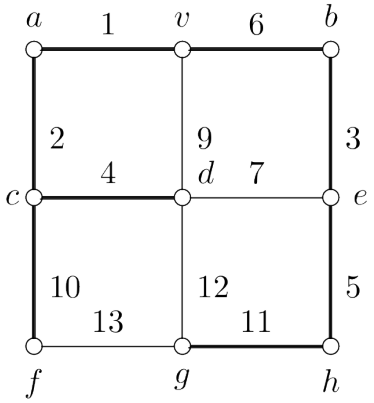
Primer 2.10.3

Primov algoritam za konstrukciju minimalnog povezujućeg drveta ilustrovaćemo primerom na slici 2.85. Čvor u prvoj koloni tabele je onaj koji je dodat u odgovarajućem koraku. Prvi dodati čvor je v, i u prvoj vrsti navedeni su čvorovi do kojih postoje grane iz čvora v sa svojim težinama. U svakoj vrsti bira se grana sa najmanjom težinom. Spisak trenutno najboljih grana i njihovih težina ažurira se u svakom koraku (prikazani su samo krajevi grana). Na slici grafa su grane koje pripadaju minimalnom povezujućem drvetu podebljane.

Primov algoritam je primer pohlepnog algoritma, jer se u svakom koraku bira grana skupa E_k sa najmanjom težinom, tj. izbor lokalnog minimuma vodi i ka globalnom minimumu.

2.10.3 Kraskelov algoritam

Intuitivno je jasno da je poželjno da minimalno povezujuće drvo sadrži grane grafa sa što manjim težinama. Ako su sve težine grafa međusobno različite, grana najmanje težine u grafu mora biti uključena u minimalno



	v	a	b	c	d	e	f	g	h
v	–	$v(1)$	$v(6)$	∞	$v(9)$	∞	∞	∞	∞
a	–	–	$v(6)$	$a(2)$	$v(9)$	∞	∞	∞	∞
c	–	–	$v(6)$	–	$c(4)$	∞	$c(10)$	∞	∞
d	–	–	$v(6)$	–	–	$d(7)$	$c(10)$	$d(12)$	∞
b	–	–	–	–	–	$b(3)$	$c(10)$	$d(12)$	∞
e	–	–	–	–	–	–	$c(10)$	$d(12)$	$e(5)$
h	–	–	–	–	–	–	$c(10)$	$h(11)$	–
f	–	–	–	–	–	–	–	$h(11)$	–
g	–	–	–	–	–	–	–	–	–

Slika 2.85: Primer izvršavanja Primovog algoritma za nalaženje minimalnog povezujućeg drveta.

povezujuće drvo. Ako ona ne bi bila uključena, onda bi njeno dodavanje minimalnom povezujućem drvetu zatvorilo neki ciklus; uklanjanjem proizvoljne druge grane iz tog ciklusa ponovo se dobija drvo, ali manje težine — što je u suprotnosti sa pretpostavkom o minimalnosti konstruisanog povezujućeg drveta. Ovo inspiriše algoritam koji obilazi grane grafa u neopadajućem redosledu težina i uključuje jednu po jednu u graf. Ako postoji više grana iste najmanje težine koje mogu biti dodane, bira se bilo koja od njih. Ipak, potrebno je voditi računa da neke grane sa ranije dodatim granama zatvaraju ciklus i njih moramo preskočiti.

Dakle, i drugi efikasan algoritam za određivanje minimalnog povezujućeg drveta datog neusmerenog težinskog grafa $G = (V, E)$ je pohlepan, jer u svakom koraku bira jednu od najkraćih raspoloživih grana, ali do minimalnog povezujućeg drveta ne dolazi dodavanjem novih grana na trenutno drvo, nego na trenutnu šumu.

Indukcija u ovom algoritmu teče po broju obrađenih grana u neopadajućem redosledu težine grana.

Induktivna hipoteza: Za zadati povezan graf $G = (V, E)$ i skup njegovih k grana $E_k \subseteq E$ najmanje težine umemo da odredimo šumu K, koja je deo nekog minimalnog povezujućeg drveta grafa G. To drvo sadrži sve grane skupa K, a ne sadrži ni jednu granu skupa $L = E_k \setminus K$.

Inicijalno svaki čvor grafa predstavlja zasebno drvo i odgovarajuća šuma sadrži $|V|$ drveta i nijednu granu. Zatim se postepeno spajaju po dva drveta ove šume, dodavanjem grana. Dodavanjem svake nove grane, broj drveta u šumi se smanjuje za 1, te će se nakon dodavanja $|V| - 1$ grane dobiti tačno jedno drvo. Pri tome se grane koje se dodaju razmatraju u neopadajućem redosledu težina; ako grana koja je sledeća na redu povezuje dva čvora u različitim drvetima trenutne šume, onda se ta grana uključuje u šumu, čime se ta dva drveta spajaju u jedno. U protivnom, ako grana povezuje dva čvora iz istog drveta trenutne šume, grana se preskače.

Ovaj algoritam je prvi objavio Džozef Kraskel 1956. godine i poznat je pod nazivom *Kraskelov algoritam*.

Dokažimo da će ovaj algoritam uvek vratiti neko minimalno povezujuće drvo datog grafa.

Teorema 2.10.2 **[Korektnost Kraskelovog algoritma]**

Za proizvoljan neusmeren povezan težinski graf $G = (V, E)$, Kraskelov algoritam konstruiše jedno minimalno povezujuće drvo.

Dokaz. Lako se pokazuje da algoritam vraća povezujuće drvo datog grafa (jer na kraju imamo n čvorova povezanih sa $n - 1$ grana, bez ciklusa, što mora biti drvo). Treba pokazati da je ono i minimalno.

Dokazaćemo indukcijom da u svakom koraku algoritma (prilikom razmatranja bilo koje grane) postoji minimalno povezujuće stablo T grafa koje sadrži sve grane skupa K koje su odabrane u prethodnim koracima

i ne sadrži ni jednu granu skupa L koje su odbačene u prethodnim koracima. Nakon obrade svih grana, K je drvo koje mora biti jednako minimalnom povezujućem drvetu koje ga sadrži.

U početnom koraku nije niti prihvaćena, niti odbačena ni jedna grana grafa. Pošto je graf povezan on ima minimalno povezujuće drvo T. To drvo zadovoljava uslov jer obuhvata početni prazan skup grana $K = \emptyset$ i ne obuhvata ni jednu granu skupa $L = \emptyset$.

Neka je e naredna grana koju razmatra Kraskelov algoritam u trenutku kada su u šumu prethodno dodate grane skupa K i odbačene grane iz nekog skupa L. Ona može biti dodata u tekuću šumu ili odbačena.

Razmotrimo slučaj dodavanja nove grane e u šumu. Neka je T neko minimalno povezujuće drvo koje obuhvata sve grane skupa K i nijednu granu iz L (ono postoji na osnovu induktivne hipoteze). Nova šuma $K' = K \cup \{e\}$ sadrži i granu e i ona ne zatvara ciklus sa ostalim granama iz K.

- Ako grana e pripada drvetu T, tada je $T' = T$ traženo minimalno povezujuće drvo koje sadrži sve grane iz skupa $K' = K \cup \{e\}$ i nijednu granu iz skupa $L' = L$.
- Ako grana e ne pripada drvetu T, onda se njenim dodavanjem u T zatvara neki ciklus u drvetu T (jer u drvetu T postoji neki drugi put koji povezuje krajnje čvorove grane e). Taj ciklus ne postoji u šumi $K \cup e$ (jer se u suprotnom grana e ne bi dodavala šumi). Zato u tom ciklusu mora da postoji neka grana e', različita od e, koja pripada drvetu T, ali ne pripada šumi K. Pošto e' pripada drvetu T, ona ne može da pripada skupu L odbačenih grana (jer T ne sadrži ni jednu odbačenu granu iz skupa L). Dakle, e' ne pripada ni skupu L ni skupu K i različita je od e. Zato je Kraskelov algoritam još nije razmatrao, pa njena težina mora biti veća ili jednaka od težine grane e. Izbacivanjem grane e' iz drveta T i dodavanjem grane e dobijamo novo drvo T' (ono ima isti broj čvorova i grana kao drvo T, a ostaje povezano, jer se jedna grana koja zatvara ciklus menja drugom granom tog ciklusa). Ukupna težina drveta T' ne može biti manja od težine drveta T, jer je T minimalno povezujuće drvo, ne može biti veća, jer težina grane e koja se dodaje nije veća od težine grane e' koja se izbacuje, pa ukupne težine drveta T i T' moraju biti jednake. T' je zato takođe minimalno povezujuće drvo, a ono sadrži sve grane iz skupa $K' = K \cup \{e\}$ i ne sadrži ni jednu granu iz skupa odbačenih grana $L' = L$.

Razmotrimo na kraju slučaj odbacivanja grane e. Jedini razlog da se ona odbaci je kada ona zatvara neki ciklus sa granama iz K. Međutim, pošto se sve grane iz K nalaze u drvetu T, a u T nema ciklusa, grana e ne pripada drvetu T. Zato je T traženo minimalno povezujuće drvo koje sadrži sve grane iz skupa $K' = K$ i ni jednu granu iz skupa $L' = L \cup \{e\}$. □

Za utvrđivanje da li su krajnji čvorovi u i v tekuće grane (u, v) u istom ili u različitim drvetima trenutne šume, može se iskoristiti struktura podataka za disjunktne skupove (union-find): trenutna drveta šume su disjunktni podskupovi skupa čvorova grafa. Operacije `pronadji(u)` i `pronadji(v)` pronalaze predstavnike u', v' dva podskupa (korene drveta kojim su oni predstavljeni), pa su čvorovi u i v u istom podskupu ako i samo ako je $u' = v'$. Ako je $u' \neq v'$, onda se ta dva podskupa zamenjuju svojom unijom, tj. primenjuje se operacija `unija(u',v')`, a dva poddrveta trenutne šume se granom spajaju u jedno.

```
// funkcija za inicijalizaciju union-find strukture
void UF_inicijalizuj(vector<int>& UF_roditelj, vector<int> &UF_rang, int n) {
  // svaki cvor je podskup za sebe
  for (int i = 0; i < n; i++) {
    UF_roditelj[i] = i;
    UF_rang[i] = 0;
  }
}

// funkcija koja izracunava kom podskupu pripada neki element
int UF_pronadji(int x, vector<int>& UF_roditelj) {
```

```
  int koren = x;
  while (koren != UF_roditelj[koren])
    koren = UF_roditelj[koren];
  while (x != koren) {
    int tmp = UF_roditelj[x];
    UF_roditelj[x] = koren;
    x = tmp;
  }
  return koren;
}

// funkcija koja pravi uniju dva podskupa
void UF_unija(int x, int y, vector<int>& UF_roditelj, vector<int>& UF_rang) {
  int fx = UF_pronadji(x, UF_roditelj);
  int fy = UF_pronadji(y, UF_roditelj);
  if (UF_rang[fx] < UF_rang[fy])
     UF_roditelj[fx] = fy;
  else if (UF_rang[fy] < UF_rang[fx])
     UF_roditelj[fy] = fx;
  else {
     UF_roditelj[fx] = fy;
     UF_rang[fy]++;
  }
}

// Kraskelov algoritam za odredjivanje minimalnog povezujuceg drveta
void minimalnoPovezujuceDrvoKraskel() {
   int brojCvorova = listaSuseda.size();

   // roditelj cvora u union-find strukturi
   vector<Cvor> UF_roditelj(brojCvorova);
   // rang cvora u union-find strukturi
   vector<int> UF_rang(brojCvorova);

   // inicijalizujemo union-find strukturu
   // svaki cvor grafa predstavlja podskup za sebe
   UF_inicijalizuj(UF_roditelj, UF_rang, brojCvorova);

   // kreiramo jedinstveni niz svih grana u grafu
   typedef pair<Cvor, Cvor> Grana;
   vector<pair<Tezina, Grana>> grane;
   for (Cvor cvor = 0; cvor < brojCvorova; cvor++)
      for (const auto& [sused, tezina]: listaSuseda[cvor]) {
        Grana grana{cvor, sused};
        grane.emplace_back(tezina, grana);
      }

   // sortiramo skup grana u neopadajucem redosledu težina
   sort(grane.begin(), grane.end());

   // grane koje pripadaju konacnom minimalnom povezujucem drvetu
```

```
    vector<pair<Grana, Tezina>> drvo;
    // drvo ce sadrzati |V| - 1 grana, pa unapred rezervisemo memoriju
    drvo.reserve(brojCvorova - 1);

    // prolazimo redom kroz skup grana
    for (const auto& [tezina, grana] : grane) {
       // krajnji cvorovi grane
       auto [u, v] = grana;
       // predstavnici cvorova u union-find strukturi
       Cvor predstavnik_u = UF_pronadji(u, UF_roditelj);
       Cvor predstavnik_v = UF_pronadji(v, UF_roditelj);

       // ako tekuca grana povezuje dva cvora koja pripadaju
       // razlicitim drvetima onda tu granu dodajemo u minimalno povezujuce drvo
       // i pravimo uniju skupova cvorova koji pripadaju tim drvetima
       if (predstavnik_u != predstavnik_v) {
          // spajamo drveta kojima pripadaju u i v (dodavanjem grane uv)
          UF_unija(predstavnik_u, predstavnik_v, UF_roditelj, UF_rang);
          // granu dodajemo u skup grana koje cine sumu (tj. drvo na kraju)
          drvo.emplace_back(grana, tezina);
          // drvo je kompletno konstruisano kada je broj grana za jedan manji
          // od broja cvorova
          if (drvo.size() == brojCvorova - 1)
             break;
       }
    }

    cout << "Minimalno povezujuce drvo se sastoji od grana: " << endl;
    for (const auto& [grana, tezina] : drvo) {
       auto [u, v] = grana;
       cout << "(" << u << "," << v << ") težine "
            << tezina << endl;
    }
}
```

Funkcija za inicijalizaciju union-find strukture za disjunktne skupove je složenosti $O(|V|)$, dodavanje grana u skup grana je složenosti $O(|E|)$, njihovo sortiranje je složenosti $O(|E|\log|E|)$, a nakon toga se glavna petlja izvršava $|E|$ puta, dok su operacije koje se izvršavaju u petlji (`UF_pronadji` i `UF_unija`) složenosti $O(\log|V|)$. Dakle, ukupna složenost Kraskelovog algoritma iznosi $O(|V|)+O(|E|)+O(|E|\log|E|)+O(|E|\log|V|)=O(|E|\log|V|)$ (koristimo činjenicu da je $O(\log|E|)=O(\log|V|)$ zbog $|E|\leq|V|^2$).

Primer 2.10.4

Primer izvršavanja Kraskelovog algoritma za graf sa slike 2.85 prikazan je u tabeli 2.3. Prolazimo redom kroz skup grana prema neopadajućem redosledu težina i završavamo obradu kada u skup dodamo $|V|-1$ (u ovom slučaju 8) grana. Kao rezultat dobijamo minimalno povezujuće drvo prikazano na slici 2.85.

Tabela 2.3: Primer izvršavanja Kraskelovog algoritma za graf sa slike 2.85.

naredna grana	*dužina grane*	*uključena u MCST?*	*trenutna šuma*
(a,v)	1	*da*	$\underline{v},\underline{a},b,c,d,e,f,g,h$
(a,c)	2	*da*	$\underline{av},b,\underline{c},d,e,f,g,h$

naredna grana	*dužina grane*	*uključena u MCST?*	*trenutna šuma*
(b, e)	3	*da*	$acv, \underline{b}, c, d, \underline{e}, f, g, h$
(c, d)	4	*da*	$a\underline{c}v, be, \underline{d}, e, f, g, h$
(e, h)	5	*da*	$acdv, b\underline{e}, f, g, \underline{h}$
(b, v)	6	*da*	$acd\underline{v}, \underline{b}eh, f, g$
(d, e)	7	*ne*	$abc\underline{\underline{de}}hv, f, g$
(d, v)	8	*ne*	$abc\underline{d}eh\underline{v}, f, g$
(c, f)	9	*da*	$ab\underline{c}dehv, \underline{f}, g$
(g, h)	10	*da*	$abcdef\underline{h}v, \underline{g}$
			$abcdefghv$

Konačno, napomenimo da se na sličan način može konstruisati i povezujuće drvo sa maksimalnom mogućom ukupnom težinom grana, tzv. *maksimalno povezujuće drvo* grafa. Jedino je potrebno grane obrađivati u obrnutom redosledu.

Zadatak: Voda do svake kuće

U jednom selu ima n kuća. Želimo da izgradimo vodovod tako da do svake kuće dolazi voda, tako što možemo da gradimo bunare i povezujemo kuće cevima. Za svaku kuću i možemo ili da izgradimo bunar ili da je cevima povežemo sa nekim susednim kućama. Kroz svaku cev voda može da ide u proizvoljnom smeru. Ako su poznate cene izgradnje bunara u svakoj kući i cene povezivanja kuća cevima, napiši program koji izračunava najmanju cenu potrebnu da svaka kuća dobije vodu.

Opis ulaza

Sa standardnog ulaza se učitava broj kuća n ($1 \leq n \leq 5 \cdot 10^4$), zatim n brojeva koji predstavljaju cene izgradnje bunara za svaku kuću, a zatim do kraja ulaza cene izgradnji cevi (po tri cela broja u redu, gde prva dva broja predstavljaju različite kuće, a treći cenu izgradnje cevi između tih kuća, pri čemu ukupan broj cevi ne prelazi 10^6). Brojevi kuća su od 1 do n.

Opis izlaza

Na standardni izlaz ispisati najmanju cenu izgradnje vodovoda.

Primer 1

Ulaz

```
3
1 2 2
1 2 1
2 3 1
```

Izlaz

```
3
```

Objašnjenje

Najbolje je napraviti bunar u kući 1 i povezati druge dve kuće cevima sa njom.

Primer 2

Ulaz

```
4
2 2 2 2
```

```
1 2 5
1 3 5
1 4 5
2 3 5
2 4 5
3 4 5
```

Izlaz

```
8
```

Objašnjenje

Najbolje je da svaka kuća dobije svoj bunar.

Rešenje

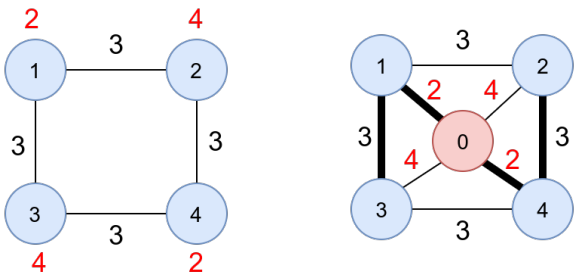
Kada bunari ne bi postojali i kada bi samo bilo potrebe povezati sve kuće cevima, problem bi se direktno svodio na pronalaženje minimalnog povezujućeg stabla datog skupa kuća.

Mogućnost i potreba da se izgrade bunarni naizgled čine zadatak komplikovanijim. Međutim, postoji veoma elegantan način da se i problem sa bunarima svede na problem minimalnog povezujućeg stabla tj. da se izgradnja bunara modeluje izgradnjom cevi. Naime, svaki se bunar može zamisliti kao cev koja spaja kuću za zemljom u kojoj se već nalazi voda. Zato polazni graf u kome se nalaze kuće i cevi koje ih mogu povezati treba dopuniti jednim posebnim čvorom koji predstavlja zemlju i svaku kuću treba povezati sa tim čvorom pri čemu je cena cevi koja spaja zemlju i kuću jednaka ceni izgradnje bunara u toj kući. Kada se pronađe minimalno povezujuće stablo, ono će sigurno povezivati zemlju sa ostalim kućama, što znači da će voda iz zemlje sigurno moći da bude dopremljena do svake druge kuće (da bi zemlja bila povezana, sigurno će bar jedan bunar biti izgrađen). Ako gradimo usmeren graf, dovoljno je da usmerimo grane od zemlje ka svakoj kući (u zemlji se već nalazi voda, pa nije potrebno vraćati vodu iz neke kuće nazad do zemlje).

Kada se napravi ovako prošireni graf, minimalno povezujuće stablo može da se pronađe uobičajenim algoritmima.

Primer 2.10.5

Na slici levo je prikazan graf gde su prikazane moguće cevi i cene bunara za svaku, dok je na slici desno prikazan proširen graf gde je uveden čvor 0 (zemlja) i bunari su predstavljeni cevima koje spajaju zemlju i kuće. Nacrtano minimalno povezujuće stablo ukazuje na to da je potrebno izgraditi bunare u kućama 1 i 4, a da je kuće 2 i 3 potrebno cevima povezati sa kućama 1 i 4.



2.11 Svi najkraći putevi

Problem rutiranja u mrežama računara (ili distribuiranim sistemima) predstavlja odabir putanje kroz mrežu kojom paketi putuju od svoje početne destinacije do odredišta. Jedan od kriterijuma po kojima se vrši odabir

putanje jeste taj da pređeni put bude što je moguće kraći. Kako bi se prenos podataka od jednog uređaja do drugog učinio što je moguće efikasnijim, potrebno je poznavati najkraći put između svaka dva uređaja. Ovaj problem moguće je modelovati u vidu grafa, tako što ćemo svaki od uređaja u mreži predstaviti čvorom grafa, a svaku od postojećih veza između uređaja granom grafa čija težina odgovara meri rastojanja između ta dva uređaja. Tada se polazni problem svodi na problem izračunavanja najkraćeg puta između svaka dva čvora u težinskom grafu.

Problem

Dat je težinski graf $G = (V, E)$ (usmereni ili neusmereni). Odrediti puteve minimalne dužine između svaka dva čvora u grafu.

Ponovo, pošto govorimo o najkraćim putevima, težine grana možemo interpretirati kao dužine grana. Ovaj problem nazivamo *problem nalaženja svih najkraćih puteva* (engl. all pairs shortest path). Za početak ćemo se zadovoljiti nalaženjem dužina svih najkraćih puteva, umesto samih najkraćih puteva. Pretpostavimo da je graf usmeren; sve što će biti rečeno važi i za neusmerene grafove. Težine grana u grafu mogu biti negativne, ali u grafu ne sme postojati ciklus negativne težine.

Jedan mogući pristup je da se Belman-Fordov algoritam (ili Dajkstrin algoritam, ako su težine grana nenegativne) pokrene iz svakog čvora grafa. Međutim, videćemo da postoje i efikasniji algoritmi koji direktno rešavaju ovaj problem.

Kao i obično, pokušajmo sa direktnim induktivno-rekurzivnim pristupom. Može se koristiti indukcija po broju grana ili po broju čvorova u grafu. Označimo sa $d(u, v)$ dužinu grane (u, v), a sa $\overline{SP}(u, v)$ dužinu trenutno poznatog najkraćeg puta od čvora u do čvora v.

2.11.1 Algoritam zasnovan na indukciji po broju grana u grafu

Razmotrimo najpre indukciju po broju grana. Pretpostavimo da smo iz grafa G uklonili granu (u, v) i da smo rešili problem na preostalom grafu G', tj. da znamo dužine najkraćih puteva između svaka dva čvora u grafu G'. Kako se menjaju najkraći putevi u grafu nakon dodavanja grane (u, v) u graf? Nova grana može pre svega da predstavlja kraći put od do sada pronađenog najkraćeg puta od čvora u do čvora v. Dakle, potrebno je proveriti da li važi $d(u, v) < \overline{SP}(u, v)$ i ako važi, postaviti vrednost $\overline{SP}(u, v)$ na $d(u, v)$. Pored toga, dodavanjem grane (u, v) može se promeniti i najkraći put između proizvoljna druga dva čvora s i t. Da bi se ustanovilo ima li promene, treba sa prethodno poznatom najmanjom dužinom puta od čvora s do čvora t uporediti zbir dužina najkraćeg puta od s do u, dužine grane (u, v) i dužine najkraćeg puta od v do t. Dakle, ako je $\overline{SP}(s, u) + d(u, v) + \overline{SP}(v, t) < \overline{SP}(s, t)$ potrebno je novu vrednost $\overline{SP}(s, t)$ postaviti na $\overline{SP}(s, u) + d(u, v) + \overline{SP}(v, t)$.

Pošto je broj parova čvorova $O(|V|^2)$, pri dodavanju grane (u, v) potrebno je izvršiti $O(|V|^2)$ provera. Ovaj postupak je potrebno sprovesti za svaku granu grafa, te je složenost algoritma za nalaženje svih najkraćih puteva zasnovanog na indukciji po broju grana u najgorem slučaju $O(|E| \cdot |V|^2)$. Pošto je broj grana najviše $O(|V|^2)$, složenost ovog algoritma iznosi $O(|V|^4)$. Ovo je veoma neefikasno, pa ovaj algoritam nećemo implementirati.

2.11.2 Algoritam zasnovan na indukciji po broju čvorova u grafu

Razmotrimo sada indukciju po broju čvorova. Pretpostavimo da smo iz grafa G uklonili čvor u i da smo između svaka druga dva čvora u grafu pronašli najkraći put. Kako se menjaju najkraći putevi u grafu ako se u graf doda novi čvor u? Potrebno je najpre pronaći najkraće puteve od čvora u do svih ostalih čvorova, i najkraće puteve od svih ostalih čvorova do čvora u. Pošto su dužine najkraćih puteva koji ne sadrže u već poznate, određivanje najkraćeg puta od čvora u do proizvoljnog čvora w svodi se na određivanje samo prve grane na tom putu; ako je to grana (u, v), onda je dužina najkraćeg puta od čvora u do čvora w jednaka zbiru dužine grane (u, v) i dužine najkraćeg puta od v do w, koji je već poznat. Potrebno je, dakle, da uporedimo ove zbirove za sve grane koje polaze iz čvora u, i da među njima izaberemo najmanji:

$$\overline{SP}(u,w) = \min_{(u,v)\in E}\{d(u,v) + \overline{SP}(v,w)\}.$$

Najkraći put od proizvoljnog čvora w do čvora u može se pronaći na sličan način. Ove provere su u najgorem slučaju (kada iz čvora u polazi/u njega dolazi $\Theta(|V|)$ grana) složenosti $O(|V|^2)$.

Međutim, dodavanje čvora u može skratiti puteve između neka druga dva čvora, jer put preko u može biti kraći nego putevi koji ne prolaze kroz u. Zato je, dodatno, potrebno za svaki par čvorova proveriti da li između njih postoji novi kraći put kroz novi čvor u. Za proizvoljna dva čvora s i t grafa, da bi se ustanovilo postoji li kraći put preko čvora u, potrebno je sa prethodno poznatom najmanjom dužinom puta od s do t uporediti zbir dužine najkraćeg puta od s do u i dužine najkraćeg puta od u do t. Ako važi $\overline{SP}(s,u) + \overline{SP}(u,t) < \overline{SP}(s,t)$, ažuriramo vrednost $\overline{SP}(s,t)$.

Ovo uključuje ukupno $O(|V|^2)$ provera i sabiranja posle dodavanja svakog novog čvora, pa je složenost ovakvog algoritma u najgorem slučaju $O(|V| \cdot |V|^2) = O(|V|^3)$. Ukupan broj koraka za određivanje dužina najkraćih puteva od i do novododatog čvora je takođe $O(|V|^3)$. Dakle, ispostavlja se da je za rešavanje ovog problema efikasnija indukcija po broju čvorova nego indukcija po broju grana.

Razmotrimo implementaciju algoritma za računanje svih najkraćih puteva u grafu zasnovanog na indukciji po čvorovima, u slučaju kada je graf zadat matricom povezanosti. Naime, ovde nam je ta reprezentacija pogodnija od listi povezanosti jer dužine grana odgovaraju inicijalnim najkraćim rastojanjima između čvorova.

```
const int INF = numeric_limits<int>::max();

// odredjujemo sve najkrace puteve indukcijom po broju cvorova
vector<vector<int>> sviNajkraciPutevi(
          const vector<vector<int>>& matricaPovezanosti) {
  int brojCvorova = matricaPovezanosti.size();
  vector<vector<int>> najkraciPut(brojCvorova);
  for (int i = 0; i < brojCvorova; i++)
    najkraciPut[i].resize(brojCvorova, INF);

  // dodajemo jedan po jedan cvor
  for (int i = 0; i < brojCvorova; i++) {
    najkraciPut[i][i] = 0;

    // odredjujemo najkrace puteve od cvora i do svih prethodnih
    // cvorova j
    for (int j = 0; j < i; j++) {
      // pretpostavljamo da je direktno rastojanje najkrace
      najkraciPut[i][j] = matricaPovezanosti[i][j];
      // proveravamo da li je mozda bolji put od i do j koji vodi preko
      // nekog prethodnog cvora k
      for (int k = 0; k < i; k++)
        // ako postoji grana od i do k i put od k do j
        if (matricaPovezanosti[i][k] != INF && najkraciPut[k][j] != INF &&
            matricaPovezanosti[i][k] + najkraciPut[k][j] < najkraciPut[i][j])
          najkraciPut[i][j] = matricaPovezanosti[i][k] + najkraciPut[k][j];
    }

    // odredjujemo najkrace puteve do cvora i od svih prethodnih
    // cvorova j
```

```
    for (int j = 0; j < i; j++) {
      // pretpostavljamo da je direktno rastojanje najkrace
      najkraciPut[j][i] = matricaPovezanosti[j][i];
      // proveravamo da li je mozda bolji put od cvora j do i koji
      // vodi preko nekog prethodnog cvora k
      for (int k = 0; k < i; k++)
        // ako postoji grana od i do k i put od k do j
        if (najkraciPut[j][k] != INF && matricaPovezanosti[k][i] != INF &&
            najkraciPut[j][k] + matricaPovezanosti[k][i] < najkraciPut[j][i])
          najkraciPut[j][i] = najkraciPut[j][k] + matricaPovezanosti[k][i];
    }

    // ažuriramo rastojanja od prethodnih cvorova j do prethodnih
    // cvorova k, analizirajuci puteve koji vode preko cvora i
    for (int j = 0; j < i; j++)
      // ako postoji put od j do i i ako postoji put od i do k
      for (int k = 0; k < i; k++)
        if (najkraciPut[j][i] != INF && najkraciPut[i][k] != INF &&
            najkraciPut[j][i] + najkraciPut[i][k] < najkraciPut[j][k])
          najkraciPut[j][k] = najkraciPut[j][i] + najkraciPut[i][k];
  }
  return najkraciPut;
}

int main() {
  // broj cvorova u grafu
  int n;
  cin >> n;
  // matrica susedstva grafa koji sadrzi n cvorova
  vector<vector<int>> M(n);
  for (int i = 0; i < n; i++) {
    M[i].resize(n);
    for (int j = 0; j < n; j++) {
      cin >> M[i][j];
      // ako je uneto -1, tezinu grane postavljamo na +beskonacno
      // pretpostavljamo da u grafu ne postoji grana negativne tezine
      if (M[i][j] == -1)
        M[i][j] = INF;
    }
  }

  // racunamo najkrace puteve
  auto duzinaPuta = sviNajkraciPutevi(M);
  // stampamo najkrace puteve
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++)
      if (duzinaPuta[i][j] != INF)
        cout << duzinaPuta[i][j] << "\t";
      else
        cout << "-\t";
    cout << endl;
```

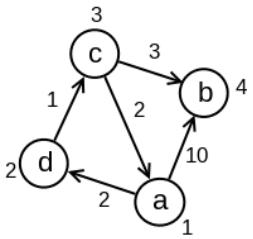
```
  }
  cout << endl;
  return 0;
}
```

2.11.3 Flojd-Varšalov algoritam

Pokazuje se da postoji još jednostavnija induktivna konstrukcija za rešavanje ovog problema. Ideja na kojoj se zasniva ovaj algoritam je da se ne menja broj čvorova ili grana u grafu, već da se uvedu ograničenja na tip dozvoljenih puteva. Naime, razmatraju se samo putevi koji kao međučvorove koriste čvorove iz zadatog skupa čvorova, pri čemu se u svakom koraku taj skup proširuje jednim novim čvorom. Na početku je taj skup prazan, pa u obzir dolaze samo grane grafa, a na kraju obuhvata sve čvorove grafa, pa u obzir dolaze svi mogući putevi. Numerišimo čvorove grafa na proizvoljan način brojevima od 1 do $|V|$. Put od čvora u do čvora v zove se *k-put*, gde je $k \in \{0, 1, \ldots, |V|\}$, ako su redni brojevi svih čvorova na tom putu (izuzev u i v) manji ili jednaki k (skup dopuštenih međučvorova je, dakle, skup čvorova označen brojevima od 1 do k). Specijalno, 0-put od čvora u do čvora v može biti samo direktna grana od čvora u do čvora v (pošto se nijedan drugi čvor ne može pojaviti na tom putu).

Primer 2.11.1

Razmotrimo primer grafa sa slike 2.86. Neka su redni brojevi čvorova a, d, c, b redom 1, 2, 3, 4. Put (a, b) od čvora a do čvora b dužine 10 je 0-put, jer se sastoji od samo jedne grane (nema usputnih čvorova na putu). Put (a, d, c, b) dužine 6 je 3-put, jer su redni brojevi svih čvorova na tom putu manji ili jednaki 3 (istovremeno je i 4-put), ali nije npr. 2-put. Slično, put (c, a, b) je 1-put, jer su redni brojevi svih čvorova (u ovom slučaju jednog jedinog čvora) na putu manji ili jednaki 1 (ovaj put je, takođe, i 2-put, 3-put i 4-put). Primetimo da je najkraći 0-put (istovremeno i najkraći 1-put i najkraći 2-put) od čvora a do čvora b put (a, b) dužine 10, dok je najkraći 3-put (i istovremeno najkraći 4-put) (a, d, c, b) dužine 6.



Slika 2.86: Ilustracija k-puteva u usmerenom težinskom grafu.

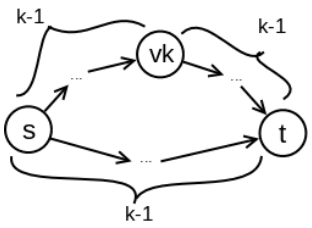
Razmotrimo algoritam zasnovan na indukciji po opadajućem broju ograničenja na tip dozvoljenih puteva.

Induktivna hipoteza: Umemo da odredimo dužine najkraćih $(k-1)$-puteva između svaka dva čvora.

Baza indukcije je slučaj $k = 1$, kad se razmatraju samo direktne grane i rešenje je očigledno: najkraći 0-put između dva čvora jednak je dužini grane ako ona postoji, a $+\infty$ ako grana ne postoji.

Pretpostavimo da je induktivna hipoteza tačna i da hoćemo da je proširimo na k-puteve. Potrebno je da odredimo najkraće k-puteve između svaka dva čvora. Neka je v_k čvor sa rednim brojem k. Proizvoljan najkraći k-put sadrži čvor v_k najviše jednom. Naime, pretpostavka je da u grafu ne postoji ciklus negativne dužine, pa najkraći put ne može dva puta da prođe kroz čvor v_k — u protivnom bi se put mogao skratiti izbacivanjem ciklusa od grana između prvog i drugog pojavljivanja čvora v_k. Najkraći k-put od čvora s do čvora t je ili najkraći $(k-1)$-put od čvora s do čvora t ili se sastoji od najkraćeg $(k-1)$-puta od čvora s do čvora v_k, i najkraćeg $(k-1)$-puta od čvora v_k do čvora t jer su čvorovi na k-putu od s do v_k i od v_k do t numerisani brojevima manjim ili jednakim $k-1$ (slika 2.87). Prema induktivnoj hipotezi mi već znamo

dužine najkraćih $(k-1)$-puteva. Dakle, da bismo pronašli dužinu najkraćeg k-puta od čvora s do čvora t dovoljno je da saberemo ove dve dužine i zbir uporedimo sa dužinom najkraćeg $(k-1)$-puta od čvora s do čvora t.



Slika 2.87: Ilustracija procesa računanja najkraćih k-puteva: najkraći $(k-1)$-put od čvora s do čvora t poredimo sa zbirom najkraćih $(k-1)$-puteva od s do v_m i od v_m do t.

Do ovog algoritma su nezavisno došli i objavili ga Robert Flojd i Stiven Varšal i poznat je pod nazivom *Flojd-Varšalov algoritam*. Napomenimo da je nekoliko godina pre njih do istog algoritma došao i Bernard Roj, međutim, taj rezultat je prošao nezapaženo. Ipak, negde u literaturi se ovaj algoritam pominje i pod nazivom Roj-Varšalov ili Roj-Flojdov algoritam. Flojd-Varšalov algoritam je za konstantan faktor brži od algoritma zasnovanog na primeni indukcije po broju čvorova i lakše ga je realizovati.

Prethodnim induktivnim razmatranjem dokazali smo korektnost ovog algoritma.

Teorema 2.11.1 **[Korektnost Flojd Varšalovog algoritma]**

Za dati usmeren težinski graf $G = (V, E)$ bez ciklusa negativne težine, Flojd-Varšalov algoritam ispravno izračunava najkraća rastojanja između svih parova čvorova.

Do sad smo se bavili izračunavanjem dužina najkraćih puteva između svaka dva čvora u grafu. Razmotrimo sada kako bismo mogli da rekonstruišemo same najkraće puteve, a da pritom čuvamo što manje informacija: poželjno je da za svaki najkraći put čuvamo samo po jednu vrednost. Prisetimo se da je u prethodnim algoritmima za svaki čvor bilo dovoljno čuvati pretposlednji čvor na najkraćem putu do tog čvora, jer je svaki naredni najkraći put bio dobijen produživanjem nekog prethodno određenog najkraćeg puta jednom novom granom. Ovde takva rekonstrukcija ne bi imala smisla, jer se najkraći putevi konstruišu na drugačiji način, pa algoritam rekonstrukcije najkraćih puteva mora da prati postupak konstrukcije najkraćih puteva. U algoritmu se tekući najkraći put između dva čvora menja onda kada je put kroz novorazmatrani čvor v_k kraći nego prehodno određeni najkraći $(k-1)$-put. Dakle, možemo angažovati dodatnu matricu put u kojoj ćemo na poziciji (i, j) pamtiti maksimalnu vrednost k takvu da čvor v_k pripada najkraćem putu od čvora i do čvora j, a ako je najkraći put direktna grana od i do j, onda vrednost $put[i][j]$ možemo postaviti na i. Ako se prilikom traženja najkraćeg puta od čvora i do čvora j nađe kraći put koji vodi preko čvora v_k, ažuriraćemo vrednost $put[i][j]$ na k. Ispisivanje najkraćeg puta od čvora i do čvora j se onda svodi na ispisivanje najkraćeg puta od čvora i do čvora sa rednim brojem $put[i][j]$ za kojim sledi najkraći put od čvora sa rednim brojem $put[i][j]$ do čvora j.

U algoritmu koji sledi pretpostavićemo da u grafu ne postoji grana negativne težine. Pretpostavljamo da je graf zadat matricom povezanosti jer ona gotovo u potpunosti kodira dužine 0-puteva u grafu. Naime, da bismo dobili dužine 0-puteva jedino je potrebno u poljima matrice koja odgovaraju parovima čvorova između kojih ne postoji grana postaviti umesto vrednosti -1 vrednost $+\infty$. Kao numeraciju čvorova iskoristićemo njihov indeks.

```
const int INF = numeric_limits<int>::max();

// funkcija koja stampa najkraci put od cvora i do cvora j
// bez ispisivanja cvora j
void odstampajPut(vector<vector<int>> put, int i, int j) {
```

```
  if (put[i][j] == -1)
    return;
  // put od i do j odgovara direktnoj grani (i,j)
  if (put[i][j] == i)
    cout << i << " - ";
  else{
    // stampamo put od cvora i do cvora k, gde je k maksimalni redni broj cvora
    // koji pripada tom putu, pa zatim put od cvora k do cvora j
    odstampajPut(put, i, put[i][j]);
    odstampajPut(put, put[i][j], j);
  }
}

// funkcija koja stampa najkrace puteve izmedju svaka dva cvora u grafu
void odstampajPuteve(vector<vector<int>> duzinaPuta,
                     vector<vector<int>> put, int brojCvorova) {
  cout << "Matrica najkracih rastojanja jednaka je: " << endl;
  for (int i = 0; i < brojCvorova; i++) {
    for (int j = 0; j < brojCvorova; j++)
      if (duzinaPuta[i][j] != INF)
        cout << duzinaPuta[i][j] << "\t";
      else
        cout << "-\t";
    cout << endl;
  }
  cout << endl;
  for (int i = 0; i < brojCvorova; i++)
    for (int j = 0; j < brojCvorova; j++)
      if (i != j && put[i][j] != -1) {
        cout << "Najkraci put od cvora " << i
             << " do cvora " << j << " je: ";
        odstampajPut(put,i,j);
        cout << j << endl;
      }
}

// funkcija koja racuna najkrace puteve izmedju svaka dva cvora
void sviNajkraciPutevi(const vector<vector<int>> &matricaPovezanosti) {

  int brojCvorova = matricaPovezanosti.size();
  // inicijalizujemo matricu koja cuva duzine najkracih puteva
  // na duzine direktnih grana, a ako direktna grana ne postoji
  // na vrednost +beskonacno
  vector<vector<int>> najkraciPut = matricaPovezanosti;
  vector<vector<int>> put(brojCvorova);

  // inicijalizujemo drugu matricu pomocu koje cemo
  // rekonstruisati najkrace puteve
  for (int i = 0; i < brojCvorova; i++) {
    put[i].resize(brojCvorova);
    for (int j = 0; j < brojCvorova; j++)
```

```
      // ako postoji direktna grana od cvora i do cvora j
      // pamtimo tu informaciju
      if (matricaPovezanosti[i][j] != INF)
        put[i][j] = i;
      // inace za i<>j postavljamo da je maksimalna oznaka cvora
      // na trenutnom putu -1
      else if (i != j)
        put[i][j] = -1;
      // slucaj kada razmatramo put od cvora do njega samog
      else
        put[i][j] = 0;
  }

  // proveravamo da li k-putevi skracuju puteve izmedju cvora i i j
  for (int k = 0; k < brojCvorova; k++)
    for (int i = 0; i < brojCvorova; i++)
      for (int j = 0; j < brojCvorova; j++)
        // ako postoji neki put od i do k i ako postoji neki put od k do j
        if (najkraciPut[i][k] != INF && najkraciPut[k][j] != INF
            && najkraciPut[i][k] + najkraciPut[k][j] < najkraciPut[i][j]) {
          najkraciPut[i][j] = najkraciPut[i][k] + najkraciPut[k][j];
          // postavljamo da je na putu od cvora i do cvora j
          // maksimalna oznaka cvora jednaka k
          put[i][j] = k;
        }

  // proveravamo da li je u grafu postojao ciklus negativne duzine
  bool negativniCiklus = false;
  for (i = 0; i < brojCvorova; i++)
    if (najkraciPut[i][i] < 0) {
      cout << "U grafu postoji ciklus negativne duzine" << endl;
      negativniCiklus = true;
      break;
    }
  // ako ne postoji ciklus negativne duzine
  // stampamo sve najkrace puteve
  if (!negativniCiklus)
    odstampajPuteve(najkraciPut, put, brojCvorova);
}

int main() {
  // broj cvorova u grafu
  int n;
  cin >> n;
  // matrica susedstva grafa koji sadrzi n cvorova
  vector<vector<int>> M(n);
  // ucitavamo matricu susedstva tezinskog grafa
  for (int i = 0; i < n; i++) {
    M[i].resize(n);
    for (int j = 0; j < n; j++) {
      cin >> M[i][j];
```

```
      // ako je uneto -1, tezinu grane postavljamo na +beskonacno
      // pretpostavljamo da u grafu ne postoji grana negativne tezine
      if (M[i][j] == -1)
        M[i][j] = INF;
    }
  }
  // racunamo najkrace puteve izmedju svaka dva cvora
  sviNajkraciPutevi(M);
  return 0;
}
```

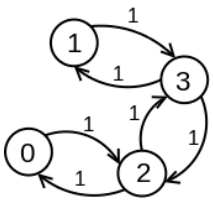
Napomenimo da je u trostruko ugnežđenoj petlji neophodno da spoljašnja petlja kontroliše parametar k koji ograničava tip dozvoljenih puteva, dok se unutrašnje dve petlje koriste se za proveru svih parova čvorova. Zapaža se da se ova provera može izvršavati sa parovima čvorova u proizvoljnom redosledu, jer je svaka od provera potpuno nezavisna od ostalih.

Primer 2.11.2

Razmotrimo primer grafa sa slike 2.88 kod koga su težine svih grana jednake 1. *Neka su redni brojevi čvorova* 0, 1, 2 *i* 3 *sa slike redom jednaki* 1, 2, 3 *i* 4. *U njemu postoji put od čvora* 0 *do čvora* 1 *dužine* 3 *i on predstavlja najkraći put od čvora* 0 *do čvora* 1. *Razmotrimo varijantu Flojd-Varšalovog algoritma sa narednim, malo izmenjenim redosledom petlji:*

```
for (int i = 0; i < brojCvorova; i++)
  for (int j = 0; j < brojCvorova; j++)
    for (int k = 0; k < brojCvorova; k++)
      if (najkraciPut[i][k] != INF && najkraciPut[k][j] != INF
          && najkraciPut[i][k] + najkraciPut[k][j] < najkraciPut[i][j]) {
        najkraciPut[i][j] = najkraciPut[i][k] + najkraciPut[k][j];
```

pri čemu promenljiva k *kontroliše tip dozvoljenih puteva,* i *polazni čvor, a* j *krajnji čvor puta. Ovaj algoritam odgovara tome da se za svaka dva fiksirana čvora numerisana vrednostima* i *i* j *prolazi kroz sve čvorove* k *i razmatra da li postoji put preko čvora* k. *Za vrednost promenljivih* $i = 0$ *i* $j = 1$ *proveravaju se redom vrednosti* $0, 1, 2$ *i* 3 *za* k *i pošto ne postoji* k *tako da istovremeno postoji i grana* (i, k) *i grana* (k, j), *put od čvora* 0 *do čvora* 1 *ne bi bio pronađen, iako on postoji u grafu. Naime, da bismo otkrili najkraći put (tj. najkraći* 4*-put) od čvora* 0 *do čvora* 1 *potrebno je prethodno odrediti najkraći* 3*-put od čvora* 0 *do čvora* 1, *što u ovoj varijanti algoritma nije slučaj. Zaključujemo da ovakav redosled petlji ne omogućava nalaženje svih najkraćih puteva. Dakle, važno je da spoljašnja od tri petlje prolazi skupom vrednosti* k, *gde* k *kontroliše tip dozvoljnog puta.*



Slika 2.88: Usmereni težinski graf za koji modifikacija Flojd-Varšalovog algoritma kod koje unutrašnja petlja kontroliše tip dozvoljenih puteva ne vraća najkraći put od čvora 0 do čvora 1.

Primetimo da Flojd-Varšalov algoritam radi korektno i za grafove koji imaju negativne težine grana (sve dok u grafu ne postoji ciklus negativne težine) jer korektnost algoritma ne zavisi od toga da su težine grana u grafu nenegativne.

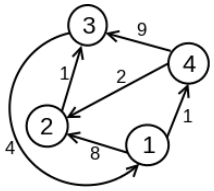
Ukoliko polazni graf ima ciklus negativne težine, to se može utvrditi tako što će nakon izvršavanja Flojd-Varšalovog algoritma dužina najkraćeg puta od nekog čvora do njega samog biti manja od 0, odnosno neka vrednost na dijagonali matrice `najkraciPut` je manja od 0. Naime, inicijalno važi `najkraciPut[i][i] = 0` za svako i. Flojd-Varšalov algoritam računa najkraće puteve između svaka dva čvora u grafu, pa i između parova istih čvorova. Ako čvor i pripada ciklusu negativne težine, onda će Flojd-Varšalov algoritam razmotriti i put od čvora i do njega samog kroz grane ovog ciklusa i inicijalnu vrednost 0 smanjiti na težinu ovog ciklusa.

Ako se u nekom koraku spoljašnje petlje ustanovi da se matrica `najkraciPut` nije promenila, algoritam se može ranije prekinuti.

Za svaku vrednost k algoritam izvršava jedno sabiranje i jedno upoređivanje za svaki par čvorova. Broj koraka indukcije je $|V|$, pa je ukupan broj sabiranja, odnosno upoređivanja, najviše $|V|^3$. Prisetimo se da je vremenska složenost Dajkstrinog algoritma za nalaženje najkraćih puteva od jednog zadatog čvora u grafu koji ne sadrži grane negativne dužine $O((|V|+|E|)\log|V|)$. Ako je graf gust, pa je broj grana $\Theta(|V|^2)$, onda je za određivanje svih najkraćih puteva u grafu Flojd-Varšalov algoritam efikasniji od izvršavanja Dajkstrinog algoritma od svakog polaznog čvora u grafu. S druge strane, ako graf nije gust (pa ima, na primer, $O(|V|)$ grana), onda je bolja vremenska složenost $O(|V|(|V|+|E|)\log|V|)$ koja potiče od $|V|$ puta upotrebljenog algoritma za najkraće puteve od jednog čvora. Jedna od prednosti Flojd-Varšalovog algoritma je, svakako, i njegova jednostavna realizacija, kao i to što je primenljiv i na grafove koji imaju negativne grane.

Primer 2.11.3

Razmotrimo postupak određivanja najkraćih puteva između svaka dva čvora u grafu sa slike 2.89.



Slika 2.89: Usmereni težinski graf za koji tražimo najkraći put između svaka dva čvora.

On se sastoji iz narednih koraka:

$k=0$: *najkraciPut:*
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{pmatrix} \end{array}$$
put:
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & - & 1 \\ - & 0 & 2 & - \\ 3 & - & 0 & - \\ - & 4 & 4 & 0 \end{pmatrix} \end{array}$$

$k=1$: *najkraciPut:*
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{pmatrix} \end{array}$$
put:
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & - & 1 \\ - & 0 & 2 & - \\ 3 & 1 & 0 & 1 \\ - & 4 & 4 & 0 \end{pmatrix} \end{array}$$

$k=2$: *najkraciPut:*
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{pmatrix} \end{array}$$
put:
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 2 & 1 \\ - & 0 & 2 & - \\ 3 & 1 & 0 & 1 \\ - & 4 & 2 & 0 \end{pmatrix} \end{array}$$

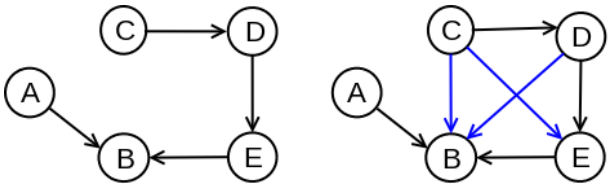
$k = 3$: najkraciPut:
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \end{array} \qquad \text{put:} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 2 & 1 \\ 3 & 0 & 2 & 3 \\ 3 & 1 & 0 & 1 \\ 3 & 4 & 2 & 0 \end{pmatrix} \end{array}$$

$k = 4$: najkraciPut:
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{pmatrix} \end{array} \qquad \text{put:} \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 4 & 4 & 1 \\ 3 & 0 & 2 & 3 \\ 3 & 4 & 0 & 1 \\ 3 & 4 & 2 & 0 \end{pmatrix} \end{array}$$

Iz poslednje matrice najkraciPut *možemo pročitati da je najkraći put od čvora* 1 *do čvora* 3 *dužine* 4*, a iz matrice* put *da je maksimalni indeks čvora na tom najkraćem putu jednak* 4*. Dakle, da bismo rekonstruisali najkraći put, potrebno je da na najkraći put od čvora* 1 *do čvora* 4 *nadovežemo najkraći put od čvora* 4 *do čvora* 3*. Iz matrice* put *možemo pročitati da je najveći indeks čvora na najkraćem putu od čvora* 1 *do čvora* 4 *baš jednak* 1*, što nam govori da je taj put direktna grana između tih čvorova. Iz matrice* put *možemo takođe pročitati da je najveći indeks čvora na najkraćem putu od čvora* 4 *do čvora* 3 *jednak* 2*, što nam govori da taj put određujemo tako što na najkraći put od čvora* 4 *do čvora* 2 *nadovežemo najkraći put od čvora* 2 *do čvora* 3*. Iz matrice* put *možemo zaključiti da oba ova puta odgovaraju direktnim granama. Konačno, zaključujemo da je najkraći put od čvora* 1 *do čvora* 3 *jednak* $(1, 4, 2, 3)$.

2.12 Tranzitivno zatvorenje i tranzitivna redukcija

Za zadati usmereni graf $G = (V, E)$ njegovo *tranzitivno zatvorenje* (engl. transitive closure) je usmereni graf $G^* = (V, E^*)$ u kome postoji grana od čvora u do čvora v ako i samo ako u grafu G postoji usmereni put od čvora u do čvora v. Dakle, skupu grana E treba dodati što manji broj grana, tako da dobijeni novi skup grana E^* određuje relaciju koja je tranzitivna. Na slici 2.90 prikazan je jedan usmeren graf i njegovo tranzitivno zatvorenje.



Slika 2.90: Graf i njegovo tranzitivno zatvorenje: plavom bojom istaknute su grane kojima je polazni graf proširen kako bi se dobilo tranzitivno zatvorenje grafa.

Postoji veliki broj različitih primena tranzitivnog zatvorenja, pa je važno imati efikasan algoritam za njegovo nalaženje. Na primer, tabelu u programu za tabelarna izračunavanja (npr. Microsoft Excel) možemo predstaviti u vidu usmerenog grafa: polja tabele odgovaraju čvorovima, a grana od čvora koji odgovara polju a do čvora koji odgovara polju b postoji ako vrednost koja se računa u polju b zavisi od vrednosti polja a. Kada se izmeni neka od vrednosti u tabeli, potrebno je ažurirati vrednosti svih polja koje od nje zavise, odnosno svih čvorova koji su dostižni iz datog polja. Ta polja se mogu odrediti na osnovu tranzitivnog zatvorenja datog grafa. Dodatno, ta polja je potrebno ažurirati u topološkom redosledu čvorova.

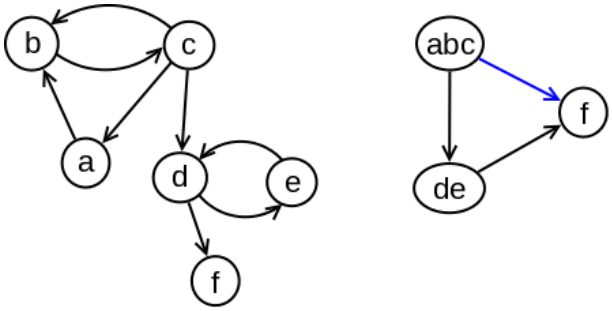
Slično, ako neki skup aerodroma razmotrimo kao skup čvorova, a postojanje direktnog leta od jednog do drugog aerodroma predstavimo usmerenom granom između odgovarajućih čvorova, onda nam tranzitivno zatvorenje grafa daje informaciju o tome sa kog aerodroma do kog aerodroma je moguće stići direktnim letom ili putem više povezanih letova.

Problem

Odrediti tranzitivno zatvorenje zadatog usmerenog grafa $G = (V, E)$.

Postoji više načina za računanje tranzitivnog zatvorenja datog grafa. Jedan način je da se iz svakog čvora pokrene pretraga u dubinu ili širinu i sačuva informacija o svim čvorovima dostižnim iz datog. Ovaj algoritam je vremenske složenosti $O(|V| \cdot (|V| + |E|))$ i ima dobre performanse ako je graf redak, dok za guste grafove postaje složenosti $O(|V|^3)$.

Ako je tranzitivno zatvorenje grafa G gust graf, efikasnije je najpre izračunati komponente jake povezanosti grafa G. Za svaka dva čvora iz iste komponente jake povezanosti važi da su međusobno dostižni, pa u tranzitivnom zatvorenju treba da budu povezani granama. Ako postoji grana (u, v) koja povezuje čvorove iz različitih jakih komponenti povezanosti, svaki čvor iz komponente kojoj pripada čvor v je dostižan iz svakog čvora komponente kojoj pripada čvor u i odgovarajuću granu treba dodati tranzitivnom zatvorenju grafa G. Dakle, problem se svodi na pronalaženje tranzitivnog zatvorenja komprimovanog grafa, sačinjenog od jakih komponenti povezanosti, koji obično ima dosta manje čvorova i grana (slika 2.91). Ako postoji K komponenti jake povezanosti, složenost određivanja tranzitivnog zatvorenja komprimovanog grafa je $O(K^3)$, što može biti znatno manje od $O(|V|^3)$ kada je K dovoljno manje od $|V|$.



Slika 2.91: Graf i tranzitivno zatvorenje njegovog komprimovanog grafa (plavom bojom označene su grane koje su dodate u graf).

Treći način da rešimo ovaj problem jeste redukcijom (svođenjem) na drugi problem. Ako granama grafa dodelimo proizvoljne težine (na primer, svakoj grani dodelimo težinu 1) i izračunamo najkraće puteve između svih parova čvorova tako dobijenog težinskog grafa, dužina najkraćeg puta između čvorova između kojih postoji put će biti konačna, dok će dužina najkraćeg puta između čvorova između kojih ne postoji put ostati $+\infty$. Dakle, problem određivanja tranzitivnog zatvorenja, što je u suštini problem ispitivanja dostižnosti između čvorova, prirodno se može svesti na određivanje dužine najkraćih puteva između svaka dva čvora.

Umesto da direktno primenimo algoritam za određivanje svih najkraćih puteva u grafu, možemo ga vremenski i prostorno optimizovati tako da direktno rešava problem tranzitivnog zatvorenja grafa. Primetimo da nas jedino interesuje da li je dužina najkraćeg puta u grafu G' konačna ili beskonačna (a u slučaju kada je konačna, ne interesuje nas njena konkretna vrednost, tj. da li je ona jednaka 10, 7 ili 2). Dakle, umesto da koristimo celobrojnu matricu u kojoj se pamte dužine najkraćih puteva između svaka dva čvora, možemo koristiti logičku matricu koja na poziciji (i, j) sadrži vrednost `true` ako je čvor j dostižan iz čvora i, a inače `false`. Takođe, umesto da koristimo aritmetičke operacije, možemo preći na logičke operacije: umesto operacije sabiranja dužina koristimo logičku konjunkciju (dužina puta koji se sastoji iz dva dela će biti konačna ako i samo ako su oba dela puta konačna).

```
// funkcija koja za svaka dva cvora utvrdjuje
// da li izmedju njih postoji put
void tranzitivnoZatvorenje(vector<vector<bool>> matricaPovezanosti) {
  // inicijalizujemo matricu tranzitivnog zatvorenja
  // na matricu povezanosti grafa
```

```
  vector<vector<bool>> zatvorenje = matricaPovezanosti;
  int brojCvorova = matricaPovezanosti.size();

  // proveravamo da li postoji put kroz cvor sa oznakom k
  for (int k = 0; k < brojCvorova; k++)
    for (int i = 0; i < brojCvorova; i++)
      for (int j = 0; j < brojCvorova; j++)
        if (zatvorenje[i][k] && zatvorenje[k][j])
          zatvorenje[i][j] = true;

  cout << "Matrica susedstva grafa koji"
       << " predstavlja tranzitivno zatvorenje je" << endl;
  for (int i = 0; i < brojCvorova; i++) {
    for (int j = 0; j < brojCvorova; j++)
      cout << zatvorenje[i][j] << " ";
    cout << endl;
  }
}

int main() {
  // broj cvorova u grafu
  int n;
  cin >> n;
  // matrica susedstva grafa koji sadrzi n cvorova
  vector<vector<bool>> M(n);
  // ucitavamo matricu susedstva
  for (int i = 0; i < n; i++) {
    M[i].resize(n);
    for (int j = 0; j < n; j++) {
      // realizujemo ucitavanje logickih vrednosti
      // citamo celobrojne vrednosti i konvertujemo ih u logicke
      int x;
      cin >> x;
      M[i][j] = x == 1;
    }
  }

  tranzitivnoZatvorenje(M);
  return 0;
}
```

Matrica `zatvorenje` na kraju izvršavanja algoritma kodira informacije o dostižnosti u polaznom grafu G, odnosno predstavlja skup grana u tranzitivnom zatvorenju grafa G.

Razmotrimo osnovni korak algoritma, naredbu `if`. Ona se sastoji od dve provere: da li važi `zatvorenje[i,k]` i da li važi `zatvorenje[k,j]`. Akcija se preduzima samo ako su oba uslova ispunjena. Ova `if` naredba izvršava se $|V|$ puta za svaki par čvorova. Svaka popravka ove naredbe vodila bi bitnoj popravci algoritma. Moraju li se svaki put proveravati oba uslova? Prva provera zavisi samo od vrednosti i i k, a druga zavisi samo od vrednosti k i j. Zbog toga se prva provera može za fiksirane vrednosti i i k izvršiti samo jednom (umesto $|V|$ puta). Naime,

- ako prvi uslov nije ispunjen, onda se drugi ne mora proveravati ni za jednu vrednost j,

- ako je pak prvi uslov ispunjen, onda se njegova ispunjenost ne mora ponovo proveravati za svaku vrednost j.

Ova promena je ugrađena u poboljšani algoritam čiji je ključni fragment dat u nastavku. Asimptotska složenost algoritma ostaje nepromenjena, ali se algoritam u proseku izvršava dva puta brže.

```
// proveravamo da li postoji put kroz cvor sa oznakom k
for (int k = 0; k < brojCvorova; k++)
  for (int i = 0; i < brojCvorova; i++)
    if (zatvorenje[i][k])
      for (int j = 0; j < brojCvorova; j++)
        if (zatvorenje[k][j])
          zatvorenje[i][j] = true;
```

Ovaj algoritam može se dalje usavršiti. Linija

```
if (zatvorenje[k][j])
   zatvorenje[i][j] = true;
```

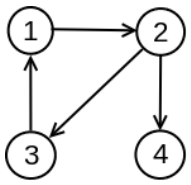
može se ekvivalentno zameniti linijom

```
zatvorenje[i][j] = zatvorenje[i][j] | zatvorenje[k][j];
```

Zapaža se da se posle ove zamene u unutrašnjoj petlji algoritma operacija `or` primenjuje na vrstu i i vrstu k matrice `zatvorenje`, a rezultat je nova vrsta i. Zbog toga, ako je $n = |V| \leq 64$, vrste matrice `zatvorenje` predstavljaju niz nula i jedinica i mogu se tumačiti kao n bitova u binarnoj reprezentacija celog broja, pa se primena operacije `or` na vrste može zameniti bitskom `or` operacijom dva cela broja, što je n puta brže. Ako je $n > 64$, onda se vrsta može predstaviti nizom 64-bitnih celih brojeva, pa se algoritam izvršava približno 64 puta brže. Asimptotska složenost je i dalje $O(|V|^3)$, ali ubrzanje za faktor 64 nije zanemarljivo.

Primer 2.12.1

Razmotrimo primer izračunavanja tranzitivnog zatvorenja grafa sa slike 2.92.



Slika 2.92: Primer grafa za koji je potrebno odrediti tranzitivno zatvorenje.

On se sastoji iz narednih koraka:

$$k=0\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \qquad k=1\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \qquad k=2\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

$$k=3\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \qquad k=4\text{:}\ \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Primetimo da u grafu sa slike 2.92 ne postoji direktna grana od čvora 3 *do čvora* 2*, ali postoji put dužine dva preko čvora* 1*. Slično, od čvora* 3 *do čvora* 4 *ne postoji direktna grana, ali postoji put dužine* 3*, preko čvorova* 1 *i* 2*.*

Čitaocima se ostavlja za razmišljanje pitanje kako bi izgledalo tranzitivno zatvorenje neusmerenog grafa.

Interesantan problem u vezi sa nalaženjem tranzitivnog zatvorenja je i problem određivanja *tranzitivne redukcije* grafa (engl. transitive reduction). Ova operacija predstavlja operaciju inverznu operaciji pronalaženja tranzitivnog zatvorenja grafa: cilj je da se za dati usmereni graf konstruiše usmereni graf sa istim skupom čvorova i što manjim skupom grana, a da se pritom ne promeni relacija dostižnosti. Tranzitivno zatvorenje grafa G jednako je tranzitivnom zatvorenju tranzitivne redukcije grafa G. Iako je tranzitivno zatvorenje grafa G jedinstveno određeno, u opštem slučaju graf može imati više različitih tranzitivnih redukcija.

3. Algebarski algoritmi

Uvek kada izvršavamo neku algebarsku operaciju, kao što je recimo množenje dva broja ili njihovo stepenovanje, mi, u stvari, izvršavamo neki algoritam. Takve operacije koristimo kao gradivne elemente prilikom izvođenja složenijih algoritama i često ne zalazimo dublje u analizu njihove složenosti. Međutim, i sami algoritmi sabiranja, oduzimanja, množenja, deljenja i stepenovanja brojeva (posebno ako su brojevi dati nizovima svojih cifara) predstavljaju važne algebarske algoritme. U algebarske algoritme spadaju i mnogi algoritmi sa kojima smo se ranije susretali, kao što su izračunavanje vrednosti broja na osnovu datih cifara (u nekoj osnovi) ili, nasuprot tome, određivanje cifara broja na osnovu njegove vrednosti, računanje najvećeg zajedničkog delioca dva data broja, zatim razni algoritmi nad polinomima kao što su izračunavanje vrednosti polinoma i množenje polinoma. U ovom poglavlju bavićemo se algebarskim algoritmima sa kojima se do sada nismo susreli. Mnogi od njih igraju važnu ulogu u oblasti kriptografije, ali i u drugim oblastima, poput algebarske teorije brojeva i kvantnog računarstva.

Algebarski algoritmi koji se razmatraju u ovom materijalu rade sa prirodnim brojevima i u velikoj meri se zasnivaju na osobinama deljivosti prirodnih brojeva (npr. pojmu prostog broja).

U ovom poglavlju bavićemo se temama iz teorije brojeva, poput računanja najvećeg zajedničkog delioca dva broja, faktorizacije broja, svojstvima i načinima računanja multiplikativnih funkcija, kao i osnovama modularne aritmetike. Uvedene koncepte i algoritme ćemo ilustrovati na primeru algoritma RSA, koji ima primenu u kriptografiji. Konačno, predstavićemo algoritam FFT koji predstavlja jedan od najznačajnih algoritama današnjice i ilustrovati kako se on može primeniti na brzo množenje polinoma.

3.1 Euklidov algoritam

Ako su a i b celi brojevi i važi $b \neq 0$, kažemo da je broj q *količnik*, a broj r *ostatak* pri deljenju broja a brojem b ako i samo ako važi $a = q \cdot b + r$ i $0 \leq r < y$. Brojevi q i r su ovim uslovom jedinstveno određeni. Pisaćemo $q = a \operatorname{div} b$ i $r = a \bmod b$. U programskim jezicima se operacija mod obično označava simbolom `%`, dok je div celobrojno deljenje uz zaokruživanje naniže (što je podrazumevano ponašanje operatora `/` primenjenog na cele brojeve), odnosno $a \operatorname{div} b = \left\lfloor \frac{a}{b} \right\rfloor$. Neki programski jezici imaju poseban operator za izračunavanje celobrojnog količnika (npr. operator `//` u jeziku Python). Treba biti obazriv jer različiti programski jezici drugačije tretiraju slučajeve kada je neki od brojeva a ili b negativan (u nekim jezicima operator `%` izračunava pozitivan, a u nekim negativan ostatak, što je u suprotnosti sa definicijom

ostatka koju smo naveli).

Broj a je *deljiv* brojem $b \neq 0$, što označavamo $b|a$, ako i samo ako je $a \bmod b = 0$. Broj b je *delilac* broja a, a broj a je *sadržalac* broja b.

Brojevi koji imaju tačno dva različita delioca nazivaju se *prosti* – $n > 1$ je prost ako su mu jedini delioci 1 i n. Prirodni brojevi koji imaju više od dva različita delioca su *složeni*. Broj 1 nije ni prost ni složen. Prostih brojeva ima beskonačno mnogo. Svi prosti brojevi manji od datog broja n se mogu efikasno odrediti algoritmom poznatim pod imenom *Eratostenovo sito*.

Najveći zajednički delilac brojeva a i b je najveći broj d takav da $d|a$ i $d|b$, a *najmanji zajednički sadržalac* brojeva a i b je najmanji broj s tako da $a|s$ i $b|s$. Podsetimo se osnovnog Euklidovog algoritma za određivanje najvećeg zajedničkog delioca brojeva a i b.

3.1.1 Osnovni Euklidov algoritam

Razmotrimo problem pronalaženja NZD dva broja.

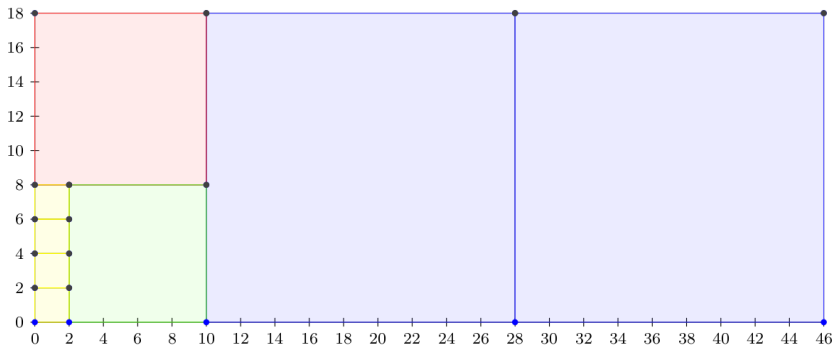
Problem

Za date nenegativne cele brojeve a i b koji nisu istovremeno jednaki nuli izračunati $\text{nzd}(a, b)$.

Razmatramo čuveni *Euklidov algoritam*. Algoritam se može ilustrovati i geometrijski i u direktnoj je vezi sa problemom određivanja maksimalne dimenzije kvadrata kojima može da se poploča pravougaonik čije su dužine stranica a i b.

Primer 3.1.1

Neka je dat pravougaonik dimenzije $a = 46$ i $b = 18$, prikazan na slici 3.1. Tada se prvo iz njega mogu seći dva kvadrata dimenzije 18×18 i ostaje nam pravougaonik dimenzije 18×10. Ukoliko nekim manjim kvadratima uspemo da popločamo taj preostali pravougaonik, tim kvadratima ćemo moći da popločamo i ove kvadrate dimenzije 18×18 (jer će dimenzija tih malih kvadrata deliti broj 18 jednak jednoj dimenziji preostalog pravougaonika), pa ćemo samim tim moći da popločamo i ceo polazni pravougaonik dimenzija 46×18. Od pravougaonika dimenzije 18×10 možemo seći kvadrat dimenzije 10×10 i preostaje nam pravougaonik dimenzije 10×8. Ponovo, kvadratići kojima se može popločati taj preostali pravougaonik biće takvi da se njima može popločati i isečeni kvadrat dimenzije 10×10. Od tog pravougaonika isecamo kvadrat dimenzije 8×8 i dobijamo pravougaonik dimenzije 8×2. Njega možemo razložiti na četiri kvadrata dimenzije 2×2 i to je najveća dimenzija kvadrata kojima se može popločati polazni pravougaonik.



Slika 3.1: Popločavanje pravougaonika kvadratima.

Implementacija koja direktno prati prethodnu definiciju bila bi rekurzivna, a moguće je jednostavno napraviti i iterativnu implementaciju, u kojoj se u svakom koraku vrednost većeg od dva broja zamenjuje njihovom razlikom, sve dok se brojevi ne izjednače.

Rekurzivno se algortam može izraziti na sledeći način.

```
int nzd(int a, int b) {
   if (a > b) return nzd(a-b, b);
   if (a < b) return nzd(a, b-a);
   return a;
}
```

Rekurzija se može jako lako ukloniti.

```
int nzd(int a, int b) {
   while (a != b) {
      if (a > b)
         a -= b;
      else
         b -= a;
   }
   return a;
}
```

Dokažimo formalno korektnost algoritma (dokazujemo samo korektnost rekurzivne varijante, pošto joj je iterativna varijanta očigledno ekvivalentna).

Teorema 3.1.1 **[Korektnost Euklidovog algoritma sa oduzimanjem]**

Za date prirodne brojeve a i b, koji nisu istovremeno jednaki nuli, Euklidov algoritam sa oduzimanjem korektno izračunava njihov NZD.

Dokaz. Korektnost rekurzivne implementacije se jednostavno može dokazati indukcijom.

Bazu čini slučaj $a = b$. Pošto su brojevi različiti od nule, očigledno je da je njihov NZD jednak $a = b$.

Pretpostavimo da je $a > b$. Tada je $\text{nzd}(a, b) = \text{nzd}(a, a - b)$. Zaista, ako neki broj d deli brojeve a i b, tada on deli i $a - b$. Dakle, $d = \text{nzd}(a, b)$ sigurno deli i b i $a - b$. Ako on ne bi bio NZD brojeva $a - b$ i b, tada bi postojao neki veći broj d' koji bi delio i $a - b$ i b. Međutim, tada bi d' delio i zbir $a = (a - b) + b$, pa bi d' bio i delilac brojeva a i b, koji je veći od d, što je kontradikcija sa tim da je d NZD brojeva a i b. Na osnovu induktivne hipoteze $\text{nzd}(a, a - b)$ umemo ispravno da izračunamo.

Slučaj $a < b$ je potpuno simetričan. □

Ovaj se algoritam lako može optimizovati. Razmotrimo određivanje NZD dva broja na jednom primeru.

Primer 3.1.2

$\text{nzd}(279, 45) = \text{nzd}(234, 45) = \text{nzd}(189, 45) = \text{nzd}(144, 45) = \text{nzd}(99, 45) = \text{nzd}(54, 45) = \text{nzd}(9, 45) = \text{nzd}(9, 36) = \text{nzd}(9, 27) = \text{nzd}(9, 18) = \text{nzd}(9, 9) = 9.$

Možemo primetiti dugačak niz koraka u kojima se malo po malo od broja 279 oduzima broj 45, sve dok se ne dođe do broja koji je manji od broja 45. Za tim nema potrebe, jer znamo da će se posle tog dugog niza koraka postupak zaustaviti kada nam jedan argument bude baš 45, a drugi bude jednak ostatku pri deljenju broja 279 brojem 45, a to je broj 9. Dakle, umesto da iterativno taj ostatak računamo uzastopnim oduzimanjima, bolji pristup je da primenimo deljenje i u jednom koraku ga izračunamo kao ostatak pri deljenju. Ako sličan princip primenimo na brojeve 9 i 45, doći ćemo do toga da će nam ostati broj 9 i ostatak pri deljenju ta dva broja, što je nula. To nije baš u potpunosti jednako kao u slučaju oduzimanja, gde smo se zaustavili kod para $(9, 9)$, međutim, sasvim je ispravno i može se smatrati produženjem prethodnog postupka u kom bi se pre

prijavljivanja rezultata uradilo još jedno oduzimanje i došlo se do toga da je jedan od brojeva jednak nuli, kada je NZD jednak drugom broju.

Brži Euklidov algoritam, u kom se koristi deljenje, zasnovan je na sledećim tvrđenjima.

- za $a \neq 0$ važi $\text{nzd}(a, 0) = a$.
- za $b \neq 0$ važi $\text{nzd}(a, b) = \text{nzd}(b, a \bmod b)$.

Primetimo da nema potrebe analizirati koji je broj manji, a koji je veći. Ako je $a < b$, tada važi $a \bmod b = a$, pa se u prvom koraku dobija da je $\text{nzd}(a, b) = \text{nzd}(b, a)$. Pošto je $a \bmod b < b$, jednom kada je prvi argument veći od drugog, to će tako ostati do kraja.

I Euklidov algoritam sa deljenjem, dakle, dopušta veoma jednostavnu rekurzivnu implementaciju.

```
int nzd(int a, int b) {
   if (b == 0)
      return a;
   return nzd(b, a % b);
}
```

Implementaciju Euklidovog algoritma možemo izvršiti iterativno, tako što u petlji koja se izvršava sve dok je broj b veći od nule par promenljivih (a, b) zamenjujemo vrednostima $(b, a \bmod b)$. Na kraju petlje je $b = 0$, tako da kao rezultat možemo prijaviti tekuću vrednost broja a.

```
int nzd(int a, int b) {
   while (b != 0) {
      int ost = a % b;
      a = b;
      b = ost;
   }
   return a;
}
```

Dokažimo formalno korektnost algoritma (dokazujemo samo korektnost rekurzivne varijante, pošto joj je iterativna varijanta očigledno ekvivalentna).

Teorema 3.1.2 **[Korektnost Euklidovog algoritma sa deljenjem]**

Za date prirodne brojeve a i b, koji nisu istovremeno jednaki nuli, Euklidov algoritam sa deljenjem korektno izračunava njihov NZD.

Dokaz. Korektnost rekurzivne varijante se dokazuje matematičkom indukcijom.

Bazu čini slučaj $\text{nzd}(a, 0)$, za $a \neq 0$. Pošto važi $a|0$ i $a|a$, a ne postoji ni jedan broj $a' > a$ takav da $a'|a$, važi da je $\text{nzd}(a, 0) = a$.

Na osnovu definicije celobrojnog deljenja važi $a = (a \operatorname{div} b) \cdot b + (a \bmod b)$. Obeležimo sa d NZD brojeva b i $a \bmod b$. Da bismo dokazali da je on ujedno NZD brojeva a i b dovoljno je dokazati da on deli ta dva broja i da svaki broj koji deli ta dva broja deli njega.

- Pošto broj d deli brojeve b i $a \bmod b$, on deli oba sabirka na desnoj strani, pa zato deli i njihov zbir koji je jednak a i zato deli i a i b.
- Dalje, ako neki broj d' deli brojeve a i b, on mora deliti i broj $a \bmod b$ (jer se on može iskazati kao razlika dva broja deljivih brojem d'), pa pošto je d' delilac brojeva b i $a \bmod b$, on mora deliti i njihov NZD, tj. mora deliti broj d.

Pošto nzd$(b, a \bmod b)$ možemo izračunati na osnovu induktivne hipoteze, tvrđenje je dokazano. □

Euklidov algoritam koji je zasnovan na deljenju možemo predstaviti i na sledeći način:

$$\begin{array}{rcll} r_0 & = & a & \\ r_1 & = & b & \\ r_2 & = & r_0 \bmod r_1 = r_0 - q_1 r_1, & 0 < r_2 < r_1 \\ \ldots & & & \\ r_{i+1} & = & r_{i-1} \bmod r_i = r_{i-1} - q_i r_i, & 0 < r_{i+1} < r_i \\ \ldots & & & \\ r_k & = & r_{k-2} \bmod r_{k-1} = r_{k-2} - q_{k-1} r_{k-1}, & 0 < r_k < r_{k-1} \\ r_{k+1} & = & r_{k-1} \bmod r_k = r_{k-1} - q_k r_k, & r_{k+1} = 0 \end{array}$$

Vrednost r_k je NZD brojeva a i b. Zaista, pošto je $r_{k+1} = 0$, važi $r_{k-1} = q_k r_k$, pa broj r_k deli r_{k-1}. Međutim, pošto je $r_{k-2} = q_{k-1} r_{k-1} + r_k$, r_k deli i r_{k-2}. Sličnim rezonovanjem, unazad, može se zaključiti da r_k deli i r_1 i r_0, tj. a i b. Obratno, ako neki broj deli i a i b onda on deli i r_0 i r_1, a pošto je $r_2 = r_0 - q_1 r_1$, on deli i r_2. Sličnim rezonovanjem, unapred, može se zaključiti da taj broj mora deliti i r_k. Stoga je r_k NZD brojeva a i b.

Dakle, polazeći od brojeva $r_0 = a$ i $r_1 = b$, izračunava se ostatak $r_2 = r_0 \bmod r_1$, zatim ostatak $r_3 = r_1 \bmod r_2$, itd. Na taj način dobija se opadajući niz ostataka r_i za koji važi:

$$r_{i-1} = q_i r_i + r_{i+1}, \quad 0 \leq r_{i+1} < r_i, \text{ za } i = 1, 2, \ldots, k \tag{3.1}$$

Pri deljenju r_{i-1} sa r_i količnik je q_i, a ostatak r_{i+1}. Dobijeni niz r_i je konačan jer je opadajući (važi $r_{i+1} < r_i$ za svako i), a sastoji se od prirodnih brojeva. Ako je $r_{k+1} = 0$ i $r_k \neq 0$ poslednji član ovog niza različit od nule, kako je

$$\text{nzd}(r_0, r_1) = \text{nzd}(r_1, r_2) = \ldots = \text{nzd}(r_{k-1}, r_k) = \text{nzd}(r_k, 0) = r_k,$$

zaključujemo da je $d = \text{nzd}(a, b)$ upravo jednako r_k, poslednjem ostatku u nizu ostataka koji je različit od nule.

Što se tiče implementacije, u svakom koraku algoritma održavamo dve uzastopne vrednosti niza ostataka. U početku, to su članovi r_0 i r_1 tj. originalne vrednosti a i b. Važi da je $q_1 = a \operatorname{div} b$, a $r_2 = a \bmod b$. U drugom koraku promenljive a i b treba da imaju vrednosti članova r_1 i r_2, što znači da se par promenljivih a, b zamenjuje vrednostima b i $a \bmod b$. Postupak se nastavlja sve dok par uzastopnih vrednosti ne postane r_k, r_{k+1}, tj. pošto par uzastopnih vrednosti održavamo u promenljivama a i b, dok b ne postane nula i tada je NZD koji je jednak r_k sadržan u promenljivoj a.

Primetimo da se posle najviše jednog koraka osigurava da je $a > b$ (jer se u svakom koraku par (a, b) zamenjuje parom $(b, a \bmod b)$, a uvek važi da je $a \bmod b < b$). Posle bilo koje dve iteracije se od para (a, b) dolazi do para $(a \bmod b, b \bmod (a \bmod b))$ (naravno, pod pretpostavkom da je $b \neq 0$ i da je $a \bmod b \neq 0$). Dokažimo da je $a \bmod b < a/2$. Ako je $b \leq a/2$, tada je $a \bmod b < b \leq a/2$. U suprotnom, za $b > a/2$ važi da je $a \bmod b = a - b < a/2$. Zato se prvi argument posle svaka dva koraka smanjuje bar dvostruko. Do vrednosti 1 prvi argument stiže u logaritamskom broju koraka u odnosu na veći od polazna dva broja i tada drugi broj sigurno dostiže nulu (jer je strogo manji od prvog) i postupak se završava. Dakle, složenost je logaritamska u odnosu na veći od dva broja, odnosno $O(\log(a + b))$. Ako se složenost računa u odnosu na broj cifara broja (što je veličina ulaza), onda je ona zapravo linearna. Ovo ukazuje na to da je problem određivanja NZD veoma lak problem i da se efikasno može rešiti i za ogromne brojeve (brojeve i sa nekoliko hiljada cifara).

3.1.2 Prošireni Euklidov algoritam

Uz malu dopunu, Euklidov algoritam se može iskoristiti i za rešavanje sledećeg problema.

Problem

Najveći zajednički delilac d dva prirodna broja a i b izraziti kao njihovu celobrojnu linearnu kombinaciju. Drugim rečima, odrediti cele brojeve x i y, tako da važi $d = \text{nzd}(a, b) = x \cdot a + y \cdot b$.

Primer 3.1.3

Najveći zajednički delilac brojeva $a = 10$ i $b = 18$ je $d = 2$ i on se može predstaviti kao njihova celobrojna linearna kombinacija na sledeći način: $2 = 2 \cdot 10 - 1 \cdot 18$.

Brojevi x i y postoje na osnovu tvrđenja koje je poznato kao *Bezuov stav*, a koje ćemo u nastavku konstruktivno dokazati na dva različita načina (iz kojih će proizaći dva algoritma za izračunavanje koeficijenata x i y).

Teorema 3.1.3 **[Bezuov stav]**

Ako su a i b dva prirodna broja (ne istovremeno jednaka nuli) i važi da je $\text{nzd}((, a), b) = d$, tada postoje celi brojevi x i y takvi da je $x \cdot a + y \cdot b = d$.

3.1.2.1 Predstavljanje nzd preko uzastopnih ostataka

Pošto je $r_0 = a$ i $r_1 = b$, problem možemo formulisati nešto drugačije: zadatak je broj $d = \text{nzd}(a, b)$ izraziti u obliku celobrojne linearne kombinacije ostataka r_0 i r_1, odnosno kao $d = r_k = x \cdot r_0 + y \cdot r_1$, gde je r_k poslednji član niza ostataka različit od nule. Ovako formulisan problem se može rešiti indukcijom.

Polazi se od baznog slučaja koji odgovara predstavljanju broja d kao celobrojne linearne kombinacije dva uzastopna ostatka r_{k-2} i r_{k-1}: $d = r_k = r_{k-2} - q_{k-1} r_{k-1}$. Ovaj izraz je ekvivalentan pretposlednjem deljenju u Euklidovom algoritmu (izraz (3.1)) za $i = k - 1$.

Korak indukcije bio bi da se polazeći od izraza za d

$$d = x' \cdot r_i + y' \cdot r_{i+1} \tag{3.2}$$

kao celobrojne linearne kombinacije ostataka r_i i r_{i+1}, broj d izrazi kao celobrojna linearna kombinacija ostataka r_{i-1} i r_i, odnosno u obliku $d = x'' \cdot r_{i-1} + y'' \cdot r_i$. Ovo se postiže zamenom vrednosti r_{i+1} u jednakosti (3.2) sa $r_{i-1} - q_i r_i$ iz jednakosti (3.1). Time se dobija

$$d = x' \cdot r_i + y' \cdot r_{i+1} = x' \cdot r_i + y' \cdot (r_{i-1} - q_i r_i) = y' \cdot r_{i-1} + (x' - q_i y') \cdot r_i \tag{3.3}$$

odnosno:

$$\begin{aligned} x'' &= y' \\ y'' &= x' - q_i y' \end{aligned}$$

tj. izrazili smo d u obliku celobrojne linearne kombinacije ostataka r_{i-1} i r_i. Dakle, indukcijom po i, $i = k - 2, k - 3, \ldots, 1$, dokazano je da se d može izraziti kao celobrojna linearna kombinacija bilo koja dva uzastopna člana r_i i r_{i+1} niza ostataka. Specijalno, za $i = 0$, dobija se traženi izraz. Ova verzija Euklidovog algoritma se ponekad naziva *prošireni Euklidov algoritam* (engl. extended Euclidean algorithm).

Primer 3.1.4

Potrebno je odrediti cele brojeve x i y tako da važi $3 = 33x + 24y$. S obzirom na to da je $\text{nzd}(33, 24) = 3$, *možemo iskoristiti prethodni postupak za određivanje brojeva x i y. Prilikom izračunavanja najvećeg zajedničkog delioca brojeva 33 i 24 imamo naredni niz jednakosti:* $\text{nzd}(33, 24) = \text{nzd}(24, 9) = \text{nzd}(9, 6) = \text{nzd}(6, 3) = \text{nzd}(3, 0) = 3$. *Prateći unazad niz deljenja sa ostatkom dobijamo:* $d = 3 = 9 - 6 = 9 - (24 - 2 \cdot 9) = 3 \cdot 9 - 24 = 3 \cdot (33 - 24) - 24 = 3 \cdot 33 - 4 \cdot 24$, *odnosno* $x = 3, y = -4$.

Prethodna induktivna konstrukcija pogodna je za rekurzivnu implementaciju jer nove vrednosti promenljivih x i y dobijamo na osnovu prethodnih. U narednoj implementaciji, vrednost koju funkcija vraća jeste najveći zajednički delilac datih brojeva, dok vrednosti koeficijenata x i y vraćamo preko referenci, kroz listu parametara funkcije.

Primetimo da je baza indukcije u implementaciji slučaj $d = r_k = 1 \cdot r_k + 0 \cdot r_{k+1}$.

```
// funkcija koja vraca nzd brojeva a i b i racuna koeficijente x i y
int nzdProsireni(int a, int b, int &x, int &y) {
  // ako je b jednako 0, onda je nzd(a, b) = a
  if (b == 0) {
    x = 1;
    y = 0;
    return a;
  }

  int x1, y1;
  // rekurzivno resavamo problem za naredna dva elementa u nizu ostataka,
  // a to su brojevi b i a%b
  int nzd = nzdProsireni(b, a % b, x1, y1);
  // azuriramo koeficijente
  x = y1;
  y = x1 - (a / b) * y1;
  return nzd;
}
```

Primetimo da se u ovoj verziji algoritama dva puta prolazi kroz količnike i ostatke. U prolazu unapred se određuje vrednost NZD, a zatim se u prolazu unazad određuju koeficijenti (u rekurzivnoj implementaciji taj drugi prolaz se izvršava tek nakon što se stigne do izlaza iz rekurzije). Zato se ovaj algoritam teže implementira nerekurzivno (i memorijska složenost mu je, kao i vremenska, $\log(a + b)$, jer zahteva istovremeno čuvanje više stek okvira).

3.1.2.2 Predstavljanje ostataka preko a i b

U prethodnom izvođenju smo sve vreme poslednji ostatak $d = r_k$ predstavljali kao celobrojnu linearnu kombinaciju dva susedna elementa iz niza ostataka počev od r_{k-1} i r_{k-2}. Alternativno, moguće je jednu po jednu vrednost iz niza ostataka počev od r_0 predstaviti kao celobrojnu linearnu kombinaciju brojeva a i b i, na kraju, i broj r_k predstaviti na ovaj način. Naime, pošto je $r_0 = a$ i $r_1 = b$, važi:

$$\begin{aligned} r_0 &= 1 \cdot a + 0 \cdot b \\ r_1 &= 0 \cdot a + 1 \cdot b \end{aligned}$$

odnosno za $r_0 = a$ važi $x_0 = 1$ i $y_0 = 0$, a za $r_1 = b$ važi $x_1 = 0$ i $y_1 = 1$.

Želimo da r_{i+1} predstavimo kao celobrojnu linearnu kombinaciju brojeva a i b, ako znamo da važi:

$$\begin{aligned} r_{i-1} &= x_{i-1} \cdot a + y_{i-1} \cdot b \\ r_i &= x_i \cdot a + y_i \cdot b \end{aligned}$$

Na osnovu jednakosti (3.1) tj. veze $r_{i+1} = r_{i-1} - q_i \cdot r_i$ dobijamo:

$$r_{i+1} = (x_{i-1} \cdot a + y_{i-1} \cdot b) - q_i(x_i \cdot a + y_i \cdot b) = (x_{i-1} - q_i x_i) \cdot a + (y_{i-1} - q_i y_i) \cdot b$$

Dakle, vrednosti x_{i+1} i y_{i+1} možemo izračunati na osnovu prethodnih vrednosti x_{i-1} i x_i, odnosno y_{i-1} i y_i, tj. važi naredna veza:

$$\begin{aligned} x_{i+1} &= x_{i-1} - q_i x_i \\ y_{i+1} &= y_{i-1} - q_i y_i \end{aligned}$$

Na osnovu ovih veza možemo doći do naredne C++ implementacije iterativnog algoritma za rešavanje polaznog problema.

```
// funkcija koja vraca nzd brojeva a i b i racuna koeficijente x i y
int nzdProsireni(int a, int b, int &x, int &y) {
  // vrednost x i y za r_0 = a
  int x_preth = 1;
  int y_preth = 0;
  // vrednost x i y za r_1 = b
  int x_tek = 0;
  int y_tek = 1;

  while (b != 0) {
    // azuriramo vrednosti za a i b
    // kao u standardnom Euklidovom algoritmu
    int q = a/b;
    int r = a - q * b;
    a = b;
    b = r;

    // azuriramo tekucu i prethodnu vrednost niza x
    int xpom = x_preth - q * x_tek;
    x_preth = x_tek;
    x_tek = xpom;

    // azuriramo tekucu i prethodnu vrednost niza y
    int ypom = y_preth - q * y_tek;
    y_preth = y_tek;
    y_tek = ypom;
  }

  // postavljamo konacne vrednosti za x i y
  // kao vrednosti x i y za r_k
  x = x_preth;
  y = y_preth;
```

```
  // vracamo a kao nzd brojeva
  return a;
}
```

Primer 3.1.5

Primenimo algoritam na brojeve 33 *i* 24.

$$
\begin{array}{lllllllllllll}
33 & & & & & & & & & & = & 1\cdot 33 & + & 0\cdot 24 \\
24 & & & & & & & & & & = & 0\cdot 33 & + & 1\cdot 24 \\
9 & = & 33 & - & 1\cdot 24 & = & (1\cdot 33+0\cdot 24) & - & 1\cdot(0\cdot 33+1\cdot 24) & = & 1\cdot 33 & - & 1\cdot 24 \\
6 & = & 24 & - & 2\cdot 9 & = & (0\cdot 33+1\cdot 24) & - & 2\cdot(1\cdot 33-1\cdot 24) & = & -2\cdot 33 & + & 3\cdot 24 \\
3 & = & 9 & - & 1\cdot 6 & = & (1\cdot 33-1\cdot 24) & - & 1\cdot(-2\cdot 33+3\cdot 24) & = & 3\cdot 33 & - & 4\cdot 24 \\
0 & = & 6 & - & 2\cdot 3 & & & & & & & & &
\end{array}
$$

Veze između (x_{i+1}, y_{i+1}) i (x_i, y_i) su bile izažene na način pogodan implementaciji (naredni koeficijenti su bili izraženi preko prethodniih). Primetimo da se ove veze mogu izraziti i na sledeći način, tako da je oblik veza isti i za količnike i ostatke r i za koeficijente x i y:

$$
\begin{array}{lcl}
r_{i-1} & = & q_i r_i + r_{i+1} \\
x_{i-1} & = & q_i x_i + x_{i+1} \\
y_{i-1} & = & q_i y_i + y_{i+1}
\end{array}
$$

Pri tom u svakoj koloni u prethodnim jednakostima važi veza istog oblika:

$$
\begin{array}{lcl}
r_{i-1} & = & x_{i-1}a + y_{i-1}b \\
r_i & = & x_i a + y_i b \\
r_{i+1} & = & x_{i+1}a + y_{i+1}b
\end{array}
$$

Broj operacija koje se izvršavaju u proširenom Euklidovom algoritmu proporcionalan je broju operacija u standardnom Euklidovom algoritmu, tj. iznosi $O(\log(a+b))$, pa je i ovo izrazito efikasan algoritam. Primetimo i da se u ovoj varijanti algoritma vrši samo jedan prolaz, pa se nakon određivanja narednih vrednosti koeficijenata, prethodne mogu zaboraviti. Memorijska složenost je, stoga, konstantna.

Iz prethodno opisanih razloga ova (jednoprolazna) varijanta algoritma je poželjnija od prethodne (dvoprolazne), pa stoga neki autori pod nazivom *prošireni Euklidov algoritam* podrazumevaju samo ovu varijantu.

3.1.3 Rešavanje linearnih Diofantovih jednačina

Jedna od primena Bezuovog stava i proširenog Euklidovog algoritma je u rešavanju jedne klase Diofantovih jednačina, tzv. *linearnih Diofantovih jednačina* koje su oblika $a \cdot x + b \cdot y = c$.

Problem

Za date nenegativne cele brojeve a *i* b *(koji nisu istovremeno jednaki nuli) i nenegativan ceo broj* c *odrediti cele brojeve* x *i* y *tako da važi* $a \cdot x + b \cdot y = c$.

Bezuov stav daje jedno rešenje jednačine $ax + by = d$, za $d = \text{nzd}(a, b)$. Naredni stav govori o strukturi svih rešenja.

Lema 3.1.1 **[Sva rešenja Bezuovog identiteta]**

Ako je (x_0, y_0) jedno rešenje jednačine $ax + by = d$, gde je $d = \text{nzd}(a, b)$, onda su sva rešenja oblika $x = x_0 - k\frac{b}{d}$, $y = y_0 + k\frac{a}{d}$, za $k \in \mathbb{Z}$. Postoje tačno dva "mala" para rešenja, za koje je $|x| \leq |b/d|$ i $|y| \leq |a/d|$. Jednakost važi ako i samo ako je jedan od brojeva a ili b umnožak onog drugog.

Dokaz. Pošto je $ax_0 + by_0 = d = ax + by$, važi $a(x_0 - x) = b(y - y_0)$. Ova se veza može podeliti sa d i i dobija se $\frac{a}{d}(x_0 - x) = \frac{b}{d}(y - y_0)$. Brojevi $\frac{a}{d}$ i $\frac{b}{d}$ su uzajamno prosti pa mora da postoji ceo broj k takav da je $x_0 - x = k\frac{b}{d}$, a $y - y_0 = k\frac{a}{d}$. Odatle se dobija i opšte rešenje:

$$(x, y) = \left(x_0 - k\frac{b}{d}, y_0 + k\frac{a}{d}\right)$$

"Mali" parovi rešenja se dobijaju tako što se za k uzme bilo koji od dva cela broja koji su najbliži vrednosti $x_0\frac{d}{b}$. □

Napomenimo da prošireni Euklidov algoritam uvek vraća jedan od dva "mala" para rešenja o kojima se govori u prethodnoj lemi.

Primer 3.1.6

Razmotrimo sada jednačinu $4x + 10y = 2$. Važi $d = \text{nzd}(4, 10) = 2$, pa korišćenjem proširenog Euklidovog algoritma najveći zajednički delilac $d = 2$ brojeva $a = 4$ i $b = 10$ izražavamo kao njihovu celobrojnu linearnu kombinaciju: $x_0a + y_0b = d$, tj. $-2 \cdot 4 + 1 \cdot 10 = 2$. Ostala rešenja jednačine su oblika $x = x_0 - k \cdot b/d = -2 - 5k$ i $y = y_0 + k \cdot a/d = 1 + 2k$. Zaista, važi $4x + 10y = 4(-2 - 5k) + 10(1 + 2k) = (-8 + 10) + (20k - 20k) = 2$. "Mala" rešenja su ona za koja važi $|x| \leq |b/d| = 5$ i $|y| \leq |a/d| = 2$ i to su jedino $(-2, 1)$ i $(3, -1)$.

Teorema 3.1.4 **[Postojanje rešenja linearne diofantske jednačine]**

Da bi jednačina $a \cdot x + b \cdot y = c$ imala bar jedno rešenje, potrebno je i dovoljno da $d = \text{nzd}(a, b)$ deli c.

Dokaz. Naime, pošto d deli levu stranu jednačine, mora da deli i desnu, pa je ovaj uslov potreban. Dokažimo da je i dovoljan, tako što ćemo opisati postupak konstrukcije rešenja. Naime, ako je ovaj uslov ispunjen, jedno od rešenja ove jednačine lako se dobija narednim postupkom: najpre se d izrazi u obliku $d = x' \cdot a + y' \cdot b$ a zatim se množenjem leve i desne strane jednakosti celim brojem c/d dobija:

$$c = \frac{x'c}{d} \cdot a + \frac{y'c}{d} \cdot b$$

tj. jedno rešenje polazne jednačine predstavlja par

$$(x_0, y_0) = \left(\frac{x'c}{d}, \frac{y'c}{d}\right)$$

Struktura svih rešenja sledi iz leme 3.1.1, tj. sva rešenja su opisana parom:

$$(x, y) = \left(\frac{x'c - kb}{d}, \frac{y'c + ka}{d}\right)$$

□

Primer 3.1.7

Razmotrimo jednačinu $4x + 10y = 7$. *Važi da je* $\text{nzd}(4, 2) = 2$, *međutim* 2 *ne deli desnu stranu jednakosti, tj. broj* 7, *te ova Diofantova jednačina nema rešenja.*

Primer 3.1.8

Razmotrimo sada jednačinu $4x+10y = 8$. *Dakle,* $a = 4, b = 10$ *i* $c = 8$. *Važi* $d = \text{nzd}(4, 10) = 2$ *i* $2 \mid 8$, *pa pošto* $d \mid c$, *ova jednačina ima rešenja. Jedno njeno rešenje možemo dobiti na sledeći način: korišćenjem proširenog Euklidovog algoritma najveći zajednički delilac brojeva* $a = 4$ *i* $b = 10$ *izražavamo kao njihovu celobrojnu linearnu kombinaciju:* $x'a + y'b = d$, *tj.* $-2 \cdot 4 + 1 \cdot 10 = 2$, *a zatim obe strane jednakosti množimo brojem* $c/d = 8/2 = 4$ *čime dobijamo:* $x_0a + y_0b = c$, *tj.* $-8 \cdot 4 + 4 \cdot 10 = 8$. *Dakle, jedno rešenje ove jednačine je* $x_0 = -8$ *i* $y_0 = 4$. *Ostala rešenja su oblika* $x = x_0 - k \cdot b/d = -8 - 5k$ *i* $y = y_0+k\cdot a/d = 4+2k$. *Zaista, važi* $4x+10y = 4(-8-5k)+10(4+2k) = (-32+40)+(20k-20k) = 8$.

Dalje primene proširenog Euklidovog algoritma biće prikazane u narednim poglavljima (u poglavlju 3.4 ovaj algoritam se primenjuje na problem određivanja modularnog inverza, a u poglavlju 3.6 na rešavanje sistema kongruencija primenom Kineske teoreme o ostacima).

3.2 Rastavljanje na proste činioce (faktorizacija)

Problem

Dat je broj n. *Rastaviti ga na proste činioce.*

Na primer, za $n = 315$, dobija se razlaganje $315 = 3 \cdot 3 \cdot 5 \cdot 7$.

Kao što ćemo videti nešto kasnije, faktorizacija brojeva igra važnu ulogu u oblasti kriptografije. Međutim, ona može biti od koristi i za rešavanje nekih standardnih matematičkih problema, poput računanja najvećeg zajedničkog delioca (bez Euklidovog algoritma) ili najmanjeg zajedničkog sadržaoca dva broja, za izračunavanje kvadratnog korena savršenih kvadrata itd. U nastavku ćemo opisati osnovni algoritam za faktorizaciju brojeva zasnovan na pokušajima deljenja potencijalnim činiocima (eng. trial division), koji je složenosti $O(\sqrt{n})$. Postoje i mnogi efikasniji algoritmi (na primer, Polardov ρ, kvadratno sito, Fermaova i Ojlerova faktorizacija itd.), međutim, i sa njima problem faktorizacije ostaje težak problem, čije velike instance (npr. brojeve sa nekoliko hiljada cifara) ni najsavremeniji računari ne mogu nikako rešiti.

Problem faktorizacije broja n možemo rešavati induktivno-rekurzivnim pristupom na sledeći način. Pretpostavimo da sve brojeve manje od n umemo da rastavimo na proste činioce. Ako nekako ustanovimo da je broj n prost, onda je jedini prost činilac broja n on sam i problem je rešen. Ako n nije prost broj, onda se on može predstaviti kao proizvod $n = d \cdot n_1$, gde je d najmanji od svih činilaca broja n (veći od 1). Najmanji činilac d broja n mora biti prost broj manji ili jednak $\sqrt{n}$.

Lema 3.2.1 **[Najmanji činilac broja]**

Ako je n *prirodan broj koji nije prost, njegov najmanji činilac* $d > 1$ *je prost broj i važi* $d \leq \sqrt{n}$.

Dokaz. Najmanji činilac d broja n mora biti prost broj, jer ako bi bio složen tj. ako bi važilo $d = d_1 \cdot d_2$, brojevi d_1 i d_2 (veći od 1) bi bili činioci broja n još manji od d. Dodatno, važi $d \leq \sqrt{n}$. Naime, ako pretpostavimo suprotno – da je $d > \sqrt{n}$, pošto je $n = d \cdot n_1$, onda bi važilo $n_1 < \sqrt{n} < d$, te bi najmanji činilac broja n_1 bio ujedno i najmanji činilac broja n, suprotno pretpostavci da je d najmanji činilac broja n. Dakle, važi $d \leq \sqrt{n}$. □

Dakle, da bi se pronašao najmanji prost činilac broja n, dovoljno je pronaći najmanji činilac broja n i proveriti da li je on manji ili jednak $\sqrt{n}$. Ovo se može uraditi prolaskom kroz skup prirodnih brojeva redom od 2 do $\sqrt{n}$ (redosled nam garantuje pronalaženje najmanjeg činioca). Nalaženjem najmanjeg prostog činioca d broja n, problem se svodi na faktorizaciju broja n_1, čije proste činioce prema induktivnoj hipotezi umemo da odredimo.

Ako ne postoji činilac $d \leq \sqrt{n}$ broja n, tada je broj n prost i on je svoj jedini prost činilac.

Na osnovu ove konstrukcije jednostavno je formulisati rekurzivni algoritam.

Iterativna varijanta ovog rekurzivnog algoritma sastoji se iz narednih koraka: prolazi se skupom brojeva od $d = 2$ do $\sqrt{n}$ i ako je n deljivo brojem d, sve dok je n deljivo sa d pamtimo ili štampamo vrednost d (to su prosti činioci) i postavljamo vrednost n na n/d. Nakon toga, ako je $n > 1$, n je prost i on je prost činilac polaznog broja.

S obzirom na to da nijedan prost broj veći od 2 nije paran, efikasnije je zasebno razmatrati broj 2, a zatim proći skupom neparnih brojeva od 3 do $\sqrt{n}$.

```
// funkcija koja ispisuje sve proste cinioce broja n
vector<int> prostiCinioci(int n) {
  // vektor u koji upisujemo rezultat
  vector<int> cinioci;
  // ako je n parno, uzimamo cinilac 2 onoliki broj puta koliko puta deli n
  while (n % 2 == 0) {
    cinioci.push_back(2);
    // azuriramo n
    n /= 2;
  }
  // n je neparno
  // prolazimo kroz neparne brojeve
  for (int d = 3; d*d <= n; d += 2) {
    // sve dok d deli n, uzimamo cinilac d
    while (n % d == 0) {
      cinioci.push_back(d);
      // azuriramo n
      n /= d;
    }
  }

  // ako je n veci od 1, on je prost
  if (n > 1)
    cinioci.push_back(n);

  return cinioci;
}
```

Invarijanta spoljašnje `for` petlje algoritma čija je implementacija u C++ prikazana je to da je u svakom koraku proizvod do tada određenih činilaca i trenutne vrednosti promenljive n jednak polaznom broju n, kao i da trenutna vrednost n nije deljiva ni sa jednim brojem koji je manji ili jednak d. Kada se spoljna petlja završi, pošto n nije deljiv ni sa jednim brojem manjim ili jednakim od svog korena, n mora biti prost, pa ako je veći od 1, on je najveći prost činilac polaznog broja.

Istaknimo jednu važnu stvar: nigde u programu se posebno ne ispituje da li su ispisani činioci prosti brojevi. Na prvi pogled ovo može delovati zbunjujuće, međutim redosled kojim tražimo činioce broja n nam garan-

tuje da će svaki ispisani delilac d biti prost. Naime, na osnovu invarijante znamo da kada se ispiše vrednost d, trenutna vrednost promenljive n nije deljiva nijednim brojem manjim od d, a ako bi d bio složen tj. ako bi važilo $d = d_1 \cdot d_2$, n bi bio deljiv sa d_1 i d_2 koji su veći od 1, a manji od d, što je na osnovu invarijante nemoguće.

Pošto se petlja `for` izvršava do korena iz n (koje je sve vreme manje ili jednako polaznoj vrednosti broja $n_0 \equiv n$) možemo zaključiti da je složenost algoritma $O(\sqrt{n_0})$. Primetimo da se za složene brojeve granica $\sqrt{n}$ petlje `for` smanjuje sa svakim novim pronađenim činiocem (jer će nova vrednost za n biti strogo manja od stare), pa u nekim slučajevima algoritam može izvršiti i dosta manje koraka od $\sqrt{n_0}$. Ako su prosti činioci (ne nužno različiti) broja n_0 redom jednaki $p_1 \leq p_2 \leq \ldots \leq p_{k-1} \leq p_k$, može se lako pokazati da je složenost ovog algoritma za faktorizaciju $O(\max(p_{k-1}, \sqrt{p_k}))$ (primetimo da je u pitanju izlazno-zavisna složenost, jer složenost zavisi od rezultata). Pri tom pretpostavljamo da radimo sa mašinskim tipovima podataka, pa se unutrašnja petlja kojom se vrši deljenje izvršava uvek mali broj puta (zbir svih stepena u faktorizaciji je ograničen brojem bitova u zapisu broja).

Primer 3.2.1

Ilustrujmo složenost na nekoliko različitih primera:

- *ako je $n = 71 \cdot 73$, prvi (a ujedno i pretposlednji) prost činilac broja n je jednak $p_{k-1} = 71$ i njega bismo odredili kada promenljiva i `for` petlje stigne do 71, nakon čega vrednost promenljive n postaje 73, a pošto granica za i ide do $\lfloor \sqrt{n} \rfloor = \lfloor \sqrt{73} \rfloor = 8$, ovde se prekida `for` petlja. Dakle, kada su brojevi p_{k-1} i p_k bliski po vrednosti, broj iteracija petlje biće jednak $O(p_{k-1})$.*
- *ako je $n = 2 \cdot 73$, prvi (a ujedno i pretposlednji) prost činilac broja n je jednak $p_{k-1} = 2$ i njega bismo odredili u startu – kada i ima vrednost 2, nakon čega vrednost promenljive n postaje 73, a pošto granica za i ide do $\lfloor \sqrt{n} \rfloor = \lfloor \sqrt{73} \rfloor = 8$, `for` petlja bi se izvršavala $O(\sqrt{p_k})$ puta. Dakle, kada je p_{k-1} dosta manje od p_k, broj iteracija petlje biće jednak $O(\sqrt{p_k})$.*
- *ako je $n = 73$, broj je prost i `for` petlja se izvršava $O(\sqrt{n})$ puta, a jedini prost činilac broja n bi bio pronađen naknadno, nakon izlaska iz `for` petlje.*

Dakle, važi sledeće: nije jednako teško faktorisati sve brojeve iste dužine. Najteže instance ovog problema, ako se primenjuje opisani algoritam, čine brojevi koji su prosti ili su proizvodi dva približno jednaka prosta broja. Naime, kada su oba ova prosta činioca velika i slične veličine, ovaj problem postaje jako teško rešiti. Kao ilustraciju ovoga, navedimo da se primenom veoma efikasnog algoritma faktorizacije, 2009. godine okončao dvogodišnji napor grupe istraživača da se faktoriše broj od 232 cifre korišćenjem više stotina računara. Pretpostavljena težina ovog problema je osnova za procenu sigurnosti velikog broja kriptografskih algoritama, kao što je RSA algoritam (videti poglavlje 3.5).

Naglasimo i da je to što su za bazu indukcije uzeti svi prosti brojevi dovela do algoritma složenosti $O(\sqrt{n})$. Ako bi se za bazu indukcije uzeo broj 1 tj. ako bi se činioci redom proveravali sve dok se broj deljenjem ne svede na vrednost 1, umesto da se stane kada se pređe $\sqrt{n}$ (kako se često radi kada se ručno određuju prosti činioci), dobio bi se algoritam složenosti $O(n)$.

3.2.1 Fermaov algoritam faktorizacije

Kada broj n ima faktore blizu vrednosti $\sqrt{n}$, oni se vrlo efikasno mogu naći Fermaovim algoritmom, zasnovanim na tome da se svaki neparan prirodan broj može napisati kao razlika dva potpuna kvadrata $n = a^2 - b^2$. Zaista, ako je $n = n_1 \cdot n_2$, tada je

$$n = \left(\frac{n_1 + n_2}{2}\right)^2 - \left(\frac{n_1 - n_2}{2}\right)^2,$$

pri čemu su n_1 i n_2 neparni (jer je n neparan), pa su kvadrati celobrojni.

Ako je $n = a^2 - b^2$, tada se on može rastaviti na činioce $(a - b) \cdot (a + b)$. Ako je $a - b$ različito od 1, tada smo pronašli dva činioca, koji se dalje mogu faktorisati.

Algoritam se zasniva na isprobavanju raznih vrednosti a, krenuvši od $\lceil \sqrt{n} \rceil$ i ispitivanju da li je $b^2 = a^2 - n$ potpun kvadrat (tj. kvadrat celog broja b). U nekom trenutku b^2 mora biti potpun kvadrat. Zaista, ako je $n = n_1 \cdot n_2$, u nekom trenutku će a dostići vrednost $(n_1 + n_2)/2$. Ako je n prost, to će se desiti tek kada se dostigne $(n+1)/2$, što je prilično neefikasno, međutim, ako postoji činilac broja n blizak vrednosti $\sqrt{n}$, on će se brzo pronaći (jer pretraga kreće baš od vrednosti a bliske broju $\sqrt{n}$, a ako postoje vrednosti n_1 i n_2 bliske broju $\sqrt{n}$, i njihova aritmetička sredina će mu biti bliska i brzo će biti pronađena). Prilikom provere da li je b^2 potpun kvadrat puno brojeva mogu biti odbačeni na osnovu jednostavnih testova (na primer, poslednja cifra tj. ostatak pri deljenju sa 10 potpunog kvadrata mora biti 0, 1, 4, 5, 6, ili 9, a slični kriterijumi se mogu formulisati i za druge brojevne osnove).

Primer 3.2.2

Faktorišimo broj 5959. Važi $\lceil \sqrt{5959} \rceil = 78$*, pa algoritam kreće od* $a = 78$.

Korak	1	2	3
a	78	79	80
$b^2 = a^2 - n$	125	282	441
b	-	-	21

Dakle, u samo tri koraka se pronalazi da je $5959 = 80^2 - 21^2$*, što daje faktore* $a - b = 89 - 21 = 59$ *i* $a + b = 80 + 21 = 101$. *Oni su dosta manji od polaznog broja i njihova faktorizacija teče brže.*

3.2.2 Faktorizacija većeg broja brojeva

Nekad je potrebno veliki broj puta izvršiti faktorizaciju broja.

Problem

Za dati broj n *pronaći faktorizaciju većeg broja brojeva iz intervala* $[1, n]$ *(na primer izvršiti faktorizaciju svih brojeva koji su manji ili jednaki* n*).*

Moguće je $O(n)$ puta izvršiti ispočetka osnovni algoritam faktorizacije i ukupna složenost ovog pristupa iznosi $O(n\sqrt{n})$.

Ključni korak razmotrenog algoritma za faktorizaciju je pronalaženje najmanjeg (prostog) činioca tekućeg broja. Kada faktorišemo više brojeva doći ćemo u situaciju da više puta za isti broj određujemo najmanji prosti činilac. Na primer, prilikom faktorisanja broja 221 odredićemo da je njegov najmanji prost činilac 13, a prilikom faktorisanja broja 442 odredićemo da je njegov najmanji prost činilac 2, međutim, tada ćemo broj 442 podeliti sa 2 i ponovo doći u situaciju da treba da određujemo najmanji prost činilac broja 221, što smo već jednom uradili. Dakle, do efikasnijeg rešenja dolazi se ako primenimo tehniku dinamičkog programiranja tj. memoizacije i konstruišemo pomoćni niz dužine n koji za svako $k \leq n$ sadrži najmanji prost činilac broja k. Naime, kad bismo znali vrednost najmanjeg prostog činioca svakog broja k za $k \leq n$, broj m bismo mogli faktorisati na sledeći način: štampamo vrednost p_1 najmanjeg prostog činioca broja m, zatim vrednost p_2 najmanjeg prostog činioca broja m/p_1, zatim vrednost najmanjeg prostog činioca broja $m/(p_1 \cdot p_2)$, ... Dakle, problem faktorizacije brojeva od 1 do n možemo svesti na problem efikasnog izračunavanja najmanjeg prostog činioca svih brojeva od 1 do n i smeštanje ovih vrednosti u odgovarajući niz.

Niz koji za svaki prirodan broj manji ili jednak n sadrži vrednost njegovog najmanjeg prostog činioca, može se dobiti malim prilagođavanjem Eratostenovog sita. Podsetimo se, Eratostenovo sito se koristi da se odrede

svi prosti brojevi do datog broja n, tako što se precrtaju svi umnošci broja 2, zatim broja 3, zatim broja 5 i tako redom, sve dok se ne precrtaju i svi umnošci broja $\lfloor\sqrt{n}\rfloor$. Neprecrtani brojevi su svi traženi prosti brojevi.

Ovaj se algoritam lako prilagođava tako da određuje najmanji prost faktor svakog broja do n. Inicijalno za svaki broj i postavljamo da je njegov najmanji prost činilac baš i. U originalnom Eratostenovom situ se prilikom prolaska kroz umnoške nekog prostog broja d označava da su ti umnošci složeni (za šta se koristi niz logičkih vrednosti). Umesto toga, u ovom algoritmu ćemo prilikom prolaske kroz umnoške nekog prostog broja, svim umnošcima kojima nije ranije umanjena vrednost najmanjeg prostog činioca, postaviti tu vrednost na d. Za broj d zaključujemo da je prost, ako je njegov najmanji prost činilac on sam. Prilikom razmatranja prostog broja d, možemo da krenemo od d^2 jer su brojevi $2d, 3d, \ldots, (d-1)d$ deljivi redom sa $2, 3, \ldots, d-1$, te sigurno imaju činilac manji od d.

Kada se prva faza završi i popuni niz najmanjih prostih činilaca, tada se u drugoj fazi popunjeni niz može upotrebiti za brzu faktorizaciju bilo kog broja od 1 do n.

Primer 3.2.3

Neka je potrebno faktorisati sve brojeve koji su manji ili jednaki 50.

- *Krećemo od niza u kome je na poziciji* i *upisana vrednost* i.

```
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

- *U prvom koraku svim parnim brojevima postavljamo da je najmanji prost činilac jednak* 2*:*

```
 1  2  3  2  5  2  7  2  9  2
11  2 13  2 15  2 17  2 19  2
21  2 23  2 25  2 27  2 29  2
31  2 33  2 35  2 37  2 39  2
41  2 43  2 45  2 47  2 49  2
```

- *Umnošcima broja* 3 *počev od* 9 *koji nisu deljivi sa* 2 *postavljamo najmanji prost činilac na* 3*: to su* 9*,* 15*,* 21*,* 27*,* 33*,* 39 *i* 45.

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2 25  2  3  2 29  2
31  2  3  2 35  2 37  2  3  2
41  2 43  2  3  2 47  2 49  2
```

- *Razmatramo broj* 4*. Zaključujemo da je on složen jer je njegov najmanji prost činilac postavljen na vrednost manju od* 4*, te broj* 4 *preskačemo.*

- *Umnošcima broja* 5 *počev od* 25*, koji nisu deljivi nekim manjim prostim brojem (*2 *i* 3*), postavljamo da je najmanji prost činilac jednak* 5*. To će biti brojevi* 25 *i* 35.

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2  5  2  3  2 29  2
31  2  3  2  5  2 37  2  3  2
41  2 43  2  3  2 47  2 49  2
```

- *Broj* 6 *kao složen preskačemo.*

- *Umnošcima broja* 7 *počev od* 49*, koji nisu deljivi sa* 2*,* 3 *niti* 5*, postavljamo da je* 7 *njihov najmanji prost činilac (to je samo broj* 49*).*

```
 1  2  3  2  5  2  7  2  3  2
11  2 13  2  3  2 17  2 19  2
 3  2 23  2  5  2  3  2 29  2
31  2  3  2  5  2 37  2  3  2
41  2 43  2  3  2 47  2  7  2
```

- *Time se obrada završava jer je* $8^2 \geq 50$*.*

Na primer, prilikom faktorizacije broja 48 *ispisuju se redom vrednosti* $2, 2, 2, 2$ *i* 3 *kao vrednosti najmanjih činilaca redom brojeva* 48*,* $48/2 = 24$*,* $24/2 = 12$*,* $12/2 = 6$ *i* $6/2 = 3$*.*

U nastavku je data implementacija algoritma faktorizacije svih brojeva od 1 do n korišćenjem Eratostenovog sita.

```
vector<int> eratosten(int n) {
  // niz koji cemo popuniti tako da se na poziciji i
  // nalazi najmanji prost cinilac broja i
  vector<int> najmanjiCinilac(n + 1);

  // postavljamo da je najmanji prost cinilac svakog broja sam taj broj
  for (int i = 1; i <= n; i++)
    najmanjiCinilac[i] = i;
  for (int d = 2; d * d <= n; d++)
    // ako je broj d prost, tj njegov najmanji prost činilac je on sam
    if (najmanjiCinilac[d] == d)
      // prolazimo kroz skup svih umnozaka tog broja
      for (int i = d * d; i <= n; i += d)
        // ako nije ranije azurirana vrednost, odnosno
        // postavljena neka manja vrednost za cinilac
        if (najmanjiCinilac[i] == i)
          // postavljamo da je najmanji prost cinilac broj d
          najmanjiCinilac[i] = d;
  return najmanjiCinilac;
}

// funkcija koja ispisuje sve proste cinioce broja n
void ispisiProsteCinioce(int n, const vector<int>& najmanjiCinilac) {
  while (n != 1) {
    // stampamo najmanji prost cinilac broja n
    cout << najmanjiCinilac[n] << " ";
    // azuriramo vrednost n
    n /= najmanjiCinilac[n];
  }
  cout << endl;
}

int main() {
  int n;
  cout << "Unesite broj n" << endl;
  cin >> n;
```

```
  auto najmanjiCinilac = eratosten(n);
  for (int i = 1; i <= n; i++)
    ispisiProsteCinioce(i, najmanjiCinilac);
  return 0;
}
```

Složenost prvog koraka algoritma – određivanja najmanjeg prostog činioca svih brojeva do n jednaka je složenosti algoritma Eratostenovo sito koja je, ako pretpostavimo da je složenost sabiranja $O(1)$, jednaka sumi n/p po prostim brojevima p, što je $O(n \log \log n)$.[1] Složenost ispisivanja prostih činilaca broja k iznosi $O(\log k)$ jer je maksimalni broj prostih činilaca broja k jednak $\log_2 k$, a pretpostavljamo da je deljenje činiocima konstantne vremenske složenosti.[2] Pošto treba ispisati proste činioce svih brojeva od 1 do n, ukupno vreme potrebno za to jednako je $\log 1 + \log 2 + \ldots + \log n$ što je, prisetimo se, jednako $\Theta(n \log n)$. Dakle, složenost prikazanog algoritma za faktorizaciju brojeva od 1 do n je jednaka $O(n \log \log n) + O(n \log n) = O(n \log n)$, što je značajno efikasnije od pristupa zasnovanog na nezavisnoj faktorizaciji svakog broja posebno.

3.3 Multiplikativne funkcije

Jedna od važnih primena faktorizacije broja je efikasno izračunavanje nekih funkcija nad prirodnim brojevima. Za funkciju $f : \mathbb{N} \to \mathbb{N}_0$ kažemo da je *multiplikativna* ako i samo ako za svaka dva uzajamno prosta broja a i b (dva broja za koje je $\text{nzd}(a, b) = 1$) važi da je $f(a \cdot b) = f(a) \cdot f(b)$. Kada se broj n faktoriše u proizvod $p_1^{k_1} p_2^{k_2} \ldots p_m^{k_m}$, činioci $p_1^{k_1}, p_2^{k_2}, \ldots, p_m^{k_m}$ su uzajamno prosti pa, ako je f multiplikativna funkcija, onda važi:

$$f(n) = f(p_1^{k_1}) \cdot f(p_2^{k_2}) \cdot \ldots \cdot f(p_m^{k_m}).$$

Dakle, da bi se mogla odrediti vrednost multiplikativne funkcije za proizvoljni broj n potrebno je odrediti vrednost funkcije za proizvoljni stepen prostog broja tj. za brojeve obilka p^k, što je često mnogo jednostavnije i može se rešiti analitički, pronalaženjem izraza kojim se ta vrednost izračunava. Naglasimo da je u opštem slučaju $f(p^k) \neq f(p)^k$, jer stepeni prostog broja p nisu međusobno uzajamno prosti (imaju zajednički delilac p).

Ako je vremenska složenost izračunavanja vrednosti $f(p^k)$ jednaka $O(1)$, što je često slučaj, onda vremenska složenost izračunavanja vrednosti $f(n)$ odgovara vremenu potrebnom za faktorizaciju broja n što je $O(\sqrt{n})$.

Ako je potrebno izračunati vrednosti nekih multiplikativnih funkcija za više ulaznih parametara (na primer za sve vrednosti od 1 do n), tada je moguće upotrebiti faktorizaciju pomoću Eratostenovog sita, koja je efikasnija od faktorizacije svakog broja pojedinačno.

U nastavku će biti prikazano izračunavanje nekoliko važnih multiplikativnih funkcija.

3.3.1 Ojlerova funkcija

Razmotrimo naredni problem: potrebno je odrediti broj nesvodljivih razlomaka u intervalu [0,1] čiji je imenilac jednak datom broju n. To su svi oni razlomci $\frac{m}{n}$, $m \leq n$ za koje važi da su m i n uzajamno prosti. Zadatak se dakle svodi na prebrojavanje koliko ima prirodnih brojeva manjih ili jednakih n koji su uzajamno prosti sa n.

[1] Napomenimo da ako bi n moglo biti proizvoljno veliko (ako bi broj bio predstavljen reprezentacijom sa proizvoljnim brojem bitova), onda bi složenost sabiranja bila $O(\log n)$, međutim u konkretnim implementacijama oslanjamo se na sabiranje brojeva ograničenog opsega koje je hardverski podržano i za koje smatramo da je složenosti $O(1)$.

[2] Ako bi broj n mogao biti proizvoljno veliki, onda bi složenost deljenja bila $O(\log^2 n)$.

Ojlerova funkcija (engl. Euler's totient function) φ broja n označava broj prirodnih brojeva manjih ili jednakih od n koji su uzajamno prosti sa n. Na primer, $\varphi(9) = 6$ jer postoji šest brojeva $1, 2, 4, 5, 7, 8$ manjih od 9 koji su uzajamno prosti sa 9, a $\varphi(17) = 16$ jer je broj 17 prost i uzajamno je prost sa svim brojevima strogo manjim od njega. Po definiciji je $\varphi(1) = 1$[3]. Označimo sa Φ_n skup svih brojeva koji su manji od n i uzajamno su prosti sa n. Na primer, $\Phi_9 = \{1, 2, 4, 5, 7, 8\}$. Važi da je $\varphi(n) = |\Phi_n|$.

Ojlerova funkcija ima primenu u teoriji brojeva, u algebri, ali, kao što ćemo videti, igra i ključnu ulogu u sistemu za šifrovanje RSA.

Problem

Za dati prirodan broj n izračunati vrednost Ojlerove funkcije $\varphi(n)$.

3.3.1.1 Direktan algoritam

Direktan način da se reši problem bio bi da se redom prođe kroz sve brojeve od 1 do $n-1$ i da se izbroji koliko je od njih uzajamno prosto sa n.

```
// funkcija koja racuna najveci zajednicki delilac dva broja
int nzd(int a, int b) {
  if (a == 0)
    return b;
  return nzd(b % a, a);
}

// funkcija koja racuna vrednost Ojlerove funkcije
int ojlerovaFunkcija(int n) {
  // 1 je uvek uzajamno prosto sa n
  int phi = 1;
  for (int i = 2; i < n; i++)
    if (nzd(i, n) == 1)
      phi++;
  return phi;
}
```

Prilikom računanja vrednosti Ojlerove funkcije, funkcija za određivanje najvećeg zajedničkog delioca dva broja se poziva $O(n)$ puta. Kako znamo da je složenost algoritma za izračunavanje najvećeg zajedničkog delioca brojeva i i n jednaka $O(\log(i+n))$ i kako je ovde uvek $i < n$, važi da je složenost računanja vrednosti $\text{nzd}(i, n)$ jednaka $O(\log n)$, te je ukupna složenost prikazanog algoritma za izračunavanje vrednosti Ojlerove funkcije broja n jednaka $O(n \log n)$.

3.3.1.2 Računanje Ojlerove funkcije broja svođenjem na faktorizaciju

Dokazaćemo da je Ojlerova funkcija multiplikativna, što omogućava njeno izračunavanje svođenjem na faktorizaciju.

Lema 3.3.1 **[Multiplikativnost Ojlerove funkcije φ]**

Ojlerova funkcija φ je multiplikativna, tj. ako su M i N uzajamno prosti brojevi, tada je $\varphi(M \cdot N) = \varphi(M) \cdot \varphi(N)$.

Dokaz. Postoji bijekcija između skupova $\Phi_M \times \Phi_N$ i Φ_{MN}. Bijekcija se uspostavlja tako što se svakom paru $(m, n) \in \Phi_M \times \Phi_N$ dodeli jedinstven broj $k \leq M \cdot N$ koji pri deljenju sa M daje ostatak m

[3] Nekad se Ojlerova funkcija broja n definiše kao broj prirodnih brojeva strogo manjih od n, što jedino utiče na vrednost Ojlerove funkcije broja 1.

(tj. važi $k \bmod M = m$), a pri deljenju sa N daje ostatak n (tj. važi $k \bmod N = n$). Taj broj postoji i jedinstven je na osnovu tzv. Kineske teoreme o ostacima (ona je opisana u poglavlju 3.6 Kineska teorema o ostacima). On mora biti uzajamno prost sa $M \cdot N$, pa pripada skupu Φ_{MN}. Zaista, na osnovu Euklidovog algoritma važi $\text{nzd}(k, M) = \text{nzd}(M, k \bmod M) = \text{nzd}(M, m) = 1$ i $\text{nzd}(k, N) = \text{nzd}(N, k \bmod N) = \text{nzd}(N, n) = 1$. Pošto je $\text{nzd}(k, M) = 1$ i $\text{nzd}(k, N) = 1$, mora da važi i $\text{nzd}(k, MN) = 1$.

Obratno, svakom broju $k \in \Phi_{MN}$ se može pridružiti par $(k \bmod M, k \bmod N) \in \Phi_M \times \Phi_N$. Zaista, ostaci su uvek manji od delioca, a $k \bmod M$ i $k \bmod N$ moraju biti uzajamno prosti sa M odnosno N (što se dokazuje analogno prethodnom smeru). Dakle, pošto je između dva konačna skupa uspostavljena bijekcija, njihovi brojevi elemenata su jednaki pa je

$$\varphi(MN) = |\Phi_{MN}| = |\Phi_M \times \Phi_N| = |\Phi_M| \cdot |\Phi_N| = \varphi(M) \cdot \varphi(N).$$

□

Primer 3.3.1

Razmotrimo brojeve $M = 5$ i $N = 6$. Uzajamno prosti sa $M = 5$ su elementi skupa $\Phi_5 = \{1, 2, 3, 4\}$, dok su uzajamno prosti sa N elementi skupa $\Phi_6 = \{1, 5\}$. Uzajamno prosti sa $MN = 30$ su elementi skupa $\Phi_{30} = \{1, 7, 11, 13, 17, 19, 23, 29\}$. Ostaci pri deljenju elemenata skupa Φ_{30} brojevima $M = 5$ i $N = 6$ su redom sledeći parovi: $\{(1, 1), (2, 1), (1, 5), (3, 1), (2, 5), (4, 1), (3, 5), (4, 5)\}$ i jasno se vidi da su to tačno svi parovi skupa $\Phi_5 \times \Phi_6$.

Pošto je Ojlerova funkcija multiplikativna, dovoljno je odrediti samo njene vrednosti za stepene prostih brojeva tj. za brojeve oblika p^k. Od svih brojeva manjih ili jednakih od p^k (kojih ima p^k) sa brojem p^k nisu uzajamno prosti samo umnošci broja p, odnosno brojevi $p, 2p, 3p, \ldots, p^{k-1}p$, kojih ima ukupno p^{k-1}. Dakle, važi:

$$\varphi(p^k) = p^k - p^{k-1} = p^k \left(1 - \frac{1}{p}\right)$$

U opštem slučaju je $n = p_1^{k_1} \cdot \ldots \cdot p_m^{k_m}$, gde su $p_1, p_2, \ldots, p_m$ različiti prosti činioci broja n, pa je

$$\begin{aligned}
\varphi(n) &= \varphi(p_1^{k_1})\varphi(p_2^{k_2}) \cdot \ldots \cdot \varphi(p_m^{k_m}) \\
&= p_1^{k_1}\left(1 - \frac{1}{p_1}\right) p_2^{k_2}\left(1 - \frac{1}{p_2}\right) \cdot \ldots \cdot p_m^{k_m}\left(1 - \frac{1}{p_m}\right) \\
&= p_1^{k_1} p_2^{k_2} \cdot \ldots \cdot p_m^{k_m}\left(1 - \frac{1}{p_1}\right)\left(1 - \frac{1}{p_2}\right) \cdot \ldots \cdot \left(1 - \frac{1}{p_m}\right) \\
&= n \cdot \prod_{\substack{p \mid n \\ p \text{ prost}}} \left(1 - \frac{1}{p}\right)
\end{aligned}$$

Dakle važi:

$$\varphi(n) = n \cdot \prod_{\substack{p \mid n \\ p \text{ prost}}} \left(1 - \frac{1}{p}\right) \tag{3.4}$$

Primer 3.3.2

- $\varphi(30) = \varphi(2^1 \cdot 3^1 \cdot 5^1) = 30(1 - \frac{1}{2})(1 - \frac{1}{3})(1 - \frac{1}{5}) = 30 \cdot \frac{1}{2} \cdot \frac{2}{3} \cdot \frac{4}{5} = 8$
- $\varphi(36) = \varphi(2^2 \cdot 3^2) = 36(1 - \frac{1}{2})(1 - \frac{1}{3}) = 36 \cdot \frac{1}{2} \cdot \frac{2}{3} = 12$

Često je u primenama (na primer u algoritmu RSA) potrebno izračunati $\varphi(n)$, gde je n proizvod dva velika prosta broja p i q; tada je $\varphi(n) = \varphi(p \cdot q) = \varphi(p) \cdot \varphi(q) = (p - 1) \cdot (q - 1)$.

Implementacija algoritma za računanje Ojlerove funkcije broja svodi se na prilagođavanje algoritma faktorizacije. Množenje promenjlive `phi` faktorom `(1 - 1/p)` u implementaciji ne bi bilo korektno, jer taj faktor nije celobrojan. Ako `phi` inicijalizujemo na `n`, možemo ga ažurirati naredbama oblika `phi = (phi / p) * (p - 1)`, jer je broj `phi` uvek deljiv brojem `p` (`phi` se inicijalizuje na vrednost `n`, a `p` je uvek delilac broja `n`). S obzirom na to da je `(phi / p) * (p - 1) = phi - phi / p`, možemo izbeći množenje tj. možemo ažurirati `phi` naredbama oblika `phi -= phi / p`.

```
// funkcija koja racuna vrednost Ojlerove funkcije
// svodjenjem na problem faktorizacije
int ojlerovaFunkcija(int n) {
  // rezultat inicijalizujemo na n
  int phi = n;
  // za svaki prost cinilac p broja n
  // rezultat mnozimo sa 1-1/p = (p-1)/p
  // usput azuriramo vrednost broja n
  for (int p = 2; p * p <= n; p++) {
    if (n % p == 0) {
      while (n % p == 0)
        n /= p;
      phi -= phi / p;
    }
  }
  // ako je n prost broj
  if (n > 1)
    phi -= phi / n;
  return fi;
}
```

Složenost ovog algoritma odgovara složenosti odgovarajućeg algoritma za faktorizaciju, odnosno jednaka je $O(\sqrt{n})$. Iako je ovaj algoritam neuporedivo efikasniji nego direktan algoritam, izračunavanje vrednosti Ojlerove funkcije kada nije poznata faktorizacija broja n se smatra računski izuzetno teškim problemom i nije ga moguće izvršiti za velike brojeve n (sa nekoliko hiljada cifara), na čemu se zasniva sigurnost nekih kriptografskih algoritama (na primer, RSA opisanog u poglavlju 3.5).

Kao što je to slučaj kada se radi faktorizacija, umesto da se u petlji prolazi redom kroz sve prirodne brojeve od 2 do $\sqrt{n}$, posebno se može razmotriti vrednost $p = 2$, a zatim se u petlji može prolaziti samo kroz skup neparnih brojeva od 3 do $\sqrt{n}$, čime bi se dobilo ubrzanje za faktor 2.

3.3.1.3 Računanje Ojlerove funkcije svih brojeva do n

Ukoliko bi zadatak bio da se izračuna vrednost Ojlerove funkcije za sve brojeve manje ili jednake n, prethodna implementacija bila bi složenosti $O(n\sqrt{n})$.

Kao i ranije, razmotrimo varijantu algoritma zasnovanu na Eratostenovom situ, kojom se angažuje dodatni memorijski prostor veličine $O(n)$. Na osnovu jednakosti (3.4) možemo zaključiti da se za sve brojeve deljive nekim prostim brojem p u izrazu za vrednost Ojlerove funkcije javlja činilac $1 - \frac{1}{p}$. Dakle, možemo

redom prolaziti kroz sve proste brojeve i vrednosti Ojlerove funkcije svih umnožaka tekućeg prostog broja p pomnožiti izrazom $1 - \frac{1}{p}$.

Pošto u izrazu (3.4) za $\varphi(n)$ kao činilac figuriše n, vrednosti elemenata pomoćnog niza $\bar{\varphi}(i)$, $1 \leq i \leq n$, inicijalizovaćemo na vrednost i. Razmatraćemo redom sve vrednosti za p počev od 2 do n: naime, za razliku od osnovne varijante Eratostenovog sita gde je vrednost za p išla do $\sqrt{n}$, ovde je neophodno da p prođe skupom svih vrednosti do n jer je potrebno obraditi vrednosti svih prostih činilaca (a ne samo najmanjih) svih brojeva. Ukoliko prilikom obrade broja p i dalje važi $\bar{\varphi}(p) = p$, to znači da je broj p prost, pa vrednost $\bar{\varphi}$ svih umnožaka broja p množimo sa $\frac{p-1}{p}$. Pritom ne možemo kao u originalnoj verziji Eratostenovog sita krenuti od vrednosti p^2, jer moramo sve umnoške broja p (i brojeve $p, 2p, 3p, \ldots, (p-1)p$) pomnožiti sa $\frac{p-1}{p}$. Ako prilikom obrade broja p važi $\bar{\varphi}(p) \neq p$, broj p je složen i preskače se. Na kraju algoritma u pomoćnom nizu $\bar{\varphi}(i)$ nalaze se vrednosti Ojlerove funkcije za sve vrednosti i od 1 do n.

Opisani algoritam se jednostavno implementira.

```
vector<int> OjlerovaFunkcijaDoN(int n) {
  // niz u kome na poziciji i formiramo vrednost phi(i)
  vector<int> phi(n + 1);
  // inicijalizujemo sve vrednosti
  for (int i = 1; i <= n; i++)
    phi[i] = i;
  // za sve vrednosti od 2 do n
  for (int p = 2; p <= n; p++) {
    // ako vrednost nije menjana, to znaci da je p prosto
    if (phi[p] == p) {
      // azuriramo vrednosti Ojlerove funkcije
      // za sve umnoske broja p
      for (int i = p; i <= n; i += p)
         phi[i] -= phi[i] / p;
    }
  }
  return phi;
}
```

Primer 3.3.3

Ilustrujmo određivanje vrednosti Ojlerove funkcije svih brojeva manjih ili jednakih 20.

- *Krećemo od niza u kome je na poziciji i upisana vrednost i.*

```
 1  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
```

- *U prvom koraku se za sve brojeve deljive sa* 2 *tekuća vrednost Ojlerove funkcije se množi sa* $1 - 1/2 = 1/2$.

```
 1  1  3  2  5  3  7  4  9  5
11  6 13  7 15  8 17  9 19 10
```

- *Nakon toga se razmatra prost broj* 3 *i za sve brojeve deljive sa* 3 *tekuća vrednost Ojlerove funkcije se množi sa* $1 - 1/3 = 2/3$.

```
 1  1  2  2  5  2  7  4  6  5
11  4 13  7 10  8 17  6 19 10
```

- *Broj* 4 *se preskače kao složen.*

- *Za sve brojeve deljive sa* 5 *tekuća vrednost Ojlerove funkcije se množi sa* $1 - 1/5 = 4/5$.

```
 1  1  2  2  4  2  7  4  6  4
11  4 13  7  8  8 17  6 19  8
```

- *Broj* 6 *se preskače kao složen.*
- *Za sve brojeve deljive sa* 7 *tekuća vrednost Ojlerove funkcije se množi sa* $1 - 1/7 = 6/7$.

```
 1  1  2  2  4  2  6  4  6  4
11  4 13  6  8  8 17  6 19  8
```

- *Brojevi* 8, 9 *i* 10 *se preskaču kao složeni.*
- *Brojevima* 11, 13, 17 *i* 19 *se (kao prostim brojevima) vrednost Ojlerove funkcije množi odgovarajućim koeficijentima i time se ove vrednosti zapravo postavljaju na* 10, 12, 16 *i* 18. *Ne postoje brojevi manji ili jednaki* 20 *koji su njima deljivi, te nemamo dodatnih ažuriranja.*

```
 1  1  2  2  4  2  6  4  6  4
10  4 12  6  8  8 16  6 18  8
```

Iako se sa prolazom kroz činioce ne ide do $\sqrt{n}$ već do n i iako se precrtavaju svi umnošci od p do n, umesto od p^2 do n, složenost prikazanog algoritma i dalje iznosi $O(n \log \log n)$, što je i složenost osnovne varijante Eratostenovog sita u koju su uključena sva ova odsecanja.

Zbir Ojlerovih funkcija delilaca

Naredna lema daje jedno interesantno svojstvo Ojlerove funkcije koje nam omogućava da na drugi način odredimo vrednosti Ojlerove funkcije za sve brojeve od 1 do n.

Lema 3.3.2 **[Zbir Ojlerovih funkcija $\phi(d]$ delilaca d broja n)**

Za svaki prirodan broj n *važi:*

$$\sum_{d|n} \varphi(d) = n$$

Pre nego što pređemo na dokaz, razmotrimo naredni primer.

Primer 3.3.4

Neka je $n = 10$. *Napišimo sve razlomke* k/n *za* k *od* 1 *do* n *u skraćenom obliku.*

$$\frac{1}{10}, \frac{1}{5}, \frac{3}{10}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{7}{10}, \frac{4}{5}, \frac{9}{10}, \frac{1}{1}.$$

Lako se primećuje da su imenioci delioci broja 10 *(svaki je dobijen deljenjem broja* 10 *nekim brojem tokom skraćivanja razlomka) i da se za svaki takav imenilac javljaju svi brojioci koji su uzajamno prosti sa njim. Tako se za imenilac* 10 *javljaju brojioci* $1, 3, 7, 9$ *i ima ih* $\varphi(10) = 4$, *za imenilac* 5 *se javljaju brojioci* $1, 2, 3, 4$ *i ima ih* $\varphi(5) = 4$, *za imenilac* 2 *se javlja samo brojilac* 1 *i važi* $\varphi(2) = 1$ *i za imenilac* 1 *se javlja samo brojilac* 1 *i važi* $\varphi(1) = 1$. *U nizu postoji* 10 *razlomaka i važi da je* $\varphi(10) + \varphi(5) + \varphi(2) + \varphi(1) = 10$.

Dokaz. Zaključak iz prethodnog primera se lako uopštava na proizvoljno n tako što se razmotre svi razlomci od $\frac{1}{n}$ do $\frac{n}{n}$ u skraćenom obliku. Njih ima n. Sa druge strane, njihovi imenioci su svi delioci $d|n$, i za svaki imenilac d u ovom nizu postoji tačno $\varphi(d)$ razlomaka (svakom odgovara brojilac uzajamno prost sa d). Zato u nizu postoji tačno $\sum_{d|n} \varphi(d)$ elemenata i tvrđenje je dokazano. □

Prethodna lema nam daje način da korišćenjem naredne formule izračunamo $\varphi(n)$.

$$\varphi(n) = n - \sum_{d|n, d<n} \varphi(d).$$

Postupak je sličan Eratostenovom situ. Inicijalizuju se sve vrednosti u nizu `fi` tako da se na poziciji i nalazi vrednost i. Zatim prolazimo kroz sve vrednosti d od 1 do n. Invarijanta algoritma je da to se u trenutku obrade vrednosti d u nizu na poziciji d već nalazi izračunata vrednost $\varphi(d)$. Tada tu vrednost oduzimamo od svih umnožaka broja d. Invarijanta zaista važi, jer kada se stigne do broja d od početne vrednosti d su oduzete sve vrednosti funkcije φ za delioce broja d (oni su svi manji od d pa su tada već obrađeni).

Implementacija je veoma jednostavna.

```
vector<int> OjlerovaFunkcijaDoN(int n) {
  vector<int> phi(n+1);
  for (int i = 1; i <= n; i++)
     phi[i] = i;
  for (int d = 1; d <= n; d++)
     for (i = d+d; i <= n; i += d)
         phi[i] -= phi[d];
  return phi;
}
```

Pošto se u ovom algoritmu unutrašnja petlja izvršava za sve vrednosti broja d, a ne samo za proste, složenost ovog algoritma je $O(n \log n)$.

Primer 3.3.5

Za $n = 10$, algoritam radi na sledeći način.

- *Najpre inicijalizujemo sve vrednosti.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  2  3  4  5  6  7  8  9  10
```

- *U tom trenutku smo sigurni jedino da je ispravno izračunata vrednost $\varphi(1) = 1$. Svi brojevi veći od 1 su umnošci broja 1, pa vrednosti u nizu umanjujemo za $\varphi(1) = 1$.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  3  4  5  6  7  8  9
```

- *Sada smo sigurni da je $\varphi(2) = 1$, pa vrednosti u nizu na pozicijama umnožaka broja 2 većih od 2 umanjujemo za $\varphi(2) = 1$.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  2  4  4  6  6  8  8
```

- *Sada smo sigurni da je $\varphi(3) = 2$, pa vrednosti u nizu na pozicijama umnožaka broja 3 većih od 3 umanjujemo za $\varphi(3) = 2$.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  2  4  2  6  6  6  8
```

- *Sada smo sigurni da je $\varphi(4) = 2$, pa vrednosti u nizu na pozicijama umnožaka broja 4 većih od 4 umanjujemo za $\varphi(4) = 2$.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  2  4  2  6  4  6  8
```

- *Sada smo sigurni da je* $\varphi(5) = 4$*, pa vrednosti u nizu na pozicijama umnožaka broja* 5 *većih od* 5 *umanjujemo za* $\varphi(5) = 4$*.*

```
i:    0  1  2  3  4  5  6  7  8  9  10
phi:     1  1  2  2  4  2  6  4  6  4
```

- *Pošto naredni brojevi nemaju umnoške koji su veći od njih a i dalje su manji od* $n = 10$*, postupak je završen i u nizu se nalaze vrednosti Ojlerove funkcije od* 1 *do* n*.*

3.3.2 Funkcije delilaca

Funkcija delilaca (engl. divisor function) je neka funkcija nad skupom delilaca datog celog broja. U funkcije delilaca spadaju broj delilaca i zbir delilaca datog broja koje ćemo u daljem tekstu detaljnije razmatrati. Često se koriste funkcije zbira stepena delilaca:

$$\sigma_l(n) = \sum_{d|n} d^l$$

Tada je zbir delilaca $\sigma_1(n)$, a broj delilaca $\sigma_0(n)$.

3.3.2.1 Broj delilaca

Problem

Za dati broj n *izračunati ukupan broj njegovih delilaca* $\sigma_0(n)$*.*

Primer 3.3.6

Delioci broja $n = 24$ *su* $1, 2, 3, 4, 6, 8, 12$ *i* 24 *i ukupno ih ima* 8*, pa je* $\sigma_0(24) = 8$*.*

Direktan algoritam

Primetimo da je svaki broj deljiv brojem 1 i da broj n uvek deli sam sebe te je broj delilaca broja n uvek veći ili jednak 2 (osim kada je $n = 1$). Delioci 1 i n su trivijalni delioci broja n. Jednostavni algoritam za određivanje broja delilaca broja n prolazi skupom svih brojeva od 1 do n i za svaku vrednost koja deli broj n se ukupan broj delilaca uvećava za 1. Pošto nema delilaca između $n/2$ i n, algoritam se može malo usavršiti tako što se delilac n obradi zasebno (na primer, pre petlje), dok se u petlji prolazi skupom brojeva od 1 do $n/2$.

```
// funkcija koja racuna broj delilaca broja n
// razmatrajuci sve potencijalne delioce pojedinacno
int brojDelilaca(int n) {
  // broj delilaca je bar 1 -- sam taj broj
  int br = 1;
  for (int i = 1; i <= n/2; i++)
    if (n % i == 0)
      br++;
  return br;
}
```

Složenost ovog algoritma je $O(n)$.

Traženje parova delilaca

Primetimo da se delioci skoro uvek javljaju u paru: u prethodnom primeru imamo da je $24 = 1 \cdot 24 = 2 \cdot 12 = 3 \cdot 8 = 4 \cdot 6$. Ovo ne važi samo kada je n kvadrat nekog broja: na primer delioci broja 16 su $1, 2, 4, 8$ i 16, gde središnji delilac 4 nema svog para (tj. sam je svoj par). Primetimo da je manji od brojeva koji čine par manji od $\sqrt{n}$ (a ako je broj potpun kvadrat, tada je broj $\sqrt{n}$ sam sebi par). Stoga je moguće formulisati efikasniji algoritam koji prolazi skupom svih brojeva manjih ili jednakih $\sqrt{n}$ i za svaki broj d koji deli broj n ako je $d \neq \frac{n}{d}$ ukupan broj delilaca uvećava za 2, a ako je $d = \frac{n}{d}$ za 1.

```
// funkcija koja racuna broj delilaca broja n
// razmatrajuci parove delilaca
int brojDelilaca(int n) {
  int br = 0;
  for (int i = 1; i * i <= n; i++)
    if (n % i == 0)
      if (i * i != n)
        br += 2;
      else
        br++;
  return br;
}
```

Primetimo da smo ovom malom modifikacijom prethodnog algoritma, smanjili asimptotsku složenost algoritma na $O(\sqrt{n})$.

Svođenje na problem faktorizacije

Funkcija broja delilaca σ_0 je multiplikativna.

Lema 3.3.3 **[Multiplikativnost funkcije broja delilaca σ_0]**

Funkcija σ_0 je multiplikativna, tj. za svaka dva uzajamno prosta broja M i N važi

$$\sigma_0(MN) = \sigma_0(M) \cdot \sigma_0(N).$$

Dokaz. Označimo sa D_K skup delilaca broja K. Funkcija $(m, n) \mapsto mn$ je bijekcija između skupova $D_M \times D_N$ i D_{MN}. Zaista, trivijalna činjenica je da ako $m|M$ i $n|N$, tada $mn|MN$. Međutim, važi i obratno: svaki delilac broja MN je proizvod neka dva delioca $m|M$ i $n|N$. Zaista, ako je d neki delilac broja MN on se može faktorisati. Neka je m proizvod njegovih prostih činilaca koji dele M, a n proizvod njegovih prostih činilaca koji dele N. Pošto su M i N uzajamno prosti nijedan činilac broja m ne deli N i nijedan činilac broja n ne deli M. Zato m deli M, a n deli N. Pošto je uspostavljena bijekcija između dva konačna skupa, važi:

$$\sigma_0(MN) = |D_{MN}| = |D_M \times D_N| = |D_M| \cdot |D_N| = \sigma_0(M) \cdot \sigma_0(N)$$

□

Dakle, dovoljno je samo da umemo da izračunamo vrednosti $\sigma_0(p^k)$. Međutim, to je jednostavno jer su svi delioci broja p^k brojevi $1, p, \ldots, p^k$ kojih ima $k + 1$, pa je $\sigma_0(p_k) = k + 1$. Zato je

$$\sigma_0(n) = \sigma_0(p_1^{k_1} \cdot \ldots \cdot p_m^{k_m}) = \sigma_0(p_1^{k_1}) \cdot \ldots \cdot \sigma_0(p_m^{k_m}) = (k_1 + 1) \cdot \ldots \cdot (k_m + 1).$$

Ovo je zapravo prilično očigledno, jer je opšti oblik svakog delioca broja n jednak $d = p_1^{l_1} \cdot p_2^{l_2} \cdot \ldots \cdot p_m^{l_m}$, gde za svako $1 \leq i \leq m$ važi $0 \leq l_i \leq k_i$. Dakle, za eksponent činioca p_i u deliocu d postoji $k_i + 1$ različitih mogućnosti, te je ukupan broj delilaca broja n jednak $(k_1+1)\cdot(k_2+1)\cdot\ldots\cdot(k_r+1)$. Primetimo da u ovoj formuli figurišu samo eksponenti prostih činilaca, a ne i sami činioci.

Primer 3.3.7

Broj 24 *možemo predstaviti na sledeći način* $24 = 2^3 \cdot 3^1$. *Svi njegovi delioci su* $2^0 \cdot 3^0$, $2^0 \cdot 3^1$, $2^1 \cdot 3^0$, $2^1 \cdot 3^1$, $2^2 \cdot 3^0$, $2^2 \cdot 3^1$, $2^3 \cdot 3^0$ *i* $2^3 \cdot 3^1$ *i ima ih* $(3+1) \cdot (1+1) = 8$.

Implementacija se dobija jednostavnom modifikacijom algoritma faktorizacije.

```
int brojDelilaca(int n) {
  // ukupan broj delilaca broja n
  int broj = 1;
  // prolazimo skupom svih brojeva od 2 do sqrt(n)
  for (int i = 2; i * i <= n; i++) {
    // broj puta koliko broj i deli n
    int k = 0;
    while (n % i == 0) {
      n /= i;
      k++;
    }
    // ukupan broj delilaca mnozimo sa k+1
    broj *= (k + 1);
  }
  // ako je preostali broj prost, njegova vrednost za k je 1,
  // te je prethodni rezultat potrebno pomnoziti sa k + 1 = 2
  if (n > 1)
    broj *= 2;

  return broj;
}
```

Složenost prethodnog algoritma za računanje broja delilaca broja n odgovara složenosti faktorizacije broja n, te iznosi $O(\sqrt{n})$. Primetimo da ako nam je unapred poznata faktorizacija broja n, broj delilaca možemo lako efektivno odrediti.

Određivanje broja delilaca brojeva od 1 do n

Po uzoru na Eratostenovo sito, lako je odrediti broj delilaca svakog broja do 1 do n. Broj 1 je delilac svakog broja, pa broj delilaca svakog broja inicijalizujemo na 1. Zatim prolazimo redom kroz sve moguće delioce d i broj delilaca njihovih umnožaka (krenuvši od samog d) uvećavamo za 1.

```
vector<int> brojDelilaca(n+1, 1);
for (int d = 2; d <= n; d++)
   for (int k = d; k <= n; k++)
      brojDelilaca[k]++;
```

Složenost ovog algoritma je $O(n \log n)$, što je efikasnije od izračunavanja broja delilaca svakog broja pojedinačno, koje bi bilo složenosti $O(n\sqrt{n})$.

3.3.2.2 Zbir delilaca

Problem

Za dati broj n izračunati zbir svih njegovih delilaca.

Primer 3.3.8

Ako je $n = 24$, njegovi delioci su $1, 2, 3, 4, 6, 8, 12$ i 24 i njihov zbir jednak je 60.

Direktan algoritam

Naivni pristup za rešavanje ovog problema je proći skupom svih brojeva manjih od n i uvećavati tekuću sumu delilaca za vrednost svakog od brojeva koji deli broj n. Složenost ovog pristupa je $O(n)$.

```
// funkcija koja racuna sumu delilaca broja n
// razmatrajuci sve moguce delioce pojedinacno
int zbirDelilaca(int n) {
  // n je uvek sam sebi delilac
  int zbir = n;
  for (int i = 1; i < n; i++)
    if (n % i == 0)
      zbir += i;
  return zbir;
}
```

Traženje parova delilaca

Efikasniji pristup je da se delioci otkrivaju u paru, tako što se prolazi skupom svih brojeva manjih ili jednakih $\sqrt{n}$ i za svaki broj i koji deli broj n vrednost i i vrednost njemu odgovarajućeg činioca n/i se dodaje na tekuću sumu, osim kada je baš $i = n/i$, odnosno kada je $i = \sqrt{n}$, kada se na sumu dodaje samo vrednost i.

```
// funkcija koja racuna sumu delilaca broja n
// razmatrajuci parove delilaca
int zbirDelilaca(int n) {
  int zbir = 0;
  for (int i = 1; i * i <= n; i++)
    if (n % i == 0)
      if (i * i != n)
        zbir += (i + n/i);
      else
        zbir += i;
  return zbir;
}
```

Složenost ovog algoritma je $O(\sqrt{n})$.

Svođenje na problem faktorizacije

I funkcija zbira delilaca je multiplikativna. Dokažimo još opštije da su sve funkcije σ_l multiplikativne (podsetimo se, σ_l izračunava zbir l-tih stepena delilaca).

Lema 3.3.4 **[Multiplikativnost funkcije zbira delilaca σ_1]**

Funkcija σ_l je multiplikativna, tj. ako su M i N uzajamno prosti brojevi, tada je

$$\sigma_l(MN) = \sigma_l(M) \cdot \sigma_l(N).$$

Dokaz. Neka su $D_M = \{a_1, \dots, a_m\}$ svi delioci broja M, $D_N = \{b_1, \dots, b_n\}$ svi delioci broja n, a $D_{MN} = \{c_1, \dots, c_{mn}\}$ svi delioci broja MN. Tada je

$$\sigma_l(M) \cdot \sigma_l(N) = (a_1^l + \dots + a_m^l) \cdot (b_1^l + \dots + b_n^l) = \left(\sum_{i=1}^{m} a_i^l\right) \cdot \left(\sum_{i=1}^{n} b_i^l\right) = \sum_{\substack{1 \le i \le m, \\ 1 \le j \le n}} a_i^l b_j^l.$$

Već smo u lemi 3.3.3 dokazali da je $(x, y) \mapsto xy$ bijekcija između skupova $D_M \times D_N$ i D_{MN}. Zato svakom proizvodu $a_i b_j$ za $1 \le i \le m$ i $1 \le j \le n$ odgovara neki $c_k \in D_{MN}$ za $1 \le k \le mn$, i obratno. Zato je

$$\sigma_l(M) \cdot \sigma_l(N) = \sum_{\substack{1 \le i \le m, \\ 1 \le j \le n}} a_i^l b_j^l = \sum_{\substack{1 \le i \le m, \\ 1 \le j \le n}} (a_i b_j)^l = \sum_{k=1}^{mn} c_k^l = \sigma_l(MN).$$

□

Dakle, dovoljno je da znamo vrednost $\sigma_1(p^k)$. Međutim, pošto znamo da su svi delioci broja p^k brojevi $1, p, \dots, p^k$, važi da je

$$\sigma_1(p^k) = 1 + p + \dots + p^k = \frac{p^{k+1} - 1}{p - 1}.$$

Zato je

$$\sigma_1(n) = \sigma_1(p_1^{k_1} \cdot \dots \cdot p_m^{k_m}) = \sigma_1(p_1^{k_1}) \cdot \dots \cdot \sigma_1(p_m^{k_m}) = \frac{p_1^{k_1} - 1}{p_1 - 1} \cdot \dots \cdot \frac{p_m^{k_m} - 1}{p_m - 1}.$$

Do ovog rezultata smo mogli doći i direktno. Opšti oblik delioca broja n jednak je $d = p_1^{l_1} \cdot p_2^{l_2} \cdot \dots \cdot p_m^{l_m}$, gde je $0 \le l_i \le k_i$. Primetimo da se u narednom izrazu posle oslobađanja od zagrada svaki delilac broja n pojavljuje u sumi tačno jednom:

$$(1 + p_1 + p_1^2 + \dots + p_1^{k_1}) \cdot (1 + p_2 + p_2^2 + \dots + p_2^{k_2}) \cdot \dots \cdot (1 + p_m + p_m^2 + \dots + p_m^{k_m})$$

odnosno izraz je jednak sumi delilaca broja n. Dakle, problem se svodi na identifikovanje prostih činalaca broja n, odnosno na faktorizaciju broja n. Iako je na papiru jednostavnije da umesto sume $1 + p_i + p_i^2 + \dots + p_i^{k_i}$ primenimo formulu za zbir geometrijskog niza i dobijemo $\frac{p_i^{k_i+1}-1}{p_i-1}$, u implementaciji stepenovanje možemo izbeći ako traženi zbir računamo inkrementalno, čime dobijamo i brži i jednostavniji kod. Naime, poznati zbir $S = 1 + p_i + \dots + p_i^k$ možemo ažurirati dodavanjem sabirka p_i^{k+1}, ali i množenjem sa p_i i sabiranjem sa 1.

```
int zbirDelilaca(int n) {
  // ukupan zbir delilaca broja n
  int zbir = 1;
  // prolazimo skupom svih brojeva od 2 do sqrt(n)
  for (int p = 2; p * p <= n; p++) {
     // 1 + p + ... + p^k
    int S = 1;
    while (n % p == 0) {
      n /= p;
      S = S * p + 1;
    }
    // azuriramo vrednost zbira delilaca
    zbir *= S;
  }
  // ako je preostali broj n prost, njegovi delioci su 1 i n
  if (n > 1)
    zbir *= (1 + n);
  return zbir;
}
```

Složenost algoritma za računanje sume svih delilaca zasnovanog na faktorizaciji je $O(\sqrt{n})$.

Određivanje zbira delilaca brojeva od 1 do n

Veoma slično broju delilaca, po uzoru na Eratostenovo sito, algoritmom složenosti $O(n \log n)$ možemo odrediti i zbirove delilaca svih brojeva od 1 do n.

```
vector<int> brojDelilaca(n+1, 1);
for (int d = 2; d <= n; d++)
   for (int k = d; k <= n; k++)
      brojDelilaca[k] += d;
```

3.4 Modularna aritmetika

U osnovi modularne aritmetike leži operacija određivanja celobrojnog količnika i ostatka. Podsetimo se, broj je *q količnik*, a broj r *ostatak* pri deljenju broja x brojem y ako i samo ako postoje brojevi q i r takvi da važi $x = q \cdot y + r$ i $0 \leq r < y$. Brojevi q i r su ovim uslovom jedinstveno određeni. Pisaćemo $q = a \operatorname{div} b$ i $r = a \bmod b$.

Pored toga što sa mod označavamo binarnu operaciju, mod se koristi i kao oznaka relacije u skupu celih brojeva. Naime, pisaćemo $a \equiv b \pmod{m}$ i reći da su a i b *kongruentni po modulu* m ako $m|a - b$ tj. ako je $a \bmod m = b \bmod m$, odnosno ako a i b daju isti ostatak pri deljenju sa m. Na primer, $12 \equiv 2 \pmod{5}$ jer $5|(12 - 2)$.

Relaciju mod koristimo u raznim svakodnevnim situacijama, a da toga često nismo ni svesni. Jedan od takvih primera je rad sa vremenom. Naime, za vreme koje je 15 časova nakon 11 sati reći ćemo da je 2 sata (što odgovara tome da je $11 + 15 \equiv 2 \pmod{24}$). Slično važi i za dane u nedelji koje možemo obeležiti brojevima od 0 do 6 (na primer, od nedelje, do subote). Operacije vršimo po modulu 7. Na primer, ako je danas četvrtak (dan obeležen brojem 4), za šest dana biće sreda (dan obeležen brojem 3), jer je $4 + 6 \equiv 3 \pmod{7}$. Analogno se računa i mesec ili redni broj nedelje u godini. Modularna aritmetika ima još puno praktičnih primena, pomenimo samo neke zanimljive: koristi se za izračunavanje kontrolnih suma za međunarodne standardne identifikatore knjiga (ISBN brojeve), međunarodne brojeve bankovnih

računa (IBAN), kao i jedinstvene identifikatore hemijskih jedinjenja (CAS registarski broj). Modularna aritmetika je i u osnovi nekih savremenih kriptografskih sistema.

Neoznačeni celi brojevi se u računarima predstavljaju sa k bitova, a operacije nad njima se izvode po modulu 2^k. Recimo, u jeziku C++ brojevi tipa `unsigned int` su širine 32 bita, pa se operacije nad njima izvode po modulu 2^{32}.

Primer 3.4.1

U narednom kodu kao rezultat kvadriranja broja 123456789 *dobija se vrednost* $123456789^2 \bmod 2^{32} = 2537071545$.

```
unsigned int x = 123456789;
unsigned int y = x * x;
cout << y << endl;
```

Ako je $a_1 \equiv b_1 \pmod{m}$ i $a_2 \equiv b_2 \pmod{m}$, onda je $a_1 \pm a_2 \equiv b_1 \pm b_2 \pmod{m}$ i $a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod{m}$; na primer,

$$\begin{array}{ccc} 12, 14 & \overset{\text{mod } 5}{\longrightarrow} & 2, 4 \\ + \downarrow & & \downarrow + \\ 26 & \overset{\text{mod } 5}{\longrightarrow} & 1 \end{array}$$

Drugim rečima, relacija mod je *saglasna* (kongruentna) sa operacijama sabiranja, oduzimanja i množenja. Rezimirajmo osnovne osobine kogruencija u narednoj lemi.

Lema 3.4.1 **[Osnovne osobine kongruencije]**

Kongruencija po modulu zadovoljava sledeće osobine:

1. *Važi* $a \equiv b \pmod{m}$ *ako i samo ako* $n|(a-b)$. *Važi* $a \equiv 0 \pmod{m}$ *akko* $n|a$.
2. *Kongruencija po modulu je relacija ekvivalencije (važi* $a \equiv a \pmod{m}$*, ako važi* $a \equiv b \pmod{m}$ *tada važi i* $b \equiv a \pmod{m}$ *i ako važi* $a \equiv b \pmod{m}$ *i* $a \equiv c \pmod{m}$ *tada važi i* $a \equiv c \pmod{m}$*).*
3. *Ako važi* $a_1 \equiv b_1 \pmod{m}$ *i* $a_2 \equiv b_2 \pmod{m}$ *tada važi* $a_1 + a_2 \equiv b_1 + b_2 \pmod{m}$, $a_1 - a_2 \equiv b_1 - b_2 \pmod{m}$ *i* $a_1 \cdot a_2 \equiv b_1 \cdot b_2 \pmod{m}$.
4. *Ako važi* $a \equiv b \pmod{m}$*, onda za svaki prirodan broj* k *važi* $a^k \equiv b^k \pmod{m}$.
5. *Ako su* c *i* n *uzajamno prosti (važi* $\text{nzd}(c, m) = 1$*), tada iz* $c \cdot a \equiv c \cdot b \pmod{m}$ *sledi* $a \equiv b \pmod{m}$ *(kongruencija se može skratiti sa* c*).*
6. *Ako važi* $c \cdot a \equiv c \cdot b \pmod{c \cdot m}$ *tada važi i* $a \equiv b \pmod{m}$ *(kongruencija se može skratiti sa* c*).*

Dokaz. Dokažimo samo poslednje dve stavke koje ćemo zvati prvi i drugi zakon skraćivanja (ostale se dokazuju direktno na osnovu definicije).

Pretpostavimo da je $\text{nzd}(c, m) = 1$ i da važi $c \cdot a \equiv c \cdot b \pmod{m}$ tj. da važi $m|(c \cdot a - c \cdot b)$. Zato postoji neki prirodan broj k takav da je $c \cdot (a - b) = km$. Pošto su c i m uzajmno prosti, svi prosti faktori broja m moraju biti sadržani u broju $a - b$, pa postoji neki prirodan broj j takav da je $a - b = jm$, odakle sledi $m|(a - b)$ tj. $a \equiv b \pmod{m}$.

Ako važi $c \cdot a \equiv c \cdot b \pmod{c \cdot m}$, tada postoji prirodan broj k takav da je $(c \cdot a - c \cdot b) = k \cdot c \cdot m$. Skraćivanjem sa c dobija se $a - b = k \cdot m$, iz čega tvrđenje sledi.

Važi i obratan smer, međutim, ovu lemu ćemo koristiti uvek za skraćivanje kongruencija, a ne za njihovo proširivanje. □

Primer 3.4.2

Prvi zakon skraćivanja je moguće primeniti samo kada su brojevi c i m uzajamno prosti. Iz $4 \equiv 10 \pmod 6$ *ne sledi* $2 \equiv 5 \pmod 6$*, međutim, na osnovu drugog zakona skraćivanja sledi* $2 \equiv 5 \pmod 3$.

Ovo inspiriše sledeće jednakosti koje se koriste za računanje vrednosti zbira ili proizvoda brojeva po modulu m.

Lema 3.4.2 **[Sabiranje i množenje po modulu]**

$$\begin{aligned}(a+b) \bmod m &= (a \bmod m + b \bmod m) \bmod m \\ (a \cdot b) \bmod m &= (a \bmod m \cdot b \bmod m) \bmod m\end{aligned}$$

Dokaz. Dokažimo drugu jednakost (prva je jednostavnija za dokazivanje).

Pretpostavimo da je $a = q_a \cdot m + r_a$ i $b = q_b \cdot m + r_b$ za $0 \leq r_a, r_b < m$. Tada je:

$$a \cdot b = (q_a \cdot m + r_a) \cdot (q_b \cdot m + r_b) = (q_a \cdot q_b \cdot m + q_a \cdot r_b + r_a \cdot q_b)m + r_a \cdot r_b.$$

Ako je $r_a \cdot r_b = q \cdot m + r,\ 0 \leq r < m$, tada je:

$$a \cdot b = (q_a \cdot q_b \cdot m + q_a \cdot r_b + r_a \cdot q_b + q)m + r,\ 0 \leq r < m$$

pa je $(a \cdot b) \bmod m = r$.

Važi, takođe, da je $(a \bmod m \cdot b \bmod m) \bmod m = (r_a \cdot r_b) \bmod m = r$, čime je tvrđenje dokazano. □

Računanje na osnovu prethodnih jednakosti je važno, jer često omogućava da se izbegnu greške nastale usled prekoračenja. Naime, ako su brojevi a i b relativno veliki, a m mali, tada izrazi $a + b$ i $a \cdot b$ mogu dovesti do prekoračenja, pa se samim tim mogu dobiti netačni rezultati prilikom izračunavanja vrednosti izraza $(a + b) \bmod m$ i $(a \cdot b) \bmod m$. Sa druge strane, u izrazima $(a \bmod m + b \bmod m) \bmod m$ i $(a \bmod m \cdot b \bmod m) \bmod m$ se računa sa manjim brojevima i oni najčešće ne dovode do prekoračenja.

Primer 3.4.3

Na primer, ako je $a = b = 3 \cdot 10^9 + 9$*, a* $m = 10$*, tada je* $a + b = 6 \cdot 10^9 + 18$ *i ne može se ispravno predstaviti pomoću* 32 *bita. Važi da je* $a \bmod m = b \bmod m = 9$*, pa se zbir* $a \bmod m + b \bmod m = 18$ *može ispravno predstaviti sa* 32 *bita, pre nego što se izračuna konačan rezultat* $(a \bmod m + b \bmod m) \bmod m = 8$.

Ipak, treba biti obazriv, jer je moguće da se desi da su i operandi ($a \bmod m$ i $b \bmod m$) i krajnji rezultat dovoljno mali da se mogu ispravno predstaviti određenim tipom, ali da su međurezultati ($a \bmod m + b \bmod m$ i naročito $a \bmod m \cdot b \bmod m$) preveliki da bi mogli da se ispravno predstave.

Primer 3.4.4

Ako je $a = b = 10^5 + 1$ *i* $m = 10^6$, *tada je* $a \bmod m = b \bmod m = 10^5 + 1$, *pa je* $a \bmod m \cdot a \bmod m = 10^{10} + 2 \cdot 10^5 + 1$, *što se ne može ispravno zapisati pomoću* 32 *bita. Konačan rezultat je* $(a \bmod m \cdot b \bmod m) \bmod m = 2 \cdot 10^5 + 1$ *i on se može ispravno zapisati pomoću* 32 *bita.*

Zato je u slučaju velikih vrednosti broja m poželjno međurezultate izračunati u širem tipu (na primer, ako su $a \bmod m$ i $b \bmod m$ predstavljeni sa 32 bita, tada je poželjno konvertovati ih i pomnožiti kao 64-bitne brojeve, pre nego što se izračunavanjem ostatka dobije konačan rezultat koji se ponovo ispravno može predstaviti pomoću 32 bita).

Prethodna svojstva nam omogućavaju da garantujemo da će se uz sva prekoračenja međurezultata koja dolaze tokom rada sa neoznačenim brojevima, na kraju izračunavanja dobiti rezultat koji je jednak ostatku pri deljenju brojem 2^{32} (tj. 2 na broj bitova upotrebljenih za zapis) rezultata koji bi se dobio bez prekoračenja.

Razmotrimo sada problem određivanja razlike nenegativnih brojeva b i a po modulu m. Na primer, potrebno je odrediti vrednost $(b - a) \bmod m$ za $a, b \geq 0$. Pretpostavimo za početak da su brojevi a i b manji od m. Ponovimo da ne postoji saglasnost između različitih programskih jezika u računanju vrednosti `a % m` kada je vrednost broja `a` negativna. Naime, u jeziku C++ se za vrednost ostatka pri deljenju može dobiti negativan broj: na primer, vrednost izraza `(2 - 7) % 3` je -2, dok se u jeziku Python kao rezultat istog ovog izraza dobija pozitivan broj 1. Često želimo da kao rezultat računanja vrednosti po modulu dobijemo nenegativnu vrednost. Umesto da vršimo analizu slučajeva, rešenje je moguće dobiti izračunavanjem vrednosti izraza $(b \bmod m - a \bmod m + m) \bmod m$.

Lema 3.4.3 **[Oduzimanje po modulu]**

Za proizvoljne brojeve a *i* b *važi:*

$$(b - a) \bmod m = (b \bmod m - a \bmod m + m) \bmod m$$

Pritom je vrednost izraza $b \bmod m - a \bmod m + m$ *uvek nenegativna.*

Dokaz. Neka je $a = q_a \cdot m + r_a$ i $b = q_b \cdot m + r_b$, za $0 \leq r_a, r_b < m$. Tada je $a \bmod m = r_a$ i $b \bmod m = r_b$. Neka je: $r_b - r_a + m = p \cdot m + r$, $0 \leq r < m$. Zato je: $(b \bmod m - a \bmod m + m) \bmod m = (r_b - r_a + m) \bmod m = r$. Takođe, važi i da je: $b - a = (q_b - q_a) \cdot m + (r_b - r_a) = (q_b - q_a - 1) \cdot m + (r_b - r_a + m) = (q_b - q_a - 1 + p) \cdot m + r$, pa je i $(b - a) \bmod m = r$.

Brojevi $a \bmod m = r_a$ i $b \bmod m = r_b$ pripadaju intervalu $[0, m)$, pa njihova razlika pripada intervalu $(-m, m)$. Dodavanjem vrednosti m, dobija se nenegativna vrednost (u intervalu $(0, 2m)$). □

Naravno, ako se unapred zna da su argumenti a i b već svedeni po modulu m (tj. važi $0 \leq a < m$ i $0 \leq b < m$), onda je dovoljno samo koristiti izraz $(b - a + m) \bmod m$.

Prethodno tvrđenje nam omogućava da u bilo kom programskom jeziku razliku po modulu možemo izračunati jednim izrazom, tako da se dobije nenegativan rezultat.

Važi sličan rezultat i za stepenovanje po modulu.

Lema 3.4.4 **[Stepenovanje po modulu]**

$a^n \bmod m = (a \bmod m)^n \bmod m$

Ova lema se jednostavno dokazuje primenom pravila za množenje brojeva po modulu. Primetimo da ono ima važnu primenu jer za iole veće vrednosti n pri računanju vrednosti a^n može doći do prekoračenja (i kada je a prethodno sveden po modulu m). Ovakvo modularno stepenovanje se najbolje izvodi algoritmom

brzog stepenovanja (pri čemu je poželjno u prvom pozivu svesti vrednost a po modulu m tj. kao prvi parametar proslediti vrednost `a % m`).

```
int stepen(int a, int n, int m) {
    if (n == 0)
       return 1;
    if (n % 2 == 0)
        return stepen((a * a) % m, n / 2, m);
    else
        return (a * stepen(a, n-1, m)) % m;
}
```

Primer 3.4.5

Ilustrujmo proceduru računanja trinaestog stepena broja 2 *po modulu* 100*:*

$$\begin{aligned}
\texttt{stepen}(2,13,100) &= \\
(2\cdot \texttt{stepen}(2,12,100)) \bmod 100 &= \\
(2\cdot \texttt{stepen}((2\cdot 2) \bmod 100,6,100)) \bmod 100 &= \\
(2\cdot \texttt{stepen}(4,6,100)) \bmod 100 &= \\
(2\cdot \texttt{stepen}((4\cdot 4) \bmod 100,3,100)) \bmod 100 &= \\
(2\cdot \texttt{stepen}(16,3,100)) \bmod 100 &= \\
(2\cdot ((16\cdot \texttt{stepen}(16,2,100)) \bmod 100)) \bmod 100 &= \\
(2\cdot ((16\cdot \texttt{stepen}((16\cdot 16) \bmod 100,1,100)) \bmod 100)) \bmod 100 &= \\
(2\cdot ((16\cdot \texttt{stepen}(56,1,100)) \bmod 100)) \bmod 100 &= \\
(2\cdot ((16\cdot (56\cdot \texttt{stepen}(56,0,100)) \bmod 100) \bmod 100)) \bmod 100 &= \\
(2\cdot ((16\cdot (56\cdot 1) \bmod 100) \bmod 100)) \bmod 100 &= \\
(2\cdot ((16\cdot 56) \bmod 100) \bmod 100) = (2\cdot 96) \bmod 100 &= 92
\end{aligned}$$

Pošto je $2^{13} = 8192$*, rezultat je ispravno izračunat.*

Aritmetičke operacije po modulu m se često koriste, pa ćemo uvesti naredne skraćene oznake: sa $+_m$ ćemo obeležavati sabiranje po modulu tj. $a +_m b$ je oznaka za $(a+b) \bmod m$, sa $\times_m$ množenje po modulu tj. $(a \cdot b) \bmod m$ itd.

U kriptografiji se obično razmatraju jako veliki brojevi (sa po nekoliko hiljada cifara) koji se ne mogu predstaviti mašinskim tipovima podataka i za koje se ni osnovne aritmetičke operacije ne mogu izvršiti u konstantnom vremenu. Broj cifara broja n je $O(\log n)$.

- Sabiranje i oduzimanje dva velika broja (reda veličine broja n) može se izvršiti u vremenu $O(\log n)$.
- Množenje takva dva broja se osnovnim algoritmom može izvršiti u vremenu $O(\log^2 n)$ (a može i efikasnije, na primer, primenom brze Furijeove transformacije opisane u poglavlju 3.7).
- I deljenje se može osnovnim algoritmom izvršiti u vremenu $O(\log^2 n)$ (a postoje i efikasniji algoritmi).

Dakle, sve osnovne aritmetičke operacije su dovoljno efikasne i kada se sprovode nad brojevima sa po nekoliko hiljada cifara.

Stvar se ne menja značajno ni kada se operacije izvršavaju po modulu. Određivanje modula zahteva dodatna deljenja, međutim, smanjuje broj cifara rezultata, pa se složeni izrazi ponekad efikasnije izračunavaju kada su operacije po modulu.

- U poglavlju 3.4.1 videćemo da deljenje po modulu, tj. određivanje modularnog inverza zahteva sprovođenje proširenog Euklidovog algoritma, pa je sporije od množenja brojeva po modulu. Međutim, prošireni Euklidov algoritam primenjen na brojeve a i n zahteva $O(\log(a+n))$ koraka koji uključuju osnovne aritmetičke operacije po modulu. Zato su u rezultujućoj složenosti deljenja po modulu svi izrazi pod logaritmom, pa se i ova operacija može izvršiti relativno efikasno (u veoma razumnom vremenu, čak i za brojeve koji imaju po nekoliko hiljada cifara).
- Modularno stepenovanje $a^k \bmod n$ zahteva $O(\log k)$ operacija deljenja i množenja po modulu, pa se i ono može izvršiti u razumnom vremenu, čak i kada vrednosti brojeva a, k i n imaju po nekoliko hiljada cifara.

Sa druge strane, zbog jako velikih vrednosti n neke operacije koje se efikasno sprovode nad mašinskim tipovima podataka su jako neefikasne i ne mogu se praktično izvršiti. Kod ovako velikih brojeva postoji ogromna razlika između algoritama složenosti $O(\sqrt{n})$ i $O(\log n)$. Ako je n reda veličine 10^{1000}, onda algoritam složenosti $O(\sqrt{n})$ zahteva oko 10^{500} operacija, što je neizvodivo, dok algoritam složenosti $O(\log n)$ zahteva samo oko 1000 operacija, što se veoma brzo izvršava. Sigurnost mnogih kriptografskih protokola zasnovana je na težini izvođenja ovakvih operacija.

- Operacija faktorizacije broja n zahteva $O(\sqrt{n})$ koraka ako se koristi osnovni algoritam i ne može se sprovesti za brojeve koji imaju velike proste faktore.
- Izračunavanje Ojlerove funkcije (i sličnih multiplikativnih funkcija) obično se svodi na faktorizaciju, pa je i ono praktično neizvodivo.
- Operacija inverzna modularnom stepenovanju, tj. operacija pronalaženja vrednosti k takve da je $a^k \equiv b \pmod n$ naziva se *diskretni logaritam* i do sada nije pronađen ni jedan algoritam koji bi omogućio njeno efikasno izvršavanje (za brojeve od po nekoliko hiljada cifara i najsavremenijim računarima bi trebali bilioni godina za rešavanje ovog problema).

3.4.1 Modularne grupe

Za svaki prirodan broj n mogući ostaci pri deljenju sa n čine skup $Z_n = \{0, 1, \ldots, n-1\}$. Kada se ovi ostaci kombinuju operacijom $+_n$ sabiranja po modulu n i operacijom $\times_n$ množenja po modulu n ponovo se dobijaju vrednosti koje pripadaju tom skupu. Pošto su obe operacije asocijativne i pošto neutral za sabiranje 0 i neutral za množenje 1 pripadaju skupu ostataka Z_n, strukture $(Z_n, +_n)$ i $(Z_n \setminus \{0\}, \times_n)$ su monoidi[4] (za bilo koju vrednost n).

Svaki element u strukturi $(Z_n, +_n)$ ima inverzni element jer je za svako $x \in Z_n$ i element $n - x \in Z_n$ i važi $x +_n (n-x) = 0$, pa je struktura $(Z_n, +_n)$ grupa[5] (za bilo koju vrednost n). Ovu grupu nazivamo *aditivna grupa celih brojeva po modulu* n ili *modularna aditivna grupa* i obeležavamo sa $\mathbb{Z}_n^+$. Inverzni element u odnosu na $+_n$ nazivamo *modularni aditivni inverz*.

U narednoj tablici prikazana je tablica sabiranja modularne aditivne grupe $\mathbb{Z}_5^+$.

$+_5$	**0**	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2

[4] Monoid je algebarska struktura koju čine skup i binarna operacija koja je zatvorena na tom skupu, asocijativna i ima neutral.

[5] Grupa je algebarska struktura koju čine skup i binarna operacija koja je zatvorena na tom skupu, asocijativna, ima neutral i svaki element ima inverzni element.

$+_5$	**0**	**1**	**2**	**3**	**4**
4	4	0	1	2	3

Da li, analogno aditivnim, možemo da razmatramo *modularne multiplikativne grupe*? Struktura $(Z_n \setminus \{0\}, \times_n)$ ne mora biti grupa, jer ne moraju svi elementi da imaju inverz (koji u ovom slučaju nazivamo *modularni multiplikativni inverz*). Na primer, razmotrimo tablicu za $n = 6$.

$\times_6$	**1**	**2**	**3**	**4**	**5**
1	1	2	3	4	5
2	2	4	0	2	4
3	3	0	3	0	3
4	4	2	0	4	2
5	5	4	3	2	1

Kada se broj 2 pomnoži bilo kojim elementom skupa $Z_6 \setminus \{0\}$, dobija se paran broj i ostatak tog broja pri deljenju sa 6 može biti samo 0, 2 ili 4. Zato 2 nema inverzni element. Slično se dešava i sa elementima 3 i 4. Element 1 je sam sebi inverzan, a isto važi i za element 5 (jer je $5 \cdot 5 = 25 \equiv 1 \pmod 6$).

Sa druge strane, razmotrimo tablicu za $n = 5$.

$\times_5$	**1**	**2**	**3**	**4**
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

Jasno je da svaki element ima sebi inverzni (u svakoj vrsti i svakoj koloni se tačno jednom javlja vrednost 1) i ova struktura je grupa. Inverz broja 1 je uvek 1 (jer je $1 \cdot 1 = 1$), inverz broja 2 je 3 (jer je $2 \cdot 3 = 6 \equiv 1 \pmod 5$), inverz broja 3 je 2, a broj 4 je sam svoj inverz (jer je $4 \cdot 4 = 16 \equiv 1 \pmod 5$). Ova struktura je, dakle, grupa.

Dakle, za neke vrednosti n, struktura $(Z_n \setminus \{0\}, \times_n)$ jeste grupa, a za neke nije.

Primetimo da su u slučaju $n = 6$ inverz imali brojevi 1 i 5 koji su uzajamno prosti sa 6, a u slučaju $n = 5$ svi brojevi 1, 2, 3 i 4, jer su svi uzajamno prosti sa 5. Razmotrimo sada opšti slučaj i formulišimo potreban i dovoljan uslov da element skupa $Z_n \setminus \{0\}$ ima inverz u odnosu na operaciju $\times_n$.

Teorema 3.4.1 **[Postojanje modularnog multiplikativnog inverza]**

Ako su brojevi a i n uzajamno prosti prirodni brojevi (ako je $\text{nzd}(a, n) = 1$*, tj. ako brojevi a i n nemaju zajedničkih prostih činilaca), onda postoji jedinstven multiplikativni inverz broja a po modulu n, tj. jednačina $ax \equiv 1 \pmod n$ ima jedinstveno rešenje u skupu $Z_n \setminus \{0\}$. Ako brojevi a i n nisu uzajamno prosti, onda modularni multiplikativni inverz broja a ne postoji.*

Dokaz. Jednačina $ax \equiv 1 \pmod n$ ima rešenje ako i samo ako postoji prirodan broj q tako da je $ax = qn + 1$, tj. ako je $ax - qn = 1$. Zato tvrđenje direktno sledi iz teoreme 3.1.4.

Ilustracije radi, dokažimo tvrđenje i direktno.

Ako bi postojao zajednički činilac $d > 1$ brojeva a i n on bi delio levu stranu ove jednačine, pa bi morao da deli i desnu tj. broj 1, što je nemoguće. Dakle, ako brojevi a i n nisu uzajamno prosti, onda a nema inverz.

Pretpostavimo sada da su brojevi a i n uzajamno prosti. Razmotrimo niz od n brojeva $a \cdot 0, a \cdot 1, a \cdot 2, \ldots, a \cdot (n-1)$. Pokazaćemo da su ostaci svih ovih vrednosti pri deljenju sa n različiti. S obzirom na to da ukupno postoji tačno n različitih vrednosti u skupu Z_n, odatle sledi da je $a \cdot x \equiv 1 \pmod{n}$ za tačno jedno x iz skupa Z_n. Pošto to ne može biti $x = 0$, x pripada skupu $Z_n \setminus \{0\}$ i on je jedinstveni modularni multiplikativni inverz broja a po modulu n.

Da bismo pokazali da su ostaci različiti, pretpostavimo suprotno, tj. pretpostavimo da je $a \cdot x_1 \equiv a \cdot x_2 \pmod{n}$ za dve različite vrednosti x_1 i x_2 iz skupa Z_n. Pošto su a i n uzajamno prosti, na osnovu prvog zakona skraćivanja kongruencija (lema 3.4.1) važi $x_1 \equiv x_2 \pmod{n}$, tj. $x_1 - x_2$ mora biti neki celobrojni umnožak broja n. Pošto su x_1 i x_2 dva različita nenegativna cela broja manja od n, to je moguće samo ako je $x_1 - x_2 = 0$, što je kontradikcija jer smo pretpostavili da važi $x_1 \neq x_2$

Dakle, sve vrednosti $a \times_n 0, a \times_n 1, \ldots, a \times_n (n-1)$ su međusobno različite i za tačno jednu vrednost iz skupa $\{1, \ldots, n-1\}$ važi da je multiplikativni inverz broja a po modulu n. □

Broj x je inverz broja a, ali je i a ujedno inverz broja x, pa i x mora da bude uzajamno prost sa n.

Ako je p prost broj, svi elementi skupa $Z_p \setminus \{0\}$ su uzajamno prosti sa p, pa je struktura $\mathbb{Z}_p^\times = (Z_p \setminus \{0\}, \times_p) = (\{1, 2, \ldots, p-1\}, \times_p)$ grupa i označavamo je sa $\mathbb{Z}_p^\times$. Još opštije, ako posmatramo samo skup Φ_n brojeva između 1 i $n-1$ koji su uzajamno prosti sa n, struktura $(\Phi_n, \times_n)$ je grupa. Primetimo da za prost broj p važi da je skup $Z_p \setminus \{0\} = \Phi_p$, pa je grupa $\mathbb{Z}_p^\times$ samo specijalni slučaj grupe $\Phi_p^\times = (\Phi_p, \times_p)$. Grupa $(\Phi_n, \times_n)$ naziva se *multiplikativna grupa celih brojeva po modulu n* ili *modularna multiplikativna grupa*. Ove grupe ćemo obeležavati sa $\Phi_n^\times$.

Primer 3.4.6

U nastavku je data tabela grupe $\Phi_9^\times$.

$\times_9$	***1***	***2***	***4***	***5***	***7***	***8***
1	1	2	4	5	7	8
2	2	4	8	1	5	7
4	4	8	7	2	1	5
5	5	1	2	7	8	4
7	7	5	1	8	4	2
8	8	7	5	4	2	1

Operacija deljenja brojem a po modulu n se svodi na množenje modularnim multiplikativnim inverzom broja a po modulu n. Ovo se u nekim situacijama može upotrebiti za optimizaciju deljenja. Pretpostavimo da je potrebno izvršiti deljenje većeg broja neoznačenih brojeva nekim brojem k, da su ti brojevi zapisani sa w bitova i svi deljivi sa k. Umesto da se svi brojevi dele sa k moguće je (jednom) izračunati modularni multiplikativni inverz broja k po modulu 2^w, a zatim sve brojeve pomnožiti tim inverzom. Pošto se po pravilu množenje izvršava brže od deljenja, ovo predstavlja optimizaciju. Naravno, da bi modularni multiplikativni inverz postojao, potrebno je da su brojevi k i 2^w uzajamno prosti, što je slučaj ako i samo ako je broj k neparan. Sličan postupak možemo upotrebiti i ako je k paran, tako što ćemo k zapisati u obliku $2^p \cdot k'$ za neparan broj k', a zatim svaki broj podeliti sa 2^p (bitovskim šiftovanjem za p mesta udesno, što je veoma brza operacija), pre množenja sa modularnim multiplikativnim inverzom broja k' po modulu 2^w (on sigurno postoji jer je k' neparan broj).

3.4.2 Mala Fermaova i Ojlerova teorema

Kardinalnost skupa Φ_n, tj. broj brojeva između 1 i $n-1$ koji su uzajamno prosti sa n izračunava se Ojlerovom funkcijom φ, čije je izračunavanje opisano u poglavlju o multiplikativnim funkcijama 3.3. Važi

$|\Phi_n| = \varphi(n)$. Ojlerova teorema nam suštinski govori o određenim stepenima elemenata modularne multiplikativne grupe, ali se uopštava i na brojeve koji su veći od n.

Teorema 3.4.2 **[Ojlerova teorema]**

Ako su brojevi a i n uzajamno prosti, tada je $a^{\varphi(n)} \equiv 1 \pmod{n}$ tj. broj $a^{\varphi(n)} - 1$ je deljiv sa n.

Primer 3.4.7

Videli smo da je $\Phi_9 = \{1, 2, 4, 5, 7, 8\}$, pa je $\varphi(9) = 6$. Neka je, na primer, $a = 5$: brojevi $a = 5$ i $n = 9$ su uzajamno prosti. Tada je $5^6 = 15625 = 1736 \cdot 9 + 1$, pa je zaista $a^{\varphi(n)} = 5^6 \equiv 1 \pmod{9}$. Tvrđenje važi i za veće brojeve a, koji ne pripadaju skupu Φ_9. Na primer, broj $a = 14$ je uzajamno prost sa 9 i važi da je $a^{\varphi(n)} = 14^6 = 7529536 = 836615 \cdot 9 + 1 \equiv 1 \pmod{9}$.

Osnovni slučaj Ojlerove teoreme čine elementi skupa Φ_n, odnosno brojevi $1 \leq a < n$, uzajamno prosti sa n. Ako pretpostavimo da teorema važi za njih, lako je dokazati i da važi za sve ostale brojeve koji zadovoljavaju uslove teoreme. Recimo da je $a > n$ i da je a uzajamno prosto sa n. Tada se a može podeliti sa n, tj. postoje brojevi q i $1 \leq r < n$ takvi da je $a = qn + r$. Ako je a uzajamno prost sa n, takav mora biti i r (jer ako bi r imao neki zajednički činilac sa n desna strana bi bila deljiva njime, pa bi deljiv njime morao da bude i broj a na levoj strani jednakosti). Tada je $a^{\varphi(n)} = (qn + r)^{\varphi(n)}$, međutim, $(qn + r)^{\varphi(n)} \equiv r^{\varphi(n)} \pmod{n}$. Pošto je i $1 \leq r < n$, važi da je $r \in \Phi_n$, pa na osnovu pretpostavke da teorema važi u osnovnom slučaju, tj. da važi za sve elemente skupa Φ_n, ona važi i za r. Dakle, važi $r^{\varphi(n)} \equiv 1 \pmod{n}$, pa važi i $a^{\varphi(n)} \equiv 1 \pmod{n}$.

Specijalni slučaj Ojlerove teoreme, kada je n prost broj, je *Mala Fermaova teorema* (dok Ojlerova teorema govori o grupama $\Phi_n^\times$, Mala Fermaova govori samo o grupama $\mathbb{Z}_p^\times$).

Teorema 3.4.3 **[Mala Fermaova teorema]**

Ako je p prost broj i a prirodan broj koji nije deljiv brojem p, tada je $a^{p-1} \equiv 1 \pmod{p}$.

Ako a i n nisu uzajamno prosti, tada jednakost $a^{\varphi(n)} \equiv 1 \pmod{n}$ ne mora da važi. Na primer, ako je a deljivo sa p tada je $a^0 = 1$, međutim važi $a^1 \equiv 0 \pmod{p}$, $a^2 \equiv 0 \pmod{p}$, pa i $a^{p-1} \equiv 0 \pmod{p}$, jer su svi stepeni broja a deljivi sa p, pa $a^{p-1} \not\equiv 1 \pmod{n}$ i jednakost $a^{p-1} \equiv 1 \pmod{p}$ ne mora da važi. Ipak, naredna, modifikovana, formulacija Male Fermaove teoreme ispravno obuhvata i ovaj slučaj.

Teorema 3.4.4 **[Mala Fermaova teorema – opšti oblik]**

Ako je p prost broj i a prirodan broj, tada je $a^p \equiv a \pmod{p}$.

Dokaz. Ako a nije deljivo sa p tvrđenje je ekvivalentno polaznoj formulaciji (kada se obe strane pomnože sa a), a ako a jeste deljivo sa p onda je $a \equiv 0 \pmod{p}$ i $a^p \equiv 0 \pmod{p}$, pa je $a^p \equiv a \pmod{p}$. □

Pokazali smo da je dovoljno da se Ojlerova teorema dokaže za svaki broj $a \in \Phi_n$. Iz toga sledi i opšti slučaj Ojlerove teoreme i Mala Fermaova teorema. Dokažimo[6] da za ovakve brojeve važi $a^{\varphi(n)} \equiv 1 \pmod{n}$. To je neposredna posledica Lagranževe teoreme iz teorije grupa koja tvrdi da red (broj elemenata) podgrupe uvek deli red grupe iz čega rezultat direktno sledi posmatrajući grupu Φ_n i njenu cikličku podrgrupu generisanu sa a (nju čine elementi 1, a, a^2, …, a^{k-1}, za neko $k|\varphi(n)$ takvo da je $a^k = 1$), međutim, daćemo direktan dokaz, bez pozivanja na rezultate teorije grupe.

[6] Korektnost se obezbeđuje bilo kojim ispravnim dokazom osnovnog slučaja Ojlerove teoreme, međutim, alternativni dokazi koje prikazujemo imaju cilj da pored opravdanja daju i dublje objašnjenje obe teoreme.

Dokaz. Osnovu ovog dokaza čini činjenica da se prilikom množenja po modulu n elemenata uređenog skupa $\Phi_n = \{r_1, \ldots, r_{\varphi(n)}\}$ nekim brojem a koji je uzajamno prost sa n dobija neka permutacija tog skupa (što smo i dokazali prilikom dokazivanja kriterijuma za postojanje inverza u teoremi 3.4.1). Zaista, na osnovu prvog zakona skraćivanja (lema 3.4.1), $ar_i \equiv ar_j \pmod n$ važi ako i samo ako je $r_i = r_j$ tj. $i = j$. Zato se množenjem različitih ostataka iz Φ_n brojem a dobijaju različiti ostaci, pa slika skupa Φ_n pri preslikavanju $r \mapsto ar$ mora biti skup Φ_n. Zato je

$$\prod_{i=1}^{\varphi(n)} r_i \equiv \prod_{i=1}^{\varphi(n)} ar_i = a^{\varphi(n)} \cdot \prod_{i=1}^{\varphi(n)} r_i \pmod n.$$

Traženo tvrđenje se dobija na osnovu prvog zakona skraćivanja (lema 3.4.1), skraćivanjem faktora $\prod_{i=1}^{\varphi(n)} r_i$, koji je uzajamno prost sa n. □

Dokaz. Jedan veoma lep, a krajnje elementaran način da se dokaže Mala Fermaova teorema je da se razmotre sve niske dužine p u kojima se javljaju slova iz azbuke koja ima a simbola. Takvih niski ima a^p. Na primer, ako je $p = 3$ i $a = 2$, i ako azbuka sadrži slova `X` i `Y`, to su niske `XXX`, `XXY`, `XYX`, `XYY`, `YXX`, `YXY`, `YYX` i `YYY`. Zamislimo sada da su slova svake niske napisana na krugu. Time sve date niske delimo u određene grupe tako da dve niske pripadaju istoj grupi ako i samo ako se dobijaju čitanjem slova sa istog kruga. U jednoj grupi je samo `XXX`, u drugoj su `XXY`, `XYX` i `YXX`, u trećoj grupi su `XYY`, `YYX` i `YXY`, dok je u poslednjoj, četvrtoj grupi samo `YYY`.

`XXX`	`XXY`	`XYY`	`YYY`
	`XYX`	`YXY`	
	`YXX`	`YYX`	

U opštem slučaju, postojaće a grupa koje sadrže samo jedan element i u njima će biti niske koje sadrže p ponavljanja jednog istog simbola iz azbuke. Sve ostale grupe imaće po p elemenata. Naime, ako bi se zapis dat na nekom krugu mogao pročitati na dva ista načina krenuvši od dva različita slova, tada bi taj zapis morao biti periodičan i period bi morao da deli dužinu niske. Međutim, pošto je p prost broj, njegovi jedini delioci su 1 i p, tako da svaka grupa ima ili 1 ili p različitih elemenata. Dakle, važi da je $a^p = x \cdot p + a$, gde je x broj grupa sa po p elemenata, da je $a^p - a = x \cdot p$, tj. da je $a(a^{p-1} - 1) = x \cdot p$. Dakle, $a^p - a$ je deljivo sa p, što je tvrđenje najopštijeg oblika Fermaove teoreme. Dodatno, ako a nije deljivo sa p, tada a ne može da ima zajedničkih prostih faktora sa p (jer je p prost), pa sledi da $a^{p-1} - 1$ mora da bude deljivo sa p, što je tvrđenje nedegenerisanog slučaja Male Fermaove teoreme. □

Malu Fermaovu teoremu je jednostavno dokazati i indukcijom.

Primene Ojlerove i Male Fermaove teoreme su razne: ispitivanje da li je broj prost, izračunavanje modularnog multiplikativnog inverza, RSA kriptografija i slično. Prikažimo sada primenu Male Fermaove teoreme na ispitivanje da li je broj prost, a ostale primene ovih teorema će biti prikazane u narednim poglavljima (na primer, u poglavlju 3.5 biće prikazana primena Ojlerove teoreme u oblasti kriptografije).

3.4.2.1 Fermaov test da li je broj prost

Jedna primena Male Fermaove teoreme je za testiranje da li je dati broj n prost ili ne (engl. Fermat primality test). Mala Fermaova teorema daje potreban uslov da bi broj bio prost. Naime, za dati broj n možemo izabrati proizvoljan broj a koji nije deljiv sa n i izračunati vrednost $a^{n-1} \bmod n$. Ukoliko je rezultat različit od 1, broj n je sigurno složen. Ipak, teorema ne daje i dovoljan uslov da bi broj bio prost, jer postoje i složeni brojevi n takvi da za neke vrednosti a koje nisu deljive sa n važi da je $a^{n-1} \bmod n$ jednako 1. Ipak, ako je $a^{n-1} \bmod n = 1$, pokazuje se da je mala verovatnoća da broj n nije prost. Posebno, ako se za veliki broj različitih vrednosti za a koje nisu deljive sa n pokaže da je vrednost izraza $a^{n-1} \bmod n$ jednaka 1, broj n je sa velikom verovatnoćom prost. Nažalost, ni ispitivanje svih vrednosti a nam ne daje garanciju,

jer postoje složeni brojevi n takvi da za sve brojeve a koji nisu deljivi sa n važi $a^{n-1} \equiv 1 (\text{mod } n)$. Takvi brojevi se zovu *Karmajklovi brojevi* (engl. Carmichael numbers) i relativno su retki. Najmanji takav broj je 561.

3.4.3 Izračunavanje modularnog multiplikativnog inverza

U poglavlju 3.4.1 o modularnim grupama smo videli da je *multiplikativni inverz broja a po modulu n* (engl. modular multiplicative inverse) broj x za koji važi $a \cdot x \equiv 1 \pmod{n}$.

Primer 3.4.8

Multiplikativni inverz broja 5 *po modulu* 8 *je* 5 *jer važi* $5 \cdot 5 \equiv 1 \pmod{8}$.

Najčešće se kao multiplikativni inverz broja a po modulu n razmatra vrednost iz skupa $Z_n = \{1, 2, \ldots, n-1\}$.

U teoremi 3.4.1 je dokazano da inverz po modulu n broja a postoji ako i samo ako su a i n uzajamno prosti i da je u tom slučaju jedinstven po modulu n. Razmotrimo sada kako ga je moguće izračunati.

Problem

Ako su data dva uzajamno prosta pozitivna cela broja a i n, odrediti multiplikativni inverz broja a po modulu n, tj. broj x takav da je $ax \equiv 1 \pmod{n}$.

3.4.3.1 Algoritam grube sile

Pri naivnom pristupu rešavanju ovog problema se za x razmatraju redom svi brojevi x od 1 do $n-1$ (elementi skupa Z_n) i proverava se da li je ostatak pri deljenju broja $a \cdot x$ sa n jednak 1.

```
// direktno racunanje multiplikativnog inverza broja a po modulu n
int modInverz(int a, int n) {
  a = a % n;
  // proveravamo redom kandidate od 1 do n-1
  for (int x = 1; x < n; x++)
     if ((a*x) % n == 1)
       return x;
}
```

Složenost ovog algoritma je $O(n)$.

3.4.3.2 Algoritam zasnovan na proširenom Euklidovom algoritmu

Na osnovu teoreme 3.4.1 modularni inverz broja a po modulu n postoji ako i samo ako su a i n uzajamno prosti. Dokaz naredne teoreme nam govori da se, kada postoji, on može odrediti proširenim Euklidovim algoritmom.

Lema 3.4.5

Ako su a i n uzajamno prosti, tada postoji jedinstven broj x iz skupa $Z_n = \{1, \ldots, n-1\}$ takav da je $a \cdot x \equiv 1 \pmod{n}$.

Dokaz. Da bi broj x bio inverz broja a, treba da važi $a \cdot x \equiv 1 \pmod{n}$ tj. $n|(a \cdot x - 1)$. Zato mora da postoji broj y takav da je $a \cdot x - 1 = y \cdot n$ tj.

$$x \cdot a - y \cdot n = 1.$$

Mi rešavamo ovu Diofantovu jednačinu po nepoznatim celobrojnim koeficijentima x i y i tražimo ono rešenje za koje je $1 \leq x < n$. Pošto su a i n uzajamno prosti, na osnovu Bezuove teoreme 3.1.3, postoji broj x koji ovo zadovoljava. Na osnovu leme 3.1.1, postoje tačno dva rešenja za koje važi $|x| < |n/d|$ i $|y| < |a/d|$ (jedno u kom je x pozitivno i jedno u kom je x negativno), pa, pošto je $d = 1$, postoji tačno jedno rešenje $x \in Z_n$. □

Dakle, pronalaženje modularnog inverza se može svesti na rešavanje jednačine $ax = yn+1$ tj. $ax-ny = 1$, a na osnovu teoreme 3.1.4, znamo da jednačina $ax + by = c$ ima celobrojna rešenja ako i samo ako $d = \text{nzd}(a, b)$ deli broj c. Dakle, za $c = 1$ jednačina $ax - ny = 1$ ima celobrojna rešenja (x, y) ako i samo ako su a i n uzajmno prosti.

Za efikasno izračunavanje modularnog inverza možemo iskoristiti prošireni Euklidov algoritam. Potrebno je da pronađemo koeficijente x i y takve da je $x \cdot a - y \cdot n = 1$, što možemo uraditi primenom proširenog Euklidovog algoritma na dva data, uzajamno prosta broja a i n (on će nam, doduše, vratiti i vrednost $-y$, ali taj koeficijent nas ionako ne zanima). S obzirom na to da dobijena vrednost koeficijenta x može biti i negativna i veća od n, ako želimo da x pripada Z_n, tj. da zadovoljava uslov $0 \leq x < n$, onda umesto x kao modularni multiplikativni inverz vraćamo vrednost $(x \bmod n + n) \bmod n$.

Dakle, određivanje modularnog multiplikativnog inverza broja možemo svesti na prošireni Euklidov algoritam.

```
// racunanje multiplikativnog inverza broja a po modulu m
// koriscenjem prosirenog Euklidovog algoritma
int modInverz(int a, int n) {
  int x, y;
  // određujemo x i y tako da vazi a*x + n*y = d
  int d = nzdProsireni(a, n, x, y);

  // ako a i n nisu uzajamno prosti, onda ne postoji inverz
  if (d != 1)
    cout << "Modularni multiplikativni inverz ne postoji" << endl;
  else{
    // obezbedjujemo da je inverz iz skupa [0, n)
    int inverz = (x % n + n) % n;
    return inverz;
   }
}
```

Primetimo da za računanje modularnog multiplikativnog inverza broja vrednost koeficijenta y nije od interesa, te se može konstruisati algoritam koji ne poziva eksplicitno prošireni Euklidov algoritam, već na isti način kao kod proširenog Euklidovog algoritma iterativno računa samo vrednost koeficijenta x.

```
// funkcija racuna x -- multiplikativni inverz broja a po modulu n,
// koriscenjem prosirenog Euklidovog algoritma
// funkcija vraca da li je uspela da izracuna broj x
bool modInverz(int a, int n, int& x) {
  // pocetne vrednosti za r su n i a
  int r_preth = n;
  int r_tek = a;
  // pocetne vrednosti za x su 0 i 1
  int x_preth = 0;
  int x_tek = 1;
```

```
  while (r_tek > 0) {
    int q = r_preth / r_tek;
    int pom;

    // azuriramo tekucu i prethodnu vrednost niza r
    pom = r_preth;
    r_preth = r_tek;
    r_tek = pom - q * r_tek;

    // azuriramo tekucu i prethodnu vrednost niza x
    pom = x_preth;
    x_preth = x_tek;
    x_tek = pom - q * x_tek;
  }

  // obezbedjujemo da je inverz iz skupa [0,m)
  x = (x_preth % n + n) % n;

  // vracamo true ako je nzd(a, n) = 1, inace false
  return r == 1;
}
```

Složenost prethodnog algoritma za izračunavanje multiplikativnog inverza broja a po modulu n jednaka je složenosti proširenog Euklidovog algoritma za brojeve a i n i iznosi $O(\log(a+n))$, odnosno kada je a reda $O(n)$ jednaka je $O(\log n)$ (što je ekstremno efikasno).

3.4.3.3 Algoritam zasnovan na Ojlerovoj i Maloj Fermaovoj teoremi

Još jedan efikasan način za računanje modularnog multiplikativnog inverza broja je korišćenjem Ojlerove teoreme. Podsetimo se njenog tvrđenja: ako su a i n uzajamno prosti brojevi onda važi: $a^{\varphi(n)} \equiv 1 \pmod{n}$. Primetimo da je uslov da su brojevi a i n uzajamno prosti takođe uslov i da bi postojao multiplikativni inverz broja a po modulu n. Iz prethodne jednačine sledi:

$$a \cdot a^{\varphi(n)-1} \equiv 1 \pmod{n}. \tag{3.5}$$

Primetimo da za proizvoljnu vrednost n važi $\varphi(n) \geq 1$, odnosno $\varphi(n) - 1 \geq 0$, te je $a^{\varphi(n)-1}$ pozitivna celobrojna vrednost. Dakle, za $x = a^{\varphi(n)-1} \bmod n$ važi $a \cdot x \equiv 1 \pmod{n}$, pa je $a^{\varphi(n)-1}$ multiplikativni inverz broja a po modulu n. Dodatno, $a^{\varphi(n)-1}$ može biti veće od n te je kao rezultat potrebno vratiti vrednost $x \bmod n$.

Na ovaj način smo problem računanja modularnog multiplikativnog inverza broja a sveli na računanje Ojlerove funkcije broja n što se dalje svodi na faktorizaciju broja n, koja se osnovnim algoritmom vrši u složenosti $O(\sqrt{n})$, što je neprihvatljivo za brojeve od recimo 100 cifara.

Primer 3.4.9

Vratimo se na primer sa početka ovog poglavlja i pronađimo inverz broja $a = 5$ po modulu $n = 8$. Važi $\varphi(n) = \varphi(8) = 4$ i stoga važi $5^4 \equiv 1 \pmod{8}$. Odatle dobijamo $5 \cdot 5^3 \equiv 1 \pmod{8}$, tj. broj $5^3 \bmod 8 = 125 \bmod 8 = 5$ je multiplikativni inverz broja 5 po modulu 8. Zaista, $5 \cdot 5 = 25 \equiv 1 \pmod{8}$.

Ukoliko je pak broj n prost, tvrđenje Ojlerove teoreme se svodi na Malu Fermaovu teoremu: $a^{n-1} \equiv 1 \pmod{n}$, odakle sledi:

$$a \cdot a^{n-2} \equiv 1 \pmod{n}. \tag{3.6}$$

Pošto je broj n prost, važi $n \geq 2$, odnosno $n - 2 \geq 0$, te je vrednost a^{n-2} celobrojna i pozitivna. Stoga važi da je $x = a^{n-2} \bmod n$ multiplikativni inverz broja a po modulu n. Primetimo da je ovde stvar jednostavnija, jer nam ne treba vrednost Ojlerove funkcije broja n.

Primer 3.4.10

Ako je $n = 7$ *i* $a = 3$, *pošto je broj* m *prost, multiplikativni inverz broja* 3 *po modulu* 7 *možemo naći po formuli* $a^{n-2} \bmod n = 3^5 \bmod 7 = 243 \bmod 7 = 5$ *i zaista važi* $3 \cdot 5 \equiv 1 \pmod{7}$.

Iz kongruencija (3.5) i (3.6) možemo jednostavno odrediti multiplikativni inverz broja a po modulu n. Jedino što preostaje jeste što efikasnije izračunati odgovarajući stepen broja a, što se može izvesti korišćenjem efikasnog algoritma za modularno stepenovanje broja, koji je složenosti $O(\log k)$, gde je k vrednost eksponenta.

Napomenimo da je u situaciji kada n nije prost broj, potrebno izračunati vrednost Ojlerove funkcije broja n što uključuje faktorizaciju broja n i može biti težak problem. Međutim, u situaciji kada je broj n prost, ali takođe i kada broj n nije prost ali je poznata njegova faktorizacija, složenost ovog algoritma je $O(\log n)$.

U nastavku je dat algoritam za računanje multiplikativnog inverza broja a po modulu n za prost broj n.

```
// funkcija za mnozenje po modulu n
int puta_mod(int a, int b, int n) {
  return ((a % n) * (b % n)) % n;
}

// funkcija za brzo stepenovanje po modulu n
int stepen_mod(int a, int b, int n) {
    // a^0 = 1
    if (b == 0)
        return 1;
    // racunamo a^(b/2) mod n
    int rez = stepen_mod(a, b / 2, n);
    // racunamo rez*rez mod n
    rez = puta_mod(rez, rez, n);
    if (b % 2)
      // racunamo rez*a mod n
      return puta_mod(rez, a, n);
    else
        return rez;
}

// racunanje multiplikativnog inverza broja a po modulu n
// koriscenjem Male Fermaove teoreme
int modInverz(int a, int n) {
  return stepen_mod(a, n - 2, n);
}
```

3.5 RSA kriptografija

RSA (Rivest, Shamir, Adleman) je algoritam koji je uveden 1977. godine i koji se i danas intenzivno koristi u mnogim oblastima primene računara. U nastavku ćemo opisati ovaj algoritam i time prikazati kako se algoritmi i koncepti koje smo do sada u ovom poglavlju upoznali lepo kombinuju i daju veoma ozbiljnu, realnu primenu.

Osnovna podela kriptografskih algoritama je na:

- *simetrične*, koji podrazumevaju da se za šifrovanje i dešifrovanje poruka koristi isti ključ i
- *asimetrične*, koji podrazumevaju da se za šifrovanje i dešifrovanje poruka koriste različiti ključevi.

Obe vrste algoritama se koriste u savremenoj elektronskoj komunikaciji, najčešće zajedno. Simetrična kriptografija je brža, ali je veliki problem razmena ključeva između stranaka koje komuniciraju. Asimetrična kriptografija ne zahteva razmenu ključeva, ali je sporija. Obično se danas asimetrična kriptografija upotrebljava u prvoj fazi komunikacije tokom koje se ustanovljava identitet stranaka koje komuniciraju i vrši razmena simetričnog ključa koji se dalje koristi u drugoj fazi komunikacije (na primer, tako radi protokol TLS, koji je osnova kriptovane komunikacije na vebu).

Asimetrična kriptografija podrazumeva da osoba koja prima poruke ima svoj:

- *javni ključ* koji je poznat svima i koji onaj ko šalje poruku koristi za šifrovanje, i
- *tajni ključ* koji je samo njoj poznat koji ona koristi za dešifrovanje.

RSA se smatra začetnikom asimetrične kriptografije. I šifrovanje i dešifrovanje se vrše pomoću stepenovanja po modulu n. U algoritmu RSA javni ključ je par brojeva (e, n), a tajni ključ (d, n) (koristimo slovo e za *encryption*, tj. šifrovanje, a d za *decryption*, tj. dešifrovanje)[7]. Šifruje se poruka (*message*) m i dobija se šifrat (*cypher*) c tako da važi $c = m^e \bmod n$. Veći dokumenti se dele na manje poruke i svaka poruka se zasebno šifruje. Veličina poruke zavisi od veličine ključeva (ako se brojevi d i e predstavljaju sa po 4096 bitova, poruke m su obično oko 500 bitova). Ključno svojstvo koje mora biti ispunjeno je da se operacije šifrovanja i dešifrovanja poništavaju, tj. da se dešifrovanjem šifrata c izračunavanjem $c^d \bmod n$ dobija originalna poruka m. Da bi ovo svojstvo važilo, dovoljno je da brojevi e i d budu međusobno inverzni po modulu $\varphi(n)$. Ovo je jednostavna posledica Ojlerove teoreme i dokazaćemo je u narednoj teoremi.

Teorema 3.5.1 **[RSA: dešifrovanje šifrovane poruke]**

Ako su brojevi e i d međusobno inverzni po modulu $\varphi(n)$ tj. ako je $e \cdot d \equiv 1 \pmod{\varphi(n)}$ i ako je poruka $m < n$ uzajamno prosta sa brojem n, tada važi $c^d = (m^e)^d \equiv m \pmod{n}$.

Dokaz. Pošto je $e \cdot d \equiv 1 \pmod{\varphi(n)}$ broj $e \cdot d - 1$ je deljiv sa $\varphi(n)$ tj. postoji k takvo da je $e \cdot d = k\varphi(n) + 1$. Pošto su m i n uzajamno prosti, na osnovu Ojlerove teoreme 3.4.2 važi da je $m^{\varphi(n)} \equiv 1 \pmod{n}$. Zato je

$$c^d \equiv (m^e)^d = m^{e \cdot d} = m^{k\varphi(n)+1} = (m^{\varphi(n)})^k \cdot m \equiv 1^k \cdot m = m \pmod{n}$$

□

Dakle, ako javni ključ sadrži broj e, tada deo tajnog ključa d treba da izaberemo kao njegov inverz po modulu $\varphi(n)$. Pretpostavljamo da su brojevi e i n poznati napadaču, ali želimo da on ne može u razumnom vremenu da izračuna broj d na osnovu njihove vrednosti. Šta je to što će nama omogućiti da prilikom generisanja ključeva izračunamo d, a što nedostaje napadaču da bi i on to mogao efikasno da uradi? To što mi u trenutku generisanja ključeva možemo da znamo vrednost $\varphi(n)$, a napadaču je potrebno ogromno vreme da od n izračuna $\varphi(n)$. Šta bi nama omogućilo da brzo izračunamo $\varphi(n)$? Ako bismo znali faktorizaciju broja n,

[7] Može se smatrati i da su ključevi brojevi e i d, međutim, pošto je n potrebno za izračunavanje stepena po modulu, i njega ćemo smatrati delom oba ključa.

onda bi izračunavanje Ojlerove funkcije bilo jako jednostavno. Pretpostavićemo zato da smo broj n odabrali u startu tako da bude jednak proizvodu neka dva prosta broja p i q. Njih ćemo iskoristiti prilikom generisanja ključeva e i d i zaboraviti odmah nakon toga (nigde ih nećemo zapisati). Poznavanje faktorizacije $n = p \cdot q$ nam omogućava da efikasno izračunamo vrednost $\varphi(n) = (p-1) \cdot (q-1)$ (koju ćemo takođe zaboraviti odmah nakon generisanja ključeva), a zatim da na osnovu vrednosti dela javnog ključa e izračunamo njegov modularni inverz d (po modulu $\varphi(n)$), što se lako radi korišćenjem proširenog Euklidovog algoritma.

Dakle, algoritam RSA radi na sledeći način.

1. Generisanje ključeva započinje generisanjem dva velika prosta broja p i q i određivanjem vrednosti modula $n = p \cdot q$. Brojevi p i q su tajni, jednokratno se upotrebljavaju i odmah zaboravljaju (nigde se ne skladište).
2. Izračunava se vrednost Ojlerove funkcije $\varphi(n)$ (i ona je tajna, upotrebljava se jednokratno i nigde ne skladišti). U pogavlju 3.3 o multiplikativnim funkcijama je dokazano da je u slučaju $n = p \cdot q$ vrednost $\varphi(n)$ jednaka $(p-1) \cdot (q-1)$, tako da se ona lako izračunava kada se znaju brojevi p i q.
3. Bira se vrednost javnog ključa e, takva da je $2 \leq e < \varphi(n)$ i da je e uzajamno prost sa $\varphi(n)$. Vrednost e se često bira kao mali prost broj (kada je e mali broj, stepenovanje se brže vrši, a to što je prost obezbeđuje da se lakše ispita da li je uzajamno prost sa $\varphi(n)$).
4. Vrednost d se određuje kao modularni multiplikativni inverz vrednosti e po modulu $\varphi(n)$ tj. tako da važi da je $e \cdot d \equiv 1 (\mathrm{mod}\ \varphi(n))$. Ona postoji (jer je e odabran tako da je uzajamno prost sa $\varphi(n)$) i može se efikasno izračunati nekim od ranije opisanih algoritama (pre svega proširenim Euklidovim algoritmom). Ako se e odabere kao mali broj, vrednost d može biti prilično veliki broj, što znači da će se šifrovanje vršiti brže nego dešifrovanje.
5. Poruka m se šifruje izračunavanjem vrednosti $c = (m^e) \bmod n$.
6. Poruka se dešifruje izračunavanjem vrednosti $m = (c^d) \bmod n$.

Dešifrovanje je korektno na osnovu teoreme 3.5.1. Uslov da je m uzajamno prost sa n se lako obezbeđuje. Ako je m manji od prostih brojeva p i q, on je uzajamno prost sa njima, pa i sa njihovim proizvodom. Čak i ako se za m uzme neka veća vrednost, verovatnoća da ona bude umnožak nekog od brojeva p ili q je strašno mala.

Primer 3.5.1

Ilustrujmo prethodni postupak na jednom primeru. Neka je $p = 61$ *i* $q = 53$. *Onda je* $n = p \cdot q = 3233$ *i važi* $\varphi(n) = (p-1) \cdot (q-1) = 60 \cdot 52 = 3120$. *Za broj* e *biramo broj manji od* 3120 *koji je uzajamno prost sa* 780, *na primer* $e = 17$. *Modularni multiplikativni inverz broja* 17 *po modulu* 3120 *jednak je* $d = 413$ *(njega možemo efikasno izračunati proširenim Euklidovim algoritmom). Šifrovanje poruke* m *svodi se na računanje* $c = m^{17} \bmod 3233$ *a dešifrovanje poruke* c *na računanje* $m = c^{413} \bmod 3233$. *Na primer, šifrovanjem poruke* $m = 65$ *dobija se* $c = 65^{17} \bmod 3233 = 2790$, *a dešifrovanjem ove poruke dobijamo* $m = 2790^{413} \bmod 3233 = 65$, *što jeste vrednost polazne poruke.*

RSA se može upotrebiti i za potpisivanje poruka. Da bi postupak bio brži, umesto potpisivanje cele poruke obično se potpisuje samo njena heš-vrednost. Na osnovu poruke m može se izračunati njena heš-vrednost $h(m)$ i uz originalnu poruku može se poslati potpis $h(m)^d$ (primetimo da se heš-vrednost stepenuje korišćenjem tajnog ključa pošaljioca d, koji se inače koristi za dešifrovanje). Primalac poruke m dešifruje potpis $h(m)$ korišćenjem javnog ključa e pošiljaoca (koji mu je poznat), stepenujući dobijeni potpis na e. Time će rekonstruisati vrednost $h(m)$. Takođe, na osnovu poruke m, primalac može ponovo da izračuna vrednost $h(m)$ i ako se ta vrednost poklopi sa onom dobijenom dešifrovanjem potpisa, može da tvrdi da poruka m nije u međuvremenu menjana, kao i da ju je poslala osoba koja je bila u posedu tajnog ključa d.

RSA algoritam, dakle, počiva na sledećim činjenicama:

- Ako su e i d inverzni elementi po modulu $\varphi(n)$ tj. ako je $e \cdot d \equiv 1 \pmod{\varphi(n)}$, tada (na osnovu Ojlerove teoreme) važi $m^{ed} \equiv m \pmod{n}$, za sve brojeve m uzajamno proste sa n.
- Ako se zna broj $\varphi(n)$, tada se multiplikativni inverz broja e po modulu $\varphi(n)$ može brzo izračunati.
- Ako se znaju prosti brojevi p i q takvi da je $n = p \cdot q$, tada se vrednost $\varphi(n)$ može izračunati jako brzo.
- Ako se zna vrednost n, ali ne i brojevi p i q tada se vrednosti p i q, niti vrednost $\varphi(n)$ praktično ne može izračunati.
- Ako se nekom greškom sazna par vrednosti poruka m i šifrat $c = m^e \bmod n$, to ne omogućava da se u realnom vremenu odredi ključ d. Naime, iako važi $c^d \bmod n = m$, pronalaženje d zahteva izračunavanje diskretnog logaritma, što praktično nije moguće uraditi.

Sigurnost algoritma RSA se, dakle, zasniva na težini faktorizacije velikih brojeva, težini izračunavanja Ojlerove funkcije i težini izračunavanja diskretnog logaritma. Naime, iako je informacija o broju n javna, ona nam ne omogućava da efikasno izračunamo vrednost $\varphi(n)$ koja je potrebna da bi se od javnog ključa e izračunao tajni ključ d (kao njegov inverz po modulu $\varphi(n)$). Sigurnost sistema RSA bi se narušila ako bi se broj n mogao efikasno faktorisati ili ako bi se $\varphi(n)$ moglo izračunati bez faktorizacije broja n, na neki efikasniji način. Pretpostavimo da je n jednako proizvodu dva velika prosta broja p i q, koji nemaju sličan red veličine. Tada je problem faktorizacije broja $n = p \cdot q$ jako težak. U poglavljima 3.2 i 3.3 posvećenim faktorizaciji brojeva i njenim primenama prikazani su algoritmi faktorizacije broja n i izračunavanja $\varphi(n)$ čija je složenost jednaka $O(\sqrt{n})$, što je za velike vrednosti broja p i q (vrednosti sa nekoliko stotina, pa i hiljada binarnih cifara) neizvodivo u realnom vremenu. Tada ni najnapredniji poznati algoritmi nemaju nikakvu šansu da izvrše faktorizaciju, niti da izračunaju vrednost $\varphi(n)$ u nekom realnom vremenu (danima, mesecima, godinama), čak ni uz masovnu paralelizaciju. Naglasimo da brojevi p i q ne smeju da budu sličnog reda veličine tj. da imaju sličan broj cifara, inače bi bili bliski vrednosti $\sqrt{n}$ i mogli bi se brzo odrediti Fermaovim algoritnom faktorizacije (koji je opisan u poglavlju 3.2.1).

Faktorizacija omogućava jednostavno izračunavanje Ojlerove funkcije broja $n = p \cdot q$, ali važi i obratno. Ako je pored broja n poznata i vrednost Ojlerove funkcije $\varphi(n)$, onda se brojevi p i q mogu jednostavno odrediti. Naime, pošto su brojevi p i q prosti oni su i uzajamno prosti, pa važi $\varphi(n) = (p-1) \cdot (q-1)$. Množenjem izraza sa desne strane znaka jednakosti dobijamo da je $n - \varphi(n) + 1 = p + q$. Na osnovu vrednosti $p + q$ i $p \cdot q$ lako se mogu odrediti brojevi p i q kao rešenja kvadratne jednačine $x^2 - (n - \varphi(n) + 1)x + n = 0$.

Za sada nije poznat efikasan algoritam za faktorizaciju velikih brojeva n niti za izračunavanje vrednosti $\varphi(n)$, ali nije ni dokazano da takav algoritam ne postoji (ovo je slično kao kod NP kompletnih problema, međutim, nije dokazano da su ovi problemi NP kompletni – ovi problemi su u klasi NP, ali je moguće da su lakši od NP kompletnih problema). Kada bi se našao efikasan algoritam kojim se rešava neki od ovih problema, to bi moglo imati ozbiljan uticaj na sigurnost i bezbednost podataka.

3.6 Kineska teorema o ostacima

Prema jednoj kineskoj legendi, general Han Xin je, da bi izbegao da špijuni saznaju sa kolikom je vojskom krenuo u boj, svom kuriru davao samo informaciju o ostatku broja vojnika u pravougaonoj formaciji. Njegove poruke su bile u narednoj formi: "Kada se vojnici organizuju u redove od po 9, preostaje njih 4; ako su u redovima od po 11, niko ne preostaje; ako su u redovima od po 13, samo jedan preostaje, a ako su u redovima od po 19, preostaje njih trojica". General je takođe svom kuriru preneo da je broj vojnika drugo najmanje rešenje ovog problema. Pitanje koje se postavlja je kolikim brojem vojnika je general zapovedao. Ovaj problem odgovara tzv. *Kineskoj teoremi o ostacima*, jednom od standardnih problema elementarne teorije brojeva. Formulišimo ovaj problem u standardnim terminima.

Problem

Data su dva niza brojeva r: $r_0, r_1, \ldots, r_{n-1}$ *i m:* $m_0, m_1, \ldots, m_{n-1}$ *pri čemu za svaki par brojeva niza* m *važi da su uzajamno prosti. Odrediti najmanji pozitivan broj* x *za koji važi:*

$$\begin{aligned} x \bmod m_0 &= r_0 \\ x \bmod m_1 &= r_1 \\ &\dots \\ x \bmod m_{n-1} &= r_{n-1} \end{aligned}$$

Traži se, dakle, da se reši sledeći sistem kongruencija po modulu.

$$\begin{aligned} x &\equiv r_0 \pmod{m_0} \\ x &\equiv r_1 \pmod{m_1} \\ &\dots \\ x &\equiv r_{n-1} \pmod{m_{n-1}} \end{aligned} \tag{3.7}$$

Teorema 3.6.1 **[Kineska teorema o ostacima]**

Postoji broj x koji zadovoljava sistem kongruencija (3.7) i on je jedinstven po modulu $M = m_0 \cdot m_1 \cdot \dots \cdot m_{n-1}$.

3.6.1 Gruba sila

U naivnom pristupu rešavanju ovog problema razmatraju se redom brojevi od 1 naviše po skupu prirodnih brojeva i proverava se da li tekući broj zadovoljava date uslove. Pošto uvek postoji broj koji zadovoljava ovaj skup jednačina, na ovaj način bismo došli do rešenja, ali u broju koraka proporcionalnom traženoj vrednosti x, što je u najgorem slučaju $O(M)$.

```
int izracunajMinX(vector<int> m, vector<int> r) {
  int n = m.size();
  // inicijalizujemo vrednost x
  int x = 1;

  // prema tvrđenju Kineske teoreme o ostacima
  // ova petlja ce se uvek zaustaviti
  while (true) {
    int j;
    // za svako i od 0 do n-1 proveravamo
    // da li je ostatak pri deljenju broja x sa m_i jednak r_i
    for (i = 0; i < n; i++)
      if (x % m[i] != r[i])
        break;

    // ako su sve iteracije prosle uspesno
    // nasli smo vrednost za x
    if (i == n)
      return x;
    // inace nastavljamo sa pretragom
    x++;
  }
  return x;
}

int main(void) {
  vector<int> m = {3, 4, 5};
  vector<int> r = {2, 3, 1};
```

```
  // m i r moraju biti istih dimenzija
  if (m.size() != r.size()) {
    cout << "Broj modula mora biti jednak broju ostataka" << endl;
  }
  else
    cout << "Najmanje x koje zadovoljava dati sistem kongruencija je "
         << izracunajMinX(m, r) << endl;
  return 0;
}
```

3.6.2 Algoritam zasnovan na prosejavanju

Postoji jedan postupak koji nije previše matematički zahtevan, ali koji nije optimalan u smislu efikasnosti (no, mnogo je bolji od grube sile i često uspeva da reši problem unutar zadatih vremenskih ograničenja). Pristup je zasnovan na pametnoj pretrazi. Ako broj x sa brojem m_0 daje ostatak r_0 onda je on sigurno jedan od članova aritmetičkog niza r_0, $r_0 + m_0$, $r_0 + 2m_0$, ... U petlji redom proveravamo ove brojeve sve dok ne naiđemo na prvi element koji pri deljenju sa m_1 daje ostatak r_1. Kada takav broj r_{01} nađemo (on će biti najmanji pozitivan broj sa osobinom da pri deljenju sa m_0 i m_1 daje redom ostatke r_0 i r_1), znamo da će svaki naredni broj koji zadovoljava to svojstvo biti član aritmetičkog niza r_{01}, $r_{01} + m_0 \cdot m_1$, $r_{01} + 2 \cdot m_0 \cdot m_1$, ... Redom pretražujemo elemente ovog niza dok ne naiđemo na element r_{012} koji pri deljenju sa m_2 daje ostatak r_2. Postupak se nastavlja tako što se nakon pronalaženja r_{012} posmatraju brojevi r_{012}, $r_{012} + m_0 \cdot m_1 \cdot m_2$, $r_{012} + 2 \cdot m_0 \cdot m_1 \cdot m_2$ i tako dalje, sve dok se ne nađe broj koji zadovoljava sva data ograničenja.

Primer 3.6.1

Prikažimo prethodni algoritam na jednom primeru. Neka je $(r_0, m_0) = (2, 3)$, $(r_1, m_1) = (3, 5)$ *i* $(r_2, m_2) = (2, 7)$. *Važi da je* $M = m_0 \cdot m_1 \cdot m_2 = 105$.

Posmatramo prvo aritmetički niz $r_0 + k \cdot m_0$ *čiji su članovi* 2, 5, 8, 11 *itd. i u njemu tražimo prvi broj koji pri deljenju sa* $m_1 = 5$ *daje traženi ostatak* $r_1 = 3$. *Prvi takav broj je* $r_{01} = 8$.

Sada posmatramo niz $r_{01} + k \cdot (m_0 \cdot m_1)$ *tj. niz* 8, 23, 38 *itd. i u njemu tražimo prvi element koji pri deljenju sa* $m_2 = 7$ *daje traženi ostatak* 2. *To je broj* $r_{012} = 23$, *koji je konačan rezultat.*

3.6.3 Algoritam zasnovan na Lagranževom pristupu

Dokažimo tvrđenje kineske teoreme o ostacima. Dokaz je konstruktivan i daje opšti algoritam za konstruisanje traženog broja x.

Dokaz. Osnovna ideja je ista kao i u konstrukciji Lagranžovog interpolacionog polinoma. Pretpostavimo da umemo da odredimo cele brojeve $w_0, w_1, \ldots, w_{n-1}$ takve da za svako i važi da w_i pri deljenju sa m_i daje ostatak 1, dok pri deljenju sa svim drugim brojevima m_j, $j \neq i$ daje ostatak 0, tj. da imamo niz brojeva w_i koji zadovoljava uslove date u narednoj tabeli.

	mod m_0	mod m_1	…	mod m_{n-1}
w_0	1	0	…	0
w_1	0	1	…	0
…	…	…	…	…
w_{n-1}	0	0	…	1

Množenjem brojeva w_i sa r_i dobijaju se brojevi pri deljenju sa m_j daju ostatak 0 za $j \neq i$ i r_i za $j = i$. Zato se jedno x koje zadovoljava dati sistem kongruencija može konstruisati kao $x = r_0 \cdot w_0 + ... + r_{n-1} \cdot w_{n-1}$. Neka je $M = m_0 \cdot m_1 \cdot ... \cdot m_{n-1}$. Najmanje rešenje x koje zadovoljava sistem kongruencija se dobija kao $x = (r_0 \cdot w_0 + ... + r_{n-1} \cdot w_{n-1}) \bmod M$ i dokazaćemo da je to jedino rešenje manje od broja M.

Pokazaćemo sada na koji način se mogu konstruisati brojevi w_i. Pošto broj w_i mora da bude deljiv svim brojevima m_j za $0 \leq j < n$ i $j \neq i$, a oni su uzajamno prosti, on mora biti neki umnožak njihovog proizvoda. Uvedimo oznake za te proizvode. Neka je:

$$z_0 = \frac{M}{m_0},\ z_1 = \frac{M}{m_1},\ ...,\ z_{n-1} = \frac{M}{m_{n-1}}$$

Dakle, z_i je proizvod svih brojeva m_j za $0 \leq j < n$ i $j \neq i$. Za svako $0 \leq i < n$ važi sledeće:

1. $z_i \equiv 0 \pmod{m_j}$ za $j \neq i$;
2. z_i i m_i su uzajamno prosti brojevi.
3. z_i pri deljenju sa m_i ne daje ostatak 1, nego $z_i \bmod m_i$ (koji može, ali ne mora biti jednak 1).

Prva dva uslova nam odgovaraju, ali posednji ne. Da bismo od ostatka $z_i \bmod m_i$ dobili željeni ostatak 1, dobijeni proizvod z_i treba nekako podeliti brojem $z_i \bmod m_i$. Da radimo sa realnim brojevima vrednost 1 bismo dobii realnim deljenjem sa $z_i \bmod m_i$ tj.
množenjem sa koeficijentom $1/(z_i \bmod m_i)$. U modularnoj aritmetici deljenje ne postoji, ali postoji nešto analogno: množenje inverznim elementom. Stvar zato možemo popraviti tako što z_i pomnožimo modularnim inverzom broja $z_i \bmod m_i$ (koji je isti kao modularni inverz broja z_i po modulu m_i). Naime, množenje bilo kojim brojem ne može narušiti deljivost i svi ostaci koji su bili nula ostaće nula. Broj z_i tj. $z_i \bmod m_i$ ima modularni multiplikativni inverz po modulu m_i (jer su z_i i m_i uzajamno prosti), pa njegovo množenje tim inverzom daje vrednost 1 po modulu m_i. Dakle, koeficijent kojim množimo treba da bude jednak modularnom multiplikativnom inverzu broja z_i tj. $z_i \bmod m_i$ po modulu m_i. Obeležimo taj inverz sa y_i. Važi:

$$\begin{aligned} &y_0 \equiv z_0^{-1} \pmod{m_0} \\ &y_1 \equiv z_1^{-1} \pmod{m_1} \\ &... \\ &y_{n-1} \equiv z_{n-1}^{-1} \pmod{m_{n-1}} \end{aligned}$$

Ove inverze možemo odrediti, na primer, proširenim Euklidovim algoritmom. Dakle, ako umesto z_i posmatramo brojeve $y_i \cdot z_i$ dobijamo brojeve za koje važe oba željena svojstva:

1. $y_i \cdot z_i \equiv 0 \pmod{m_j}$ za $j \neq i$;
2. $y_i \cdot z_i \equiv 1 \pmod{m_i}$.

Ova svojstva nam omogućavaju da definišemo brojeve $w_0, w_1, ..., w_{n-1}$ koji zadovoljavaju zahteve date u prethodnoj tabeli na sledeći način (računanjem ostatka sa M svojstva ostaju da važe, a vrednosti brojeva se potencijalno smanjuju što olakšava račun):

$$\begin{aligned} w_0 &= (y_0 \cdot z_0) \bmod M \\ w_1 &= (y_1 \cdot z_1) \bmod M \\ &... \\ w_{n-1} &= (y_{n-1} \cdot z_{n-1}) \bmod M \end{aligned}$$

Neposredno se proverava da broj

$$x = (r_0 \cdot w_0 + ... + r_{n-1} \cdot w_{n-1}) \bmod M$$

predstavlja rešenje sistema (3.7) tj. daje ostatak r_i pri deljenju sa m_i. Naime,

$$\begin{aligned} x \bmod m_i &= ((r_0 y_0 z_0 + r_1 y_1 z_1 + ... + r_{n-1} y_{n-1} z_{n-1}) \bmod M) \bmod m_i \\ &= (r_0 y_0 z_0 + r_1 y_1 z_1 + ... + r_{n-1} y_{n-1} z_{n-1}) \bmod m_i \\ &= r_i y_i z_i \bmod m_i \\ &= r_i \end{aligned}$$

Druga jednakost u ovom nizu važi na osnovu toga što je $M \bmod m_i = 0$, treća na osnovu toga što je $y_j \cdot z_j \equiv 0 \pmod{m_i}$ za $j \neq i$, a četvrta na osnovu svojstva $y_i \cdot z_i \equiv 1 \pmod{m_i}$ i činjenice da je $0 \leq r_i < m_i$.

Pokažimo da sva rešenja sistema (3.7) daju isti ostatak pri deljenju sa M. Razmotrimo dva rešenja x_1 i x_2. Važi:

$$m_0 \mid (x_1 - x_2),\ m_1 \mid (x_1 - x_2), ...,\ m_{n-1} \mid (x_1 - x_2).$$

S obzirom na to da su svi $m_0, m_1, ..., m_{n-1}$ uzajamno prosti, važi i da

$$m_0 \cdot m_1 \cdot ... \cdot m_{n-1} \mid (x_1 - x_2).$$

Dakle, važi da je $x_1 \equiv x_2 \pmod{M}$, tj. rešenje je jedinstveno po modulu $M = m_0 \cdot m_1 \cdot ... \cdot m_{n-1}$.

Pošto u intervalu $[0, M)$ ne postoji broj različit od $x = (r_0 \cdot w_0 + ... + r_{n-1} \cdot w_{n-1}) \bmod M$ koji je kongruentan sa x po modulu M, x je jedino pozitivno rešenje sistema (3.7) manje od M. Time je dokazana Kineska teorema o ostacima. □

Na ovoj ideji zasniva se i efikasniji algoritam za rešavanje polaznog problema. Pretpostavljamo da su brojevi m_i i r_i predstavljeni mašinskim tipovima i da je n mali broj, pa se izračunavanje proizvoda M i z_i vrši u složenosti $O(1)$. Najzahtevnija operacija je pronalaženje modularnih inverza brojeva z_i, i ona se izvršava u vremenu $O(\log M)$, što je i složenost celog algoritma.

Primer 3.6.2

Vratimo se na polazni primer i izračunajmo kolikom je vojskom general Han Xin rukovodio. Ako sa x označimo broj vojnika, onda zadate uslove možemo da zapišemo na sledeći način:

$$\begin{aligned} x &\equiv 4 \pmod{9} \\ x &\equiv 0 \pmod{11} \\ x &\equiv 1 \pmod{13} \\ x &\equiv 3 \pmod{19} \end{aligned}$$

Označimo sa $M = 9 \cdot 11 \cdot 13 \cdot 19 = 24453$ i sa: $z_0 = \frac{M}{9} = 2717$, $z_1 = \frac{M}{11} = 2223$, $z_2 = \frac{M}{13} = 1881$, $z_3 = \frac{M}{19} = 1287$.

Izračunajmo modularne multiplikativne inverze ovih brojeva:

$$\begin{aligned} y_0 &= z_0^{-1} = 2717^{-1} \bmod 9 = 8^{-1} \bmod 9 = 8 \\ y_1 &= z_1^{-1} = 2223^{-1} \bmod 11 = 1^{-1} \bmod 11 = 1 \\ y_2 &= z_2^{-1} = 1881^{-1} \bmod 13 = 9^{-1} \bmod 13 = 3 \\ y_3 &= z_3^{-1} = 1287^{-1} \bmod 19 = 14^{-1} \bmod 19 = 15 \end{aligned}$$

Na osnovu ovih vrednosti, možemo izračunati potrebne brojeve w_i:

$$\begin{aligned} w_0 &= y_0 \cdot z_0 \bmod M = 8 \cdot 2717 \bmod 24453 = 21736 \\ w_1 &= y_1 \cdot z_1 \bmod M = 1 \cdot 2223 \bmod 24453 = 2223 \\ w_2 &= y_2 \cdot z_2 \bmod M = 3 \cdot 1881 \bmod 24453 = 5643 \\ w_3 &= y_3 \cdot z_3 \bmod M = 15 \cdot 1287 \bmod 24453 = 19305 \end{aligned}$$

Napokon dobijamo:

$$\begin{aligned} x &= (r_0 \cdot w_0 + \ldots + r_{n-1} \cdot w_{n-1}) \bmod M \\ &= (4 \cdot 21736 + 0 \cdot 2223 + 1 \cdot 5643 + 3 \cdot 19305) \bmod 24453 \\ &= 3784 \end{aligned}$$

S obzirom na to da je rečeno da je broj vojnika drugo najmanje rešenje ovog problema, ukupan broj vojnika jednak je $x = 3784 + 24453 = 28237$

Razmotrimo sada implementaciju rešenja polaznog problema zasnovanog na tvrđenju Kineske teoreme o ostacima. Zasebno su implementirane sabiranja i množenja po modulu (ovim se ostatak pri deljenju računa i češće nego što je to zaista neophodno, ali je dobijeni program pregledniji).

```
// proizvod brojeva x i y po modulu m
long long pm(long long x, long long y, long long m) {
  return ((x % m) * (y % m)) % m;
}

// proizvod brojeva x, y i z po modulu m
// dva puta se poziva funkcija za racunanje proizvoda po modulu m
long long pm(long long x, long long y, long long z, long long m) {
  return pm(pm(x, y, m), z, m);
}

// zbir brojeva x i y po modulu m
long long zm(long long x, long long y, long long m) {
  return ((x % m) + (y % m)) % m;
}

// na osnovu Kineske teoreme o ostacima se izracunava rezultat tako da
// za sve elemente nizova m i a vazi da je rezultat mod m[i] = r[i]
// funkcija vraca da li je rezultat bilo moguce naci
bool kto(int m[], int r[], int n, long long& rezultat) {
  // racunamo proizvod svih modula
```

```
  long long M = 1;
  for (int i = 0; i < n; i++)
    M *= m[i];

  rezultat = 0;
  for (int i = 0; i < n; i++) {
    // racunamo zi
    long long zi = M / m[i];
    long long yi;
    // racunamo yi kao inverz broja zi po modulu m[i]
    if (!modInverz(zi, m[i], yi))
      return false;
    // na rezultat dodajemo sabirak ri*yi*zi mod M
    rezultat = zm(rezultat, pm(r[i], yi, zi, M), M);
  }
  return true;
}

int main() {
  // ucitavamo ulazne podatke
  int r[3], m[3];
  for (int i = 0; i < 3; i++)
    cin >> r[i] >> m[i];

  // rezultat izracunavamo na osnovu Kineske teoreme o ostacima
  long long rezultat;
  if (kto(m, r, 3, rezultat))
    // ispisujemo rezultat
    cout << rezultat << endl;
  return 0;
}
```

3.6.4 Algoritam zasnovan na Bezuovoj teoremi

Pored opšteg postupka za rešavanje problema za k ostataka, u slučaju samo dva ostatka postoji i veoma sličan, ali malo drugačije formulisan postupak, koji se zasniva na tome da umemo da rešimo sistem od dve jednačine (dve kongruencije), primenom Bezuove teoreme.

Za uzajamno proste brojeve m_1 i m_2, na osnovu Bezuove teoreme mogu se pronaći brojevi y_1 i y_2 takvi da važi da je $y_1 \cdot m_1 + y_2 \cdot m_2 = 1$. Može se lako pokazati da koeficijent y_1 predstavlja modularni inverz broja m_1 po modulu m_2, dok koeficijent y_2 predstavlja modularni inverz broja m_2 po modulu m_1. Zato broj $x_{12} = r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1$ tj. njegov ostatak po modulu $m_1 \cdot m_2$, eventualno uvećan za $m_1 \cdot m_2$ ako je negativan, pri deljenju sa m_1 daje ostatak r_1, a pri deljenju sa m_2 daje ostatak r_2.

Važi da je

$$\begin{aligned} x_{12} &= r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1 \\ &= r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1 + r_1 \cdot y_1 \cdot m_1 - r_1 \cdot y_1 \cdot m_1 \\ &= r_1 \cdot (y_2 \cdot m_2 + y_1 \cdot m_1) + y_1 \cdot m_1 \cdot (r_2 - r_1) \\ &= r_1 + y_1 \cdot m_1 \cdot (r_2 - r_1) \end{aligned}$$

Zato je ostatak pri deljenju x_{12} brojem m_1 jednak r_1.

Slično se dokazuje i da je $x_{12} = r_2 + y_2 \cdot m_2 \cdot (r_1 - r_2)$ pa je ostatak pri deljenju x_{12} brojem m_2 jednak r_2.

Dakle, ako Kinesku teoremu o ostacima primenjujemo na dva broja, ne moramo dva puta nezavisno računati modularni inverz, nego oba potrebna koeficijenta možemo dobiti jednom primenom proširenog Euklidovog algoritma.

Ako ima više brojeva na koje je potrebno primeniti Kinesku teoremu o ostacima, nakon određivanja rešenja za prva dva, isti postupak se primenjuje da se odredi broji koji pri deljenju sa $m_1 \cdot m_2$ daje ostatak x_{12}, a pri deljenju sa m_3 daje ostatak r_3. Ako je $k > 3$, postupak se nastavlja na isti način, dok se ne dobije konačan rezultat.

Primetimo da se u formuli $(r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1) \bmod (m_1 \cdot m_2)$ množe dva puta po tri broja, kao i da je modul $m_1 \cdot m_2$ prilično veliki broj, pa privremeni rezultati mogu da prekorače opseg 64-bitnog tipa, čak iako su svi r_i i m_i 32-bitni celih brojevi i ako se moduli računaju pre i nakon svake operacije množenja. To se može predupredititi ako se primeti da je $(r_1 \cdot y_2 \cdot m_2) \bmod (m_1 \cdot m_2) = m_2 \cdot ((r_1 \cdot y_2) \bmod m_1)$ i da je $(r_2 \cdot y_1 \cdot m_1) \bmod (m_1 \cdot m_2) = m_1 \cdot ((r_2 \cdot y_1) \bmod m_2)$.

Primer 3.6.3

Prikažimo prethodni algoritam na jednom primeru. Neka je $(r_1, m_1) = (2, 3)$, $(r_2, m_2) = (3, 5)$ *i* $(r_3, m_3) = (2, 7)$. *Važi da je* $M = m_0 \cdot m_1 \cdot m_2 = 105$.

Pronađimo prvo brojeve y_1 *i* y_2 *takve da je* $y_1 \cdot m_1 + y_2 \cdot m_2 = 3y_1 + 5y_2 = 1$. *Važi, na primer, da je* $y_1 = 2$ *i* $y_2 = -1$, *jer je* $3 \cdot 2 + 5 \cdot (-1) = 1$. *Traženi broj je* $x_{12} = r_1 \cdot y_2 \cdot m_2 + r_2 \cdot y_1 \cdot m_1 = 2 \cdot (-1) \cdot 5 + 3 \cdot 2 \cdot 3 = 8$. *On pri deljenju sa* 3 *daje ostatak* 2, *a pri deljenju sa sa* 5 *daje ostatak 3.*

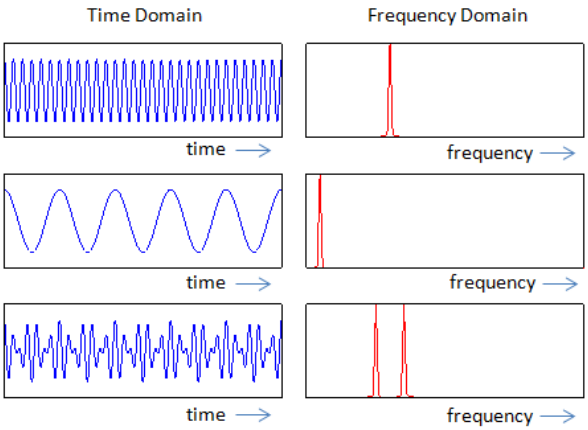
U narednom koraku tražimo broj takav da pri deljenju sa $m_1 \cdot m_2 = 15$ *daje ostatak* $r_{12} = x_{12} = 8$, *a pri deljenju sa* $m_3 = 7$ *daje ostatak* $r_3 = 2$. *Zato određujemo brojeve* y_1' *i* y_2' *takve da je* $15y_1' + 7y_2' = 1$. *To mogu biti brojevi* $y_1' = 1$ *i* $y_2' = -2$. *Traženi broj je tada* $8 \cdot 7 \cdot (-2) + 2 \cdot 15 \cdot 1 = -82$ *tj.* $-82 + 105 = 23$ *(pošto je* -82 *negativan uvećavamo ga za* $(m_1 m_2) m_3 = 105$*).*

3.7 Brza Furijeova transformacija

Brza Furijeova transformacija (ili *FFT*, što je skraćenica od engl. *Fast Fourier transform*) od svog otkrića sredinom šezdesetih godina dvadesetog veka spada u najznačajnije algoritme 20. veka.[8] Iako su značajne primene brze Furijeove transformacije nastale nakon rada Kulija i Tukija 1965. godine, na polju obrade signala (u cilju detektovanja nuklearnih proba obradom seizmičkih merenja), ispostavilo se da su ova dva autora nezavisno iznova "izmislili" algoritam koji je osmislio Gaus još davne 1805. godine. Brza Furijeova transformacija ima brojne primene u inženjerstvu, u obradi digitalnih signala poput obrade zvuka i obrade digitalnih slika i u matematici. *Furijeovom transformacijom* neprekidnog signala vrši se njegovo prevođenje iz vremenskog u frekvencijski domen, tj. signal se razlaže na zbir sinusoidalnih elementarnih talasa (na primer, muzički akord se može razložiti na pojedinačne tonove koja ga sačinjavaju). Ako je signal predstavljen vektorom dobijenim merenjem njegovog intenziteta u diskretnim vremenskim intervalima, primenjuje se *diskretna Furijeova transformacija*. Na slici 3.2 prikazana su tri signala, posebno u vremenskom, a posebno u frekvencijskom domenu. Vidi se da su prva dva pravilne sinusoide (imaju samo jednu dominantnu frekvenciju), dok je treći dobijen sabiranjem dve pravilne sinusoide (ima dve dominantne frekvencije).

Brza Furijeova transformacija je zapravo samo efikasan algoritam za izračunavanje diskretne Furijeove transformacije. Mi ćemo se u ovom udžbeniku ograničiti na jednu njenu primenu, a to je množenje polinoma.

[8] Na primer, udruženje IEEE objavilo je ovu listu: https://www.andrew.cmu.edu/course/15-355/misc/Top%20Ten%20Algorithms.html



Slika 3.2: Signali u vremenskom domenu (levo) u kom su na x-osi prikazani vremenski trenuci, a na y-osi amplituda signala i frekvencijskom domenu (desno) u kom su na x-osi prikazane frekvencije, a na y-osi mera prisustva te frekvencije u signalu.

Problem

Izračunati proizvod dva zadata polinoma $P(x)$ i $Q(x)$.

Formulacija problema koji treba rešiti je precizna samo na prvi pogled, jer nije preciziran način na koji su polinomi predstavljeni. Obično se polinom $P(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0$, stepena $n-1$, predstavlja nizom svojih n koeficijenata $a_0, a_1, a_2, ... a_{n-1}$ uz $1, x, x^2, ..., x^{n-1}$, međutim, to nije jedina mogućnost. Alternativno, polinom stepena $n-1$ moguće je predstaviti njegovim vrednostima u n različitih tačaka: te vrednosti jednoznačno određuju polinom. Na primer, polinom stepena 1, odnosno linearna funkcija, jedinstveno je određena vrednostima u dve različite tačke, a polinom stepena 2, odnosno kvadratna funkcija, vrednostima u tri različite tačke.

Množenje dva polinoma koja su zadata nizom svojih koeficijenata može se izvršiti osnovnim algoritmom složenosti $O(n^2)$ koji se sastoji u množenju svakog monoma prvog polinoma svakim monomom drugog, ili korišćenjem Karacubinog algoritma zasnovanog na pametnoj primeni dekompozicije koji je složenosti $O(n^{\log_2 3})$. Cilj ovog poglavlja je da primenom algoritma FFT izvedemo algoritam složenosti $O(n \log n)$, ali za to je potrebno da detaljnije razmotrimo drugu reprezentaciju.

Predstavljanje polinoma vrednostima na skupu tačaka je interesantno zbog jednostavnosti množenja. Naime, proizvod dva polinoma stepena $n-1$ je polinom stepena $2n-2$, pa je određen svojim vrednostima u $2n-1$ tačaka. Ako pretpostavimo da su vrednosti polinoma-činilaca stepena $n-1$ date u $2n-1$ različitih tačaka, onda se proizvod ova dva polinoma može izračunati pomoću $2n-1$, odnosno $O(n)$ običnih množenja. Naime, vrednost polinoma PQ u bilo kojoj tački x jednaka je proizvodu vrednosti polinoma P u tački x i vrednosti polinoma Q u tački x, tj. važi $(P \cdot Q)(x) = P(x) \cdot Q(x)$.

Nažalost, predstavljanje polinoma njegovim vrednostima na skupu tačaka nije pogodno za neke primene. Primer je izračunavanje vrednosti polinoma u proizvoljnoj tački: pri reprezentaciji polinoma vrednostima na skupu tačaka, ovo je mnogo teže u odnosu na to kada je polinom zadat nizom svojih koeficijenata (potrebno je rešiti problem interpolacije). Sa druge strane, vrednost polinoma $P(x)$ stepena $n-1$ zadatog nizom svojih koeficijenata u proizvoljnoj tački x može se pomoću Hornerove šeme izračunati korišćenjem n množenja tj. u složenosti $O(n)$:

$$P(x) = a_0 + x(a_1 + x(a_2 + ... + x \cdot a_{n-1} ...))$$

Dakle, svaka od reprezentacija je pogodna za jednu, a nije pogodna za drugu operaciju, kako je ilustrovano u tabeli 3.1.

Tabela 3.1: Složenost osnovnih operacija za rad sa polinomima u zavisnosti od reprezentacije polinoma

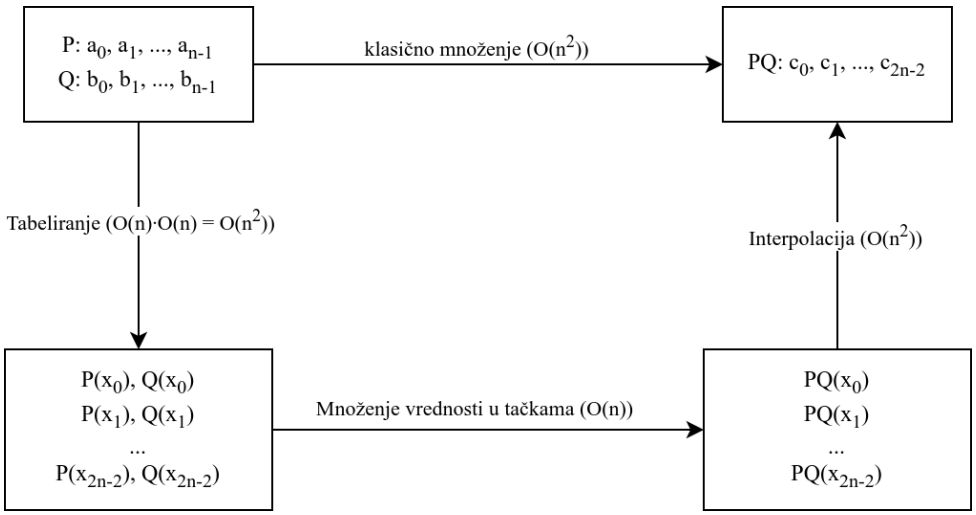
reprezentacija	računanje vrednosti polinoma u tački	množenje polinoma
koeficijenti	$\mathbf{O(n)}$	$O(n^2)$ ili $O(n^{\log_2 3})$
vrednosti	$O(n^2)$	$\mathbf{O(n)}$

Ako bismo mogli da efikasno prevodimo polinome iz jedne u drugu reprezentaciju, dobili bismo mogućnost da obe najznačajnije operacije (množenje i izračunavanje vrednosti) izvršavamo efikasno. Upravo to se postiže primenom brze Furijeove transformacije, tj. algoritma FFT, koji prevođenje iz jedne u drugu reprezentaciju polinoma vrši u složenosti $O(n \log n)$.

Bez primene algoritma FFT prelaz od predstavljanja polinoma koeficijentima na predstavljanje polinoma vrednostima u tačkama (koji ćemo zvati *tabeliranje*), rešava se uzastopnim izračunavanjem vrednosti polinoma u svakoj od tih tačaka. Izračunavanje vrednosti polinoma $P(x)$ stepena $n-1$ u n proizvoljnih tačaka (što je potrebno za jednoznačno predstavljanje polinoma) izvodljivo je primenom Hornerove sheme pomoću $O(n^2)$ množenja. Prelaz od predstavljanja polinoma vrednostima na skupu tačaka na predstavljanje koeficijentima se zove *interpolacija*. Vrednosti koeficijenta polinoma na osnovu vrednosti $(x_j, y_j), 0 \leq j \leq n$ polinoma u $n+1$ različitih tačaka mogu se odrediti korišćenjem Lagranžove interpolacione formule (koja se obično izučava u sklopu numeričke matematike):

$$P(x) = \sum_{j=0}^{n-1} a_j x^j = \sum_{j=0}^{n-1} y_j \cdot \frac{\prod_{k \neq j}(x - x_k)}{\prod_{k \neq j}(x_j - x_k)}$$

Na osnovu ovog izraza koeficijenti polinoma mogu se odrediti algoritmom složenosti $O(n^2)$.



Slika 3.3: Dve mogućnosti za računanje proizvoda dva polinoma $P(x) = \sum_{i=0}^{n-1} a_i x^i$ i $Q(x) = \sum_{i=0}^{n-1} b_i x^i$: direktnim množenjem polinoma što je složenosti $O(n^2)$ (gornja strelica udesno) ili preko reprezentacije polinoma vrednostima na skupu tačaka (strelica nadole, donja strelica udesno, strelica nagore). Ubrzavanjem koraka tabeliranja i interpolacije možemo dobiti efikasniji algoritam.

Postavlja se pitanje kako je moguće ubrzati korake tabeliranja i interpolacije? Ključna ideja da se ne koristi proizvoljnih n tačaka: mi imamo slobodu da po želji izaberemo pogodan skup od n različitih tačaka. Brza Furijeova transformacija koristi specijalan skup tačaka, tako da se i tabeliranje i interpolacija mogu efikasnije izvršavati.

3.7.1 Direktna brza Furijeova transformacija

Osnovna ideja na kojoj počiva brza Furijeova transformacija je izložena u narednom primeru.

Primer 3.7.1

Pretpostavimo da treba izračunati vrednost polinoma $P(x) = x^7+2x^6+3x^5+4x^4+5x^3+6x^2+7x+8$ *u dve različite tačke. Ako su one međusobno suprotne, možemo smanjiti količinu potrebnih izračunavanja na osnovu toga što su polinomi* x^k *neparnog stepena* k *neparne, a parnog stepena* k *parne funkcije. Pretpostavimo da, na primer, treba da izračumo* $P(2)$ *i* $P(-2)$*. Važi*

$$\begin{aligned} P(2) &= 2^7 + 2\cdot 2^6 + 3\cdot 2^5 + 4\cdot 2^4 + 5\cdot 2^3 + 6\cdot 2^2 + 7\cdot 2 + 8 \\ P(-2) &= (-2)^7 + 2\cdot(-2)^6 + 3\cdot(-2)^5 + 4\cdot(-2)^4 + 5\cdot(-2)^3 + 6\cdot(-2)^2 + 7\cdot(-2) + 8 \\ &= -2^7 + 2\cdot 2^6 - 3\cdot 2^5 + 4\cdot 2^4 - 5\cdot 2^3 + 6\cdot 2^2 - 7\cdot 2 + 8. \end{aligned}$$

Ako znamo vrednosti $P_n(2) = 2^7 + 3\cdot 2^5 + 5\cdot 2^3 + 7\cdot 2$ *i* $P_p(2) = 2\cdot 2^6 + 4\cdot 2^4 + 6\cdot 2^2 + 8$*, tada se* $P(2)$ *može izračunati kao* $P(2) = P_p(2) + P_n(2)$*, a* $P(-2) = P_p(2) - P_n(2)$*. Dakle, vrednost polinoma* P *u tačkama* -2 *i* 2 *je izračunata pomoću vrednosti polinoma* P_p *i* P_n *u tački* 2*. Polinomi* P_p *i* P_n *imaju po* 4 *koeficijenta, ali im je stepen veći od broja koeficijenata, no to možemo lako popraviti.*

Važi da je $P_p(2) = 2\cdot(2^2)^3 + 4\cdot(2^2)^2 + 6\cdot 2^2 + 8$*, pa je* $P_p(2)$ *zapravo vrednost polinoma* $P_0(x) = 2x^3 + 4x^2 + 6x + 8$ *u tački* 2^2*, a* $P_n(2) = 2\cdot((2^2)^3 + 3\cdot(2^2)^2 + 5\cdot 2^2 + 7)$ *je zapravo vrednost polinoma* $P_1(x) = x^3 + 3x^2 + 5x + 7$ *u tački* 2^2 *pomnožena sa* 2*. Važi* $P(2) = P_0(2^2) + 2\cdot P_1(2^2)$ *i* $P(-2) = P_0(2^2) - 2\cdot P_1(2^2)$*. Dakle, izračunavanje vrednosti polinoma stepena* 7 *u dve suprotne tačke se svelo na izračunavanje vrednosti dva polinoma stepena* 3 *u jednoj tački (njihovom kvadratu). Ta dva polinoma nastala su od polaznog izdvajanjem koeficijenata na parnim i koeficijenata na neparnim pozicijama.*

Brza Furijeova transformacija je algoritam tipa podeli-pa-vladaj. Kao što je to obično slučaj kod algoritama tog tipa, pretpostavlja se da je dimenzija ulaza stepen dvojke. Ako niz koeficijenata nije dužine $n = 2^k$, on se uvek može dopuniti nulama. Razmotrimo za početak slučaj $n = 8 = 2^3$ tj. pretpostavimo da je potrebno izračunati vrednost polinoma

$$P(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

u nekih 8 različitih tačaka $x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7$.

Pregrupišimo sabirke na sledeći način:

$$\begin{aligned} P(x) &= a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7 \\ &= (a_0 + a_2x^2 + a_4x^4 + a_6x^6) + x(a_1 + a_3x^2 + a_5x^4 + a_7x^6). \end{aligned}$$

Ovim dobijamo dva manja polinoma $P_0(x) = a_0+a_2x+a_4x^2+a_6x^3$ i $P_1(x) = a_1+a_3x+a_5x^2+a_7x^3$. Važi da je $P(x_i) = P_0(x_i^2)+x_iP_1(x_i^2)$. Potrebno je, dakle, da izračunamo sledećih 8 vrednosti polinoma P:

$$\begin{aligned}
P(x_0) &= P_0(x_0^2) + x_0 P_1(x_0^2) &&= (a_0 + a_2x_0^2 + a_4(x_0^2)^2 + a_6(x_0^2)^3) + x_0(a_1 + a_3x_0^2 + a_5(x_0^2)^2 + a_7(x_0^2)^3)\\
P(x_1) &= P_0(x_1^2) + x_1 P_1(x_1^2) &&= (a_0 + a_2x_1^2 + a_4(x_1^2)^2 + a_6(x_1^2)^3) + x_1(a_1 + a_3x_1^2 + a_5(x_1^2)^2 + a_7(x_1^2)^3)\\
P(x_2) &= P_0(x_2^2) + x_2 P_1(x_2^2) &&= (a_0 + a_2x_2^2 + a_4(x_2^2)^2 + a_6(x_2^2)^3) + x_2(a_1 + a_3x_2^2 + a_5(x_2^2)^2 + a_7(x_2^2)^3)\\
P(x_3) &= P_0(x_3^2) + x_3 P_1(x_3^2) &&= (a_0 + a_2x_3^2 + a_4(x_3^2)^2 + a_6(x_3^2)^3) + x_3(a_1 + a_3x_3^2 + a_5(x_3^2)^2 + a_7(x_3^2)^3)\\
P(x_4) &= P_0(x_4^2) + x_4 P_1(x_4^2) &&= (a_0 + a_2x_4^2 + a_4(x_4^2)^2 + a_6(x_4^2)^3) + x_4(a_1 + a_3x_4^2 + a_5(x_4^2)^2 + a_7(x_4^2)^3)\\
P(x_5) &= P_0(x_5^2) + x_5 P_1(x_5^2) &&= (a_0 + a_2x_5^2 + a_4(x_5^2)^2 + a_6(x_5^2)^3) + x_5(a_1 + a_3x_5^2 + a_5(x_5^2)^2 + a_7(x_5^2)^3)\\
P(x_6) &= P_0(x_6^2) + x_6 P_1(x_6^2) &&= (a_0 + a_2x_6^2 + a_4(x_6^2)^2 + a_6(x_6^2)^3) + x_6(a_1 + a_3x_6^2 + a_5(x_6^2)^2 + a_7(x_6^2)^3)\\
P(x_7) &= P_0(x_7^2) + x_7 P_1(x_7^2) &&= (a_0 + a_2x_7^2 + a_4(x_7^2)^2 + a_6(x_7^2)^3) + x_7(a_1 + a_3x_7^2 + a_5(x_7^2)^2 + a_7(x_7^2)^3)
\end{aligned}$$

Da bi se ovo direktno izračunalo, potrebno je izračunati ukupno 16 vrednosti polinoma P_0 i P_1, međutim, pametnim izborom tačaka moguće je taj broj smanjiti na 8. Naime mi imamo slobodu izbora tačaka x_k (dok god ih biramo tako da su različite) i možemo odabrati tačke za koje, na primer, važi $x_0^2 = x_4^2$, $x_1^2 = x_5^2$, $x_2^2 = x_6^2$, $x_3^3 = x_7^2$. Ne smemo da uzmemo iste vrednosti tačaka, ali možemo suprotne. Naime, način da se postigne jednakost kvadrata je da se vrednosti izaberu tako da važi $x_4 = -x_0$, $x_5 = -x_1$, $x_6 = -x_2$ i $x_7 = -x_3$. Tada se izračunavanje svodi na:

$$\begin{aligned}
P(x_0) &= P_0(x_0^2) + x_0 P_1(x_0^2) &&= (a_0 + a_2x_0^2 + a_4(x_0^2)^2 + a_6(x_0^2)^3) + x_0(a_1 + a_3x_0^2 + a_5(x_0^2)^2 + a_7(x_0^2)^3)\\
P(x_1) &= P_0(x_1^2) + x_1 P_1(x_1^2) &&= (a_0 + a_2x_1^2 + a_4(x_1^2)^2 + a_6(x_1^2)^3) + x_1(a_1 + a_3x_1^2 + a_5(x_1^2)^2 + a_7(x_1^2)^3)\\
P(x_2) &= P_0(x_2^2) + x_2 P_1(x_2^2) &&= (a_0 + a_2x_2^2 + a_4(x_2^2)^2 + a_6(x_2^2)^3) + x_2(a_1 + a_3x_2^2 + a_5(x_2^2)^2 + a_7(x_2^2)^3)\\
P(x_3) &= P_0(x_3^2) + x_3 P_1(x_3^2) &&= (a_0 + a_2x_3^2 + a_4(x_3^2)^2 + a_6(x_3^2)^3) + x_3(a_1 + a_3x_3^2 + a_5(x_3^2)^2 + a_7(x_3^2)^3)\\
P(x_4) &= P_0(x_0^2) - x_0 P_1(x_0^2) &&= (a_0 + a_2x_0^2 + a_4(x_0^2)^2 + a_6(x_0^2)^3) - x_0(a_1 + a_3x_0^2 + a_5(x_0^2)^2 + a_7(x_0^2)^3)\\
P(x_5) &= P_0(x_1^2) - x_1 P_1(x_1^2) &&= (a_0 + a_2x_1^2 + a_4(x_1^2)^2 + a_6(x_1^2)^3) - x_1(a_1 + a_3x_1^2 + a_5(x_1^2)^2 + a_7(x_1^2)^3)\\
P(x_6) &= P_0(x_2^2) - x_2 P_1(x_2^2) &&= (a_0 + a_2x_2^2 + a_4(x_2^2)^2 + a_6(x_2^2)^3) - x_2(a_1 + a_3x_2^2 + a_5(x_2^2)^2 + a_7(x_2^2)^3)\\
P(x_7) &= P_0(x_3^2) - x_3 P_1(x_3^2) &&= (a_0 + a_2x_3^2 + a_4(x_3^2)^2 + a_6(x_3^2)^3) - x_3(a_1 + a_3x_3^2 + a_5(x_3^2)^2 + a_7(x_3^2)^3)
\end{aligned}$$

Potrebno je, dakle, rekurzivno izračunati samo sledećih 8 vrednosti polinoma P_0 i P_1 (umesto polaznih 16), a onda ih iskombinovati pomoću prethodnih formula:

$$\begin{aligned}
P_0(x_0^2) &= a_0 + a_2x_0^2 + a_4(x_0^2)^2 + a_6(x_0^2)^3\\
P_0(x_1^2) &= a_0 + a_2x_1^2 + a_4(x_1^2)^2 + a_6(x_1^2)^3\\
P_0(x_2^2) &= a_0 + a_2x_2^2 + a_4(x_2^2)^2 + a_6(x_2^2)^3\\
P_0(x_3^2) &= a_0 + a_2x_3^2 + a_4(x_3^2)^2 + a_6(x_3^2)^3
\end{aligned}
\qquad
\begin{aligned}
P_1(x_0^2) &= a_1 + a_3x_0^2 + a_5(x_0^2)^2 + a_7(x_0^2)^3\\
P_1(x_1^2) &= a_1 + a_3x_1^2 + a_5(x_1^2)^2 + a_7(x_1^2)^3\\
P_1(x_2^2) &= a_1 + a_3x_2^2 + a_5(x_2^2)^2 + a_7(x_2^2)^3\\
P_1(x_3^2) &= a_1 + a_3x_3^2 + a_5(x_3^2)^2 + a_7(x_3^2)^3
\end{aligned}$$

Da li možemo još uštedeti na broju operacija pametnim izborom tačaka x_0, x_1, x_2 i x_3? Razmotrimo izračunavanje vrednosti polinoma P_0 (vrednosti polinoma P_1 se računaju analogno). Grupišimo ponovo koeficijente na parnim i na neparnim pozicijama. Time dobijamo još manje polinome $P_{00}(x) = a_0 + a_4x$ i $P_{01}(x) = a_2 + a_6x$.

$$\begin{aligned}
P_0(x_0^2) &= P_{00}(x_0^4) + x_0^2 P_{01}(x_0^4) &&= (a_0 + a_4x_0^4) + x_0^2(a_2 + a_6x_0^4)\\
P_0(x_1^2) &= P_{00}(x_1^4) + x_1^2 P_{01}(x_1^4) &&= (a_0 + a_4x_1^4) + x_1^2(a_2 + a_6x_1^4)\\
P_0(x_2^2) &= P_{00}(x_2^4) + x_2^2 P_{01}(x_2^4) &&= (a_0 + a_4x_2^4) + x_2^2(a_2 + a_6x_2^4)\\
P_0(x_3^2) &= P_{00}(x_3^4) + x_3^2 P_{01}(x_3^4) &&= (a_0 + a_4x_3^4) + x_3^2(a_2 + a_6x_3^4)
\end{aligned}$$

Za izračunavanje 4 vrednosti polinoma P_0 potrebno je izračunati ukupno 8 vrednosti polinoma P_{00} i P_{01} i želimo to da smanjimo. Međutim, sada imamo problem. Da bismo mogli da smanjimo broj izračunavanja na isti način kao u prvom koraku, potrebno je da odaberemo x_0 i x_2 tako da važi $x_0^4 = x_2^4$. Ne možemo da odaberemo da je $x_0^2 = x_2^2$, jer je tada ili $x_0 = x_2$ ili $x_0 = -x_2 = x_6$, što nije dopušteno, jer sve tačke moraju biti različite. Dakle, mora da važi da je $x_2^2 = -x_0^2$. Međutim to nije moguće ako su x_0 i x_2 realni različiti brojevi, jer su tada i x_0^2 i x_2^2 nenegativni i bar jedan od njih je pozitivan. Da li je onda uopšte moguće izabrati različite vrednosti x_0 i x_2 tako da je $x_2^2 = -x_0^2$? Jeste, ako dopustimo da vrednosti x_i budu *kompleksni brojevi*. Naime, tražimo kompleksan koeficijent k takav da je $x_2 = kx_0$ i $x_2^4 = x_0^4$ tj. $x_2^2 = (kx_0)^2 = k^2x_0^2 = -x_0^2$. Pošto je $k^2 = -1$, za k se može uzeti vrednost imaginarne jedinice i, za koju važi $i^2 = -1$. Zaista, ako je $x_2 = ix_0$, tada je $x_2^2 = (ix_0)^2 = i^2x_0^2 = -x_0^2$ i $x_2^4 = (x_2^2)^2 = (-x_0^2)^2 = x_0^4$. Brojeve x_3 i x_1 možemo izabrati tako da je $x_3 = ix_1$ iz čega sledi

da je $x_3^2 = -x_1^2$ i $x_3^4 = x_1^4$. Nakon toga, izračunavanje potrebnih vrednosti poinoma P_0 se svodi na izračunavanje samo 4 vrednosti polinoma P_{00} i P_{01}:

$$\begin{array}{lll} P_0(x_0^2) & = P_{00}(x_0^4) + x_0^2 P_{01}(x_0^4) & = (a_0 + a_4 x_0^4) + x_0^2(a_2 + a_6 x_0^4) \\ P_0(x_1^2) & = P_{00}(x_1^4) + x_1^2 P_{01}(x_1^4) & = (a_0 + a_4 x_1^4) + x_1^2(a_2 + a_6 x_1^4) \\ P_0(x_2^2) & = P_{00}(x_0^4) - x_0^2 P_{01}(x_0^4) & = (a_0 + a_4 x_0^4) + x_2^2(a_2 + a_6 x_0^4) \\ P_0(x_3^2) & = P_{00}(x_1^4) - x_1^2 P_{01}(x_1^4) & = (a_0 + a_4 x_1^4) + x_3^2(a_2 + a_6 x_1^4) \end{array}$$

Prethodni način izbora tačaka x_2 i x_3 donosi sličnu uštedu i prilikom računanja vrednosti polinoma P_1. Ovim se ceo problem svodi na izračunavanje sledećih 8 vrednosti polinoma P_{00}, P_{01}, P_{10} i P_{11}:

$$\begin{array}{ll|ll} P_{00}(x_0^4) & = a_0 + a_4 x_0^4 & P_{01}(x_0^4) & = a_2 + a_6 x_0^4 \\ P_{00}(x_1^4) & = a_0 + a_4 x_1^4 & P_{01}(x_1^4) & = a_2 + a_6 x_1^4 \\ P_{10}(x_0^4) & = a_1 + a_5 x_0^4 & P_{11}(x_0^4) & = a_3 + a_7 x_0^4 \\ P_{10}(x_1^4) & = a_1 + a_5 x_1^4 & P_{11}(x_1^4) & = a_3 + a_7 x_1^4 \end{array}$$

Posednja ušteda se dobija time što se vrednosti x_0 i x_1 odaberu tako da je $x_1^8 = x_0^8$ tj. $x_1^4 = -x_0^4$. To se može postići ako je $x_1 = kx_0$, gde se koeficijent k određuje tako da je $k^4 = -1$. Pošto je $e^{i\pi} = -1$, jedan takav broj je broj $k = e^{i\frac{\pi}{4}} = \cos(\frac{\pi}{4}) + i\sin(\frac{\pi}{4}) = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$. Podsetimo se Ojlerove formule:

$$e^{i\phi} = \cos\phi + i\sin\phi$$

Tada se prethodno izračunavanje svodi na:

$$\begin{array}{llll} P_{00}(x_0^4) = a_0 + x_0^4 \cdot a_4 & P_{00}(x_1^4) = a_0 - x_0^4 \cdot a_4 & P_{01}(x_0^4) = a_2 + x_0^4 \cdot a_6 & P_{01}(x_1^4) = a_2 - x_0^4 \cdot a_6 \\ P_{10}(x_0^4) = a_1 + x_0^4 \cdot a_5 & P_{10}(x_1^4) = a_1 - x_0^4 \cdot a_5 & P_{11}(x_0^4) = a_3 + x_0^4 \cdot a_7 & P_{11}(x_1^4) = a_3 - x_0^4 \cdot a_7 \end{array}$$

Mogli bismo reći da se vrednosti ovih linearnih polinoma izračunavaju kombinovanjem polinoma stepena 0 (u pitanju su 8 polinoma koji odgovaraju koeficijentima a_k: $P_{000} = a_0$, $P_{001} = a_4$, $P_{010} = a_2$, $P_{011} = a_6$, $P_{100} = a_1$, $P_{101} = a_5$, $P_{110} = a_3$ i $P_{111} = a_7$) i čija se vrednost trivijalno izračunava (bez daljih rekurzivnih poziva). Da bi se sve uklopilo u opštu shemu, možemo reći da se vrednost tih konstantnih 8 polinoma izračunava u tački x_0^8.

Ako rezimiramo sve odnose između promenljivih koje smo do sada uspostavili, vidimo da još jedino možemo slobodno da biramo vrednost x_0, dok se sve ostale vrednosti tačaka izračunavaju na osnovu vrednosti x_0. Imamo niz tačaka

$$\begin{array}{ll} x_0 & x_4 = -x_0 \\ x_1 = (\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2})x_0 & x_5 = -x_1 = (-\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2})x_0 \\ x_2 = ix_0 & x_6 = -x_2 = -ix_0 \\ x_3 = ix_1 = (-\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2})x_0 & x_7 = -x_3 = (\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2})x_0 \end{array}$$

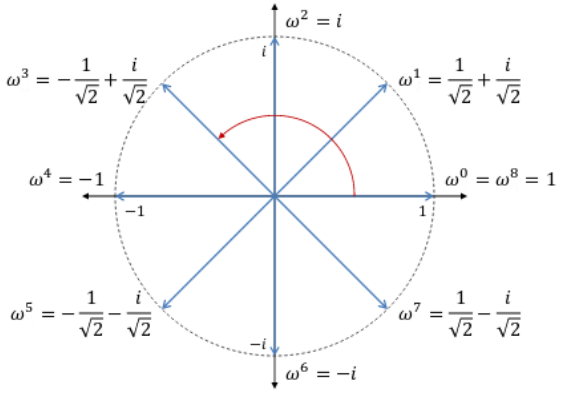
Najjednostavnije formule se dobijaju, naravno, ako se odabere $x_0 = 1$. Tada se može primetiti da su tačke x_i osmi koreni iz jedinice. Nadalje ćemo ove tačke obeležavati sa w_k umesto x_k (da bismo naglasili da su

u pitanju kompleksni brojevi). Dakle, izračunavaćemo vrednosti polinoma redom u tačkama $w_0 = 1 = e^0$, $w_1 = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2} = e^{i\frac{\pi}{4}}$, $w_2 = i = e^{i\frac{\pi}{2}}$, $w_3 = -\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2} = e^{i\frac{3\pi}{4}}$, $w_4 = -1 = e^{i\pi}$, $w_5 = -\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2} = e^{i\frac{5\pi}{4}}$, $w_6 = -i = e^{i\frac{3\pi}{2}}$, $w_7 = \frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2} = e^{i\frac{7\pi}{4}}$, tj. u tačkama

$$w_k = e^{\frac{k\pi i}{4}} = e^{\frac{2k\pi i}{8}}, \quad 0 \leq k < 8.$$

U opštem slučaju ćemo izračunavati vrednosti polinoma stepena n (gde se n obično bira tako da bude stepen broja 2) i izračunavaćemo vrednosti polinoma u tačkama w_k definisanim na sledeći način:

$$w_k = e^{\frac{2k\pi i}{n}}, \quad 0 \leq k < n$$



Slika 3.4: Vizuelizacija osmog primitivnog korena iz jedinice i njegovih stepena u kompleksnoj ravni.

Na slici 3.4 je prikazano da se svi brojevi w_k mogu izraziti kao neki stepen broja $w_1 = e^{\frac{2\pi i}{n}}$, koji ćemo obeležiti sa w. Broj w je *primitivni n-ti koren iz jedinice* (u našem primeru, osmi), što znači da je $w^n = 1$, dok je $w^k \neq 1$ za sve $0 < k < n$. Na primer, imaginarna jedinica i je primitivni četvrti koren iz jedinice jer je $i^1 = i \neq 1$, $i^2 = -1 \neq 1$, $i^3 = -i \neq 1$ i $i^4 = 1$, dok broj -1 jeste četvrti koren iz jedinice (jer je $(-1)^4 = 1$), ali nije primitivan, jer je $(-1)^2 = 1$ (on je primitivni drugi koren iz jedinice). Može se pokazati i da je recipročna vrednost broja w tj. broj $w^{-1} = e^{\frac{-2\pi i}{n}}$ takođe primitivni n-ti koren iz jedinice (što će nam biti važno za inverznu Furijeovu transformaciji o kojoj će biti reči kasnije). Svi brojevi w_k se mogu izraziti kao stepeni broja w (kao i bilo kog drugog primitivnog n-tog korena iz jedinice). Naime, očigledno važi da je

$$w_k = e^{\frac{2k\pi i}{n}} = \left(e^{\frac{2\pi i}{n}}\right)^k = w^k.$$

Još jedna važna činjenica u vezi sa primitivnim n-tim korenima iz jedinice je sledeća. Ako je w primitivni n-ti koren iz jedinice, za neki paran broj $n > 0$, tada je w^2 primitivni $\frac{n}{2}$-ti koren iz jedinice (videćemo da se ovo koristi tokom rekurzivnih poziva u brzoj Furijeovoj transformaciji). Zaista, važi:

$$w^2 = \left(e^{\frac{2\pi i}{n}}\right)^2 = e^{\frac{2\cdot 2\pi i}{n}} = e^{\frac{2\pi i}{\frac{n}{2}}}$$

Naglasimo da u izvođenjima poput prethodnih treba biti jako obazriv. Naime, koristili smo nekoliko puta zakon $(a^b)^c = a^{bc}$ koji ne mora biti tačan za proizvoljne kompleksne (pa čak ni realne brojeve). Na primer, $1 = \sqrt{1} = \sqrt{(-1)^2} = ((-1)^2)^{\frac{1}{2}} \neq (-1)^{2\cdot\frac{1}{2}} = (-1)^1 = -1$. Ipak, može se pokazati da su prethodna izvođenja korektna (pre svega jer je spoljni izložilac stepenovanja uvek bio celobrojan).

Primer 3.7.2

Ilustrujmo sada postupak izračunavanja, tako što ćemo ga primeniti na određivanje vrednosti polinoma $7x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + 1x^1$ *tj. na vektor koeficijenata* $[7, 6, 5, 4, 3, 2, 1, 0]$.

Polinomi stepena 0 *su konstantni i njihova vrednost ne zavisi od* x*, ali da bismo naglasili opšte pravilo, pretpostavićemo da se izračunavaju samo u tački* $(w_0)^8 = (w_8)^0 = w^0 = 1$*:*

$$\begin{array}{llll} P_{000}(w_8^0) & = P_{000}(1) & = a_0 & = 0, \\ P_{001}(w_8^0) & = P_{001}(1) & = a_4 & = 4, \\ P_{010}(w_8^0) & = P_{010}(1) & = a_2 & = 2, \\ P_{011}(w_8^0) & = P_{011}(1) & = a_6 & = 6, \\ P_{100}(w_8^0) & = P_{100}(1) & = a_1 & = 1, \\ P_{101}(w_8^0) & = P_{101}(1) & = a_5 & = 5, \\ P_{110}(w_8^0) & = P_{110}(1) & = a_3 & = 3, \\ P_{111}(w_8^0) & = P_{111}(1) & = a_7 & = 7 \end{array}$$

Polinomi stepena 1 *(to su* P_{00}*,* P_{01}*,* P_{10} *i* P_{11}*) se izračunavaju samo u tačkama* $(w_0)^4 = (w_4)^0 = w^0 = 1$ *i* $(w_1)^4 = (w_4)^1 = w^4 = -1$*:*

$$\begin{array}{llll} P_{00}(w_4^0) & = P_{000}(w_8^0) + w_4^0 \cdot P_{001}(w_8^0) & \text{tj.} & P_{00}(1) & = P_{000}(1) + 1 \cdot P_{001}(1) & = 4, \\ P_{00}(w_4^1) & = P_{000}(w_8^0) - w_4^0 \cdot P_{001}(w_8^0) & \text{tj.} & P_{00}(-1) & = P_{000}(1) - 1 \cdot P_{001}(1) & = -4, \\ P_{01}(w_4^0) & = P_{010}(w_8^0) + w_4^0 \cdot P_{011}(w_8^0) & \text{tj.} & P_{01}(1) & = P_{010}(1) + 1 \cdot P_{011}(1) & = 8, \\ P_{01}(w_4^1) & = P_{010}(w_8^0) - w_4^0 \cdot P_{011}(w_8^0) & \text{tj.} & P_{01}(-1) & = P_{010}(1) - 1 \cdot P_{011}(1) & = -4, \\ P_{10}(w_4^0) & = P_{010}(w_8^0) + w_4^0 \cdot P_{011}(w_8^0) & \text{tj.} & P_{10}(1) & = P_{100}(1) + 1 \cdot P_{101}(1) & = 6, \\ P_{10}(w_4^1) & = P_{010}(w_8^0) - w_4^0 \cdot P_{011}(w_8^0) & \text{tj.} & P_{10}(-1) & = P_{100}(1) - 1 \cdot P_{101}(1) & = -4, \\ P_{11}(w_4^0) & = P_{010}(w_8^0) + w_4^0 \cdot P_{011}(w_8^0) & \text{tj.} & P_{11}(1) & = P_{110}(1) + 1 \cdot P_{111}(1) & = 10, \\ P_{11}(w_4^1) & = P_{010}(w_8^0) - w_4^0 \cdot P_{011}(w_8^0) & \text{tj.} & P_{11}(-1) & = P_{110}(1) - 1 \cdot P_{111}(1) & = -4, \end{array}$$

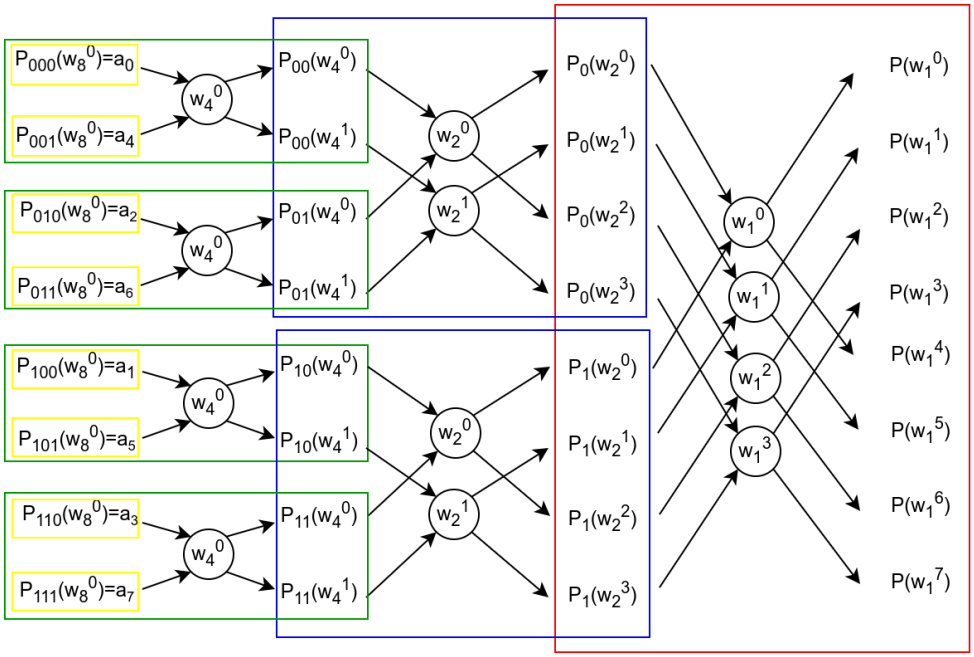
Polinomi stepena 3 *(to su* P_0 *i* P_1*) se izračunavaju samo u tačkama* $(w_0)^2 = (w_2)^0 = w^0 = 1$*,* $(w_1)^2 = (w_2)^1 = w^2 = i$*,* $(w_2)^2 = i^2 = -1$ *i* $(w_3)^2 = (w_2)^3 = w^6 = -i$*:*

$$\begin{array}{llll} P_0(w_2^0) & = P_{00}(w_4^0) + w_2^0 \cdot P_{01}(w_4^0) & \text{tj.} & P_0(1) & = P_{00}(1) + 1 \cdot P_{01}(1) & = 12, \\ P_0(w_2^1) & = P_{00}(w_4^1) + w_2^1 \cdot P_{01}(w_4^1) & \text{tj.} & P_0(i) & = P_{00}(-1) + i \cdot P_{01}(-1) & = -4 - 4i, \\ P_0(w_2^2) & = P_{00}(w_4^0) - w_2^0 \cdot P_{01}(w_4^0) & \text{tj.} & P_0(-1) & = P_{00}(1) - 1 \cdot P_{01}(1) & = -4, \\ P_0(w_2^3) & = P_{00}(w_4^1) - w_2^1 \cdot P_{01}(w_4^1) & \text{tj.} & P_0(-i) & = P_{00}(-1) - i \cdot P_{01}(-1) & = -4 + 4i, \\ P_1(w_2^0) & = P_{10}(w_4^0) + w_2^0 \cdot P_{11}(w_4^0) & \text{tj.} & P_1(1) & = P_{10}(1) + 1 \cdot P_{11}(1) & = 16, \\ P_1(w_2^1) & = P_{10}(w_4^1) + w_2^1 \cdot P_{11}(w_4^1) & \text{tj.} & P_1(i) & = P_{10}(-1) + i \cdot P_{11}(-1) & = -4 - 4i, \\ P_1(w_2^2) & = P_{10}(w_4^0) - w_2^0 \cdot P_{11}(w_4^0) & \text{tj.} & P_1(-1) & = P_{10}(1) - 1 \cdot P_{11}(1) & = -4, \\ P_1(w_2^3) & = P_{10}(w_4^1) - w_2^1 \cdot P_{11}(w_4^1) & \text{tj.} & P_1(-i) & = P_{10}(-1) - i \cdot P_{11}(-1) & = -4 + 4i, \end{array}$$

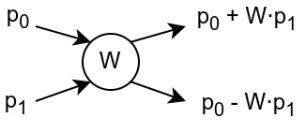
Na kraju, polinom stepena 7 *(to je* P*) se izračunava u svim tačkama* $w_0 = (w_1)^0 = w^0 = 1$*,* $w_1 = (w_1)^1 = w^1 = w = \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}$*,* $w_2 = (w_1)^2 = w^2 = i$*,* $w_3 = (w_1)^3 = w^3 = iw$*,* $w_4 = (w_1)^4 = w^4 = -1$*,* $w_5 = (w_1)^5 = w^5 = -w$*,* $w_6 = (w_1)^6 = w^6 = -i$ *i* $w_7 = (w_1)^7 = w^7 = -iw$*:*

$$
\begin{array}{llllll}
P(w_1^0) & = P_0(w_2^0) + w_1^0 \cdot P_1(w_2^0) & \text{tj.} & P(1) & = P_0(1) + 1 \cdot P_1(1) & = 28, \\
P(w_1^1) & = P_0(w_2^1) + w_1^1 \cdot P_1(w_2^1) & \text{tj.} & P(\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}) & = P_0(i) + w \cdot P_1(i) & = -4 - (4 + 4\sqrt{2})i, \\
P(w_1^2) & = P_0(w_2^2) + w_1^2 \cdot P_1(w_2^2) & \text{tj.} & P(i) & = P_0(-1) + i \cdot P_1(-1) & = -4 - 4i, \\
P(w_1^3) & = P_0(w_2^3) + w_1^3 \cdot P_1(w_2^3) & \text{tj.} & P(-\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}) & = P_0(-i) + w^3 \cdot P_1(-i) & = -4 + (4 - 4\sqrt{2})i, \\
P(w_1^4) & = P_0(w_2^0) - w_1^0 \cdot P_1(w_2^0) & \text{tj.} & P(-1) & = P_0(1) - 1 \cdot P_1(1) & = -4, \\
P(w_1^5) & = P_0(w_2^1) - w_1^1 \cdot P_1(w_2^1) & \text{tj.} & P(-\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}) & = P_0(i) - w \cdot P_1(i) & = -4 - (4 - 4\sqrt{2})i, \\
P(w_1^6) & = P_0(w_2^2) - w_1^2 \cdot P_1(w_2^2) & \text{tj.} & P(-i) & = P_0(-1) - i \cdot P_1(-1) & = -4 + 4i, \\
P(w_1^7) & = P_0(w_2^3) - w_1^3 \cdot P_1(w_2^3) & \text{tj.} & P(\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}) & = P_0(-i) - w^3 \cdot P_1(-i) & = -4 + (4 + 4\sqrt{2})i,
\end{array}
$$

Ovo izračunavanje se shematski prikazuje korišćenjem tzv. *"leptir" sheme* (slika 3.5). Jedan "leptir" opisuje izračunavanje prikazano na slici 3.6.



Slika 3.5: Leptir shema brze Furijeove transformacije

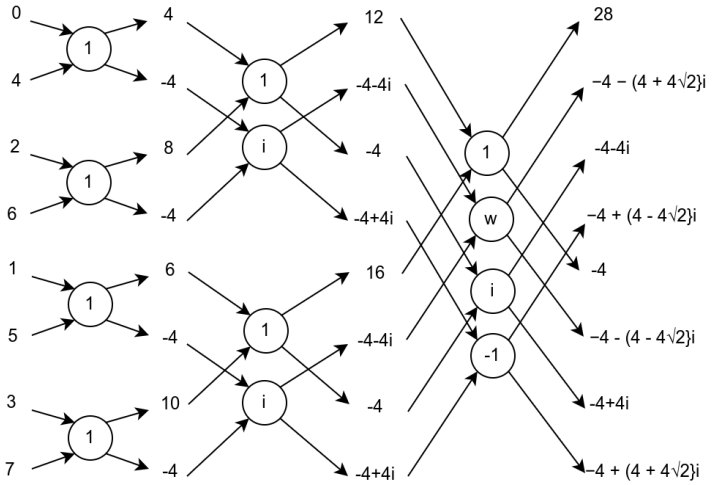


Slika 3.6: Definicija "leptir" izračunavanja

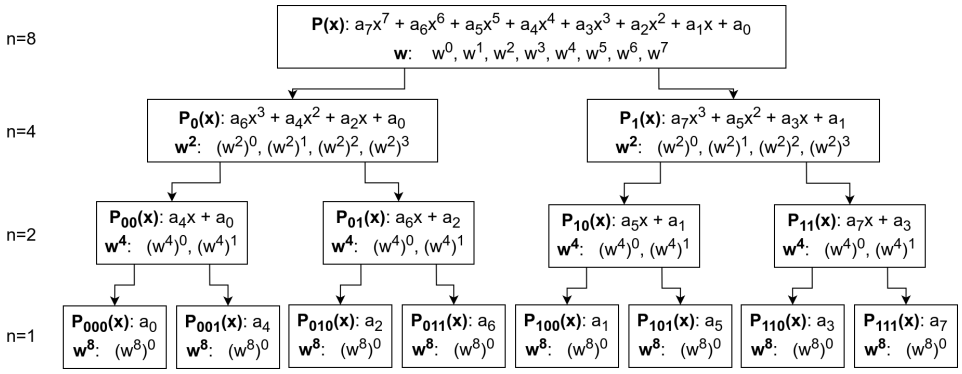
Izračunavanje iz prethodnog primera je prikazano "leptir" shemom na slici 3.7.

Na slici 3.5 se jasno vidi rekurzivna struktura algoritma (svaki rekurzivni poziv je predstavljen jednim obojenim pravougaonikom). Struktura rekurzivnih poziva prikazana je i na slici 3.8.

Svaki rekurzivni poziv prima polinom P_n tj. niz njegovih koeficijenata a_i dužine n i broj W koji je neki stepen polaznog primitivnog korena iz jedinice (a zapravo je primitivni n-ti koren iz jedinice). Rekurzivni poziv izračunava vrednosti polinoma P_n u tačkama $W^0, W^1, \ldots W^{n-1}$. Ako je $n = 1$, izlazi se iz rekurzije i rezultat je jedini koeficijent a_i. U suprotnom se vrše dva rekurzivna poziva (za polinome dobijene od koeficijenata na parnim i neparnim pozicijama) i rezultati se objedinjavaju na osnovu pravila, čija korektnost sledi iz naredne leme.



Slika 3.7: Leptir shema brze Furijeove transformacije za primer polinoma $P(x) = 7x^7 + 6x^6 + 5x^5 + 4x^4 + 3x^3 + 2x^2 + x$



Slika 3.8: Struktura rekurzivnih poziva

Lema 3.7.1

Neka je

$$P_n = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_0 = \sum_{j=0}^{n-1} a_j x^j,$$

polinom stepena n (za neki stepen dvojke $n = 2^k$) i neka je $W = e^{\frac{2\pi i}{n}}$ primitivni n-ti koren iz jedinice (važi $W^n = 1$). Neka je P_{n0} polinom koji se dobija izdvajanjem koeficijenta uz parne stepene tj.

$$P_{n0} = a_{n-2}x^{\frac{n}{2}-1} + ... + a_2 x + a_0 = \sum_{j=0}^{\frac{n}{2}-1} a_{2j} x^j,$$

a P_{n1} polinom koji se dobija izdvajanjem koeficijenata uz neparne stepene tj.

$$P_{n1} = a_{n-1}x^{\frac{n}{2}-1} + ... + a_3 x + a_1 = \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} x^j.$$

Oba ta polinoma su stepena $\frac{n}{2}$.

Za $0 \leq k < \frac{n}{2}$ važi:

$$\begin{aligned} P_n(W^k) &= P_{n0}((W^2)^k) + W^k P_{n1}((W^2)^k) \\ P_n(W^{k+\frac{n}{2}}) &= P_{n0}((W^2)^k) - W^k P_{n1}((W^2)^k). \end{aligned}$$

Dokaz. Zapišimo vrednosti sva tri polinoma iz prve jednakosti u odgovarajućim tačkama:

$$\begin{aligned} P_n(W^k) &= \sum_{j=0}^{n-1} a_j \cdot (W^k)^j \\ P_{n0}((W^2)^k) &= \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot ((W^2)^k)^j = \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot W^{2kj} \\ P_{n1}((W^2)^k) &= \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} \cdot ((W^2)^k)^j = \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} \cdot W^{2kj} \end{aligned}$$

Zato je

$P_{n0}((W^2)^k) + W^k P_{n1}((W^2)^k) = \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot W^{k \cdot 2j} + \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} \cdot W^{k \cdot (2j+1)} = \sum_{j=0}^{n-1} a_k W^{kj} = P_n(W^k)$

Naglasimo da smo u prethodnom izvođenju koristili isključivo celobrojne izložioce (brojeve 2, i i k) i može se lako pokazati da u tom slučaju važe svi uobičajeni zakoni stepenovanja.

Dokažimo sada da je $W^{k+\frac{n}{2}} = -W^k$.

$$W^{k+\frac{n}{2}} = W^k \cdot W^{\frac{n}{2}} = W^k \cdot \left(e^{\frac{2\pi i}{n}}\right)^{\frac{n}{2}} = W^k \cdot e^{\frac{2\pi i}{n}\cdot\frac{n}{2}} = W^k \cdot e^{\pi i} = W^k \cdot (-1) = -W^k$$

Naglasimo da je n paran broj, pa se u prethodnom izvođenju sve vreme radi sa celobrojnim izložiocima.

Zato je

$$\begin{aligned}
P(W^{k+\frac{n}{2}}) = P(-W^k) &= \sum_{j=0}^{n-1} a_j \cdot (-W^k)^j \\
&= \sum_{j=0}^{n-1} a_j \cdot (-1)^j \cdot W^{kj} \\
&= \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot W^{k\cdot 2j} + \sum_{j=0}^{\frac{n}{2}-1} (-1) \cdot a_{2j+1} \cdot W^{k\cdot(2j+1)} \\
&= \sum_{j=0}^{\frac{n}{2}-1} a_{2j} \cdot W^{2kj} - W^k \sum_{j=0}^{\frac{n}{2}-1} a_{2j+1} \cdot W^{2kj} \\
&= P_{n0}((W^2)^k) - W^k P_{n1}((W^2)^k)
\end{aligned}$$

□

3.7.2 Inverzna Furijeova transformacija

Brza Furijeova transformacija rešava samo pola problema: na osnovu koeficijenata polinoma stepena n određuju se njegove vrednosti u n specijalno odabranih tačaka tj. vrši se tabeliranje polinoma. Ostaje pitanje kako da na osnovu vrednosti polinoma u tim tačkama odredimo njegove koeficijente, tj. kako da efikasno izvršimo interpolaciju. Ispostavlja se da je problem interpolacije vrlo sličan problemu tabeliranja i da ga rešava praktično isti algoritam.

Primetimo prvo da se tabeliranje tj. izračunavanje vrednosti polinoma u tačkama $1, w, \dots, w^{n-1}$ može predstaviti i matrično.

$$\begin{pmatrix}
1 & 1 & 1 & \dots & 1 \\
1 & w & w^2 & \dots & w^{n-1} \\
1 & w^2 & w^{2\cdot 2} & \dots & w^{2\cdot(n-1)} \\
\dots & \dots & \dots & \dots & \dots \\
1 & w^{n-1} & w^{(n-1)\cdot 2} & \dots & w^{(n-1)\cdot(n-1)}
\end{pmatrix}
\begin{pmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{pmatrix} =
\begin{pmatrix} P(1) \\ P(w) \\ \dots \\ P(w^{n-1}) \end{pmatrix}$$

Uvedimo sledeće oznake:

- vektor koeficijenata polinoma P obeležimo sa $\mathbf{a} = (a_0, a_1, \dots, a_{n-1})^T$,
- vektor vrednosti polinoma sa $\mathbf{v} = (P(1), P(w), \dots, P(w^{n-1}))^T$,
- Vandermondovu matricu koja sadrži stepene broja w obeležimo sa $V(w)$.

Ako su zadati koeficijenti polinoma $\mathbf{a}$, njegove vrednosti $\mathbf{v}$ u n tačaka $1, w, \dots, w^{n-1}$ dobijaju se izračunavanjem proizvoda:

$$\mathbf{v} = V(w)\mathbf{a}.$$

Ovaj proizvod se naivnim algoritmom množenja matrice i vektora izvršava u vremenu $O(n^2)$, a direktnom brzom Furijeovom transformacijom u vremenu $O(n \log n)$.

S druge strane, ako su zadate vrednosti polinoma $\mathbf{v} = (P(1), P(w), \dots, P(w^{n-1}))^T = (v_0, v_1, \dots, v_{n-1})^T$, a potrebno je izračunati njegove koeficijente $\mathbf{a}$, matrična jednakost postaje sistem linearnih jednačina po nepoznatim koeficijentima $\mathbf{a}$. Rešavanje sistema jednačina svodi se na računanje inverzne matrice $V(w)^{-1}$, što se uvek može uraditi Gausovom eliminacijom, što je algoritam složenosti $O(n^3)$. S druge strane, ako bi se matrica $V(w)^{-1}$ unapred znala, sistem bi se mogao rešiti množenjem jednakosti $\mathbf{v} = V(w)\mathbf{a}$ s leve strane matricom $V(w)^{-1}$:

$$\mathbf{a} = V(w)^{-1}\mathbf{v}.$$

U opštem slučaju, ovo je algoritam složenosti $O(n^2)$. Primetimo, međutim, da Vandermondova matrica ima veoma pravilnu strukturu koja se može iskoristiti i za efikasnije izračunavanje njoj inverzne matrice i za efikasnije množenje vektora njome. Neposredno se proverava da je $V(w)V(w^{-1}) = nI$, gde je sa w^{-1} označena recipročna vrednost broja w tj. broj $e^{\frac{-2\pi i}{n}}$, a sa I je označena jedinična matrica dimenzije n.

Lema 3.7.2

Inverzna matrica matrice $V(w)$ Furijeove transformacije jednaka je

$$V(w)^{-1} = \frac{1}{n}V(w^{-1}).$$

Dokaz. Zaista, ako je $r \neq s$, onda je za $r, s = 0, \dots, n-1$ proizvod $(r+1)$-e vrste matrice $V(w)$ i $(s+1)$-e kolone matrice $V(w^{-1})$ jednak

$$\sum_{k=0}^{n-1} w^{rk}w^{-sk} = \sum_{k=0}^{n-1} w^{(r-s)k} = \frac{1 - w^{n(r-s)}}{1 - w^{r-s}} = 0,$$

jer je $w^n = 1$.

Ako je pak $r = s$, onda je taj proizvod jednak

$$\sum_{k=0}^{n-1} w^{rk}w^{-rk} = \sum_{k=0}^{n-1} 1 = n.$$

Time je teorema dokazana. □

Rešavanje sistema jednačina $\mathbf{v} = V(w)\mathbf{a}$ svodi se, dakle, na izračunavanje proizvoda $\mathbf{a} = \frac{1}{n}V(w^{-1})\mathbf{v}$. Posao se dalje pojednostavljuje zahvaljujući sledećoj teoremi.

Lema 3.7.3

Ako je w primitivni n-ti koren iz jedinice, onda je w^{-1} takođe primitivni n-ti koren iz jedinice.

Pošto brza Furijeova transformacija efikasno računa proizvod neke Vandermondove matrice i vektora, proizvod $\frac{1}{n}V(w^{-1})\mathbf{v}$ može se izračunati primenom brze Furijeove transformacije, zamenjujući u tom algoritmu polaznu vrednost w sa w^{-1}, izračunavanjem proizvoda $V(w^{-1})\mathbf{v}$ i zatim deljenjem komponenti dobijenog vektora sa n. Ova transformacija zove se *inverzna Furijeova transformacija* (engl. inverse Fourier transform).

Napomenimo da se u primenama obrade signala terminologija razlikuje tako što se menja uloga direktne i inverzne transformacije.

Uzimajući sve do sada rečeno u obzir, proizvod polinoma A i B stepena n može se izračunati korišćenjem $O(n \log n)$ operacija (sa kompleksnim brojevima) na sledeći način:

- računamo vrednost polinoma A i B u $2n-1$ tačaka (stepeni $2n$-tog primitivnog korena iz jedinice w) brzom Furijeovom transformacijom,
- u svakoj od tačaka množimo vrednost polinoma A i B,
- računamo brzu Furijeovu transformaciju dobijenog vektora pri čemu umesto vrednosti w koristimo vrednost w^{-1} i rezultat množimo sa $\frac{1}{2n}$.

3.7.3 Algoritam brze Furijeove transformacije

Prikažimo sada algoritam brze Furijeove tranformacije, najpre u pseudokodu (algoritam 9), a zatim i u jeziku C++.

Algoritam 9 Brza Furijeova transformacija

Input: n – broj koeficijenata polinoma P
Input: $a_0, \dots, a_{n-1}$ – koeficijenti polinoma P
Input: w - primitivni n-ti koren iz jedinice
Output: vrednosti $P_0, \dots, P_{n-1}$ polinoma P na skupu tačaka $1, w, \dots, w^{n-1}$
procedure FFT(n, $[a_0, a_1, \dots a_{n-1}]$, w)
    if $n = 1$ **then**
        $P_0 = a_0$
    else
        P^{even} = FFT($n/2$, $[a_0, a_2, \dots, a_{n-2}]$, w^2)
        P^{odd} = FFT($n/2$, $[a_1, a_3, \dots, a_{n-1}]$, w^2)
        for $k = 0$ to $n/2 - 1$ **do**
            $P_k = P_k^{even} + w^k \cdot P_k^{odd}$
            $P_{k+n/2} = P_k^{even} - w^k \cdot P_k^{odd}$

Jasno je da algoritam realizovan prethodnom procedurom zadovoljava jednačinu $T(n) = 2T(n/2)+O(n)$, pa joj je složenost $O(n \log n)$. Nakon transformacije (promene reprezentacije izračunavanjem vrednosti) dva polinoma njihovo množenje je moguće u vremenu $O(n)$, pa je ukupna složenost množenja polinoma pomoću algoritma FFT jednaka $O(n \log n)$.

Inverzna transformacija se može izračunati korišćenjem prethodnog algoritma tako što se u prvom rekurzivnom pozivu umesto w pošalje w^{-1} i tako što se na kraju svaki element rezultata podeli sa n.

Razmotrimo sada C++ implementaciju. U jeziku C++ kompleksne brojeve imamo na raspolaganju u obliku tipova `complex<double>` i `complex<float>` (zapis u dvostrukoj i jednostrukoj tačnosti). Direktan način implementacije je sledeći.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// brza Furijeova transformacija vektora a duzine n=2^k
// bool parametar inverse odredjuje da li je direktna ili inverzna
ComplexVector fft(const ComplexVector& a, bool inverse) {
  // broj koeficijenata polinoma
  int n = a.size();

  // ako je stepen polinoma 0, vrednost u svakoj tacki jednaka
  // je jedinom koeficijentu
  if (n == 1)
    return ComplexVector(1, a[0]);
```

```
  // izdvajamo koeficijente na parnim i na neparnim pozicijama
  ComplexVector Peven(n / 2), Podd(n / 2);
  for (int i = 0; i < n / 2; i++) {
    Peven[i] = a[2 * i];
    Podd[i] = a[2 * i + 1];
  }

  // rekurzivno izracunavamo Furijeove transformacije tih polinoma
  ComplexVector fftEven = fft(Peven, inverse),
                fftOdd  = fft(Podd, inverse);

  // objedinjujemo rezultat
  ComplexVector result(n);
  for (int k = 0; k < n / 2; k++) {
    // odredjujemo primitivni n-ti koren iz jedinice
    double coeff = inverse ? -1.0 : 1.0;
    Complex w = exp((coeff * 2 * k * M_PI / n) * 1i);
    // racunamo vrednost polinoma u toj tacki
    result[k] = fftEven[k] + w * fftOdd[k];
    result[k + n/2] = fftEven[k] - w * fftOdd[k];
  }
  // vracamo konacan rezultat
  return result;
}

// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  return fft(a, false);
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  ComplexVector rez = fft(a, true);
  // nakon izracunavanja vrednosti, potrebno je jos podeliti
  // sve koeficijente duzinom vektora
  int n = a.size();
  for (int k = 0; k < n; k++)
    rez[k] /= n;
  return rez;
}
```

Prethodnu direktnu implementaciju je moguće poboljšati. Najveći problem je to što se n-ti koreni iz jedinice računaju zasebno u svakom rekurzivnom pozivu. Efikasnost bi se značajno popravila ako bi se niz korena izračunao samo jednom, pre funkcije. Problem je u tome što se tokom rekurzije n smanjuje, pa su nam u svakom pozivu potrebni različiti koreni. Međutim, ako je neki broj k-ti koren iz jedinice, onda je on i $2k$-ti koren iz jedinice. Zato, ako znamo niz n-tih korena iz jedinice ($e^{\frac{2k\pi i}{n}}$, za k od 0 do $n-1$), tada su elementi na parnim pozicijama tog vektora $n/2$-ti koreni iz jedinice. Zaista, ako je $k = 2k'$, tada je $e^{\frac{2k\pi i}{n}} = e^{\frac{2k'\pi i}{\frac{n}{2}}}$ i važi da ako je $0 \leq k < n$, tada je $0 \leq k' < n/2$. Dakle, u početku možemo izračunati

niz svih n-tih korena iz jedinice i njih koristiti na početnom nivou rekurzije, na narednom nivou rekurzije ćemo koristiti svaki drugi element tog vektora, na narednom svaki četvrti, zatim svaki osmi i tako dalje. Na primer, ako je $n = 8$, početni niz korena je $1, \frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}, i, -\frac{\sqrt{2}}{2} + i\frac{\sqrt{2}}{2}, -1, -\frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}, -i, \frac{\sqrt{2}}{2} - i\frac{\sqrt{2}}{2}$, na narednom nivou rekurzije je $n = 4$ i koriste se koreni $1, i, -1, -i$, na narednom nivou rekurzije je $n = 2$ i koriste se koreni 1 i -1, a na poslednjem nivou rekurzije je $n = 1$ i tada se izlazi iz rekurzije bez korišćenja ovih korena.

Još jedan problem prethodne implementacije je to što se tokom rekurzije alociraju i popunjavaju pomoćni vektori, što dovodi do gubitka i vremena i memorije. Furijeovu transformaciju je moguće realizovati i bez korišćenja pomoćne memorije. Na početnom nivou rekurzije koeficijenti ulaznog polinoma su svi dati početnim vektorima koeficijenata. Na narednom se posmatraju elementi na pozicijama $0, 2, 4, \ldots, n-2$ i na pozicijama $1, 3, \ldots, n-1$. Na narednom se posmatraju elementi na pozicijama $0, 4, 8, n-4$, zatim elementi na pozicijama $1, 5, \ldots, n-3$, zatim elemeni na pozicijama $2, 6, \ldots, n-2$, i na kraju elementi na pozicijama $3, 7, \ldots, n-1$. Slično se nastavlja i na daljim nivoima rekurzije. Dakle, umesto formiranja pomoćnog ulaznog vektora sa pogodno odabranim ulaznim koeficijentima, prosleđivaćemo originalni vektor, poziciju početka s i pomeraj d i posmatraćemo njegove elemente na pozicijama $s + dk$, za $0 \leq k < n$, gde je $n = n_0/d$, a n_0 je dužina početnog vektora. Rezultate rekurzivnih poziva možemo smestiti u dve polovine rezultujućeg niza. Nakon toga rezultate objedinjujemo. Vrednosti na pozicijama k i $k + n/2$ rezultujućeg vektora određene su vrednostima na poziciji k u rezultatu prvog i drugog rekurzivnog, međutim, one se nalaze upravo na pozicijama k i $k + n/2$ rezultujućeg vektora (jer smo rezultate rekurzivnih poziva smestili u prvu i drugu polovinu rezultata). Moramo voditi računa da te dve vrednosti moramo istovremeno izračunati i ažurirati.

```
typedef complex<double> Complex;
typedef vector<Complex> ComplexVector;

// funkcija vrši Furijeovu transformaciju (direkntu ili inverznu) elemenata
//   a[start_a], a[start_a + step], a[start_a + 2step], ...
// i rezultat smešta u niz
//   result[start_result], result[start_result + 1], result[start_result + 2], ...
// koristeci primitivne korene iz jedinice smestene u niz
//   w[0], w[step], w[2*step], ...
void fft(const ComplexVector& a, int start_a,
         const ComplexVector& w,
         ComplexVector& result, int start_result,
         int step) {
  // broj elemenata niza koji se transformise
  int n = a.size() / step;

  // stepen polinoma je nula, pa mu je vrednost u svakoj tacki jednaka
  // konstantnom koeficijentu
  if (n == 1) {
    result[start_result] = a[start_a];
    return;
  }

  // rekurzivno transformisemo niz koeficijenata na parnim pozicijama
  // smestajuci rezultat u prvu polovinu niza rez
  fft(a, start_a, w, result, start_result, step*2);
  // rekurzivno transformisemo niz koeficijenata na neparnim pozicijama
  // smestajuci rezultat u drugu polovinu niza rez
  fft(a, start_a + step, w, result, start_result + n/2, step*2);
```

```
  // objedinjujemo dve polovine u rezultujuci niz
  for (int k = 0; k < n/2; k++) {
    Complex r1 = result[start_result + k];
    Complex r2 = result[start_result + (k + n/2)];
    result[start_result + k] = r1 + w[k*step] * r2;
    result[start_result + (k + n/2)] = r1 - w[k*step] * r2;
  }
}

// funkcija vrsi direktnu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector fft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n/2);
  for (int k = 0; k < n/2; k++)
    w[k] = exp((2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector result(n);
  // vrsimo transformaciju
  fft(a, 0, w, result, 0, 1);
  // vracamo dobijeni rezultat
  return result;
}

// funkcija vrsi inverznu Furijeovu transformaciju polinoma ciji su
// koeficijenti odredjeni nizom a duzine 2^k
ComplexVector ifft(const ComplexVector& a) {
  // duzina niza koeficijenata polinoma
  int n = a.size();
  // izracunavamo primitivne n-te korene iz jedinice
  ComplexVector w(n);
  for (int k = 0; k < n; k++)
    w[k] = exp((- 2 * k * M_PI / n) * 1i);
  // vektor u koji se smesta rezultat
  ComplexVector result(n);
  // vrsimo transformaciju
  fft(a, 0, w, result, 0, 1);
  // popravljamo rezultat
  for (int i = 0; i < n; i++)
    result[i] /= n;
  // vracamo dobijeni rezultat
  return result;
}

vector<double> product(const vector<double>& p1,
                       const vector<double>& p2) {
  // duzina niza koeficijenata
  int n = p1.size();
```

```
  // kreiramo nizove kompleksnih koeficijenata dopunjavajuci ih nulama
  // do dvostruke duzine
  int N = 2*n;
  ComplexVector a(N, 0.0);
  copy(begin(p1), end(p1), begin(a));
  ComplexVector b(N, 0.0);
  copy(begin(p2), end(p2), begin(b));

  // vrsimo Furijeove transformacije oba vektora koeficijenata
  // (slozenost je O(n log(n)))
  ComplexVector va = fft(a), vb = fft(b);
  // mnozimo vrednosti u pojedinacnim tackama (slozenost je O(n))
  ComplexVector vc(N);
  for (int i = 0; i < N; i++)
    vc[i] = va[i] * vb[i];
  // inverznom Furijeovom transformacijom rekonstruisemo koeficijente proizvoda
  // (slozenost je O(n log(n))
  ComplexVector c = ifft(vc);

  // realne delove kompleksnih brojeva smestamo u niz result i vracamo ga
  vector<double> result(N);
  transform(begin(c), end(c), begin(result),
            [](Complex x){
              return real(x);
            });
  return result;
}
```

Pažljivom analizom je moguće ukloniti rekurziju iz prethodne implementacije (tako se dobija Kuli-Tukijev nerekurzivni algoritam za FFT, zasnovan na ranije prikazanoj leptir-shemi). Ta implementacija neće biti prikazana u ovom udžbeniku.

3.7.4 NTT: Furijeova transformacija u modularnoj aritmetici

Prelaz sa realnih brojeva na kompleksne bio je neophodan zato što kompleksni brojevi garantuju to da za svako n postoji primitivan n-ti koren iz jedinice tj. to da svaka jednačina oblika $w^n = 1$ ima n različitih rešenja, što kod realnih brojeva ne mora biti slučaj (već za $n > 2$). Umesto kompleksnih brojeva moguće je koristiti i neke druge brojevne strukture. Na primer, moguće je koristiti modularnu aritmetiku. Naime, važi sledeća teorema.

Lema 3.7.4

Neka je p neki prost broj. Tada u multiplikativnoj modularnoj grupi $(\mathbb{Z}_p \setminus \{0\}, \times_p)$ primitivni n-ti koren iz jedinice postoji ako i samo ako je $p-1$ deljiv brojem n (primitivni n-ti koren iz jedinice je broj w takav da je $w^n \equiv 1 \pmod p$ i da za bilo koje $m < n$ važi $w^m \not\equiv 1 \pmod p$).

Dakle, u modularnim grupama primitivni n-ti koreni ne postoje za svako n, međutim, mi uvek imamo mogućnost izbora prostog broja p na osnovu n tako da uslov prethodne teoreme bude ispunjen (tj. da je $p-1$ deljivo sa n). Pošto nas zanimaju vrednosti n koji su stepeni dvojke potrebno je da se pronađe neki prost broj p takav da je $p-1$ deljivo sa što više stepena dvojke (onda taj isti broj možemo koristiti za različite vrednosti n koje su stepeni dvojke). Na primer, jedan takav prost broj je $p = 2^{23} \cdot 7 \cdot 17 + 1 = 998\,244\,353$ i on ima primitivni n-ti koren za sve stepene dvojke n do 2^{23} (što je u većini primena i više nego dovoljno, jer nam omogućava množenje polinoma sa po oko 8 miliona koeficijenata koji su u opsegu do skoro milijardu).

Ostaje pitanje kako za datu vrednost n pronaći primitivni n-ti koren iz jedinice po modulu p. Pokažimo jedan efektivni postupak za to. Pretpostavimo da je $p-1=kn$. Za svaki broj $a \in Z_P \setminus \{0\}$, na osnovu male Fermaove teoreme važi $a^{p-1} \equiv 1 \pmod{p}$ tj. $(a^k)^n \equiv 1 \pmod{p}$. Broj a^k je, dakle, uvek n-ti koren iz jedinice, ali ne mora biti primitivan.

Da bismo garantovali da je a^k primitivan n-ti koren, pronađimo element a čiji je red $p-1$ tj. element $a \in Z_P \setminus \{0\}$ takav da je $a^{p-1} \equiv 1 \pmod{p}$ (što uvek važi), ali ni za jedno $m < p-1$ ne važi $a^m \equiv 1 \pmod{p}$. Da bi ovaj uslov važio za neko m potrebno je da m bude delilac broja $p-1$. Dovoljno je, dakle, da proverimo da je $a^m \not\equiv 1 \pmod{p}$ za sve delioce m broja $p-1$. Ovo se može dalje redukovati time što se ispitaju samo oni delioci koji su oblika $\frac{p-1}{f_i}$, gde je f_i neki prost činilac broja $p-1$. Naime, ovi delioci su maksimalni u mreži deljivosti (za svaki od njih, naredni delilac je upravo $p-1$) i ako za neki delilac m ispred njih važi $a^m \equiv 1 \pmod{p}$, tada će bar neki od njih tj. bar neki delilac oblika $\frac{p-1}{f_i}$ biti deljiv sa m i važiće $a^{\frac{p-1}{f_i}} \equiv 1 \pmod{p}$.

Primer 3.7.3

Da bismo za broj $p = 2^{23} \cdot 7 \cdot 17 + 1 = 998\,244\,353$ *odredili broj* a *čiji je red* $p-1$*, dovoljno je da isprobavamo redom vrednosti* a *(na primer, možemo gledati redom male proste brojeve) i da pronađemo prvi broj* a *takav da je* $a^{2^{22} \cdot 7 \cdot 17} \not\equiv 1 \pmod{p}$*,* $a^{2^{23} \cdot 17} \not\equiv 1 \pmod{p}$ *i* $a^{2^{23} \cdot 7} \not\equiv 1 \pmod{p}$*. Prvi takav broj je* $a = 3$*.*

Sada primitivni n-ti koren možemo dobiti kao stepen a^k. Zaista, važi da je $(a^k)^n = a^{p-1}$ što je kongruentno sa 1 po modulu p. Ako bi za neko $m < n$ važilo $(a^k)^m \equiv 1 \pmod{p}$, tada bi važilo $a^{km} \equiv 1 \pmod{p}$ i red elementa a bio manji ili jednak $km < kn = p-1$, što je u suprotnosti sa izborom elementa a (koji je izabran kao element reda $p-1$).

Kada je poznat primitivni n-ti koren iz jedinice w, izvođenje Furijeove transformacije teče na potpuno isti način, jedino što se umesto kompleksnih brojeva koriste prirodni brojevi i sve operacije se izvršavaju po modulu p. Ovaj oblik Furijeove transformacije se nekada naziva NTT (engl. number theoretic transform). Ona može da ima prednosti u odnosu na klasičan pristup sa kompleksnim brojevima, jer ne postoje problemi sa zapisom broja niti računske greške.

Primer 3.7.4

Prikažimo izračunavanje proizvoda polinoma $P(x) = 1 + x + x^2$ *i* $Q(x) = 3 + 5x$ *primenom NTT za vrednost* $p = 998\,244\,353$ *(naravno, pošto su polinomi malog stepena, sa malim koeficijentima, mogli bismo koristiti i neku manju vrednost za* p*). Koeficijente polinoma dopunjavamo tako da se dobiju vektori čija je dužina stepen dvojke i dovoljna je da se smesti proizvod koji je stepena 3, pa dobijama dva vektora dužine* $n = 4$*: [1, 1, 1, 0] i [3, 5, 0, 0]. Vrednost* k *jednaka je* $(p-1)/n = 249\,561\,088$*, pa je primitivni* n *koren iz jedinice jednak* $w = a^k \bmod p = 911\,660\,635$*.*

Furijeova transformacija može dobiti množenjem matricom (naravno, može i brže, korišćenjem algoritma FFT):

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 \\ 1 & w^2 & w^4 & w^6 \\ 1 & w^3 & w^6 & w^9 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 911\,660\,635 & 998\,244\,352 & 86\,583\,718 \\ 1 & 998\,244\,352 & 1 & 998\,244\,352 \\ 1 & 86\,583\,718 & 998\,244\,352 & 911\,660\,635 \end{pmatrix}$$

Množenjem ove matrice vektorima polinoma (po modulu p*), dobijaju se njihove Furijeove transformacije:* $[3, 911\,660\,635, 1, 86\,583\,718]$ *i* $[8, 565\,325\,766, 998\,244\,351, 432\,918\,593]$*. Njih množimo element po element (po modulu* p*) i dobijamo vektor:* $[24, 738\,493\,194, 998\,244\,351, 259\,751\,149]$*.*

Modularni inverz broja w *je* $86\,583\,718$*. Matrica inverzne Furijeove transformacije je zato:*

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 86\,583\,718 & 998\,244\,352 & 911\,660\,635 \\ 1 & 998\,244\,352 & 1 & 998\,244\,352 \\ 1 & 911\,660\,635 & 998\,244\,352 & 86\,583\,718 \end{pmatrix}$$

Množenjem ove matrice vektorom $[24, 738\,493\,194, 998\,244\,351, 259\,751\,149]$ *(po modulu* p*) i skraćivanjem dobijenog rezultata sa* $n = 4$ *(što je dužina vektora), dobija se vektor [3, 8, 8, 5] koji predstavlja polinom* $3 + 8x + 8x^2 + 5x^3$*, što je zaista proizvod naša dva polazna polinoma.*

Zadatak: Poklapanje tačaka

Niske s i t se sastoje od slova i tačaka. Niska t je kraća od niske s i može se poravnati sa niskom s počevši od bilo kog karaktera niske s. Napiši program koji za svako moguće poravnavanje niske s i t određuje broj tačaka koji im se poklapaju.

Opis ulaza

Sa standardnog ulaza se učitava niska s (dužine najviše 10^5 karaktera) i niska t dužine najviše 10^4 karaktera).

Opis izlaza

Na standardni izlaz ispisati broj poklopljenih tačaka za svako poravnavanje niske s i t.

Primer

Ulaz

```
ab.c..ab..a.a.baa.a.c..a
b.ac..b.
```

Izlaz

```
2 3 1 2 4 1 2 2 2 1 2 0 3 1 2 2 2
```

Objašnjenje

U prvom poravnavanju poklapaju se dve tačke.

```
ab.c..ab..a.a.baa.a.c..a
b.ac..b.
```

U drugom poravnavanju poklapaju se tri tačke.

```
ab.c..ab..a.a.baa.a.c..a
 b.ac..b.
```

U trećem poravnavanju poklapa se jedna tačka.

```
ab.c..ab..a.a.baa.a.c..a
  b.ac..b.
```

Slično se broje tačke i kod ostalih poravnavanja.

Rešenje

Gruba sila

Rešenje grubom silom podrazumeva da se za svako moguće poravnavanje eksplicitno prebroje tačke koje se poklapaju.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s, t;
  cin >> s >> t;
  for (int i = 0; i + t.length() <= s.length(); i++) {
    int k = 0;
    for (int j = 0; j < t.length(); j++)
      if (s[i+j] == '.' && t[j] == '.')
        k++;
    cout << k << " ";
  }
  cout << endl;
}
```

Množenje polinoma

Kreirajmo binarne nizove S i T istih dužina kao niske s i t, takve da je $S_k = 1$ akko je `s[k] == '.'` i da je $T_k = 1$ akko je `t[k] == '.'`. Razmotrimo poravnanje koje počinje na poziciji i. Tada se broj tačaka koje se poklapaju može izračunati izrazom:

$$\sum_{j=0}^{|t|-1} S_{i+j} \cdot T_j.$$

Ovaj zbir je potrebno izračunati za svaku poziciju $0 \leq i \leq |s| - |t|$. Operacija u kojoj se kraći niz pomera duž dužeg i računaju se zbirovi proizvoda elemenata na odgovarajućim pozicijama nazivaju se *konvolucija* i ona se jednostavno svodi na problem množenja polinoma.

Razmotrimo množenje polinoma $P(x) = \sum_{i=0}^{m} a_i x^i$ i $Q(x) = \sum_{j=0}^{n} b_j x^j$. Njihov proizvod je polinom:

$$P(x)Q(x) = \sum_{i=0}^{m} \sum_{j=0}^{n} a_i b_j x^{i+j}.$$

Grupšući koeficijente uz isti stepen broja x dobijamo:

$$P(x)Q(x) = \sum_{k=0}^{m+n} \left(\sum_{j=0}^{k} a_{k-j} b_j \right) x^k,$$

pri čemu podrazumevamo da je $a_i = 0$ za $i \geq m$ i $b_j = 0$ za $j \geq n$. Fokusirajmo se na koeficijente uz stepene x^k za k između $m-1$ i $n-1$. Oni su jednaki:

$$\sum_{j=0}^{m-1} a_{k-j} b_j,$$

što liči na sume koje želimo da izračunamo konvolucijom, jedino što se elementi množe u obratnom redosledu (prvi element niza b množi sa poslednjim elementom odgovarajućeg dela niza a, drugi sa pretposlednjim itd.). Jedna operacija se lako svodi na drugu obrtanjem redosleda koeficijenata jednog od polinoma (stoga u literaturi ne postoji konsenzus koji se tačno oblik ovih operacija naziva konvolucija). Uzmimo da je $P(x) = \sum_{i=0}^{|s|-1} S_i x^i$, a da je $Q(x) = \sum_{j=0}^{|t|-1} T_{|t|-1-j} x^j$. Tada su koeficijenti uz x^k za $|t| - 1 \leq k \leq |s| - 1$ jednaki:

$$\sum_{j=0}^{|t|-1} S_{k-j} T_{|t|-1-j} = \sum_{j=0}^{|t|-1} S_{k-|t|+1+j} T_j$$

Primetimo da su ovo upravo zbirovi koji su nama potrebni: zbir za dati pomeraj i je koeficijent uz x^k, za $k = |t| - 1 + i$. Dakle, potrebno je samo formirati polinome P i Q na opisani način, pomnožiti ih (što možemo efikasno uraditi korišćenjem brze Furijeove transformacije) i pročitati koeficijente uz odgovarajuće stepene.

```
string s, t;
cin >> s >> t;

// najmanji stepen dvojke veći od dužine prve niske
int n = pow2(s.size());

// koeficijenti prvog polinoma
ComplexVector S(n, 0);
for (int i = 0; i < s.size(); i++)
  if (s[i] == '.')
    S[i] = 1;

// koeficijenti drugog polinoma (smešteni u obratnom redosledu)
ComplexVector T(n, 0);
for (int i = 0; i < t.size(); i++)
  if (t[t.size() - i - 1] == '.')
    T[i] = 1;

// proizvod polinoma (korišćenjem fft i ifft)
ComplexVector st = mult(ss, tt);

// traženi brojevi su koeficijenti proizvoda
for (int i = 0; i <= s.length() - t.length(); i++)
  cout << round(st[i + t.length() - 1].real()) << " ";
```

Zadatak: Digitalni brojač

$2n$-to cifreni digitalni brojač koji odbrojava od $000 \ldots 000$ do $999 \ldots 999$ emituje zvučni signal svaki put kada je suma prvih n cifara jednaka sumi poslednjih n cifara. Na primer, za šestocifreni digitalni brojač zvučni signal se pušta za $000000, 001001, 001010, \ldots, 999999$. Napisati program koji će odrediti koliko puta će biti emitovan zvučni signal.

Opis ulaza

U prvoj liniji standardnog ulaza nalazi se prirodan broj n ($1 \leq n \leq 9$).

Opis izlaza

Na standarnom izlazu prikazati koliko postoji $2n$-cifrenih brojeva sa traženim svojstvom.

Primer

Ulaz

```
3
```

Izlaz

```
55252
```

Rešenje

Funkcija generatrisa

Prva i druga polovina broja se mogu birati nezavisno. Ako postoji b_k načina da se odabere polovina broja tako da je zbir cifara u toj polovini jednak k, onda postoji b_k^2 načina da se odabere ceo broj tako da je zbir obe polovine međusobno jednak i jednak k. Konačno rešenje se onda dobija sabiranjem ovih vrednosti za sve moguće zbirove k (a to su zbirovi od 0 do $9n$).

Razmotrimo polinom $P(x) = 1 + x + x^2 + ... + x^9$. Svaki njegov koeficijent je 1, što je ujedno i broj načina da jednocifreni broj ima zbir cifara redom 0, 1, pa do 9. Razmotrimo polinom

$$\begin{aligned} P(x)^2 &= (1 + x + x^2 + ... + x^9)(1 + x + x^2 + ... + x^9) \\ &= 1 + 2x + 3x^2 + x^3 + ... + 9x^8 + 10x^9 + 9x^{10} + ... 2x^{17} + 1x^{18}. \end{aligned}$$

Koeficijent uz x^k određuje da se broj k dobije kao zbir cifara nekog dvocifrenog broja. Na sličan način koeficijent uz x^k, polinoma $P(x)^m$ određuje broj načina da se broj k dobije kao zbir cifara nekog m-tocifrenog broja.

Zaista, polinom $P(x)^m$ dobija se sabiranjem proizvoda svih mogućih m-torki monoma iz $P(x)$. Postoji bijekcija između m-tocifrenih brojeva i tih m-torki monoma iz $P(x)$, takva da se svakoj cifri c dodeljuje monom x^c. Proizvod monoma takve m-torke je monom x^k, gde je k zbir cifara m-tocifrenog broja koji odgovara toj m-torki. Zato je broj m-tocifrenih brojeva čiji je zbir cifara k (broj b_{mk}) jednak broju m-torki monoma čiji je proizvod x^k. Zbir svih tih monoma je $b_{mk}x^k$, tako da se broj b_{mk} zaista može odrediti kao koeficijent uz x^k polinoma $P(x)^m$.

Za učitani paran broj n možemo odrediti koeficijente polinoma $P(x)^{\frac{n}{2}}$ i zatim rezultat možemo dobiti kao zbir kvadrata svih dobijenih koeficijenata. Stepen se može odrediti kombinacijom brzog stepenovanja (svako množenje polinoma se može vršiti pomoću brze Furijeove transformacije). Takođe, moguće je samo pronaći brzu Furijeovu transformaciju polinoma P, zatim vrednost u svakoj tački zasebno stepenovati (brzim stepenovanjem), pa rezultat rekonstruisati inverznom Furijeovom transformacijom.

Primećujemo da smo prebrojavanje kombinatornih objekata ovaj put sveli na operacije nad polinomima (drugi uobičajeni način prebrojavanja je zasnovan na dinamičkom progamiranju). Polinomi ili redovi čiji koeficijenti daju broj kombinatornih objekata nazivaju se *funkcije generatrise* i predstavljaju veoma značajnu tehniku u kombinatorici.

Moguća je varijanta zadatka u kom ne bi bilo dozvoljeno korišćenje svih cifara, već samo nekih od njih. U tom slučaju bismo polinom $P(x)$ definisali tako da sadži samo one stepene promenljive x koji odgovaraju dopuštenim ciframa (na primer, ako du dopuštene samo parne cifre, koristili bismo polinom $P(x) = 1 + x^2 + x^4 + x^6 + x^8$.

Svako množenje polinoma primenom FFT je složenosti $O(n \log n)$. Ako se izvodi brzo stepenovanje na stepen $n/2$, vrši se $O(\log n)$ množenja polinoma, pa je ukupna složenost $O(n \log^2 n)$. Ako bi se izvršila

samo brza Furijeova transformacija polinoma $P(x)$, zatim stepenovale njegove vrednosti u tačkama, pa rezultat dobio inverznom brzom Furijeovom transformacijom, složenost bi bila $O(n \log n)$.

```
// brzim stepenovanjem odredjuje se stepen polinoma P^k
ComplexVector stepen(const ComplexVector& a, int k) {
  if (k == 0)
    // Polinom P(x) = 1 predstavljen vektorom date duzine
    return one(a.size());
  if (k % 2 == 0)
    return stepen(proizvod(a, a), k/2);
  else
    return proizvod(a, stepen(a, k-1));
}


// za svaki broj iz [0, maxZbir] odredjujemo broj nacina da se taj
// broj predstavi kao zbir brojCifara cifara (pri cemu se u obzir
// uzima i redosled cifara)
vector<long long> brojParticija(int brojCifara, int maksZbir) {
  // najmanji stepen dvojke veci ili jednak od maksZbir+1
  int n = pow2(maksZbir + 1);
  // polinom P(x) = 1 + x + ... + x^9 sa vektorom koeficijenata duzine n
  ComplexVector P(n, 0);
  for (int i = 0; i < 10; i++)
    P[i] = 1;
  // stepen P(x)^brojCifara
  ComplexVector Pbc = power(P, brojCifara);
  // konvertujemo kompleksne brojeve (koeficijente polinoma) u cele
  vector<long long> rezultat(n);
  for (int i = 0; i < n; i++)
    rezultat[i] = round(Pn2[i].real());
  return rezultat;
}

// izracunava broj 2n-tocifrenih brojeva kod kojih je zbir prvih n i
// zbir drugih n cifara jednak
long long brojBrojeva(int n) {
  // za svaki broj od 0 do 9*n nas zanima koliko postoji razlicitih
  // n-tocifrenih brojeva ciji je zbir cifara taj broj
  vector<long long> bp = brojParticija(n, 9*n);

  // ukupan broj brojeva cija leva i desna polovina imaju isti broj cifara
  long long ukupnoBrojeva = 0;
  // za svaki moguci zbir cifara polovine broja
  for (int zbirCifara = 0; zbirCifara <= 9*n; zbirCifara++) {
    // postoji bp[zbirCifara] nacina da napravimo levu polovinu i
    // bp[zbirCifara] nacina da napravimo desnu polovinu broja
    // tj. bp[zbirCifara]*bp[zbirCifara] nacina da odaberemo broj kome
    // i leva i desna polovina imaju zbir cifara zbirCifara
    ukupnoBrojeva += bp[zbirCifara] * bp[zbirCifara];
  }
  return ukupnoBrojeva;
```

```
}
```

4. Algoritmi za analizu i obradu teksta

Računari se koriste za obradu velikih količina tekstualnih podataka i da bi to bilo moguće uraditi na efikasan način razvijen je velikih broj algoritama koji se bave analizom i obradom teksta. Najčešći problemi koji se rešavaju su pretraga teksta (provera da li jedna niska sadrži drugu), pronalaženje najdužih zajedničkih podniski dve niske, pronalaženje najduže podniske koja se ponavlja unutar niske, pronalaženje najkraće podniske koji se javlja samo jednom unutar niske, pronalaženje najdužeg palindroma unutar niske i slično. Svi ovi problemi imaju velike praktične primene. Na primer, ispitivanje da li se niska javlja u tekstu se koristi u pretrazi dokumenata (veb-strana, PDF dokumenata, datoteka itd.), proveri pravopisa, detekciji plagijata, filtriranju neželjene pošte i slično. Za sve ove probleme nije teško razviti algoritme složenosti $O(n^2)$, međutim, pokazuje se da je moguće doći i do efikasnijih algoritama, složenosti $O(n)$.

U nastavku će biti izloženi neki klasični algoritmi i strukture podataka koje se koriste za analizu i obradu teksta: algoritmi zasnovani na heširanju niski, algoritam KMP, z-niz i z-algoritam za njegovu konstrukciju i Manačerov algoritam za pronalaženje najduže palindromske podniske. Veliki broj operacija nad tekstom se efikasno sprovodi korišćenjem sufiksnih drveta i sufiksnih nizova, pa se čitaocu zainteresovanom za temu obrade teksta savetuje njihovo samostalno proučavanje.

4.1 Heširanje niski

Heširanje niski (engl. string hashing) podrazumeva da se svakoj niski dodeli ceo broj iz određenog opsega. Iako ne može da se garantuje da će svakoj niski biti dodeljen različiti broj, verovatnoća da su dve niske predstavljene istim brojem treba da bude jako mala. Na taj način se problem poređenja dugačkih niski može svesti na poređenje brojeva koji ih reprezentuju, što je mnogo efikasnija operacija. Ako brojevi kojim su predstavljene niske nisu jednaki, onda niske sigurno nisu jednake. Ako su brojevi jednaki, polazne niske su gotovo izvesno jednake (ako želimo da budemo baš apsolutno sigurni, možemo ih eksplicitno uporediti). Na primer, ako želimo da uporedimo da li su dve velike datoteke (na primer, knjige u PDF formatu ili video-zapisi isti), možemo uporediti samo njihove heš-vrednosti.

Tehnika heširanja niski je dobila na popularnosti kada su Rabin i Karp osmislili efikasan algoritam za traženje jedne niske u drugoj (proveru da li se jedna niska javlja kao podniska tj. segment uzastopnih karaktera druge) zasnovan na heširanju. Algoritmi heširanja mogu biti korisni i u rešavanju nekih drugih problema nad tekstom, kao što su određivanje broja različitih podniski date niske, pronalaženje najduže niske koja se

javlja bar dva puta kao podniska date niske, kompresija niske, određivanje broja palindromskih podniski u datoj niski i dr.

Heširanje ima primenu u raznim domenima.

U kriptografiji se definišu *kriptografske heš-funkcije* (na primer, SHA, MD5) koje pored uobičajenih svojstava heš-funkcija imaju i dodatna sigurnosna svojstva (na primer, iako se zna algoritam heširanja, složenost pronalaženja niske koja bi bila predstavljena datom heš-vrednošću je ogromna i ovaj zadatak je praktično neizvodiv). Jedna od uobičajenih primena kriptografskih heš-funkcija je u sistemima za proveru lozinki. Naime, iz bezbedonosnih razloga, lozinke se obično čuvaju u heširanom obliku (jer u slučaju da neko uspe da dobije neovlašćeni pristup sistemu, on neće moći da sazna stvarne lozinke, već samo njihove heš-vrednosti na osnovu kojih je gotovo nemoguće rekonstruisati originalne lozinke). Kada korisnik unese lozinku, računa se njena heš-vrednost i upoređuje sa onom sačuvanom. Ako se te dve heš-vrednosti ne poklope, tada je sigurno uneta pogrešna lozinka. Ako se poklope, postoji izvesna verovatnoća da je uneta lozinka pogrešna (jer heš-funkcije nisu injektivne), međutim, heš-funkcije se prave tako da ta verovatnoća bude jako mala (smatra se da je veća verovatnoća da hardver pogrešno izračuna neku heš-vrednost nego da neka nasumično odabrana lozinka ima istu heš-vrednost kao prava lozinka).

Heširanje se koristi i za obezbeđivanje integriteta poruke koja se šalje putem kanala za komunikaciju. Provera da li je poslata poruka neizmenjena se vrši poređenjem heš vrednosti poruke pre i posle slanja: ako su ove dve vrednosti jednake, smatramo da je prenos uspešno završen. Često se prilikom preuzimanja dokumenata sa interneta (na primer, instalacionih datoteka) pored dokumenata na sajtu navode i njihove heš-vrednosti da bi korisnik bio siguran da je preuzeo dokumente koji nisu u međuvremenu izmenjeni.

Heš-vrednosti se koriste i prilikom digitalnog potpisa. Umesto da se čitav dokument potpisuje tajnim ključem (što može biti veoma neefikasno), izračunava se njegova heš-vrednost koja se potpisuje. Heš-vrednost garantuje i da nije došlo do izmena dokumenta nakon njegovog potpisivanja.

Heširanje ima primenu i u prevodiocima programskih jezika. Naime, većina programskih jezika ima veliki broj rezervisanih reči, poput ključnih reči `if`, `for` i `while`, koje je potrebno obraditi na drugačiji način od ostalih identifikatora. Kako bi utvrdio da li je pročitana reč iz izvornog koda rezervisana, prevodilac čuva heširane vrednosti rezervisanih reči tog programskog jezika.

4.1.1 Heš-funkcije i njihova svojstva

Prilikom heširanja, niski s se korišćenjem neke *heš-funkcije* (engl. hash function) h pridružuje *heš-vrednost* (engl. hash value) $h(s)$ koja obično pripada nekom intervalu prirodnih brojeva $[0, m)$. Heš-funkcija zadovoljava naredni uslov: ako su dve niske s i t jednake, i njihove heš-vrednosti $h(s)$ i $h(t)$ su jednake. Naime, heš-funkcija je uvek *deterministička* – za jedan ulaz uvek daje jedan isti izlaz. Sa druge strane, heš-funkcija ne mora biti injektivna: ako je $h(s) = h(t)$, ne mora da važi $s = t$. Situacija kada za dve različite niske s i t važi $h(s) = h(t)$ naziva se *kolizija* (engl. collision).

Pošto je broj različitih niski koje je moguće heširati obično jako veliki, heš-funkcije gotovo nikada nisu injektivne i kolizije je nemoguće izbeći. Na primer, ako bismo hteli da vrednosti heš-funkcije budu jedinstvene za svaku nisku dužine do 14 karaktera koja se sastoji samo od malih slova engleske abecede, potrebno bi nam bilo $1+26^1+\ldots+26^{14} = \frac{26^{15}-1}{26-1}$ različitih heš-vrednosti, što je oko 26^{14}. Pošto je $26^{14} > 2^{64}$, te heš-vrednosti ne bi stale u opseg 64-bitnih celih brojeva, što je najširi primitivni celobrojni tip podataka. U praksi obično razmatramo i mnogo duže niske i širi skup karaktera i jasno je da čak ni sa povećanjem broja bitova (danas se često koriste heš-vrednosti predstavljene pomoću 256 bitova) injektivnost nije moguće postići.

Poželjno je da heš-funkcija bude surjektivna na svom kodomenu $[0, m)$ kao i da verovatnoća da heš-vrednosti dve različite niske budu jednake, odnosno da dođe do kolizije, bude što je moguće manja. U velikom broju slučajeva ta je verovatnoća toliko mala, da se mogućnost dolaska do kolizije ignoriše, čime se teorijski narušava korektnost algoritma. Moguće je, pak, da se u slučajevima kada se dobiju jednake heš-vrednosti dve niske, proveri jednakost ove dve niske karakter po karakter. Time se garantuje tačnost,

ali se potencijalno žrtvuje efikasnost algoritma. Da li će se ta provera vršiti zavisi od toga da li je primena takva da je u redu tolerisati jako malu verovatnoću netačnog odgovora.

Prilikom heširanja dve fiksirane niske obično ne dolazi do kolizije (tj. ne dešava se da dve različite niske imaju istu heš-vrednost). Na primer, za odabir vrednosti parametra $m = 10^9$ (što odgovara heš-vrednostima od tridesetak bitova), pod pretpostavkom da su sve heš-vrednosti jednako verovatne, verovatnoća da dođe do kolizije je samo $1/m = 10^{-9}$. Međutim, ukoliko jednu fiksiranu nisku s poredimo sa 10^6 drugih različitih niski, verovatnoća da dođe do bar jedne kolizije iznosi oko $10^6 \cdot 10^{-9} = 10^{-3}$, dok ako poredimo 10^6 niski međusobno verovatnoća da dođe do bar jedne kolizije je skoro jednaka 1.[1]

Postoji jednostavni "trik" kojim se može smanjiti verovatnoća da do kolizije dođe – računanjem vrednosti dve različite heš-funkcije (štaviše, može se iskoristiti i ista heš-funkcija za različite vrednosti svojih parametara). Ako je vrednost parametra m oko 10^9 za obe heš-funkcije, onda je ovo uporedivo sa korišćenjem jedne heš-funkcije za vrednost parametra m koja je približno jednaka 10^{18}. Sada, ako poredimo 10^6 niski međusobno, verovatnoća da dođe do kolizije smanjiće se na $(10^6 \cdot 10^6) \cdot 10^{-18} = 10^{-6}$.

Heširanje se često koristi da bi se algoritmi grube sile učinili efikasnijim. Razmotrimo problem upoređivanja dve niske s i t. Algoritam grube sile poredi date niske s i t karakter po karakter i složenosti je $O(\min\{|s|, |t|\})$, gde su $|s|$ i $|t|$ dužine niski s i t redom. Postavlja se pitanje da li postoji efikasniji algoritam za poređenje niski s i t. Svaku od niski s i t možemo heširati i preslikati u celobrojnu vrednost i umesto niski uporediti dobijene celobrojne vrednosti. Upoređivanje niski svođenjem na upoređivanje njihovih heš-vrednosti je složenosti $O(1)$. Međutim, ne treba zanemariti činjenicu da je samo heširanje niski s i t, kao što ćemo uskoro videti, složenosti $O(|s| + |t|)$.

4.1.2 Definisanje heš-funkcije

U jeziku C++ na raspolaganju nam je struktura `hash`, koja se može upotrebiti za računanje heš-vrednosti niski (ali i heš-vrednosti ostalih osnovnih tipova podataka).

```
hash<string> h;
cout << h("abrakadabra") << endl;
```

Ipak, u većini narednih algoritama koristićemo heš-funkcije koje ćemo sami definisati.

Jasno je da heš-funkcija treba da zavisi od multiskupa karaktera koji se javljaju u niski i od redosleda karaktera u niski. Niske `maja` i `mara` bi trebalo da imaju različite heš-vrednosti, kao i niske `maja` i `jama`. Dobar i široko rasprostranjen način definisanja heš-funkcije niske s dužine n je korišćenjem *polinomske heš-funkcije* (engl. polynomial rolling hash function). Ona dolazi u varijanti sleva-nadesno i zdesna-nalevo.

4.1.2.1 Varijanta sleva-nadesno

Polinomska heš-funkcija sleva-nadesno za nisku s se definiše na sledeći način:

$$\begin{aligned} h(s) &= (s[0] \cdot p^{n-1} + s[1] \cdot p^{n-2} + \ldots + s[n-1]p^0) \bmod m \\ &= (\sum_{i=0}^{n-1} s[i] \cdot p^{n-i-1}) \bmod m, \end{aligned} \tag{4.1}$$

pri čemu je $s[i]$ celobrojni kôd i-tog karaktera niske s dužine n, a p i m su neki unapred odabrani, pozitivni brojevi. Primetimo da prethodna definicija polinomske heš-funkcije odgovara predstavljanju "broja" s u brojevnom sistemu sa osnovom p (uz dodatno određivanje ostatka pri deljenju sa m na kraju).

Potrebno je svaki karakter niske s kodirati celim brojem. Na primer, kada se kodiraju samo mala slova engleske abecede, moguće je koristiti sledeće kodiranje: $a \to 1, b \to 2, \ldots, z \to 26$. Pridruživanje koda

[1] Poslednji od pomenutih scenarija poznat je pod nazivom *rođendanski paradoks* (engl. birthday paradox) i odnosi se na sledeći kontekst: ako se u jednoj sobi nalazi n osoba, verovatnoća da neke dve osobe imaju rođendan istog dana za $n = 23$ iznosi oko 50%, dok za $n = 70$ ona iznosi čak 99.9%.

0 nekom karakteru (na primer, $a \to 0$), nije pogodno jer bi onda vrednosti heš-funkcije svih niski koje se sastoje od tog karaktera (na primer, `a`, `aa`, `aaa`, `aaaa`, ...) bile jednake 0, dok bi niske od kojih se jedna dobija dodavanjem nekoliko tih karaktera na početak druge (na primer, niske `na` i `ana`) imale istu vrednost heš-funkcije.

Pažljiv odabir parametara p i m je važan da bi se osigurala dobra svojstva heš-funkcije. Često se za p bira prvi prost broj veći od broja karaktera u ulaznoj azbuci. Na primer, ako se niske sastoje samo od malih slova engleske abecede, onda je dobar izbor $p = 31$. Naime, pokazuje se da kada je vrednost parametra p manja od veličine azbuke, onda je `p` kôd nekog mogućeg karaktera niske i lako je pronaći dve niske već dužine 2 kod kojih se javlja kolizija (na primer, ako je dopušten kôd 0, to mogu biti niske čiji su kodovi karaktera `[1, 0]` i `[0, p]`). Poželjno je, takođe, da vrednost parametra m bude neki veliki broj, jer je verovatnoća da dve slučajno odabrane niske imaju istu heš-vrednost približno jednaka $1/m$. Ipak, izbegavaćemo prevelike vrednosti za parametar m, jer je pogodno da vrednosti $m \cdot m$ i $p \cdot m$ ne izazivaju prekoračenje (što ćemo obrazložiti nešto kasnije).

Kao i za parametar p, i za parametar m je pogodno birati neki prost broj. Prosti brojevi se koriste da se ne bi dešavao veliki broj kolizija kada skup niski koje se heširaju ispoljava određena matematička svojstva (na primer kada su kodovi svih karaktera koji se javljaju u tom skupu umnošci istog broja, recimo parni brojevi). U narednim primerima za vrednost parametra m biraćemo prost broj $10^9 + 9$. Ona je manja od 2^{30}, pa vrednost $m \cdot m$ staje u 64-bitni ceo broj. Primetimo i sledeće: kada izračunavanja ne bismo vršili po modulu m, odnosno ako u definiciji polinomske heš-funkcije ne bi figurisala vrednost m, onda bi se, usled prekoračenja, operacije vršile po modulu broja podržanih vrednosti odgovarajućeg celobrojnog tipa (dakle po modulu 2^{32} ili 2^{64}).

Ako za $m = 10^9 + 9$ razmotrimo skup niski sastavljenih od malih slova engleske abecede, takvih da je dužina niske manja ili jednaka 7, ukupan broj ovakvih niski iznosi: $26^0 + 26^1 + 26^2 + 26^3 + 26^4 + 26^5 + 26^6 + 26^7 = 8353082583 > 10^9 + 9 = m$. U ovoj situaciji garantovano je postojanje kolizije.

Izračunavanje heš-vrednosti se na osnovu definisane polinomske heš-funkcije sprovodi primenom Hornerove šeme i linearne je složenosti $O(|s|)$ u odnosu na dužinu niske $|s|$. Implementacija u jeziku C++ može biti sledeća:

```
long long izracunajHesVrednost(const string &s) {
  int p = 31;
  int m = 1e9 + 9;

  // vrednost hes-funkcije niske s racunamo u duhu Hornerove sheme
  long long h = 0;
  for (int i = 0; i < s.size(); i++) {
    h = (h * p + (s[i] - 'a' + 1)) % m;
  }
  return h;
}
```

Da bi se račun sveo na manje brojeve, u funkciji za računanje heš-vrednosti niske s dužine n međurezultati su računati po modulu m, odnosno razmatran je niz međuvrednosti h_i za $i = 0, 1, \ldots, n$.

$$\begin{aligned} h_0 &= 0 \\ h_{i+1} &= (h_i \cdot p + s[i]) \bmod m, \quad 0 \leq i < n \end{aligned}$$

Konačna heš-vrednost je vrednost h_n.

Računanje međurezultata po modulu m matematički je korektno na osnovu pravila modularne aritmetike.

Primetimo da pošto je $h_i < m$, $s[i] < p$ i pošto $p \cdot m$ ne izaziva prekoračenje, prilikom izračunavanja vrednosti h_{i+1} ne dolazi do prekoračenja.

Ova polinomska heš-funkcija ima svojstvo da joj se vrednost može lako ažurirati kada se simboli dodaju ili uklanjaju sa krajeva niske (odatle potiče termin `rolling` u engleskom nazivu). Naime, ako poznajemo heš-vrednost niske $s_0 s_1 \ldots s_{n-1}$, tada lako možemo izračunati heš-vrednost niske $s_1 s_2 \ldots s_n$. Prva vrednost je jednaka $h_0 = (s[0] \cdot p^{n-1} + s[1] \cdot p^{n-2} + \ldots + s[n-1]) \bmod m$, a druga $h_1 = (s[1] \cdot p^{n-1} + s[2] \cdot p^{n-2} + \ldots + s[n]) \bmod m$, pa je

$$h_1 = ((h_0 - s[0]p^{n-1}) \cdot p + s[n]) \bmod m.$$

4.1.2.2 Varijanta zdesna-nalevo

Polinomska heš-funkcija zdesna-nalevo se definiše na sledeći način:

$$\begin{aligned} h'(s) &= (s[0] + s[1] \cdot p + \ldots + s[n-1] \cdot p^{n-1}) \bmod m \\ &= (\textstyle\sum_{i=0}^{n-1} s[i] \cdot p^i) \bmod m \end{aligned} \tag{4.2}$$

kod koje je, nasuprot prethodno definisanoj heš-funkciji, najmanji stepen broja p uz prvi karakter niske s, a najveći uz poslednji karakter niske.

I ova polinomska heš-funkcija ima svojstvo da joj se vrednost može lako ažurirati kada se simboli dodaju ili uklanjaju sa krajeva niske. Naime, ako poznajemo heš-vrednost niske $s_0 s_1 \ldots s_{n-1}$, tada lako možemo izračunati heš-vrednost niske $s_1 s_2 \ldots s_n$. Prva vrednost je jednaka $h_0 = (s[0] + s[1] \cdot p + \ldots + s[n-1] \cdot p^{n-1}) \bmod m$, a druga $h_1 = (s[1] + s[2] \cdot p + \ldots + s[n] \cdot p^{n-1}) \bmod m$, pa je

$$h_1 = \left(\frac{h_0 - s[0]}{p} + s[n] \cdot p^{n-1} \right) \bmod m.$$

Umesto deljenja sa p, moguće je odrediti njegov modularni inverz (po modulu m) i umesto deljenja vršiti množenje.

Za izračunavanje ove varijante heš-funkcije, poželjno je razmatrati niz h'_i za $i = n, n-1, \ldots, 0$, pri čemu je cilj izračunati vrednost h'_0:

$$\begin{aligned} h'_n &= 0 \\ h'_i &= (h'_{i+1} \cdot p + s[i]) \bmod m, \quad 0 \leq i < n \end{aligned}$$

U jeziku C++ implementacija može biti sledeća:

```
long long izracunajHesVrednost(const string &s) {
  int p = 31;
  int m = 1e9 + 9;

  // vrednost hes-funkcije niske s racunamo u duhu Hornerove sheme
  // razmatranjem karaktera niske u obratnom poretku
  long long h = 0;
  for (int i = s.size() - 1; i >= 0; i--) {
    h = (h * p + (s[i] - 'a' + 1)) % m;
  }
  return h;
}
```

4.1.3 Neke primene heširanja niski

4.1.3.1 Identifikovanje duplikata

Razmotrimo jedan problem koji se efikasno rešava tehnikom heširanja niski.

Problem

Dato je n niski $s_1, s_2, \ldots, s_n$ od kojih je svaka dužine maksimalno m. Pronaći sve duplikate među niskama i podeliti skup niski na grupe jednakih niski.

Primer 4.1.1

Ako se sa ulaza učitaju redom niske `ana`, `a`, `dvorana`, `ana`, `banana`, `ana`, `kopakabana` *i* `banana`, *očekujemo da dobijemo 5 različitih grupa: prva se sastoji od tri pojavljivanja niske* `ana`, *druga od jednog pojavljivanja niske* `a`, *treća grupa od jednog pojavljivanja niske* `dvorana`, *četvrta od dva pojavljivanja niske* `banana` *i peta od jednog pojavljivanja niske* `kopakabana`.

Direktan pristup sastoji se u poređenju svake niske sa svakom drugom: poređenje dve niske maksimalne dužine m karakter po karakter je u najgorem slučaju je složenosti $O(m)$ (doduše, može se očekivati da se najgori slučaj ne dešava često tj. da će se različitost dve niske ustanoviti već nakon gledanja njihovih nekoliko početnih karaktera). Ukupan broj poređenja niski je reda $O(n^2)$, te je ukupna složenost odgovarajućeg algoritma $O(n^2 m)$.

Efikasnije rešenje dobija se sortiranjem svih niski i traženjem duplikata u sortiranom nizu. Sortiranje niski uključuje $O(n \log n)$ upoređivanja niski, a svako poređenje niski je u najgorem slučaju složenosti $O(m)$, te je ukupna složenost sortiranja niski jednaka $O(nm \log n)$. Dodatno, prolazak kroz skup niski u sortiranom redosledu i identifikovanje istih niski je složenosti $O(nm)$, te je ukupna složenost ovog algoritma $O(nm \log n)$.

Pristup zasnovan na heširanju niski redukuje vreme poređenja dve niske na $O(1)$ (pod pretpostavkom da ne proveravamo da li je došlo do kolizije). Na taj način dobijamo algoritam složenosti $O(nm + n \log n)$, gde složenost $O(nm)$ potiče od računanja heš-vrednosti svih niski, a $O(n \log n)$ od sortiranja niski na osnovu njihovih heš-vrednosti. Dodatno, prolazak kroz heš-vrednosti niski u sortiranom poretku i identifikovanje istih niski je složenosti $O(n)$. Ako želimo da budemo sigurni u ispravnost algoritma, kada su heš-vrednosti nekih niski jednake, potrebno je eksplicitno uporediti te niske, za šta je u najgorem slučaju, kada su sve heš-vrednosti jednake, potrebno n^2 poređenja niski, koja zahtevaju vreme $O(m)$, što zahteva dodatno vreme $O(n^2 m)$. Međutim, realno je očekivati da su grupe jednakih heš-vrednosti mnogo manje i da će ovo vreme biti mnogo manje.

Implementacija u jeziku C++ je data u nastavku. U njoj se ne proverava postojanje kolizija tj. pretpostavlja se da jednakim heš-vrednostima odgovaraju jednake niske.

```
void grupisiIsteNiske(const vector<string> &niske){
  // broj niski
  int n = niske.size();
  // vektor parova hash vrednosti i pozicije niske u polaznom nizu
  vector<pair<long long, int>> h(n);
  // izracunavamo hash vrednost svake niske;
  // uz hes vrednost niske cuvamo i njen indeks u polaznom nizu
  for (int i = 0; i < n; i++)
    h[i] = {izracunajHesVrednost(niske[i]), i};

  // sortiramo niz hes vrednosti
  sort(h.begin(),h.end());
```

```
  // svaki element vektora sadrzi niz indeksa niski
  // koje su medjusobno jednake
  vector<vector<int>> grupe;

  // prolazimo skupom svih niski u sortiranom poretku
  for (int i = 0; i < n; i++){
    // ukoliko se radi o prvoj niski u sortiranom poretku
    // ili o niski koja nije jednaka prethodnoj u sortiranom poretku
    // onda je potrebna nova grupa
    if (i == 0 || h[i].first != h[i-1].first){
      vector<int> novaGrupa;
      grupe.push_back(novaGrupa);
    }
    // u poslednju (tekucu) grupu dodajemo na kraj indeks niske
    // koja ima tekucu hash vrednost
    grupe.back().push_back(h[i].second);
  }

  // stampamo niske po grupama
  for (int i = 0; i < grupe.size(); i++){
    cout << "Grupa broj " << i << endl;
    for (int j = 0; j < grupe[i].size(); j++)
      cout << niske[grupe[i][j]] << " ";
    cout << endl;
  }
}
```

4.1.3.2 Računanje heš-vrednosti podniski (segmenata) niske

Problem

Data je niska s dužine n. Napisati algoritam kojim se za date parove indeksa i i j za koje važi $0 \leq i \leq j < n$, efikasno izračunavaju heš-vrednosti podniski $s[i..j]$ polazne niske (podniske su segmenti susednih karaktera polazne niske s od pozicije i zaključno sa pozicijom j).

Primer 4.1.2

Za nisku s koja je jednaka `ananas` *i za vrednosti $i = 1$ i $j = 4$, potrebno je izračunati heš-vrednost niske* `nana`.

Polinomska heš-funkcija sleva-nadesno

Razmotrimo najpre prvu varijantu polinomske heš-funkcije koja je zadata formulom (4.1). Prema definiciji heš-funkcije važi:

$$\begin{aligned} h(s[i..j]) &= (s[i] \cdot p^{j-i} + s[i+1] \cdot p^{j-i-1} + \ldots + s[j]) \bmod m \\ &= \left(\sum_{k=i}^{j} s[k] \cdot p^{j-k} \right) \bmod m \end{aligned}$$

Ova vrednost može se direktno izračunati algoritmom vremenske složenosti $O(j-i)$, odnosno u najgorem slučaju složenosti $O(|s|)$, gde je $|s|$ dužina niske s. Pretpostavimo da je veliki broj puta, npr. q puta,

potrebno računati heš-vrednosti različitih segmenata date niske. Naivni pristup bi u najgorem slučaju ovaj problem rešio u vremenu $O(q \cdot |s|)$. Da li možemo efikasnije rešiti ovaj problem?

Upotrebimo ideju prefiksnih suma, koja omogućuje da se zbir svakog segmenta niza može izraziti kao razlika dva zbira prefiksa. Definišimo niz H_k tako da je $H_0 = 0$, a $H_k = h(s[0..k-1])$. Zapišimo čemu su jednake heš-vrednosti prefiksa niske s dužina redom $j+1$ i i:

$$\begin{aligned} H_{j+1} = h(s[0..j]) &= (s[0] \cdot p^j + s[1] \cdot p^{j-1} + ... + s[j]) \bmod m, \\ H_i = h(s[0..i-1]) &= (s[0] \cdot p^{i-1} + s[1] \cdot p^{i-2} + ... + s[i-1]) \bmod m. \end{aligned}$$

Pri tom smatramo da je $h(s[0..-1]) = 0$ (što može da se desi za $i = 0$). Primetimo da je:

$$\begin{aligned} h(s[i..j]) \bmod m &= (s[i] \cdot p^{j-i} + s[i+1] \cdot p^{j-(i+1)} + ... + s[j]) \bmod m \\ &= h(s[0..j]) - p^{j-(i-1)} \cdot h(s[0..i-1]) = H_{j+1} - p^{j+1-i} \cdot H_i \end{aligned} \tag{4.3}$$

Pošto promenljive i i j uzimaju vrednosti od 0 do $|s|-1$, izraz $j+1-i$ uzima vrednosti od 1 do $|s|$, te je u stvari potrebno izračunati vrednost p^k za sve vrednosti $0 \leq k \leq |s|$. Dakle, ako za datu nisku znamo heš-vrednosti svih njenih prefiksa i vrednosti p^k za $k = 0, 1, ..., |s|$, onda na osnovu jednakosti (4.3) algoritmom složenosti $O(1)$ možemo izračunati heš-vrednost proizvoljnog segmenta te niske. Heš-vrednost svih prefiksa niske s i vrednosti $p^k \bmod m$ za svako $k = 1, 2, ..., |s|$ mogu se izračunati inkrementalno, algoritmom čija je vremenska složenost $O(|s|)$, pa je ukupna složenost računanja heš-vrednosti q različitih segmenata $O(|s| + q)$.

Implementacija u jeziku C++ je prikazana u nastavku.

```
int p = 31;
long long m = 1e9 + 9;

// stepeni broja p
vector<long long> pStepen;

// hes vrednosti svih prefiksa niske tj. vrednosti niza Hk
vector<long long> hesPrefiksa;

void izracunajStepeneBrojaP(int n){
  pStepen.resize(n+1);
  pStepen[0] = 1;
  for (int i = 1; i <= n; i++)
    pStepen[i] = (pStepen[i-1] * p) % m;
}

void izracunajHesevePrefiksa(string s){
  int n = s.size();
  hesPrefiksa.resize(n+1,0);
  // racunamo heseve svih prefiksa datog stringa
  for (int i = 0; i < n; i++)
    hesPrefiksa[i+1] = (hesPrefiksa[i] * p + (s[i]-'a'+1)) % m;
}

long long hesSegmenta(string const& s, int i, int j){
  // hes-vrednost segmenta racunamo preko hes-vrednosti prefiksa
```

```
  long long hash = (hesPrefiksa[j+1] -
                    (hesPrefiksa[i] * pStepen[j+1-i]) % m + m) % m;
  return hash;
}

int main(){
  string s;
  cin >> s;
  int n = s.size();

  izracunajStepeneBrojaP(n);
  izracunajHesevePrefiksa(s);

  int i, j;
  cin >> i >> j;
  cout << "Hes-vrednost segmenta " <<  s.substr(i, j-i+1)
       << " je: " << hesSegmenta(s, i, j) << endl;
  return 0;
}
```

Primetimo da se prilikom računanja heš-vrednosti segmenta izvršava množenje heš-vrednost prefiksa odgovarajućim stepenom broja p po modulu m. Da ovaj proizvod ne bi doveo do prekoračenja, potreban je uslov da $m \cdot m$ ne izaziva prekoračenje.

Polinomska heš-funkcija zdesna-nalevo

Razmotrimo sada korišćenje druge predložene heš-funkcije. Tada je heš-vrednost segmenta $s[i..j]$ jednaka:

$$\begin{aligned} h'(s[i..j]) &= (s[i] + s[i+1] \cdot p + ... + s[j] \cdot p^{j-i}) \bmod m \\ &= \left(\sum_{k=i}^{j} s[k] \cdot p^{k-i} \right) \bmod m \end{aligned}$$

Ako obe strane ove jednakosti pomnožimo sa p^i dobijamo:

$$\begin{aligned} h'(s[i..j]) \cdot p^i &= \left(\sum_{k=i}^{j} s[k] \cdot p^{k} \right) \bmod m \\ &= (h'(s[0..j]) - h'(s[0..i-1])) \bmod m \\ &= (H'_{j+1} - H'_i) \bmod m \end{aligned} \tag{4.4}$$

gde važi da je $h'(s[0..-1]) = 0$ (što može da se desi za $i = 0$) i gde smo niz H' definisali tako da je $H'_0 = 0$, a $H'_{k+1} = h'(s[0..k])$.

Primetimo da je za računanje vrednosti heš-funkcije nekog segmenta prema formuli (4.4) potrebno izvršiti deljenje izraza $H'_{j+1} - H'_i$ vrednošću p^i po modulu m. Za to je potrebno odrediti multiplikativni inverz broja p^i po modulu m, odnosno broj x tako da važi $p^i \cdot x \equiv 1 \bmod m$, a zatim izvršiti množenje izraza $H'_{j+1} - H'_i$ brojem x. Pošto sva izračunavanja vršimo po modulu m, a s obzirom na to da smo na početku pretpostavili da su brojevi m i p prosti, na osnovu male Fermaove teoreme za svaki broj a koji je uzajamno prost sa m važi $a^{m-1} \equiv 1 \bmod m$, odnosno multiplikativni inverz broja a po modulu m je $a^{m-2} \bmod m$. Ako u ovu jednakost umesto a uvrstimo vrednost p^i (koji jeste uzajamno prost sa m), dobijamo multiplikativni inverz za proizvoljno p^i po modulu m.

Dakle, ako su unapred poznate heš-vrednosti svih prefiksa niske s i vrednosti multiplikativnog inverza broja $p^i, 1 \leq i \leq |s|$ po modulu m, izračunavanje heš-vrednosti proizvoljnog segmenta polazne niske može se na osnovu formule (4.4) izvršiti algoritmom složenosti $O(1)$. Primetimo da je faza pretprocesiranja koja se sastoji od računanja heš-vrednosti svih prefiksa niske i vrednosti inverza broja p^i za svako i vremenske složenosti $O(n \cdot \log m)$.

```
// stepeni broja p i njihovi inverzi po modulu m
vector<long long> pStepen, invpStepen;

// funkcija za brzo stepenovanje po modulu m
long long stepen_mod(long long a, long long b, long long m) {
  if (b == 0)
    return 1;
  if (b % 2 == 0)
    return stepen_mod((a * a) % m, b / 2, m);
  else
    return (a * stepen_mod(a, b - 1, m)) % m;
}

// racunanje inverza koriscenjem male Fermaove teoreme
long long modInverz(long long a, long long m) {
  return stepen_mod(a, m - 2, m);

}

// racunamo stepene broja p i njihove inverze
void izracunajStepeneBrojaP(int n){
  pStepen.resize(n+1);
  invpStepen.resize(n+1);
  pStepen[0] = 1;
  invpStepen[0] = 1;
  for (int i = 1; i <= n; i++) {
    pStepen[i] = (pStepen[i-1] * p) % m;
    invpStepen[i] = modInverz(pStepen[i], m);
  }
}

void izracunajHesevePrefiksa(string s) {
  int n = s.size();
  hesPrefiksa.resize(n+1, 0);
  // racunamo hasheve svih prefiksa datog stringa
  for (int i = 0; i < n; i++)
    hesPrefiksa[i+1] = (hesPrefiksa[i] + (s[i]-'a'+1) * pStepen[i]) % m;
}

long long hesSegmenta(string const& s, int i, int j) {
  int n = s.size();
  // hes vrednost segmenta racunamo preko hash vrednosti prefiksa
  long long hash = (((hesPrefiksa[j+1] - hesPrefiksa[i] + m) % m) *
                   invpStepen[i]) % m;
  return hash;
}
```

4.1.3.3 Traženje niske u tekstu (Rabin-Karpov algoritam)

Rabin i Karp su 1987. godine predložili algoritam koji rešava problem traženja niske (engl. pattern) u tekstu (engl. text) korišćenjem tehnike heširanja niski.

> **Problem**
>
> *Za dati tekst $T = t_0 t_1 \dots t_{n-1}$ i nisku $P = p_0 p_1 \dots p_{m-1}$ odrediti sve podniske uzastopnih elemenata (segmente) niske T koje su jednake P.*

Ideja Rabin-Karpovog algoritma je sledeća: korišćenjem polinomske heš-funkcije izračunati heš-vrednost niske P i svih segmenata teksta T, koji imaju dužinu $|P| = m$ i uporediti njihove vrednosti. Ako se heš-vrednost niske i segmenta ne poklopi, tekući segment se sigurno ne poklapa sa podniskom koju tražimo i prelazi se na naredni segment. Ako se heš-vrednosti poklope, tada treba izvršiti eksplicitnu proveru da li se tekući segment poklapa sa podniskom koju tražimo. Videli smo da se heš-vrednost proizvoljnog segmenta teksta može izračunati algoritmom vremenske složenosti $O(1)$ na osnovu unapred izračunatih heš-vrednosti svih prefiksa teksta T i stepena broja p (koji se koristi u heš-funkciji) po modulu m. Izračunavanje heš-vrednosti niske P je složenosti $O(|P|)$, izračunavanje heš-vrednosti svih prefiksa datog teksta (tj. date niske) T je složenosti $O(|T|)$, dok je ukupna složenost poređenja heš vrednosti svih segmenata teksta T koji imaju dužinu $|P|$ sa heš-vrednošću niske P reda $O(|T|)$. Stoga je ukupna složenost Rabin-Karpovog algoritma $O(|T| + |P|) = O(n + m)$.

Razmotrimo implementaciju Rabin-Karpovovog algoritma u slučaju kada je heš funkcija zadata formulom (4.1).

```
// Rabin-Karpov algoritam za trazenje niske u tekstu
vector<int> traziRabinKarp(const string &niska, const string &tekst) {
  int M = niska.size();
  int N = tekst.size();
  int p = 31;
  int m = 1e9 + 9;

  // racunamo stepene broja p po modulu m
  vector<long long> pStepen(max(M, N));
  pStepen[0] = 1;
  for (int i = 1; i < pStepen.size(); i++)
    pStepen[i] = (pStepen[i-1] * p) % m;

  // racunamo inkrementalno hes-vrednosti svih prefiksa datog teksta
  vector<long long> hesPrefiksa(N+1,0);
  for (int i = 0; i < N; i++)
    hesPrefiksa[i+1] = (hesPrefiksa[i]*p + (tekst[i] - 'a' + 1)) % m;

  // racunamo hes-vrednost date niske
  long long hesNiske = 0;
  for (int i = 0; i < M; i++)
    hesNiske = (hesNiske*p + (niska[i] - 'a' + 1)) % m;

  // vektor pocetnog indeksa pojavljivanja niske u tekstu
  vector<int> pojave;
  for (int i = 0; i <= N - M; i++){
    // racunamo hes vrednosti segmenta duzine M
```

```
    // koji pocinje na poziciji i
    long long hesSegmenta = (hesPrefiksa[i+M] - heshPrefiksa[i] * pStepen[M] + m) % m;
    // ako se poklapaju hesh vrednosti segmenta teksta i niske
    // pomnozene istim stepenom broja p, pamtimo indeks i
    if (hesSegmenta == hesNiske)
      // ovde se može dodati i eksplicitna provera poklapanja
      pojave.push_back(i);
  }
  return pojave;
}
```

Rabin-Karpov algoritam ima različite zanimljive primene: koristi se za detekciju plagijarizma u dokumentu, kao i za kompresiju podataka, identifikovanje i evidentiranje dupliranih podataka radi efikasnijeg skladištenja podataka.

4.1.3.4 Broj različitih podniski (segmenata) niske

Problem

Data je niska s dužine n, koja se sastoji samo od malih slova engleske abecede. Izračunati broj različitih segmenata ove niske.

Primer 4.1.3

Ako je niska s jednaka ananas*, onda postoje tri različita segmenta dužine 1:* a*,* n *i* s*, tri različita segmenta dužine 2:* an*,* na *i* as*, tri različita segmenta dužine 3:* ana*,* nan *i* nas*, tri različita segmenta dužine 4:* anan*,* nana*,* anas*, dva različita segmenta dužine 5:* anana*,* nanas *i jedan segment dužine 6:* ananas*, te niska s ima ukupno* $3+3+3+3+2+1=15$ *različitih segmenata.*

Direktan pristup bi bio da se sve podniske ubace u skup (bilo uređen ili neuređen) i da se proveri broj elemenata tog skupa, međutim, ova ideja se može optimizovati.

Naime, dovoljno je porediti samo segmente istih dužina jer samo oni mogu biti međusobno jednaki.

Ukoliko koristimo heš-funkciju zadatu formulom (4.2), umesto da računamo i poredimo tačne heš-vrednosti dva segmenta iste dužine korišćenjem modularnih multiplikativnih inverza, dovoljno je izračunati heš-vrednosti segmenata pomnožene nekim (istim) stepenom broja p. Pretpostavimo da već imamo izračunate heš-vrednosti dva segmenta x i y, jednog pomnoženog sa p^i, a drugog sa p^j. Bez narušavanja opštosti možemo pretpostaviti da je $i < j$; da bismo dobili obe heš-vrednosti pomnožene istim stepenom broja p, dovoljno je da heš-vrednost segmenta x pomnožimo sa p^{j-i}.

Primer 4.1.4

Razmotrimo, na primer, segmente $s[2..4]$ i $s[5..7]$ dužine 3 niske s koja je dužine 10. Iz jednakosti (4.4) sledi:

$$\begin{aligned} h_1 &= h(s[2..4]) \cdot p^2 = h(s[0..4]) - h(s[0..1]) \bmod m \\ h_2 &= h(s[5..7]) \cdot p^5 = h(s[0..7]) - h(s[0..4]) \bmod m \end{aligned}$$

Umesto da računamo multiplikativne inverze brojeva p^2 i p^5, da bismo dobili tačne heš-vrednosti segmenata $s[2..4]$ i $s[5..7]$, možemo porediti vrednosti $h_1 \cdot p^{5-2}$ i h_2.

Dakle, prolazićemo redom kroz sve moguće dužine $l = 1, 2, \ldots, n$ segmenata niske s i konstruisati seriju heš-vrednosti svih segmenata dužine l koji su pomnoženi nekim (istim, maksimalnim) stepenom broja p. Broj različitih elemenata te serije (što je, pod pretpostavkom da nema kolizija, broj različitih segmenata dužine l) možemo odrediti korišćenjem skupa (na primer, uređenog). Ukupan Broj različitih segmenata jednak je zbiru broja različitih segmenata dužine l u niski, za svako moguće l. Implementacija ovog pristupa u jeziku C++ je data u nastavku.

```
int izbrojRazliciteSegmente(const string &s) {
  int n = s.size();
  int p = 31;
  int m = 1e9 + 9;

  // racunamo stepene broja p po modulu m
  vector<long long> pStepen(n);
  pStepen[0] = 1;
  for (int i = 1; i < n; i++)
    pStepen[i] = (pStepen[i-1] * p) % m;

  // racunamo hes-vrednosti svih prefiksa date niske
  vector<long long> h(n+1, 0);
  for (int i = 0; i < n; i++)
    h[i+1] = (h[i] + (s[i] - 'a' + 1) * pStepen[i]) % m;

  // broj razlicitih segmenata
  int brSegmenata = 0;
  // za svaku mogucu duzinu segmenta
  for (int l = 1; l <= n; l++) {
    // skup hes vrednosti segmenata duzine l pomnozenih sa p^{n-1}
    set<long long> hesevi_duzine_l;
    // prolazimo kroz sve segmente duzine l
    for (int i = 0; i <= n - l; i++) {
      // hes-vrednost segmenta racunamo
      // kao razliku hes-vrednosti odgovarajucih prefiksa
      long long hTekuce = (h[i+l] - h[i] + m) % m;
      // racunamo hes vrednost segmenta pomnozenu sa p^{n-1} po modulu m
      // tako sto je mnozimo sa p^{n-i-1}
      hTekuce = (hTekuce * pStepen[n - i - 1]) % m;
      // hes-vrednost dodajemo u skup,
      // isti segmenti imace istu hesh vrednost pa se nece dva puta racunati
      heshevi_duzine_l.insert(hTekuce);
    }
    brSegmenata += heshevi_duzine_l.size();
  }
  return brSegmenata;
}
```

Primetimo da se u prethodnoj implementaciji prilikom računanja proizvoda heš-vrednosti segmenta i vrednosti p^{n-1} po modulu m množe dve vrednosti iz opsega $[0, m)$, te da ne bi bilo prekoračenja prilikom računanja ovog proizvoda, vrednost parametra m biramo tako da $m \cdot m$ staje u 64-bitni ceo broj.

Operacije za rad sa uređenim skupom od k elemenata su složenosti $O(\log k)$, a različitih segmenata ima ukupno $O(n^2)$, te složenost ovog algoritma iznosi $O(n^2 \log n)$.

Umetanjem heš-vrednosti u skup, umesto umetanja samih podniski u skup značajno je smanjena memorijska složenost skupa, koja uz heširanje iznosi $O(n)$, a bez heširanja bi iznosila $O(n^2)$.

Istaknimo i to da se u ovom kontekstu, kada radimo sa različitim segmentima niske, isplati uložiti dodatno vreme za računanje heš-vrednosti svih prefiksa date niske, jer ćemo izračunate heš-vrednosti veći broj puta koristiti.

4.2 Z-niz

Mnogi problemi nad niskama mogu se efikasno rešiti korišćenjem *z-niza* (engl. z-array, z-function); na primer, ispitivanje da li se jedna niska javlja unutar druge, određivanje perioda niske (u smislu određivanja najkraće niske takve da se polazna niska može predstaviti nadovezivanjem te niske određen broj puta), itd. z-niz niske s dužine n je niz dužine n koji na poziciji $k = 0, 1, \ldots, n-1$ sadrži dužinu z_k najduže podniske (tj. najdužeg segmenta) niske s, koja počinje na poziciji k i prefiks je niske s. Dakle, $s[0..z_k-1]$ se poklapa sa $s[k..k+z_k-1]$, a karakteri $s[z_k]$ i $s[k+z_k]$ su različiti ili je pak niska s dužine $k+z_k$.

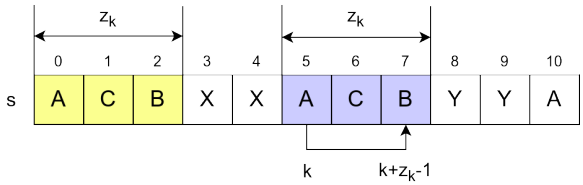
Primer 4.2.1

Za nisku `alfalfa` *je* $z_3 = 4$*, jer podniska* `alfa` *koja počinje na poziciji* 3 *i završava se na kraju niske ima dužinu* 4 *i poklapa sa prefiksom niske dužine* 4*.*

Primer 4.2.2

Za nisku `photophosphorescent` *važi* $z_5 = z_9 = 3$*, jer i na poziciji* 5 *i na poziciji* 9 *počinje podniska* `pho` *koja ima dužinu* 3*, poklapa se prefiksom niske i ne može da se produži tako da se i dalje poklapa sa odgovarajućim prefiksom.*

Podnisku koja počinje na poziciji k i preklapa se sa nekim prefiksom dužine z_k nazivamo i *z-kutija* (engl. z-box). Primer z-kutije ilustrovan je na slici 4.1. Ako je vrednost $z_k = 0$, tada na poziciji k ne postoji z-kutija.



Slika 4.1: Primer z-kutije koja počinje na poziciji 5 i završava se na poziciji 7.

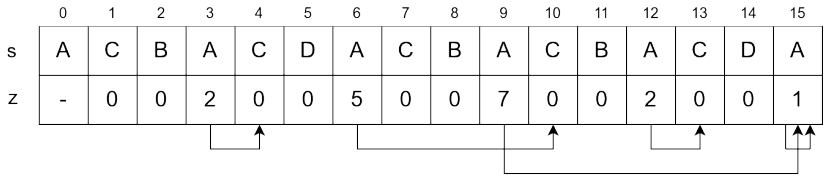
Vrednost niza z na poziciji 0 jednaka je dužini niske jer je kompletna niska uvek prefiks same sebe; međutim ta vrednost se nikada ne koristi.

Primer 4.2.3

z-niz niske `ACBACDACBACBACDA` *prikazan je na slici 4.2. Vrednost* $z_6 = 5$ *označava da je podniska* `ACBAC` *(koja počinje na poziciji* 6 *i dužine je* 5*) prefiks niske* s*, dok podniska* `ACBACB` *(koja počinje na istoj poziciji i dužine je* 6*) nije.*

4.2.1 Konstrukcija z-niza

Jedno važno pitanje je kako što efikasnije konstruisati z-niz za datu nisku.



Slika 4.2: Ilustracija z-niza. z-kutije su označene strelicama.

Problem

Za datu nisku s konstruisati niz z tako da se na svakoj poziciji k niza z nalazi vrednost z_k jednaku najvećoj dužini podniske niske s koja počinje na poziciji k i ujedno je i prefiks niske s (tj. karakteri niske s na pozicijama $[0, z_k)$ i $[k, k + z_k)$ se poklapaju).

4.2.1.1 Algoritam grube sile

Direktno rešenje se sastoji u tome da se za svaki indeks iznova traži najduži poklapajući prefiks niske koji počinje na tekućoj poziciji u niski.

```
vector<int> izracunajZNizTrivijalno(string s) {
  int n = s.size();
  // inicijalizujemo sve vrednosti z-niza na 0
  vector<int> z(n, 0);

  for (int k = 1; k < n; k++) {
    // sve dok ne izađemo iz opsega niske
    // i odgovarajući karakteri se poklapaju
    while (k+z[k] < n && s[z[k]] == s[k+z[k]]) {
      // inkrementiramo vrednost z-niza na odgovarajucoj poziciji
      z[k]++;
    }
  }
  return z;
}
```

Ovo rešenje sadrži dve ugnežđene petlje, a složenost u najgorem slučaju je $O(n^2)$. Zaista, za nisku `AAA...AAA`, algoritam grube sile za računanje z-niza izvršava $\Theta(n^2)$ koraka. U praksi, očekuje se da se na razlike odgovarajućih karaktera naiđe dosta brže.

4.2.1.2 z-algoritam

z-algoritam (engl. z-algorithm) je algoritam za konstrukciju z-niza složenosti $O(n)$. Dobio je ime po strukturi podataka koja se njome konstruiše, tj. po z-nizu. Do njega je prvi došao Gustav Fridrih Hartman 1975. godine, ali je svoju popularnost stekao tek kasnije, nakon što su Martin Eskardo i Rajnard Vilhelm predstavili efikasniju varijantu algoritma. Ideja z-algoritma jeste da se vrednosti z-niza izračunavaju iterativno, sleva nadesno, na osnovu prethodno izračunatih vrednosti z-niza.

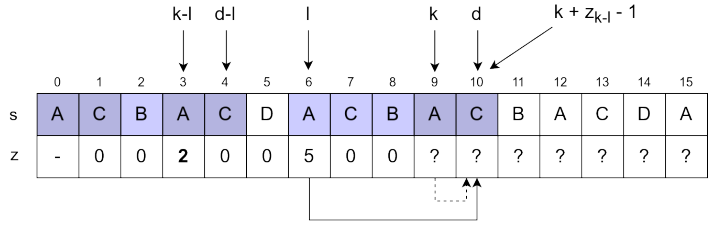
U algoritmu se održava par $[l, d]$ koji ograničava z-kutiju sa najvećom desnom granicom d od svih do sada pronađenih z-kutija (tada se segment $s[l..d]$ poklapa sa prefiksom $s[0..d - l]$ niske s). Činjenicu da je $s[0..d - l] = s[l..d]$ koristićemo za efikasnije računanje vrednosti z-niza na pozicijama $l + 1, l + 2, \ldots, d$.

Dakle, za tekući indeks k za koji treba izračunati vrednost z_k moguća su dva slučaja:

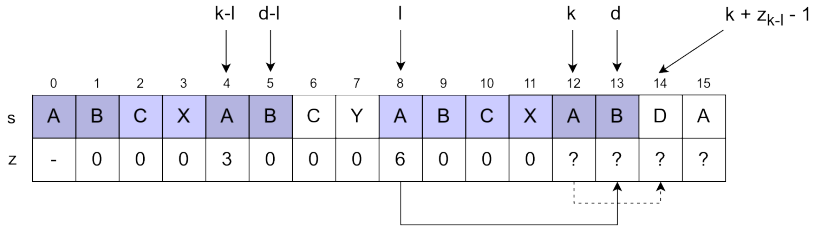
- $l < k \leq d$ – tekuća pozicija je unutar opsega $[l, d]$, pa možemo iskoristiti već izračunate vrednosti z-niza za inicijalizaciju vrednosti z_k (umesto da krećemo od vrednosti 0). Naime, segmenti $s[l..d]$ i $s[0..d - l]$ su jednaki, pa pošto je $l < k \leq d$, jednaki su i segmenti $s[k..d]$ i $s[k - l..d - l]$

pa prilikom računanja vrednosti z_k možemo krenuti od vrednosti z_{k-l} koja je već izračunata (slika 4.3). Međutim, vrednost z_{k-l} u nekim slučajevima može biti prevelika, jer kada se doda poziciji k može da bude veća od vrednost indeksa d, a to nije dobro jer mi ne znamo ništa o karakterima niske s nakon pozicije d. Recimo, u primeru prikazanom na slici 4.4 vrednost z_k ne bi bilo ispravno postaviti na $z_{k-l} = 3$ jer bi nas to izvelo van granica opsega $[l, d]$. Dakle, maksimalna dužina poklopljenog dela može biti jednaka broju karaktera od tekuće pozicije k do poslednje pregledane pozicije d, a to je $d - k + 1$. Stoga se početna vrednost z_k postavlja na $z_k^0 = \min\{d - k + 1, z_{k-l}\}$. Nakon toga utvrđujemo da li se vrednost z_k može povećati pokretanjem trivijalnog algoritma od pozicije $k + z_k^0$.

- $k > d$ – tekuća pozicija k je van opsega $[l, d]$, pa nemamo nikakvih informacija o poziciji k. Stoga vrednost z_k računamo trivijalnim algoritmom, tj. poređenjem karakter po karakter niske s počev od pozicija 0 i k;



Slika 4.3: Slučaj kada se z_k inicijalizuje na z_{k-l}. Provera se nastavlja poređenjem pozicija z_{k-l} i $k + z_{k-l}$.



Slika 4.4: Slučaj kada se z_k inicijalizuje na $d-k+1$, jer je $k+z_{k-l} > d$. Provera se nastavlja poređenjem pozicija $d - k + 1$ i $d + 1$.

Dakle, algoritam razmatra dva slučaja koji se razlikuju samo po inicijalizaciji vrednosti $z[k]$, nakon čega se postupak nastavlja primenom trivijalnog algoritma. Ukoliko dođe do poklapanja dela niske počev od pozicije k sa nekim prefiksom date niske i ako je desna granica preklopljenog segmenta veća od prethodne vrednosti desne granice ($k + z[k] - 1 > d$), ažuriramo granice $[l, d]$ z-kutije koja se prostire najviše nadesno od svih do tada pronađenih z-kutija novim granicama $[k, k + z[k] - 1]$.

Razmotrimo implementaciju z-algoritma.

```
// funkcija koja izracunava sve elemente z-niza
vector<int> izracunajZNiz(const string &s) {
  int n = s.size();
  // inicijalizujemo sve vrednosti z-niza na 0
  vector<int> z(n,0);
  int l = 0;
  int d = 0;

  for (int k = 1; k < n; k++) {
    // ako je tekuca pozicija unutar opsega [l,d]
    // koristimo prethodno izracunatu vrednost za inicijalizaciju
    if (k <= d)
```

```
      z[k] = min(d-k+1, z[k-l]);

    // preskacemo proveru karaktera od pozicije k do pozicije k+z[k]-1;
    // od pozicije k+z[k] poredimo karakter po karakter u niski
    // i sve dok se karakteri poklapaju povecavamo vrednost z[k]
    while (k+z[k] < n && s[z[k]] == s[k+z[k]])
      z[k]++;

    // ako je nova vrednost desnog kraja najdesnije z-kutije
    // veca od prethodne vrednosti, azuriramo interval [l,d]
    if (k + z[k] - 1 > d) {
      l = k;
      d = k + z[k] - 1;
    }
  }
  return z;
}
```

Primetimo da se `while` petlja može uspešno izvršiti samo kada je $d - k + 1 \leq z_{k-l}$, odnosno kada postoji poklapanje dela niske od pozicije k do pozicije d (uključujući i njih) sa nekim prefiksom niske. Kada je $z_{k-l} < d - k + 1$, tada se petlja `while` može preskočiti. To nije urađeno u prethodnom kodu, međutim, u tim slučajevima uslov petlje `while` nije nikada ispunjen, pa se telo petlje ne izvršava ni jednom i ne narušava se efikasnost algoritma (a programski kôd je malo jednostavniji).

Pokažimo da je prikazani algoritam linearne vremenske složenosti, iako sadrži dve ugnežđene petlje. Naime, nakon svakog neuspešnog poređenja karaktera u petlji `while`, tekuća iteracija petlje `for` se završava. Pošto je ukupan broj iteracija spoljašnje petlje `for` jednak n, ukupno možemo imati najviše n nepoklapanja karaktera u petlji `while`. Svaki karakter niske s na poziciji $k + z_k$ koji se uspešno poklopi u unutrašnjoj `while` petlji, više se nikada ne poredi, te je i broj uspešno poklopljenih karaktera u petlji `while` maksimalno n. Dakle, u svim iteracijama spoljašnje petlje `for`, ukupan broj koraka petlje `while` (provera njenog uslova i izvršavanja njenog tela) je $O(n)$, te je ukupna složenost algoritma $O(n)$.

Može se pokazati i da svako izvršavanje tela petlje `while` povećava desnu granicu segmenta d najdesnije z-kutije. Naime, svaka uspešna iteracija petlje `while` poklapa neki novi karakter niske, desno od granice najdesnije z-kutije, te se vrednost promenljive d uvećava za onoliko koliki je broj uspešnih iteracija petlje `while`. Pošto je maksimalna vrednost za d jednaka $n-1$, a početna je jednaka 0, dobijamo da se telo petlje `while` u svim iteracijama spoljašnje petlje zajedno ukupno neće izvršiti više od $n - 1$ puta. I odavde sledi da je složenost algoritma $O(n)$.

Primer 4.2.4

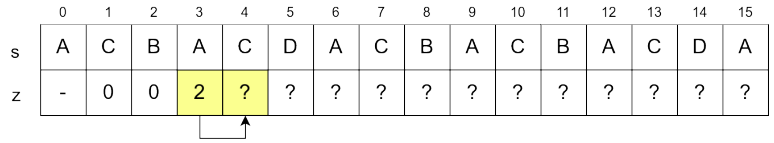
Razmotrimo konstrukciju z-niza za nisku ACBACDACBACBACDA.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s	A	C	B	A	C	D	A	C	B	A	C	B	A	C	D	A
z	-	0	0	2	?	?	?	?	?	?	?	?	?	?	?	?

Početno stanje z-niza.

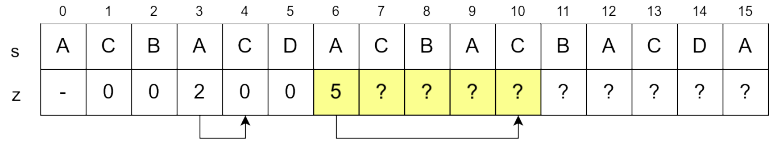
Na početku je $[l, d] = [0, 0]$, pa vrednosti z-niza na pozicijama 1, 2 i 3 računamo trivijalnim algoritmom i dobijamo $z_1 = z_2 = 0, z_3 = 2$. Nakon izračunate vrednosti z_3 ažuriramo opseg $[l, d] = [3, 4]$ (slika 4.5).

Nakon toga, vrednost z-niza na poziciji $k = 4$ dobijamo na osnovu vrednosti $z_{k-l} = z_{4-3} = z_1 = 0$. Vrednosti z_5 i z_6 dobijamo pokretanjem trivijalnog algoritma (jer za $k = 5$ i $k = 6$ i $d = 4$ važi $k > d$) i



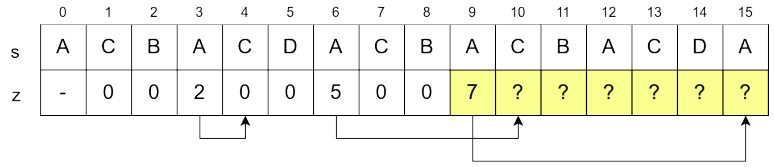
Slika 4.5: Pomeranje najdesnije z-kutije na poziciju $l = 3$, $d = 4$.

dobijamo $z_5 = 0$, *a* $z_6 = 5$. *Pošto za* $k = 6$ *važi* $k + z_k - 1 = 6 + 5 - 1 = 10$, *desna granica tekuće* z*-kutije je veća od prethodne vrednosti* 4, *pa tekući* $[l, d]$ *opseg najdesnije* z*-kutije postaje* $[6, 10]$ *(slika 4.6).*



Slika 4.6: Pomeranje najdesnije z-kutije na poziciju $l = 6$, $d = 10$

Nakon ovog koraka, možemo efikasno izračunati naredne vrednosti z*-niza, jer znamo da su niske* $s[0..4]$ *i* $s[6..10]$ *jednake. Najpre, pošto je* $z_1 = z_2 = 0$ *dobijamo i da je* $z_7 = z_8 = 0$. *Nakon toga, pošto je* $z_3 = 2$ *i* $d - k + 1 = 10 - 9 + 1 = 2$, *znamo da je* $z_9 \geq 2$. *Pošto nemamo informacije o karakterima niske nakon pozicije* 10, *poredimo segmente dalje karakter po karakter. Ispostavlja se da je* $z_9 = 7$. *Pošto za* $k = 9$ *važi* $k + z[k] - 1 = 9 + 7 - 1 = 15$, *desna granica tekuće* z*-kutije je veća od prethodne vrednosti* 10, *pa je novi* $[l, d]$ *opseg najdesnije* z*-kutije jednak* $[9, 15]$ *(slika 4.7).*



Slika 4.7: Pomeranje najdesnije z-kutije na poziciju $l = 9$, $d = 15$

Nakon ovoga, sve preostale vrednosti z*-niza mogu se odrediti na osnovu informacija koje se već nalaze u* z*-nizu (slika 4.8). Obratimo pažnju samo na to da je kod poslednjeg elementa* z_{15} *vrednost* $z_{k-l} = z_6 = 5$, *međutim, ona je veća od* $d - k + 1 = 1$, *pa se zato* z_{15} *inicijalizuje na* $d - k + 1 = 1$ *umesto na* $z_6 = 5$ *i, pošto se došlo do kraja niza, na toj vrednosti i ostaje.*

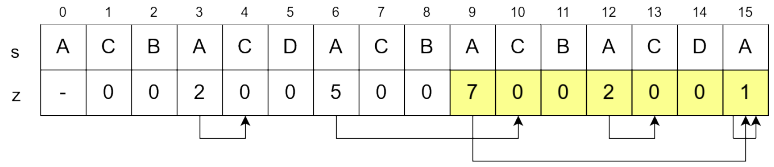
4.2.2 Pretraga teksta primenom z-niza

U ovom poglavlju ćemo prikazati kako se z-niz upotrebljava za traženje niske u tekstu (traženje podniske uzastopnih karaktera unutar duže niske).

Nekada se prilikom obrade niski konstruiše jedna duža niska koja se sastoji od većeg broja niski razdvojenih specijalnim karakterima. To je slučaj i kod algoritma traženja niske u tekstu zasnovanog na z-nizu. Naime, u ovom slučaju možemo konstruisati nisku $p\#t$, gde su niska koja se traži p i tekst t razdvojeni specijalnim karakterom $\#$ koji se ne javlja ni u p ni u t. Z-niz niske $p\#t$ nam može ukazati na to gde se u tekstu t pojavljuje niska p jer će takve pozicije imati z-vrednost jednaku dužini niske p. Dobijene vrednosti z-niza na pozicijama 0 do m, gde je $m = |p|$ dužina niske p, nisu nam bitne, jer se odnose na pozicije unutar niske p, ali ih je neophodno izračunati u okviru z-algoritma.

Primer 4.2.5

Razmotrimo slučaj kada je niska $p = aaba$, *a tekst* $t = aabaacaadaabaaba$. *Formiramo* Z*-niz niske* $p\#t$ *(slika 4.9). Primetimo da su vrednosti* z*-niza na pozicijama* 5, 14 *i* 17 *jednake* 4, *što je dužina niske* p.



Slika 4.8: Izračunati z-niz.

Pošto je ispred teksta t dodato 5 karaktera (4 karaktera niske p i specijalni karakter $\#$), te pozicije odgovaraju pozicijama 0, 9 i 12 u tekstu i na tim pozicijama u tekstu se javlja niska p.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A	A	B	A	#	A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A
-	1	0	1	0	4	1	0	2	1	0	2	1	0	4	1	0	4	1	0	1

Slika 4.9: Traženje niske $aaba$ u tekstu $aabaacaadaabaaba$ korišćenjem z-niza.

Složenost ovog algoritma je $O(m+n)$, gde je $m = |p|$ dužina niske p, a $n = |t|$ dužina teksta t, jer se algoritam svodi na konstrukciju z-niza niske $p\#t$ (za koju smo videli da je linearne složenosti u odnosu na dužinu niza) i prolazak kroz njegove vrednosti.

```
vector<int> traziZNiz(const string& niska, const string& tekst) {
  vector<int> pojavljivanja;
  string zajedno = niska + "#" + tekst;
  int n = zajedno.size();
  int m = niska.size();
  vector<int> zNiz = izracunajZNiz(zajedno);
  for (int i = 0; i < n; i++)
    if (zNiz[i] == m)
       pojavljivanja.push_back(i - m - 1);
  return pojavljivanja;
}

int main() {
  string tekst = "banana";
  string niska = "ana";
  for (int i : traziZNiz(niska, tekst))
      cout << "Niska se javlja u tekstu pocev od pozicije " << i << endl;
  return 0;
}
```

4.3 Knut-Moris-Pratov algoritam

Direktni algoritam za traženje niske p dužine $|p| = m$ u tekstu t dužine $|t| = n$ poredi nisku p sa svim mogućim segmentima $t_k t_{k+1} \dots t_{k+m-1}$ teksta t, za $k = 0, 1, \dots, n-m+1$. Upoređivanje niske p sa tekućim segmentom vrši se karakter po karakter sleva udesno, sve dok se ne ustanovi da su svi karakteri niske jednaki odgovarajućim karakterima segmenta (u tom trenutku prekida se dalje pregledanje segmenta) ili dok se ne naiđe na neslaganje karaktera, odnosno kada se desi $p_i \neq t_{k+i-1}$ za neko i, $0 \leq i \leq m-1$. U drugom slučaju niska koja se traži se "pomera" za jedan karakter udesno, odnosno nastavlja se sa proverom jednakosti karaktera p_0 niske p sa karakterom t_{k+1} teksta t. Broj upoređivanja karaktera je u najgorem slučaju reda mn (mada će se u praksi često mnogo brže naići na neslaganje niske i tekućeg segmenta), pa je složenost ovog algoritma $O(mn)$ u najgorem slučaju.

U opisanom direktnom algoritmu za traženje niske u tekstu nakon pomeranja niske za jedno mesto udesno ignorišu se sve informacije o prethodno poklopljenim karakterima. Stoga je moguće da se jedan isti karakter teksta više puta obrađuje tj. da se poredi sa različitim karakterima niske.

Primer 4.3.1

Razmotrimo problem traženja niske $p = aaaab$ u tekstu $t = aaaaaaaaaa$. Nakon uspešnog poređenja četiri karaktera a teksta t i niske p i nepoklapanja karaktera a teksta i karaktera b niske p, nisku p bismo pomerili udesno za jednu poziciju. Nakon toga bismo ponovo (uspešno) poredili tri karaktera a teksta t i niske p (iako smo već mogli da zaključimo da će se ti karakteri poklopiti) a zatim nakon nepoklapanja karaktera a sa karakterom b opet pomerili nisku p za jednu poziciju udesno itd.

Algoritam koji su nezavisno osmislili Donald Knut, Džejms Moris i Von Prat a koji je poznat pod nazivom *Knut-Moris-Pratov algoritam* (ili skraćeno *algoritam KMP*) zasniva se na ideji da se iskoriste informacije dobijene prethodnim poređenjima karaktera i da se nikada iznova ne porede karakteri teksta t koji su se prethodno uspešno poklopili sa niskom p. Na ovaj način faza traženja niske u tekstu je složenosti $O(n)$. Da bi ovo bilo izvodljivo, na početku algoritma vrši se pretprocesiranje niske p u cilju analize njene strukture. Faza pretprocesiranja je složenosti $O(m)$, te je ukupna složenost algoritma KMP $O(m + n)$. Algoritam KMP se često koristi kada se niska traži u dugačkom tekstu čija je azbuka male kardinalnosti, na primer kada se pretražuju molekuli DNK koji se opisuju nizom slova iz četvoroslovne azbuke A, C, G, T.

Pre opisa algoritma KMP uvedimo osnovnu terminologiju koja se u njemu koristi. Neka je $x = x_0 \dots x_{k-1}$ niska dužine k. Važi sledeće:

- niska y dužine l $(0 \leq l \leq k)$ je *prefiks* niske x ako je $y = x_0 \dots x_{l-1}$;
- niska y dužine l $(0 \leq l \leq k)$ je *sufiks* niske x ako je $y = x_{k-l} \dots x_{k-1}$;
- za prefiks (sufiks) y kažemo da je *pravi prefiks* (*pravi sufiks*) niske x ako je $y \neq x$, odnosno ako je njegova dužina strogo manja od dužine niske x;
- niska y dužine l je *prefiks-sufiks* niske x ako je $0 \leq l < k$, $y = x_0 \dots x_{l-1}$ i istovremeno $y = x_{k-l} \dots x_{k-1}$, odnosno ako je niska y istovremeno i pravi prefiks i pravi sufiks niske x.

Primer 4.3.2

Neka je $x = abacab$. Označimo sa ϵ praznu nisku. Pravi prefiksi niske x su $\epsilon, a, ab, aba, abac, abaca$, a pravi sufiksi $\epsilon, b, ab, cab, acab, bacab$. Dakle, prefiks-sufiksi niske x su ϵ (dužine 0) i ab (dužine 2).

Primer 4.3.3

Prefiks-sufiksi niske $x = aaaaa$ su $\epsilon, a, aa, aaa, aaaa$.

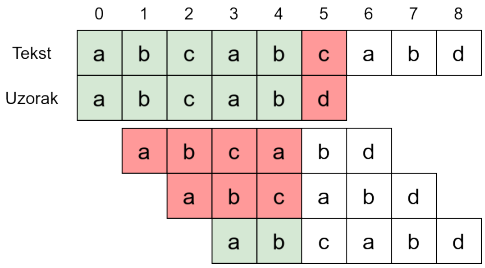
Prazna niska je uvek prefiks-sufiks niske x, osim ako je niska x prazna (prazna niska nema prefiks-sufiks).

U narednom primeru ilustrovaćemo kako pojam prefiks-sufiksa može pomoći prilikom izračunavanja vrednosti za koju treba pomeriti nisku p u odnosu na tekst t, kada dođe do nepoklapanja karaktera niske p i teksta t.

Primer 4.3.4

Na slici 4.10 prikazan je početak traženja niske $abcabd$ unutar teksta $abcabcabd$.

Karakteri $abcab$ na pozicijama od 0 do 4 u niski p i tekstu t su se poklopili, dok se karakteri na poziciji 5 razlikuju (u tekstu t se nalazi karakter c, a u niski p karakter d). Interesuje nas za koliko najmanje treba pomeriti nisku p udesno u odnosu na tekst t, tako da se ponovo neki (kraći) prefiks niske p poklopi sa tekstom t.



Slika 4.10: Primer pretrage niske.

- *Da bi se poklapanje dogodilo nakon pomeranja za jedno mesto udesno, potrebno je da se deo teksta $bcab$ poklopi sa delom niske $abca$, što nije slučaj. Primetimo da je $bcab$ sufiks, a $abca$ prefiks dela niske (prefiksa niske) $abcab$ za koji smo utvrdili da se poklapa sa tekstom.*
- *Da bi se poklapanje dogodilo nakon pomeranja za dva mesta udesno, potrebno je da se deo teksta cab poklopi sa delom niske abc, što nije slučaj. Opet možemo primetiti da je cab sufiks, a abc prefiks dela niske (prefiksa niske) $abcab$ za koji smo utvrdili da se poklapa sa tekstom.*
- *Da bi se poklapanje dogodilo nakon pomenranja za tri mesta udesno, potrebno je da se deo teksta ab poklopi sa delom niske ab, što ovaj put jeste slučaj. Niska ab je i sufiks i prefiks dela niske (prefiksa niske) $abcab$ i zapravo je najduži prefiks-sufiks tog poklopljenog dela niske.*

Dakle, nekoliko poslednjih karaktera poklopljenog prefiksa niske p treba da se poklope sa nekoliko prvih karaktera tog prefiksa niske p, te je nama, u stvari, potreban neki prefiks-sufiks poklopljenog prefiksa niske p. S obzirom na to da želimo da se pomerimo udesno što je manje moguće, mi u stvari tražimo najduži prefiks-sufiks poklopljenog prefiksa niske p.

U ovom primeru, poklopljeni prefiks niske p je $abcab$ i njegova dužina je 5. Njegov najduži prefiks-sufiks je ab i dužine je 2. Dakle, nisku pomeramo udesno u odnosu na tekst za $5 - 2 = 3$ karaktera.

Ako je niska $abcabd$ i tekst počinje sa $abcdxy$, poklopljeni prefiksa niske je abc i dužina mu je 3. Njegov najduži prefiks-sufiks je prazna niska ε i dužina mu je 0, te nisku pomeramo udesno za $3 - 0 = 3$ karaktera.

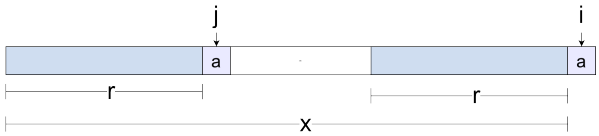
Pošto dužina poklopljenog dela niske p i teksta t zavisi i od samog teksta t, odnosno ne znamo unapred za koje prefikse niske p će nam biti potrebni najduži prefiksi-sufiksi, u fazi pretprocesiranja potrebno je odrediti dužinu najdužeg prefiks-sufiksa svakog prefiksa date niske p. Nakon toga, u fazi pretrage, na osnovu prefiksa niske p koji se poklopio sa tekstom t tj. dužine njegovog najdužeg prefiks-sufiksa, jednostavno se utvrđuje za koliko mesta treba pomeriti nisku p u odnosu na tekst t.

4.3.1 Pretprocesiranje niske koja se traži

Posvetimo se sada pitanju kako se za datu nisku $p = p_0 \dots p_{m-1}$ dužine m mogu efikasno odrediti dužine najdužih prefiks-sufiksa svih njenih prefiksa. U fazi pretprocesiranja niske p izračunaćemo vrednosti niza b dužine $m + 1$ (vrednosti $b_0, \dots, b_m$), tako da je b_i dužina najdužeg prefiks-sufiksa prefiksa $p_0 \dots p_{i-1}$ dužine i date niske p. Primenićemo induktivni pristup, tj. dužine najdužih prefiks-sufiksa ćemo izračunavati inkrementalno.

Bazu indukcije čini slučaj $i = 0$, tj. slučaj praznog prefiksa ϵ (prefiksa dužine 0) niske p. On nema prefiks-sufikse, a vrednost b_0 ćemo postaviti na -1.

Pretpostavimo da smo odredili vrednosti $b_0, \dots, b_i$ i razmotrimo kako se na osnovu tih vrednosti može odrediti vrednost b_{i+1}. Neka je $x = p_0 \dots p_{i-1}$. Vrednost b_{i+1} jednaka je dužini najdužeg prefiks-sufiksa niske $p_0 \dots p_{i-1}p_i = xp_i$. Svaki prefiks-sufiks niske xp_i dobija se tako što se neki prefiks-sufiks r (dužine $j < i$) niske x proširi karakterom $a = p_i$ (slika 4.11).



Slika 4.11: Proširivanje prefiks-sufiksa.

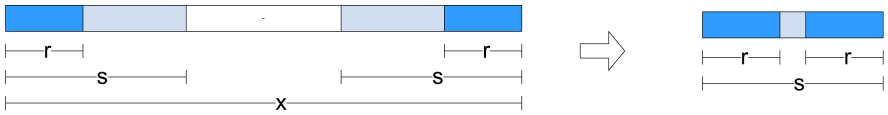
Da bi prefiks-sufiks niske xp_i bio što duži, potrebno je da i prefiks-sufiks r bude što duži. Dakle, do najdužeg prefiks-sufiksa niske xp_i možemo doći tako što analiziramo redom prefiks-sufikse niske x od najdužeg, ka najkraćem (a to je ε). Ako za neki prefiks-sufiks $r = p_0 \dots p_{j-1}$ dužine j važi $p_j = p_i$, tada je najduži prefiks-sufiks niske xp_i niska rp_i, pa je $b_{i+1} = j + 1$. Ako ni za jedan prefiks-sufiks r niske x to ne važi, tada je jedini prefiks-sufiks niske xp_i prazna niska ϵ, pa je $b_{i+1} = 0$.

Ostaje još pitanje kako nabrojati sve prefiks-sufikse niske $x = p_0 \dots p_{i-1}$. Dužina najdužeg od njih je poznata i jednaka je b_i, pa je najduži prefiks-sufiks niske x niska $p_0 \dots p_{b_i-1}$. Naredna lema nam daje način da poznajući jedan prefiks-sufiks niske x odredimo i sve ostale.

Lema 4.3.1

Neka su r i s prefiks-sufiksi niske x, pri čemu važi $|r| < |s|$. Tada je niska r prefiks-sufiks niske s.

Dokaz. Niska r je prefiks niske x, pa je i pravi prefiks niske s, jer je kraća od nje. Niska r je istovremeno sufiks niske x, te je stoga i pravi sufiks niske s. Stoga je r prefiks-sufiks niske s (slika 4.12).

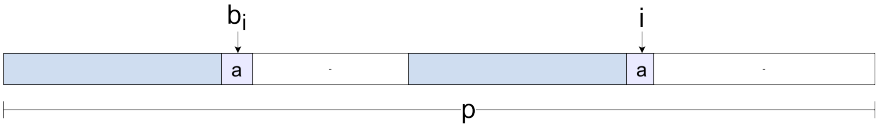


Slika 4.12: Prefiks-sufiksi r i s niske x.

□

Dakle, ako je niska r_1 najduži prefiks-sufiks niske x, sledeći po redu najduži prefiks-sufiks r_2 niske x se dobija kao najduži prefiks-sufiks niske r_1. Naredni po redu najduži prefiks-sufiks r_3 niske x je najduži prefiks-sufiks niske r_2 i tako dalje.

Pošto je b_i dužina najdužeg prefiks-sufiksa niske x, karakteri $p_0, p_1, \dots, p_{b_i-1}$ i $p_{i-b_i}, \dots, p_{i-1}$ niske se poklapaju i potrebno je proveriti da li važi $p_{b_i} = p_i$ (slika 4.13).



Slika 4.13: Prefiks dužine i niske sa prefiks-sufiksom dužine b_i.

Ako se ne poklapaju, razmatra se prvi naredni kraći prefiks-sufiks niske x. On će biti jednak najdužem prefiks-sufiksu najdužeg prefiks-sufiksa prefiksa niske dužine i. Dakle, on će biti jednak prefiksu niske x dužine b_{b_i}, te je potrebno uporediti karaktere $p_{b_{b_i}}$ i p_i. Ako se ti karakteri ne poklope, postupak se nastavlja na isti način.

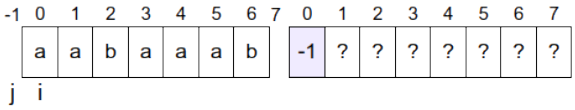
Dakle, prilikom računanja vrednosti b_{i+1}, promenljiva j će u petlji uzimati redom vrednosti $b_i, b_{b_i}, b_{b_{b_i}} \dots$ Prefiks-sufiks dužine j može se proširiti karakterom p_i ako je $p_j = p_i$. Ako to nije slučaj, naredni najduži prefiks-sufiks niske x se određuje postavljanjem $j = b_j$. Petlja se prekida ako se nijedan prefiks-sufiks ne može proširiti. Zahvaljući tome što smo postavili $b_0 = -1$, kada se to desi, petlja se prekida kada je

$j = -1$, jer uslov $j \geq 0$ nije ispunjen. Na kraju izvršavanja petlje vrednost promenljive j uvećana za 1 sadržaće dužinu najdužeg prefiks-sufiksa niske $x = p_0 \dots p_i$ i nju treba smestiti na poziciju b_{i+1} (ovo važi i u slučaju kada je došlo do poklapanja $p_j = p_i$, ali, opet zahvaljujući odluci da je $b_0 = -1$, i kada je petlja prekinuta jer je $j = -1$, što znači da je prazna reč najduži prefiks-sufiks).

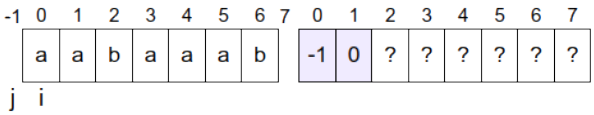
```
vector<int> kmpIzracunajPrefiksSufikse(const string &p) {
  int m = p.size() + 1;
  vector<int> b(m);
  int i = 0;
  int j = -1;
  // prazna niska nema prefiks-sufikse
  b[i] = j;
  // za prefiks duzine i polazne niske racunamo najduzi prefiks-sufiks
  while (i < m) {
    // proveravamo da li se najduzi prefiks-sufiks prefiksa niske duzine i
    // moze prosiriti: to vazi ako je p[i] = p[j]
    while (j >= 0 && p[i] != p[j])
      j = b[j];
    b[++i] = ++j;
  }
  return b;
}
```

Primer 4.3.5

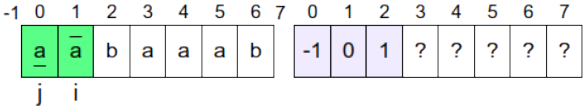
Prikažimo pretprocesiranje tj. izračunavanje najdužih prefiks-sufiksa niske $aabaaab$. Nakon inicijalizacije promenljivih, važi $j = -1$, $i = 0$, $b_0 = -1$.



- *Sada određujemo vrednost b_1 tj. dužinu najdužeg prefiks-sufiksa niske a (tj. prefiksa niske p dužine 1, koji se završava na poziciji $i = 0$). Njen jedini prefiks-sufiks je prazna niska, pa je $b_1 = 0$ (u unutrašnju petlju se ne ulazi jer je $j = -1 < 0$). Ovo je ilustrovano na narednoj slici.*

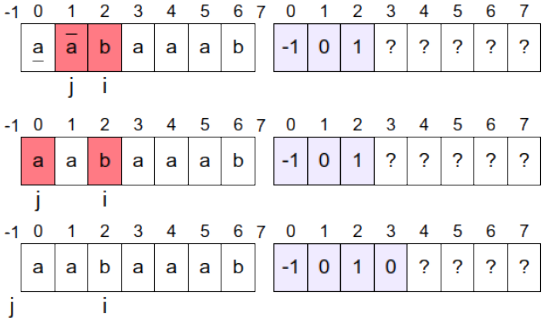


Nakon toga se uvećavaju vrednosti promenljivih i i j i postaju $i = 1$ i $j = 0$ (tada se vrši dodela `b[1] = 0`) i prelazi se na određivanje vrednosti b_2 tj. dužine najdužeg prefiks-sufiksa niske aa (tj. prefiksa niske p dužine 2, koji se završava na poziciji $i = 1$). Najduži prefiks-sufiks niske a je prazna niska, a pošto je $p_j = p_0 = a = p_1 = p_i$, on se može produžiti karakterom a, pa je a najduži prefiks-sufiks niske aa i $b_2 = 1$. Ovo je ilustrovano na narednoj slici.

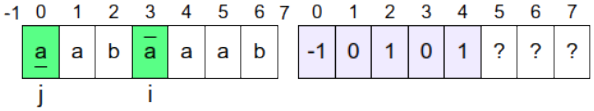


Nakon toga se uvećavaju vrednosti promenljivih i i j i postaju $i = 2$ i $j = 1$ (tada se vrši dodela `b[2]=1`) i prelazi se na određivanje vrednosti b_3 tj. dužine najdužeg prefiks-sufiksa niske aab (tj. prefiksa niske p dužine 3, koji se završava na poziciji $i = 2$). Najduži prefiks-sufiks niske aa je a, ali pošto je $p_j = p_1 = a \neq b = p_2 = p_i$, on se ne može produžiti. Zato analiziramo kraći prefiks-sufiks niske aa, a to je prazna

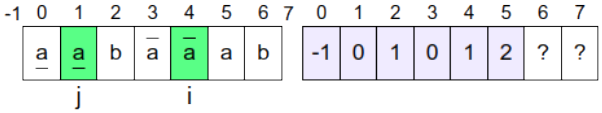
reč (postavljajući vrednost j na $b_j = b_1 = 0$). Pošto je $p_j = p_0 = a \neq p_2 = p_i$, ni on se ne može produžiti. Postavljajući j na vrednost $b_j = b_0 = -1$, prekidamo unutrašnju petlju i zaključujemo da je najduži prefiks-sufiks niske aab prazna niska i $b_3 = 0$. Ovo je ilustrovano na narednoj slici.



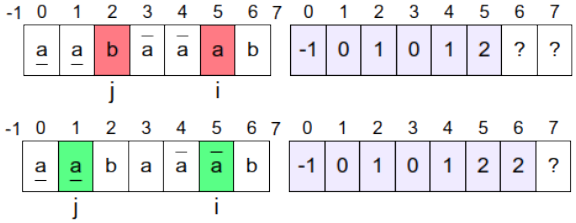
Nakon toga se uvećavaju vrednosti promenljivih i i j i postaju $i = 3$ i $j = 0$ (tada se vrši dodela `b[3]=0`) i prelazi se na određivanje vrednosti b_4 tj. dužine najdužeg prefiks-sufiksa niske $aaba$ (tj. prefiksa niske p dužine 4, koji se završava na poziciji $i = 3$). Najduži prefiks-sufiks niske aab je prazna niska, a pošto je $p_j = p_0 = a = p_3 = p_i$, on se može produžiti karakterom a, pa je a najduži prefiks-sufiks niske $aaba$ i $b_4 = 1$. Ovo je ilustrovano na narednoj slici.



Nakon toga se uvećavaju vrednosti promenljivih i i j i postaju $i = 4$ i $j = 1$ (tada se vrši dodela `b[4]=1`) i prelazi se na određivanje vrednosti b_5 tj. dužine najdužeg prefiks-sufiksa niske $aabaa$ (tj. prefiksa niske p dužine 5, koji se završava na poziciji $i = 4$). Najduži prefiks-sufiks niske $aaba$ je niska a, a pošto je $p_j = p_1 = a = p_4 = p_i$, on se može produžiti karakterom a, pa je aa najduži prefiks-sufiks niske $aabaa$ i $b_5 = 2$. Ovo je ilustrovano na narednoj slici.

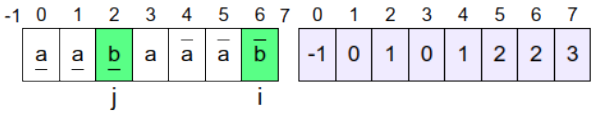


Nakon toga se uvećavaju vrednosti promenljivih i i j i postaju $i = 5$ i $j = 2$ (tada se vrši dodela `b[5]=2`) i prelazi se na određivanje vrednosti b_6 tj. dužine najdužeg prefiks-sufiksa niske $aabaaa$ (tj. prefiksa niske p dužine 6, koji se završava na poziciji $i = 5$). Najduži prefiks-sufiks niske $aabaa$ je niska aa, ali pošto je $p_j = p_2 = b \neq a = p_5 = p_i$, on se ne može produžiti. Zato analiziramo kraći prefiks-sufiks niske $aabaa$, a to je a (postavljajući vrednost j na $b_j = b_2 = 1$). Pošto je sada $p_j = p_1 = a = p_5 = p_i$, ovaj prefiks-sufiks se može produžiti karakterom a, pa je aa najduži prefiks-sufiks niske $aabaaa$ i $b_6 = 2$. Ovo je ilustrovano na narednoj slici.



Nakon toga se uvećavaju vrednosti promenljivih i i j i postaju $i = 6$ i $j = 2$ (tada se vrši dodela `b[6]=2`) i prelazi se na određivanje vrednosti b_7 tj. dužine najdužeg prefiks-sufiksa niske $aabaaab$ (tj. prefiksa niske

p dužine 7, koji se završava na poziciji $i = 6$). Najduži prefiks-sufiks niske $aabaaa$ je niska aa, a pošto je $p_j = p_2 = b = p_6 = p_i$, on se može produžiti karakterom b, pa je aab najduži prefiks-sufiks niske $aabaaab$ i $b_7 = 3$. Ovo je ilustrovano na narednoj slici.



Ovim su određeni najduži prefiks-sufiksi svih prefiksa niske p, pa se algoritam završava (jer nakon uvećavanja vrednosti promenljivih na $j = 3$ i $i = 7$, vrednost i dostiže dužinu niske $m = 7$).

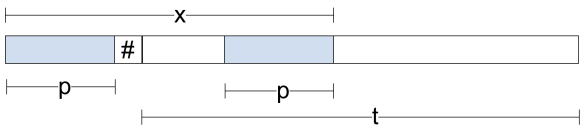
4.3.2 Pretraživanje teksta

Osnovna varijanta pretrage teksta se može dobiti prilagođavanjem algoritma grube sile. Spoljašnja petlja obilazi pozicije i u tekstu sa kojih se počinje pretraga niske. U unutrašnjoj petlji se prolazi kroz karaktere niske (od početka) i karaktere teksta (od pozicije i) sve dok se ne dođe do kraja niske ili do različitog karaktera. Ako se dođe do kraja niske, pronađeno je poklapanje. Ako nije, algoritam grube sile uvećava poziciju i za 1, dok se kod algoritma KMP pozicija i uvećava za $j - b_j$, gde je j pozicija u niski na kojoj se naišlo na nepoklapajući karakter (tj. dužina uspešno poklopljenog prefiksa niske), a b_j dužina najdužeg prefiks-sufiksa tog dela niske.

```
vector<int> traziKMP(const string &t, const string &p, vector<int>& b) {
   vector<int> pozicije; // pozicije poklapanja u tekstu
   // proveravamo pojavljivanje nniske krenuvsi od pozicije i u tekstu
   int i = 0;
   while (i + niska.size() <= tekst.size()) {
     // analiziramo jedan po jedan karakter niske
     int j;
     for (j = 0; j < niska.size(); j++)
       if (tekst[i + j] != niska[j])
         break;
     // ako smo stigli do kraja niske, ona je pronadjena na poziciji i u tekstu
     if (j == niska.size())
       pozicije.push_back(i);
     // pomeramo se udesno na osnovu algoritma KMP
     i += j - b[j];
   }
   return pozicije;
}
```

Moguće su i drugačije implementacije.

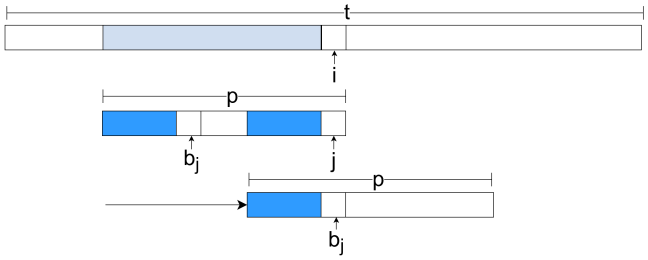
Slično kao kod z-algoritma, pronalazak svih pojavljivanja niske p u tekstu t se može svesti na pretprocesiranje niske $p\#t$, gde je sa $\#$ označen specijalni karakter koji se ne javlja ni u tekstu t ni u niski p. Tada svakom pojavljivanju niske p u tekstu t odgovara po jedan prefiks-sufiks dužine m (slika 4.14).



Slika 4.14: Prefiks-sufiks dužine $m = |p|$ prefiksa x niske $p\#t$ odgovara pojavljivanju prefiksa p u tekstu t.

Alternativno, drugu fazu, tj. algoritam kojim se traži niska p u tekstu t na osnovu poznatih dužina prefiks-sufiksa dobijenih u fazi pretprocesiranja niske, možemo implementirati u vidu zasebne funkcije. Taj algoritam je vrlo sličan algoritmu pretprocesiranja niske, jedino što se porede karakteri t_i (karakter na poziciji i u tekstu t) i p_j (karakter na poziciji j u niski p).

Kada se u unutrašnjoj `while` petlji naiđe na nepoklapanje, razmatra se najduži prefiks-sufiks poklapajućeg prefiksa dužine j niske (slika 4.15) i niska se pomera tako što se vrednost j zameni njenom dužinom b_j. To se radi sve dok se karakteri p_j i t_i ne poklope ili dok se zaključi da kraći prefiks-sufiks prefiksa niske ne postoji ($j = -1$). Nakon toga imamo novi poklapajući prefiks niske i nastavljamo sa spoljašnjom `while` petljom. Ukoliko smo naišli na poklapanje svih m karaktera niske (kada važi $j = m$), evidentiramo da smo pronašli nisku u tekstu počev od pozicije $i - j$. Nakon toga, niska se pomera udesno u odnosu na tekst na osnovu svog najdužeg prefiks-sufiksa.



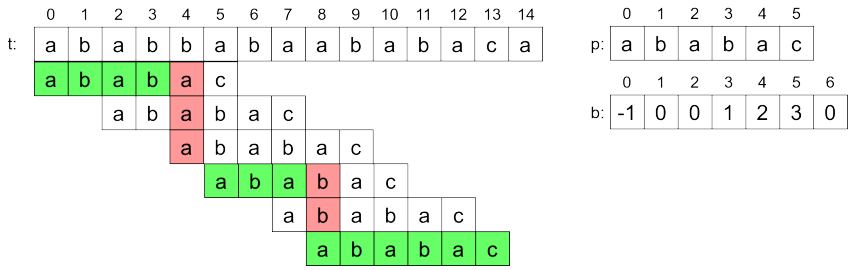
Slika 4.15: Kada se naiđe na nepoklapanje karaktera p_j (karaktera na poziciji j u niski) i karaktera t_i (karaktera na poziciji i u tekstu), prelazi se na poređenje karaktera p_{b_j} (karaktera na poziciji b_j u niski) i karaktera t_i, što odgovara pomeranju niske udesno za $j - b_j$.

Implementacija pretrage u jeziku C++ može biti sledeća:

```
vector<int> traziKMP(const string &t, const string &p, vector<int>& b) {
   vector<int> pozicije; // pozicije poklapanja u tekstu
   int n = t.size();
   int m = p.size();
   int i = 0;
   int j = 0;
   // proveravamo tekuci karakter teksta sve dok postoji mogućnost da se
   // do kraja teksta pronadje niska
   while (i + m - j <= n) {
     // proveravamo da li se najduzi prefiks-sufiks
     // prefiksa niske p koji se zavrsava na prethodnoj poziciji
     // moze prosiriti: to vazi ako je t[i] = p[j]
     while (j >= 0 && t[i] != p[j])
       j = b[j];
     i++; j++;
     // ako smo poklopili svih m karaktera niske pamtimo poziciju poklapanja
     if (j == m) {
       pozicije.push_back(i - m);
       j = b[m];
     }
   }
   return pozicije;
}
```

Primer 4.3.6

Razmotrimo koja se poređenja izvršavaju u algoritmu KMP, prilikom traženja niske $ababac$ u tekstu $abababaababaca$. Uspešna poređenja karaktera su priikazana zelenom, a poređenja kod kojih se nije naišlo na poklapanje, crvenom bojom (slika 4.16).



Slika 4.16: Druga faza KMP-algoritma.

Na početku se uspešno poklapaju karakteri $abab$ nakon čega se nailazi na razliku između karaktera $t_4 = b$ i $p_4 = a$. Pošto je ab najduži prefiks-sufiks poklopljenog dela $abab$ i njegova dužina je 2 (tj. pošto je $b_4 = 2$), vrednost promenljive j se postavlja na 2 i porede se karakteri $t_4 = b$ i $p_2 = a$. Ponovo nema poklapanja. Pošto je sada poklopljena niska ab, a njen najduži prefiks-sufiks je prazna reč (tj. pošto je $b_2 = 0$), prelazi se na poređenje karaktera $t_4 = b$ i $t_0 = a$. I oni su različiti, važi $b_0 = -1$, što znači da se i uvećava, j postavlja na nulu i prelazi se na karakter $t_5 = a$. Uspešno se poklapa prefiks aba, nakon čega se nailazi na razliku $t_8 = a$, a $p_3 = b$. Pošto je a najduži prefiks-sufiks poklopljenog dela aba (tj. pošto je $b_3 = 1$), promenljiva j se postavlja na 1 i porede se karakteri $t_8 = a$ i $p_1 = b$. Pošto su i oni različiti, a pošto je prazna reč najduži prefiks-sufiks poklopljenog dela a (tj. pošto je $b_1 = 0$), promenljiva j se postavlja na 0 i porede se karakteri $t_8 = a$ i $p_0 = a$. Ti karakteri su jednaki, pa se sa poređenjem nastavlja dalje i uspešno se pronalazi podniska. Pošto je prazna reč jedini prefiks sufiks cele niske (tj. pošto je $b_6 = 0$), naredni karakter bi trebalo porediti sa karakterom 0 niske, međutim, jasno je da se došlo do kraja teksta i da neće biti pronađena druga pojavljivanja niske.

Koja je složenost funkcije za pretprocesiranje niske p dužine $|p| = m$, prikazane u poglavlju 4.3.1? Unutrašnja petlja smanjuje vrednost promenljive j bar za 1, jer je $b_j < j$. Petlja se završava najkasnije kada vrednost j postane -1 (a na početku je j jednako -1), te se stoga može smanjiti najviše onoliko puta koliko je prethodno bila povećana naredbom `j++`. S obzirom na to da se naredba `j++` izvršava u spoljašnjoj petlji tačno m puta, ukupan broj izvršavanja unutrašnje `while` petlje je ograničen sa m, te je ukupna složenost pretprocesiranja niske $O(m)$. Potpuno analogno se može zaključiti da je složenost algoritma za traženje niske p u tekstu t dužine $|t| = n$ jednaka $O(n)$, te je ukupna složenost algoritma KMP $O(m+n) = O(|p|+|t|)$.

U primeru 4.3.6 smo imali finu ilustraciju složenosti faze pretrage, s obzirom na to da poređenja (uspešna i neuspešna) formiraju "stepenice" koje u najgorem slučaju mogu biti jednako visoke i duge, te je u najgorem slučaju potrebno $2n$ poređenja prilikom traženja niske u tekstu.

4.3.3 Ispitivanje periodičnosti niske

Pronalaženje najdužih prefiks-sufiksa niski ima i druge primene. Ilustrujmo jednu od njih.

Reč w je *periodična* ako postoji neprazna reč $p = p_1p_2$ i prirodan broj $n \geq 2$, tako da je $w = p^np_1$. Na primer, reč $abacabacabacab$ je periodična jer se u njoj ponavlja reč $abac$, pri čemu se poslednje ponavljanje ne završava celo već se zaustavlja sa ab, tj. reč je jednaka $(abac)^3ab$.

Ispitivanje periodičnosti niske igra važnu ulogu u različitim domenima. Nekada se koristi kao mera samosličnosti u primenama koje se tiču obrade teksta, analize podataka, računarske biologije i sl.

Problem

Napisati algoritam koji proverava da li je reč w periodična.

Problem možemo rešiti grubom silom, tako što ćemo za svaku vrednost d takvu da je $2d \leq |w|$ proveriti da li je reč periodična, pri čemu je period prefiks reči w dužine d. Jednostavno se dokazuje da je reč w periodična sa periodom p čija je dužina d ako i samo ako za svako i za koje je $0 \leq i$ i $i + d < |w|$ važi $w_i = w_{i+d}$. Problem se onda rešava sa dve ugnežđene linearne pretrage – u spoljnoj proveravamo sve potencijalne vrednosti dužine d, a u unutrašnjoj proveravamo da li postoji vrednost i takva da je $w_i \neq w_{i+d}$. Ako u unutrašnjoj petlji utvrdimo da takvo i ne postoji, tada je niska periodična. Ako pronađemo takvo i, možemo prekinuti unutrašnju petlju (reč nije periodična sa periodom dužine d) i preći na sledeće d (za jedan veće). Ako takvo d ne postoji, tada možemo konstatovati da reč nije periodična.

```
// provera da li je niska periodicna
bool periodicna(const string &str){

  int m = str.size();
  // proveravamo za svaku mogucu duzinu periode
  for (int d = 1; 2 * d <= m; d++) {
    bool greska = false;
    for (int i = 0; i + d < m; i++)
      // ako naidjemo na nepoklapanje, znamo da prefiks duzine d
      // nije perioda niske
      if (str[i] != str[i + d]) {
         greska = true;
         break;
      }
    // ako smo uspesno poklopili sve karaktere niske
    // pronasli smo periodu
    if (!greska) {
      return true;
    }
  }
  return false;
}

int main() {
  string rec;
  cin >> rec;
  cout << (periodicna(rec) ? "Rec je periodicna"
       : "Rec nije periodicna") << endl;
  return 0;
}
```

Složenost najgoreg slučaja ovog algoritma je kvadratna. Zaista, unutrašnja linearna pretraga može u najgorem slučaju zahtevati $O(|w|)$ iteracija, i ona se ponavlja $O(|w|)$ puta. Ipak, ako je niska nasumična, realno je očekivati da će se za većinu vrednosti d veoma brzo ustanovljavati da je $w_i \neq w_{i+d}$, pa program može raditi dosta brže od najgoreg slučaja.

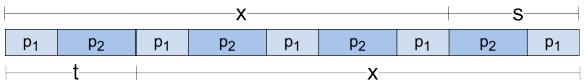
Efikasnije rešenje se zasniva na narednoj lemi:

Lema 4.3.2 **[Karakterizacija periodične niske]**

Niska w je periodična ako i samo ako ima prefiks-sufiks x čija je dužina najmanje polovina niske, tj. ako postoje neprazni x, s i t takvi da je $w = xs = tx$ i $2|x| \geq |w|$.

Na primer, ako je niska $abacabacaba$, tada je traženi prefiks-sufiks x jednak $abacaba$, sufiks s jednak je $caba$, dok je prefiks t jednak $abac$.

Dokaz. Pretpostavimo da je niska w periodična i pokažimo da ona ima prefiks-sufiks dužine bar polovine niske. Iz uslova da je niska w periodična sledi da postoji neprazna reč $p = p_1p_2$ tako da je $w = p^np_1$, za neko $n \geq 2$. Tada je $t = p_1p_2 = p$, $x = p^{n-1}p_1$, dok je $s = p_2p_1$ (slika 4.17). Zaista, važi $xs = p^{n-1}p_1 \cdot p_2p_1 = p^np_1 = w$ i $tx = p \cdot p^{n-1}p_1 = p^np_1 = w$.



Slika 4.17: x je prefiks-sufiks periodične niske koji se prostire preko njene sredine.

Pokažimo i da je $2|x| \geq |w|$. Važi $|x| = (n-1)|p| + |p_1|$, a iz $n \geq 2$ sledi $(n-1) \cdot |p| \geq |p|$, pa je $|x| = (n-1) \cdot |p| + |p_1| \geq |p| + |p_1| = |t| + |p_1| \geq |t|$, te je $2 \cdot |x| \geq |x| + |t| = |w|$.

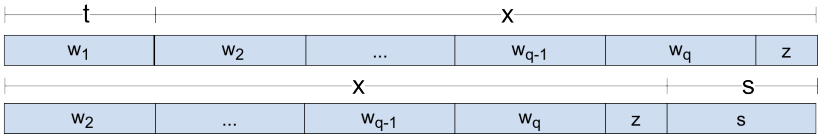
Dokažimo suprotnu implikaciju.

Pretpostavimo da niska w ima prefiks-sufiks x dužine bar $|w|/2$ i da važi $w = xs = tx$. Pokažimo da je niska w periodična.

Jasno je da su dužine niski s i t jednake. Iz uslova $|x| \geq |w|/2$ sledi da je $|s| = |t| \leq |x|$.

Neka je $|w| = q|t| + r, 0 \leq r < |t|$. Tada je $q \geq 2$ (u protivnom bi važilo $|w| \leq |t| + r < 2|t|$, što zajedno sa pretpostavkom $|w| \leq 2|x|$ daje $2|w| < 2|x| + 2|t| \leq 2|w|$ i dobili bismo kontradikciju).

Neka je $w = w_1w_2 \ldots w_qz$, gde je $|w_1| = |w_2| = \ldots = |w_q| = |t|$ i $|z| = r$. Iz $w = tx$ sledi $w_1 = t$ i $w_2 \ldots w_qz = x$. Zato je i $w = xs = w_2 \ldots w_qzs$, pa važi redom $w_2 = w_1, w_3 = w_2, \ldots, w_{q-1} = w_q$ (slika 4.18). Iz uslova $zs = w_qz$ i $|z| = r < q = |w_q|$, sledi da je z prefiks niske $w_q = t$. Odavde dobijamo $w = t^qz$, pri čemu je z prefiks niske t, te je reč w periodična.



Slika 4.18: Određivanje periode kada je poznat prefiks-sufiks koji se prostire preko sredine niske.

□

Dakle, rešenje ovog problema se svodi na pronalaženje dužine d najdužeg prefiks-sufiksa reči w i proveru da li važi $2d \geq |w|$. To možemo uraditi već prikazanom funkcijom `kmpIzracunajPrefiksSufikse`.

```
int main() {
  string w;
  cin >> w;
  int m = w.size();

  vector<int> b(m+1);
  // racunamo duzinu najduzeg prefiks-sufiksa niske w
  kmpIzracunajPrefiksSufikse(w, b);
```

```
  if (2 * b[m] >= m)
    cout << "Niska je periodicna" << endl;
  else
    cout << "Niska nije periodicna" << endl;

  return 0;
}
```

Složenost ovog algoritma jednaka je složenosti algoritma za pretprocesiranje u KMP algoritmu, odnosno jednaka je $O(|w|)$.

4.4 Najduži palindromski segment - Manačerov algoritam

Problem

Data je niska s koja sadrži samo mala slova. Odrediti najduži segment niske s koji je palindrom. Ako ima više najdužih segmenata koji su palindromi, prikazati segment čiji početak ima najmanji indeks.

Primer 4.4.1

Za niske `ananas`*,* `najjaci` *i* `list` *najduži segmenti koji su palindromi su redom* `anana`*,* `ajja` *i* `l`*.*

4.4.1 Provera svih segmenata

Problem je moguće rešiti analizom svih segmenata, proverom da li je tekući segment palindrom i određivanjem najdužeg pronađenog palindroma. Pošto je provera da li je niska dužine n palindrom složenosti $O(n)$, a segmenata niske dužine n ukupno ima $O(n^2)$, složenost ovog pristupa je $O(n^3)$.

```
// provera da li je segment s[i, j] palindrom
bool palindrom(const string &s, int i, int j){
  while (i < j && s[i] == s[j]) {
    i++;
    j--;
  }
  // ako je i < j onda smo nasli s[i] != s[j]
  // te segment nije palindrom, inace jeste
  return i >= j;
}

int main() {
  string s;
  cin >> s;
  int n = s.size();

  int maxDuzina = 0, maxPocetak = 0;
  // za sve moguce segmente date niske
  for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
      // ako je u pitanju palindrom
      if (palindrom(s, i, j)) {
        int duzina = j - i + 1;
```

```
        // proveravamo da li je duzi od tekuceg maksimuma
        if (duzina > maxDuzina) {
          maxDuzina = duzina;
          maxPocetak = i;
        }
      }

  // stampamo najduzi palindrom
  cout << s.substr(maxPocetak, maxDuzina) << endl;
  return 0;
}
```

4.4.2 Provera svih segmenata redom prema opadajućim dužinama

Implementacija se može malo pojednostaviti i dugački palindromi se mogu pronaći brže, ako se primeti da segmente možemo analizirati redom počev od najdužeg segmenta pa unazad do segmenta dužine 1. Primetimo da će sa ovim redosledom obilaska prvi segment za koji se utvrdi da je palindrom upravo biti najduži palindrom koji tražimo. Ako segmente iste dužine razmatramo u rastućem redosledu indeksa leve granice, u slučaju kada postoji više palindroma iste najveće dužine, kao prvi će biti pronađen onaj koji ima najmanju vrednost indeksa leve granice, kao što je i traženo.

```
int main() {
  string s;
  cin >> s;
  // duzina niske s
  int n = s.size();
  // potrebno za prekid dvostruke petlje
  bool nasli = false;

  // proveravamo sve duzine d redom, od najvece do najmanje
  // dok ne naidjemo na prvi palindrom
  for (int d = n; d >= 1 && !nasli; d--) {
    // proveravamo rec odredjenu indeksima [p, p + d - 1]
    for (int p = 0; p + d - 1 < n && !nasli; p++) {
      // ako smo naisli na palindrom
      if (palindrom(s, p, p + d - 1)) {
        // ispisujemo ga
        cout << s.substr(p, d) << endl;
        // prekidamo dvostruku petlju
        nasli = true;
      }
    }
  }
  return 0;
}
```

Primetimo da izlaz iz ugnežđenih petlji nije moguće ostvariti naredbom `break` (na taj način bi se izašlo iz unutrašnje, ali ne i spoljašnje petlje), već izlaz moramo realizovati pomoću pomoćne logičke promenljive. Složenost najgoreg slučaja ovog pristupa je i ovde $O(n^3)$.

4.4.3 Provera centara

Palindromi poseduju određeno svojstvo inkrementalnosti koje nam može pomoći da pronađemo efikasniji algoritam. Naime, ako je poznat centar palindroma (to može biti bilo neko slovo, bilo pozicija tačno između dva susedna slova) i ako znamo da se k slova oko tog centra slikaju kao u ogledalu i na taj način grade palindrom, onda za proveru da li se $k+1$ slova oko tog centra slikaju kao u ogledalu ne treba proveravati sve iz početka, već je dovoljno samo proveriti da li su dva slova na spoljnim pozicijama ($(k+1)$-vo slovo levo tj. desno od centra) jednaka. Zato efikasnije rešenje dobijamo ako:

- za svako slovo reči odredimo najduži palindrom neparne dužine takav da mu je izabrano slovo centar i
- za svaku poziciju između dva susedna slova odredimo najduži palindrom parne dužine kome je ta pozicija centar.

Da bismo odredili najduži palindrom sa centrom u slovu s_i, širimo palindrom s_i u desno i u levo za k slova dok se nalazimo unutar reči ($i-k \geq 0$ i $i+k < n$) i dok su odgovarajuća slova jednaka ($s_{i-k} = s_{i+k}$). U trenutku kada se to prvi put naruši, pronađen je najduži palindrom sa centrom u slovu s_i (ako se izađe iz reči dalje proširivanje nije moguće, a ako se pronađe različit par slova dalja proširivanja ne mogu više da daju palindrom).

Određivanje najdužeg palindroma sa centrom između slova s_i i s_{i+1} vršimo na analogan način: dok smo unutar reči ($i-k \geq 0$ i $i+k+1 < n$) širimo palindrom sve dok važi $s_{i-k} = s_{i+k+1}$.

Za svaku poziciju koja može biti centar palindroma (a njih ima n za slova niske i $n-1$ za pozicije između dva slova niske, tj. ukupno $2n-1$, odnosno $O(n)$) nalazimo najduži palindrom sa centrom u njoj šireći tekući palindrom nalevo i nadesno. Globalno najduži palindrom nalazimo kao najduži od dobijenih palindroma.

Širenje za fiksirani centar palindroma se obavlja u jednom prolasku i zahteva vreme $O(n)$, pa je ukupna složenost ovog algoritma $O(n^2)$.

```
int main() {
  string s;
  cin >> s;
  // duzina ucitane reci
  int n = s.size();
  // duzina i pocetak najduzeg palindroma
  int maxDuzina = 0, maxPocetak;

  // prolazimo kroz sva slova reci
  for (int i = 0; i < n; i++) {
    int duzina, pocetak;

    // nalazenje najduzeg palindroma neparne duzine ciji je centar
    // slovo s[i]
    int k = 1;
    while (i - k >= 0 && i + k < n && s[i - k] == s[i + k])
      k++;
    // duzina i pocetak maksimalnog palindroma
    duzina = 2 * k - 1;
    pocetak = i - k + 1;
    // azuriramo maksimum ako je to potrebno
    if (duzina > maxDuzina) {
      maxDuzina = duzina;
      maxPocetak = pocetak;
```

```
    }

    // nalazenje najduzeg palindroma parne duzine ciji je centar
    // izmedju slova s[i] i s[i+1]
    k = 0;
    while (i - k >= 0 && i + k + 1 < n && s[i - k] == s[i + k + 1])
      k++;
    // duzina i pocetak maksimalnog palindroma
    duzina = 2 * k;
    pocetak = i - k + 1;
    // azuriramo maksimum ako je to potrebno
    if (duzina > maxDuzina) {
      maxDuzina = duzina;
      maxPocetak = pocetak;
    }
  }
  // izdvajamo i ispisujemo odgovarajuci palindrom
  cout << s.substr(maxPocetak, maxDuzina) << endl;
  return 0;
}
```

4.4.4 Eksplicitna dopuna reči i pozicija

Postoji nekoliko tehnika koje mogu da uproste implementaciju prethodnog algoritma, objedinjavajući slučajeve palindroma parne i neparne dužine. Zamislimo da se pre prvog, nakon poslednjeg i između svaka dva susedna slova reči postavi specijalni karakter `|`. Na primer, reč `aabcbab` dopunjavamo do reči `|a|a|b|c|b|a|b|`. Na taj način postiže se da su svi centri palindroma karakteri proširene reči i dovoljno je analizirati samo palindrome neparne dužine u proširenoj reči. Ovo dopunjavanje polazne reči je moguće fizički realizovati tako što se u programu eksplicitno kreira dopunjena niska. Ovo uprošćava implementaciju po cenu nešto sporijeg izvršavanja (doduše ne asimptotski) i dodatnog zauzeća memorije.

Napomenimo još i da se i provera pripadnosti indeksa tj. pozicija opsegu reči može eliminisati ako se polazna reč proširi dodatnim specijalnim karakterima na početku i na kraju: ti karakteri moraju biti međusobno različiti i različiti od ostalih karaktera u niski. U tu svrhu mogu da se iskoriste `^` i `$`, jer se ti karakteri koriste za označavanje početka i kraja u regularnim izrazima.

Dužinu palindroma razmatraćemo u odnosu na polaznu (a ne dopunjenu) reč.

```
// da bismo uniformno posmatrali palindrome i parne i neparne duzine,
// prosirujemo nisku dodajuci ^ i $ oko njega i umecuci | izmedju
// svih slova, na primer abc -> ^|a|b|c|$
string dopuni(const string &s) {
  string rez = "^";
  for (int i = 0; i < s.size(); i++)
    rez += "|" + s.substr(i, 1);
  rez += "|$";
  return rez;
}

int main() {
  string s;
  cin >> s;
```

```
  string t = dopuni(s);

  // dovoljno je pronaci najveci palindrom neparne duzine u prosirenoj reci
  int maxDuzina = 0, maxCentar;
  // proveravamo sve pozicije u dopunjenoj reci
  for (int i = 1; i < t.size() - 1; i++) {
    // prosirujemo palindrom sa centrom na poziciji i dokle god je to
    // moguce
    int d = 0;
    while (t[i - d - 1] == t[i + d + 1])
      d++;

    // azuriramo maksimum ako je potrebno
    if (d > maxDuzina) {
      maxDuzina = d;
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  cout << s.substr(maxPocetak, maxDuzina) << endl;
}
```

4.4.5 Implicitna dopuna reči i pozicija

Da bismo olakšali naredno izlaganje, indekse u dopunjenoj reči (bez dodatih oznaka početka i kraja reči) nazivaćemo pozicije, a u originalnoj reči samo indeksi. Na primer,

```
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14   - pozicije
|  a  |  a  |  b  |  c  |  b  |  a  |  b  |
   0     1     2     3     4     5     6       - indeksi
```

Primetimo nekoliko činjenica. Ako je polazna reč dužine n, ukupno imamo $N = 2n + 1$ pozicija u dopunjenoj reči. Slova polazne reči se nalaze na neparnim pozicijama, dok se na parnim pozicijama nalazi specijalni karakter `|`. Slovo sa indeksom k se nalazi na poziciji $p = 2k + 1$, što znači da se na neparnoj poziciji p nalazi slovo polazne reči sa indeksom $k = \lfloor \frac{p}{2} \rfloor$.

Za svaku poziciju i, $0 \leq i < N$, dužinu palindroma sa centrom na toj poziciji možemo odrediti na isti način, bez obzira na to da li je na toj poziciji slovo ili specijalni karakter `|`. Razmatraćemo dužinu palindroma u polaznoj reči (a ne dopunjenoj) i ona je jednaka broju karaktera sa leve strane date pozicije u dopunjenoj reči koji su jednaki odgovarajućim karakterima sa desne strane te pozicije u dopunjenoj reči. Na primer, u prethodnom primeru za poziciju 7 to je 5, jer je palindrom `abcba` dužine 5, što odgovara tome da pet karaktera `|a|b|` sa leve strane karaktera `c` u dopunjenoj reči odgovaraju karakterima `|b|a|` sa desne strane karaktera `c`. Slično važi i za parne pozicije (na primer 2).

Na početku, dužinu palindroma d postavljamo na 1, ako je pozicija i neparna, tj. 0 ako je parna. Zaista, ako je pozicija neparna, na njoj se nalazi slovo polazne reči koje je samo za sebe palindrom dužine 1 (iz drugog ugla gledano, levo i desno od ove pozicije se nalaze karakteri `|`, pa je bar jedan karakter jednak). Ako je pozicija parna, oko nje se nalaze dva slova (osim u slučaju krajnjih pozicija) i ne znamo unapred da li su ona jednaka, tako da dužinu palindroma inicijalno postavljamo na 0. Na ovaj način postižemo da su brojevi $i - d$ i $i + d$ parni, što znači da su brojevi $i - d - 1$ i $i + d + 1$ neparni i ako su u opsegu $[0, N)$, oni ukazuju na naredna dva karaktera polazne niske čiju jednakost treba proveriti. Ako su karakteri polazne

reči na odgovarajućim indeksima jednaki (to su indeksi $\lfloor \frac{i-d-1}{2} \rfloor$ i $\lfloor \frac{i+d+1}{2} \rfloor$), onda se d uvećava za 2 (iz jednog ugla gledano, ta dva jednaka karaktera se dodaju tekućem palindromu, pa mu se dužina povećava za 2, a iz drugog ugla gledano, ispred prvog i iza drugog se nalaze specijalni znaci | koji su sigurno jednaki i njihovu jednakost nije potrebno eksplicitno proveravati). Na taj način se održava i invarijanta da su brojevi $i + d$ i $i - d$ parni i petlja se može nastaviti na isti način sve dok se ne naiđe na dva različita slova ili se izađe van opsega dopunjene reči.

```
int main() {
  string s;
  cin >> s;
  // broj pozicija u reci s
  int N = 2 * s.size() + 1;
  int maxDuzina = 0, maxCentar;
  // za svaki moguci centar palindroma
  for (int i = 0; i < N; i++) {
    // d postavljamo na 0 za parnu, a na 1 za neparnu poziciju
    int d = 0;
    if (i % 2 == 1)
      d = 1;

    // dok god su pozicije u opsegu dozvoljenih indeksa i slova
    // na odgovarajucim indeksima jednaka uvecavamo d za 2
    while (i - d - 1 >= 0 && i + d + 1 < N &&
       s[(i - d - 1) / 2] == s[(i + d + 1) / 2])
      // ukljucujemo dva slova u palindrom, pa se duzina uvecava za 2
      d += 2;

    if (d > maxDuzina) {
      maxDuzina = d;
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  cout << s.substr(maxPocetak, maxDuzina) << endl;
  return 0;
}
```

4.4.6 Manačerov algoritam

Primer 4.4.2

Posmatrajmo sada kako izgleda dužina najdužeg palindroma sa centrom u svakoj od pozicija p u reči `s = babcbabcbaccba`. *Obeležimo ovu dužinu sa d_p. Obeležimo dopunjenu reč sa t. U prvom redu narednog prikaza dati su indeksi i u polaznoj reči s, u drugom redu proširena reč t, u trećem redu date su pozicije p u proširenoj reči, a u poslednjem vrednost d_p.*

```
   0   1   2   3   4   5   6   7   8   9   10  11  12  13
|  b  |  a  |  b  |  c  |  b  |  a  |  b  |  c  |  b  |  a  |  c  |  c  |  b  |  a  |
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
0  1  0  3  0  1  0  7  0  1  0  9  0  1  0  5  0  1  0  1  0  1  2  1  0  1  0  1  0
```

Posmatrajmo, na primer, najduži palindrom sa centrom u poziciji 11 – njegova dužina je $d_{11} = 9$ i on se prostire od pozicije 2 do pozicije 20.

Posmatrajmo sada dužinu najdužeg palindroma sa centrom u poziciji 12. Pošto je prethodno određeni palindrom simetričan oko pozicije 11, poziciji 12, odgovara pozicija 10. Znamo da je $d_{10} = 0$. To je zato što je $t_9 \neq t_{11}$. Međutim, mi znamo da je $t_9 = t_{13}$ (pošto su obe pozicije unutar našeg palindroma), pa je zato $t_{11} \neq t_{13}$ i važi $d_{12} = d_{10} = 0$. Primetimo da ovo možemo konstatovati bez ikakve potrebe za novim upoređivanjem karaktera.

Posmatrajmo sada dužinu najdužeg palindroma sa centrom u poziciji 13. Njemu odgovara palindrom sa centrom na poziciji 9. Važi $d_9 = 1$, jer je $t_8 = t_{10}$ i $t_7 \neq t_{11}$. Na osnovu simetrije palindroma sa centrom u 11, važi $t_8 = t_{14}$, $t_{10} = t_{12}$, $t_7 = t_{15}$ i $t_{11} = t_{11}$. Zato je $t_{14} = t_{12}$ i $t_{15} \neq t_{11}$, pa je $d_{13} = d_9 = 1$.

Naizgled, važi da je za svako i unutar šireg palindroma sa centrom u nekoj poziciji C broj d_i jednak broju $d_{i'}$, gde se i' određuje kao simetrična pozicija poziciji i u odnosu na poziciju C (važi da je rastojanje od C do i i i' jednako, pa je $C - i' = i - C$, tj. $i' = C - (i - C)$). No, to nije uvek tačno.

Posmatrajmo sada vrednost d_{15} i njoj odgovarajuću vrednost d_7. One nisu jednake. Zašto? Posmatrajmo šta je ono što možemo zaključiti iz simetrije palindroma sa centrom na poziciji 11. Važi da je t_2 do t_{12} jednako odgovarajućim karakterima t_{20} do t_{10} – to je garantovano simetrijom i nije potrebno proveravati. Međutim, važi $d_7 = 7$. Znamo zato i da je $t_1 = t_{13}$, međutim, ne možemo da tvrdimo da simetrično u odnosu na poziciju 11 važi $t_{21} = t_9$, zato što t_{21} nije više deo palindroma sa centrom na poziciji C. Dakle, proveru da li važi $t_{21} = t_9$ je potrebno posebno izvršiti.

Razmotrimo sada opšti slučaj. Pretpostavimo da je $[L, R]$ palindrom sa centrom na poziciji C (tada je $C-L = R-C$), da je i neki indeks unutar tog palindroma (neka je $C < i < R$) i da je $i' = C-(i-C)$ njemu simetričan indeks. Ako je palindrom sa centrom u i' u potpunosti sadržan u palindromu (L, R) (bez uračunatih krajeva), tj. ako je $L < i' - d_{i'}$, tj. $d_{i'} < i' - L = (C-(i-C))-(C-(R-C)) = R-i$, tada je $d_i = d_{i'}$. Dokažimo ovo.

Za svaku vrednost $0 \leq j \leq d_{i'}$ treba dokazati da je $t_{i-j} = t_{i+j}$. Zaista, pošto važi $L < i' - j$ i $i + j < R$, zaključuje se da je $t_{i-j} = t_{i'+j}$ i $t_{i+j} = t_{i'-j}$. Međutim, pošto je i' centar palindroma dužine $d_{i'}$, važi $t_{i'-j} = t_{i'+j}$. Zato se na poziciji i nalazi centar palindroma dužine bar $d_{i'}$. Dokažimo da je ovo i gornje ograničenje, tj. dokažimo da je $t_{i-d_{i'}-1} \neq t_{i+d_{i'}+1}$. Pošto je $i + d_{i'} < R$, važi $i + d_{i'} + 1 \leq R$, pa je $t_{i-d_{i'}-1} = t_{i'+d_{i'}+1}$ i $t_{i+d_{i'}+1} = t_{i'-d_{i'}-1}$. Međutim, pošto je palindrom sa centrom u i' dužine $d_{i'}$, važi $t_{i'-d_{i'}-1} \neq t_{i'+d_{i'}+1}$.

Ako je $[L, R]$ palindrom sa centrom na poziciji C i ako je i neki indeks unutar tog palindroma ($C < i < R$), ali takav da je $d_{i'} \geq R - i$, onda možemo samo da zaključimo da je $d_i \geq R - i$.

Ovo inspiriše naredni algoritam, poznat pod nazivom *Manačerov algoritam*. Slično kao u prethodnoj verziji algoritma obrađujemo sve pozicije i od 0 do $N - 1$. Pri tom održavamo indekse C i R takve da je $[C - (R - C), R]$ najdesniji do sad pronađeni palindrom. Ako je $i \geq R$, tada palindrom sa centrom u i određujemo iz početka, povećavajući za dva dužinu palindroma d_i koja kreće od 0 ili 1 (u zavisnosti od parnosti pozicije i), sve dok je to moguće, isto kao u prethodno opisanom algoritmu. Međutim, ako je $i < R$, tada određujemo vrednost $i' = C - (i - C)$ i ako važi $d_{i'} < R - i$, postavljamo odmah $d_i = d_{i'}$. Ako je $d_{i'} \geq R - i$, tada dužinu d_i postavljamo na početnu vrednost $R - i$ tj. na $R - i + 1$ tako da su $i - d_i$ i $i + d_i$ parni brojevi, i onda je postepeno povećavamo za 2, sve dok je to moguće (opet, veoma slično kao u prethodno opisanom algoritmu). Ako je pronađeni palindrom sa centrom u i takav da mu desni kraj desno od pozicije R, onda njega proglašavamo za novi palindrom $[L, R]$, postavljajući mu centar $C = i$ i desni kraj $R = i + d_i$. Na početku možemo inicijalizovati $R = C = 0$ (na taj način obezbeđujemo da ne može da važi $i < R$ i da se na početku neće koristiti simetričnost okružujućeg palindroma).

```
int main() {
  string s;
  cin >> s;
  // broj pozicija u reci s (pozicije su ili slova originalnih reci,
  // ili su izmedju njih)
  int N = 2 * s.size() + 1;
  // d[i] je duzina najduzeg palindroma ciji je centar na poziciji i
  vector<int> d(N);

  // znamo da je [L, R] palindrom sa centrom u C
  int C = 0, R = 0; // L = C - (R - C)
  for (int i = 0; i < N; i++) {
    // karakter simetrican karakteru i u odnosu na centar C
    int i_sim = C - (i - C);
    if (i < R && i + d[i_sim] < R)
      // nalazimo se unutar palindroma [L, R], ciji je centar C
      // palindrom sa centrom u i_sim i palindrom sa centrom u i su
      // celokupno smesteni u palindrom (L, R)
      d[i] = d[i_sim];
    else {
      // ili se ne nalazimo u okviru nekog prethodnog palindroma ili
      // se nalazimo unutar palindroma [L, R], ali je palindrom sa
      // centrom u i_sim takav da nije celokupno smesten u (L, R);
      // u tom slucajmo znamo da je duzina palindroma sa centrom u i bar
      // bar R - i, a da li je vise od toga, treba proveriti
      d[i] = i <= R ? R - i : 0;

      // osiguravamo da je i + d[i] stalno paran broj
      if ((i + d[i]) % 2 == 1)
        d[i]++;

      // dok god su pozicije u dozvoljenom opsegu i slova na odgovarajucim
      // indeksima jednaka uvecavamo d[i] za 2 (jedno slovo s leva i
      // jedno slovo zdesna)
      while (i - d[i] - 1 >= 0 && i + d[i] + 1 < N &&
         s[(i - d[i] - 1) / 2] == s[(i + d[i] + 1) / 2])
        // ukljucujemo dva slova u palindrom, pa se duzina uvecava za 2
        d[i] += 2;
    }

    // ako palindrom sa centrom u i prosiruje desnu granicu
    // onda njega uzimamo za palindrom [L, R] sa centrom u C
    if (i + d[i] > R) {
      C = i;
      R = i + d[i];
    }
  }

  // pronalazimo najvecu duzinu palindroma i pamtimo njegov centar
  int maxDuzina = 0, maxCentar;
  for (int i = 0; i < N; i++) {
```

```
    if (d[i] > maxDuzina) {
      maxDuzina = d[i];
      maxCentar = i;
    }
  }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  int maxPocetak = (maxCentar - maxDuzina) / 2;
  cout << s.substr(maxPocetak, maxDuzina) << endl;
  return 0;
}
```

Složenost ovog algoritma je linearna. Intuitivno, pronalaženje kratkih palindroma zahteva mali broj izvršavanja unutrašnje petlje, dok pronalaženje jednog dugačkog palindroma zahteva duže izvršavanje unutrašnje petlje, ali dovodi do toga da će se u narednim koracima u velikom broju slučajeva u potpunosti izbegavati njeno izvršavanje. Preciznije, svako izvršavanje unutrašnje `while` petlje povećava vrednost promenljive R (jer je u slučaju `else` grane $i + d[i] \geq R$, a svako uspešno izvršavanje `while` petlje uvećava vrednost $d[i]$ za 2, te će važiti $i + d[i] > R$) koja se nigde ne smanjuje, a čija je maksimalna vrednost N, te je ukupan broj izvršavanja unutrašnje petlje ograničen sa $O(N)$.

Možemo razmotriti i varijantu koja eliminiše proveru pripadnosti indeksa opsegu reči na taj način što eksplicitno pravi dopunjenu reč.

```
string dopuni(const string &s) {
  string rez = "^";
  for (int i = 0; i < s.size(); i++)
    rez += "|" + s.substr(i, 1);
  rez += "|$";
  return rez;
}

int main() {
  string s;
  cin >> s;

  // jednostavnosti radi dopunjavamo rec s
  string t = dopuni(s);
  // d[i] je najveci broj takav da je [i - d[i], i + d[i]] palindrom
  // to je ujedno i duzina najduzeg palindroma ciji je centar na
  // poziciji i (pozicije su ili slova originalnih reci, ili su
  // izmedju njih)
  vector<int> d(t.size());
  // znamo da je [L, R] palindrom sa centrom na poziciji C
  int C = 0, R = 0; // L = C - (R - C)
  for (int i = 1; i < t.size() - 1; i++) {
    // karakter simetrican karakteru i u odnosu na centar C
    int i_sim = C - (i - C);
    if (i < R && i + d[i_sim] < R)
      // nalazimo se unutar palindroma [L, R], ciji je centar C
      // palindrom sa centrom u i_sim i palindrom sa centrom u i su
      // celokupno smesteni u palindrom (L, R)
      d[i] = d[i_sim];
```

```
    else {
      // ili se ne nalazimo u okviru nekog prethodnog palindroma ili
      // se nalazimo unutar palindroma [L, R], ali je palindrom sa
      // centrom u i_sim takav da nije celokupno smesten u (L, R);
      // u tom slucajmo znamo da je duzina palindroma sa centrom u i bar
      // R - i, a da li je vise od toga, treba proveriti
      d[i] = i <= R ? R-i : 0;
      // prosirujemo palindrom dok god je to moguce krajnji karakteri
      // ^$ obezbedjuju da nije potrebno proveravati granice
      while (t[i - d[i] - 1] == t[i + d[i] + 1])
        d[i]++;
    }

    // ako palindrom sa centrom u i prosiruje desnu granicu
    // onda njega uzimamo za palindrom [L, R] sa centrom u C
    if (i + d[i] > R) {
      C = i;
      R = i + d[i];
    }
  }

  // pronalazimo najvecu duzinu palindroma i pamtimo njegov centar
  int maxDuzina = 0, maxCentar;
  for (int i = 1; i < t.size() - 1; i++)
    if (d[i] > maxDuzina) {
      maxDuzina = d[i];
      maxCentar = i;
    }

  // ispisujemo konacan rezultat, odredjujuci pocetak najduzeg palindroma
  cout << s.substr((maxCentar - maxDuzina) / 2, maxDuzina) << endl;
  return 0;
}
```

5. Geometrijski algoritmi

Geometrijski algoritmi igraju važnu ulogu u mnogim oblastima, poput računarske grafike, projektovanja pomoću računara, projektovanju integrisanih kola visoke rezolucije (VLSI), robotike i baza podataka. U računarski generisanoj slici može biti na hiljade ili čak na milione tačaka, linija, trouglova, kvadrata ili krugova; slično, projektovanje računarskog čipa može da zahteva rad sa milionima elemenata koji se predstavljaju geometrijskim figurama. Sve ove primene zahtevaju obradu geometrijskih objekata. Pošto veličina ulaza za ove probleme može biti vrlo velika, veoma je značajno razviti što efikasnije algoritme za njihovo rešavanje.

Česta neugodna karakteristika geometrijskih problema je postojanje mnogih specijalnih slučajeva. Na primer, dve prave u ravni se seku u jednoj tački, sem ako su paralelne ili se poklapaju. Pri rešavanju problema sa dve prave, moraju se predvideti sva tri moguća slučaja. Složenije figure prouzrokuju pojavu mnogo većeg broja specijalnih slučajeva i o njima treba voditi računa. Obično se većina tih specijalnih slučajeva neposredno rešava, ali potreba da se oni uzmu u obzir čini ponekad konstrukciju geometrijskih algoritama vrlo iscrpljujućom. Mi ćemo ponekad ignorisati detalje koji nisu od suštinskog značaja za razumevanje osnovnih ideja algoritama, ali čitaocu savetujemo da razmotri rešavanje svakog od specijalnih slučajeva na koji se pri konstrukciji algoritma naiđe, kako bi se dobili algoritmi koji su potpuno ispravni.

Na početku ovog poglavlja razmotrićemo neke osnovne pojmove koji nam mogu biti od pomoći prilikom rešavanja različitih geometrijskih problema: korišćenjem orijentacije trojke tačaka možemo, na primer, utvrditi da li se dve duži seku ili ispitati da li je tačka unutar ili van datog trougla. Nakon toga ćemo razmotriti nekoliko fundamentalnih geometrijskih algoritama nad skupom tačaka u ravni — konstrukciju prostog mnogougla, ispitivanje da li tačka pripada prostom mnogouglu, kao i različite algoritme za konstrukciju konveksnog omotača skupa tačaka.

5.1 Osnove geometrijskih algoritama

Svi objekti se u računaru predstavljaju brojevima (koordinatama, koeficijentima i slično). Pretpostavlja se da je čitalac upoznat sa osnovama analitičke geometrije. Ipak, u nastavku ćemo rezimirati neke osnovne pojmove korisne prilikom konstrukcije geometrijskih algoritama, pri čemu ćemo većinu operacija zasnovati na vektorima i operacijama sa njima (alternativno, u tradicionalnoj analitičkoj geometriji se geometrijski objekti obično predstavljaju implicitnim jednačinama). Bavićemo se pitanjima poput izračunavanja rasto-

janja između dve tačke, ispitivanja kolinearnosti tri tačke ili izračunavanja presečne tačke dveju duži. Sve ove operacije mogu se izvesti algoritmima konstantne vremenske složenosti, korišćenjem osnovnih aritmetičkih operacija. Pretpostavljamo i da se različite elementarne funkcije (kvadratni koren, trigonometrijske funkcije i slično) mogu izračunati za konstantno vreme.

5.1.1 Tačke, koordinate

Osnovni geometrijski objekti su *tačke* i većina drugih objekata se predstavlja pomoću tačaka.

5.1.1.1 Dekartove koordinate tačaka

Tačke se u računaru najčešće predstavljaju svojim *Dekartovim koordinatama*. Tačke u ravni se predstavljaju parom Dekartovih koordinata (x, y), a tačke u trodimenzionom prostoru trojkom Dekartovih koordinata (x, y, z). U zavisnosti od primene, koordinate su najčešće ili celobrojne ili realne (u računaru najčešće predstavljene brojevima u pokretnom zarezu).

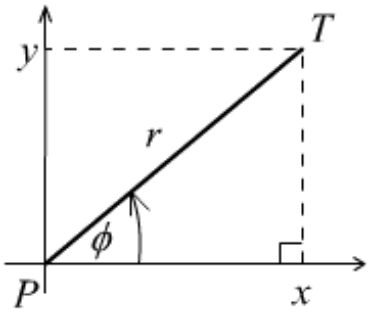
Tačke se mogu predstaviti ili strukturnim tipom ili ugrađenim tipom za predstavljanje uređenih parova. Na primer, tačke sa celobrojnim koordinatama se mogu prestaviti na bilo koji od sledeća dva načina.

```
typedef struct {
   int x, y;
} Tacka;

typedef pair<int, int> Tacka;
```

Naravno, umesto tipa `int` mogu se koristiti i drugi tipovi za predstavljanje koordinata (npr. `long long`, `float`, `double` itd.).

5.1.1.2 Polarne koordinate

U nekim algoritmima je pogodnija reprezentacija tačaka pomoću njihovih *polarnih koordinata*. Zbog toga je za date Dekartove koordinate x i y tačke T ponekad potrebno odrediti njene polarne koordinate: rastojanje r od koordinatnog početka P i ugao ϕ koji duž PT zahvata sa pozitivnim delom x ose i, obratno, na osnovu polarnih koordinata odrediti Dekartove. Ugao se često razmatra kao orijentisani ugao iz intervala $(-\pi, \pi]$ ili $[0, 2\pi)$, gde ugao 0 predstavlja pozitivan smer x ose i ugao raste u pozitivnom matematičkom smeru.



Slika 5.1: Veza između Dekartovih i polarnih koordinata tačke.

Da bismo transformisali polarne koordinate u Dekartove, možemo iskoristiti sledeću vezu (slika 5.1):

$$\begin{aligned} x &= r\cos\phi \\ y &= r\sin\phi \end{aligned}$$

Za transformaciju Dekartovih koordinata u polarne u većini slučajeva možemo iskoristiti naredne jednačine:

$$\begin{aligned} r &= \sqrt{x^2+y^2} \\ \phi &= \arctan\frac{y}{x} \end{aligned}$$

Treba obratiti pažnju na to da prethodne formule nisu uvek ispravne, jer vrednost arctan $\frac{y}{x}$ nije definisana za tačke kod kojih je $x = 0$, tj. za tačke na y-osi, kao i da je vrednost funkcije arctan uvek u intervalu $[0, \pi)$ (ili u intervalu $(-\pi/2, \pi/2]$), a ne u intervalu $(-\pi, \pi]$ (ili u intervalu $[0, 2\pi)$), kako bismo želeli. Zbog toga se definiše i koristi funkcija atan2, koja ima dva argumenta. Ona je direktno podržana u većini programskih jezika i vraća ugao iz intervala $(-\pi, \pi]$.

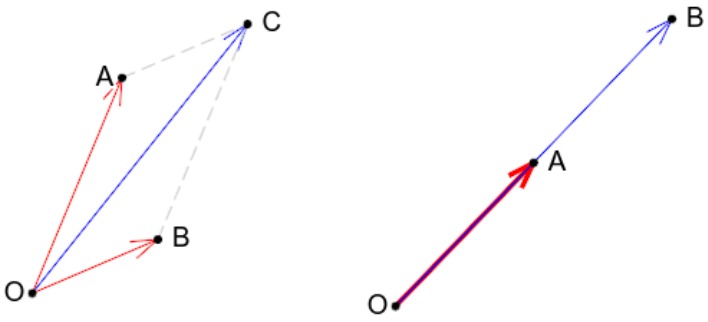
$$\text{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{za } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{za } x < 0 \text{ i } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{za } x < 0 \text{ i } y < 0 \\ +\frac{\pi}{2} & \text{za } x = 0 \text{ i } y > 0 \\ -\frac{\pi}{2} & \text{za } x = 0 \text{ i } y < 0 \\ \text{nedefinisano} & \text{za } x = 0 \text{ i } y = 0 \end{cases}$$

Polarni ugao tačke (x, y) se tada izračunava kao $\phi = \text{atan2}(y, x)$ (obratiti pažnju na redosled argumenata).

5.1.2 Vektori

Pored tačaka, osnovni pojam analitičke geometrije predstavljaju *vektori* koji se predstavljaju usmerenim dužima. Usmerena duž koja ima početnu tačku A i krajnju tačku B određuje vektor AB. Svi drugi parovi tačaka $A'B'$ takvih da postoji translacija kojom se AB preslikava u $A'B'$ određuju isti vektor. Ako su početna i krajnja tačka jednake, one određuju tzv. *nula-vektor* (govorićemo i da je u pitanju vektor jednak nuli).

Vektori se mogu sabirati i skalirati (množiti brojem tj. skalarom). Izraz $\vec{a} + \vec{b}$ označava zbir vektora $\vec{a}$ i $\vec{b}$, dok izraz $k \cdot \vec{a}$ označava skaliranje vektora $\vec{a}$ skalarom k. Vektori se mogu i oduzimati. Razlika vektora $\vec{a}$ i $\vec{b}$ se definiše kao $\vec{a} - \vec{b} = \vec{a} + (-1) \cdot \vec{b}$. Sabiranje i skaliranje imaju svoju geometrijsku interpretaciju (na slici 5.2 prikazani su primeri ovih operacija).



Slika 5.2: Vektori se sabiraju na osnovu pravila paralelograma (zbir vektora je dijagonala paralelograma koji obrazuju vektori sabirci). Na slici levo vektor $\overrightarrow{OC}$ je zbir vektora $\overrightarrow{OA}$ i $\overrightarrow{OB}$. Skaliranje vektora podrazumeva njegovo izduživanje ili skraćivanje (a ako je skalar negativan, tada i promenu smera). Na slici desno vektor $\overrightarrow{OB}$ je $2 \cdot \overrightarrow{OA}$.

Vektori se najčešće predstavljaju analitički, svojim Dekartovim koordinatama (parom koordinata za vektore u ravni i trojkom koordinata za vektore u prostoru). Svaki vektor u ravni se može jednoznačno izraziti kao

linearna kombinacija bilo koja dva nekolinearna (obično međusobno normalna) tzv. jedinična vektora $\vec{i}$ i $\vec{j}$, tj.
za svaki vektor $\vec{a}$ postoje brojevi x i y takvi da je $\vec{a} = x \cdot \vec{i} + y \cdot \vec{j}$. Brojevi (x, y) predstavljaju *koordinate* vektora $\vec{a}$.

Sabiranje i skaliranje se lako izražavaju analitički. Zbir vektora $\vec{a} = (a_x, a_y)$ i $\vec{b} = (b_x, b_y)$ je vektor $\vec{a} + \vec{b} = (a_x + b_x, a_y + b_y)$. Proizvod vektora $\vec{a} = (a_x, a_y)$ i skalara k je vektor $k \cdot \vec{a} = (ka_x, ka_y)$.

Vektor se u jeziku C++ može predstaviti sledećom strukturom.

```
struct vektor {
  double x, y;
};
```

Tada je lako definisati i osnovne operacije sa vektorima.

```
// odredjuje se vektor v1 + v2
vektor zbir(const vektor& v1, const vektor& v2) {
  vektor rez;
  rez.x = v1.x + v2.x;
  rez.y = v1.y + v2.y;
  return rez;
}

// odredjuje se vektor k * v
vektor skaliranje(double k, const vektor& v) {
  vektor rez;
  rez.x = v.x * k;
  rez.y = v.y * k;
  return rez;
}
```

Dekartov koordinatni sistem je određen međusobno normalnim jediničnim vektorima $\vec{i}$ i $\vec{j}$ i koordinatnim početkom O. Koordinate bilo koje tačke A su zapravo koordinate vektora $\overrightarrow{OA}$.

- *Sabiranjem tačke i vektora tačka se translira za dati vektor.* Ako su poznate koordinate tačke A i vektora $\overrightarrow{AB}$, njihovim sabiranjem dobijaju se koordinate tačke B. Zaista, važi $\overrightarrow{OB} = \overrightarrow{OA} + \overrightarrow{AB}$. Ovo ćemo kraće zapisivati kao: $B = A + \overrightarrow{AB}$.
- *Oduzimanjem tačaka dobija se vektor koji ih spaja.* Ako su poznate koordinate početne i krajnje tačke vektora $\overrightarrow{AB}$, tada se koordinate vektora $\overrightarrow{AB}$ mogu izračunati oduzimanjem koordinata tačke A od koordinata tačke B. Zaista, važi $\overrightarrow{AB} = \overrightarrow{OB} - \overrightarrow{OA}$. Ovo ćemo kraće zapisivati kao: $\overrightarrow{AB} = B - A$.

5.1.2.1 Skalarni proizvod

Skalarni proizvod $\vec{a} \circ \vec{b}$ vektora $\vec{a}$ i $\vec{b}$ je skalar (broj):

$$\vec{a} \circ \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \cos \phi,$$

gde je sa $|\vec{a}|$ označen intenzitet vektora $\vec{a}$, a sa ϕ konveksni ugao (manji ili jednak π) koji zahvataju vektori $\vec{a}$ i $\vec{b}$. Jednostavno se dokazuje da skalarni proizvod ima sledeće osobine:

$$\begin{aligned}
\vec{a} \circ \vec{b} &= \vec{b} \circ \vec{a} && \text{simetričnost} \\
(k_1\vec{a}_1 + k_2\vec{a}_2) \circ \vec{b} &= k_1(\vec{a}_1 \circ \vec{b}) + k_2(\vec{a}_2 \circ \vec{b}) && \text{linearnost po prvom argumentu} \\
\vec{a} \cdot (k_1\vec{b}_1 + k_2\vec{b}_2) &= k_1(\vec{a} \circ \vec{b}_1) + k_2(\vec{a} \circ \vec{b}_2) && \text{linearnost po prvom argumentu} \\
\vec{a} \cdot \vec{a} &\geq 0 && \text{nenegativnost} \\
\vec{a} \cdot \vec{a} = 0 &\Leftrightarrow \vec{a} = \vec{0} && \text{pozitivna definitnost}
\end{aligned}$$

Ako umemo da izračumo skalarni proizvod bilo koja dva vektora, na osnovu ove formule moguće je izračunati:

- intenzitet, tj. dužinu vektora $\vec{u}$, na osnovu formule $|\vec{u}|^2 = \vec{u} \circ \vec{u}$, jer je $\cos 0^\circ = 1$,
- konveksni ugao ϕ između vektora $\vec{u}$ i $\vec{v}$, na osnovu formule $\phi = \arccos \frac{\vec{u} \circ \vec{v}}{|\vec{u}| \cdot |\vec{v}|}$. Obratite pažnju na to da je vrednost funkcije arccos uvek između 0 i π, pa se skalarnim proizvodom uvek izračunava vrednost konveksnog ugla između vektora.

Specijalno, skalarni proizvod dva vektora različita od nule je nula ako i samo ako je $\cos \phi = 0$ tj. ako su oni normalni, pa se skalarni proizvod može koristiti i za proveru da li su vektori međusobno normalni. Negativna vrednost skalarnog proizvoda ukazuje na to da je manji (konveksni) ugao između vektora tup, a pozitivna da je taj ugao oštar.

Skalarni proizvod se lako izračunava analitički, kada su vektori zadati koordinatama u nekom koordinatnom sistemu određenom jediničnim, međusobno normalnim vektorima (što je jedini slučaj koji razmatramo). Naime, ako su dva vektora $\vec{a}(a_1, \ldots, a_n)$ i $\vec{b}(b_1, \ldots, b_n)$ iste dimenzije, njihov *skalarni proizvod* je skalar čiju vrednost računamo po formuli:

$$\vec{a} \circ \vec{b} = \sum_{i=1}^{n} a_i \cdot b_i$$

Dokažimo prethodno tvrđenje u slučaju kada su vektori dimenzije 2 (on se lako uopštava na slučaj vektora dimenzije veće od 2). Naime, formula za izračunavanje skalarnog proizvoda pomoću koordinata sledi iz činjenice da za jedinične, međusobno normalne vektore $\vec{i}$ i $\vec{j}$ važi da je $\vec{i} \circ \vec{i} = \vec{j} \circ \vec{j} = 1$ i $\vec{i} \circ \vec{j} = \vec{j} \circ \vec{i} = 0$ (jer je $\cos 0 = 1$, a $\cos \frac{\pi}{2} = 0$). Svaki vektor se može razložiti na komponente duž x-ose i duž y-ose korišćenjem svojih koordinata. Zato je (korišćenjem linearnosti):

$$(a_x\vec{i} + a_y\vec{j}) \circ (b_x\vec{i} + b_y\vec{j}) = a_xb_x(\vec{i} \circ \vec{i}) + a_xb_y(\vec{i} \circ \vec{j}) + a_yb_x(\vec{j} \circ \vec{i}) + a_yb_y(\vec{j} \circ \vec{j}) = a_xb_x + a_yb_y$$

Izračunavanje skalaranog proizvoda na osnovu prethodne formule se lako može implementirati (pretpostavljamo da su vektori predstavljeni pomoću n-točlanih nizova koordinata).

```
// odredjuje se skalarni proizvod n-dimenzionih vektora v1 i v2
double skalarniProizvodN(const vector<double>& v1,
                         const vector<double>& v2) {
  // obe dimenzije moraju biti jednake
  assert(v1.size() == v2.size());
  double rez = 0.0;
  for (int i = 0; i < v1.size(); i++)
    rez += v1[i] * v2[i];
  return rez;
}
```

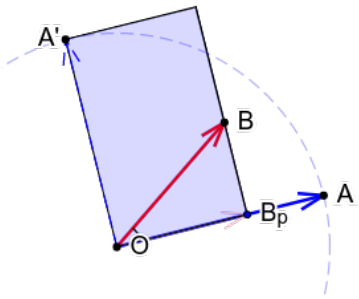
Ukoliko su koordinate vektora $\vec{a}$ i $\vec{b}$ celobrojne, i vrednost skalarnog proizvoda je celobrojna.

U daljem tekstu ćemo razmatrati vektore određene tačkama u ravni, te će dimenzija vektora biti jednaka 2. Skalarni proizvod vektora $\vec{a}(a_x, a_y)$ i vektora $\vec{b}(b_x, b_y)$ dimenzije 2 jednak je $\vec{a} \circ \vec{b} = a_x \cdot b_x + a_y \cdot b_y$.

Ovu funkciju je još lakše implementirati.

```
// odredjuje se skalarni proizvod vektora v1 i v2 u ravni
double skalarniProizvod(const vektor& v1, const vektor& v2) {
  return v1.x*v2.x + v1.y*v2.y;
}
```

U prethodnom izvođenju smo projektovali oba vektora na koordinatne ose. Umesto toga, moguće je odrediti *projekciju jednog vektora na pravac drugog*. Time se dobijaju dva kolinearna vektora čiji se skalarni proizvod dobija množenjem njihovih intenziteta (vodeći računa o znaku).



Slika 5.3: Veza između skalarnog proizvoda i projekcije vektora. Apsolutna vrednost skalarnog proizvoda jednaka je površini pravougaonika određenog vektorima $\overrightarrow{OB_p}$ i $\overrightarrow{OA'}$.

Na slici 5.3, tačka B_p je projekcija tačke B (krajnje tačke vektora $\overrightarrow{OB}$) na pravac vektora $\overrightarrow{OA}$. Dužina OB_p je jednaka apsolutnoj vrednosti izraza $|\overrightarrow{OB}| \cos \phi$, pri čemu negativan znak tog izraza ukazuje na to da je vektor $\overrightarrow{OB_p}$ suprotno usmeren od vektora $\overrightarrow{OA}$, a pozitivan znak na to da su ta dva vektora isto usmerena.

Apsolutna vrednost skalarnog proizvoda $\overrightarrow{OA} \circ \overrightarrow{OB} = |\overrightarrow{OA}||\overrightarrow{OB}| \cos \phi$ jednaka je proizvodu dužina OB_p i OA (pri čemu je znak sklarnog proizvoda negativan ako su vektori $\overrightarrow{OB_p}$ i $\overrightarrow{OA}$ suprotno usmereni, a pozitivan ako su isto usmereni). Da bismo vizuelno predstavili apsolutnu vrednost skalarnog proizvoda, vektor $\overrightarrow{OA}$ rotiramo za $90°$ tj. $\frac{\pi}{2}$ radijana, čime se dobija vektor $\overrightarrow{OA'}$, koji sa vektorom $\overrightarrow{OB_p}$ gradi pravougaonik (prikazan plavom bojom na slici 5.3) čija je površina jednaka apsolutnoj vrednosti skalarnog proizvoda.

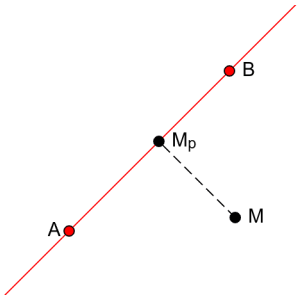
Naravno, pošto je sve simetrično, moguće je projektovati tačku A na pravac vektora $\overrightarrow{OB}$ (tako dobijamo tačku A_p i pravougaonik određen vektorima $\overrightarrow{OA_p}$ i $\overrightarrow{OB'}$, čija je površina takođe jednaka apsolutnoj vrednosti skalarnog proizvoda).

Projekcija tačke na pravu

Videli smo da je skalarni proizvod u tesnoj vezi sa projekcijom tačke na pravu (pravac jednog od vektora koji se množe). Stoga se ta projekcija se veoma jednostavno određuje korišćenjem skalarnog proizvoda.

> **Problem**
>
> *Izračunati koordinate normalne projekcije M_p tačke M na pravu AB (slika 5.4).*

Za tačku M_p važi $M_p = A + \overrightarrow{AM_p}$. Vektor $\overrightarrow{AM_p}$ kolinearan je sa vektorom $\overrightarrow{AB}$, a dužina vektora $\overrightarrow{AM_p}$ jednaka je apsolutnoj vrednosti izraza $|AM| \cos \phi$. Vrednost skalarnog proizvoda vektora $\overrightarrow{AB}$ i $\overrightarrow{AM}$ jednaka je $|AB||AM| \cos \phi$, pa je dužina vektora $\overrightarrow{AM_p}$ jednaka apsolutnoj vrednosti izraza



Slika 5.4: Projekcija tačke M na pravu AB.

$$\frac{\overrightarrow{AB} \circ \overrightarrow{AM}}{|\overrightarrow{AB}|}$$

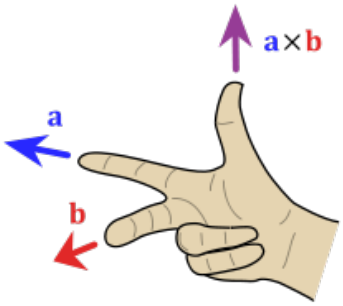
pri čemu znak ove vrednosti govori o tome da li su vektori $\overrightarrow{AB}$ i $\overrightarrow{AM_p}$ isto ili suprotno usmereni. Vektor $\overrightarrow{AM_p}$ se može dobiti tako što se jedinični vektor $\frac{\overrightarrow{AB}}{|\overrightarrow{AB}|}$ pomnoži ovim koeficijentom. Zato je

$$M_p = A + \overrightarrow{AM_p} = A + \frac{\overrightarrow{AB} \circ \overrightarrow{AM}}{|\overrightarrow{AB}|^2}\overrightarrow{AB}.$$

Naglasimo da se ova formula može koristiti i za tačke u ravni i za tačke u prostoru.

5.1.2.2 Vektorski proizvod

Vektorski proizvod vektora $\vec{a}(a_x, a_y, a_z)$ i $\vec{b}(b_x, b_y, b_z)$ dimenzije 3 je vektor normalan na ravan određenu vektorima $\vec{a}$ i $\vec{b}$, čiji je smer određen pravilom desne ruke (slika 5.5), a intenzitet jednak površini paralelograma koji određuju vektori $\vec{a}$ i $\vec{b}$, odnosno može se izračunati korišćenjem formule $|\vec{a} \times \vec{b}| = |\vec{a}| \cdot |\vec{b}| \cdot \sin \alpha$, gde je sa α označen manji od uglova koji obrazuju vektori $\vec{a}$ i $\vec{b}$ (tj. konveksni ugao, čiji je sinus uvek pozitivan).



Slika 5.5: Pravilo desne ruke: ako kažiprst i srednji prst pokazuju redom u smeru vektora $\vec{a}$ i $\vec{b}$, palac će određivati smer njihovog vektorskog proizvoda $\vec{a} \times \vec{b}$.

Lako se pokazuju osnovna svojstva vektorskog proizvoda:

$$\begin{aligned}
\vec{a} \times \vec{b} &= -\vec{b} \times \vec{a} && \text{antikomutativnost} \\
\vec{a} \times (\vec{b}_1 + \vec{b}_2) &= \vec{a} \times \vec{b}_1 + \vec{a} \times \vec{b}_1 && \text{desna distributivnost} \\
(\vec{a}_1 + \vec{a}_2) \times \vec{b} &= \vec{a}_1 \times \vec{b} + \vec{a}_2 \times \vec{b} && \text{leva distributivnost} \\
\vec{a} \times \vec{a} &= 0 && \text{samoproizvod} \\
(k \cdot \vec{a}) \times b = \vec{a} \times (k \cdot \vec{b}) &= k \cdot (\vec{a} \times \vec{b}) && \text{odnos sa skaliranjem}
\end{aligned}$$

Naglasimo i da nam skalarni proizvod daje način da lako odredimo kosinus ugla između dva vektora, a odatle i veličinu konveksnog ugla između njih. Sa druge strane, iz vektorskog proizvoda se lako određuje sinus ugla između dva vektora, međutim, vrednost sinusa nije dovoljna da se lako odredi konveksni ugao (jer istu vrednost sinusa daju jedan oštar i jedan tup ugao). Stoga se za određivanje veličine konveksnog ugla između dva vektora koristi skalarni, a ne vektorski proizvod.

Pošto važi da je:

$$\begin{aligned}
\vec{a} &= a_x \cdot \vec{i} + a_y \cdot \vec{j} + a_z \cdot \vec{k} \\
\vec{b} &= b_x \cdot \vec{i} + b_y \cdot \vec{j} + b_z \cdot \vec{k}
\end{aligned}$$

gde su sa $\vec{i}$, $\vec{j}$ i $\vec{k}$ označeni međusobno normalni jedinični vektori u smeru x, y i z koordinatne ose, i važi:

$$\begin{aligned}
\vec{i} \times \vec{j} = \vec{k} = -\vec{j} \times \vec{i} \\
\vec{j} \times \vec{k} = \vec{i} = -\vec{k} \times \vec{j} \\
\vec{k} \times \vec{i} = \vec{j} = -\vec{i} \times \vec{k}
\end{aligned}$$

važi da je vektorski proizvod vektora $\vec{a}$ i $\vec{b}$ jednak:

$$\vec{a} \times \vec{b} = (a_y b_z - a_z b_y)\vec{i} + (a_z b_x - a_x b_z)\vec{j} + (a_x b_y - a_y b_x)\vec{k}$$

Ova formula se može zapisati u obliku naredne determinante:

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

u čijoj se prvoj vrsti nalaze jedinični vektori u smeru x, y i z ose, u drugoj vrsti koordinate vektora $\vec{a}$, a u trećoj koordinate vektora $\vec{b}$. Ukoliko su koordinate vektora $\vec{a}$ i $\vec{b}$ celobrojne, i koordinate vektora $\vec{a} \times \vec{b}$ biće celobrojne.

Vektorski proizvod je definisan samo za vektore u trodimenzionom prostoru. Ipak, i vektori u ravni se mogu vektorski množiti ako se ravan xOy utopi u trodimenzioni prostor. Tada se koordinate svakog od dva vektora ravni prošire nulom (z-koordinata tih vektora je nula) i izvrši se vektorsko množenje. Rezultat je vektor koji je normalan na ravan xOy tj. vektor čije su prve dve koordinate jednake nuli. Zaista

$$\vec{a} \times \vec{b} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ a_x & a_y & 0 \\ b_x & b_y & 0 \end{vmatrix} = (0, 0, a_x b_y - a_y b_x).$$

Jasno je da će u ovom slučaju prve dve koordinate biti nula i da nam je jedino značajna z koordinata vektorskog proizvoda, pa se vektorski proizvod dva ravanska vektora $\vec{a} = (a_x, a_y)$ i $\vec{b} = (b_x, b_y)$ poistovećuje sa skalarom $a_x b_y - a_y b_x$ (što može biti zbunjuće, jer nazivi skalarni i vektorski proizvod potiču od toga što je u prvom slučaju rezultat množenja skalar, a u drugom vektor). Da bismo izbegli zabunu, naglašavaćemo uvek da je u pitanju množenje dvodimenzionalnih vektora (tj. *dvodimenzionalni vektorski proizvod*[1]) i umesto oznake $\times$ koristićemo oznaku $\times_{2d}$. Dakle, za dvodimenzionalne vektore $\vec{a} = (a_1, a_2)$ i $\vec{b} = (b_1, b_2)$ definišemo da je:

$$\vec{a} \times_{2d} \vec{b} = a_x b_y - a_y b_x.$$

Naredna funkcija izračunava vektorski proizvod dva dvodimenzionalna vektora (tj. z-koordinatu vektorskog proizvoda odgovarajuća dva vektora u prostoru).

```
// odredjuje se vektorski proizvod (tj. njegova z koordinata) vektora v1 i v2 u ravni
double vektorskiProizvod2d(const vektor& v1, const vektor& v2) {
  return v1.x*v2.y - v1.y*v2.x;
}
```

Ako dvodimenzionalni vektori obrazuju ugao ϕ, vrednost dvodimenzionalnog vektorskog proizvoda $\vec{a} \times_{2d} \vec{b}$ jednaka je $|\vec{a}| \cdot |\vec{b}| \cdot \sin\phi$.

Vrednost $\vec{a} \times_{2d} \vec{b} = 0$ nam govori da su vektori $\vec{a}$ i $\vec{b}$ kolinearni (da je ugao koji obrazuju 0 ili π) i to se obično koristi kao potreban i dovoljan uslov ispitivanje kolinearnosti vektora. Zaista, pošto je $\vec{a} \times_{2d} \vec{b} = |\vec{a}| \cdot |\vec{b}| \cdot \sin\phi$, važi da je $\sin\phi = \frac{\vec{a} \times \vec{b}}{|\vec{a}| \cdot |\vec{b}|}$, a sinus je jednak nuli za ugao 0 ili π.

Jedna od osnovnih primena vektorskog proizvoda može biti da se odredi da li su tri tačke kolinearne.

Problem

Date su tri tačke u prostoru (ili u ravni). Ispitati da li su one kolinearne.

Ako razmatramo tačke u trodimenzionalnom prostoru, tačka X pripada pravoj koja je određena tačkama A i B ako i samo ako je $\overrightarrow{XA} \times \overrightarrow{XB} = \vec{0}$ (a slično i, na primer, ako i samo ako je $\overrightarrow{AB} \times \overrightarrow{AX} = \vec{0}$). Pri tome treba biti obazriv ako se radi sa realnim koordinatama (jer je zbog računskih grešaka malo verovatno da će se dobiti vrednosti koje su baš tačno jednake nuli). Ako su tačke A, B i X date koordinatama u ravni, tada možemo razmatratimo dvodimenzionalni vektorski proizvod $\overrightarrow{XA} \times_{2d} \overrightarrow{XB} = 0$. i proveravati da li je njegova vrednost nula.

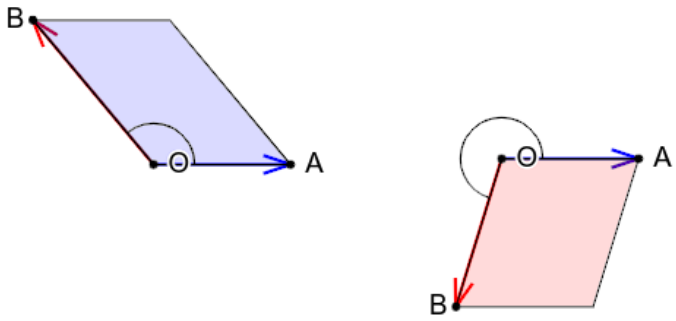
Znak dvodimenzionalnog vektorskog proizvoda

Znak vrednosti dvodimenzionalnog vektorskog proizvoda $\vec{a} \times_{2d} \vec{b}$ nam govori o orijentisanom uglu između vektora $\vec{a}$ i $\vec{b}$, pri čemu se sada ugao posmatra u intervalu $[0, 2\pi)$ (slika 5.6). Pozitivne vrednosti govore da je ugao koji se opisuje rotiranjem vektora $\vec{a}$ ka vektoru $\vec{b}$ u pozitivnom matematičkom smeru konveksan (u intervalu je $(0, \pi)$), a negativne vrednosti da je taj ugao nekonveksan (u intervalu je $(\pi, 2\pi)$). Ovo, naravno, odgovara znaku sinusa ugla.

Još jedna interpretacija je da pozitivni znak dvodimenzionalnog vektorskog proizvoda ukazuje da je najbliži put od vektora $\vec{a}$ do vektora $\vec{b}$ "nalevo" tj. u pozitivnom matematičkom smeru, a da negativni znak ukazuje da je najbliži put "nadesno".

Znak dvodimenzionalnog vektorskog proizvoda se može iskoristiti i da se tačke klasifikuju u dve poluravni na koje prava AB deli ravan. Negativna vrednost vektorskog proizvoda $\overrightarrow{OA} \times_{2d} \overrightarrow{OB}$ ukazuje da tačka O

[1] Nekada se ova vrednost naziva i *označena površina* (engl. signed area) paralelograma određenog preko dva vektora u ravni.



Slika 5.6: Pozitivan znak vektorskog proizvoda ukazuje na konveksan, a negativni na nekonveksan orijentisani ugao između vektora $\overrightarrow{OA}$ i $\overrightarrow{OB}$

pripada jednoj poluravni, a pozitivna da pripada drugoj (dok vrednost 0 ukazuje na to da tačka O pripada graničnoj pravoj AB).

Znak vrednosti $\overrightarrow{OA} \times_{2d} \overrightarrow{OB}$ određuje takozvanu *orijentaciju trojke tačaka* OAB i mnogi geometrijski problemi se rešavaju primenom orijentacije (neke primename orijentacije opisane su u poglavlju 5.1.4).

Da rezimiramo, sledeći uslovi su ekvivalentni:

- Trojka tačaka OAB u ravni je pozitivno orijentisana.
- Znak dvodimenzionalnog vektorskog proizvoda $\overrightarrow{OA} \times_{2d} \overrightarrow{OB}$ je pozitivan.
- Najbliži put od vektora $\overrightarrow{OA}$ do vektora $\overrightarrow{OB}$ je "nalevo" tj. u pozitivnom matematičkom smeru.
- Ugao od vektora $\overrightarrow{OA}$ do vektora $\overrightarrow{OB}$ u pozitivnom matematičkom smeru je konveksan.

5.1.3 Površina i primene

Problem

Izračunati površinu trougla čija su temena tri tačke u ravni zadate svojim koordinatama.

Površina trougla čije su stranice dva data vektora $\vec{a}$ i $\vec{b}$ je polovina površine paralelograma koji oni grade i jednaka je polovini apsolutne vrednosti njihovog vektorskog proizvoda. Ako su poznate koordinate temena trougla $A(a_x, a_y)$, $B(b_x, b_y)$ i $C(c_x, c_y)$, vektori stranica su $\overrightarrow{AB} = \vec{b} - \vec{a} = (b_x - a_x, b_y - a_y)$ i $\overrightarrow{AC} = \vec{c} - \vec{a} = (c_x - a_x, c_y - a_y)$, pa je površina jednaka

$$\frac{|\overrightarrow{AB} \times_{2d} \overrightarrow{AC}|}{2} = \frac{|(b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)|}{2}.$$

Izrazom $\frac{|\overrightarrow{AB} \times \overrightarrow{AC}|}{2}$ se može računati i površina trougla u prostoru (ali će tada njegovo izražavanje na osnovu koordinata tačaka biti malo komplikovanije).

5.1.3.1 Rastojanje tačke od prave

Problem

Za date tačke A, B i C u ravni, odrediti najkraće rastojanje od tačke C do prave AB.

Površina paralelograma je sa jedne strane jednaka apsolutnoj vrednosti dvodimenzionalnog vektorskog proizvoda vektora njegovih stranica, dok je sa druge strane jednaka proizvodu dužine njegove osnove i visine. Ovo se može upotrebiti da bi se izračunalo rastojanje tačke C od prave određene tačkama A i B, jer to rastojanje predstavlja visinu paralelograma određenog vektorima $\overrightarrow{AB}$ i $\overrightarrow{AC}$. Pošto je dužina osnovice jednaka $|\overrightarrow{AB}|$, visina tj. rastojanje tačke C do prave AB je jednako

$$\frac{|\overrightarrow{AB} \times_{2d} \overrightarrow{AC}|}{|\overrightarrow{AB}|} = \frac{|(b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)|}{\sqrt{(b_x - a_x)^2 + (b_y - a_y)^2}}$$

Izrazom $\frac{|\overrightarrow{AB} \times \overrightarrow{AC}|}{|\overrightarrow{AB}|}$ se može računati i rastojanje od tačke do prave u prostoru.

Problem

Za dati skup tačaka S u ravni, odrediti najkraće rastojanje od neke tačke $C \in S$ do prave AB.

Ako je potrebno odrediti najbližu ili najdalju tačku nekog skupa od prave AB, dovoljno je porediti samo vrednosti brojioca, jer će za sve tačke vrednost imenioca biti ista.

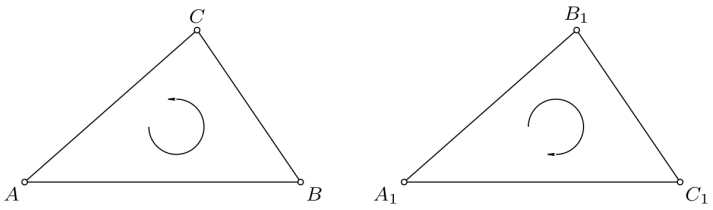
Narednom funkcijom se određuje rastojanje date tačke od date prave.

```
// rastojanje tacke (x, y) od prave odredjene tackama (ax, ay) i (bx, by)
double rastojanje(double ax, double ay, double bx, double by,
                  double x, double y) {
  double dx = bx - ax;
  double dy = by - ay;
  return abs(dx*(y-ay) - dy*(x-ax)) / sqrt(dx*dx + dy*dy);
}
```

5.1.4 Orijentacija trojke tačaka i primene

Uređena trojka nekolinearnih tačaka u ravni može imati:

- orijentaciju u smeru suprotnom od kretanja kazaljke na časovniku – matematički pozitivna orijentacija (slika 5.7 (a))
- orijentaciju u smeru kretanja kazaljke na časovniku – matematički negativna ordijentacija (slika 5.7 (b))



Slika 5.7: (a) Trougao ABC ima matematički pozitivnu orijentaciju. (b) Trougao $A_1B_1C_1$ ima matematički negativnu orijentaciju.

Važe naredna svojstva:

- ako uređena trojka ABC ima pozitivnu (negativnu) orijentaciju, onda i uređena trojka BCA ima pozitivnu (negativnu) orijentaciju (pa i trojka CAB);
- ako uređena trojka ABC ima pozitivnu (negativnu) orijentaciju, onda uređena trojka BAC ima negativnu (pozitivnu) orijentaciju (pa i trojke ACB i CBA).

Orijentacija se može zasnovati i aksiomatski[2], ali se lako može analitički izračunati kada su poznate koordinate tačaka $A(a_x, a_y)$, $B(b_x, b_y)$, $C(c_x, c_y)$. Translacijom tačke A u koordinatni početak, možemo

[2] Aksiomatsko zasnivanje orijentacije detaljno je opisano u knjizi Donada Knuta "Aksiome i omotači".

povezati orijentaciju trojke ABC sa orijentacjijom trojke $O(B-A)(C-A)$, za koju znamo da je određena znakom vektorskog proizvoda dvodimenzionalnih vektora $\overrightarrow{AB}$ i $\overrightarrow{AC}$. Dakle, uređena trojka ABC ima pozitivnu orijentaciju ako i samo ako dvodimenzionalni vektorski proizvod vektora $\overrightarrow{AB}$ i $\overrightarrow{AC}$ ima pozitivnu vrednost. Ta vrednost je jednaka:

$$\overrightarrow{AB} \times_{2d} \overrightarrow{AC} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

Pošto postoje tri moguće orijentacije, u cilju postizanja čitljivog programa moguće je definisati nabrojivi tip za predstavljanje orijentacije.

```
// moguce orijentacije trojke tacaka
enum Orijentacija {POZITIVNA, KOLINEARNE, NEGATIVNA};
```

Kada su koordinate tačaka celobrojne, funkcija za izračunavanje orijentacije se veoma jednostavno definiše.

```
// orijentacija trojke tačka sa koordinatama A(xa, ya), B(xb, yb) i C(xc, yc)
// orijentacija je pozitivna akko je orijentisani ugao ABC konveksan
Orijentacija orijentacija(int xa, int ya, int xb, int yb, int xc, int yc) {
  int d = (xb-xa)*(yc-ya) - (xc-xa)*(yb-ya);
  if (d > 0)
    return POZITIVNA;
  else if (d < 0)
    return NEGATIVNA;
  else
    return KOLINEARNE;
}
```

Stvar je komplikovanija kada se radi sa koordinatama koje su zapisane pomoću brojeva u pokretnom zarezu. Naime, klasifikovanje orijentacije, a posebno ispitivanje kolinearnosti može biti veoma problematično usled grešaka u zapisu brojeva. Naime, umesto da vrednost dvodimenzionalnog vektorskog proizvoda bude tačno nula, ona može da bude neki veoma mali broj blizak nuli. Najjednostavnije je da se tada fiksira neka vrednost ε i da se, kada god je vrednost proizvoda u intervalu $[-\varepsilon, \varepsilon]$, tačke proglase za kolinearne. Međutim, nije unapred jasno kolika bi trebalo da bude ta vrednost ε. Vrednost vektorskog proizvoda ne zavisi samo od ugla između vektora, već i od njihove dužine, pa bi precizniji pristup bio da se na osnovu vrednosti dvodimenzionalnog vektorskog proizvoda izračuna sinus ugla između vektora i da se za proveru kolinearnosti zahteva da on pripada nekom malom intervalu, međutim, to je računski intenzivnije. U nekim primenama nije previše bitno kako će se klasifikovati tačke koje su skoro kolinearne. Na primer, u računarskoj grafici, kako god da se klasifikuju tačke koje su gotovo kolinearne, ljudsko oko neće primetiti razliku.

5.1.4.1 Provera da li su tačke sa iste strane prave

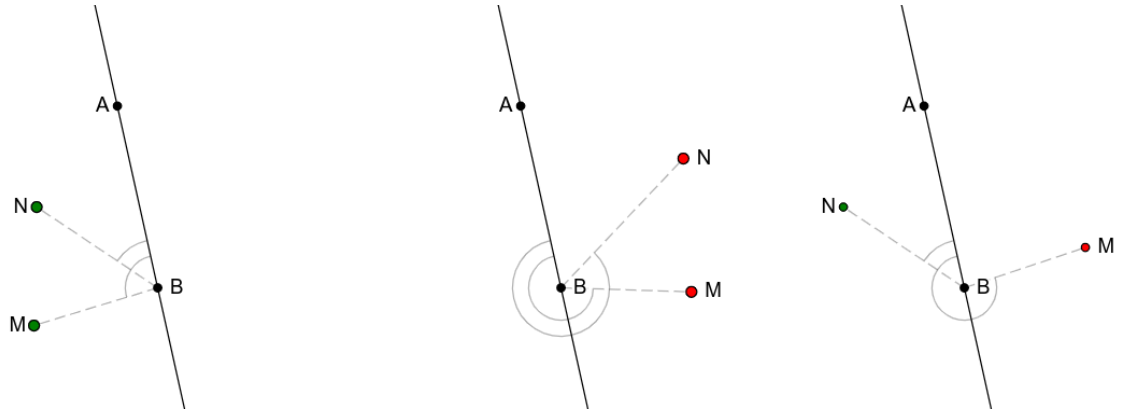
Problem

Za date tačke M i N utvrditi da li su sa iste strane prave p određene tačkama A i B ($A \neq B$).

Jednostavnosti radi pretpostavićemo da nijedna od tačaka M i N ne pripada na pravoj AB.

Tačke M i N su sa iste strane prave p ako i samo ako su trouglovi ABM i ABN, $A, B \in p$ iste orijentacije (slika 5.8). Dakle, dovoljno je da ispitamo da li su odgovarajući dvodimenzionalni vektorski proizvodi istog znaka.

Kada imamo na raspolaganju funkciju za računanje orijentacije, implementacija provere da li su dve tačke sa iste strane prave je veoma jednostavna.



Slika 5.8: U prvom slučaju tačke M i N se nalaze sa iste strane prave AB jer su obe orijentacije ABM i ABN pozitivne (pa su oba ugla ABM i ABN konveksna). U drugom slučaju su tačke ponovo sa iste strane jer obe orijentacije ABM i ABN negativne (pa su oba ta ugla nekonveksna). U trećem slučaju tačke se nalaze sa raznih strana, jer je jedna orijentacija pozitivna, a jedna negativna (pa je jedan ugao konveksan, a drugi nekonveksan).

```
// provera da li su tačke (x1, y1) i (x2, y2) sa iste strane prave
// određene tačkama (xa, ya) i (xb, yb).
// Pretpostavlja se da nijedna od dve tačke ne pripada toj pravoj.
bool saIsteStrane(int xa, int ya, int xb, int yb,
                  int x1, int y1, int x2, int y2) {
  return orijentacija(xa, ya, xb, yb, x1, y1) ==
         orijentacija(xa, ya, xb, yb, x2, y2);
}
```

Proširimo sada zahtev time da želimo da ako su tačke sa raznih strana prave odredimo i presečnu tačku duži koja ih spaja i prave.

Problem

Za date tačke M i N utvrditi da li su sa iste strane prave P određene tačkama A i B ($A \neq B$) i ako jesu, odrediti koordinate presečne tačke duži MN i prave p.

Ovaj metod jeste zasnovan na korišćenju dvodimenzionalnog vektorskog proizvoda, ali ne za izračunavanje orijentacije trojke tačaka, jer ćemo koristiti proizvode vektora oblika $\overrightarrow{XY} \times_{2d} \overrightarrow{ZW}$ (koji, podsetimo se, daju vrednost označene površine paralelograma određenog pomoću ta dva vektora).

Degenerisani slučaj nastupa kada su vektori $\overrightarrow{MN}$ i $\overrightarrow{AB}$ kolinearni, što se dešava ako i samo ako je $\overrightarrow{MN} \times_{2d} \overrightarrow{AB} = 0$. Ako su u tom slučaju tačke M, A i B kolinearne tj. ako je $\overrightarrow{MA} \times_{2d} \overrightarrow{MB} = 0$, tada duž pripada pravoj, a ako nisu kolinearne, onda presek ne postoji.

U nedegenerisanom slučaju je $\overrightarrow{MN} \times_{2d} \overrightarrow{AB} \neq 0$. Tada postoji presečna tačka X pravih AB i MN i važi da je:

$$X = M + k\overrightarrow{MN}, \quad \text{tj.} \quad \overrightarrow{MX} = k\overrightarrow{MN}.$$

Tačka X pripada duži MN ako i samo ako je $0 \leq k \leq 1$. Pošto je $\overrightarrow{MX} = \overrightarrow{MA} + \overrightarrow{AX}$, množenjem prethodne jednačinu vektorski sa $\overrightarrow{AB}$ sleva, dobija se:

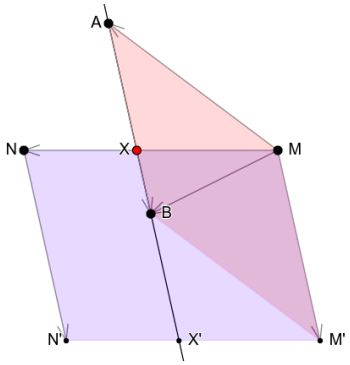
$$\overrightarrow{AB} \times_{2d} (\overrightarrow{MA} + \overrightarrow{AX}) = k \cdot (\overrightarrow{AB} \times_{2d} \overrightarrow{MN})$$

Pošto važi da je $\overrightarrow{AB} \times_{2d} \overrightarrow{AX} = 0$, jer tačka X pripada pravoj AB, pa su ti vektori kolinearni, možemo izračunati koeficijent k.

$$k = \frac{\overrightarrow{AB} \times_{2d} \overrightarrow{MA}}{\overrightarrow{AB} \times_{2d} \overrightarrow{MN}}$$

Da bi se proverilo da li tačka X pripada duži MN dovoljno je proveriti da li su brojevi $\overrightarrow{AB} \times_{2d} \overrightarrow{MA}$ i $\overrightarrow{AB} \times_{2d} \overrightarrow{MN}$ istog znaka i da li je $|\overrightarrow{AB} \times_{2d} \overrightarrow{MA}| \leq |\overrightarrow{AB} \times_{2d} \overrightarrow{MN}|$, što se sve može uraditi i u celobrojnoj aritmetici (ako su početne koordinate tačaka celobrojne).

Ova tehnika ima i jasnu geometrijsku interpretaciju (slika 5.9).



Slika 5.9: Geometerijska interpretacija određivanja koeficijenta k

Neka je $M' = M + \overrightarrow{AB}$ i $N' = N + \overrightarrow{AB}$. Vrednost $\overrightarrow{AB} \times_{2d} \overrightarrow{MA}$ predstavlja označenu površinu paralelograma određenog vektorima $\overrightarrow{AB}$ i $\overrightarrow{MA}$. Ta je površina jednaka označenoj površini paralelograma P_1 određenog vektorima $\overrightarrow{MX}$ i $\overrightarrow{XX'}$. Vrednost $\overrightarrow{AB} \times_{2d} \overrightarrow{MN}$ predstavlja označenu površinu paralelograma P_2 određenog vektorima $\overrightarrow{AB} = \overrightarrow{MM'}$ i $\overrightarrow{MN}$. Pošto paralelogrami P_1 i P_2 imaju istu visinu, odnos njihovih površina jednak je odnosu dužina njihovih stranica MX i MN.

5.1.4.2 Ispitivanje da li tačka pripada unutrašnjosti trougla

Problem

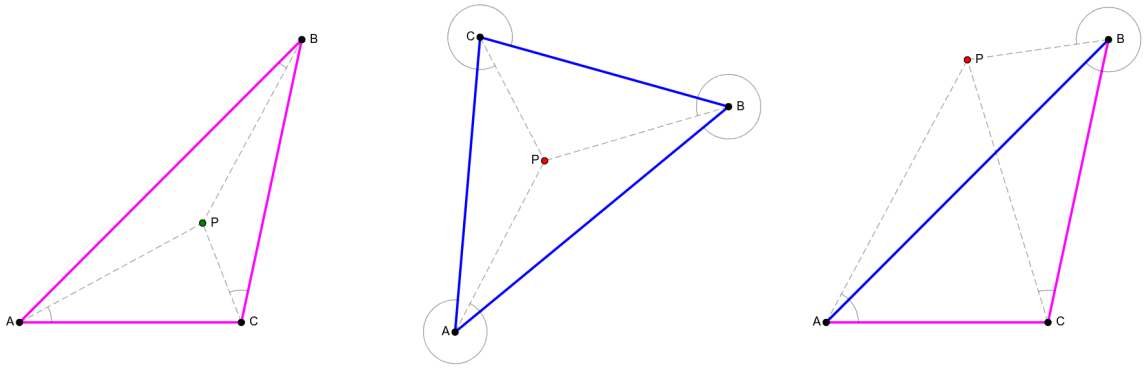
Dat je trougao ABC i tačka P. Ispitati da li tačka P pripada unutrašnjosti trougla ABC ili ne (slika 5.10).

Kao što je to obično slučaj u geometriji, treba obratiti pažnju na degenerisane i granične slučajeve. Na primer, mi ćemo, jednostavnosti radi, pretpostavljati da je trougao nedegenerisan tj. da su tačke A, B i C nekolinearne. Smatraćemo i da je trougao otvoren lik, tj. da tačke na ivicama trougla ne pripadaju njegovoj unutrašnjosti.

Tačka pripada trouglu ako i samo ako su tačke C i P sa iste strane prave AB, tačke A i P sa iste strane prave BC i tačke B i P sa iste strane prave CA. Ako to važi, tada je ista orijentacija trojki ABC i ABP, ista orijentacija trojki BCA i BCP i trojki CAB i CAP. Pošto su orijentacije trojki ABC, BCA i CAB jednake, jednake su i orijentacije trojki ABP, BCP i CAP. Pokazuje se da je ovo dovoljan uslov da bi tačka P pripadala trouglu ABC.

Obratimo pažnju da ovaj uslov ne obuhvata pripadnost ivicama trougla, što je u skladu sa pretpostavkom da je trougao otvoren lik (ako bi tačka pripadala nekoj ivici, neka orijentacija bi bila jednaka nuli i ne bi bilo moguće da su sve orijentacije jednake).

Kada imamo funkciju za izračunavanje orijentacije na raspolaganju, implementacija provere pripadnosti trouglu je veoma jednostavna.



Slika 5.10: Situacija kada tačka P pripada i kada ne pripada trouglu ABC. U prvom slučaju tačka pripada trouglu i svi sve trojke ABP, BCP i CAP su negativno orijentisane (svi uglovi AB, BCP i CAP su konveksni). U drugom slučaju tačka pripada trouglu, a sve te tri trojke su pozitivno orijentisane (svi ti uglovi su nekonveksni). U trećem slučaju tačka ne pripada trouglu i trojka ABP je pozitivno orijentisana (ugao ABP je nekonveksan), dok su trojke BCP i CAP negativno orijentisane (uglovi BCP i CAP konveksni). Dakle, tačka P pripada trouglu ABC ako i samo ako su ili sve trojke ABP, BCP i CAP isto orijentisane tj. ako su uglovi ABP, BCP i CAP konveksni ili su svi nekonveksni.

```
bool tackaUTrouglu(const Tacka& T,
                   const Tacka& A, const Tacka& B, const Tacka& C) {
  Orijentacija o1 = orijentacija(A, B, T);
  Orijentacija o2 = orijentacija(B, C, T);
  Orijentacija o3 = orijentacija(C, A, T);
  return o1 == o2 && o2 == o3;
}
```

Još jedan mogući pristup proveri da li tačka pripada je da se proveri da li je zbir površina trouglova ABT, BCT i CAT jednak površini trougla ABC. Ovaj pristup može biti računski neefikasniji (naročito ako se površine računaju preko Heronovog obrasca koji uključuje korenovanje). Dodatno, ako su tačke predstavljene realnim koordinatama, ovaj pristup može da bude numerički nestabilan. Naime, i za tačku koja je unutar trougla, može da se desi da zbir površina tri manja trougla ne bude tačno jednak, već samo blizak površini većeg trougla. Zato je jednakost potrebno proveravati do na neku vrednost ε (pri čemu ostaje osetljivo pitanje kako odrediti tu vrednost ε). Iz svih navedenih razloga, ovaj pristup je poželjno izbegavati.

```
bool tackaUTrouglu(const Tacka& T,
                   const Tacka& A, const Tacka& B, const Tacka& C) {
  // racunamo povrsinu trougla ABC
  double P = povrsinaTrougla(A, B, C);
  // racunamo povrsine trouglova TBC, TAC i TAB
  double PA = povrsinaTrougla(T, B, C);
  double PB = povrsinaTrougla(A, T, C);
  double PC = povrsinaTrougla(A, B, T);
  // proveravamo da li one u zbiru daju povrsinu trougla ABC
  // poredimo dve realne vrednosti na jednakost
  return abs(P - (PA + PB + PC)) < EPS;
}
```

5.1.4.3 Presek duži

> **Problem**
>
> *Za duži AB i CD zadate koordinatama tačaka A, B, C i D odrediti da li se seku i, ako se seku, odrediti bar jednu presečnu tačku. Pretpostaviti da su duži zatvorene tj. da sadrže svoje krajnje tačke.*

Ako su sve tačke u opštem položaju, tj. sve tačke su različite i nikoje tri nisu kolinearne, duži se seku ako i samo ako duž AB seče pravu CD i duž CD seče pravu AB, pa se ispitivanje preseka dve duži može lako svesti na ispitivanje preseka duži i prave. Duž seče pravu ako i samo ako su njene krajnje tačke sa raznih strana te prave, a to je problem koji se lako rešava pomoću ispitivanja orijentacije tačaka. Dovoljno je proveriti da li su orijentacije trojki CDA i CDB različitog znaka i da su orijentacije ABC i ABD različitog znaka tj. da im je proizvod negativan (pri čemu se za orijentaciju trojke XYZ može uzeti znak dvodimenzionalnog vektorskog proizvoda $\overrightarrow{XY} \times_{2d} \overrightarrow{XZ}$). Ovaj test se lako prilagođava slučajevima u kojima su neke tri tačke kolinearne. Naime, umesto da se zahteva da su dve tačke sa striktno različitih strana prave, moguće je uključiti i slučaj da neka od njih pripada pravoj, što se svodi na ispitivanje da li je orijentacija nepozitivna (vrednost joj je manja ili jednaka od nule).

Problem predstavlja degenerisani slučaj kada su sve četiri tačke kolinearne. Naime, ako su sve tačke kolinearne, sve orijentacije trojki biće jednake 0 i iz toga nećemo moći da izvučemo nikakav zaključak o međusobnom odnosu dve duži (znamo da one pripadaju jednoj pravoj, ali su svi njihovi međusobni odnosi mogući). U tom slučaju njihov odnos je moguće ispitati ispitivanjem njihovih projekcija na x-osu (tj. njihovih x-koordinata). Nažalost, ni ovo nije dovoljno u slučaju kada sve 4 tačke imaju istu x koordinatu, no tada možemo ispitati njihove projekcije na y-osu (tj. njihove y-koordinate). Da bi se projekcije na x-osu $[a_x, b_x]$ i $[c_x, d_x]$ sekle dovoljno je da ne važi da je jedan od intervala levo od drugog tj. ne sme da važi $b_x < c_x$ niti $d_x < a_x$, što je ekvivalentno tome da je $b_x \geq c_x \wedge d_x \geq a_x$. Prilikom primene ovoga uslova potrebno je obezbediti da važi da se zna šta je levi, a šta desni kraj intervala tj. da je $a_x \leq b_x$ i $c_x \leq d_x$. Analogno se postavlja uslov koji mora da važi i za projekcije na y-osu.

Sledi implementacija provere da li se dve duži seku. Funkcija koja izračunava orijentaciju vraća vrednosti -1, 0, 1, čime je kôd možda malo manje čitljiv, ali je omogućeno da se provera da li su dva broja istog znaka izvrši tako što se proveri da li je proizvod dve orijentacije nenegativan. Obratimo pažnju na to da umesto vrednosti vektorskog proizvoda funkcija vraća njen znak, čime se izbegava opasnost od prekoračenja prilikom množenja dve orijentacije.

```
struct Tacka {
  int x, y;
};

// 1 za pozitivne vrednosti, -1 za negativne i 0 za nulu
inline int znak(long long x) {
  return (x > 0) - (x < 0);
}

// orijentacija tj. znak vektorskog proizvoda AB x AC
inline int orijentacija(const Tacka& A, const Tacka& B, const Tacka& C) {
  return znak((B.x - A.x) * (C.y - A.y) - (C.x - A.x) * (B.y - A.y));
}

// provera da li prava AB sece duz MN
inline bool pravaSeceDuz(const Tacka& A, const Tacka& B,
                         const Tacka& M, const Tacka& N) {
  return orijentacija(A, B, M) * orijentacija(A, B, N) <= 0;
}
```

```
// provera da li se intervali realne prave [a, b] i [c, d] seku
// (pri cemu se ne zna odnos vrednosti a i b tj. c i d)
inline bool projekcijeSeSeku(long long a, long long b, long long c, long long d) {
  return max(a, b) >= min(c, d) && max(c, d) >= min(a, b);
}

// provera da li se duzi AB i CD seku
bool duziSeSeku(const Tacka& A, const Tacka& B,
                const Tacka& C, const Tacka& D) {
  return pravaSeceDuz(A, B, C, D) &&
         pravaSeceDuz(C, D, A, B) &&
         // seku se projekcije na x osu
         projekcijeSeSeku(A.x, B.x, C.x, D.x) &&
         // seku se projekcije na y osu
         projekcijeSeSeku(A.y, B.y, C.y, D.y);
}
```

Funkcija koju smo implementirali ispravno radi u svim slučajevima (i nedegenerisanim i degenerisanim), ali u mnogim slučajevima vrši neke nepotrebne provere uslova (na primer, ako nisu sve četiri tačke kolinearne, nema potrebe proveravati projekcije na ose). Uz to, nakon što se detektuje da presek postoji, potrebno je izračunati ga sasvim iznova. Prikažimo sada drugačiji metod, koji je malo efikasniji i kojim se može odmah odrediti i presečna tačka.

Na početku računamo $\overrightarrow{AB} \times_{2d} \overrightarrow{CD}$ i proveravamo da li je taj broj različit od nule. Time smo efektivno ispitali da li su vektori $\overrightarrow{AB}$ i $\overrightarrow{CD}$ kolinearni.

1. Ako je $\overrightarrow{AB} \times_{2d} \overrightarrow{CD} \neq 0$, u pitanju je nedegenerisani slučaj i ispitujemo da li postoji presečna tačka prave AB i duži CD, kao i da li postoji presečna tačka prave CD i duži AB, što možemo uraditi na način koji je opisan u poglavlju 5.1.4.1. Pošto smo vrednost $\overrightarrow{AB} \times_{2d} \overrightarrow{CD}$, već izračunali, dovoljno je da se izračunaju još samo vrednosti $\overrightarrow{AC} \times_{2d} \overrightarrow{CD}$ i $\overrightarrow{AB} \times_{2d} \overrightarrow{CA}$. Primetimo da je potrebno izračunati vrednosti samo 3 dvodimenzionalna vektorska proizvoda.
2. Ako je $\overrightarrow{AB} \times_{2d} \overrightarrow{CD} = 0$, u pitanju je nedegenerisani slučaj. Ako tačke AB i C nisu kolinearne, tj. ako je $\overrightarrow{AB} \times_{2d} \overrightarrow{AC} \neq 0$, tada su prave AB i AC paralelne i presek ne postoji. U suprotnom su sve 4 tačke kolinearne i potrebno je ispitati projekcije na x-osu, osim u slučaju kada su sve na vertikalnoj pravoj, kada možemo ispitati projekcije na y-osu.

```
// z koordinata vektorskog proizvoda vektora MN i PQ
inline long long vektorski_proizvod_2d(const Tacka& M, const Tacka& N,
                                       const Tacka& P, const Tacka& Q) {
   return (N.x - M.x)*(Q.y - P.y) - (N.y - M.y)*(Q.x - P.x);
}

// provera da li je 0 <= brojilac/imenilac <= 1
inline bool kolicnikUIntervalu01(int brojilac, int imenilac) {
  return brojilac == 0 ||
         (istogZnaka(brojilac, imenilac) && abs(brojilac) <= abs(imenilac));
}

// provera da li se intervali realne prave [a, b] i [c, d] seku
// (pri cemu se ne zna odnos vrednosti a i b tj. c i d)
inline bool projekcijeSeSeku(long long a, long long b, long long c, long long d) {
```

```
   return max(a, b) >= min(c, d) && max(c, d) >= min(a, b);
}

// provera da li se duzi AB i CD seku
bool duziSeSeku(const Tacka& A, const Tacka& B,
                const Tacka& C, const Tacka& D) {
   int ABCD = vektorski_proizvod_2d(A, B, C, D);
   if (ABCD != 0) {
      int ABCA = vektorski_proizvod_2d(A, B, C, A);
      int ACCD = vektorski_proizvod_2d(A, C, C, D);
      // presecna tacka se moze odrediti kao: C + (ABCA/ABCD)CD
      return kolicnikUIntervalu01(ACCD, ABCD) &&
             kolicnikUIntervalu01(ABCA, ABCD);
   } else {
     int ABAC = vektorski_proizvod_2d(A, B, A, C);
     if (ABAC != 0)
        return false;
     if (A.x == B.x && C.x == D.x)
        return projekcijeSeSeku(A.y, B.y, C.y, D.y);
     else
        return projekcijeSeSeku(A.x, B.x, C.x, D.x);
   }
}
```

Prethodni program je veoma jednostavno preraditi tako da izračunava i presečnu tačku duži ako ona postoji.

Treće moguće rešenje bi bilo da za tačke A i B ispitamo da li pripadaju duži CD, a za tačke C i D da li pripadaju duži AB (ako je bilo šta od ovoga zadovoljeno, duži se seku). Da bi se ispitalo da li tačka P pripada duži MN dovoljno je proveriti da li su one kolinearne (tj. da li je orijentacija MNP jednaka nuli) i da li je njena x-koordinata između x-koordinata tačaka M i N kao i da je njena y-koordinata između y-koordinata tačaka M i N.

5.1.5 Preseci horizontalnih i vertikalnih duži

Često se nailazi na probleme nalaženja preseka. Nekad je potrebno pronaći preseke više objekata, a ponekad samo treba otkriti da li je presek neprazan skup. U nastavku ćemo prikazati jedan problem nalaženja preseka, koji ilustruje važnu tehniku za rešavanje geometrijskih problema. Ista tehnika može se primeniti i na druge slične probleme.

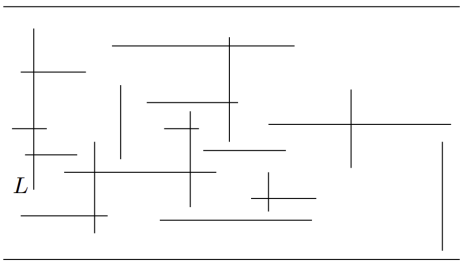
U prethodnom izlaganju fokus je bio na tome da se odredi presek duži koje se nalaze u opštem položaju. Videli smo da težina rešenja tog problema leži kako u matematičkom aparatu, tako u obradi velikog broja specijalnih slučajeva. U nastavku ćemo razmotriti problem u kom su duži uvek samo horizontalne i vertikalne, pa je za dve duži veoma jednostavno ispitati postojanje preseka, međutim, tih duži će biti puno i potrebno je upotrebiti geometrijske osobine (poredak koordinata) da bi se smanjio broj ispitivanja preseka.

Problem

Za zadati skup od n horizontalnih i m vertikalnih duži pronaći sve njihove preseke.

Ovaj problem važan je, na primer, pri projektovanju kola VLSI (integrisanih kola sa ogromnim brojem elemenata). Kolo može da sadrži na hiljade "žičica", a projektant treba da bude siguran da ne postoje neočekivani preseci. Na problem se takođe nailazi pri eliminaciji skrivenih linija kada je potrebno zaključiti koje stranice ili delovi stranica su skriveni samim objektom ili nekim drugim objektom; taj problem je obično

komplikovaniji, jer se ne radi samo o horizontalnim i vertikalnim linijama. Primer ulaza za ovaj problem prikazan je na slici 5.11.



Slika 5.11: Preseci horizontalnih i vertikalnih duži.

Pretpostavimo zbog jednostavnosti da ne postoje preseci između proizvoljne dve horizontalne, odnosno proizvoljne dve vertikalne duži.

Ako pokušamo da problem rešimo dodavanjem jedne po jedne duži (bilo horizontalne, bilo vertikalne), onda će biti neophodno da se pronađu preseci nove duži sa svim ostalim dužima, pa se dobija algoritam sa $O(mn)$ nalaženja preseka duži (što je zapravo algoritam grube sile). U opštem slučaju broj preseka može da bude $O(mn)$, pa algoritam može da utroši vreme $O(mn)$ već samo za prikazivanje svih preseka. Međutim, broj preseka može da bude mnogo manji od mn. Voleli bismo da konstruišemo algoritam koji radi dobro kad ima malo preseka, a ne previše loše ako preseka ima mnogo. Dakle cilj nam je da problem rešimo korišćenjem algoritma čija složenost zavisi i od veličine ulaza i od veličine izlaza (tzv. algoritma sa izlazno zavisnom složenošću). To se može postići kombinovanjem dveju ideja: izbora specijalnog redosleda indukcije i pojačavanja induktivne hipoteze.

Redosled primene indukcije može se odrediti *brišućom pravom* (engl. sweeping line) koja prelazi ("skenira") ravan sleva udesno; duži se razmatraju onim redom kojim na njih nailazi pokretna (brišuća) prava. Značajni *događaji* (engl. events) su kada prava naiđe na neku karakterističnu tačku i algoritam se izvršava tako što se ti događaji redom obrađuju. Pored nalaženja presečnih tačaka, treba čuvati i neke podatke o (horizontalnim) dužima koje je brišuća prava već zahvatila. Ti podaci biće korisni za efikasnije nalaženje narednih preseka.

Zamislimo vertikalnu pravu koja prelazi ravan sleva udesno i obradimo duži u redosledu u kojem prava nailazi na njih. Da bismo implementirali ovaj redosled, sortiramo sve krajeve duži prema njihovim x-koordinatama. Dve krajnje tačke vertikalne duži imaju iste x-koordinate, pa se registruje samo jedna x-koordinata. Za horizontalne duži moraju se koristiti oba kraja. Posle sortiranja krajeva, duži se razmatraju jedna po jedna utvrđenim redosledom. Kao i obično pri induktivnom pristupu, pretpostavljamo da smo pronašli presečne tačke prethodnih duži, i da smo obezbedili neke dopunske informacije, pa sada pokušavamo da obradimo sledeću duž i da unesemo neophodne dopune informacija. Prema tome struktura algoritma je u osnovi sledeća. Razmatramo krajeve duži jedan po jedan, sleva udesno. Koristimo informacije prikupljene do sada (nismo ih još specificirali) da obradimo kraj duži, pronađemo preseke u kojima ona učestvuje, i dopunjujemo informacije da bismo ih koristili pri sledećem nailasku na neki kraj duži. Osnovni problem je definisanje informacija koje treba prikupljati. Pokušajmo da preciziramo algoritam da bismo otkrili koje su to informacije potrebne.

Prirodno je u induktivnu hipotezu uključiti poznavanje svih presečnih tačaka duži koje se nalaze levo od trenutnog položaja pokretne prave. Da li je bolje proveravati preseke kad se obrađuje horizontalna ili vertikalna duž? Kad obrađujemo vertikalnu duž, horizontalne duži koje je mogu seći se još uvek moraju pamtiti (pošto nije dostignut njihov desni kraj). S druge strane, kad posmatramo bilo levi, bilo desni kraj horizontalne duži, mi ili još nismo naišli na vertikalne duži koje je seku, ili smo ih već obradili. Dakle, bolje je preseke brojati prilikom nailaska na vertikalnu duž.

Pretpostavimo da je pokretna prava trenutno preklopila vertikalnu duž L (videti sliku 5.11). Kakve informacije su potrebne da se pronađu svi preseci u kojima učestvuje duž L? Pošto se pretpostavlja da su svi

preseci levo od trenutnog položaja pokretne prave već poznati, nema potrebe razmatrati horizontalnu duž ako je njen desni kraj levo od pokretne prave. Prema tome, treba razmatrati samo one horizontalne duži čiji su levi krajevi levo, a desni krajevi desno od trenutnog položaja pokretne prave (na slici 5.11 takvih duži ima šest). Potrebno je čuvati listu takvih horizontalnih duži. Kad se naiđe na vertikalnu duž L, potrebno je i dovoljno proveriti da li se ona seče sa tim horizontalnim dužima. Važno je primetiti da za nalaženje preseka sa L ovde x-koordinate krajeva horizontalnih duži nisu od značaja, pa je dovoljno čuvati semo listu njihovih y-koordinata. Naime, mi već znamo da horizontalne duži iz liste imaju x-koordinate koje "pokrivaju" x-koordinatu duži L. Potrebno je proveriti samo y-koordinate horizontalnih duži iz liste, da bi se proverilo da li su one obuhvaćene opsegom y-koordinata duži L. Sada smo spremni da formulišemo induktivnu hipotezu.

Induktivna hipoteza: Neka je zadata lista od prvih k sortiranih x-koordinata krajeva duži kao što je opisano, pri čemu je x_k najveća od tih x-koordinata. Umemo da pronađemo sve preseke duži koji su levo od x_k, i da eliminišemo sve horizontalne duži koje su levo od x_k.

Za horizontalne duži koje se još uvek razmatraju reći ćemo da su kandidati (to su horizontalne duži čiji su levi krajevi levo, a desni krajevi desno od tekućeg položaja pokretne prave). Formiraćemo i održavati strukturu podataka koja sadrži skup kandidata. Odložićemo za trenutak analizu realizacije ove strukture podataka.

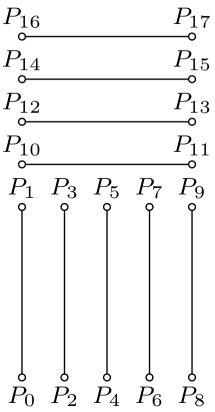
Bazni slučaj $k = 1$ za navedenu induktivnu hipotezu je jednostavan. Da bismo je proširili, potrebno je da obradimo $(k + 1)$-i kraj duži. Postoje tri mogućnosti tj. tri vrste događaja koje obrađujemo.

1. Naišli smo na desni kraj horizontalne duži; duž se tada prosto eliminiše iz spiska kandidata. Kao što je rečeno, preseci se pronalaze pri razmatranju vertikalnih duži, pa se ni jedan od preseka ne gubi eliminacijom horizontalne duži. Ovaj korak dakle proširuje induktivnu hipotezu.

2. Naišli smo na levi kraj horizontalne duži; duž se tada dodaje u spisak kandidata. Pošto desni kraj duži nije dostignut, duž se ne sme još eliminisati, pa je i u ovom slučaju induktivna hipoteza proširena na ispravan način.

3. Naišli smo na vertikalnu duž tj. na neko njeno teme. Preseci sa ovom vertikalnom duži mogu se pronaći upoređivanjem y-koordinata svih horizontalnih duži iz skupa kandidata sa y-koordinatama krajeva vertikalne duži.

Algoritam je sada kompletan. Broj upoređivanja će obično biti mnogo manji od mn, međutim, u najgorem slučaju ovaj algoritam ipak zahteva $O(mn)$ upoređivanja, čak i kad je broj preseka mali. Ako se, na primer, sve horizontalne duži prostiru sleva udesno ("celom širinom"), onda se mora proveriti presek svake vertikalne duži sa svim horizontalnim dužima, što implicira složenost $O(mn)$. Ovaj najgori slučaj pojavljuje se čak i ako nijedna vertikalna duž ne seče nijednu horizontalnu duž (videti primer na slici 5.12).



Slika 5.12: Slučaj kada nema preseka duži, a broj poređenja u okviru osnovnog algoritma je $O(mn)$.

Da bi se algoritam usavršio, potrebno je smanjiti broj upoređivanja y-koordinata vertikalne duži sa y-koordinatama horizontalnih duži u skupu kandidata. Neka su y-koordinate vertikalne duži koja se trenutno razmatra y_D i y_G, i neka su y-koordinate horizontalnih duži iz skupa kandidata $y_0, y_1, \dots, y_{r-1}$. Ako pretpostavimo da su horizontalne duži u skupu kandidata sortirane prema y-koordinatama (odnosno niz $y_0, y_1, \dots, y_{r-1}$ je rastući), preseci se mogu pronaći izvođenjem dve binarne pretrage, jedne za y_D, a druge za y_G. Neka je $y_i < y_D \leq y_{i+1} \leq y_j \leq y_G < y_{j+1}$. Vertikalnu duž seku horizontalne duži sa koordinatama $y_{i+1}, y_{i+2}, \dots, y_j$, i samo one, pa je njihov broj $j - i$. Ako je potrebno da se navedu svi preseci, tada se može izvršiti jedna binarna pretraga za y_D, a zatim prolaziti y-koordinate, dok se ne dođe do vrednosti y_{j+1} veće od y_G.

Iako je polazni problem dvodimenzionalan, nalaženje y_{i+1}, $\dots y_j$ je jednodimenzionalni problem. Traženje brojeva u jednodimenzionalnom opsegu (u ovom slučaju od y_D do y_G) zove se *jednodimenzionalna pretraga opsega*. Ako su brojevi sortirani, onda je vreme izvršenja jednodimenzionalne pretrage opsega proporcionalno zbiru vremena traženja prvog preseka i broja pronađenih preseka. Naravno, ne možemo da priuštimo sebi sortiranje horizontalnih duži pri svakom nailasku na vertikalnu duž, već je potrebno da duži čuvamo u nekoj pogodnoj strukturi podataka.

Prisetimo se još jednom zahteva. Potrebna je struktura podataka pogodna za čuvanje kandidata, koja dozvoljava umetanje novog elementa, brisanje elementa i efikasno izvršenje jednodimenzionalne pretrage opsega. Na sreću, postoji više struktura podataka koje omogućuju izvršenje umetanja, brisanja i traženja elemenata složenosti $O(\log n)$ po operaciji (gde je n broj elemenata u strukturi podataka), i linearno pretraživanje za vreme proporcionalno broju pronađenih elemenata — na primer, skup implementiran pomoću uravnoteženog binarnog drveta koji je dostupan u standadnoj biblioteci većine savremeniih programskih jezika (u C++-u to je struktura podataka `set`).

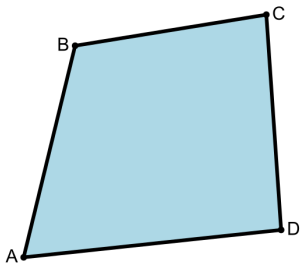
5.2 Mnogouglovi

Objekti koji se najčešće razmatraju u računarskoj geometriji su tačke, prave i duži i mnogouglovi. Prilikom računarske obrade svi objekti se predstavljaju analitički.

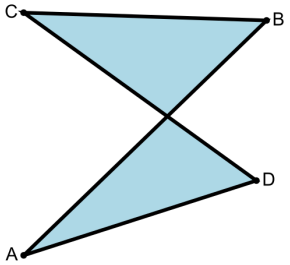
- *Prava* je najčešće predstavljena parom različitih tačaka P i Q koje joj pripadaju.
- *Duž* se najčešće zadaje parom tačaka P i Q koje predstavljaju krajeve te duži, i označavaćemo je sa PQ.
- *Put* P je niz tačaka $P_1, P_2, \dots, P_k$ i duži $P_1P_2, P_2P_3, \dots, P_{k-1}P_k$ koje ih povezuju. Duži koje čine put su njegove *stranice* (*ivice*). Primetimo da je u opštem slučaju dopušteno da tačke na putu budu i kolinearne.
- *Zatvoreni put* je put čija se poslednja tačka poklapa sa prvom i nazivamo ga *mnogougao* ili *poligon*. Tačke koje definišu mnogougao su njegova *temena*.

  Na primer, trougao je mnogougao sa tri temena. Mnogougao se predstavlja nizom, a ne skupom tačaka, jer je bitan redosled kojim se tačke zadaju; promenom redosleda tačaka iz istog skupa u opštem slučaju dobija se drugi mnogougao.
- *Prost mnogougao* je onaj kod koga odgovarajući put nema preseka sa samim sobom; drugim rečima, jedine stranice koje imaju zajedničke tačke su susedne stranice sa svojim zajedničkim temenom. Prost mnogougao deli ravan na dve oblasti: *unutrašnjost* i *spoljašnost*. Na slici 5.13 prikazan je jedan prost mnogougao, dok je na slici 5.14 prikazan jedan mnogougao koji nije prost (tj. mnogougao koji je samopresecajući).

  Ako ne postoji mogućnost zabune, nekada se termin *mnogougao* može upotrebiti i da označi *mnogougaonu površ* tj. skup tačaka unutrašnjosti mnogougla. Ako je mnogougao *zatvoren* smatra se da mu pored tačaka u unutrašnjosti pripadaju i tačke na ivicama.

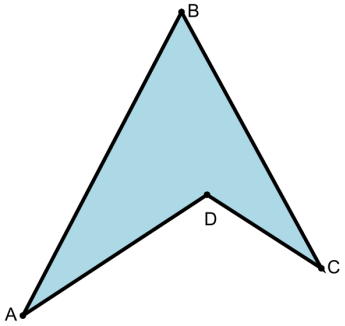


Slika 5.13: Prost mnogougao



Slika 5.14: Samopresecajući mnogougao

- *Konveksni mnogougao* je mnogougao čija unutrašnjost sa svake dve tačke koje sadrži, sadrži i sve tačke duži koje te tačke određuju (ekvivalentno, mnogougao je konveksan ako su mu svi unutrašnji uglovi manji ili jednaki 180° tj. π radijana). *Konveksni put* je put sastavljen od tačaka $P_1, P_2, \ldots, P_k$ takav da je mnogougao $P_1P_2 \ldots P_k$ konveksan. Mnogougao na slici 5.13 je konveksan, dok je mnogougao na slici 5.15 nekonveksan.



Slika 5.15: Nekonveksan mnogougao

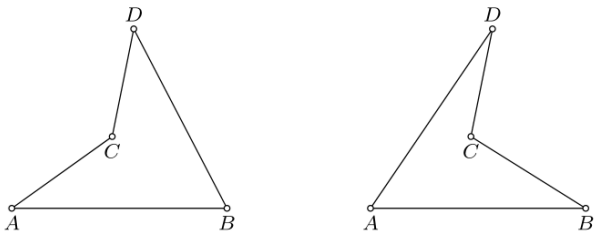
5.2.1 Konstrukcija prostog mnogougla

Skup tačaka u ravni definiše veći broj različitih mnogouglova, zavisno od izabranog redosleda tačaka. Razmotrićemo sada pronalaženje prostog mnogougla sa zadatim skupom temena.

Problem

Dato je n različitih tačaka u ravni, takvih da nisu sve kolinearne. Povezati ih prostim mnogouglom.

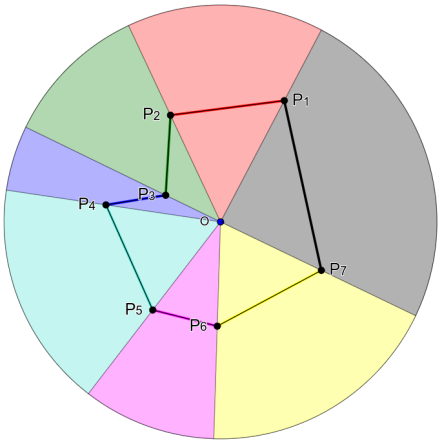
Postoji više metoda za konstrukciju traženog prostog mnogougla; uostalom, jasno je da u opštem slučaju problem nema jednoznačno rešenje (slika 5.16).



Slika 5.16: Primer dva različita prosta mnogougla sa istim temenima

Prirodna ideja je da se tačke obilaze nekako u krug. Neka je C neki krug, čija unutrašnjost sadrži sve tačke. Površina C može se "prebrisati" (pregledati) rotirajućom polupravom kojoj je početak centar kruga C i tačke se mogu povezati u mnogougao u redosledu u kojem poluprava na njih nailazi.

Očekujemo da ćemo spajanjem tačaka onim redom kojim poluprava nailazi na njih dobiti prost mnogougao. Pokušajmo da to dokažemo. Pretpostavimo za trenutak da rotirajuća poluprava u svakom trenutku sadrži najviše jednu tačku. Označimo tačke, uređene u skladu sa redosledom nailaska poluprave na njih, sa $P_1, P_2, \ldots, P_n$ (prva tačka bira se proizvoljno). Poluprave koje spajaju tačku O sa ovim tačkama dele krug na n međusobno disjunktnih kružnih isečaka. Temena svake duži P_iP_{i+1} pripadaju ivicama tih isečaka. Ako su svi isečci konveksni (ako im je centralni ugao manji od π), tada je to što temena duži P_iP_{i+1} pripadaju isečku dovoljno da bi moglo da se garantuje da cela duž pripada tom isečku. Dakle, ako bi svi isečci bili konveksni, dobijeni mnogougao bi morao da bude prost (što je i slučaj na slici 5.17).

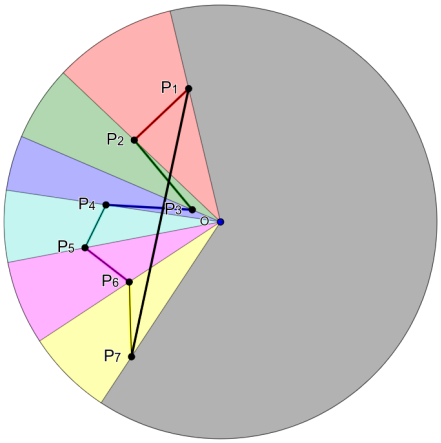


Slika 5.17: Prost mnogougao u krugu

Međutim, ugao između polupravih kroz neke dve uzastopne tačke P_i i P_{i+1} može da bude veći od π. Tada isečak koji sadrži duž P_iP_{i+1} sadrži više od pola kruga i nije konveksna figura, a duž P_iP_{i+1} ne pripada tom isečku, već prolazi kroz druge isečke kruga, pa može da seče druge stranice mnogougla.

Primer 5.2.1

U primeru ilustrovanom na slici 5.18 duž P_1P_7 prolazi kroz druge isečke kruga i seče stranice P_2P_3 i P_3P_4.



Slika 5.18: Loš izbor centra kruga koji dovodi do toga da konstruisani mnogougao nije prost.

Da bi se uočeni problem rešio, krug je potrebno odabrati tako da ne postoji prava koja prolazi kroz centar tako da se sve tačke nalaze sa iste strane te prave. Jedna prirodna ideja je da se tačka bira tako bude "na sredini" datog skupa tačaka. Mogu se, na primer, fiksirati proizvoljne tri tačke iz skupa, a za centar kruga izabrati neka tačka O unutar njima određenog trougla (na primer težište, koje se lako nalazi). Ovakav izbor garantuje da ni jedan od dobijenih sektora kruga neće imati ugao veći od π.

Tačke sortiramo prema položaju u krugu sa centrom O. Preciznije, tačke P_i se sortiraju na osnovu veličine *ugla* između x-ose i poluprave OP_i. Ako dve ili više tačaka imaju isti ugao, one se dalje sortiraju rastuće prema rastojanju od tačke O. Na kraju, tačke povezujemo u skladu sa dobijenim uređenjem, po dve uzastopne. Osnovna komponenta vremenske složenosti ovog algoritma potiče od sortiranja tačaka, te je složenost algoritma $O(n \log n)$.

Ugao ϕ koji prava OP zahvata sa x osom jednak je vrednosti atan2$(P_y - O_y, P_x - O_x)$. Primetimo i da kad dve tačke imaju isti nagib nije neophodno računati njihova rastojanja od tačke O — dovoljno je izračunati kvadrate rastojanja. Dakle, nema potrebe za izračunavanjem kvadratnih korenova.

Prethodni algoritam se može implementirati na sledeći način:

```
double ugao(double x0, double y0, double x, double y) {
  // ako su tacke t0=(x0, y0) i t=(x, y) racunamo
  // ugao koji vektor t0t zaklapa sa negativnim smerom x ose
  return atan2(y - y0, x - x0);
}

bool kolinearne(const Tacka& t1, const Tacka& t2, const Tacka& t3) {
  return abs(ugao(t1.x, t1.y, t2.x, t2.y) - ugao(t1.x, t1.y, t3.x, t3.y)) < EPS;
}

void prostMnogougao(vector<Tacka>& tacke) {
  // pronalazimo centar kruga kao teziste neke 3 nekolinearne tacke
  int i = 2;
  while (kolinearne(tacke[0], tacke[1], tacke[i]))
    i++;
  double x0 = (tacke[0].x + tacke[1].x + tacke[i].x) / 3.0;
  double y0 = (tacke[0].y + tacke[1].y + tacke[i].y) / 3.0;
```

```
  // sortiramo tacke na osnovu ugla
  sort(begin(tacke), end(tacke),
       [x0, y0](const Tacka& t1, const Tacka& t2) {
         // ugao koji vektor t0t1 zaklapa sa negativnim smerom x ose
         double ugao1 = ugao(x0, y0, t1.x, t1.y);
         // ugao koji vektor t0t2 zaklapa sa negativnim smerom x ose
         double ugao2 = ugao(x0, y0, t2.x, t2.y);

         if (ugao1 < ugao2 - EPS)
           return true;

         if (ugao2 < ugao1 - EPS)
           return false;

         // razlika uglova je u intervalu [-EPS, EPS] pa tacke smatramo kolinearnim
         return kvadratRastojanja(x0, y0, t1.x, t1.y) <
                kvadratRastojanja(x0, y0, t2.x, t2.y);
       });
  // obrcemo redosled tacaka na poslednjoj pravoj
  auto it = prev(end(tacke));
  while (kolinearne(*prev(it), *it, tacke[0]))
    it = prev(it);
  reverse(it, end(tacke));
}
```

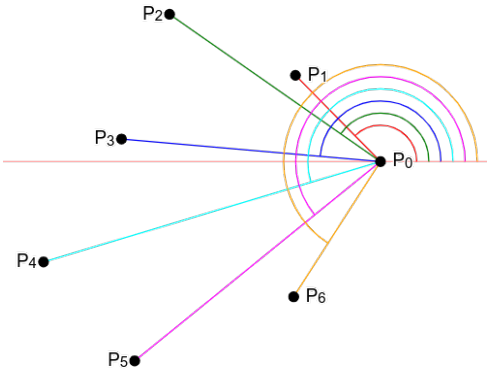
Umesto težišta trougla određenog sa neke tri nekolinearne tačke iz skupa, za centar kruga O može se uzeti i jedna *ekstremna tačka* tačka iz skupa – na primer, tačka sa najvećom x-koordinatom (i sa najmanjom y-koordinatom, ako ima više tačaka sa najvećom x-koordinatom)[3]. Ovakve tačke se često koriste prilikom rešavanja geometrijskih problema. Ekstremnu tačku povezujemo sa svim ostalim tačkama datog skupa i tačke sortiramo rastuće u odnosu na ugao koji poluprava od tačke O kroz tu tačku zahvata sa polupravom iz temena O koja je paralelna sa x-osom (a taj je ugao isti kao ugao koji poluprava od tačke O kroz tu tačku zahvata sa x-osom, jer su to uglovi sa paralelnim kracima). Tačke označene u ovom redosledu su prikazane na slici 5.19.

Pošto sve tačke leže levo od tačke O, ugao između polupravih kroz dve uzastopne tačke ne može biti veći od π, te do degenerisanog slučaja o kome je bilo reči ne može doći. Ako dve ili više tačaka zahvataju isti ugao, one se dalje sortiraju prema rastojanju od tačke O i to na sledeći način: ukoliko prvih nekoliko tačaka zahvataju isti ugao, njih sortiramo u rastućem redosledu rastojanja od tačke O, ukoliko je poslednjih dve ili više tačaka kolinearno sa tačkom O njih sortiramo u opadajućem redosledu rastojanja od tačke O, dok je za ostale tačke koje su kolinearne sa tačkom O svejedno da li ćemo ih sortirati rastuće ili opadajuće. Primetimo i to da je umesto računanja uglova moguće razmatrati orijentaciju tačaka. Naime, za potrebe sortiranja skupa tačaka prema uglu koji zahvataju sa horizontalnom pravom kroz tačku O iskoristićemo činjenicu da tačka P_i zahvata manji ugao od tačke P_j ako je orijentacija trojke tačaka P_j, O, P_i pozitivna (slika 5.20).

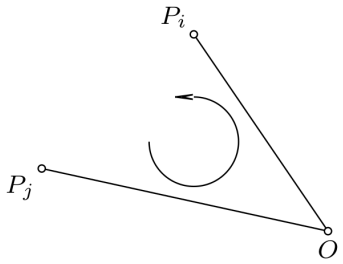
Prethodni algoritam se može implementirati na sledeći način:

```
void prostMnogougao(vector<Tacka>& tacke) {
  // trazimo tacku sa maksimalnom x koordinatom,
  // u slucaju da ima vise tacaka sa maksimalnom x koordinatom
```

[3] Naravno, pošto je problem simetričan i duž horizontalne i duž vertikalne ose, bilo koja druga kombinacija uslova minimalnosti i maksimalnosti koordinata je u redu.



Slika 5.19: Ekstremna tačka $O = P_0$ i tačke sortirane prema uglu koji prava P_0P_i zahvata sa pozitivnim smerom x-ose.



Slika 5.20: Kada se P_i i P_j nalaze levo od O, ugao između x-ose i prave OP_i je manji od ugla između x-ose i prave OP_j ako i samo ako je trojka P_jOP_i pozitivno orijentisana (isto kao i OP_iP_j).

```
  // biramo onu sa najmanjom y koordinatom
  auto max = max_element(begin(tacke), end(tacke),
                         [](const Tacka& t1, const Tacka& t2) {
                           return t1.x < t2.x ||
                                  (t1.x == t2.x && t1.y > t2.y);
                         });
  // dovodimo je na početak niza - ona predstavlja centar kruga
  swap(*begin(tacke), *max);
  const Tacka& t0 = tacke[0];

  // sortiramo ostatak niza (tačke sortiramo na osnovu ugla koji
  // zaklapaju u odnosu vertikalnu polupravu koja polazi naviše iz
  // centra kruga), a kolinearne na osnovu rastojanja od centra kruga
  sort(next(begin(tacke)), end(tacke),
       [&t0](const Tacka& t1, const Tacka& t2) {
         Orijentacija o = orijentacija(t0, t1, t2);
         if (o == KOLINEARNE)
           return kvadratRastojanja(t0, t1) <= kvadratRastojanja(t0, t2);
         return o == POZITIVNA;
       });

  // obrcemo redosled tacaka na poslednjoj pravoj
  auto it = prev(end(tacke));
  while (orijentacija(*prev(it), *it, t0) == KOLINEARNE)
    it = prev(it);
  reverse(it, end(tacke));
}
```

5.2.2 Ispitivanje da li tačka pripada unutrašnjosti prostog mnogougla

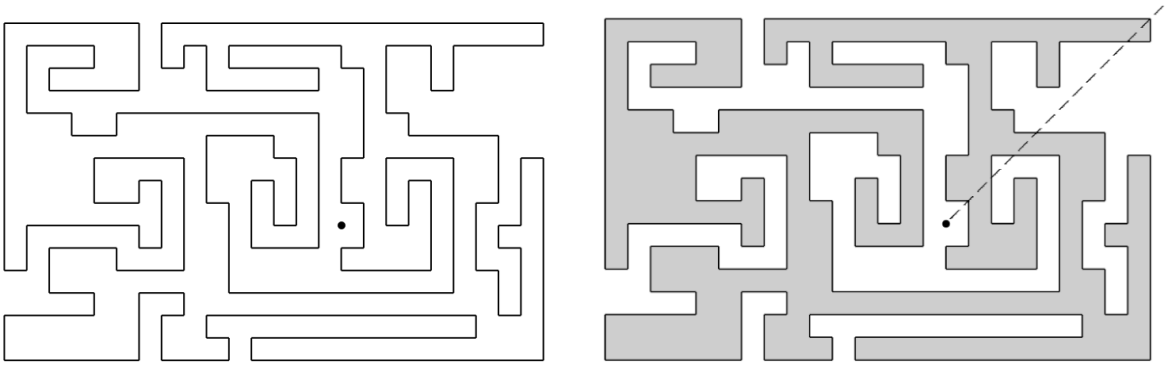
Zadatak geokodiranja jeste proces koji za zadato mesto ili adresu vraća njegove geografske koordinate. Jednako važan zadatak je i njemu inverzan, kojim se za date koordinate određuje mesto – grad, država ili neka oblast kojoj tačka sa tim koordinatama pripada. Ovaj problem se svodi na ispitivanje da li tačka sa datim koordinatama pripada unutrašnjosti mnogougla koji ograničava neku oblast.

U prethodnom odeljku razmotrili smo na koji način možemo utvrditi da li se tačka nalazi unutar datog trougla. Razmotrimo kako se ovo može uopštiti na proizvoljni prost mnogougao.

> **Problem**
>
> *Zadat je prost mnogougao $P_1 \ldots P_n$ i tačka Q, koja ne leži na nekoj od njegovih ivica. Ustanoviti da li je tačka Q unutar tog mnogougla ili van njega.*

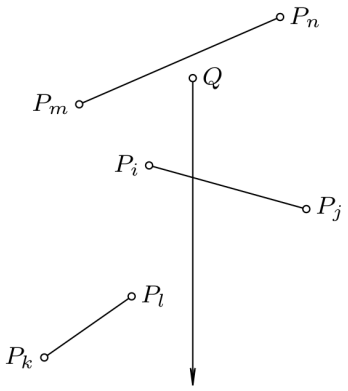
Problem na prvi pogled izgleda jednostavno, ali ako se razmatraju složeni nekonveksni mnogouglovi, kao onaj prikazan na slici 5.21, vidi se da to nije uvek slučaj. Intuitivni pristup je pokušati nekako "izaći napolje", polazeći od zadate tačke Q. Posmatrajmo proizvoljnu polupravu sa temenom Q. Vidi se da je dovoljno prebrojati preseke sa stranicama mnogougla. U primeru na slici 5.21, idući na severoistok[4] od date tačke (prateći isprekidanu liniju), nailazimo na šest preseka sa mnogouglom. Pošto nas poslednji presek pre izlaska izvodi iz mnogougla, a pretposlednji nas vraća u mnogougao, i tako dalje, tačka Q je van mnogougla. Dakle, možemo zaključiti da je tačka Q u mnogouglu ako i samo ako je broj preseka poluprave iz Q sa stranicama mnogougla neparan.

[4] Radi jednostavnijeg objašnjenja algoritma, pretpostavljamo da se u Dekartovom koordinatom sistemu zna šta je gore, dole, desno, levo, tj. gde su sever, jug, istok i zapad.



Slika 5.21: Utvrđivanje pripadnosti tačke unutrašnjosti prostog mnogougla. Na slici levo deluje prilično nejasno da li je tačka unutra ili napolju. Tek kada se unutrašnjost mnogougla oboji, postaje jasno da je tačka van mnogougla, što se može utvrditi i brojanjem preseka poluprave sa iviciama tog mnogougla.

Razmotrimo kako se ovaj algoritam može implementirati ako je mnogougao zadat nizom koordinata temena, što je uobičajeni scenario. Kada se problem rešava vizuelno, lako je naći dobar put (polupravu sa malo preseka) i nije potrebno razmatrati ivice koje su daleko od tog puta. U slučaju mnogougla datog nizom koordinata, to nije jednostavno i najveći deo vremena troši se na ispitivanje postojanja preseka poluprave i stranica mnogougla, jer se ne vidi način da se izbegne obrada svih stranica. Ispitivanje preseka ivice i poluprave se može uprostiti ako je poluprava paralelna jednoj od osa — na primer y-osi i recimo usmerena nadole (kao na slici 5.22).



Slika 5.22: Različiti položaji stranica mnogougla i poluprave sa temenom u tački Q usmerene nadole: poluprava sa početkom u tački Q seče stranicu P_iP_j, a ne seče stranice P_kP_l i stranicu P_mP_n (iako se u sva tri slučaja prave seku, u slučaju P_kP_l ta presečna tačka je van segmenta, a u slučaju P_mP_n ta tačka je van poluprave).

Naime, presek vertikalne poluprave kroz tačku Q sa stranicom P_iP_{i+1} mnogougla postoji ako se x-koordinata tačke Q nalazi između vrednosti x-koordinata temena P_i i P_{i+1} i ako je y-koordinata presečne tačke poluprave kroz Q i prave P_iP_{i+1} manja od y-koordinate tačke Q. Vrednost y koordinate presečne tačke dobićemo kao presek prave $x = x_Q$ i prave P_iP_{i+1} čija je jednačina $y - y_{P_i} = \frac{y_{P_{i+1}} - y_{P_i}}{x_{P_{i+1}} - x_{P_i}}(x - x_{P_i})$ je tačka čija je x-koordinata x_Q, a y-koordinata je jednaka:

$$y = y_{P_i} + \frac{y_{P_{i+1}} - y_{P_i}}{x_{P_{i+1}} - x_{P_i}}(x_Q - x_{P_i}).$$

Dakle, potrebno je da važi $y \leq y_Q$, što se, kada se izraz transformiše da bi se izbeglo deljenje, svodi na:

$$y_{P_i}(x_{P_{i+1}} - x_{P_i}) + (y_{P_{i+1}} - y_{P_i})(x_Q - x_{P_i}) \leq y_Q(x_{P_{i+1}} - x_{P_i})$$

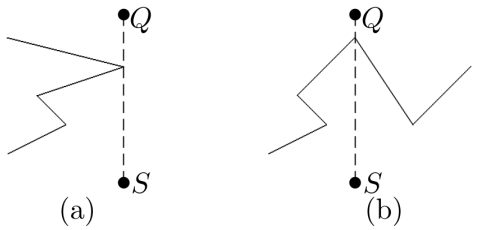
tj.

$$(y_{P_i} - y_Q)(x_{P_{i+1}} - x_{P_i}) - (y_{P_{i+1}} - y_{P_i})(x_{P_i} - x_Q) \leq 0$$

Do ovog izraza se može doći i ako se proveri orijentacija trojke tačaka $P_iP_{i+1}Q$.

Kao što je već pomenuto, obično postoje neki specijalni slučajevi koje treba posebno razmotriti. Neka je S tačka van mnogougla i pretpostavimo da umesto poluprave iz Q razmatramo duž QS. Cilj je utvrditi da li tačka Q pripada unutrašnjosti mnogougla P na osnovu broja preseka duži QS sa stranicama mnogougla P. Prvi specijalni slučaj je kada duž QS sadrži neko teme P_i mnogougla. Na slici 5.23 (a) prikazan je slučaj kad presek duži QS u temenu ne treba brojati, a na slici 5.23 (b) slučaj kad taj presek treba brojati. Postoji više načina za utvrđivanje da li neki ovakav presek treba brojati ili ne. Navedimo dva:

- ako su dva temena mnogougla susedna temenu P_i sa iste strane prave kojoj pripada duž QS, presek se ne računa (oba susedna temena na slici 5.23 (a) su sa leve strane duži QS). U protivnom, ako su temena susedna temenu P_i sa različitih strana ove prave, presek se broji (na 5.23 (b) jedno susedno teme je sa leve, a drugo sa desne strane duži QS). Ovo u većini slučajeva rešava problem, međutim, ostaje problematičan slučaj kada je neka od te dve tačke leži na pravoj QS.
- s obzirom na to da razmatramo pravu koja je paralelna sa y osom, za svaku stranicu mnogougla trebalo bi računati preseke sa levim temenom, a ne sa desnim (temena neke stranice mnogougla klasifikujemo kao levo i desno u odnosu na vrednosti x-koordinate temena). Na taj način nećemo računati presek nalik onom na slici 5.23 (a) jer je za obe stranice mnogougla susedne temenu P_i presek kroz desno teme (i slično dva puta bismo računali presek sa temenom koje je levi ekstremum i opet se ne bi promenila parnost brojača). Za presek prikazan na slici 5.23 (b) ubrajamo jedan presek, jer je za jednu od stranica koje se sustiču u tom temenu to levo teme, a za drugu desno.



Slika 5.23: Specijalni slučajevi kad vertikalna poluprava sa početkom u tački Q sadrži neko teme mnogougla.

Drugi specijalan slučaj se javlja kada se duž QS delom preklapa sa nekim stranicama mnogougla P. Ovo preklapanje očigledno ne treba brojati u preseke.

U nastavku ćemo razmotriti implementaciju algoritma za ispitivanje da li je tačka u mnogouglu koja ne uključuje obradu specijalnih slučajeva.

Neka je n broj stranica mnogougla. Kroz osnovnu petlju algoritma prolazi se n puta. U svakom prolasku kroz petlju računa se presek dve prave i izvršavaju se još neke operacije za konstantno vreme. Dakle, ukupno vreme izvršavanja algoritma iznosi $O(n)$. Algoritam se može ubrzati tako što se izbegne analiza nekih stranica mnogougla, što se može učiniti smeštanjem stranica u specijalne strukture podataka koje omogućavaju brzu prostornu pretragu (npr. R-stabla).

5.2.3 Ispitivanje pripadnosti tačke unutrašnjosti konveksnog mnogougla

Provera pripadnosti tačke unutrašnjosti mnogougla može biti mnogo efikasnija ako je mnogougao konveksan.

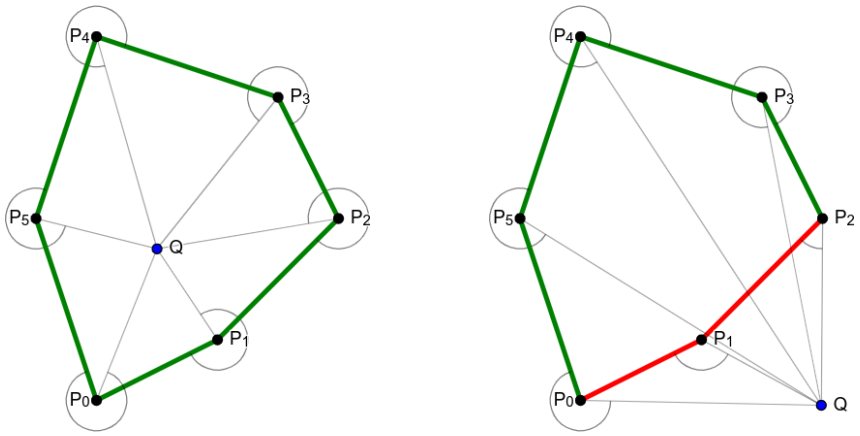
Problem

Ispitati da li tačka Q pripada unutrašnjosti konveksnog mnogougla $P_0P_1 \dots P_{n-1}$.

Jednostavnosti radi, ponovo za početak pretpostavljamo da tačka Q ne pripada ivicama datog mnogougla $P_0P_1 \dots P_{n-1}$.

Jedno moguće rešenje je uopštenje algoritma za ispitivanje da li tačka pripada datom trouglu. Ono nije efikasnije nego opšte rešenje za prost mnogougao, ali je jednostavnije od njega.

Jednostavnosti radi, pretpostavimo da su temena konveksnog mnogougla data u pozitivnom smeru (smeru suprotnom od smera kazaljke na časovniku). Tačka Q pripada mnogouglu ako i samo ako se nalazi s leve strane svake usmerene stranice P_iP_{i+1} mnogougla, odnosno ako za svako i trougao $P_iP_{i+1}Q$ ima pozitivnu orijentaciju. U primeru prikazanom na slici 5.24 ovaj uslov važi za tačku Q na slici levo, ali ne i za tačku Q na slici desno. Složenost ovog algoritma iznosi $O(n)$.



Slika 5.24: Na slici levo tačka Q pripada konveksnom mnogouglu jer su svi trouglovi $P_iP_{i+1}Q$ pozitivno orijentisani. Na slici desno tačka Q ne pripada (trouglovi P_0P_1Q i P_1P_1Q imaju negativnu orijentaciju)

Ako nije poznata orijentacija temena mnogougla (već samo njihov redosled) potrebno je proveriti da li svaka trojka $P_iP_{i+1}Q$ ima istu orijentaciju (bila ona pozitivna ili negativna).

Ako je orijentacija neke trojke $P_iP_{i+1}Q$ jednaka nuli, to znači da tačka Q pripada pravoj P_iP_{i+1}, pa pošto smo pretpostavili da Q ne pripada ivici P_iP_{i+1}, ona sigurno pripada spoljašnjosti mnogougla.

Možemo razmotriti i opštiji problem, u kom nije unapred poznato da tačka ne pripada ivicama mnogougla. Tada, ako je orijentacija neke trojke $P_iP_{i+1}Q$ jednaka nuli, tačka Q pripada duži P_iP_{i+1} ako i samo ako su sve ostale orijentacije istog znaka, pa rezultat određujemo u zavisnosti od toga da li želimo da smatramo da stranice pripadaju oblasti mnogougla ili ne (da li je mnogougao zatvorena ili otvorena figura).

Ideja na kojoj se zasniva efikasniji algoritam se oslanja na binarnu pretragu.

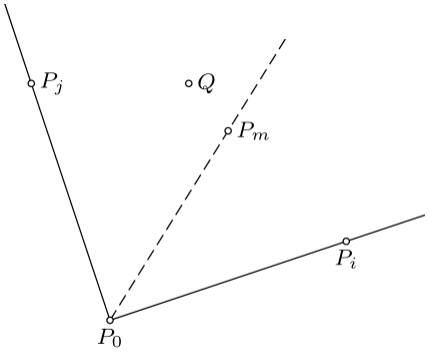
Lema 5.2.1

Neka se tačke Q i P_m nalaze unutar ugla $\angle P_iP_0P_j$ (slika 5.25). Tada važi:

- *ako su tačke P_i i Q sa iste strane prave P_0P_m, tj. ako su trojke $P_0P_mP_i$ i P_0P_mQ isto orijentisane, tada tačka Q pripada uglu $\angle P_iP_0P_m$. Ako pretpostavimo da su tačke mnogougla date u pozitivno*

orijentisanom redosledu, trojka $P_0P_mP_i$ je negativno orijentisana, pa se ovaj uslov svodi na to da je i trojka P_0P_mQ negativno orijentisana.

- *ako su tačke P_j i Q sa iste strane prave P_0P_m (kao na slici 5.25), tj. ako su trojke $P_0P_mP_j$ i P_0P_mQ isto orijentisane, tada tačka Q pripada uglu $\angle P_mP_0P_j$. Ako pretpostavimo da su tačke mnogougla date u pozitivno orijentisanom redosledu, trojka $P_0P_mP_j$ je pozitivno orijentisana, pa se ovaj uslov se svodi na to da je trojka P_0P_mQ negativno orijentisana (što je, naravno, suprotan uslov nego onaj u prethodnom slučaju).*
- *ako tačka Q baš pripada pravoj P_0P_m, usvajamo dogovor da tačka Q pripada uglu $\angle P_mP_0P_j$.*



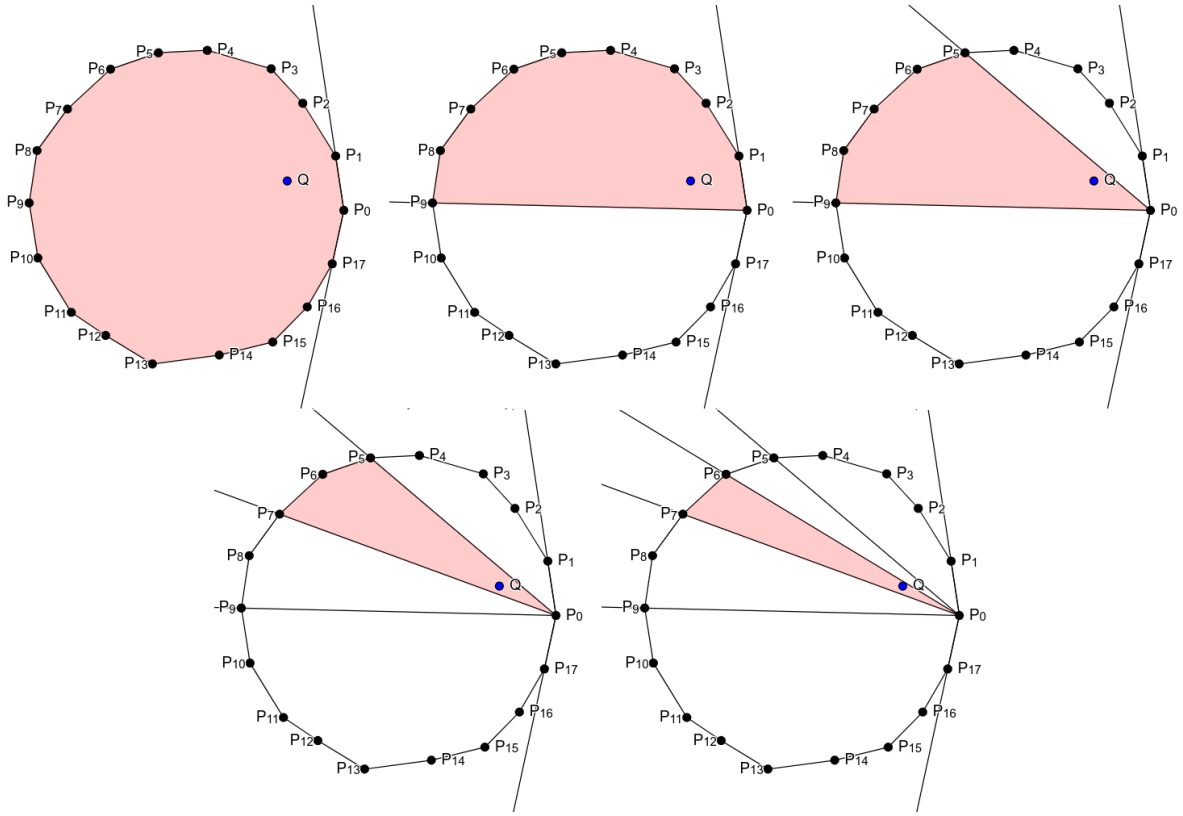
Slika 5.25: Polovljenje ugla

Iskoristićemo prethodnu lemu da binarnom pretragom pronađemo vrednost i takvu da tačka Q pripada uglu $P_iP_0P_{i+1}$. Održavaćemo tekući ugao $\angle P_iP_0P_j$ kome pripada tačka Q, koji inicijalizujemo na $\angle P_1P_0P_{n-1}$. U svakom koraku za tačku P_m biramo središnju tačku – tačku sa indeksom $m = \lfloor (i+j)/2 \rfloor$ i ažuriramo ugao kome pripada tačka Q prema prethodnoj lemi. Zaustavljamo se kada tačke P_i i P_j postanu susedna temena mnogougla tj. kada je $j = i + 1$. Posle najviše $O(\log n)$ koraka pronalazi se ugao $\angle P_iP_0P_{i+1}$ kome tačka mora pripadati ako pripada mnogouglu. Tada znamo da tačka Q pripada mnogouglu ako i samo ako pripada trouglu $\triangle P_0P_iP_{i+1}$. Na slici 5.26 prikazan je primer postupka polovljenja ugla kome pripada tačka Q.

Na početku je moguće proveriti da li tačka pripada uglu $\angle P_1P_0P_{n-1}$ i ako ne pripada odmah se može zaključiti da tačka ne pripada mnogouglu. Međutim, ta provera nije neohpodna, jer ako tačka ne pripada tom uglu binarnom pretragom će se doći ili do trougla $\triangle P_0P_1P_2$ ili $P_0P_{n-2}P_{n-1}$ i tada će se zaključiti da mu tačka ne pripada, pa ne pripada ni mnogouglu.

Ovaj algoritam se može veoma jednostavno implementirati.

```
// provera da li se tacka A nalazi unutar preseka poligona A i ugla A[l]A[0]A[d]
bool sadrzi(const vector<Tacka>& poligon, const Tacka& A, int l, int d) {
  if (d - l == 1)
    return tackaUTrouglu(poligon[0], poligon[l], poligon[d], A);
  int s = l + (d - l) / 2;
  if (orijentacija(poligon[0], poligon[s], A) == POZITIVNA)
    return sadrzi(poligon, A, s, d);
  else
    return sadrzi(poligon, A, l, s);
}

// provera da li konveksni poligon sadrzi tacku A
bool sadrzi(const vector<Tacka>& poligon, const Tacka& A) {
```



Slika 5.26: Ispitivanje pripadnosti tačke konveksnom mnogouglu metodom polovljenja ugla. Na kraju je potrebno ispitati da li tačka pripada trouglu $P_0P_6P_7$

```
  return sadrzi(poligon, A, 1, poligon.size()-1);
}
```

Prethodna implementacija podrazumeva da se proverava pripadnost zatvorenom mnogouglu (stranice se smatraju delom mnogougla) što dalje podrazumeva da provera pripadnosti trouglu takođe mora da uključi i stranice. U suprotnom je potrebno posebno obraditi slučaj kada središnja tačka A_s leži na pravoj A_0A i umesto da se procedura nastavi rekurzivno (što je trenutno slučaj), treba proveriti da li je A strogo između A_0 i A_s.

5.2.4 Konveksni omotač

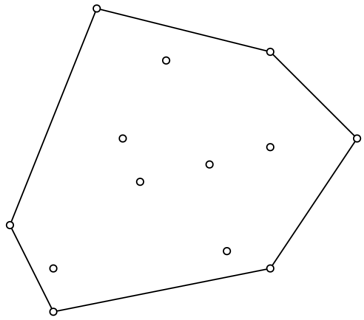
Konveksni omotač (engl. convex hull) konačnog skupa tačaka definiše se kao najmanji konveksni mnogougao takav da sve tačke tog skupa pripadaju unutrašnjosti ili ivicama tog mnogougla (slika 5.27). Definisaćemo i da je za jednu tačku konveksni omotač ta tačka, a za par tačaka duž koja ih spaja. Jasno je iz definicije da je konveksni omotač tri nekolinearne tačke trougao kome su te tačke temena. Konveksni omotač se predstavlja na isti način kao bilo koji mnogougao — nizom svojih uzastopnih temena.

Kao što smo već videli, obrada konveksnih mnogouglova jednostavnija je od obrade proizvoljnih mnogouglova. Na primer, provera pripadnosti pripadnosti tačke unutrašnjosti proizvoljnog prostog mnogougla je složenosti $O(n)$, dok se pripadnost tačke konveksnom n-touglu vrši se u složenosti $O(\log n)$.

Problem

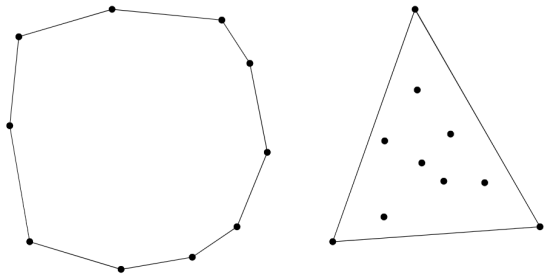
Konstruisati konveksni omotač zadatih n tačaka u ravni.

Temena konveksnog omotača su neke od tačaka iz zadatog skupa. Kazaćemo da tačka *pripada* omotaču ako je teme omotača. Ako zanemarimo trivijalne slučajeve kada polazni skup ne sadrži bar tri nekolinearne



Slika 5.27: Primer skupa tačaka i njegovog konveksnog omotača.

tačke, konveksni omotač može se sastojati od najmanje tri, a najviše n tačaka (slika 5.28).



Slika 5.28: Konveksni omotač koji sadrži sve tačke skupa (levo) i konveksni omotač skupa koji sadrži samo tri tačke (desno)

Konveksni omotači imaju široku primenu (na primer, u praćenju širenja epidemija, modelovanju glatkih krivih, smanjenju broja boja na slici, planiranju kretanja robota, kao gradivni element u drugim algoritmima, kao što je recimo traženje dijametra skupa tačaka), pa su zbog toga razvijeni mnogobrojni algoritmi za njihovu konstrukciju. Razmotrimo problem navigacije robota u ravni od tačke A do tačke B. Najjednostavniji način da robot realzuje ovo kretanje je pravolinijski, duž duži AB. Međutim, na ovoj pravolinijskoj putanji se može naći neka od prepreka. Stoga se svaka od prepreka može uzorkovati tačkom u ravni, a onda napraviti konveksni omotač svih ovih tačaka. Putanja koja ne seče ovaj omotač garantuje da neće biti sudara robota sa nekom od prepreka.

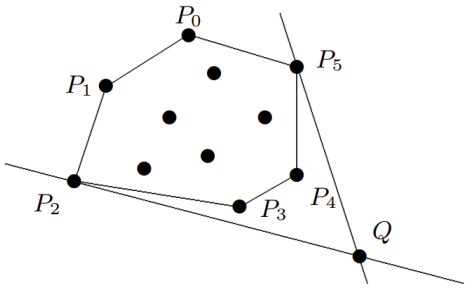
5.2.4.1 Direktni induktivni pristup

Kao i obično, pokušaćemo najpre sa direktnim induktivnim pristupom. Konveksni omotač za jednu tačku je jednočlan i to može biti baza indukcije. Pretpostavimo da umemo da konstruišemo konveksni omotač skupa sa manje od n tačaka i pokušajmo da konstruišemo konveksni omotač skupa sa n tačaka. Na koji način n-ta tačka može da promeni konveksni omotač prvih $n-1$ tačaka? Postoje dva moguća slučaja:

- nova tačka u prethodnom konveksnom omotaču, pa on ostaje nepromenjen,
- nova tačka je van prethodnog konveksnog omotača, pa se omotač "širi" da obuhvati i novu tačku (slika 5.29).

Prilikom dodavanja svake nove tačke, potrebno je, dakle, rešiti dva potproblema: utvrđivanje da li je nova tačka unutar omotača i proširivanje omotača novom tačkom.

Stvar se može uprostiti pogodnim izborom n-te tačke, tako da ne moramo eksplicitno vršiti proveru da li ona pripada prethodnom omotaču. Zvuči primamljivo pokušati da uvek biramo tačku unutar omotača, jer tada ne moramo da menjamo omotač da bi se proširila induktivna hipoteza; to, međutim, nije uvek moguće jer u nekim slučajevima sve tačke polaznog skupa pripadaju konveksnom omotaču. Sasvim suprotna mogućnost je da se u svakom koraku bira tačka za koju znamo da ne pripada unutrašnjosti tekućeg omotača.



Slika 5.29: Proširivanje konveksnog omotača novom tačkom u slučaju kada nova tačka ne pripada prethodnom omotaču.

To će biti zadovoljeno ako se u svakom koraku bira neka ekstremna tačka novog mnogougla, pri čemu nam taj izbor, kao što ćemo videti, dodatno uprošćava postupak ažuriranja konveksnog omotača. Podsetimo se, ekstremne tačke su se pokazale uspešne pri rešavanju problema konstrukcije prostog mnogougla i definisali smo ih kao tačke sa maksimalnom x-koordinatom (pri čemu se ako ima više takvih tačaka bira ona sa minimalnom y-koordinatom). Da bi se u svakom koraku koristila ekstremna tačka, potrebno je tačke obrađivati u neopadajućem redosledu x-koordinata (pri čemu tačke sa istom x-koordinatu treba obrađivati u nerastućem redosledu y-koordinata), što se najlakše postiže ako se u startu tačke sortiraju na taj način.

Neka je tekuća tačka kojom se proširuje konveksni omotač konstruisan od početnih $n-1$ tačaka tačka Q, koja je ekstremna u tekućem skupu od n tačaka. Tačka Q mora biti teme novog konveksnog omotača. Pitanje je kako promeniti konveksni omotač ostalih $n-1$ tačaka tako da novi omotač obuhvati tačku Q. Potrebno je najpre pronaći temena starog omotača koja su u unutrašnjosti novog omotača (P_3 i P_4 u primeru na slici 5.29) i ukloniti ih, a zatim je potrebno umetnuti novo teme Q između dva postojeća (P_2 i P_5 na slici 5.29).

Određivanje tačaka tekućeg omotača koje treba da budu zamenjene tačkom Q se može izvršiti korišćenjem pravih oslonca. *Prava oslonca* (engl. supporting plane) konveksnog mnogougla je prava koja sadrži bar jednu tačku mnogougla i sve ostale tačke mnogougla se nalaze sa iste strane te prave. U našem slučaju su bitne prave oslonca koje sadrže tačku Q. Na primer, na slici 5.29 prave QP_2 i QP_5 su dve prave oslonca i polaznog i proširenog konveksnog mnogougla. Obično samo jedno teme mnogougla povezivanjem sa Q određuje pravu oslonca (postoji i specijalni slučaj kad više temena mnogougla leži na pravi oslonca, koji ćemo na trenutak zanemariti). Mnogougao leži između dve prave oslonca, što ukazuje na to na koji način treba izvesti modifikaciju omotača. Prave oslonca zahvataju minimalni i maksimalni ugao sa x-osom među svim pravama koje sadrže tačku Q i neko teme mnogougla. Da bismo odredili ta dva temena, potrebno je da razmotrimo prave iz tačke Q ka svim temenima, da izračunamo uglove koje one zahvataju sa x-osom, i među tim uglovima izaberemo minimalan (čime dobijamo tačku P_i) i maksimalan (čime dobijamo tačku P_j). Umesto eksplicitnog računanja uglova, moguće ih je samo upoređivati na osnovu orijentacije (ugao koji prava QP_m zahvata sa x-osom manji je od ugla koji prava QP_n zahvata sa x-osom ako i samo ako je trojka QP_mP_n pozitivno orijentisana. U specijalnom slučaju se može desiti da su neke tačke P_m i P_n kolinearne sa tačkom Q i tada uzimamo onu od njih koja se nalazi dalje od Q (jer pretpostavljamo da je konveksni omotač zatvoren mnogougao i da pošto bliža tačka pripada ivici, pripada i omotaču).

Posle pronalaženja dva ekstremna temena P_i i P_j modifikovani omotač dobijamo tako što onaj niz temena između P_i i P_j (bez P_i i P_j) čije su tačke sa iste strane prave P_iP_j kao i tačka Q zamenimo tačkom Q (ako je taj niz prazan, samo se umeće nova tačka Q).

Primer 5.2.2

U primeru prikazanom na slici 5.29 temena P_3 i P_4 se nalaze se sa iste strane prave P_5P_2 kao i tačka Q, pa se niz temena starog omotača $[P_3, P_4]$ zamenjuje novim temenom Q. Tačke P_0 i P_1 su sa suprotne strane prave P_5P_2 od tačke Q, pa one ostaju deo konveksnog omotača.

Da bi se realizovao prethodni algoritam, nije neophodno eksplicitno ispitivati koji od dva niza tačaka između P_i i P_j leži sa iste strane prave P_iP_j kao i Q. Naime, ako sva temena trenutnog konveksnog omotača čuvamo u redosledu pozitivne orijentacije, pošto je tačka Q ekstremna i nalazi se desno od prave P_iP_j, tačke između P_j i P_i će biti sa iste strane prave P_iP_j kao i Q, dok će tačke između P_i i P_j biti sa suprotne strane. Dakle, uvek je potrebno niz tačaka od P_{j+1} do P_{i-1} zameniti tačkom Q (pri čemu se uvećanje i umanjenje indeksa računa ciklično tj. po modulu n). Ako su tačke P_j i P_i susedne, taj niz može biti i prazan.

U nastavku je prikazana C++ implementacija ovog algoritma.

```
vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  // sortiramo tacke po x koordinati neopadajuce,
  // ako postoje dve tacke sa istom vrednoscu x koordinate
  // prednost dajemo onoj sa vecom y koordinatom
  sort(begin(tacke), end(tacke),
       [](const Tacka &p1, const Tacka &p2) {
         return (p1.x < p2.x) || (p1.x == p2.x && p1.y > p2.y);
       });

  // u konveksni omotac dodajemo prvu tacku
  vector<Tacka> omotac;
  omotac.push_back(tacke[0]);

  // dodajemo jednu po jednu tacku u omotac u redosledu rastucih x koordinata
  for (int k = 1; k < tacke.size(); k++) {
    // trazimo gornju pravu oslonca
    // to je prava P_kP_i tako da za svako ii vazi
    // da trojka tacaka {P_k, P_i, P_ii} ima pozitivnu orijentaciju ili su
    // tacke kolinearne, ali je P_ii dalje od tacke P_k nego P_i
    int i = 0;
    for (int ii = 0; ii < omotac.size(); ii++) {
      int o = orijentacija(tacke[k], omotac[i], omotac[ii]);
      if (o == NEGATIVNA ||
          (o == KOLINEARNE && kvadratRastojanja(tacke[k], omotac[i]) <
                              kvadratRastojanja(tacke[k], omotac[ii])))
        i = ii;
    }

    // trazimo donju pravu oslonca
    // to je prava P_kP_j tako da za svako jj vazi
    // da trojka tacaka {P_k, P_jj, P_j} ima pozitivnu orijentaciju ili su
    // tacke kolinearne, ali je P_jj dalje od tacke P_k nego P_j
    int j = 0;
    for (int jj = 0; jj < omotac.size(); jj++) {
      int o = orijentacija(tacke[k], omotac[jj], omotac[j]);
      if (o == NEGATIVNA ||
          (o == KOLINEARNE && kvadratRastojanja(tacke[k], omotac[j]) <
                              kvadratRastojanja(tacke[k], omotac[jj])))
        j = jj;
    }

    zameni(omotac, (j + 1) % omotac.size(), i, tacke[k]);
```

```
  }
  return omotac;
}

// podniz niza a_i,...,a_{j-1} menjamo elementom x
// ako je i > j, brisu se elementi do kraja, pa zatim sa pocetka niza
void zameni(vector<Tacka>& a, int i, int j, const Tacka& x) {
    if (i < j)
      // ako je i < j, onda su tacke koje brisemo susedne
      a.erase(a.begin() + i, a.begin() + j);
    else if (i > j) {
      // ako je j > i, onda elemente brisemo iz dva dela
      a.erase(a.begin() + i, a.end());
      a.erase(a.begin(), a.begin() + j);
    }
    // ako je i = j, onda se ne brise nista

    // ubacujemo novi broj x
    a.insert(a.begin() + i, x);
}
```

Primetimo da se u funkciji sortira dati niz tačaka, što znači da argument funkcije nije konstantan. Ako bismo insistirali da se niz ne sme menjati tokom izračunavanja omotača (što je korisna praksa), morali bismo napraviti sortiranu kopiju niza unutar funkcije, čime bi se dobio malo neefikasniji algoritam.

Razmotrimo složenost ovog algoritma. Prvo se sortira niz temena, za šta je potrebno vreme $O(n \log n)$. Za svaku novu tačku koja se dodaje skupu treba izračunati uglove koje grade prave određene tačkom Q i svakim od temena prethodnog omotača i x-osa, pronaći među njima minimalni i maksimalni ugao, izbaciti neka temena iz prethodnog omotača i dodati novo teme. Dakle, složenost dodavanja k-te tačke u tekući skup tačaka je $O(k)$, a ukupno razmatramo n tačaka pa se složenost ovog algoritma može opisati rekurentnom jednačinom $T(n) = T(n-1) + O(n)$ čije je rešenje $O(n^2)$. Pošto je vreme sortiranja $O(n \log n)$ asimptotski manje od vremena $O(n^2)$ potrebnog za ostale operacije, ono ne utiče na ukupnu složenost $O(n^2)$. Naglasimo i da izbor strukture podataka za čuvanje trenutnog niza temena konveksnog mnogougla (liste ili vektora) ne utiče na asimptotsku složenost (složenost funkcije `zameni` bi mogla biti snižena kada bi se koristila lista, ali bi asimptotska složenost ostala kvadratna).

5.2.4.2 Uvijanje poklona

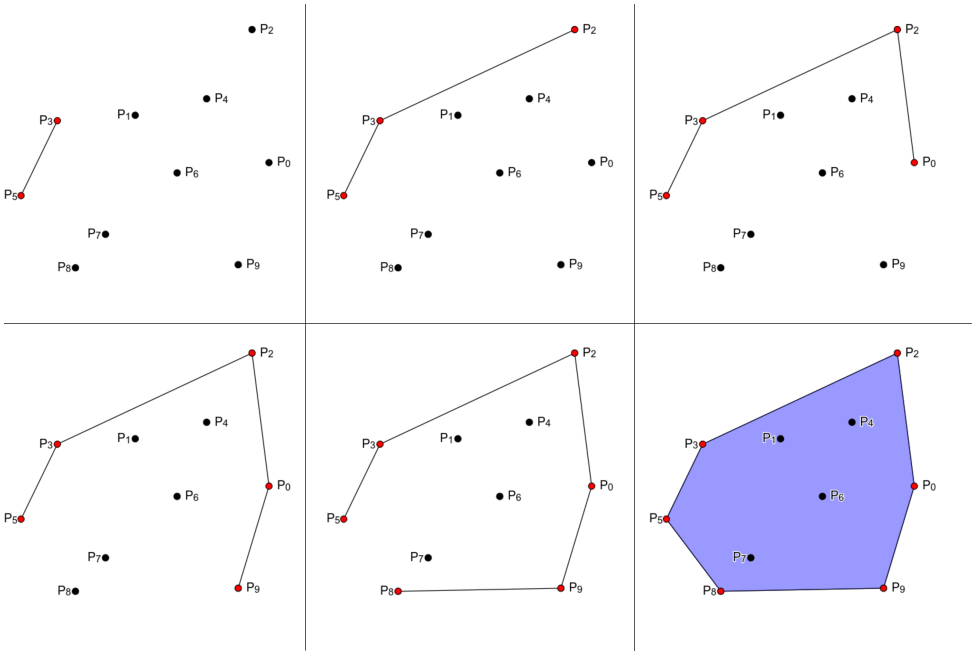
Na koji način se opisani algoritam može poboljšati? Kad proširujemo mnogougao teme po teme, dosta vremena trošimo na formiranje konveksnih omotača od tačaka za koje će se kasnije možda pokazati da su unutrašnje za konačni konveksni omotač. Može li se to izbeći? Umesto da pravimo konveksne omotače podskupova datog skupa tačaka, možemo da posmatramo kompletan skup tačaka i da direktno pravimo njegov konveksni omotač. Može se, kao i u prethodnom algoritmu, krenuti od ekstremne tačke. U principu, svejedno je koja se ekstremna tačka koristi. Da bismo ilustrovali malo drugačiji pristup od dosadašnjih rešenja, možemo, na primer, krenuti od tačke sa najmanjom vrednošću x koordinate, a ako ih ima više one među njima koja ima najmanju y koordinatu. Kao što smo već pomenuli, ekstremna tačka uvek pripada konveksnom omotaču. Ideja je da se u svakom koraku u omotač doda naredno teme koje je deo konačnog konveksnog omotača (pri čemu temena obilazimo, na primer, u negativnom matematičkom smeru). Kako nalazimo narednu stranicu konveksnog omotača? Jedno teme te stranice biće poslednja tačka do sada određenog dela konveksnog omotača. Da bismo odredili drugo teme stranice, analiziraćemo sve tačke i tražiti onu koja daje duž koja sa prethodnom duži gradi što veći ugao.

Umesto računanja uglova možemo upotrebiti orijentaciju. Neka je P_t tekuća tačka u konveksnom omotaču.

Obrađivaćemo preostale tačke i suštinski primenjivati algoritam određivanja maksimuma. Pretpostavićemo da se maksimalni ugao postiže za neku tačku P_l različitu od P_t. Zatim ćemo redom obrađivati sve ostale tačke.

- Ako je orijentacija $P_tP_lP_i$ negativna, tada je ugao duži P_tP_l veći od ugla P_tP_i i P_l ostaje kandidat za teme duži sa maksimalnim uglom.
- Ako je orijentacija $P_tP_lP_i$ pozitivna, tada je ugao duži P_tP_i veći od ugla P_tP_l i P_i postaje novi kandidat za teme duži sa maksimalnim uglom.
- Ako je orijentacija $P_tP_lP_i$ jednaka nuli, tri tačke su kolinearne. Pošto smatramo da je konveksni omotač zatvoren mnogougao, želimo da uzmemo onu tačku koja je dalja od P_t, tako da se tekući kandidat menja ako je P_l između P_t i P_i.

Postupak se završava kada se ustanovi da je tačka koja određuje najveći ugao sa prethodnom tačkom jednaka početnoj tački. Koraci u izvršavanju ovog algoritma za jedan skup tačaka su prikazani na slici 5.30.



Slika 5.30: Koraci u izvršavanju algoritma uvijanja poklona

Ovaj algoritam se iz razumljivih razloga zove *uvijanje poklona* (engl. gift wrapping algorithm). Polazi se od jednog temena "poklona", i onda se on uvija u konveksni omotač pronalaženjem temena po temena omotača. Poznat je i pod nazivom *Džarvisov marš* (engl. Jarvis March algorithm), po svom autoru.

Algoritam uvijanja poklona je direktna posledica primene sledeće induktivne hipoteze (po k):

Induktivna hipoteza: Za zadati skup od n tačaka u ravni, umemo da pronađemo konveksni put dužine $k < n$ koji je deo konačnog konveksnog omotača datog skupa tačaka.

Kod ove hipoteze naglasak je na proširivanju *puta*, a ne omotača. Umesto da pronalazimo konveksne omotače podskupova, mi pronalazimo deo konačnog konveksnog omotača.

Algoritam uvijanja poklona je moguće implementirati na sledeći način:

```
vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  // odredjujemo redni broj ekstremne tacke (one sa najmanjom x koordinatom)
  auto cmp = [] (const Tacka& t1, const Tacka& t2) {
    return t1.x < t2.x || (t1.x == t2.x && t1.y < t2.y);
  };
```

```
  int prva = distance(begin(tacke), min_element(begin(tacke), end(tacke), cmp));

  // niz tacaka koje cine omotac
  vector<Tacka> omotac;
  // krecemo od ekstremne tacke
  int tekuca = prva;
  do {
    omotac.push_back(tacke[tekuca]);
    // odredjujemo narednu tacku u omotacu, zahtevajuci da se sve
    // ostale tacke nalaze sa desne strane prave odredjene tekucom i
    // tom narednom tackom
    int naredna = tekuca == 0 ? 1 : 0;
    for (size_t i = 0; i < tacke.size(); i++) {
      if (i == tekuca || i == naredna)
        continue;
      Orijentacija o = orijentacija(tacke[tekuca], tacke[naredna], tacke[i]);
      if (o == POZITIVNA ||
          (o == KOLINEARNE && izmedju(tacke[tekuca], tacke[naredna], tacke[i])))
        naredna = i;
    }
    tekuca = naredna;
  } while (tekuca != prva);
  return omotac;
}
```

Ako je m broj temena konačnog konveksnog omotača, vremenska složenost uvijanja poklona je $O(mn)$. Dakle, ovaj algoritam spada u grupu algoritama čija složenost zavisi ne samo od dimenzije ulaza, već i od dimenzije izlaza (kaže se nekada kaže da ima *izlazno zavisnu složenost*). U slučaju kada je konveksni omotač skupa tačaka trougao (slika 5.28 desno), algoritam uvijanja poklona ima složenost $O(n)$, a u slučaju kada sve tačke (ili u opštem slučaju $O(n)$ tačaka) pripadaju konveksnom omotaču (slika 5.28 levo) složenost algoritma je $O(n^2)$.

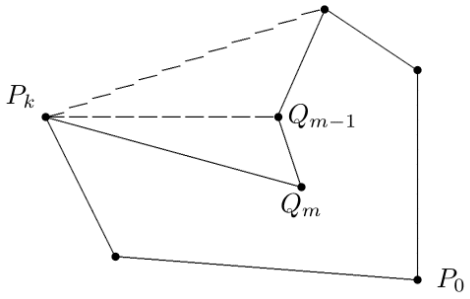
5.2.4.3 Grejemov algoritam

U nastavku ćemo razmotriti algoritam za nalaženje konveksnog omotača složenosti $O(n \log n)$. Započinje se sortiranjem tačaka prema uglovima, slično kao pri konstrukciji prostog mnogougla. Neka je P_0 ekstremna tačka, recimo ona sa najvećom x-koordinatom (i sa najmanjom y-koordinatom, ako ima više tačaka sa najvećom x-koordinatom). Za svaku tačku P_i iz datog skupa tačaka izračunavamo ugao između prave P_0P_i i x-ose, i sortiramo tačke prema veličini ovih uglova (videti primer na slici 5.19). Tačke koje zahvataju isti ugao sortiramo na osnovu rastojanja od P_0, opadajuće (tako da se prvo obrađuju dalje tačke). Alternativno, moguće je i da izbacimo sve kolinearne tačke osim one koja je najdalja od P_0.

Tačke, dakle, obilazimo redosledom kojim se pojavljuju u (prostom) mnogouglu, i prilikom obilaska pokušavamo da identifikujemo temena konveksnog omotača. Kao i kod uvijanja poklona pamtimo put sastavljen od tačaka koje su obrađene tokom obilaska. Preciznije, to je konveksni put koji određuje konveksni mnogougao (dobijen povezivanjem prve i poslednje tačke puta), koji sadrži sve do sada pregledane tačke. Zbog toga, je u trenutku kad su sve tačke pregledane, konveksni omotač skupa tačaka konstruisan. Osnovna razlika između ovog algoritma i algoritma uvijanja poklona je u činjenici da tekući konveksni put ne mora da bude deo konačnog konveksnog omotača. On određuje konveksni omotač do sada pregledanih tačaka. Dakle tekući konveksni put može da sadrži tačke koje ne pripadaju konačnom konveksnom omotaču; te tačke biće eliminisane kasnije. Na primer, put od P_0 do Q_m na slici 5.31 je konveksan, ali tačke Q_m i Q_{m-1} očigledno ne pripadaju konveksnom omotaču kompletnog skupa tačaka. Ovo razmatranje sugeriše algoritam zasnovan na sledećoj induktivnoj hipotezi.

Induktivna hipoteza: Ako je dato n tačaka u ravni, uređenih prema algoritmu za konstrukciju prostog mnogougla, onda umemo da konstruišemo konveksni put preko nekih od prvih k tačaka, takav da njemu odgovarajući konveksni mnogougao obuhvata prvih k tačaka.

Slučaj $k = 1$ je trivijalan (omotač sadrži samo tačku P_0). Označimo konveksni put dobijen (induktivno) za prvih k tačaka sa $P = Q_0, Q_1, \ldots, Q_m$ (pri čemu je $\{Q_0, \ldots Q_m\} \subseteq \{P_0, \ldots, P_{k-1}\}$). Proširimo induktivnu hipotezu na $k + 1$ tačaka. Posmatrajmo ugao između pravih $Q_{m-1}Q_m$ i Q_mP_k (videti sliku 5.31).



Slika 5.31: Ilustracija osnovnog koraka u Grejemovom algoritmu.

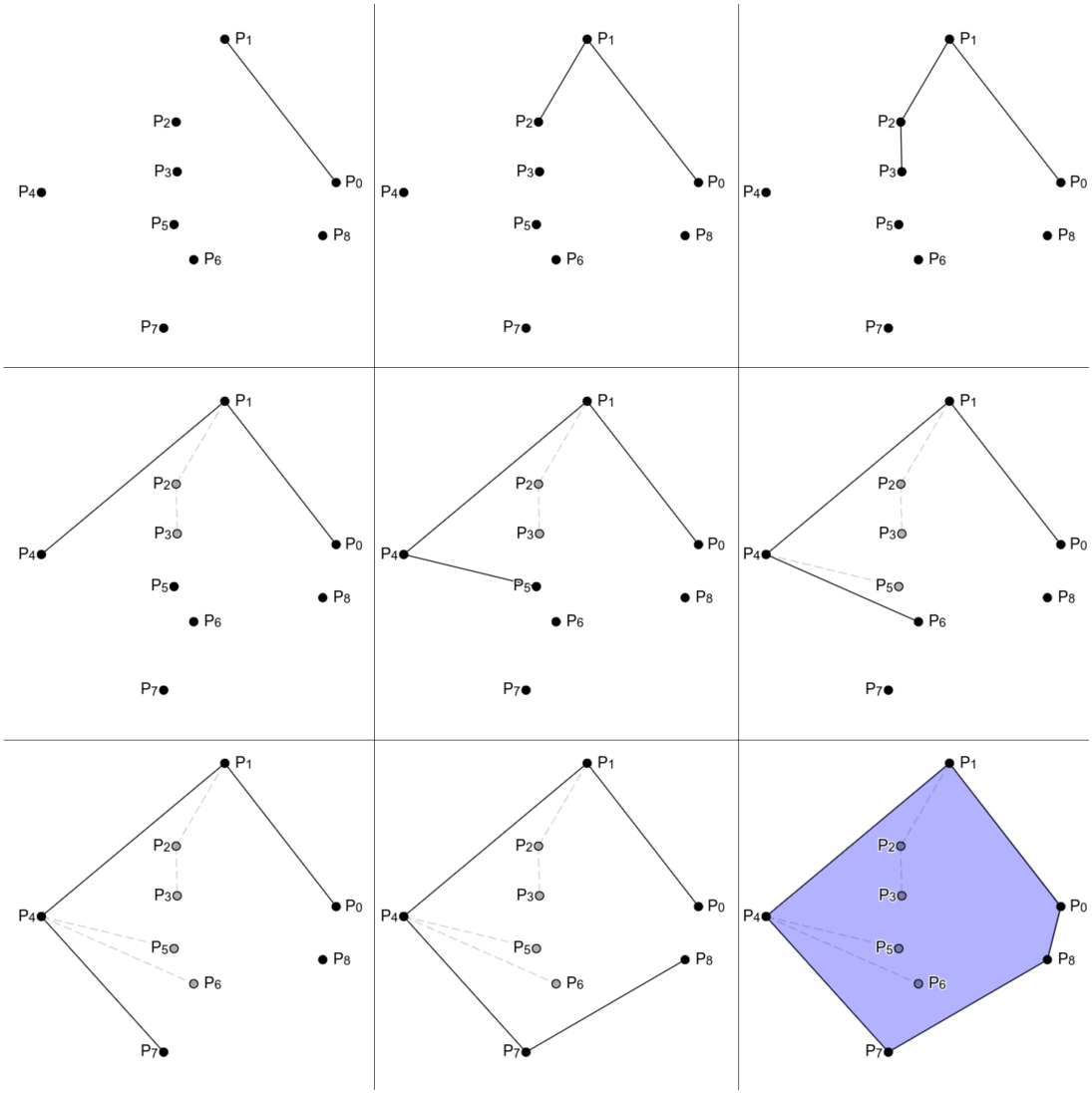
Ako je taj ugao manji od π (pri čemu se ugao meri iz unutrašnjosti mnogougla), onda se tačka P_k može dodati postojećem putu (novi put je zbog toga takođe konveksan), čime je korak indukcije završen. U protivnom, tvrdimo da tačka Q_m leži u mnogouglu dobijenom zamenom tačke Q_m u putu P tačkom P_k, i povezivanjem tačke P_k sa tačkom P_0. Ovo je tačno jer su tačke uređene prema odgovarajućim uglovima, pa se prava P_0P_k nalazi "levo" od prvih k tačaka. Zbog toga Q_m jeste unutar gore definisanog mnogougla, može se izbaciti iz puta P, a tačka P_k se može dodati tekućem putu. Da li je time završena obrada $(k+1)$-ve tačke? Ne mora biti. Nakon izbacivanja tačke Q_m dobijeni put ne mora uvek da bude konveksan. Zaista, slika 5.31 pokazuje da postoje slučajevi kad treba eliminisati još tačaka. Na primer, tačka Q_{m-1} može da bude unutar mnogougla definisanog modifikovanim putem. Moramo da (unazad) nastavimo sa proverama poslednje dve stranice puta, sve dok ugao između njih ne postane manji od π: ovo se uvek mora desiti pre nego što se iscrpu sve tačke, jer će u najgorem slučaju barem tačke Q_0, Q_1 i P_k činiti konveksni put (zbog prethodnog uređenja tačaka). Put je tada konveksan, a hipoteza je proširena na $k + 1$ tačku.

Umesto da računamo ugao između pravih $Q_{m-1}Q_m$ i Q_mP_k, može se razmatrati orijentacija trojke $Q_{m-1}Q_mP_k$: ukoliko je orijentacija ovog trougla pozitivna, onda je ugao između pravih $Q_{m-1}Q_m$ i Q_mP_k manji od π, a ako je pozitivna, taj ugao je veći od π. Ako je orijentacija jednaka nuli, tačke $P_kQ_mQ_{m-1}$ su kolinearne. Na osnovu sortiranja tačaka znamo da tačka Q_m leži između Q_{m-1} i P_k, pa se i u tom slučaju Q_m može zameniti tačkom P_k.

Na slici 5.32 prikazani su koraci prilikom izvršavanja Grejemovog algoritma na jednom primeru.

Grejemov algoritam je moguće implementirati na sledeći način:

```
vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  // trazimo ekstremnu tacku sa maksimalnom x koordinatom,
  // u slucaju da ima vise tacaka sa maksimalnom x koordinatom
  // biramo onu sa najmanjom y koordinatom
  auto max = max_element(begin(tacke), end(tacke),
                         [](const Tacka& t1, const Tacka& t2) {
                           return t1.x < t2.x ||
                                  (t1.x == t2.x && t1.y > t2.y);
                         });
  // dovodimo je na početak niza
  swap(*begin(tacke), *max);
```



Slika 5.32: Koraci prilikom izvršavanja Grejemovog algoritma. Primetimo kako se tačke P_2 i P_3 prvo uključuju u omotač, a kada se naiđe na tačku P_4 onda se isključuju iz njega. Slično se kasnije događa sa tačkama P_5, a zatim i P_6.

```
  const Tacka& t0 = tacke[0];

  // sortiramo ostatak niza (tačke sortiramo na osnovu ugla koji
  // zaklapaju u odnosu horizontalnu polupravu koja polazi nadesno
  // iz ekstremne tacke), a kolinearne na osnovu rastojanja te tacke
  sort(next(begin(tacke)), end(tacke),
       [t0](const Tacka& t1, const Tacka& t2) {
         Orijentacija o = orijentacija(t0, t1, t2);
         if (o == KOLINEARNE)
           return kvadratRastojanja(t0, t1) <= kvadratRastojanja(t0, t2);
         return o == POZITIVNA;
       });

  // obradjujemo tacke u sortiranom redosledu izbacujuci prethodne sve
  // dok se tekuca tacka sa dve poslednje ne cini levi zaokret
  vector<Tacka> omotac;
  omotac.push_back(tacke[0]);
  omotac.push_back(tacke[1]);
  for (size_t i = 2; i < tacke.size(); i++) {
    while (omotac.size() >= 2 &&
           orijentacija(omotac[omotac.size()-2],
                        omotac[omotac.size()-1],
                        tacke[i]) != POZITIVNA)
      omotac.pop_back();
    omotac.push_back(tacke[i]);
  }
  return omotac;
}
```

Glavni deo složenosti Grejemovog algoritma potiče od početnog sortiranja. Ostatak algoritma izvršava se za vreme $O(n)$. Svaka tačka skupa razmatra se tačno jednom u induktivnom koraku kao P_k. U tom trenutku tačka se uvek dodaje konveksnom putu. Ista tačka biće razmatrana i kasnije (možda čak i više nego jednom) da bi se proverila njena pripadnost konveksnom putu. Broj tačaka na koje se primenjuje ovakav povratni test može biti veliki, ali se sve one, sem dve (tekuća tačka i tačka za koju se ispostavlja da dalje pripada konveksnom putu) eliminišu, a tačka može biti eliminisana samo jednom! Prema tome, troši se najviše konstantno vreme za eliminaciju svake tačke i konstantno vreme za njeno dodavanje, te je ukupna složenost ove faze $O(n)$. Zbog početnog sortiranja tačaka vreme izvršavanja kompletnog algoritma iznosi $O(n \log n)$.

5.2.4.4 Brzi algoritam za traženje konveksnog omotača

Pokušajmo da isti problem rešimo tehnikom dekompozicije, nalik algoritmu brzog sortiranja (QuickSort). Odredimo tačke iz datog skupa sa minimalnom i maksimalnom x koordinatom (ako ima više takvih onda biramo onu sa minimalnom ili maksimalnom y koordinatom): neka su to tačke P_i i P_j. Obe ove tačke sigurno pripadaju konveksnom omotaču. Duž P_iP_j deli skup tačaka na dva podskupa: svaki od podskupova čine tačke koje se nalaze sa iste strane te duži. Razmotrimo jedan od ova dva podskupa: u njemu tražimo tačku na maksimalnom rastojanju od duži P_iP_j: neka je to tačka P_k. I ona sigurno pripada konveksnom omotaču. Tačke iz skupa koje pripadaju trouglu $P_iP_jP_k$ ne mogu biti deo konveksnog omotača i ne moraju se dalje razmatrati. Prethodno razmatranje sada možemo primeniti na dve nove duži koje formiraju trougao: P_iP_k i P_kP_j: tražimo najudaljeniju tačku od duži P_iP_k sa one strane prave P_iP_k sa koje nije P_j i, slično, najudaljeniju tačku od duži P_kP_j sa one strane prave P_kP_j sa koje nije tačka P_i. Sa postupkom nastavljamo sve dok na raspolaganju ima još tačaka. Tačke koje su tokom izvršavanja algoritma birane kao najudaljenije

pripadaju konveksnom omotaču. Na ovaj način došli smo do tzv. *brzog algoritma za određivanje konveksnog omotača* (engl. QuickHull algorithm).

Napomenimo da prilikom traženja najudaljenije tačke P_k od duži P_iP_j, nije neophodno računati tačno rastojanje po formuli:

$$d = \frac{|\overrightarrow{P_kP_i} \times \overrightarrow{P_kP_j}|}{|\overrightarrow{P_iP_j}|}$$

Naime, mi tražimo najudaljeniju tačku od prave kroz fiksirane dve tačke, te će imenilac ovog izraza uvek biti isti. Stoga se mera rastojanja ovde može računati po formuli $d = |\overrightarrow{P_kP_i} \times \overrightarrow{P_kP_j}|$.

U nastavku je prikazana C++ implementacija ovog algoritma. Tokom rekurzivnih poziva održava se skup temena konveksnog omotača, da bi se nakon prikupljanja svih temena ona sortirala tako da uzastopna temena idu jedno iza drugog (to se postiže tako što se neka od ekstremnih tačaka, u ovom slučaju ona sa maksimalnom x-koordinatom stavi na početak, a zatim se ostale tačke sortiraju po uglu u odnosu na nju). Pošto je osnovni algoritam takav da je moguće da su u izgrađenom omotaču neka tri temena kolinearna, nakon formiranja omotača vršimo njegovo uprošćavanje zadržavajući samo kranja temena u nizu kolinearnih temena.

```
// funkcija koja racuna konveksni omotac skupa tacaka
vector<Tacka> konveksniOmotac(vector<Tacka>& tacke) {
  // funkcija poredjenja koja je potrebna za bibliotecke funkcije
  // min_element i max_element
  auto porediKoordinate =
    [](const Tacka& A, const Tacka& B) {
      return (A.x < B.x) || (A.x == B.x && A.y > B.y);
    };

  // trazimo tacke sa najmanjom i najvecom x koordinatom
  int min = distance(tacke.begin(),
                     min_element(tacke.begin(), tacke.end(), porediKoordinate));
  int max = distance(tacke.begin(),
                     max_element(tacke.begin(), tacke.end(), porediKoordinate));

  // skup temena omotaca
  set<Tacka> temena;

  // rekurzivno trazimo konveksne omotace tacaka sa jedne strane
  // duzi odredjene tackama tacka[min] i tacka[max]
  vector<Tacka> iznad = izdvojOrijentaciju(tacke, tacke[min], tacke[max], POZITIVNA);
  konveksniPoluomotac(iznad, tacke[min], tacke[max], temena);
  // i sa druge strane
  vector<Tacka> ispod = izdvojOrijentaciju(tacke, tacke[min], tacke[max], NEGATIVNA);
  konveksniPoluomotac(ispod, tacke[min], tacke[max], temena);

  // prebacujemo tacke iz skupa u vektor i soritramo ga po uglovima
  vector<Tacka> omotac(temena.begin(), temena.end());
  const Tacka& maxTacka = tacke[max];
  auto porediUgao =
    [maxTacka](const Tacka& A, const Tacka& B) {
      if (A == maxTacka) return true;
```

```
      if (B == maxTacka) return false;
      return orijentacija(maxTacka, A, B) == POZITIVNA;
    };
  sort(begin(omotac), end(omotac), porediUgao);

  // eliminisemo susedne kolinearne tacke
  vector<Tacka> omotacBezKolinearnih;
  for (int i = 0; i < omotac.size(); i++) {
    int prethodna = i == 0 ? omotac.size() - 1 : i-1;
    int sledeca = i == omotac.size() - 1 ? 0 : i + 1;
    if (orijentacija(omotac[prethodna], omotac[i], omotac[sledeca]) !=
        KOLINEARNE)
      omotacBezKolinearnih.push_back(omotac[i]);
  }

  return omotacBezKolinearnih;
}

// trazimo konveksni omotac tacaka sa jedne strane duzi PQ
void konveksniPoluomotac(const vector<Tacka>& tacke, const Tacka& P, const Tacka& Q,
                         set<Tacka>& temena) {
  // dodajemo tacke P i Q u omotac
  temena.insert(P);
  temena.insert(Q);

  // ako ne postoji tacka sa date strane prave PQ zavrsavamo postupak
  if (tacke.empty())
    return;

  // trazimo tacku sa date strane duzi PQ
  // koja je na najvecem rastojanju od prave PQ

  // pozicija najdalje tacke i njeno rastojanje od prave PQ
  int iMax = -1; long long dMax = 0;
  for (int i = 0; i < tacke.size(); i++) {
    long long di = rastojanjeOdPrave(P, Q, tacke[i]);
    if (di > dMax) {
      iMax = i;
      dMax = di;
    }
  }
  // najdalja tacka
  const Tacka& M = tacke[iMax];

  // rekurzivno pozivamo za dva podskupa dobijena tackom `tacke[iMax]`

  // analiziramo tacke koje su na suprotnoj strani prave MP u odnosu na Q
  vector<Tacka> suprotnoOdQ = izdvojOrijentaciju(tacke, M, P, -orijentacija(M, P, Q));
  konveksniPoluomotac(suprotnoOdQ, M, P, temena);

  // analiziramo tacke koje su na suprotnoj strani prave MQ u odnosu na P
```

```
  vector<Tacka> suprotnoOdP = izdvojOrijentaciju(tacke, M, Q, -orijentacija(M, Q, P));
  konveksniPoluomotac(suprotnoOdP, M, Q, temena);
}

// funkcija koja vraca vrednost koja je proporcionalna rastojanju
// od tacke A do prave odredjene tackama P i Q
long long rastojanjeOdPrave(Tacka P, Tacka Q, Tacka A) {
  return abs((long long)(A.y - P.y) * (long long)(Q.x - P.x) -
             (long long)(Q.y - P.y) * (long long)(A.x - P.x));
}
```

Složenost ovog algoritma može se opisati rekurentnom jednačinom oblika $T(n) = T(k)+T(n-k-1)+O(n)$, gde je sa k označena veličina jednog, a sa $n-k-1$ veličina drugog problema na koji se vrši podela (nalik algoritmu QuickSort). U najboljem slučaju problem se uvek deli na dva potproblema iste dimenzije pa dobijamo rekurentnu jednačinu $T(n) = 2T(n/2) + O(n)$, čije je rešenje $T(n) = O(n \log n)$. U najgorem slučaju dobijamo rekurentnu jednačinu oblika $T(n) = T(n-1)+O(n)$ čije rešenje zadovoljava jednačinu $T(n) = O(n^2)$. Ovo prilično odgovara na složenost algoritma QuickSort i moguće je dokazati da je prosečna složenost ovog algoritma jednaka $O(n \log n)$.

Literatura

1. Miodrag Živković, *Algoritmi*, Matematički fakultet, Beograd, 2000.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.
3. Robert Sedgewick, Kevin Wayne, *Algorithms*, Addison-Wesley Professional; 4th edition, 2011.
4. Udi Manber, *Introduction to algorithms, a creative approach*, Addison-Wesley, Reading, 1989.
5. Jeff Erickson, *Algorithms*, available at https://jeffe.cs.illinois.edu/teaching/algorithms/#book 2019.
6. Antti Laaksonen, *Guide to Competitive Programming, Learning and Improving Algorithms Through Contests*, Springer, 2017.
7. Data Structures and Algorithms, OpenDSA hypertext project, https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/#
8. Algorithms for Competitive Programming, Editors: Jakob Kogler, Oleksandr Kulkov, Rodion Gorkovenko, https://cp-algorithms.com/
9. Dragan Urošević, Algoritmi i strukture podataka, Računarski fakultet, Beograd.