Čas 3.1, 3.2, 3.3 - Primeri induktivno-rekurzivne konstrukcije

Rešavanje problema rekurzijom

Grejovi kodovi

Problem: Grejov kôd dužine 2^n definiše se kao niz koji sadrži sve zapise n-tocifrenih binarnih brojeva takve da se svaka dva susedna zapisa (kao i prvi i poslednji zapis) razlikuju tačno u jednom bitu. Na primer 00, 01, 11, 10 je Grejov kôd dužine 2^2. Zaista 00 i 01 se razlikuju samo na drugom bitu, 01 i 11 samo na prvom, 11 i 10 samo na drugom, a 10 i 00 samo na prvom bitu. Definisati funkciju koja konstruiše Grejov kôd dužine 2^n za proizvoljno n.

Posmatrajmo nekoliko Grejovih kodova. Grejov kôd dužine 2^0 sadrži samo praznu nisku, dužine 2^1 je 0, 1, dužine 2^2 je 00, 01, 11, 10, dužine 2^3 je 000, 001, 011, 010, 110, 111, 101, 100 itd. Razmotrimo kako je dobijen Grejov kôd dužine 2^3. Prva 4 elementa su dobijena tako što je ispred odgovarajućeg elementa u Grejovom kodu dužine 2^2 dodata nula. Naredna 4 elementa su dobijena tako što je ispred elemenata u Grejovom kodu dužine 2^2 postavljena cifra 1, a onda su tako dobijeni elementi poređani u obratnom redosledu. Ovo je opšta shema koja se može upotrebiti za dobijanje Grejovog koda.

- Ako je dužina Grejovog koda 0, taj kôd sadrži samo praznu nisku.
- Ako je dužina Grejovog koda 2^n tada se Grejov kôd dobija tako što se poređaju redom elementi Grejovog koda dužine 2^{n-1} prošireni početnom nulom, a zatim se u obratnom redosledu poređaju redom elementi Grejovog koda dužine 2^{n-1} prošireni početnom jedinicom.

Dokaz da prethodna procedura daje Grejov može jednostavno izvesti matematičkom inducijom. Bazu inducije čini kôd dužine 2^0 - u njemu postoji jedan zapis dužine 0, nema susednih zapisa, pa je kôd trivijalno Grejov. Pretpostavimo da zapisi g_0, g_1, $\ldots$, $g_{2^{n-1}-1}$ čine Grejov kôd dužine 2^{n-1}. Posmatrajmo niz zapisa $0g_0$, $0g_1$, $\ldots$, $0g_{2^{n-1}-1}$, $1g_{2^{n-1}-1}$, $\ldots$, $1g_1$, $1g_0$ i dokažimo da on čini Grejov kôd. Svaka dva susedna zapisa oblika $0g_i$ i $0g_{i+1}$ se razlikuju samo na jednom bitu, jer im je prvi bit isti, a po induktivnoj pretpostavci se g_i i g_{i+1} razlikuju tačno na jednom bitu. Slično važi i za zapise $1g_i$ i $1g_{i-1}$. Susedni su još zapisi $0g_{2^{n-1}-1}$ i $1g_{2^{n-1}-1}$ i oni se razlikuju samo na prvom bitu i zapisi $1g_0$ i $0g_0$ (poslednji i prvi zapis) i oni se razlikuju samo na prvom bitu. Dakle svaka dva susedna zapisa se razlikuju samo na jednom bitu, pa je kôd Grejov.

Definišimo funkciju koja određuje k-ti po redu zapis Grejovog koda dužine 2^n (podrazumevaćemo da je $0 \leq k < 2^n$). Na osnovu prethodnog razmatranja nju je veoma jednostavno definisati rekurzivno. Ako je n nula, u pitanju je prazna niska, a u suprotnom razlikujemo slučaj kada je $0 \leq k < 2^{n-1}$ u kom se vraća k-ti element Grejovog koda dužine 2^{n-1} dopunjen početnom nulom

i slučaj $2^{n-1} \leq k < 2^n$ u kom se vraća $2^n - 1 - k$-ti elemet Grejovog koda dužine 2^{n-1} dopunjen početnom jedinicom (jer se zapisi u ovom deluređaju obratnim redosledom, pa se, na primer, za $k = 2^{n-1}$ vraća zapis na poziciji $2^n - 1 - 2^{n-1} = 2^{n-1} - 1$, tj. poslednji zapis u manjem Grejovom kodu, a za $k = 2^n - 1$ vraća zapis na poziciji 0 tj. prvi zapis u manjem Grejovom kodu).

Izračunavanje stepena dvojke trivijalno se vrši bitovskim operacijama.

```
string grej(int n, int k) {
   if (n == 0)
      return "";
   if (k < (1 << (n - 1)))
      return "0" + grej(n - 1, k);
   else
      return "1" + grej(n - 1, (1 << n) - 1 - k);
}
```

Primetimo da je ova funkcija prilično efikasna, jer nam za izračunavanje Grejovog koda dužine 2^n na poziciji k nije potrebno poznavanje celog Grejovog koda dužine 2^{n-1} već samo jednog njegovog elementa. Složenost ove funkcije je zato $O(n)$ (a ne $O(2^n)$ što je složenost ukupne konstrukcije celog Grejovog koda). Zaista, prethodna implementacija zadovoljava jednačinu $T(n) = T(n-1) + O(1)$ i $T(0) = 1$, čije je rešenje $O(n)$.

Funkciju možemo realizovati i iterativno.

```
string grej(int n, int k) {
   string rez = "";
   while (n > 0) {
     if (k < 1 << (n-1))
        rez = rez + "0";
     else {
        rez = rez + "1";
        k = (1 << n) - 1 - k;
     }
     n--;
   }
   return rez;
}
```

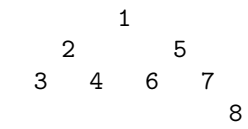
Invarijanta petlje je da je $n_0 \geq n \geq 0$ i da se u promenljivoj `rez` nalazi prefiks dužine $n_0 - n$ traženog Grejovog koda (u svakom koraku petlje taj prefiks produžavamo za jedan karakter).

Napomenimo i da se Grejov kôd može izračunati i direktno, ako se predstavi u obliku neoznačenog broja (zbog vodećih nula dužina tada nije bitna).

```
unsigned grej(unsigned k) {
  return k ^ (k >> 1);
}
```

Određivanje prefiksnog obilaska drveta na osnovu infiksnog i postfiksnog obilaska

Problem: Poznati su infiksni i postfiksni obilazak binarnog drveta koje nije nužno uređeno (čvorovi drveta sadrže karaktere i obilasci su zadati pomoću dve niske iste dužine). Napisati program koji na osnovu njih određuje prefiksni obilazak. Pretpostaviti da svi čvorovi u drvetu sadrže različite vrednosti (tada je rešenje jedinstveno). Na primer, razmotrimo naredno drvo.



Njegov postfiksni obilazak je `34268751` a infiksni `32416578`. Potrebno je da rekonstruišemo prefiksni obilazak koji je u ovom slučaju `12345678`.

Ključni uvid za rešenje zadatka je taj da koren drveta možemo jednostavno odrediti kao poslednji element u postfiksnom obilasku (u našem primeru to je `1`). Tada taj element možemo pronaći i u infiksnom obilasku i na osnovu toga odrediti koji elementi pripadaju levom, a koji desnom poddrvetu (u primeru se određuje da su `324` elementi infiksnog obilaska levog, a `6578` infiksnog obilaska desnog poddrveta). Znajući broj elemenata u svakom poddrvetu, možemo pročitati i njihove prefiksne obilaske (u primeru će to biti `342` i `6875`). Dakle, veoma jednostavno možemo odrediti koren drveta, infiksni i postfiksni obilazak levog poddrveta i infiksni i postfiksni obilazak desnog poddrveta. Ovim smo problem sveli na dva manja potproblema koja se mogu rešavati rekurzivno. Bazu predstavlja slučaj praznog stabla (kada su sva tri obilaska prazna).

Na osnovu prethodne diskusije možemo lako napraviti rekurzivnu implementaciju.

```
string nadjiPre(const string& post, const string& in) {
   if (in == "" && post == "")
      return "";

   char koren = post[post.size() - 1];
   size_t levo = in.find(koren);
   size_t desno = in.size() - levo - 1;

   string in_l   = in.substr(0, levo);
   string in_d   = in.substr(levo + 1, desno);
   string post_l = post.substr(0, levo);
   string post_d = post.substr(levo, desno);
   return koren +
          nadjiPre(post_l, in_l) +
          nadjiPre(post_d, in_d);
}
```

Kreiranje velikog broja pomoćnih niski možemo jednostavno izbeći tako što pamtimo indekse koji određuju delove niski koje se obrađuju.

```
void nadjiPre(const string& post, int post_od, int post_do,
              const string& in, int in_od, int in_do,
              string& pre, int pre_od) {
  if (post_od > post_do && in_od > in_do)
    return;

  char koren = post[post_do];
  size_t levo = in.find(koren, in_od) - in_od;
  size_t desno = (in_do - in_od) - levo;
  pre[pre_od] = koren;
  nadjiPre(post, post_od, post_od + levo - 1,
        in, in_od, in_od + levo - 1,
        pre, pre_od + 1);
  nadjiPre(post, post_od + levo, post_od + levo + desno - 1,
        in, in_od + levo + 1, in_od + levo + desno,
        pre, pre_od + 1 + levo);
}

string nadjiPre(const string& post, const string& in) {
   string pre(post.size(), ' ');
   nadjiPre(post, 0, post.size() - 1,
            in, 0, in.size() - 1,
            pre, 0);
   return pre;
}
```

Recimo i to da bi algoritam ispravno rekonstruisao i stablo u kom neki čvorovi imaju isti sadržaj, ali u tom slučaju rešenje ne bi bilo jedinstveno (u zavisnosti od toga koje pojavljivanje korena u infiksnom obilasku odaberemo, dobijali bismo različita ispravna rešenja). Pokušajte da obrazložite zašto je to tako.

Izvođenje iterativnih algoritama induktivno-rekurzivnim pristupom

Rotiranje niza za k mesta

Problem: Neka je dat niz od n elemenata. Definisati funkciju složenosti $O(n)$ koja njegov sadržaj rotira za k mesta ulevo, bez korišćenja pomoćnog niza.

Jedno, naivno rešenje bilo bi da se niz k puta rotira za po jedno mesto u levo, no to bi bilo prilično neefikasno (složenost bi bila $O(kn)$). Moguće je napraviti rešenje složenosti $O(n)$. Primetimo prvo da uvek možemo da obezbedimo da je $k < n$, jer rotacija za n mesta vraća niz u početnu poziciju, tako da umesto

rotacije za k mesta možemo da uradimo rotaciju za broj koji se dobije kao ostatak pri deljenju k sa n.

Ako bi niz imao $2k$ elemenata, rotacija za k mesta bi se vršila tako što bi se razmenili blokovi elemenata koji čine prvu i drugu polovinu niza. U opštem slučaju situacija je malo komplikovanija, ali ponovo svodi na razmenu blokova elemenata. Neka niz ima n elemenata i neka ga rotiramo za k mesta ulevo. Neka prvih k elemenata čine blok koji ćemo označiti sa L, a preostalih $n-k$ elemenata čine blok koji ćemo označiti sa D. Razmotrimo sledeće slučajeve, u zavisnosti od odnosa dužina ta dva bloka.

- Ako je blokovi L i D jednake dužine, njihovom razmenom se dobija traženo rešenje.
- Ako je blok L kraći, označimo sa D_1 početni deo bloka D dužine k, a sa D_2, preostali deo bloka D. Elementi bloka D_1 treba da budu početni elementi u traženom rešenju i da bismo to postigli, možemo ih razmeniti sa elementima bloka L. Time iz situacije LD_1D_2 dolazimo u situaciju D_1LD_2, dok je traženo rešenje oblika DL, tj. D_1D_2L. Da bismo to postigli, potrebno je da niz LD_2 zarotiramo za dužinu bloka L ulevo (a to je opet k), što je problem istog oblika, ali manje dimenzije od polaznog.
- Ako je blok L duži, označimo sa L_1 početni deo bloka L dužine $n-k$, a sa L_2, preostali deo bloka L. Elementi bloka D treba da budu početni elementi u traženom rešenju i da bismo to postigli, možemo ih razmeniti sa elementima bloka L_1. Time iz situacije L_1L_2D dolazimo u situaciju DL_2L_1, dok je traženo rešenje oblika DL, tj. DL_1L_2. Da bismo to postigli, potrebno je da niz L_2L_1 zarotiramo za dužinu bloka L_2 ulevo (a to je $2k-n$), što je problem istog oblika, ali manje dimenzije od polaznog.

Dakle, zadatak se može rešiti induktivno-rekurzivnom konstrukcijom. Prvi slučaj možemo podvesti pod drugi (ili treći) jer se kompletan levi blok zamenjuje sa desnim, nakon čega ostaje da se razmene prazni blokovi, što ne proizvodi nikakav efekat, pa izlaz iz rekurzije može biti slučaj kada je bilo koji od blokova prazan.

Razmotrimo sledeći primer. Želimo da rotiramo sledeći niz za 4 mesta ulevo.

```
1 2 3 4 5 6 7 8 9 10
```

Nakon rotacije trebalo bi da dobijemo

```
5 6 7 8 9 10 1 2 3 4
```

Rotacija u prethodnom primeru se vrši tako što razmenjujemo blok `1 2 3 4` sa blokom `5 6 7 8 9 10`. Razmenu tih blokova možemo započeti tako što zamenimo ceo blok `1 2 3 4` sa prva 4 elementa bloka `5 6 7 8 9 10`.

Tako dobijamo

```
5 6 7 8   1 2 3 4 9 10
```

Primetimo da je u ovom primeru iza prva četiri elementa bilo više od četiri elementa, pa smo mogli da napravimo zamenu celog levog bloka sa početkom desnog. Brojevi `5 6 7 8` su došli na svoje mesto, a da bi se od ovog niza dobio krajnji rezultat, potrebno je u delu niza iza njih razmeniti blok `1 2 3 4` i blok `9 10`. Taj slučaj je malo drugačiji od prethodnog, jer se iza prva četiri elementa nalazi manje od 4 elementa. Ponovo elementi iza prva četiri dolaze na početak, ali ovaj put je to moguće uraditi samo za dva takva elementa, tako da se zaustavljamo u narednoj situaciji.

```
5 6 7 8    9 10    3 4 1 2
```

Do konačnog rešenja sada možemo doći tako što razmenimo blokove `3 4` i `1 2`. U ovom slučaju je dužina oba bloka jednaka, pa se nakon njihove razmene procedura zaustavlja.

```
5 6 7 8   9 10    1 2   3 4
```

Na osnovu ovog razmatranja nameće se rekurzivni algoritam.

```
// funkcija razmenjuje blok koji pocinje na poziciji p1 i duzine je d1
// i blok koji pocinje na poziciji p2 i duzine je d2
void razmeniBlokove(int a[], int p1, int d1, int p2, int d2) {
  // ako je neki blok prazan nema sta da se razmenjuje
  if (d1 == 0 || d2 == 0)
     return;
  if (d1 <= d2) {
     // razmenjujemo kompletan levi blok sa pocetkom desnog
     razmeni(a, p1, p2, d1);
     // iz situacije L.D1.D2 dosli smo u situaciju D1.L.D2 i
     // da bismo dosli u zeljenu situaciju D1.D2.L moramo
     // da razmenimo L i D2
     razmeniBlokove(a, p1 + d1, d1, p2 + d1, d2 - d1);
  } else {
     // razmenjujemo kompletan desni blok sa pocetkom levog
     razmeni(a, p1, p2, d2);
     // iz situacije L1.L2.D dosli smo u situaciju D.L2.L1 i
     // da bismo dosli u zeljenu situaciju D.L1.L2 moramo
     // da razmenimo L1 i L2
     razmeniBlokove(a, p1 + d2, d1 - d2, p2, d2);
  }
}

void rotiraj(int a[], int n, int k) {
  // razmenjujemo pocetni blok duzine k i ostatak niza
  razmeniBlokove(a, 0, k, k, n - k);
}
```

U implementaciji smo pretpostavili da imamo na raspolaganju funkciju `razmeni(a, p1, p2, m)` koja razmenjuje blokove duzine `m` koji počinju na

pozicijama `p1` i `p2` u nizu `a` (ona se može implementirati trivijalno pomoću jedne petlje i jedne pomoćne promenljive). Njena složenost je $O(m)$.

```
void razmeni(int a[], int p1, int p2, int m) {
   for (int i = 0; i < m; i++)
      swap(a[p1+i], a[p2+i]);
}
```

Centralna mera progresa u algoritmu je zbir dužina blokova koji se razmenjuju. Ona kreće od `n` i smanjuje se sve dok ne dođe do nule. U prvom slučaju se vrši razmena blokova dužine d_1 za šta je potrebno $O(d_1)$ koraka i nakon toga se vrši rekurzivni poziv takav da je zbir dužina blokova upravo za d_1 manji od polaznog zbira dužina (novi zbir je $d_1 + (d_2 - d_1) = d_2$, a polazni $d_1 + d_2$). Slično, u drugom slučaju se vrši zamena blokova dužine d_2 za šta je potrebno $O(d_2)$ koraka i nakon toga se vrši rekurzivni poziv takav da je zbir dužina blokova upravo za d_1 manji od polaznog zbira dužina (novi zbir je $(d_1 - d_2) + d_2 = d_1$, a polazni $d_1 + d_2$). Dakle, rekurentna jednačina je u oba slučaja jednaka $T(n) = T(n - d) + O(d)$ i njeno rešenje je $T(n) = O(n)$.

Još jednostavnije, u svakom koraku petlje unutar funkcije `razmeni` se tačno jedan element dovodi na svoje finalno mesto, odakle se više ne mrda, a u rekurzivnoj funkciji se osim toga (i rekurzivnih poziva) ne vrši nikakav drugi posao. Pošto se po završetku rekurzije svaki element nalazi na svom mestu izvšeno je tačno n razmena.

Ovaj algoritam možemo izraziti veoma jednostavno i iterativno (rekurzija je repna, pa se može lako eliminisati).

```
// rotira niz a duzine n za k mesta ulevo
void rotiraj(int a[], int n, int k) {
  k %= n;
  int p1 = 0, d1 = k;
  int p2 = k, d2 = n - k;
  // razmenjujemo blokove [p1, p1 + d1) i [p2, p2 + d2)
  // razmenu vrsimo dok neki od blokova ne postane prazan
  while (d1 != 0 && d2 != 0) {
     if (d1 <= d2) {
        // razmenjujemo kompletan levi blok sa pocetkom desnog
        razmeni(a, p1, p2, d1);
        // iz situacije L.D1.D2 dosli smo u situaciju D1.L.D2 i
        // da bismo dosli u zeljenu situaciju D1.D2.L moramo
        // da razmenimo L i D2
        p1 += d1; p2 += d1;
        d2 -= d1;
     } else {
        // razmenjujemo kompletan desni blok sa pocetkom levog
        razmeni(a, p1, p2, d2);
        // iz situacije L1.L2.D dosli smo u situaciju D.L2.L1 i
```

```
        // da bismo dosli u zeljenu situaciju D.L1.L2 moramo
        // da razmenimo L1 i L2
        p1 += d2;
        d1 -= d2;
      }
  }
}
```

Na kraju, mozemo ukloniti i pomoćnu funkciju `razmeni` i dobiti veoma elegantan i efikasan algoritam.

```
#include <iostream>
#include <algorithm>
using namespace std;

// maksimalni broj elemenata niza odredjen tekstom zadatka
const int N_MAX = 100000;

int main() {
  // ucitavanje podataka
  int k, n;
  cin >> n >> k;
  int a[N_MAX];
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ciklicko pomeranje za k mesta je isto sto i ciklicko pomeranje za
  // k % n mesta
  k %= n;

  // vrsimo rotiranje niza razmenom blokova
  int l = 0; // pocetak levog bloka
  int d = k; // pocetak desnog bloka
  // razmenjujemo blokove L i D tj. blokove [l, k) i [d, n)
  // razmenu vrsimo dok neki od blokova ne postane prazan
  while (l != k && d != n) {
    // menjamo tekuce elemente u blokovima
    swap(a[l++], a[d++]);

    if (l == k) // stigli smo do kraja levog bloka
      // razmenili smo levi blok sa pocetkom desnog i iz L.D1.D2
      // dosli u situaciju D1.L.D2 i jos treba da razmenimo blokove
      // L i D2, a to su blokovi [l, d) i [d, n),
      // tako da k treba postaviti na d
      k = d;

    if (d == n) // stigli smo do kraja desnog bloka
```

```
      // razmenili smo pocetak levog bloka sa desnim i iz L1.L2.D
      // dosli u situaciju D.L2.L1 i jos treba da razmenimo blokove
      // L2 i L1, a to su blokovi [l, k) i [k, n),
      // tako da d treba postaviti na k
      d = k;
  }

  // ispisujemo rotirani niz
  for (int i = 0; i < n; i++)
    cout << a[i] << endl;

  return 0;
}
```

Bez komentara, centralni deo prethodnog koda izgleda ovako.

```
  int l = 0;
  int d = k;
  while (l != k && d != n) {
    swap(a[l++], a[d++]);
    if (l == k)
      k = d;
    if (d == n)
      d = k;
  }
```

Veoma interesantno bi bilo dokazati direktno da je ovaj algoritam korektan. Za početak ni njegovo zaustavljanje nije trivijalno. Dokažimo da se prethodni algoritam zaustavlja. Jedna od invarijanti prethodne petlje je da je $0 \leq l \leq k \leq d \leq n$. Dokažimo to. Pošto je $0 < n$, $0 \leq k$ i $k < n$, na početku važi $0 = l \leq k = d < n$, pa je invarijanta ispunjena.

Na osnovu uslova petlje, kada se uđe u telo petlje znamo da je $0 \leq l < k \leq d < n$. Nakon toga se l i d uvećavaju za 1, pa važi $0 \leq l \leq k \leq d \leq n$. Ako je $l = k$, tada se k postavlja na d, pa odnos i dalje ostaje da važi. Slično, ako je $d = n$, tada se d postavlja na k, pa odnos i dalje ostaje da važi. Na osnovu ovoga znamo da ova invarijanta ostaje ispunjena i nakon izvršavanja tela petlje. Potrebno je još dokazati da se u nekom trenutku mora dogoditi da je $l = k$ ili da je $d = n$. Međutim, u svakom koraku petlje promenljiva l se uvećava za 1, dok se gornja granica n ne menja. Stoga znamo da će se nakon najviše n koraka desiti da je $l = n$. Na osnovu invarijante tada će morati da važi da je $l = k = d = n$, pa će se petlja zaustaviti. Ujedno vidimo i da se u najgorem slučaju petlja izvršava n puta, pa je algoritam složenosti $O(n)$.

Duplikati u nizu od 0 do $n-1$

Problem: Dat je niz od n brojeva u kom svi elementi pripadaju intervalu od 0 do $n-1$. Neki elementi iz ovog intervala možda nedostaju, a neki se možda ponavljaju. Napisati program koji ispisuje sve one koji nedostaju i sve one koji se ponavljaju (u proizvoljnom redosledu).

Direktno rešenje bi linearnom pretragom za svaki broj proverilo da li se pojavilo 0, 1 ili više puta i na osnovu toga odredilo rezultat. Složenost takvog rešenja bi bila $O(n^2)$, jer bi se svaki nedostajući element tražio n koraka, a njih može biti $n-1$.

Mnogo bolje rešenje bi podrazumevalo da se niz sortira. U sortiranom nizu duplikati se nalaze jedan do drugog, a nedostajuće elemente bismo identifikovali kao one za koje razlika između susedna dva elementa nije 1. Složenošću bi dominiralo sortiranje, koje bi se moglo obaviti u $O(n \log n)$, dok bi druga faza zatim mogla biti urađena u $O(n)$.

Međutim, možemo bolje od toga. Zahvaljujući veoma specifičnom rasponu elemenata niza, sortiranje možemo uraditi posebno prilagođenim algoritmom u vremenu $O(n)$. Pokušavamo da niz uredimo tako da ako se neki element i javlja u nizu, da bar jedno njegovo pojavljivanje bude na njegovoj poziciji i (tj. da je $a_i = i$). Kada se niz uredi tako, onda u jednom prolazu možemo odrediti i duplikate i nedostajuće elemente. Ako se neki element javlja na poziciji koja nije njegova (ako je $a_i \neq i$), tada možemo zaključiti dve stvari. Prvo, pošto se a_i javlja i na svojoj poziciji a_i, ovo je njegovo bar drugo pojavljivanje, pa je to duplikat. Drugo, element i ne postoji u nizu, jer da postoji, morao bi se javiti na svojoj poziciji i. Ostaje samo još pitanje kako efikasno možemo da uredimo niz na opisani način.

Ključna ideja je da stalno pokušavamo da unapredimo stanje niza tako što elemente postavljamo na mesto na kom treba da budu. Krećemo od početka niza i pokušavamo element koji je na tekućem mestu dovedemo na njegovo mesto. Dve situacije su moguće. Ako se na mestu na kom treba da bude tekući element već nalazi odgovarajući element, onda je ili tekući element na svom mestu ili ne može biti stavljen na svoje mesto, jer se tamo već takav element nalazi, pa je tekući element duplikat. U toj situaciji smo završili sa obradom tekućeg elementa i možemo preći na sledeći. Ako se na mestu na kom treba da bude tekući element nalazi neki drugi element, onda se oni mogu razmeniti, čime se povećava broj elemenata koji su na svom mestu. Pošto element koji je tom razmenom došao na tekuću poziciju možda nismo još analizirali, tekuću poziciju ne menjamo.

```
void sortiraj(int a[], int n) {
   int i = 0;
   while (i < n) {
      if (a[a[i]] == a[i])
         i++;
      else
```

```
            swap(a[i], a[a[i]]);
    }
}
```

Pokažimo prvo kako ovaj algoritam radi na jednom primeru.

```
0 1 2 3 4 5 6 7 8
-----------------
.8 1 6 5 0 5 2 3 1      8 dovodimo na njegovo mesto
.1 1 6 5 0 5 2 3 8      1 je duplikat
1 .1 2 5 0 5 6 3 8      1 je na svom mestu
1 1 .2 5 0 5 6 3 8      2 je na svom mestu
1 1 2 .5 0 5 6 3 8      5 je duplikat
0 1 2 5 .1 5 6 3 8      0 dovodimo na njegovo mesto
0 1 2 5 .1 5 6 3 8      1 je duplikat
0 1 2 5 1 .5 6 3 8      5 je na svom mestu
0 1 2 5 1 5 .6 3 8      6 je na svom mestu
0 1 2 5 1 5 6 .3 8      3 dovodimo na njegovo mesto
0 1 2 3 1 5 6 .5 8      5 je duplikat
0 1 2 3 1 5 6 5 .8      8 je na svom mestu
```

Dokažimo korektnost ovog algoritma.

Lema: Važi da je $0 \leq i \leq n$ i da su svi elementi u intervalu $[0, i)$ takvi da je $a_{a_i} = a_i$.

Pretpostavimo da tvrđenje važi na ulasku u petlju. Tada je $i < n$. Ako je $a_{a_i} = a_i$ važi da se niz ne menja i da je $i' = i + 1$. Jasno je da se u tom slučaju invarijanta održava (jer je važila da za sve elemente pre pozicije i i važi i za element na poziciji i). Ako je $a_{a_i} \neq a_i$ tada se vrši razmena elemenata na pozicijama i i a_i. To znači da se nakon razmene na poziciji a_i nalazi element a_i. Ako je $a_i \geq i$, tada se razmenom ne utiče na elemente na pozicijama $[0, i)$ i invarijanta trivijalno ostaje da važi. Ako je $a_i < i$, tada se niz menja, međutim, i u novom nizu će na poziciji a_i biti element koji je jednak a_i, pa će invarijanta ostati da važi.

Teorema: Po završetku petlje, za svaki element $0 \leq i < n$ važi da ako je $a_i \neq i$, tada je a_i duplikat, a i se ne javlja u nizu.

Po završetku petlje za svaki element niza važi da je $a_{a_i} = a_i$. Naime, na osnovu invarijante je $0 \leq i \leq n$, a pošto je petlja završena važi da je $i \geq n$, pa je $i = n$. Ponovo na osnovu invarijante za sve pozicije i iz intervala $[0, n)$, a to su sve pozicije u nizu, znamo da je $a_{a_i} = a_i$. Ako je $a_i \neq i$, znamo da se a_i javlja i na poziciji i i na poziciji a_i, pa je duplikat. Dokažimo i da i ne može da se javi u nizu. Ako bi za neko i' važilo da je $a_{i'} = i$, tada bi važilo $a_{a_{i'}} = a_{i'}$, jer to važi za svaku poziciju, pa i i'. Uvrštavanjem $a_{i'} = i$, dobijamo da bi moralo da važi $a_i = i$, što je direktna kontradikcija u odnosnu na pretpostavku da je $a_i \neq i$.

Dokaz zaustavljanja i analiza složenosti je možda još interesantnija. Algoritam napreduje tako što se ili i pomera ka desnom kraju niza ili tako što se povećava

broj pozicija k takvih da je $a_k = k$. Ako je m broj takvih pozicija, možemo razmatrati broj $i + m$. Dokažimo da svako izvršavanje tela petlje povećava ovaj broj bar za 1. Ako je $a_{a_i} = a_i$, tada se i povećava za 1. U suprotnom, se vrši razmena elemenata na poziciji i i a_i. Na osnovu pretpostavke, na poziciji a_i nije mogao da bude element a_i. Ni na poziciji i nije mogao da bude element a_i, jer da jeste, bilo bi $a_i = i$ i važilo bi da je $a_{a_i} = a_i$, što je opet suprotno u odnosu na pretpostavku. Dakle, pre razmene nijedna od ove dve pozicije nije doprinosila broju m. Nakon razmene, znamo da je sigurno na poziciji a_i element a_i, jer razmenom a_i prebacujemo na poziciju a_i. Ne znamo da li se nakon razmene na poziciji i nalazi a_i, ali to nam nije ni bitno, jer već na osnovu pozicije a_i znamo da je m uvećan bar za jedan (jer se ostali elementi u nizu nisu pomerali tako da oni nastavljaju da doprinose vrednosti m kao i pre razmene). Veličina $i + m$ je odozgo ograničena sa $2n$. Naime znamo da je $i \leq n$, kao i da je broj m odozgo ograničen sa n (jer u nizu dužine n ima ukupno n različitih pozicija). Zbog toga znamo da se telo petlje može izvršiti najviše $2n$ puta, pa je složenost ovog algoritma sortiranja $O(n)$.

Primenimo sada isti ovaj algoritam da rešimo naredno uopštenje problema koji smo već razmatrali.

Problem: Iz niza u koji su bili upisani svi brojevi od 0 do n, izbačeno je k elemenata. Odredi koji su elementi izbačeni.

Zadatak se lako svodi na prethodni tako što se niz dužine $n - k$ proširi do dužine n, popunjavajući ga nekim duplikatima tj. elementima koji se već nalaze u nizu. Najlakše je da na kraj niza dodamo k pojavljivanja elementa a_0 i da zatim primenimo prethodni algoritam za nalaženje nedostajućih elemenata.

Određivanje zvezde

Problem: U nekoj grupi ljudi osoba se naziva zvezda (engl. superstar) ako je svi prisutni znaju, a ona ne poznaje nikoga od prisutnih. Definisati funkciju za ispitivanje da li u datom skupu ljudi postoji zvezda. Data je matrica logičkih vrednosti kojom se određuje ko koga poznaje (na poziciji (i, j) se nalazi vrednost tačno ako i samo ako osoba i poznaje osobu j).

Direktan način je da za svaku osobu proverimo da li zadovoljava uslov zvezde.

```
bool poznajeNekog(bool poznaje[MAX][MAX], int i) {
    for (int j = 0; j < poznaje[i].size(); j++)
       if (i != j && poznaje[i][j])
          return true;
    return false;
}

bool sviJePoznaju(bool poznaje[MAX][MAX], int i) {
    for (int j = 0; j < poznaje.size(); j++)
```

```
        if (i != j && !poznaje[j][i])
            return false;
    return true;
}

int zvezda(bool poznaje[MAX][MAX]) {
   for (int i = 0; i < poznaje.size(); i++)
        if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
            return i;
   return -1;
}
```

Složenost najgoreg slučaja ovog algoritma je $O(n^2)$, mada je fer priznati da se ta složenost veoma teško dostiže, jer je za očekivati da će se obe provere u slučaju kada kandidat nije zvezda prekinuti mnogo pre nego što se stigne do kraja niza.

Poboljšanje možemo pokušati induktivno-rekurzivnim pristupom. Prva ideja je da problem pronalaženja zvezde u skupu od n osoba svodimo na problem pronalaženja zvezde u skupu od $n-1$ osoba.

- Baza indukcije je prazan skup osoba u kome ne postoji zvezda.
- Pretpostavimo kao induktivnu hipotezu da umemo da izračunamo zvezdu u skupu od $n-1$ osoba. Razdvojmo skup od n osoba na podskup od početnih $n-1$ osoba i poslednju osobu o_{n-1}. Zvezda celog skupa može biti ili zvezda tog podskupa ili osoba o_{n-1} (jer ako je osoba zvezda nekog skupa onda je ona ujedno i zvezda svakog podskupa kojem pripada).
    1. Ako u podskupu od $n-1$ osoba postoji zvezda, da bi ona bila zvezda celog skupa potrebno je da je poznaje osoba o_{n-1} i da ona ne poznaje osobu o_{n-1}. To možemo proveriti u vremenu $O(1)$.
    2. U suprotnom (ako u podskupu ne postoji zvezda ili ako zvezda postoji, ali ona poznaje osobu o_{n-1} ili osoba o_{n-1} ne poznaje nju) moramo još ispitati da li je osoba o_{n-1} zvezda. To pitanje nikako ne zavisi od toga da li u podskupu postoji zvezda i da bi se to ispitalo potrebno je posebno proveriti da li svih prethodnih $n-1$ osoba poznaje osobu o_{n-1} i da li ona ne poznaje nikoga od njih. Za to je potrebno $O(n)$ koraka.

Dakle, prethodna konstrukcija nam daje korektan induktivni algoritam, međutim, složenost najgoreg slučaja takvog algoritma je $T(n) = T(n-1) + O(n)$, što je, kao što znamo, $O(n^2)$.

```
int zvezda(const vector<vector<bool>>& poznaje, int n) {
  if (n == 0)
    return -1;
  int z = zvezda(poznaje, n-1);
  if (z != -1 && poznaje[n-1][z] && !poznaje[z][n-1])
    return z;
```

```
  if (!poznajeNekog(poznaje, n-1) && sviJePoznaju(poznaje, n-1))
    return n-1;
  return -1;
}

int zvezda(const vector<vector<bool>>& poznaje) {
  return zvezda(poznaje, poznaje.size());
}
```

Rekurzija se veoma jednostavno uklanja, ali složenost i dalje ostaje nepovoljna.

```
int zvezda(const vector<vector<bool>>& poznaje) {
  int z = -1;
  for (int i = 0; i < poznaje.size(); i++) {
     if (z == -1 || !poznaje[i][z] || poznaje[z][i])
       if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
         z = i;
       else
         z = -1;
  }
  return z;
}
```

Ideja rešenja je da se problem posmatra "unazad". Broj osoba koje nisu zvezde je sigurno mnogo veći od broja osoba koje jesu zvezde, pa je identifikovanje ne-zvezde mnogo jednostavnije od identifikovanja zvezde. Ključna ideja ove efikasne induktivne konstrukcije je da veoma brzo i jednostavno (samo jednim pitanjem) iz svakog skupa možemo ukloniti osobu za koju znamo da nije zvezda i na taj način smanjiti dimenziju problema. Kada u skupu ostane samo jedna osoba, ona je jedini kandidat da bude zvezda polaznog skupa (jer su sve ostale osobe eliminisane na osnovu toga što smo utvrdili da ne mogu biti zvezde). Za tu jedinu preostalu osobu onda možemo direktno ispitati da li je zvezda ili nije. Eliminacija ne-zvezde iz skupa se može izvršiti krajnje jednostavno. Odaberemo proizvoljne dve osobe u skupu i pitamo se da li osoba A zna osobu B. Ako je odgovor potvrdan, onda osoba A ne može biti zvezda (jer zvezda nikoga ne poznaje). Ako je odgovor odričan, onda osoba B nije zvezda (jer zvezdu svi poznaju). Algoritam zasnovan na ovom postupku zadovoljava jednačinu $T(n) = O(1) + T(n-1)$, čije je rešenje $O(n)$. Posle ovoga, preostaje još da se proveri da li je preostali jedini kandidat stvarno zvezda. To je moguće uraditi grubom silom za šta nam je dovoljno $2n-1$ pitanja, tako da je složenost i ove faze $O(n)$, pa je ukupna složenost obe faze $O(n)$, čime se dobija algoritam čiji je najgori slučaj mnogo efikasniji od prethodnog.

Ostaje pitanje tehničke realizacije, odnosno pitanje kako da čuvamo skup kandidata koji nisu još eliminisani i kako da iz njega biramo dve osobe koje ćemo porediti. Jedna mogućnost bi bila da su sve osobe složene na jedan stek (o njima ste već govorili u kursevima programiranja, a još detaljnije ćemo govoriti kasnije). Poredimo dve osobe sa vrha steka i na stek vraćamo samo onu koja nije

eliminisana njihovim poređenjem. Postupak nastavljamo dok stek ne postane jednočlan.

Ipak, biće prikazano još jednostavnije rešenje koje koristi dva pokazivača. Prvi, i, pokazuje na osobu koja je trenutni kandidat za zvezdu, tj. prvu osobu u nizu za koju još nije ustanovljeno da nije zvezda. Drugi, j, pokazuje na osobu za koju se proverava da li je tekući kandidat za zvezdu poznaje. Ukoliko je ne poznaje, i je i dalje kandidat, a j sigurno nije zvezda (jer svi moraju da poznaju zvezdu), pa se j pomera na sledeću osobu. Odavde se vidi da osobe strogo između i i j nisu zvezde. Ukoliko je poznaje, onda i nije zvezda (jer zvezda ne poznaje nikoga), ali j možda jeste, pa, pošto između i i j niko nije zvezda, prvi sledeći kandidat je tekuća osoba j i i se pomera na j, a j na prvu sledeću osobu.

```
int zvezda(const vector<vector<int>>& poznaje) {
    int i = 0; j = 1;
    while (j < n) {
       if (poznaje[i][j])
           i = j;
       j++;
    }
    if (!poznajeNekog(poznaje, i) && sviJePoznaju(poznaje, i))
       return i;
    return -1;
}
```

Lema: Dokažimo da je invarijanta prethodne petlje da nijedan element u intervalu $[0, i)$ i nijedan element u intervalu (i, j) ne može biti zvezda (pri čemu je $0 \leq i < j \leq n$).

- Na početku je $i = 0$ i $j = 1$, pa su oba intervala prazna, a uslov važi (pod pretpostavkom da je $n > 0$).
- Pretpostavimo da uslov važi pri ulasku u petlju. Ako osoba i poznaje osobu j tada ona ne može biti zvezda. Pošto na osnovu pretpostavke znamo da zvezde ne mogu biti ni osobe $[0, i)$ kao ni osobe (i, j) i kako je $i' = j$, znamo da nakon tela petlje zvezde sigurno nisu ni osobe u intervalu $[0, i')$. Pošto je $j' = j + 1$, u tom slučaju je interval (i', j') prazan, pa trivijalno zadovoljava uslov. Ako osoba i ne poznaje osobu j tada znamo da j ne može da bude zvezda. Tada je $i' = i$, a $j' = j + 1$, pa na osnovu pretpostavke znamo da zvezde ne mogu biti osobe iz intervala $[0, i')$, znamo i da ne mogu biti osobe iz intervala (i', j), a pošto ni j ne može biti zvezda, znamo da zvezde ne mogu biti osobe iz intervala (i', j'). Odnos između promenljivih trivijalno ostaje očuvan u oba slučaja.

Teorema: Funkcija ili ispravno pronalazi zvezdu ili vraća -1 ako zvezda ne postoji.

Kada se petlja završi nije $j < n$, pa je na osnovu invarijante $j = n$. Tada znamo da zvezde ne mogu biti osobe iz intervala $[0, i)$ kao ni osobe iz intervala (i, n).

Dakle, jedini kandidat za zvezdu je osoba i. Za nju se eksplicitno proverava traženi uslov, tako da funkcija vraća korektnu vrednost.

Naglasimo još da je vreme učitavanja matrice $O(n^2)$, čime se, nažalost, poništava dobitak na efikasnosti ovog algoritma, pa je vreme realnog izvršavanja programa uključujući fazu učitavanja praktično nepromenjeno ovom divnom optimizacijom.

Apsolutni pobednik na glasanju

Problem: Održano je glasanje i glasalo se za više kandidata. Osoba je apsolutni pobednik ako je dobila strogo više glasova nego svi ostali kandidati zajedno. Definisati algoritam koji na osnovu niza svih glasačkih listića sa glasanja određuje da li postoji apsolutni pobednik i koji je.

Postoji mnogo načina da se ovaj zadatak reši. Ako imamo na raspolaganju efikasne strukture podataka, možemo prebrojati glasove svih kandidata, odrediti onog sa najvećim brojem glasova i videti da li je osvojio strogo više od polovine ukupnog broja glasova. Možemo odrediti medijanu niza glasova i tako utvrditi kandidata za apsolutnog pobednika (jer ako postoji apsolutni pobednik, on se sigurno nalazi na središnjoj poziciji tj. središnje dve pozicije u nizu, po kom god kriterijumu sortirali glasove, dok god su nam glasovi za istog kandidata uzastopni). O efikasnim algoritmima za određivanje medijane (bez sortiranja celog niza) biće više reči kasnije.

Ipak, u ovom poglavlju prikazaćemo sasvim elementaran i izrazito elegantan algoritam za rešenje ovog problema koji su uveli Bojer i Mur u radu *"A fast majority vote algorithm"* (interesantno, u originalnom radu algoritam je uveden u cilju prikaza mogućnosti automatske formalne verifikacije softvera).

Pretpostavimo da želimo da ispitamo postojanje apsolutnog pobednika u skupu od n glasova. Pitanje je kako problem svesti na problem manje dimenzije. Izbacivanje bilo kog pojedinačnog glasa može da promeni rešenje (jer ako apsolutni pobednik ima za jedan glas više od svih ostalih, nakon izbacivanja tog glasa on više neće biti apsolutni pobednik).

Ključni uvid je da ako izbacimo bilo koja dva različita glasa, onda veći skup ima apsolutnog pobednika samo ako ga ima manji i u pitanju je isti apsolutni pobednik. Dokažimo ovo. Ako postoji postoji apsolutni pobednik p u širem skupu od n glasova i on ima m glasova, onda je $m > n/2$. Ako su iz skupa izbačena dva glasa koja nisu za apsolutnog pobednika onda p ima i dalje m glasova, a redukovani skup ima $n-2$ elementa, pa važi $m > n/2 > (n-2)/2$. Ako je izbačen jedan glas za osobu p i jedan koji nije za nju, tada p ima $m-1$ glas, a u skupu ima $n-2$ elementa, pa je $m-1 > (n-2)/2 = n/2-1$, jer je $m > n/2$. Dakle, izbacivanjem dva različita glasa, redukujemo dimenziju problema. Baza indukcije nastaje kada ne možemo više izbacivati parove različitih elemenata. Prazan skup nema apsolutnog pobednika, a neprazan skup u kome ne postoje

dva različita elementa sadrži glasove samo za jednog kandidata koji je očigledno apsolutni pobednik.

Obratimo pažnju na to da postojanje apsolutnog pobednika u manjem skupu i dalje ne implicira to da u većem skupu postoji apsolutni pobednik (jer smo možda apsolutnog pobednika manjeg dobili izbacivanjem dva različita glasa za neke druge dve osobe, čijim ponovnim dodavanjem apsolutni pobednik manjeg skupa prestaje da bude apsolutni pobednik većeg skupa).

Opisali smo, dakle, induktivno-rekurzivnu konstrukciju koja nam može omogućiti da dobijemo kandidata za apsolutnog pobednika (što će se desiti ako skup glasova izbacivanjem parova različitih glasova svedemo na neprazan skup u kom su svi glasovi za istog kandidata) ili da utvrdimo da apsolutni pobednik ne postoji (što će se desiti ako skup glasova izbacivanjem parova različitih glasova svedemo na prazan skup). Naglasimo i da je apsolutni pobednik jedinstven, pa nije moguće da postoje dva različita kandidata za apsolutne pobednike.

Ostaje pitanje pronalaženja efikasne implementacije. Pretpostavimo da se skup svih glasova može podeliti na dva disjunktna podskupa. Svi glasovi u prvom skupu su za istu osobu, dok se u drugom skupu nalaze parovi glasova za različite osobe (ovo će biti centralna invarijanta petlje). Izbacivanjem jednog po jednog para glasova iz drugog skupa, problem se može svesti na sve manju i manju dimenziju, sve dok ne ostanu samo elementi prvog skupa. Ako je prvi skup prazan, možemo zaključiti da ne postoji apsolutni pobednik u početnom skupu svih glasova, a ako nije, onda je osoba za koju su svi glasovi u prvom skupu jedini kandidat za apsolutnog pobednika.

Ostaje pitanje kako da napravimo neku podelu skupa svih glasačkih listića na ova dva podskupa. Krenućemo od toga da su oba podskupa prazna i polako ćemo određivati njihov sadržaj obrađujući jedan po jedan glas. Dovoljno je samo da pamtimo osobu za koju su svi glasovi iz prvog skupa, kao i broj tih glasova. Ako je prvi skup prazan ili ako je tekući glas za osobu za koju su svi glasovi iz prvog skupa, tekući glas možemo dodati u prvi skup (u drugom skupu će se i dalje nalaziti parovi različitih glasova, a svi glasovi u prvom skupu će i dalje biti za istu osobu). Ako prvi skup nije prazan i ako je tekući glas za neku drugu osobu od one za koju su svi glasovi iz prvog skupa, onda ćemo taj glas kao i jedan glas za osobu iz prvog skupa prebaciti u drugi skup (u prvom skupu će i dalje svi glasovi biti za istu osobu, a u drugom će biti svi raniji parovi različitih glasova i ovaj novododati par za koji smo takođe sigurni da je par glasova za različite osobe).

```
bool apsolutniPobednik(int glasovi[], int n, int& pobednik) {
  // određujemo kandidata za pobednika (ako postoji) i broj glasova
  // koji preostane kada se ponište parovi različitih glasova
  int kandidat;
  int glasovaZaKandidata = 0;
  for (int i = 0; i < n; i++)
    if (glasovaZaKandidata == 0 || glasovi[i] == kandidat) {
```

```
      kandidat = glasovi[i];
      glasovaZaKandidata++;
    } else
      glasovaZaKandidata--;

  // proveravamo da li postoji kandidat za pobednika i da li je
  // ostvario više od n/2 glasova
  if (glasovaZaKandidata > 0 &&
      count(glasovi, next(glasovi, n), kandidat) > n / 2) {
    pobednik = kandidat;
    return true;
  } else
    return false;
}
```

Složenost i faze određivanja i faze provere kandidata je očigledno linearna, pa je ceo algoritam složenosti $O(n)$. Osim za čuvanje niza glasova, ne angažuje se nikakva značajna dodatna memorija.

Ojačavanje induktivne hipoteze

U nekim situacijama se dobitak na efikasnosti može postići time što pretpostavimo da je induktivna hipoteza jača tj. da se u svakom koraku umesot jedne izračunava nekoliko različitih vrednosti. Razmotrimo nekoliko takvih primera.

Izračunavanje vrednosti polinoma

Problem: Dat je niz realnih brojeva $a_0, a_1, \ldots, a_n$ i realni broj x. Izračunati vrednost polinoma $P_n(x) = \sum_{i=0}^{n} a_i x^i$.

Jedan način da se problem svede na problem manje dimenzije je da se polinom $P_n(x)$ razloži na zbir $a_n x^n + P_{n-1}(x)$. Izlaz iz rekurzije može predstavljati slučaj $n = 0$ kada je vrednost $P_0(x) = a_0$, a može predstavljati i slučaj $n < 0$ kada je vrednost $P_n(x) = 0$ (jer je suma $\sum_{i=0}^{n} a_i x^i$ prazna, pa joj je zbir po definiciji 0). Da bismo od vrednosti $P_{n-1}(x)$ koju znamo na osnovu induktivne hipoteze mogli da izračunamo $P_n(x)$ moramo da umemo da izračunamo i x^n. Postoje razni načini da se to efikasno uradi, ali to svakako nije trivijalan problem. Ako bismo vrednost stepena izračunavali naivno (množenjem sa x n puta), dobili bismo ovakav kôd.

```
int stepen(int x, int n) {
   if (n == 0)
      return 1;
   return stepen(x, n-1) * x;
}
```

```
int vrednostPolinoma(int a[], int n, int x) {
   if (n == 0)
      return a[0];
   return a[n] * stepen(x, n) + vrednostPolinoma(a, n-1, x);
}
```

Pod pretpostavkom da funkcija `stepen` ispravno vrši stepenovanje, dokažimo da je naša rekurzivna definicija dobra.

Teorema: Funcija `vrednost_polinoma` za dati niz a čija je dužina veća ili jednaka n, izračunava vrednost $P_n(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + \ldots a_1 x + a_0$.

Tvrđenje dokazujemo indukcijom po n. Kao bazu možemo uzeti slučaj $n = 0$ kada funkcija ispravno izračunava zbir $P_0(x) = \sum_{i=0}^{0} a_i x^i = a_0$. Kao induktivnu hipotezu pretpostavimo da rekurzivni poziv za vrednost $n-1$ ispravno izračunava $P_{n-1}(x) = \sum_{i=0}^{n-1} a_i x^i = a_{n-1} x^{n-1} + \ldots a_1 x + a_0$ i da poziv funkcije `stepen` ispravno izračunava x^n. Funkcija vraća zbir vrednosti $a_n \cdot x^n$ i vrednosti rekurzivnog poziva za koji na osnovu induktivne hipoteze znamo da je jednak $P_{n-1}(x) = a_{n-1} x^{n-1} + \ldots a_1 x + a_0$. Zato je povratna vrednost jednaka $P_n(x) = a_n x^n + \ldots a_1 x + a_0$, što je i trebalo dokazati.

Naravno, mnogo prirodnije za imperativne jezike je osloboditi se rekurzije, što je u ovom slučaju prilično jednostavno.

```
int v = 0;
for (int i = 0; i <= n; i++) {
   int s = 1;
   for (int j = 0; j < i; j++)
      s *= x;
   v += a[i] * s;
}
```

Broj množenja u ovom kodu je prilično očigledno $O(n^2)$.

Efikasnost možemo značajno popraviti ako primetimo da vrednost x^n možemo izračunati *inkrementalno*, ne računajući stepen iz početka, već na osnovu prethodno izračunate vrednosti x^{n-1}. Time *ojačavamo induktivnu hipotezu* i gradimo funkciju za koju ne pretpostavljamo samo da ume da izračuna vrednost $P_n(x)$ nego ujedno i vrednost x^{n+1}.

Dve povratne vrednosti možemo realizovati preko para ili preko prenosa po referenci (ili preko pokazivača).

```
pair<int, int> vrednostPolinoma(int a[], int n, int x) {
   if (n == 0)
      return make_pair(a[0], x);
   int v, s;
   tie(v, s) = vrednostPolinoma(a, n-1, x);
```

```
   return make_pair(a[n] * s + v, s * x);
}

void vrednostPolinoma(int a[], int n, int x, int& v, int& s) {
   if (n == 0) {
      v = a[0]; s = x;
      return;
   }
   vrednostPolinoma(a, n-1, x, v, s);
   v = a[n] * s + v;
   s = s * x;
}
```

Dokažimo korektnost ove implementacije.

Teorema: Funcija `vrednost_polinoma` za dati niz a čija je dužina veća ili jednaka n, izračunava par koji čini vrednost polinoma $P_n(x) = \sum_{i=0}^{n} a_i x^i = a_n x^n + \ldots a_1 x + a_0$ i vrednost x^{n+1}.

Bazu indukcije čini slučaj $n = 0$ i tada funkcija ispravno izračunava vrednost $\sum_{i=0}^{0} a_i x^i = a_0$ i $x^{0+1} = x$. Pretpostavimo da rekurzivni poziv ispravno izračunava par vrednosti $v = \sum_{i=0}^{n-1} a_i x^i$ i $s = x^{(n-1)+1} = x^n$. Funkcija tada uspešno izračunava par vrednosti $v' = a_n \cdot s + v$, što je na osnovu induktivne hipoteze jednako $a_n x^n + \sum_{i=0}^{n-1} a_i x^i = \sum_{i=0}^{n} a_i x^i$ i vrednosti $s' = s \cdot x$, što je na osnovu induktivne hipoteze jednako $x^n \cdot x = x^{n+1}$, čime je tvrđenje dokazano.

Ojačavanje induktivne hipoteze se u ovom slučaju ogledalo u tome da umesto jedne vrednosti, funkcija izračunava i vraća par vrednosti. Vrednost stepena je korišćenja da bi se izračunala nova vrednost polinoma i jako je zgodno bilo što smo i nju dobili direktno iz rekurzivnog poziva. To što smo dobili, moramo da nadoknadimo time što se nakon rekurzivnog poziva moramo malo više potruditi da izračunamo i vratimo ne samo vrednost polinoma koja nas zanima, nego i vrednost stepena, koja će nam biti potrebna na narednom nivou rekurzije.

Osnovna teorema indukcije glasi

$$A(0) \wedge (\forall n)(A(n) \Rightarrow A(n+1)) \Rightarrow (\forall n)(A(n)).$$

Ojačavanje induktivne hipoteze se dakle svodi na to da umesto hipoteze $A(n)$ pretpostavimo ojačanu hipotezu oblika $A(n) \wedge B(n)$. Teorema indukcije u ovom slučaju glasi

$$(A(0) \wedge B(0)) \wedge ((\forall n)(A(n) \wedge B(n) \Rightarrow A(n+1) \wedge B(n+1)) \Rightarrow (\forall n)(A(n)).$$

Prethodno rekurzivno rešenje vraća uređen par brojeva i direktno odgovara pomenutoj induktivno-rekurzivnoj konstrukciji, ali je prilično neprirodno za

imperativni jezik. Mnogo prirodnije je osloboditi se rekurzije. Iterativni kôd se može napisati na sledeći način.

```
int v = 0, s = 1;
for (int i = 0; i <= n; i++) {
    v += a[i] * s;
    s *= x;
}
```

Dokažimo korektnost ove iterativne implementacije korišćenjem invarijante petlje.

Lema: Invarijanta petlje je da promenljiva `v` sadrži vrednost $P_{i-1}(x)$, a promenljiva `s` vrednost x^i, kao i da je $0 \leq i \leq n+1$.

Pre petlje je $i = 0$, $s = 1 = x^0$ a $v = 0$ (što je vrednost $P_{-1}(x)$). Efekat tela petlje je to da je $v' = v + a_i \cdot s$, da je $s' = s \cdot x$ i da je $i' = i + 1$, pa je, na osnovu pretpostavke $v' = a_i \cdot x^i + P_{i-1}(x) = P_i(x) = P_{i'-1}(x)$ i da je $s' = x^i \cdot x = x^{i+1} = x^{i'}$.

Teorema: Nakon petlje promenljiva `v` sadrži vrednost $P_n(x)$.

Na osnovu invarijante znamo da na kraju petlje važi $i = n + 1$ (jer nije $i \leq n$, a važi $0 \leq i \leq n + 1$). Opet na osnovu invarijante znamo da promenljiva `v` sadrži $P_{i-1}(x)$, što je jednako $P_n(x)$, jer je $i = n + 1$.

Prilično je očigledno i da je broj množenja u ovom kodu linearan tj. $O(n)$, pa je ojačavanje induktivne hipoteze pomoglo da se dođe do efikasnijeg algoritma.

Na kraju, recimo i da postoji drugačija induktivno-rekurzivna konstrukcija koja pomaže da se problem reši i jednostavnije i efikasnije (u ovoj varijanti nije nam potrebno ojačavanje induktivne hipoteze). Ključna ideja je da izračunavanje vršimo sdesna nalevo, umesto sa leva na desno. Naime, polinom $P_n(x)$ možemo zapisati kao $\left(a_n x^{n-1} + a_{n-1} x^{n-2} + a_1\right) \cdot x + a_0$. Dakle, prvo određujemo vrednost polinoma stepena $n - 1$ čiji su koeficijenti svi osim poslednjeg, i onda tu vrednost objedinjavamo sa poslednjim koeficijentom. Ova tehnika poznata je kao *Hornerova šema*.

```
int vrednostPolinoma(int a[], int i, int n, int x) {
   if (i > n)
      return 0;
   return vrednostPolinoma(a, i+1, n, x)*x + a[i];
}

int vrednostPolinoma(int a[], int n, int x) {
   return vrednostPolinoma(a, 0, n, x);
}
```

Oslobađanje od rekurzije daje mnogo prirodniju formulaciju.

```
int v = 0;
for (int i = n; i >= 0; i--)
```

```
v = v*x + a[i];
```

Dokažimo korektnost ove implementacije korišćenjem invarijante petlje.

Lema: Invarijanta petlje je da je za $-1 \leq i \leq n$ vrednost promenljive `v` jednaka $\sum_{k=i+1}^{n} a_k x^{k-i-1}$.

U početku je $i = n$, a $v = 0$, pa tvrđenje važi jer je suma prazna, pa joj je zbir po definiciji nula. Efekat tela petlje je da je $v' = v \cdot x + a_i$, pa je, na osnovu pretpostavke, $v' = \left(\sum_{k=i+1}^{n} a_k x^{k-i-1}\right) \cdot x + a_i = \sum_{k=i+1}^{n} a_k x^{k-i} + a_i = \sum_{k=i}^{n} a_k x^{k-i-1}$. Pošto je $i' = i - 1$ i pošto je $i \geq 0$, invarijanta ostaje očuvana.

Teorema: Nakon petlje promenljiva `v` sadrži vrednost $P_n(x)$.

Pošto uslov petlje nije ispunjen znamo da je nakon petlje $i < 0$. Na osnovu invarijante znamo da je $-1 \leq i \leq n$, pa zato nakon petlje mora da važi $i = -1$. Na osnovu invarijante promenljiva v sadrži vrednost, $\sum_{k=i+1}^{n} a_k x^{k-i-1}$, što je za $i = -1$ jednako $\sum_{k=0}^{n} a_k x^k$, što je tražena vrednost polinoma $P_n(x)$.

I u ovom algoritmu je broj množenja $O(n)$, međutim, ovde imamo jedno množenje i jedno sabiranje po koraku tela petlje, a u prethodnom smo imali dva množenja i jedno sabiranje, pa je ova implementacija za konstantni faktor brža.

Faktor ravnoteže binarnog drveta

Problem: Definišimo u ovom problemu visinu čvora binarnog drveta kao broj čvorova na putanji od tog čvora do njemu najudaljenijeg lista. Faktor ravnoteže čvora v binarnog drveta definišimo kao razliku visina njegovog levog i desnog poddrveta. Definisati funkciju koja za svaki čvor drveta izračunava faktor ravnoteže.

Pretpostavićemo da je drvo predstavljeno sledećom strukturom.

```
struct cvor {
  int faktorRavnoteze;
  int vrednost;
  cvor *levo, *desno;
};
```

Prazno stablo predstavlja se specijalnim pokazivačem `nullptr`.

Direktna induktivno-rekurzivna konstrukcija u ovom primeru ne daje rezultate jer faktor ravnoteže čvora ne zavisi od faktora ravnoteže levog i desnog poddrveta (tj. njihovih korenih čvorova), već od visine tih drveta. Sa druge strane, visinu drveta veoma jednostavno možemo izračunati na osnovu induktivno-rekurzivne konstrucije.

```
int visina(cvor* koren) {
  // visina praznog stabla je nula
  if (koren == nullptr)
```

```
    return 0;
  // visina untrašnjeg čvora
  return max(visina(koren->levo), visina(koren->desno)) + 1;
}
```

Ako pretpostavimo da se levo i desno nalazi isti broj elemenata ova funkcija zadovoljava jednačinu $T(n) = 2 \cdot T(n/2) + O(1)$, čije je rešenje na osnovu master teoreme $O(n)$. Složenost je takva i bez te pretpostavke (ako je drvo degenerisano u listu, složenost zadovoljava jednačinu $T(n) = T(n-1) + O(1)$ čije je rešenje takođe $O(n)$). I intuitivno je jasno da funkcija u svakom pozivu mora da obiđe i levo i desno podstablo da bi odredila visinu, tako da funkcija obilazi sve čvorove (jednom) i stoga je linearne složenosti u odnosu na broj čvorova stabla.

Sa ovom funkcijom na raspolaganju, faktor ravnoteže možemo izračunati veoma jednostavno.

```
// izracunava faktore ravnoteze svih cvorova
void izracunajFaktoreRavnoteze(cvor* koren) {
  if (koren != nullptr) {
    izracunajFaktoreRavnoteze(koren->levo);
    izracunajFaktoreRavnoteze(koren->desno);
    koren -> faktorRavnoteze =
      abs(visina(koren->levo) - visina(koren->desno));
  }
}
```

Pod pretpostavkom da je drvo koliko-toliko balansirano, složenost ove funkcije zadovoljava jednačinu $T(n) = 2 \cdot T(n/2) + O(n)$ i njena je složenost, znamo na osnovu master teoreme, $O(n \log n)$. Ako je drvo izdegenerisano u listu (što se može desiti u najgorem slučaju), dobija se da vreme izvršavanja zadovoljava jednačinu $T(n) = T(n-1) + O(n)$ čije je rešenje $O(n^2)$.

Algoritam se može poboljšati ako se visina i faktor ravnoteže računaju istovremeno, tj. ako pojačamo induktivnu hipotezu i pretpostavimo da rekurzivnim pozivima možemo da izračunamo i faktore ravnoteže i visinu poddrveta. Bazu čini slučaj praznog stabla čija je visina nula i koje ne sadrži čvorove čiji faktor ravnoteže treba izračunati. Ako pretpostavimo da umemo da izračunamo faktore ravnoteže svih čvorova poddrveta, kao i njihove visine, tada faktor ravnoteže korena možemo jednostavno izračunati kao apsolutnu vrednost razlike tih visina, a visinu celog drveta kao broj za jedan veći od maksimuma visine levog i desnog poddrveta. Implementaciju je jednostavno napraviti.

```
// izracunava faktore ravnoteze svih cvorova i vraca visinu stabla
int izracunajFaktoreRavnoteze(cvor* koren) {
    if (koren == nullptr)
      return 0;
    int visina_levo = izracunajFaktoreRavnoteze(koren->levo);
    int visina_desno = izracunajFaktoreRavnoteze(koren->desno);
    koren->faktorRavnoteze = abs(visina_levo - visina_desno);
```

```
    return max(visina_levo, visina_desno) + 1;
}
```

Pod pretpostavkom da je drvo balansirano, složenost ovog algoritma zadovoljava jednačinu $T(n) = 2 \cdot T(n/2) + O(1)$, čije je rešenje $O(n)$. Čak i kada drvo nije balansirano, složenost je linearna (jer je rešenje jednačine $T(n) = T(n-1) + O(1)$ takođe $O(n)$).

Prvi algoritam drvo od milion nasumično odabranih elemenata obrađuje za oko $0,672$ sekundi, a drugi za oko 0.0731 sekundi. Za drvo degenerisano u listu od 10,000 elemenata prva implementacija izračunava faktore ravnoteže za $0,003$ sekunde, a druga za $0,133$ sekunde (usled razlike između linearne i kvadratne složenosti).

Red je primetiti i da se zbog velike cene dinamičke alokacije čvorova drvo od milion elemenata formira za oko $0,750$ sekundi i briše za $0,128$ sekundi, tako da su oba vremena izračunavanja faktora ravnoteže znatno manja od vremena kreiranja i brisanja drveta (pod pretpostavkom kolike-tolike balansiranosti koja je opravdana u slučaju ubacivanja nasumično odabranih elemenata opravdano je očekivati je da će složenost kreiranja drveta biti $O(n \log n)$, uz veliki konstantni faktor, dok će složenost brisanja uvek biti linearna $O(n)$). U slučaju izdegenerisanog drveta od 10,000 elemenata, formiranje traje oko $0,166$ sekundi, ne toliko zbog alokacije, koliko zbog kvadratne složenosti koja se javlja.

Dijametar binarnog drveta

Problem: Rastojanje između dva čvora binarnog drveta je broj grana na jedinstvenom putu koji ih povezuje. Dijametar drveta je najveće moguće rastojanje dva njegova čvora. Konstruisati efikasan algoritam za određivanje dijametra datog drveta i odrediti njegovu vremensku složenost.

Problem se jednostavno rešava korišćenjem induktivno-rekurzivne konstrukcije. Najduži put između dva čvora ili prolazi ili ne prolazi kroz koren. Ako ne prolazi, onda se oba čvora nalaze ili u levom ili u desnom poddrvetu, pa se rastojanje rastojanje između njih može odrediti na osnovu induktivne hipoteze. Da bismo odredili najduži put koji prolazi kroz koreni čvor, potrebno je da znamo visinu levog i desnog poddrveta (visina je jednaka broju grana na najdaljem putu između korena i lista).

```
int visina(cvor* koren) {
  if (koren == nullptr)
     return 0;
  return max(visina(koren->levo), visina(koren->desno)) + 1;
}

int dijametar(cvor* koren) {
```

```
  if (koren == nullptr)
    return 0;
  int dijametar_l = dijametar(koren->levo);
  int dijametar_d = dijametar(koren->desno);
  int dijametar_c = visina(koren->levo) + 2 + visina(koren->desno);
  return max({dijametar_l, dijametar_c, dijametar_d});
}
```

Slično kao i u prethodnom zadatku, složenost funkcije visina je $O(n)$, dok je složenost izračunavanja dijametra $O(n \log n)$ u slučaju balansiranog stabla tj. $O(n^2)$ u opštem slučaju.

Ponovo možemo ojačati induktivnu hipotezu i visinu izračunavati paralelno sa dijametrom.

```
void visina_i_dijametar(cvor* koren, int& visina, int& dijametar) {
  if (koren == nullptr) {
     visina = 0;
     dijametar = 0;
     return;
  }
  int visina_l, dijametar_l;
  visina_i_dijametar(koren->levo, visina_l, dijametar_l);
  int visina_d, dijametar_d;
  visina_i_dijametar(koren->desno, visina_d, dijametar_d);
  int dijametar_c = visina_l + 2 + visina_d;
  visina = max(visina_l, visina_d) + 1;
  dijametar = max({dijametar_l, dijametar_c, dijametar_d});
}
```

U ovom slučaju složenost najgoreg slučaja je $O(n)$, čak i kada drvo nije balansirano.

Segment maksimalnog zbira

Vratimo se sada problemu određivanja segmenta maksimalnog zbira koji smo već ranije razmatrali i prikažimo još nekoliko rešenja zasnovanih na induktivno-rekurzivnoj konstrukciji.

Problem: Definisati efikasnu funkciju koja pronalazi najveći mogući zbir segmenta (podniza uzastopnih elemenata) datog niza brojeva. Proceni joj složenost.

Kadanov algoritam (dinamičko programiranje)

Prikažimo sada još nekoliko algoritama linearne složenosti za rešavanje ovog problema. Možda najpoznatiji algoritam je Kadanov algoritam, koji se ubraja u algoritme dinamičkog programiranja (o kojima će više biti reči kasnije).

Pokušavamo da algoritam zasnujemo na induktivnoj konstrukciji. Za prazan niz, jedini segment je prazan i njegov je zbir nula (to je ujedno najveći zbir koji se može dobiti). Smatramo da umemo da problem rešimo za proizvoljan niz dužine n i na osnovu toga pokušavamo da rešimo zadatak za niz dužine $n + 1$ (polazni niz proširen jednim dodatnim elementom). Segment najvećeg zbira u proširenom nizu se ili ceo sadrži u polaznom nizu dužine n ili čini sufiks proširenog niza, tj. završava se na poslednjoj poziciji (uključujući i mogućnost da je tu i prazan segment). Na osnovu induktivne hipoteze znamo da izračunamo najveći zbir segmenta. Jedan način je da prilikom svakog proširenja niza iznova analiziramo sve segmente koji se završavaju na tekućoj poziciji, ali čak iako to radimo inkrementalno (krenuvši od praznog sufiksa, pa dodajući unazad jedan po jedan element) najviše što možemo dobiti je algoritam kvadratne složenosti (probajte da se uverite da je to zaista tako). Ključni uvid je to da najveći zbir sufiksa koji se završava na tekućoj poziciji možemo inkrementalno izračunati znajući najveći zbir segmenta koji se završava na prethodnoj poziciji (tj. najvećeg sufiksa niza pre proširenja). Naime, ako je zbir najvećeg segmenta koji se završava na prethodnoj poziciji i tekućeg elementa pozitivan, onda je to upravo najveći zbir sufiksa proširenog niza (prazan segment ima zbir nula, pa je njegov zbir manji od pronađenog pozitivnog zbira segmenta, a neprazni sufiksi moraju da uključe tekući element i neki sufiks niza pre proširenja, pa nam je jasno da zbir tog sufiksa mora biti određen optimalno). Ako je zbir najvećeg sufiksa pre proširenja niza i tekućeg elementa negativan, onda je optimalan zbir sufiksa nakon proširenja niza 0 (uzimamo prazan segment).

Dakle, proširićemo induktivnu hipotezu i pretpostavićemo da za niz umemo da izračunamo najveći zbir segmenta, ali i najveći zbir sufiksa.

Ako bismo formirali rekurzivnu funkciju koja vrši takvo izračunavanje, dobili bismo neefikasan algoritam jer bi se isti pozivi ponavljali više puta. Umesto toga možemo napraviti iterativan algoritam kome je invarijanta da u svakom koraku petlje znamo ove dve vrednosti (maksimum segmenta i maksimum sufiksa).

```
int maxSufiks = 0, maxSegment = maxSufiks;
for (int i = 0; i < n; i++) {
  maxSufiks += a[i];
  if (maxSufiks < 0)
    maxSufiks = 0;
  if (maxSegment < maxSufiks)
    maxSegment = maxSufiks;
}

cout << maxSegment << endl;
```

Analiza složenosti je veoma jednostavna. Telo petlje koje je konstantne složenosti se izvršava tačno n puta, pa je složenost ovog algoritma takođe $O(n)$.

Zbirovi prefiksa (parcijalne sume niza)

Još jedan algoritam kojim možemo efikasno rešiti ovaj zadatak se zasniva na korišćenju *zbirova prefiksa*. Naime, ako znamo zbir svakog prefiksa niza, onda zbir svakog segmenta možemo dobiti kao razliku zbirova dva prefiksa. Naime, važi da je

$$\sum_{i=l}^{d} a_i = \sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$$

Računamo da je zbir praznog segmenta $\sum_{i=0}^{-1} a_i$ po definiciji jednak nuli.

Obratite pažnju na to da je ova tehnika zapravo analogon Njutn-Lajbnicove formule koju ste sretali tokom izučavanja matematičke analize.

Ponovo koristimo induktivnu konstrukciju iz prethodnog zadatka i niz proširujemo za jedan po jedan element. Da bismo umeli da izračunamo maksimum segmenta proširenog niza potrebno je da znamo maksimum segmenta polaznog niza i da izračunamo maksimalni sufiks proširenog niza. Na osnovu razlaganja na zbirove prefiksa, maksimalni zbir sufiksa proširenog niza se dobija kao razlika zbira celog proširenog niza (tj. zbira prefiksa do tekuće pozicije) i zbira nekog prefiksa neproširenog niza (prazan sufiks ne moramo analizirati, jer je prazan niz već uzet u obzir u sklopu baze indukcije). Pošto je umanjenik konstantan, da bismo maksimizovali razliku potrebno da znamo najmanji mogući umanjilac, tj. da znamo najmanji zbir prefiksa koji se završava na nekoj poziciji ispred tekuće. I tekući i minimalni zbir prefiksa možemo održavati inkrementalno. Kada niz proširimo jednim elementom, zbir prefiksa uvećavamo za taj element, poredimo ga sa dotadašnjim minimalnim zbirom prefiksa i ako je manji, ažuriramo minimalni zbir prefiksa. Naravno, održavamo i globalni maksimalni zbir segmenta koji ažuriramo svaki put kada naiđemo na segment (sufiks) čiji je zbir veći od dotadašnjeg maksimuma.

Dakle, i u ovom rešenju je induktivna hipoteza pojačana i pretpostavljamo da pored segmenta najvećeg zbira u obrađenom delu niza umemo da odredimo i maksimalni zbir sufiksa obrađenog dela niza.

```
int zbirPrefiksa = 0;
int minZbirPrefiksa = zbirPrefiksa;
int maxZbir = 0;
for (int i = 0; i < n; i++) {
  zbirPrefiksa += a[i];
  int zbirSufiksa = zbirPrefiksa - minZbirPrefiksa;
  if (zbirSufiksa > maxZbir)
    maxZbir = zbirSufiksa;
  if (zbirPrefiksa < minZbirPrefiksa)
    minZbirPrefiksa = zbirPrefiksa;
}
cout << maxZbir << endl;
```

Analiza složenosti je veoma jednostavna. Telo petlje koje je konstantne složenosti se izvršava tačno n puta, pa je složenost ovog algoritma takođe $O(n)$.

Zbirovi prefiksa - maksimalni zbir segmenta deljiv sa k

Rešenje sa zbirovima prefiksa je pogodno, jer se može uopštiti i na neke srodne probleme. Razmatrajmo naredno uopštenje polaznog zadatka.

Problem: Dat je niz od n nenegativnih celih brojeva. Definisati efikasan algoritam koji određuje najveći zbir koji ima neki njegov segment (podniz uzastopnih elemenata) koji je deljiv datim brojem k.

Invarijanta petlje je da za svaku vrednost i za svaki ostatak o od 0 do $k-1$ u nizu `minZbirPrefiksaModK` pamtimo najmanji zbir svih prefiksa oblika $[0, j)$ za $0 \leq j \leq i$ koji pri deljenju sa k daje ostatak o tj. -1 ako takav prefiks ne postoji, da u promenljivoj `zbirPrefiksa` čuvamo vrednost prefiksa $[0, i)$, a da u promenljivoj `maxZbir` čuvamo maksimalni zbir svih segmenata $[a, b]$ za $0 \leq a \leq b < i$.

Za $i = 0$ jedini prefiks je $[0, 0)$ tj. prazan prefiks čiji je zbir 0 (on pri deljenju sa k daje ostatak 0). Za ostale ostatke ne postoji prefiks koji se završava do pozicije i čiji bi zbir dao te ostatke. Zato niz `minZbirPrefiksaModK` inicijalizujemo tako da na poziciju 0 upisujemo 0, a na ostale pozicije upisujemo -1. Promenljivu `zbirPrefiksa` inicijalizujemo na nulu, isto kao i `maxZbir` (prazan prefiks ima zbir nula, koji jeste deljiv sa k).

Nakon tela petlje vrši se uvećanje promenljive `i` za jedan tako da je $i' = i + 1$. Promenljiva `zbirPrefiksa` je na ulazu u petlju sadržala vrednost zbira svih elemenata na pozicijama $[0, i)$, i da bi nakon izvršavanja tela petlje sadržala vrednosti na pozicijama $[0, i') = [0, i]$ potrebno je uvećati je za a_i. Od svih prefiksa oblika $[0, j)$, za $j \leq i'$ jedini koji ranije nije razmatran je $[0, i')$, čiji je zbir sadržan u promenljivoj `zbirPrefiksa` nakon njenog uvećanja. Niz `minZbirPrefiksaModK` je potrebno ažurirati u skladu sa pojavom tog novog prefiksa. Jedino mesto u nizu koje se može promeniti je ono koje odgovara ostatku pri deljenju tog zbira sa k. Ako na tom mestu u nizu stoji -1 znači da je ovo prvi prefiks čiji zbir pri deljenju sa k daje baš taj ostatak, pa je ujedno i najmanji i u skladu sa tim na mesto tekućeg ostatatka upisujemo tekući zbir prefiksa. Ako ne stoji -1, znači da se ranije već javljao neki prefiks čiji zbir daje isti ostatak. Pošto su elementi polaznog niza po pretpostavci nenegativni, taj kraći prefiks mora da ima manji (ili eventualno jednak) zbir, pa u tom slučaju niz najmanjih zbirova prefiksa ne menjamo. U tom slučaju znamo da smo našli dva različita prefiksa koji daju isti ostatak, pa smo našli i segment (sufiks koji se završava na poziciji i) čiji je zbir deljiv brojem k, pa je potrebno taj sufiks još uporediti sa tekućim maksimumom i ako je potrebno ažurirati maksimum.

```
vector<int> minZbirPrefiksaModK(k, -1);
minZbirPrefiksaModK[0] = 0;
```

```
int zbirPrefiksa = 0;
int maxZbir = zbirPrefiksa;
for (int i = 0; i < n; i++) {
  zbirPrefiksa += a[i];

  if (minZbirPrefiksaModK[zbirPrefiksa % k] == -1)
      minZbirPrefiksaModK[zbirPrefiksa % k] = zbirPrefiksa;
  else {
    int zbirSufiksa =
        zbirPrefiksa - minZbirPrefiksaModK[zbirPrefiksa % k];
    if (zbirSufiksa > maxZbir)
       maksZbir = zbirSufiksa;
  }
}

cout << maxZbir << endl;
```

Za vežbu pokušajte da napišete program koji određuje koliko ima segmenata u nizu prirodnih brojeva čiji je zbir paran broj.

Broj rastućih segmenata

Vratimo se na problem određivanja broja rastućih segmenata datog niza prirodnih brojeva.

U prethodnim algoritmima u kojima smo analizirali sve segmente, prvo smo analizirali broj rastućih segmenata koji počinju na poziciji 0, zatim broj rastućih segmenata koji počinju na poziciji 1, zatim na poziciji 2, itd., sve do pozicije $n-2$ (na poziciji $n-1$ ne može počinjati ni jedan rastući segment, jer zahtevamo da su segmenti bar dvočlani).

Problem možemo rešiti i efikasnije ako promenimo redosled obilaska i umesto fiksiranja levog kraja, fiksiramo desni kraj segmenata. Dakle, želimo da pronađemo broj rastućih segmenata koji se završavaju na poziciji 1, zatim koji se završavaju na poziciji 2, zatim na poziciji 3, itd., sve do pozicije $n-1$ (na poziciji 0 se ne može završiti rastući segment, jer zahtevamo da su segmenti bar dvočlani).

Ključni uvid je da broj segmenata koji se završavaju na poziciji j možemo odrediti veoma jednostavno inkrementalno, znajući broj segmenata koji se završavaju na poziciji $j-1$. Naime, ako je $a_j \leq a_{j-1}$ na poziciji j se ne završava ni jedan rastući segment. U suprotnom, svaki rastući segment koji se završavao na poziciji a_{j-1} može biti produžen elementom a_j, a rastući je i dvočlani segment $[a_{j-1}, a_j]$.

Dakle, ojačavamo induktivnu hipotezu pretpostavljajući da za svaku poziciju j pored broja svih rastućih segmenata među elementima niza zaključno sa pozicijom j umemo da izračunamo i broj rastućih segmenata koji se završavaju na poziciji j.

Na osnovu ovog zapažanja možemo izgraditi naredni, veoma efikasni algoritam.

```
// ukupan broj rastucih serija
int ukupanBrojRastucih = 0;
// broj rastucih koji se zavrsavaju na tekucoj poziciji
int brojRastucih = 0;
for (int i = 1; i < n; i++) {
  if (a[i] > a[i-1]) {
    // tekuci element produzava sve rastuce segmente koji su
    // se zavrsili na prethodnoj poziciji i dodaje jos jedan
    // nov dvoclan rastuci segment
    brojRastucih++;
    // dodajemo broj rastucih koji se zavrsavaju na poziciji i na
    // ukupan broj rastucih segmenata
    ukupanBrojRastucih += brojRastucih;
  } else {
    // na tekucoj poziciji se ne zavrsava ni jedan rastuci segment
    brojRastucih = 0;
  }
}
```

Složenost ovog postupka je očigledno $O(n)$.

Da zaključimo, videli smo nekoliko različitih algoritama za rešavanje nekih odabranih problema, koji su se razlikovali po svojoj složenosti. Ključna opaska je da su mnogi od njih nastali korišćenjem prilično standardnog dijapazona alatki koje dobar algoritmičar ima u svom arsenalu: tehnika grube sile, optimizacija izvođenjem eksplicitne formule, optimizacija sortiranjem, optimizacije na osnovu inkrementalnosti, tehnika dva pokazivača, induktivno-rekurzivna konstrukcija, tehnika dinamičkog programiranja, ojačavanje induktivne hipoteze, zbirovi prefiksa, tehnika podeli-pa-vladaj, ... U nastavku kursa ćemo se detaljnije fokusirati na svaku od njih i ilustrovaćemo ih sa dosta konkretnih primera. Naravno, uvešćemo još mnoge korisne tehnike koje nismo ilustrovali na ovom zadatku. Kada naučite te opšte tehnike imaćete veoma moćan arsenal oružija kojim ćete moći da napadate algoritamske probleme i da dolazite do veoma elegantnih i efikasnih rešenja. Dakle, konstrucija algoritama se da naučiti i prilično šablonizovati, ali je u okviru primene svakog šablona potrebno dosta razmišljanja!

Maksimalna suma nesusednih elemenata pozitivnog niza

Problem: Napiši program koji određuje najveći zbir podniza datog niza nenegativnih brojeva koji ne sadrži dva uzastopna člana niza. Na primer, za niz $7, 3, 2, 4, 1, 5$ najveći takav podniz je $7, 4, 5$, čiji je zbir 16.

Pokušamo da zadatak rešimo induktivno-rekurzivnom konstrukcijom. Niz elemenata možemo razložiti na poslednji element i prefiks bez njega. Maksimalni zbir

je veći od dva zbira: prvog koji se dobija tako što se poslednji element (jer je uvek nenegativan) doda se na maksimalni zbir elemenata prefiksa koji ne uključuje pretposlednji element i drugog koji se dobija kao zbir elemenata prefiksa koji uključuje pretposlednji element. Dakle, pretpostavljamo da za prefiks znamo maksimalni zbir bez njegovog poslednjeg i sa njegovim poslednjim elementom. Zato ojačavamo induktivnu hipotezu i pretpostavljamo da za svaki prefiks niza umemo da odredimo upravo te dve vrednosti. Bazu indukcije čini jednočlani prefiks niza. Maksimalni zbir sa uključenim njegovim jedinim elementom jednak je tom elementu, dok je maksimalni zbir bez njega jednak nuli. Induktivnu hipotezu proširujemo tako što prilikom dodavanja novog elementa maksimalni zbir tekućeg prefiksa sa tim novim elementom određujemo kao zbir tekućeg elementa i zbira prethodnog prefiksa bez tog elementa, dok maksimalni zbir tekućeg prefiksa bez tog novog elementa određujemo kao veći od maksimalnog zbira prethodnog prefiksa sa njegovim poslednjim i maksimalnog zbira prethodnog prefiksa bez njegovog poslednjeg elementa. Kada se petlja završi, veći od dva maksimalna zbira predstavlja traženi globalni maksimum.

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
  int n;
  cin >> n;

  int x;
  cin >> x;
  int maks_zbir_bez = 0;
  int maks_zbir_sa = x;
  for (int i = 1; i < n; i++) {
    int x;
    cin >> x;
    int novi_maks_zbir_bez =
      max(maks_zbir_sa, maks_zbir_bez);
    maks_zbir_sa = maks_zbir_bez + x;
    maks_zbir_bez = novi_maks_zbir_bez;
  }

  cout << max(maks_zbir_bez, maks_zbir_sa) << endl;

  return 0;
}
```

Složenost ovog algoritma je prilično očigledno $O(n)$.