Čas 12.1, 12.2, 12.3 - pohlepni (gramzivi) algoritmi

Algoritmi kod kojih se u svakom koraku uzima *lokalno optimalno* rešenje i koji garantuju da će takvi izbori na kraju dovesti do *globalno optimalnog* rešenja nazivaju se pohlepni ili gramzivi algoritmi (engl. greedy algorithms). Pohlepni algoritmi ne vrše ispitivanje različitih slučajeva niti iscrpnu pretragu i stoga su po pravilu veoma efikasni (pretraga je isečena na osnovu određenih meta-argumenata). Takođe, obično se veoma jednostavno implementiraju. Sa druge strane, potrebno je dokazati da se pohlepnim algoritmom dobija korektno (optimalno) rešenje, što u nekim slučajevima može biti veoma izazovno. Samo nalaženje ispravnog pohlepnog algoritma može predstavljati ozbiljan problem i često nije trivijalno odrediti da li za neki problem postoji ili ne postoji pohlepno rešenje.

Algoritmi zasnovani na pretrazi ili na dinamičkom programiranju obično u svakom koraku razmatraju više mogućnosti (kojima se dobija više potproblema) i nakon razmatranja svih mogućnosti biraju onu najbolju. Dakle, izbor se vrši tek nakon rešavanja potproblema. Za razliku od toga gramzivi algoritmi unapred znaju koja mogućnost će voditi do optimalnog rešenja i izbor vrše odmah, nakon čega rešavaju samo jedan potproblem. U slučaju optmizacionih problema i u slučaju gramzivih algoritma potrebno je da važi svojstvo *optimalne podstrukture* tj. da se optimalno rešenje polaznog problema dobija pomoću optimalnog rešenja potproblema.

Da bi se dokazala korektnost pohlepnog algoritma, obično je potrebno dokazati nekoliko stvari.

Prvo je potrebno dokazati da strategija daje rešenje koje je ispravno tj. rešenje koje zadovoljava sve uslove zadatka.

Nakon toga je potrebno dokazati i da je rešenje dobijeno strategijom optimalno. Ti dokazi su po pravilu teži i postoji nekoliko tehnika kako se oni izvode. Obično se krene od nekog rešenja za koje pretpostavljamo da je optimalno i koje ne mora biti identično onome koje smo dobili pohlepnom strategijom. Ono ne može biti gore od rešenja nađenog na osnovu pohlepne strategije (jer ona vraća jedno korektno rešenje, pa optimum može biti samo eventualno bolji od tog rešenja), a potrebno je dokazati da ne može biti bolje.

Jedna tehnika da se optimalnost dokaže je to da se pokaže da se optimalno rešenje može malo po malo, primenom transformacije pojedinačnih koraka može pretvoriti u rešenje dobijeno na osnovu naše strategije. Obično je dovoljno dokazati da se prvi korak optimalnog rešenja može zameniti prvim korakom koji gramziva strategija sugeriše, tako da se korektnost i kvalitet rešenja time

ne narušavaju i korektnost dalje sledi na osnovu induktivnog argumenta. Ovu tehniku nazivaćemo tehnikom *razmene* (engl. exchange).

Druga tehnika da se optimalnost dokaže je to da se dokaže da je rešenje dobijeno na osnovu pohlepne strategije uvek po nekom kriterijumu ispred pretpostavljenog optimalnog rešenja. Ovu tehniku nazivaćemo *pohlepno rešenje je uvek ispred* (engl. greedy stays ahead).

Treća tehnika da se optimalnost dokaže je da se odredi teorijska granica vrednosti optimuma i da se onda dokaže da pohlepni algoritam daje rešenje čija je vrednost upravo jednaka optimumu. Ovu tehniku nazivaćemo tehnikom *granice* (engl. structural bound).

Broj žabinih skokova

Problem: Kamenje je postavljeno duž pozitivnog dela x-ose i za svaki kamen je poznata njegova koordinata x. Žaba kreće da skače sa prvog kamena (koji se nalazi u koordinatnom početku) i želi da u što manje skokova dođe do poslednjeg kamena. U svakom skoku ona može da skoči samo u napred i da preskoči najviše rastojanje r (a može da skoči i manje, ako je to potrebno). Napisati program koji određuje da li žaba može stići do poslednjeg kamena i ako može u koliko najmanje skokova to može učiniti.

Možemo slobodno pretpostaviti da je niz koordinata kamenja sortiran (ako nije, uvek ga možemo na početku sortirati).

Rešenje grubom silom bi podrazumevalo da se u svakom koraku razmotre sve mogućnosti za žabin skok i rekurzivno rešavao problem gde je polazni kamen promenjen. Pošto bi u toj situaciji sigurno došlo do preklapanja rekurzivnih poziva (jer bi žaba na više načina stizala do istog kamena), poželjno bi bilo upotrebiti dinamičko programiranje. Naredni program koristi tehniku memoizacije.

```
// najmanji broj skokova od i-tog kamena
// (-1 ako nije moguće stići do kraja)
int brojSkokova(const vector<int>& kamenje, int i, int r,
                vector<int>& memo) {
  // broj kamenova
  int n = kamenje.size();

  // ako smo stigli do poslednjeg kamena, nije potrebno više skakati
  if (i == n - 1)
    return 0;

  // već smo računali najmanji broj skokova sa ovog kamena
  if (memo[i] != 0)
    return memo[i];
```

```
  // najmanji broj skokova - n glumi vrednost +beskonacno
  int min = n;
  // proveravamo sve kamenove na koje žaba može doskočiti
  for (int j = i+1; j < n && kamenje[j] - kamenje[i] <= r; j++) {
    // najmanji broj skokova od kamena j do kraja
    int broj = brojSkokova(kamenje, j, r, memo);
    // ako sa kamena j žaba može stići do kraja u manje skokova od
    // tekućeg minimuma, ažuriramo tekući minimum
    if (broj != -1 && broj + 1 < min)
      min = broj + 1;
  }
  // ako je minimum ostao +beskonačno nije moguće doskočiti do kraja
  if (min == n)
    return memo[i] = -1;
  // pamtimo i vraćamo najmanji broj skokova
  return memo[i] = min;
}

int brojSkokova(const vector<int>& kamenje, int r) {
  // skakanje kreće sa kamena broj 0
  vector<int> memo(kamenje.size(), 0);
  return brojSkokova(kamenje, 0, r, memo);
}
```

Rešenje se može kreirati i dinamičkim programiranjem naviše. Tada bi se popunjavao niz koji za svaki kamen određuje broj potrebnih skokova i popunjavanje bi teklo od poslednjeg kamena unazad.

```
int brojSkokova(const vector<int>& kamenje, int r) {
  int n = kamenje.size();
  vector<int> dp(kamenje.size());
  dp[n-1] = 0;

  for (int i = n-2; i >= 0; i--) {
    dp[i] = n;
    for (int j = i+1; j < n && kamenje[j] - kamenje[i] <= r; j++)
      if (dp[j] != n && dp[j] + 1 < dp[i])
        dp[i] = dp[j] + 1;
  }

  return dp[0];
}
```

Vremenska složenost pristupa zasnovanog na dinamičkom programiranju je $O(n^2)$.

Međutim, iako u svakom koraku žaba može imati izbor između nekoliko kamenova na koje može da skoči, intuitivno nam je jasno da ona ništa ne gubi time što skoči što dalje. Stoga deluje prilično jasno da žaba u svakom koraku treba da skoči na što dalji kamen koji je na rastojanju manjem od r. Ako takav kamen

ne postoji, tada žaba ne može stići do kraja. Nakon što žaba napravi prvi skok, postupak se ponavlja na isti način, sve dok ne stigne do kraja ili dok se ne dođe do situacije u kojoj žaba ne može da skoči dalje (ovo, naravno, ukazuje na induktivno-rekurzivnu prirodu algoritma).

U svakom koraku tražimo najdalji kamen na koji žaba može da skoči i to tako što tražimo prvi kamen na koji žaba ne može da doskoči. Kada takav kamen nađemo (ili kada ustanovimo da žaba može da skoči na svaki kamen ispred sebe, jer smo došli do kraja niza), žaba skače na kamen ispred njega i postupak se nastavlja sve dok žaba ne skoči na poslednji kamen ili dok se ne dogodi da žaba ne može da skoči već na prvi kamen ispred sebe.

```
int brojSkokova(const vector<int>& kamenje, int r) {
   // broj kamenova
   int n = kamenje.size();
   // broj napravljenih skokova
   int broj = 0;
   // redni broj kamena na kome se nalazi žaba
   int kamen = 0;
   // dok žaba ne doskoči na poslednji kamen
   while (kamen < n - 1) {
      // određujemo redni broj kamena na koji žaba skače
      int noviKamen = kamen;
      while (noviKamen + 1 < n &&
             kamenje[noviKamen + 1] - kamenje[kamen] <= r)
         noviKamen++;
      // žaba ne može da skoči ni na jedan kamen
      if (noviKamen == kamen)
         return n;
      // žaba pravi skok
      broj++;
      kamen = noviKamen;
   }
   // vraćamo izračunati broj skokova
   return broj;
}
```

Iako postoje dve ugnežđene petlje, složenost prethodnog algoritma je $O(n)$ (pokazivači `kamen` i `noviKamen` se nikada ne umanjuju).

Ovaj algoritam je tipičan pohlepni algoritam jer se u svakom koraku uzima što je više moguće i tako da se dolazi i do globalnog optimuma. Ostaje pitanje kako dokazati da je ova strategija korektna.

Prvo dokazujemo da prethodni algoritam uvek daje korektno rešenje tj. rešenje u skladu sa uslovom zadatka (žaba kreće sa prvog kamena, dolazi na poslednji i u svakom koraku skače samo napred, ne više od r metara). Zaista, implementacija je određena tako da je svaki skok koji žaba pravi u prethodnom algoritmu skok na kamen koji je udaljen najviše r (to se u algoritmu eksplicitno proverava), pa smo sigurni da je niz skokova, ako se pronađe, ispravan. Dodatno je potrebno da

dokažemo da je i u slučaju kada funkcija vraća -1, taj odgovor korektan tj. da tada zaista nije moguće da žaba dođe do kraja. To se dešava kada postoje dva kamena sa koordinatama x_i i x_{i+1} takvi da je $x_{i+1} - x_i > r$. Ako bi postojalo neko ispravno rešenje, žaba bi morala da skoči sa nekog kamena pre i (ili sa kamena i) na neki kamen nakon x_{i+1} (ili na kamen x_{i+1}), no to je nemoguće jer je zbog sortiranosti niza rastojanje svakog takvog para kamenova strogo veće od r.

Drugo što je potrebno dokazati je da je rešenje koje se dobija strategijom optimalno. U ovom slučaju to znači da pohlepna strategija daje najmanji mogući broj skokova.

U ovom konkretnom slučaju ćemo dokazati da se kamen na kom se žaba nalazi nakon i koraka primene strategije nikada ne nalazi strogo iza kamena na kom se žaba nalazi nakon i koraka u optimalnom rešenju (žaba skače po strategiji je u svakom koraku ili ispred ili na istom kamenu u odnosu na sve žabe koje dostižu cilj u najmanjem broju koraka). Ovo je tipičan dokaz tehnikom *pohlepno rešenje je uvek ispred*. Formalno, neka je optimalna vrednost broja skokova k i neka je $x_0^*, x_1^*, \ldots, x_{k-1}^*$ sortiran niz x-koordinata kamenja na koje žaba staje u jednom takvom optimalnom rešenju. Neka je $x_0, x_1, \ldots, x_m$ rešenje dobijeno na osnovu naše strategije. Indukcijom dokazujemo da je $x_i \geq x_i^*$. Bazu čini slučaj $i = 0$ i tada je $x_0 = x_0^*$, jer se žaba u oba slučaja nalazi na početnom kamenu. Pod pretpostavkom da važi $x_i \geq x_i^*$ dokazujemo da važi $x_{i+1} \geq x_{i+1}^*$. Ako je $x_i \geq x_{i+1}^*$, pošto žaba skače samo napred, važi da je $x_{i+1} > x_i \geq x_{i+1}^*$. U suprotnom je $x_i < x_{i+1}^*$. Pošto je optimalno rešenje korektno, znamo da je $x_{i+1}^* - x_i^* \leq r$. Pošto na osnovu induktivne hipoteze znamo da važi $x_i \geq x_i^*$ važi i $x_{i+1}^* - x^i \leq r$. Dakle, žaba sa kamena x_i može sigurno da doskoči na kamen x_{i+1}^* (on se nalazi ispred, na rastojanju manjem od r), a možda može i dalje. Pošto gramziva strategija uzima uvek najdalji skok važi da je $x_{i+1} \geq x_{i+1}^*$. Na osnovu dokazanog važi da je $x_{k-1} \geq x_{k-1}^*$, međutim, pošto je x_{k-1}^* koordinata poslednjeg kamena, to mora biti i x_{k-1}. Dakle, i optimalna strategija stiže do poslednjeg kamena u k koraka, pa optimalno rešenje nije bolje od pohlepnog.

Provera podniza

Problem: Date su dve reči zapisane malim slovima. Napisati program kojim se proverava da li slova druge reči čine podniz slova prve reči, tj. da li se druga reč može dobiti izbacivanjem nekih slova (ne obavezno susednih) iz prve reči. Dokazati korektnost algoritma.

Pri precrtavanju slova, njihov redosled se ne menja što znači da i u drugoj reči redosled slova mora biti isti kao u prvoj. Svako slovo druge reči mora da se javi u prvoj. Ako postoji rešenje problema, onda se rešenje može dobiti i tako što se u prvoj reči zadrži prvo pojavljivanje prvog slova druge reči (sva slova ispred njega se precrtaju) i ostala slova druge reči potraže iza tog prvog pojavljivanja. Naime, ako bi postojalo rešenje u kojem je prvo pojavljivanje

prvog slova druge reči u prvoj reči precrtano, a zadržano je neko njegovo kasnije pojavljivanje, pošto se sva ostala zadržana slova druge reči u prvoj reči nalaze iza tog kasnijeg pojavljivanja prvog slova, mogli bismo precrtati to zadržano kasnije pojavljivanje, a zadržati prvo pojavljivanje i tako dobiti ispravno precrtavanje u kojem je zadržano baš prvo pojavljivanje. Sličan je slučaj i sa daljim slovima (uvek u prvoj reči biramo prvo neprecrtano pojavljivanje tekućeg slova druge reči, jer ako reč možemo dobiti precrtavanjem slova, možemo je dobiti i korišćenjem te strategije). Stoga se rešenje zasniva na jednostavnom pohlepnom algoritmu. Primetimo da smo prethodnim razmatranjem zapravo dokazali korektnost ove strategije, tako što smo pokazali da se bilo koje drugo ispravno rešenje može korak po korak transformisati u ono dobijeno pohlepnom strategijom (upotrebili smo dokaz zasnovan na tehnici *razmene*).

Na osnovu ovoga, dovoljno prolaziti u petlji kroz prvu reč, karakter po karakter, i proveravati da li je tekući karakter jednak karakteru koji je na redu u drugoj reči (na početku, na redu je prvi karakter sa indeksom 0). Ako su odgovarajući karakteri jednaki, prelazi se na naredni karakter u obe reči, a inače samo u prvoj. Petlju treba prekinuti kada prođemo kroz sve karaktere bar jedne reči. Ako smo prošli kroz sve karaktere druge reči, zaključujemo da se druga reč može dobiti precrtavanjem slova prve, inače ne.

```
bool jePodniz(const string& s1, const string& s2) {
  // redom prolazimo kroz slova obe reči dok ne dodjemo do kraja jedne
  // od njih
  int i = 0, j = 0;
  while (i < s1.size() && j < s2.size()) {
    // ako je tekuće slovo prve reci jednako tekućem slovu druge reči
    // onda ga zadržavamo, a u suprotnom ga precrtavamo
    if (s1[i] == s2[j])
      // ako smo slovo zadržali, prelazimo na naredno slovo druge reči
      j++;
    // prelazimo na naredno slovo prve reči
    i++;
  }

  // reč se moze dobiti ako i samo ako smo stigli do kraja druge reči
  // (sva slova druge reči smo pronasli u prvoj)
  return j == s2.size();
}
```

Drugi pogled na isti ovaj postupak je da se za svaki karakter druge reči proveri da li postoji u prvoj reči i ako postoji, pronađe njegovo prvo pojavljivanje, pri čemu se pretraga za prvim karakterom vrši od početka prve reči, a za svakim narednim od pozicije iza one na kojoj je prethodni karakter nađen. Ako se neki karakter ne može pronaći, tada drugu reč nije moguće dobiti od prve. Ako se svi karakteri druge reči uspešno pronađu, onda je drugu reč moguće dobiti od prve. Pretragu karaktera možemo vršiti ili algoritmom linearne pretrage ili bibliotečkom metodom `find` koja vraća specijalnu vrednost `string::npos` ako

karakter nije nađen.

```
bool jePodniz(const string& s1, const string& s2) {
  // tražimo jedno po jedno slovo druge reči u prvoj, krenuvši od pozicije i
  int i, j;
  for (i = 0, j = 0; j < s2.size(); i++, j++) {
    // trazimo tekuce slovo druge reči u prvoj, krenuvši od pozicije i
    i = s1.find(s2[j], i);
    // ako ga nismo našli, tada prekidamo pretragu
    if (i == string::npos)
      break;
    // prelazimo na sledeću poziciju u obe reči
  }
  return j == s2.size();
}
```

Plesni parovi

Problem: Poznate su visine n momaka i n devojaka. Napisati program koji određuje koliko se najviše plesnih parova može formirati tako da je momak uvek viši od devojke i dokazati njegovu korektnost.

I ovaj problem je moguće rešiti pohlepnom strategijom. Jedna mogućnost je da se parovi formiraju tako što se upari k najviših momaka sa k najnižih devojaka. Ta strategija bi bila korektna, ali njena implementacija nije trivijalna, jer nije jasno koliko maksimalno može da bude k. Varijacija koju ćemo jednostavno implementirati je sledeća. Ako postoji bar jedan plesni par, u njemu može da učestvuje najviši mladić. Naime, ako on ne bi učestvovao ni u jednom paru, uvek bismo nekog od mladića mogli zameniti njime (jer je on viši od svih mladića) i dobiti isti broj plesnih parova (primetimo da u ovom razmatranju koristimo tehiku *razmene*). Postavlja se pitanje sa kojom devojkom on treba da pleše. Cilj nam je da nakon formiranja tog para preostanu što niže devojke, da bi niži mladići imali šanse da se sa njima upare. Jasno je da moramo da eliminišemo sve devojke koje su više od tog najvišeg mladića (jer sa njima niko ne može da pleše), a od preostalih devojaka možemo da izaberemo najvišu. Nakon eliminisanja tog mladića, svih devojaka viših od njega i devojke sa kojom pleše, problem je sveden na problem istog oblika, ali manje dimenzije. Izlaz predstavlja slučaj kada su sve devojke više od najvišeg među preostalim mladićima.

Prilikom implementacije skup momaka i devojaka možemo čuvati u nizovima uređenim u opadajućem redosledu visine. Niz momaka obilazimo redom, element po element, a u niz devojaka razdvajamo na one koje su eliminisane (one koje su do sada uparene i one koje nisu uparene, ali su više od tekućeg momka). Održavamo mesto početka niza devojaka koje još nisu obrađene i prilikom traženja devojke za tekućeg momka niz devojaka obilazimo od te pozicije. Svaku devojku ili eliminišemo, jer je viša od tekućeg momka ili je dodeljujemo tekućem momku i onda ih oboje eliminišemo. Naglasimo da se u implementaciji ne

moramo vraćati na eliminisane devojke, jer ako je neka devojka viša od tekućeg momka, biće viša i od svih narednih. Stoga se oba pokazivača kreću samo u jednom smeru i složenost faze dodeljivanja je linearna. Ukupnim algoritmom, dakle, dominira složenost sortiranja, pa je ukupna složenost $O(n \log n)$.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

int main() {
   // ucitavamo ulazne podatke
   int n;
   cin >> n;
   vector<int> momci(n);
   for (int i = 0; i < n; i++)
     cin >> moci[i];
   vector<int> devojke(n);
   for (int i = 0; i < n; i++)
     cin >> devojke[i];

   // želimo da i devojke i mladiće obilazimo u opadajućem redosledu visine
   sort(begin(momci), end(momci), greater<int>());
   sort(begin(devojke), end(devojke), greater<int>());
   int brojParova = 0;
   // tekući indeks momka i devojke
   int m = 0, d = 0;
   while (true) {
       // trazimo najvišu devojku sa kojom može da pleše momak na poziciji m
       while (d < n && momci[m] < devojke[d])
          d++;
       // ako takva devojka ne postoji, ne možemo povećati broj parova
       if (d >= n) break;
       // našli smo par
       brojParova++;
       // prelazimo na narednog momka i devojku
       m++, d++;
   }
   // ispisujemo konačan rezultat
   cout << brojParova << endl;

   return 0;
}
```

Dokažimo i formalno korektnost (koristimo tehniku razmene). Rešenje koje prethodni algoritam daje zadovoljava uslove zadatka jer se svaki momak uparuje sa devojkom koja nije viša od njega (to se eksplicitno proverava) i nakon uparivanja se eliminiše iz razmatranja, tako da smo sigurni da zaista postoji

korektno uparivanje za broj parova koji se vraća. Pokažimo i da naša strategija pravi optimalni broj parova. Dokaz će ići tehnikom razmene, tj. time što ćemo se pokazati da se optimalno uparivanje može transformisati u ono dobijeno gramzivom strategijom, održavajući ukupan broj formiranih parova. Posmatrajmo neko optimalno uparivanje. Neka je m_i najviši momak koji učestvuje u uparivanju. Ako on nije ukupno najviši momak m_s, tada najviši momak sigurno nije uparen. Možemo momka m_i izbaciti iz uparivanja i njemu pridruženu devojku d_i pridružiti ukupno najvišem momku m_s (to je moguće jer je $m_s \geq m_i \geq d_i$). Takvo uparivanje je i dalje optimalno (jer se broj parova nije promenio). Neka je d_s devojka koja bi bila odabrana strategijom (najviša devojka koja nije viša od d_s, tj. najviša devojka za koju važi $m_s \geq d_s$). Ako ona nije angažovana u trenutnom uparivanju, onda devojku d_i koja trenutno pleše sa m_s možemo izbaciti i zameniti njome (to je moguće jer je $m_s \geq d_s$). Ako jeste raspoređena da pleše sa nekim m_j, onda možemo napraviti razmenu tako da m_s pleše sa d_s, a m_j sa d_i. Dokažimo da je ovo i dalje korektno uparivanje. Važi da je $m_s \geq d_s$ i $m_s \geq d_i$. Pošto je d_s najviša devojka koja može da igra sa m_s, važi da je $d_s \geq d_i$. Zato je $m_j \geq d_s \geq d_i$. Sa ove dve eventualne razmene dobijamo i dalje optimalan raspored koji je u skladu sa našom strategijom što se tiče prvog plesnog para. Nastavljajući razmene po istom principu (tj. na osnovu induktivnog argumenta), uparivanje možemo transformisati u ono formirano našom strategijom, zadržavajući sve vreme optimalnost.

Dualno rešenje bi bilo ono u kome obrađujemo devojke u rastućem redosledu visine i svakoj devojci dodeljujemo što nižeg momka koji može da pleše sa njom.

Mentori

Problem: Šahisti jednog tima se spremaju za dolazeće turnire. Odlučeno je da šahisti sa boljim rejtingom pomažu onim sa lošijim rejtingom, tako što će im biti mentori. Da bi jedan šahista mogao da bude mentor drugome, njegov rejting treba da bude bar dva puta veći nego rejting prvoga. Ako su poznati rejtinzi svih šahista, napisati program koji određuje koji je najveći broj parova učenik-mentor koji se može formirati, pri čemu šahista može istovremeno biti i učenik i mentor (mentor onome ko ima bar dva puta manji rejting od njega, a učenik onome koji ima bar dva puta veći rejting od njega), ali ni jedan šahista ne može da ima dva učenika niti dva mentora. Dokazati korektnost.

Primetimo sličnost ovog zadatka sa prethodnim (ponovo je u pitanju određivanje maksimalnog uparivanja dva uređena niza, jedino što je sada uslov da se a može upariti sa b umesto $a \geq b$, uslov $a \geq 2b$).

Jedan način da dođemo do rešenja je da primetimo da ako možemo odrediti mentore za nekih k takmičara, onda ti isti mentori mogu biti dodeljeni i najslabijim takmičarima. Dakle, rešenju možemo pristupiti tako što pokušavamo redom da dodelimo mentore najslabijim takmičarima, što podrazumeva da niz rejtinga najpre sortiramo i da niz takmičara obilazimo u rastućem redosledu. Svakom

narednom takmičaru pokušavamo da dodelimo mentora i to radimo tako da od svih potencijalnih mentora (takmičara koji imaju bar dvostruko veći rejting od njega koji nisu već ranije dodeljeni nekome kao mentori) pronalazimo onog sa najmanjim rejtingom. Takvim izborom ostavljamo mogućnost da potencijalni mentori sa još većim rejtingom kasnije postanu mentori jačim takmičarima.

Prilikom implementacije potrebno je održavati skup dodeljenih mentora (najjednostavnije tako što održavamo asocijativni niz logičkih vrednosti tako da je na mestu k vrednost tačno ako i samo ako je takmičar sa rednim brojem k već dodeljen nekome kao mentor, čime u konstantnom vremenu za svakog takmičara možemo proveriti da li je slobodan da postane mentor). Jedna važna opaska za efikasnu implementaciju je to da ako je učeniku u dodeljen mentor m, tada mentori učenika $u+1$ mogu biti isključivo takmičari sa rednim brojem $m+1$ pa naviše (naime, ako bi neki takmičar zaključno sa rednim brojem m mogao biti mentor učeniku $u+1$, onda bi on sigurno mogao biti mentor i učeniku u, a pošto svakom učeniku dodeljujemo najslabije mentore, on bi morao biti dodeljen učeniku u, što je kontradikcija sa time da je na raspolaganju da postane mentor učeniku $u+1$). Na ovaj način se promenljive koje vrše iteraciju kroz učenike i kroz mentore samo uvećavaju (nema potrebe ni u jednom trenutku ih umanjivati), pa je složenost faze pridruživanja linearna i algoritmom dominira složenost sortiranja koja je $O(n \log n)$. Varijanta u kojoj se ova optimizacija ne koristi (koja za svakog učenika u proverava skup potencijalnih mentora iz početka, krenuvši od učenika $u+1$) dovodi do kvadratne složenosti faze pridruživanja, što kvari složenost celog zadatka na $O(n^2)$.

Dualno, možemo takmičare obilaziti u opadajućem redosledu rejtinga i svakom od njih pokušavati da dodelimo ulogu mentora i to baš najjačem takmičaru među onima koji su bar dvostruko slabiji od njega i koji do tada nisu proglašeni za nečijeg učenika (argumenti su slični: ako možemo da napravimo k parova učenik-mentor, onda možemo da promenimo mentore u takvoj dodeli i da ih zamenimo za k najjačih takmičara, a kada je neki takmičar fiksiran da postane mentor, najbolje je dodeliti ga najjačem takmičaru, da bi slabijim kandidatima bila ostavljena veća mogućnost da nekom postanu mentori).

Još jedna mogućnost, koja je ispravna, ali manje intuitivna je da takmčare obrađujemo u rastućem redosledu rejtinga i da za svakog od njih proveravamo da li je uopšte moguće da postane mentor, tako što proveravamo sve takmičare čiji je rejting bar duplo manji od rejtinga tekućeg kandidata i među onima koji još nisu nikome dodeljeni za učenike tražimo onog koji ima najmanji rejting (opet, da bi kasnijim kandidatima bila ostavljena veća šansa da postanu nekome mentori).

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;
```

```cpp
int main()  {
  // učitavamo rejtinge učenika
  int n;
  cin >> n;
  vector<int> rejting(n);
  for (int i = 0; i < n; i++)
    cin >> rejting[i];

  // sortiramo učenike po rastućem redosledu rejtinga
  sort(begin(rejting), end(rejting));

  // broj do sada formiranih parova učenik-mentor
  int brojParova = 0;
  // za svakog učenika podatak o tome da li već dodeljen nekome kao mentor
  vector<bool> jeMentor(n, false);
  // trenuti potencijalni mentor
  int mentor = 1;
  // dodeljujemo mentore najslabijim učenicima, sve dok postoji bar
  // jedan potencijalnih mentor
  for (int ucenik = 0; mentor < n; ucenik++)
    // dokle god postoji potencijalni mentor
    while (mentor < n)
      if (!jeMentor[mentor] && rejting[mentor] >= 2 * rejting[ucenik]) {
    // pronašli smo potencijalnog mentora za tekućeg učenika
    brojParova++;
    jeMentor[mentor] = true;
    break;
      } else
    // prelazimo na sledećeg kandidata za mentora
    mentor++;

  // ispisujemo ukupan broj parova
  cout << brojParova << endl;

  return 0;
}
```

Razlomljeni problem ranca

Problem: U jednoj prodavnici se prodaju slatkiši (bombonice, čokoladice, keksići) "na meru". Postoji n vrsta slatkiša i znamo da i-tog slatkiša ima w_i grama, po ukupnoj ceni od v_i dinara. Prodavnica je u okviru svoje promocije organizovala nagradnu igru u kojoj je nagradila jednu svoju mušteriju tako da na poklon može da uzme sve slatkiše koji staju u ranac nosivosti W grama. Kolika je najveća vrednost slatkiša koje sretni dobitnik može da uzme. Kako to da postigne? Dokazati korektnost.

Pošto sretni dobitnik želi da pokupi što veću vrednost slatkiša, jasno je da treba

da krene uzimanje onih slatkiša koji su najvredniji tj. čija je cena po gramu najveća. Ako tom vrstom slatkiša može da ispuni ceo ranac, najbolje mu je da to da uradi (na ovom mestu pretpostavljamo da je moguće da ne pokupi celokupnu raspoloživu količinu tog slatikša). U suprotnom, pokupiće celokupnu količinu tog slatkiša, a zatim će preostalu nosivost ranca popuniti preostalim slatkišima, po istom principu (ovde se može uočiti induktivno-rekurzivna konstrukcija).

```
double razlomljeniRanac(const vector<int>& cena,
                        const vector<int>& kolicina,
                        int nosivostRanca) {
  // broj vrsta slatkiša
  int n = cena.size();
  // vektor u kome čuvamo jedinične cene i količine svih slatkiša
  vector<pair<double, int>> jedinicnaCenaIKolicina(n);
  for (int i = 0; i < n; i++) {
    // jedinična cena slatkiša slatkiša broj i
    double jedinicnaCena = (double)cena[i] / (double)kolicina[i];
    jedinicnaCenaIKolicina[i] =
      make_pair(jedinicnaCena, kolicina[i]);
  }
  // sortiramo opadajuće na osnovu jedinične cene
  sort(begin(jedinicnaCenaIKolicina), end(jedinicnaCenaIKolicina),
       greater<pair<double, int>>());
  // ukupna vrednost slatkiša koja se može poneti u rancu
  double ukupnaVrednost = 0.0;
  // obrađujemo slatkiše po opadajućoj vrednosti jedinične cene
  for (int i = 0; nosivostRanca > 0 && i < n; i++) {
    // čitamo jediničnu cenu i količinu slatkiša broj i
    double jedinicnaCena = jedinicnaCenaIKolicina[i].first;
    int kolicina = jedinicnaCenaIKolicina[i].second;
    // uzimamo što više, ali smo ograničeni raspoloživom količinom
    // i preostalom nosivišću ranca
    int uzetaKolicina = min(kolicina, nosivostRanca);
    // preostala nosivost ranca
    nosivostRanca -= uzetaKolicina;
    // ažuriramo ukupnu vrednost
    ukupnaVrednost += uzetaKolicina * jedinicnaCena;
  }
  // vraćamo ukupnu vrednost uzetih slatkiša
  return ukupnaVrednost;
}
```

Jasno je da prethodni gramzivi algoritam daje uvek korektno rešenje, jer se ni za jedan predmet ne uzima veća količina od dostupne i ukupna masa uzetih slatkiša ne prevazilazi masu ranca.

Dužni smo još da dokažemo da naša gramziva strategija dovodi do optimalnog rešenja. Koristićemo metod razmene. Pretpostavimo da smo sortirali slatkiše tako da važi

$$\frac{v_0}{w_0} \geq \frac{v_1}{w_1} \geq \ldots \geq \frac{v_{n-1}}{w_{n-1}}.$$

Svako rešenje je određeno uzetom masom (u gramima) svakog od slatkiša. Neka je rešenje na osnovu strategije određeno nizom masa $(s_0, s_1, \ldots, s_{n-1})$, pri čemu za svako $0 \leq i < n$ važi $0 \leq s_i \leq w_i$ i $\sum_{i=0}^{n-1} s_i \leq W$, gde je W ukupna nosivost ranca. Pretpostavimo da je optimalno rešenje određeno nizom masa $(o_0, o_1, \ldots, o_{n-1})$, pri čemu za svako $0 \leq i < n$ važi $o_i \leq w_i$ i $\sum_{i=0}^{n-1} o_i \leq W$. Pošto je o optimalno rešenje, mora da važi da je $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$. Naime, znamo da će kada se slatkiši uzimaju na osnovu strategije ukupna masa uzetih slatkiša bili ili jednaka nosivosti ranca ili ukupnoj raspoloživoj masi svih slatkiša (kada je ona veća ili jednaka od nosivosti ranca). Ako bi se u optimalnom rešenju sadržala manja masa slatkiša od toga, mogli bismo masu nekog slatkiša da povećamo i tako da dobijemo još bolje rešenje, što je u kontradikciji sa pretpostavkom optimalnosti.

Pretpostavimo da strategija daje rešenje s koje je različito od optimalnog rešenja o. Neka je j prva pozicija na kojoj se nizovi s i o razlikuju (za svako $0 \leq i < j$ važi da je $s_i = o_i$). Gramziva strategija uzima redom celokupne raspoložive mase svih predmeta, sve do poslednjeg uzetog predmeta gde se uzima maksimalna masa koja staje u ranac, tako da je jedina mogućnost da je $o_j < s_j$ i to za $j < n - 1$ (zbog uslova $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$). Razmotrimo rešenje o' koje dobijamo tako što umesto mase o_j predmeta j uzmemo masu s_j (to sigurno možemo, jer je $s_j \leq w_j$). Time smo povećali ukupnu masu u rancu za $s_j - o_j$ i zato ukupnu masu uzetih predmeta nakon pozicije j moramo da smanjimo za $s_j - o_j$ (to možemo jer je $\sum_{i=0}^{n-1} o_i = \sum_{i=0}^{n-1} s_i$ i $\sum_{i=0}^{j-1} o_i = \sum_{i=0}^{j-1} s_i$). Dokažimo da ovo rešenje ne može biti lošije od optimalnog. Njegova vrednost se povećala za $(s_j - o_j) \cdot \frac{v_j}{w_j}$ i umanjila za $\sum_{k=j+1}^{n-1} c_k \frac{v_k}{w_k}$, gde je $\sum_{k=j+1}^{n-1} c_k = s_j - o_j$. Dokažimo da je povećanje veće nego smanjenje. Važi da je

$$\sum_{k=j+1}^{n-1} c_k \frac{v_k}{w_k} \leq \sum_{k=j+1}^{n-1} c_k \frac{v_{j+1}}{w_{j+1}} = \frac{v_{j+1}}{w_{j+1}} \sum_{k=j+1}^{n-1} c_k = \frac{v_{j+1}}{w_{j+1}}(s_j - o_j) \leq \frac{v_j}{w_j}(s_j - o_j).$$

Ovom promenom smo, dakle, napravili rešenje o' čija je vrednost jednaka optimalnoj (jer nijedno rešenje ne može biti bolje od optimalnog rešenja o, a upravo smo dokazali da rešenje o' nije lošije od njega), a koje se poklapa sa strategijskim rešenjem s na jednoj poziciji više nego polazno optimalno rešenje o. Nastavljajući postupak na isti način dobićemo rešenje koje je optimalno i jednako je s.

Vraćanje kusura

Problem: U Srbiji se koriste apoeni od 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000 i 5000 dinara. Napiši program koji formira datih iznos dinara od što manjeg

broja novčanica i novčića. Šta bi se desilo da postoji i novčić od 4 dinara?

Jedan direktan način da se problem reši je da se ispitaju sva moguća razlaganja datog iznosa na zbirove sastavljene od ovih brojeva i da se pronađe onaj zbir koji ima najmanji broj novčića (jednostavnosti radi, zanemarimo razliku između novčića i novčanica). Rešenje ovog tipa bi se moglo zasnovati na dinamičkom programiranju kombinovanom sa odsecanjem tokom pretrage. Takvo rešenje je opšte i već smo ga implementirali u poglavlju o dinamičkom programiranju, kada ništa nismo znali o konkretnim apoenima. Dokaz korektnosti ne bi bio težak, ali bi ovaj pristup bio prilično komplikovan i neefikasan. Naime, i bez formalnog matematičkog objašnjenja, svaki prodavac u prodavnici i na pijaci zna da se optimalno rešenje dobija tako što se u svakom trenutku vraća najveći apoen koji je manji ili jednak od trenutnog iznosa i nakon toga se isti princip primenjuje na preostali iznos sve dok se ne vrati ceo kusur (u pitanju je, dakle, induktivno-rekurzivna konstrukcija). Ovo rešenje je veoma efikasno, lako se implementira, međutim, dokaz njegove korektnosti nije nimalo očigledan. Naime, postojanje novčića od 4 dinara bi pokvarilo situaciju. 8 dinara bi se moglo dobiti od dva novčića od 4 dinara, dok bi gramziva strategija upotrebila tri novčića (od 5, 2 i 1 dinar). Dakle, dokaz korektnosti mora da uključi analizu konkretnih apoena koji su u opticaju i male promene ovih apoena mogu da utiču na to da opisani pristup daje ili ne daje uvek optimalno rešenje.

Dokažimo sada korektnost. Jednostavnosti radi, pretpostavićemo da su u opticaju samo apoeni od 1, 2, 5, 10, 20 i 50 dinara (za veće novčiće dokaz ide po istom principu). Metodom razmene dokazaćemo gornje granice broja novčića od svih apoena u optimalnom rešenju.

- Optimalno rešenje ne može da sadrži više od jednog novčića od 1 dinar. Kada bi postojala makar dva novčića od 1 dinar, oni bi mogli biti zamenjeni jednim novčićem od 2 dinara, čime bi se broj upotrebljenih novčića smanjio, što je u kontradikciji sa pretpostavkom da je polazno rešenje optimalno. Potpuno analogno se dokazuje da optimalno rešenje ne može sadržati ni više od jednog novčića od 10 dinara.
- Dalje, optimalno rešenje ne može sadržati više od dva novčića od 2 dinara. Naime, ako bi sadržalo bar tri novčića od 2 dinara, oni bi mogli biti zamenjeni jednim novčićem od 1 i jednim novčićem od 5 dinara, čime bi se dobilo manje rešenje, što je u kontradikciji sa pretpostavkom da je polazno rešenje optimalno. Potpuno analogno, optimalno rešenje ne može da sadrži ni više od dva novčića od 20 dinara.
- Na kraju, u optimalnom rešenju ne može biti više od jednog novčića od 5 dinara. Naime, ako bi postojala bar dva, ona bi mogla biti zamenjena jednim novčićem od 10 dinara čime bi se dobilo manje rešenje, što je kontradikcija.
- U rešenju nije moguće ni da istovremeno postoje dva novčića od 2 dinara i novčić od 1 dinara, jer bi se svi oni mogli zameniti sa jednim novčićem od 5 dinara, što je opet kontradikcija. Analogno važi i za novčiće od 10 i 20 dinara.

Uzevši u obzir prethodna ograničenja, razmotrimo maksimalne iznose sa optimalnim brojem novčića, koji se mogu dobiti korišćenjem samo određenih skupova novčića. Ispostaviće se da su maksimalni iznosi uvek za jedan manji od prvog većeg apoena.

- Novčići od 1 dinar mogu da naprave najviše iznos od 1 dinara (jer se smeju pojaviti samo jednom).
- Novčići od 1 i 2 dinara mogu da naprave najviše iznos od 4 dinara (jer može biti najviše dva novčića od 2 dinara i u tom slučaju se ne sme koristiti i novčić od 1 dinar).
- Novčići od 1, 2 i 5 dinara mogu da naprave najviše iznos od 9 dinara (jer ne može biti više od jednog novčića od 5 dinara, dva novčića od 2 i jednog novčića od 1 dinara, a ako ima dva novčića od 2 dinara, ne sme se javiti i novčić od 1 dinara).
- Slično, novčići od 1, 2, 5 i 10 dinara mogu da naprave najviše iznos od 19 dinara (jer se 10 dinara može javiti samo jednom, a od 1, 2 i 5 se može napraviti najviše 9).
- Novčići od 1, 2, 5, 10 i 20 mogu da naprave najviše iznos od 49 dinara (jer 1, 2 i 5 mogu da naprave najviše 9, kako smo objasnili, a pošto se 10 dinara javlja najviše jednom, a 20 dinara najviše dva puta, ali ne sva tri takva zajedno, od 10 i 20 se može napraviti najviše 40).

Dokažimo sada da se za svaki pozitivan iznos u optimalnom rešenju mora nalaziti najveći novčić koji je manji ili jednak od tog iznosa (sve navedene konstatacije se odnose samo na optimalna rešenja).

- Za iznos 1 javlja se samo novčić 1.
- Iznosi između 2 i 4 dinara moraju da sadrže novčić 2. Naime, ne može da se javi novčić od 5 dinara, samo od novčića od 1 dinar može da se napravi najviše 1 dinara, pa za iznose od 2 do 4 dinara mora da se javi bar jedan novčić od 2 dinara.
- Iznosi između 5 i 9 dinara moraju da sadrže novčić od 5 dinara. Naime, ne mogu da sadrže novčić od 10 dinara, a pošto se od novčića od samo 1 i 2 dinara može napraviti najviše 4 dinara, za iznose od 5 do 9 dinara mora da se upotrebni novčić od 5 dinara.
- Iznosi između 10 i 19 dinara moraju da sadrže novčić 10. Naime, 20 dinara ne može da se javi, a pomoću novčića od 1, 2 i 5 dinara najviše se može napraviti 9 dinara.
- Iznosi između 20 i 49 dinara moraju da sadrže bar jedan novčić od 20 dinara. Naime, ne mogu da sadrže 50, a samo sa 1, 2, 5 i 10 se može dobiti najviše 19.
- Iznosi preko 50 dinara moraju da sadrže novčić od 50 dinara. Naime, samo sa novčićima od 1, 2, 5, 10 i 20 je moguće napraviti samo 49 dinara.

Dakle, uspeli smo da za svaki iznos pronađemo novčić koji optimalno rešenje mora da sadrži, čime onda uspevamo da smanjimo dimenziju problema i da do rešenja dođemo direktno, bez bilo kakve pretrage i isprobavanja raznih mogućnosti.

```
void stampajKusur(int iznos) {
  vector<int> apoeni {500, 200, 100, 50, 20, 10, 5, 2, 1};
  while (iznos > 0) {
    for (int apoen : apoeni)
      if (iznos >= apoen) {
        cout << apoen << endl;
        iznos -= apoen;
        break;
      }
  }
}
```

Prethodni algoritam je takav da se u svako pojedinačnom koraku uzima tačno određeni element rešenja i nema potrebe proveravati različite kombinacije da bi se pronašlo optimalno rešenje, što je, naravno, odlika pohlepnih algoritama.

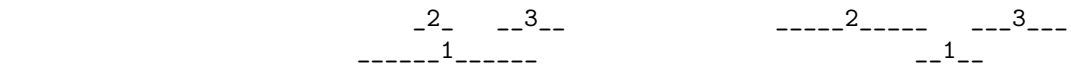
Raspored aktivnosti

Problem: U jednom kabinetu se subotom održava obuka programiranja. Svaki nastavnik drži jedno predavanje i napisao je vreme u kom želi da drži nastavu (poznat je sat i minut početka i sat i minut završetka predavanja). Odredi kako je moguće napraviti raspored časova tako da što više nastavnika bude uključeno. Napisati program koji određuje optimalan raspored i dokazati njegovu korektnost.

Dakle, pretpostavljamo da nam je dat niz od n intervala oblika $[s_i, f_i)$. Dva intervala $[s_i, f_i)$ i $[s_j, f_j)$ su kompatibilna ako im je presek prazan tj. ako je ili $f_i \leq s_j$ ili je $f_j \leq s_i$. Potrebno je pronaći kardinalnost maksimalnog podskupa međusobno kompatibilnih intervala.

Jedan način da se problem reši je da se ispitaju svi mogući podskupovi skupa časova, odaberu oni u kojima se časovi ne preklapaju i među njima pronađu oni koji sadrže maksimalni broj nastavnika. Složenost ovog pristupa bila bi ekspoencijalna i jasno je da on ne bi mogao da se u praksi primeni na probleme koji imaju više od nekoliko desetina predavanja.

Razmislimo koji čas ima smisla prvi rasporediti. Iako neko može pomisliti da to može da bude čas koji prvi počinje ili čas koji najkraće traje, ni jedan, ni drugi pristup ne daju optimalno rešenje (kako je ilustrovano sa dva kontra-primera na slici).



U oba primera rasporedio bi se samo čas broj 1, a bolje rešenje je da se umesto njega rasporede časovi 2 i 3.

Pokazaće se da će se optimalno rešenje dobiti na osnovu intuicije koja nam govori da je dobro prvo zakazati onaj čas nakon čijeg održavanja učionica ostaje što duže slobodna, tj. od svih časova prvo zakazati onaj čas koji se najranije od

svih časova završava. Nakon toga, potrebno je iz skupa izbaciti taj čas i sve one časove koji se sa njim preklapaju i nastaviti rekurzivno rešavanje problema sve dok se skup potencijalnih časova ne isprazni.

Implementacija je prilično jednostavna.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <utility>

using namespace std;

// časove predstavljamo uređenim parovima (početak, kraj)
typedef pair<int, int> cas;

// kreiramo čas na osnovu sata i minuta početka i kraja
cas napraviCas(int pocSat, int pocMin, int krajSat, int krajMin) {
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

// očitavamo početak časa
inline int pocetakCasa(const cas& c) {
  return c.first;
}

// očitavamo kraj časa
inline int krajCasa(const cas& c) {
  return c.second;
}

int main() {
  // učitavamo podatke o svim časovima
  int n;
  cin >> n;
  vector<cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(pocSat, pocMin, krajSat, krajMin);
  }
  // sortiramo časove po vremenu završetka
  sort(begin(casovi), end(casovi),
       [](const cas& a, const cas& b) {
          return krajCasa(a) < krajCasa(b);
       });

  // prvi čas sigurno može biti održan
  int brojOdrzanihCasova = 1;
  // kraj poslednje održanog časa
```

```
  int kraj = krajCasa(casovi[0]);
  // obrađujemo sve časove po redosledu završetka
  for (int i = 1; i < n; i++)
    // ako je čas broj i kompatibilan sa svim ranijim
    if (pocetakCasa(casovi[i]) >= kraj) {
      // njega zakazujemo i ažuriramo broj održanih časova i
      // kraj poslednjeg održanog časa
      brojOdrzanihCasova++;
      kraj = krajCasa(casovi[i]);
    }

  // ispisujemo konačan rezultat
  cout << brojOdrzanihCasova << endl;

  return 0;
}
```

Dokažimo optimalnost korišćenjem tehnike razmene. Pretpostavimo da je $O = \{c_1, \ldots, \ldots, c_k\}$, skup časova koji predstavlja neko optimalno rešenje, pri čemu su časovi c_1 do c_k sortirani neopadajuće po redosledu njihovog završetka (ako sa $[s_i, f_i)$ obeležimo interval časa c_i, tada je $f_1 \leq f_2 \leq \ldots \leq f_k$). Pošto se svi ti časovi mogu održati, između njih nema preklapanja i svaki naredni počinje nakon završetka prethodnog (važi da je $s_{i+1} \geq f_i$). Neka je $c_i = [s_i, f_i)$ prvi čas u ovom skupu koji ne bi bio izabran našom strategijom. Pretpostavimo da bi naša strategija umesto njega odabrala čas $c'_i = [s'_i, f'_i)$. Pokažimo da se zamenom časa c_i časom c'_i dobija takođe raspored koji je optimalan (on već sadrži k elemenata i potrebno je samo pokazati da je taj raspored takođe ispravan, tj. da su svi časovi međusobno kompatibilni). Na osnovu definicije strategije, čas c'_i se bira između časova koji počinju posle časa c_{i-1}, pa je $s'_i \geq f_{i-1}$ i taj čas je kompatibilan sa svim ranije održanim časovima (časovima $c_1, c_2, \ldots, c_{i-1}$). Potrebno je da pokažemo da je kompatibilan i sa svim časovima koji se održavaju kasnije (časovima $c_{i+1}, \ldots, c_k$). Pokažimo da se c'_i završava pre c_i (ili se eventualno završavaju istovremeno), tj. da je $f'_i \leq f_i$. Zaista, ako je $i = 0$, tada naša strategija bira c'_i koji se prvi završava, pa se stoga c_i ne može završavati pre njega. Ako je $i > 0$, tada naša strategija bira onaj c'_i koji se najranije završava iz skupa svih časova koji koji počinju nakon c_{i-1}. Pošto je početni raspored korektan, znamo da c_i mora pripadati tom skupu, tj. znamo da je $s_i \geq f_{i-1}$. Zato znamo da se c'_i mora završiti pre c_i tj. da je $f'_i \leq f_i$. Dakle, ako postoje časovi u O pre časa c_i, oni ostaju nepromenjeni i čas c'_i se ne preklapa sa njima (jer je $s'_i \geq f_{i-1}$). Pošto se c'_i ne završava kasnije nego c_i tj. pošto je $f'_i \leq f_i$, on se sigurno ne preklapa ni sa jednim časom iz O koji ide posle c_i (jer svi oni počinju nakon kraja časa c_i tj. nakon trenutka f_i, pa zato počinju i nakon f'_i). Dakle, kada se c_i zameni sa c'_i i dalje se dobija ispravan raspored sa istim brojem održanih časova kao O za koji smo pretpostavili da je optimalan. Po istom prinicpu možemo menjati naredne časove (ovo se mora zaustaviti jer u svakom narednom optimalnom skupu imamo po jedan čas više koji je u skladu sa našom strategijom) i tako pokazati da će naša strategija vratiti optimalan skup.

Raspored sa najmanjim brojem učionica

Problem: Za svaki od n časova poznato je vreme početka i završetka. Napiši program koji određuje minimalni broj učionica potreban da se svi časovi održe.

Iako donekle deluje slično prethodnom, rešenje ovog zadatka zahteva drugačiju strategiju. Pošto se zahteva da se svi časovi održe, obilazićemo ih u određenom redosledu i svaki čas ćemo pridruživati nekoj od slobodnih učionica. U trenutku u kom nema više slobodnih učionica koje su ranije otvorene, otvaraćemo novu učionicu.

Naredni primer pokazuje da obilazak u redosledu prvog vremena završetka ne daje uvek optimalno rešenje. Naime, raspored se može napraviti u dve učionice (u učionici A časovi 1 i 4, a u učionici B časovi 2 i 3), a ako bismo učionice obilazili po vremenu završetka, za raspored bi bilo potrebno tri učionice (kako je prikazano na slici).

```
C:            _____4_____
B:   _____2____
A:  ____1____   ___3___
```

Znamo da je najmanji broj učionica sigurno veći ili jednak najvećem broju časova koji se istovremeno održavaju u nekom trenutku. U nastavku ćemo dokazati da je minimalan broj učionica uvek jednak tom broju i da se raspored može napraviti ako se časovi obilaze u rastućem redosledu njihovog početka. Za prethodni primer dobio bi se raspored prikazan na narednoj slici.

```
B:   _____2____ ___3___
A:  ____1____ _____4_____
```

Pogledajmo još jedan primer.

```
C:                __5__   ___8____
B:  __2__  ___3___  ___6___    __10__
A: _____1_____  __4__ ___7___ _9_
```

Dokažimo da je naša strategija optimalna. Strategija je takva da je jedini razlog da se nova učionica otvori to da su sve ranije otvorene učionice već popunjene, tj. da postoji neki čas (recimo $[s_j, f_j)$) koji se seče sa svim časovima koji su raspoređeni u trenutnih d otvorenih učionica. Pošto su časovi sortirani na osnovu vremena početka, svih tih d časova počinje pre trenutka s_j i završava se nakon trenutka s_j (jer traju u trenutku s_j). To znači da u trenutku s_j sigurno postoji $d+1$ časova (tih d već raspoređenih i čas $[s_j, f_j)$) koji se u tom trenutku održavaju, pa broj učionica mora biti bar $d+1$. Dakle, ako se na osnovu strategije rezerviše nova učionica, sigurni smo da je to neophodno. Ako je naša strategija napravila raspored u nekom broju učionica, sigurni smo da nije bilo moguće napraviti raspored u manjem broju učionica, što znači da je napravljeni raspored optimalan.

Primetimo da je ovaj dokaz drugačiji od ranije viđenih dokaza zasnovanih na

tehnici razmene ili na tehnici u kojoj se pokazuje da je pohlepno rešenje uvek ispred po nekom kriterijumu. U ovom dokazu je dokazana granica kvaliteta rešenja (raspored ne može biti u manje učionica nego što je broj časova u najvećoj grupi časova koji se održavaju u nekom trenutku) i zatim je pokazano da pohlepno rešenje dostiže tu granicu.

Prvi korak u implementaciji je veoma jednostavan - učitavamo sve časove u niz i sortiramo ih na osnovu početnog vremena. Ključni korak u drugoj fazi je određivanje učionice u koju može biti smešten tekući čas. Za sve do tada otvorene učionice znamo vremena završetka časova u njima. Možemo pronaći učionicu u kojoj se čas najranije završava i proveriti da li je moguće da u nju rasporedimo tekući čas. Ako jeste, njoj ažuriramo vreme završetka časa, a ako nije, onda moramo otvoriti novu učionicu. Da bismo efikasno mogli da nađemo učionicu u kojoj se čas najranije završava, sve učionice možemo čuvati u redu sa prioritetom sortiranom po vremenu završetka časa u svakoj od učionica. Ako taj red nije prazan i ako je vreme završetka časa u učionici na vrhu reda manje ili jednako vremenu početka tekućeg časa, vreme završetka časa u toj učionici ažuriramo na vreme završetka tekućeg časa (najlakše tako što tu učionicu izbacimo iz reda i ponovo je dodamo sa ažuriranim vremenom). U suprotnom u red dodajemo novu učionicu kojoj je vreme završetka časa postavljeno na vreme završetka tekućeg časa (broj učionice je za jedan veći od dotadašnjeg broja učionica u redu).

Ukupna složenost algoritma je $O(n \log n)$ - i u fazi sortiranja i u fazi raspoređivanja. Kada bismo umesto reda sa prioritetom koristili običan niz i u njemu stalno tražili minimum, složenost najgoreg slučaja bi porasla na $O(n^2)$.

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <utility>

using namespace std;

struct Cas {
  int broj, pocetak, kraj;
};

Cas napraviCas(int broj, int pocSat, int pocMin, int krajSat, int krajMin) {
  Cas c;
  c.broj = broj;
  c.pocetak = pocSat*60 + pocMin;
  c.kraj = krajSat*60 + krajMin;
  return c;
}

struct Ucionica {
  int broj;
  int slobodnaOd;
```

```
};

Ucionica napraviUcionicu(int slobodnaOd, int broj) {
  Ucionica u;
  u.broj = broj;
  u.slobodnaOd = slobodnaOd;
  return u;
}

struct PorediUcionice {
  bool operator()(const Ucionica& u1, const Ucionica& u2) {
    return u1.slobodnaOd > u2.slobodnaOd;
  }
};

int main() {
  int n;
  cin >> n;
  vector<Cas> casovi(n);
  for (int i = 0; i < n; i++) {
    int pocSat, pocMin, krajSat, krajMin;
    cin >> pocSat >> pocMin >> krajSat >> krajMin;
    casovi[i] = napraviCas(i, pocSat, pocMin, krajSat, krajMin);
  }

  // sortiramo časove na osnovu vremena njihovog početka
  sort(begin(casovi), end(casovi),
       [](const Cas& c1, const Cas& c2) {
          return c1.pocetak < c2.pocetak;
       });

  // raspored učionica - svakom času dodeljujemo učionicu
  vector<int> ucionica(n);
  // red u kome čuvamo sve do sada dodeljene učionice, sortirano po redosledu
  // završetka časova u njima
  priority_queue<Ucionica, vector<Ucionica>, PorediUcionice> redUcionica;
  // obilazimo sve časove u redosledu njihovog početka
  for (const Cas& c : casovi) {
    // broj učionice u kojoj će se čas održati
    int brojUcionice;
    if (redUcionica.empty() || redUcionica.top().slobodnaOd > c.pocetak)
      // ako nema slobodnih učionica, otvaramo novu
      brojUcionice = redUcionica.size() + 1;
    else {
      // u suprotnom čas raspoređujemo u onu učionicu koja se najranije ispraznila
      brojUcionice = redUcionica.top().broj;
      redUcionica.pop();
    }
    // beležimo broj učionice za tekući čas
    ucionica[c.broj] = brojUcionice;
```

```
    // dodajemo učionicu u red
    redUcionica.push(napraviUcionicu(c.kraj, brojUcionice));
  }

  // ispisujemo ukupni broj učionica i učionice pridružene časovima
  cout << redUcionica.size() << endl;
  for (int u : ucionica)
    cout << u << endl;

  return 0;
}
```

Raspored sa najmanjim zakašnjenjenjem

Problem: Za svaki od n poslova koje treba završiti poznato je trajanje u minutima i rok do kojeg se posao mora završiti. Svaki posao može da počne u minutu 0. Kvalitet obavljenog posla se ceni na osnovu najvećeg napravljenog zakašnjenja - potrebno je da ono bude što manje. Napisati program koji određuje optimalan raspored i dokazati njegovu korektnost. Na primer, neka su u tabeli data trajanja i rokovi završetka za svaki posao.

```
br  1  2  3  4  5  6
---------------------
ti  1  2  2  3  3  4
di  9  8 15  6 14  9
```

Jedan mogući raspored (napravljen tako da se prvo rade što kraći poslovi) je sledeći:

```
 1 2 2 3 3 4 4 4 5 5 5 6 6 6 6    - broj posla
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5   - minut
```

U njemu se posao 4 završava u minutu 8 i kasni 2 minuta u odnosu na svoj rok 6, dok se posao 6 završava u minutu 15 i kasni 6 minuta u odnosu na svoj rok 9. Najveće kašnjenje u ovom rasporedu je 6 minuta.

Jedan optimalni raspored je sledeći. U njemu se posao 6 završava u minutu 10 i kasni jedan minut u odnosu na svoj rok 9 (to je jedino kašnjenje).

```
 4 4 4 2 2 1 6 6 6 6 5 5 5 3 3   - broj posla
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5  - minut
```

Zadatak je moguće rešiti pomoću gramzivog algoritma. Ključno pitanje je u kom redosledu treba zakazivati poslove. Optimalno rešenje se dobija ako se poslovi zakazuju u redosledu roka njihovog završetka, pri čemu svaki posao počinje odmah nakon što se prethodni završio (ne pravi se pauza ni jedan jedini minut).

Dokažimo da je ova strategija optimalna. Ako su dva posla i i j takvi da je rok završetka prvog r_i pre roka završetka drugog r_j (tj. važi $r_i < r_j$), a da je

početak prvog posla s_i zakazan nakon početka drugog posla s_j (tj. važi $s_i > s_j$), reći ćemo da su oni raspoređeni pogrešno.

Pretpostavimo da je O neki optimalni raspored. Ako u tom rasporedu postoje dva pogrešno raspoređena posla, tada u rasporedu sigurno postoje i dva pogrešno raspoređena uzastopna posla. Naime, pretpostavimo da su poslovi i i j raspoređeni naopako. Pošto je rok završetka posla j ispred roka završetka posla i, prilikom obilaska poslova između i i j u redosledu njihovog zakazanog početka, nije moguće da rok njihovog završetka stalno raste i neophodno je da se prilikom prelaska sa jednog posla na sledeći rok završetka smanji. Dva uzastopna posla kod kojih se to desi su sigurno pogrešno raspoređena.

Dva susedna posla se mogu razmeniti tako da se ostali poslovi ne diraju. Dokažimo da se razmenom dva naopako raspoređena posla maksimalno kašnjenje ne može povećati. Pošto se ostali poslovi ne menjaju razmenom dva susedna naopako raspoređena posla, jedino je relevanto posmatrati njihova kašnjenja. Neka posao i počinje u u trenutku s_i, traje t_i minuta i neka mu je rok završetka d_i. Tada posao j počinje u trenutku $s_i + t_i$, neka traje t_j minuta i neka mu je rok završetka d_j. Kašnjenje posla i je $\max(s_i + t_i - d_i, 0)$ dok je kašnjenje posla j jednako $\max(s_i + t_i + t_j - d_j, 0)$. Pošto su oni raspoređeni naopako važi da je $d_i > d_j$ (i da je $s_i < s_i + t_i$). Nakon razmene njihova kašnjenja postaju $\max(s_i + t_j + t_i - d_i, 0)$ i $\max(s_i + t_j - d_j, 0)$. Posao j je pomeren unapred (za t_i minuta) i njegovo kašnjenje se time samo moglo smanjiti (jer je $\max(s_i + t_j - d_j, 0) \leq \max(s_i + t_i + t_j - d_j, 0)$). Posao i je pomeren unazad, međutim, pošto je $d_i > d_j$, važi da je $\max(s_i + t_i + t_j - d_i, 0) \leq \max(s_i + t_i + t_j - d_j, 0)$, pa posao i ne kasni više nego što je ranije kasnio posao j.

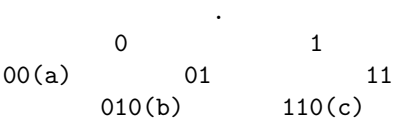
Prethodno opisanom razmenom se broj naopako raspoređenih poslova smanjuje za 1. Razmene možemo ponavljati sve dok ne stignemo do rasporeda u kome nema naopako raspoređenih poslova. Pošto se lako dokazuje da svi rasporedi u kojima nema naopako raspoređenih poslova i nema pauza imaju isto maksimalno kašnjenje (razmenom uzastopnih poslova sa istim rokom ne menja se maksimalno kašnjenje), a naš pohlepni algoritam gradi jedan baš takav raspored, to znači da će se maksimalno kašnjenje našeg rasporeda poklapati sa optimalnim i naš raspored će takođe biti optimalan.

Napomenimo i da naizgled jednostavna modifikacija teksta zadatka dovodi do problema koji nema jednstavno polinomijalno rešenje. Naime, ako bismo umesto najvećeg kašnjenja pokušavali da smanjimo zbir svih kašnjenja dobio bi se NP kompletan problem.

Hafmanovo kodiranje

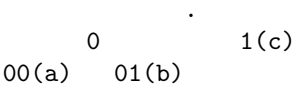
Problem: Neka je zadat tekst koji je potrebno zapisati sa što manje bitova. Pritom je svaki znak teksta predstavljen jedinstvenim nizom bitova - kodom tj. kodnom rečju tog znaka. Ako su dužine kodova svih znakova jednake (što

je upravo slučaj sa standardnim kodovima, kao što je ASCII), broj bitova koji predstavljaju tekst zavisi samo od broja karaktera u njemu. Da bi se postigla ušteda, moraju se neki znaci kodirati manjim, a neki većim brojem bitova. Neka se u tekstu javlja n različitih karaktera i neka se karakter c_i javlja f_i puta $(0 \leq i < n)$ i neka je za njegovo kodiranje potrebno b_i bitova. Jedan od uslova da se obezbedi jednoznačno dekodiranje je da nijedna kodna reč ne bude prefiks nekoj drugoj (takvi se kodovi nazivaju prefiksni kodovi). Binarnim prefiksnim kodovima jednoznačno odgovaraju binarna drveta. Kôd svakog čvora se dobija obilaskom drveta - korak na levo dodaje simbol 0 na kod, a korak na desno simbol 1, pri čemu se karakteri koje je potrebno kodirati nalaze u listovima drveta (zato nije moguće da je kôd bilo kog karaktera prefiks koda nekog drugog karaktera).



Za dati skup karaktera i frekvencije njihovog pojavljivanja konstruisati optimalni binarni prefiksni kôd (onaj kod kojeg je broj bitova $\sum_{i=0}^{n-1} f_i b_i$ potreban za zapis celog teksta najmanji). Dokazati optimalnost.

Razmotrimo prvo strukturu drveta koje odgovara binarnom prefiksnom kodu. Prvi važan zaključak je da u optimalnom prefiksnom kodu svi unutrašnji čvorovi moraju imati oba deteta. U suprotnom se kôd može skratiti tako što se unutrašnji čvor ukloni i zameni svojom decom. U prethodnom primeru, drvo možemo transformisati tako što unutrašnji čvor `01` zamenimo svojim detetom i tako karakteru `b` dodelimo kod `01` umesto `010`. Slično, čvor `11`, a zatim i čvor `1` možemo zameniti svojom decom i tako karakteru `c` možemo dodeliti kôd `1`.



Intuicija nam govori da je poželjno karakterima koji se pojavljuju često dodeljivati kraće kodove. Zato će karakteri koji se javljaju ređe imati duže kodove. Ovo nije teško formalno dokazati. Pretpostavimo da karakter c_i ima frekvenciju pojavljivanja f_i i da se kodira sa b_i bitova, a da karakter c_j ima frekvenciju pojaljivanja f_j i da se kodira sa b_j bitova, pri čemu je $f_i \geq f_j$, dok je $b_i \leq b_j$. Ako se zamene kodovi ta dva karaktera, ukupan potreban broj bitova za kodiranje teksta se ne može povećati. Pošto se ostalim karakterima kodovi ne menjaju, broj bitova zavisi samo od ova dva karaktera. U prvom slučaju on je jednak $f_i b_i + f_j b_j$, a u drugom je jednak $f_i b_j + f_j b_i$. Ako posmatramo razliku $(f_i b_j + f_j b_i) - (f_i b_i + f_j b_j)$ dobijamo izraz $f_i(b_j - b_i) + f_j(b_i - b_j)$ tj. $(f_i - f_j)(b_j - b_i)$. Pošto su oba činioca nenegativna i polazna razlika je nenegativna, pa se ukupan potreban broj bitova ovom transformacijom nije mogao povećati (ako je razlika pozitivna, onda se broj bitova smanjio).

Naredno važno tvrđenje je to da se dva karaktera koji se najređe javljaju u

optimalnom drvetu mogu naći kao dva susedna lista najudaljenija od korena. Zaista, ako su to karakteri c_1 i c_2 i ako se na mestu najudaljenijem od korena nalaze karakteri c_1' i c_2', tada možemo zameniti c_1 i c_1' i zameniti c_2 i c_2'. Struktura drveta ostaje identična, pa se nakon razmene i dalje dobija ispravan binarni prefiksni kôd. Pošto su nakon ove razmene kodovi zamenjeni tako da su češći karakteri dobili kraće kodove, na osnovu prethodno dokazanog tvrđenja ukupan broj bitova za kodiranje teksta se nije mogao povećati.

Dakle, znamo da postoji optimalni binarni prefiksni kôd u kome su dva najređa karaktera c_i i c_j deca istog čvora i dva najudaljenija lista. Njihovim uklanjajem iz optimalnog drveta A za polaznu azbuku dobija se drvo B za koje tvrdimo da je optimalno za azbuku u kojoj su ta dva karaktera uklonjena i zamenjena sa novim karakterom c čija je frekvencija $f_i + f_j$ (gde su f_i i f_j frekvencije karaktera c_i i c_j). Zaista, ako drvo B ne bi bilo optimalno za modifikovanu azbuku, postojalo bi bolje drvo za nju (recimo B'). U tom drvetu B' list c možemo zameniti unutrašnjim čvorom čija su deca c_i i c_j i tako dobiti drvo A' za polaznu azbuku. Dokažimo da ako bi drvo B' bilo bolje od drveta B, tada bi drvo A' bilo bolje od drveta A, što je kontradikcija. Ukupan broj bitova u drvetu A i B se razlikuje samo za bitove kojima se kodiraju karakteri c_i, c_j i novi karakter c. U drvetu A to je $f_i b + f_j b$ bitova, gde je b broj bitova potreban za kodiranje karaktera c_i i c_j, dok je u drvetu B to $(f_i + f_j)(b-1)$, jer je karakteru c pridružena frekvencija $f_i + f_j$, dok je visina čvora c za 1 manja od visine čvorova c_i i c_j. Dakle, razlika između broja bitova u ta dva drveta je $f_i + f_j$, što je konstanta. Slično, broj bitova za koji se razlikuju A' i B' je takođe $f_i + f_j$. Dakle, ako je broj bitova za B' manji od broja bitova za B, tada bi broj bitova za A' morao da bude manji od broja bitova za A, što je kontradikcija.

Na osnovu prethodnog razmatranja lako se može formulisati induktivno-rekurzivni algoritam. Određujemo dva karaktera sa najmanjim frekvencijama pojavljivanja, menjamo ih novim karakterom čija je frekvencija jednaka zbiru njihovih frekvencija, rekurzivno konstruišemo optimalno drvo i na kraju u tom drvetu na list koji odgovara novom karakteru dopisujemo dva nova lista koji odgovaraju uklonjenim karakterima sa najmanjim frekvencijama. Bazu indukcije tj. izlaz iz rekurzije predstavlja slučaj kada ostanu samo dva karaktera (i tada i jedan i drugi kodiramo sa po jednim bitom, tj. kreiramo koren drveta čija su ta dva karaktera listovi).

Potrebno je precizirati još nekoliko detalja da bismo napravili implementaciju. U svakom koraku je u skupu karaktera potrebno određivati dva sa najmanjim frekvencijama i menjati ih sa novim, čija je frekvencija jednaka zbiru frekvencija. Ove operacije se mogu veoma efikasno izvršavati ako se karakteri ubace u red sa prioritetom (hip) sortiran rastuće na osnovu frekvencija. Drugo pitanje je to kako reprezentovati drvo koje se gradi. Unapred znamo da će ono imati $2n-1$ čvorova i znamo da će n karaktera biti listovi tog drveta. Svaki čvor možemo numerisati brojevima od 0 do $2n-2$. Za svaki od $n-1$ unutrašnjih čvorova treba da znamo indeks levog i desnog deteta (uvek postoje oba). Te informacije možemo čuvati u dva pomoćna niza. Unutrašnji čvorovi imaju indekse od n

do $2n-2$ i za čvor broj k u prvom nizu na poziciji $k-n$ čuvamo indeks levog deteta, a u drugom nizu na poziciji $k-n$ čuvamo indeks desnog deteta.

Kada je drvo kreirano, kodove svih karaktera možemo dobiti iscrpnim obilaskom celom drveta (rekurzivnom funkcijom).

```
// mapu kodovi popunjava kodovima svih karaktera koji su naslednici čvora i
// čiji je kod jednak niski kod
void procitajKodove(const vector<int>& levo,
                    const vector<int>& desno,
                    const vector<char>& karakteri,
                    int i, int n, const string& kod,
                    map<char, string>& kodovi) {
  if (i < n)
    // stigli smo do lista
    kodovi[karakteri[i]] = kod;
  else {
    // u pitanju je unutrašnji čvor
    procitajKodove(levo, desno, karakteri, levo[i-n], n, kod + "0", kodovi);
    procitajKodove(levo, desno, karakteri, desno[i-n], n, kod + "1", kodovi);
  }
}

// mapu kodovi popunjava kodovima svih karaktera iz datog drveta
void procitajKodove(const vector<int>& levo,
                    const vector<int>& desno,
                    const vector<char>& karakteri, int n,
                    map<char, string>& kodovi) {
  // čitanje kreće od korena, čiji je kod prazna niska
  procitajKodove(levo, desno, karakteri, 2*n-2, n, "", kodovi);
}

// određuje najmanji broj bitova potrebnih
int brojBitova(const vector<char>& karakteri, const vector<int>& frekvencije) {
  int n = karakteri.size();
  // indeksi dece unutrašnjih čvorova drveta
  vector<int> levo(n-1), desno(n-1);
  // red sa prioritetom u kome čuvamo parove frekvencija i indeksa čvorova, uređen
  // rastuće po frekvencijama
  priority_queue<pair<int, int>, vector<pair<int, int>>,
                 greater<pair<int, int>>> pq;
  // sve karaktere ubacujemo u red
  for (int i = 0; i < n; i++)
    pq.push(make_pair(frekvencije[i], i));

  // određujemo jedan po jedan unutrašnji čvor, sve dok ne završimo sa korenom (2n-2)
  for (int i = n; i < 2*n - 1; i++) {
    // uklanjamo dva karaktera sa najmanjim frekvencijama
    auto f1 = pq.top(); pq.pop();
    auto f2 = pq.top(); pq.pop();
```

```
    // dodeljujemo tim karakterima novi unutrašnji čvor
    levo[i - n] = f1.second; desno[i - n] = f2.second;
    // kreiramo novi kombinovani karakter
    pq.push(make_pair(f1.first + f2.first, i));
  }

  // čitamo iz drveta kodove svih karaktera
  map<char, string> kodovi;
  procitajKodove(levo, desno, karakteri, n, kodovi);

  // svakom karakteru pridružujemo njegovu frekvenciju
  map<char, int> frekvencijeKaraktera;
  for (int i = 0; i < n; i++)
     frekvencijeKaraktera[karakteri[i]] = frekvencije[i];

  // izračunavamo i vraćamo ukupan broj bitova
  int brojBitova = 0;
  for (auto it : kodovi)
    brojBitova += frekvencijeKaraktera[it.first] * it.second.length();
  return brojBitova;
}
```

Vagoni

Problem: Na pruzi se nalaze vagoni označeni brojevima od 1 do n, ali su ispremetani. Potrebno je prebaciti ih na drugi kraj pruge u tako da su poređani od 1 do n. Sa strane pruge nalaze se tri pomoćna koloseka na koji se privremeno mogu smeštati vagoni. Pruga sa pomoćnim kolosecima ima oblik ćiriličkog slova Š sa produženom donjom osnovom.

```
     3 pomoćna koloseka
         |   |   |
ulaz ----+---+---+----  izlaz
```

Vagon se može prebaciti sa ulaza na pomoćni kolosek i sa pomoćnog koloseka na izlaz - nije moguće prebacivanje sa jednog na drugi pomoćni kolosek. Svaki pomoćni kolosek može čuvati proizvoljan broj vagona. Pomoćni koloseci su slepi i redosled izlaska vagona sa pomoćnih koloseka je obratan od redosleda njihovog ulaska.

Razmislimo kako možemo modelovati zadati problem. Vagoni koje treba premestiti se mogu smestiti u red, dok se pomoćni koloseci ponašaju kao stekovi. Čim vagon sa ulaza ili sa nekog pomoćnog koloseka može da se prebaci na izlaz to bez odlaganja treba uraditi (to pomeranje ne može zasmetati ni jednom narednom pomeranju, a odlaganje potencijalno može dovesti do toga da vagon koje je sada moguće prevesti na izlaz postane blokiran). Ključni uvid za rešenje zadatka je da se vagon koji ne može odmah da izađe na izlaz stavlja na onaj pomoćni kolosek na na čijem vrhu se nalazi vagon većom oznakom od oznake

tekućeg vagona. U slučaju da ima više takvih bira se onaj sa najmanjom oznakom vagona na vrhu. U slučaju da takvih nema, a postoji prazan kolosek, vagon se postavlja na prazan kolosek. U suprotnom znamo da raspoređivanje nije moguće. Ovo bi trebalo da je intuitivno jasno. Vagon stavljamo na pomoćni kolosek samo kada ne može da pređe direktno na izlaz i cilj je da postavljanje uradimo tako da on u budućnosti što manje smeta budućim vagonima. Ako bi se postavio iznad vagona sa manjom oznakom, došli bismo u situaciju da taj donji vagon nikada ne možemo izvesti na izlaz (jer da bi se vagon sa većom oznakom koji postavljamo iznad njega mogao izvesti, potrebno je da budu izvedeni svi vagoni sa manjim oznakama od njega, pa i ovaj vagon koji je njime upravo zaklonjen). Dakle, vagon moramo postaviti ili na onaj kolosek na čijem vrhu je vagon sa većim brojem ili na prazan kolosek. Taj vagon će sigurno ograničiti jedan pomoćni kolosek tako da na njega mogu da stanu samo brojevi sa manjim oznakama. Potrebno je jedino voditi računa da druga dva koloseka budu što manje ograničena tj. vagon treba postaviti na onaj kolosek koji je već sada najviše ograničen, jer se tako u budućnosti ništa neće pokvariti. Naime, svaki vagon koji bi mogao da bude postavljen na taj najviše ograničen kolosek, moći će da bude postavljen i na druge koloseke (jer su oni manje ograničeni). Možemo smatrati da je prazan kolosek je najmanje ograničen (na njega može stati bilo koji vagon), što možemo modelovati tako da na dno svih koloseka u početku postavimo fiktivnu vrednost $+\infty$.

Sve ovo omogućava da se napravi pohlepni algoritam koji izbegava proveru raznih mogućnosti nego do optimalnog rešenja (u ovom slučaju to je premeštanje svih vagona) tako što u svakom pojedinačnom koraku vagon postavlja na optimalno mesto (na drugi kraj pruge ako je to moguće, tj. na kolosek koji je u tom trenutku najviše ograničen).

```
#include <iostream>
#include <stack>
#include <queue>
#include <limits>
using namespace std;

int main() {
  // učitavamo vagone na ulazu i smeštamo ih u red iz kojeg će
  // izlaziti jedan po jedan
  queue<int> ulaz;
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    ulaz.push(x);
  }

  // tri pomoćna koloseka - jednostavnosti dalje analize radi na dno
  // svakog postavljamo vrednost beskonačno
```

```
stack<int> s[3];
for (int i = 0; i < 3; i++)
  s[i].push(numeric_limits<int>::max());

// broj vagona koji treba da izađe
int treba_da_izadje = 1;

// da li je moguće sve vagone izvesti
bool moze = true;
while (treba_da_izadje <= n) {
  // ako se onaj koji treba da izađe nalazi na vrhu
  // nekog koloseka, njega izvodimo
  bool izasao_sa_pomocnog = false;
  for (int i = 0; i < 3; i++) {
    if (s[i].top() == treba_da_izadje) {
      cout << "Sa koloseka " << i << " izlazi "
           << treba_da_izadje << endl;
      s[i].pop(); treba_da_izadje++;
      izasao_sa_pomocnog = true;
    }
  }
  if (izasao_sa_pomocnog)
    continue;

  // uzimamo sledeći vagon sa ulaza
  int vagon = ulaz.front();
  ulaz.pop();

  // ako je on na redu, samo ga prevodimo na izlaz
  if (vagon == treba_da_izadje) {
    cout << "Sa ulaza izlazi " << treba_da_izadje << endl;
    treba_da_izadje++;
    continue;
  }

  // vagon stavljamo na onaj pomoćni kolosek na koji može
  // da stane, ali tako da je na vrhu tog koloseka najmanji
  // mogući broj
  int min = -1;
  for (int i = 0; i < 3; i++) {
    if (vagon < s[i].top() &&
        (min == -1 || s[i].top() < s[min].top()))
      min = i;
  }

  // ako nismo našli pomoćni kolosek,ređanje vagona
  // nije moguce
  if (min == -1) {
    moze = false;
    break;
```

```
    }

    // postavljamo vagon na odabrani pomoćni kolosek
    cout << "Sa ulaza vagon " << vagon
         << " ide na kolosek " << min << endl;
    s[min].push(vagon);
  }

  if (moze)
    cout << "moze" << endl;
  else
    cout << "ne moze" << endl;

  return 0;
}
```

Permutacija niske bez istih susednih elemenata

Problem: Naći neku permutaciju niske sastavljene od malih slova engleske abecede u kojoj nikoja dva susedna elementa nisu jednaka ili odrediti da takva permutacija ne postoji. Na primer, ako niska `aabbcc`, se može permutovati u `abacbc`, dok se niska `aabbbb` ne može permutovati. Dokazati korektnost algoritma.

I ovaj zadatak se može rešiti gramzivim pristupom. Naime, nakon popunjavanja određenog dela niske, na sledeće mesto možemo staviti karakter iz preostalog skupa karaktera, različit od poslednjeg postavljenog karaktera, koji se najčešće javlja u tom preostalom skupu karaktera.

Razmotrimo kako efikasno možemo implementirati prethodni gramzivi pristup. Pošto je u svakom koraku potrebno da određujemo karakter sa najvećom frekvencijom, zgodno je održavati red sa prioritetom u kom su karakteri poređani opadajuće po frekvencijama. Nakon uzimanja karaktera sa najvećom frekvencijom i postavljanja u nisku, na tekuće mesto, taj karakter možemo izbaciti iz reda sa prioritetom, jer je na sledećoj poziciji zabranjeno njegovo pojavljivanje. Međutim, nakon postavljanja sledećeg karaktera, prethodni karakter ponovo postaje kandidat, tako da ga u tom trenutku vraćamo nazad u red, ali sa frekvencijom umanjenom za jedan.

```
bool permutacija(string& s) {
  priority_queue<pair<int, char>> red;
  int frekvencije[26] = {0};
  for (char c : s)
    frekvencije[c - 'a']++;
  for (int i = 0; i < 26; i++)
    if (frekvencije[i] > 0)
      red.push(make_pair(frekvencije[i], 'a' + i));
```

```
  int preostalo_prethodnih;
  for (int i = 0; i < s.size(); i++) {
    if (red.empty()) return false;
    s[i] = red.top().second;
    int preostalo_trenutnih = red.top().first - 1;
    red.pop();
    if (i > 0 && preostalo_prethodnih > 0)
      red.push(make_pair(preostalo_prethodnih, s[i-1]));
    preostalo_prethodnih = preostalo_trenutnih;
  }
  return true;
}
```

Formiranje brojeva najmanjeg zbira

Problem: Od datog niza od n cifara (od 0 do 9) odrediti minimalnu moguću sumu dva broja formirana od cifara datog niza. Sve cifre niza moraju biti iskorišćene onoliko puta koliko se pojavljuju. Na primer, za $5, 3, 0, 7, 4$ dobija se 82 (tj. $35 + 047$). Dokazati korektnost.

Da bi zbir bio što manji, potrebno je da oba broja budu što kraća i da počinju što manjim ciframa. Kada odredimo najmanji zbir dobijen pomoćnu cifara različitih od nule, sve nule možemo dopisati na početak bilo kog od brojeva (ili ih ravnomerno rasporediti na oba broja) i zbir se neće promeniti. Ako bi se neka nula javljala na poziciji koja nije na početku broja, tada bi broj počinjao nekom cifrom koja nije nula i razmenom te cifre i nule dobio bi se manji broj, pa time i manji zbir. Dakle, sve nule možemo nadalje eliminisati iz razmatranja. Preostale cifre moramo ravnomerno raspoređivati u oba broja i ređati ih tako da su u svakom broju cifre opadajuće. Sve raspoložive cifre ćemo sortirati neopadajuće, održavaćemo tekuća dva broja i svaku narednu cifru ćemo dodavati na kraj manjeg od dva broja (ako su jednaki, svejedno je kojem će se cifra dodati). Naime, neka su tekući brojevi a i b, neka je $a \leq b$ i neka je tekuća cifra c. Dodavanjem cifre c na kraj manjeg broja a dobijamo zbir $10a + c + b$, a dodavanjem cifre c na kraj većeg broja b dobijamo zbir $10b + c + a$. Drugi zbir je uvek veći ili jednaki od prvog, jer je njihova razlika uvek nenegativna. Naime $(10b + c + a) - (10a + c + b) = 9(b - a) \geq 0$, jer je $b - a \geq 0$. Recimo još i da se u slučaju dodavanje cifre c nekom od brojeva na mesto koje nije poslednje daje zbir koji je veći ili jednak. Naime, pošto su cifre uređene neopadajuće važi da su sve cifre u broju manje ili jednake c. Ako poredimo broj u kome je cifra c dopisana na kraj i broj u kojem je cifra c dopisana negde u sredinu ili na početak, drugi će biti veći ili jednak od prvog, jer imaju isto cifara, a drugi će leksikografski biti veći od prvog.

```
int minZbir(int cifre[], int n) {
   sort(cifre, next(cifre, n));
   int a = 0, b = 0;
   for (int i = 0; i < n; i++) {
```

```
      a = 10 * a + cifre[i];
      if (a > b)
        swap(a, b);
   }
   return a + b;
}
```

Recimo i da bi dodavanje cifara naizmenično na kraj jednog, pa na kraj drugog broja takođe daje optimalno rešenje. Naime, svako dodavanje cifre na kraj broja koji je manji ili jednak od drugog čini da taj broj postane veći ili jednak od drugog (zašto?).

Najveći broj sa cik-cak parnošću cifara

Problem: Od cifara datog broja (zadatog kao niska karaktera) sastaviti što je moguće veći broj, tako da nikoje dve susedne cifre tog broja nisu iste parnosti. Dokazati korektnost.

Najmanji broj dobijen konkatenacijom brojeva

Problem: Dato je n brojeva. Odredi najmanji broj koji se može dobiti njihovom konkatenacijom. Dokazati korektnost.