Sadržaj

Predgovor

Očekivano predznanje

1. Ispravnost programa

1.1 Ispravnost programa

Jedno od centralnih pitanja u razvoju programa je pitanje njegove ispravnosti (korektnosti). Softver je u današnjem svetu prisutan na svakom koraku: softver kontroliše mnogo toga — od bankovnih računa i komponenti televizora i automobila, do nuklearnih elektrana, aviona i svemirskih letelica. U svom tom softveru neminovno su prisutne i greške. Greška u funkcionisanju daljinskog upravljača za televizor može biti tek uznemirujuća, ali greška u funkcionisanju nuklearne elektrane može imati razorne posledice. Najopasnije greške su one koje mogu da dovedu do velikih troškova, ili još gore, do gubitka ljudskih života. Krupne softverske greške i dalje se neprestano javljaju i one koštaju svetsku ekonomiju milijarde dolara. Neki primeri softverskih grešaka koji su izazvali velike probleme opisani su u dodatku 1.A.1.

1.2 Specifikacija programa

Postupak pokazivanja da je program ispravan naziva se *verifikovanje programa*. Međutim, često se pokazuje da pitanje šta uopšte znači da je program ispravan nije uvek očigledno. Na primer, da li je naredna funkcija koja izračunava stepen ispravna?

```
double stepen(double x, int n) {
    double s = 1.0;
    for (int i = 0; i < n; i++)
        s *= x;
    return s;
}
```

Koji je rezultat sledećih poziva `stepen(2.0, 10)`, `stepen(-2.0, 10)`, `stepen(2.0,-10)`, `stepen(0.0, 0)`, `stepen(1e200, 2)`? Da li su ti rezultati ispravni? Da li je uopšte dozvoljeno da izložilac bude negativan ili da se izračunava 0^0? Da li rezultat $(10^{200})^2$ može uopšte da se ispravno zapiše pomoću tipa `double`? Sve ovo treba precizirati da bi se uopšte moglo diskutovati da li je ova implementacija stepenovanja ispravna.

Dakle, pre programa potrebno je najpre precizno formulisati pojam ispravnosti programa. Ispravnost programa počiva na pojmu *specifikacije*. Specifikacija je, intuitivno, opis željenog ponašanja programa koji treba napisati. Specifikacija je obično stvar dogovora između naručioca programa (korisnika) i tima programera koji program razvijaju. Veoma važno je da se pre same implementacije što preciznije dogovori šta je očekivano ponašanje programa. Često se specifikacija menja i precizira tokom razvoja programa.

Specifikacija se obično zadaje u terminima *preduslova* tj. uslova koje ulazni parametri programa moraju da zadovolje, kao i *postuslova* tj. uslova koje rezultati izračunavanja moraju da zadovolje. U primeru funkcije stepenovanja preduslov može zahtevati da je vrednost broja $n \geq 0$ i da ako je $n = 0$, tada je $x \neq 0$, a postuslov da funkcija vraća vrednost x^n. Kada je poznata specifikacija, potrebno je verifikovati program, tj. dokazati da on zadovoljava specifikaciju. Prikazana implementacija je ispravna u odnosu na ovu specifikaciju samo ako se zanemare problemi preciznosti i opsega zapisa realnih brojeva u računaru.

U okviru verifikacije programa, veoma važno pitanje je pitanje zaustavljanja programa. *Parcijalna korektnost* podrazumeva da neki program, ukoliko se zaustavi, daje korektan rezultat (tj. rezultat koji zadovoljava specifikaciju). *Totalna korektnost* podrazumeva da se program za sve (specifikacijom dopuštene) ulaze zaustavlja, kao i da su dobijeni rezultati korektni.

Dva osnovna pristupa verifikaciji su:

- **dinamička verifikacija** koja podrazumeva proveru ispravnosti u fazi izvršavanja programa, najčešće putem testiranja;
- **statička verifikacija** koja podrazumeva analizu izvornog koda programa, često korišćenjem formalnih metoda i matematičkog aparata.

O svakom od njih će biti više reči u nastavku.

1.2.1 Najčešći izvori grešaka

Sintaksičke greške u programima su najmanje opasne i najlakše otklonjive, jer ih prijavljuje kompilator i program nije moguće pokrenuti dok se sve sintaksičke greške ne isprave.

Neke greške (na primer, deljenja nulom ili pokušaja pristupa nedozvoljenom delu memorije) mogu dovesti do nasilnog prekida izvršavanja (sintaksički ispravnog) programa (kaže se ponekada da "aplikacija puca"). Mnogo opasnije su logičke greške tj. situacije u kojima se programi uspešno kompilira, pokreće i izvršava ali ne radi ono što je od njega očekivano tj. daje pogrešne rezultate. Greška može biti i u samoj specifikaciji tj. program može da radi tačno ono što je od programera zahtevano, ali to može biti drugačije od onoga što korisnik očekuje ili što je želeo. Veoma važna grupa grešaka su one koje su vezane za bezbednost i sigurnost tj. greške usled kojih zlonamerni korisnici mogu neovlašćeno doći u posed nekih podataka ili resursa.

Navedimo i nekoliko najčešćih uzroka grešaka na nivou implementacije i savete kako da se one zaobiđu.

- **Greške prekoračenja.** Čest razlog grešaka u programima je korišćenje neodgovarajućih tipova podataka, tj. tipova koji uzrokuju da se usled prekoračenja ili potkoračenja dobiju netačni rezultati izračunavanja. Stoga je uvek poželjno imati u vidu moguće raspone vrednosti ulaznih podataka i na osnovu njih proveriti da li su rasponi svih međurezultata i završnih rezultata takvi da mogu da se predstave odabranim tipovima promenljivih. U sklopu provere graničnih vrednosti, poželjno je uvek uključiti i namjanje i najveće moguće vrednosti ulaznih veličina.

- **Provera graničnih vrednosti.** Veliki broj grešaka javlja se na granicama opsega petlje, graničnim indeksima niza, graničnim vrednostima argumenata aritmetičkih operacija i slično. Zbog toga je testiranje na graničnim vrednostima izuzetno važno i program koji prolazi takve ulazne veličine često je ispravan i za druge. Na primer, ako se testira program koji učitava jedan red teksta u neki niz, trebalo bi proveriti da li program ispravno radi kada je na ulazu prazan red (tj. red koji sadrži samo karakter `'\n'`). Drugom granicom mogu se smatrati ulazni redovi koji su veoma dugi (duži od broja elemenata niza u koji se učitavaju podaci) ili pre čijeg kraja se nalazi kraj toka podataka. Stoga bi trebalo proveriti i da li se program ispravno ponaša pri učitavanju redova koji su dugi koliko i niz u koji se učitavaju podaci ili kraći za jedan karakter, duži za jedan karakter i slično.

- **Proveravanje pre-uslova.** Da bi neko izračunavanje imalo smisla često je potrebno da važe neki pre-uslovi. Na primer, ako je potrebno izračunati prosečan broj poena n studenata, pitanje je šta raditi ukoliko je n jednako 0 i ponašanje programa treba da bude specifikovano i testirano u takvim situacijama. Moguće je, na primer, da se ako je n jednako 0, vrati 0 kao rezultat. Moguće je i da se, ako je n jednako 0, ne dozvoli izračunavanje proseka. U ovom drugom pristupu, pre koda za izračunavanje proseka može da se navede naredba `assert(n > 0);`, koja u fazi razvoja programa uzrokuje da se prijavi poruka o tome da preduslov zahtevan specifikacijom nije bio ispunjen.

- **Proveravanje uspešnosti poziva funkcija.** Čest izvor grešaka tokom izvršavanja programa je neproveravanje onih povratnih vrednosti funkcija koje ukazuju na to da li je funkcija uspešno uradila ono što što je trebalo. Na primer, funkcije za alokaciju memorije, funkcije za rad sa datotekama itd., naročito ako potiču iz programskog jezika C, kroz svoju povratnu vrednost obaveštavaju programera da li je došlo do neke greške. Povratne vrednosti ovih funkcija ukazuju na potencijalni problem i ukoliko se ignorišu – problem će samo postati veći. Slično, mnoge funkcije u jeziku C++ dovode do izuzetaka ako ne mogu uspešno da izvrše svoj zadatak. Te izuzetke bi trebalo "uhvatiti" i obraditi tj. učiniti ono što je moguće da program nastavi sa radom ili da bar korisniku prijavi jasnu poruku o tome zašto je došlo do greške i zašto je izvršavanje programa moralo biti prekinuto.

1.3 Dinamičko verifikovanje programa – testiranje i debagovanje

Dinamičko verifikovanje programa podrazumeva proveravanje ispravnosti u fazi izvršavanja programa. Najčešći vid dinamičkog verifikovanja programa je testiranje.

1.3.1 Testiranje

Najznačajnija vrsta dinamičkog ispitivanja ispravnosti programa je testiranje. Testiranje može da obezbedi visok stepen pouzdanosti programa, ali najčešće ne može da dokaže da je program u potpunosti korektan.

Neka tvrđenja o programu je moguće testirati, dok neka nije. Na primer, tvrđenja poput "program za ovaj ulaz vraća ovaj izlaz" ili "program ima prosečno vreme izvršavanja pola sekunde" su (u principu) proveriva testovima. Međutim, tvrđenje

"prosečno vreme izvršavanja programa je dobro" suviše je neodređeno da bi moglo da bude testirano.

U idealnom slučaju, treba sprovesti iscrpno testiranje rada programa za sve moguće ulazne vrednosti i proveriti da li izlazne vrednosti zadovoljavaju specifikaciju. Međutim, ovakav iscrpan pristup testiranju skoro nikada nije praktično primenljiv. Na primer, iscrpno testiranje korektnosti programa koji sabira dva 32-bitna broja, zahtevalo bi ukupno $2^{32} \cdot 2^{32} = 2^{64}$ različitih testova. Pod pretpostavkom da svaki test traje jednu nanosekundu, iscrpno testiranje bi zahtevalo približno $1{,}8 \cdot 10^{10}$ sekundi što je oko 570 godina. Dakle, testiranjem nije praktično moguće dokazati ispravnost čak ni prilično trivijalnih programa. S druge strane, testiranjem je moguće dokazati da program nije ispravan tj. pronaći greške u programima.

S obzirom na to da iscrpno testiranje nije praktično primenljivo, obično se koristi tehnika testiranja tipičnih ulaza programa kao i specijalnih, karakterističnih ulaznih vrednosti za koje postoji veća verovatnoća da dovedu do neke greške. U slučaju pomenutog programa za sabiranje, tipični slučaj bi se odnosio na testiranje korektnosti sabiranja nekoliko slučajno odabranih parova brojeva, dok bi za specijalne slučajeve mogli biti proglašeni slučajevi kada je neki od sabiraka 0, 1, -1, najmanji negativan broj, najveći pozitivan broj i slično.

Postoje različite metode testiranja, a neke od njih su:

- **Testiranje zasebnih jedinica (engl. unit testing)** U ovoj metodi testiranja, nezavisno se testovima proverava ispravnost zasebnih jedinica koda. "Jedinica" je obično najmanji deo programa koji se može testirati. U proceduralnim jezicima, "jedinica" je obično jedna funkcija. Svaki *jedinični test* treba da bude nezavisan od ostalih, ali puno jediničnih testova može da bude grupisano u baterije testova, u jednoj ili više funkcija sa ovom namenom. Jedinični testovi treba da proveravaju ponašanje funkcije, za tipične, granične i specijalne slučajeve. Ova metoda veoma je važna u obezbeđivanju veće pouzdanosti kada se mnoge funkcije u programu često menjaju i zavise jedna od drugih. Kad god se promeni željeno ponašanje neke funkcije, potrebno je ažurirati odgovarajuće jedinične testove. Ova metoda veoma je korisna zbog toga što često otkriva trivijalne greške, ali i zbog toga što jedinični testovi predstavljaju svojevrsnu specifikaciju.

  Postoje specijalizovani softverski alati i biblioteke koje omogućavaju jednostavno kreiranje i održavanje ovakvih testova. *Jedinične testove* obično pišu

i koriste, u toku razvoja softvera, sami autori programa ili testeri koji imaju pristup kodu.

- **Regresiono testiranje (engl. regression testing)** U ovom pristupu, proveravaju se izmene programa kako bi se utvrdilo da se nova verzija ponaša isto kao stara (na primer, generiše se isti izlaz). Za svaki deo programa implementiraju se testovi koji proveravaju njegovo ponašanje. Pre nego što se napravi nova verzija programa, ona mora da uspešno prođe sve stare testove kako bi se osiguralo da ono što je ranije radilo radi i dalje, tj. da nije narušena ranija funkcionalnost programa.

  Regresiono testiranje primenjuje se u okviru samog implementiranja softvera i obično ga sprovode testeri.

- **Integraciono testiranje (engl. integration testing)** Ovaj vid testiranja primenjuje se kada se više programskih modula objedinjuje u jednu celinu i kada je potrebno proveriti kako funkcioniše ta celina i komunikacija između njenih modula. Integraciono testiranje obično se sprovodi nakon što su pojedinačni moduli prošli kroz druge vidove testiranja. Kada nova programska celina, sastavljena od više modula uspešno prođe kroz integraciono testiranje, onda ona može da bude jedna od komponenti celine na višem nivou koja takođe treba da prođe integraciono testiranje.

- **Testiranje valjanosti (engl. validation testing)** Testiranje valjanosti treba da utvrdi da sistem ispunjava zadate zahteve i izvršava funkcije za koje je namenjen. Testiranje valjanosti vrši se na kraju razvojnog procesa, nakon što su uspešno završene druge procedure testiranja i utvrđivanja ispravnosti. Testovi valjanosti koji se sprovode su testovi visokog nivoa koji treba da pokažu da se u obradama koriste odgovarajući podaci i u skladu sa odgovarajućim procedurama, opisanim u specifikaciji programa.

1.3.2 Automatsko testiranje

Testiranje, kao jedan od ključnih vidova za osiguranje kvaliteta softvera (eng. quality assurance), se danas razvilo se u dobro utemeljenu i široko podržanu oblast računarstva. Jedan od pravaca unapređivanja procesa testiranja je njegova automatizacija. Umesto da testiranje sprovodi osoba (tester), automatskim testiranjem taj postupak preuzima računar i sprovodi ga automatski i mnogo brže, dok tester može da se posveti dizajnu automatskih testova i analizi rezultata automatskog testiranja.

U savremenim metodologijama za razvoj softvera, testiranje se u proces razvoja uvodi rano i zahteva saradnju programera i testera. Autor automatskih testova realizuje ih korišćenjem namenskih biblioteka i alata (koji mogu, na primer, da simuliraju i komunikaciju korisnika putem grafičkog korisničkog interfejsa). Programer tako automatizovane testove može da koristi u okviru nekih integrisanih razvojnih okruženja. Postoji mnoštvo alata za automatizovano sprovođenje testiranja. Neki od njih deo su integrisanih razvojnih okruženja a neki se koriste kao samostalni sistemi. Oni se razlikuju i po veličini, zahtevanom umeću i po vrsti podrške za timski rad. Na primer, naredni testovi u biblioteci Google test (`googletest`) se mogu upotrebiti za automatsko testiranje funkcije koja izračunava stepen.

```
double stepen(double x, int n);

TEST(StepenTest, PozitivanEksponent) {
   EXPECT_DOUBLE_EQ(stepen(2.0, 3), 8.0);
}
TEST(StepenTest, NulaEksponent) {
   EXPECT_DOUBLE_EQ(stepen(2.0, 0), 1.0);
}
TEST(StepenTest, NegativanEksponent) {
   EXPECT_DOUBLE_EQ(stepen(2.0, -1), 0.5);
}
```

Testovi se pokreću automatski i nakon pokretanja prijavljuje se sledeći izveštaj, koji ukazuje na to da funkcija nije korektno implementirana za negativne izložioce.

```
[==========] Running 3 tests from 1 test suite.

[ RUN      ] StepenTest.PozitivanEksponent
[       OK ] StepenTest.PozitivanEksponent

[ RUN      ] StepenTest.NulaEksponent
[       OK ] StepenTest.NulaEksponent

[ RUN      ] StepenTest.NegativanEksponent
stepen_test.cpp:25: Failure
Expected equality of these values:
  stepen(2.0, -1)  Which is: 1
```

```
  0.5
[  FAILED  ] StepenTest.NegativanEksponent

[==========] 4 tests ran.
[  PASSED  ] 3 tests.
[  FAILED  ] 1 test.
```

I pored automatizacije, obično ostaje i važan prostor za ručno testiranje, posebno za aspekte korišćenja programa koje nije lako automatizovati (poput subjektivnog osećaja korisnika).

1.3.3 Debagovanje

Testiranje je proces provere da li program radi ispravno, odnosno sistematičan pokušaj da se pronađu greške. Debagovanje se koristi onda kada znamo (ili sumnjamo) da greška postoji i želimo da je pronađemo i otklonimo.

Debager je alat koji omogućava praćenje izvršavanja programa korak po korak. Umesto da se program izvrši "odjednom", debager omogućava programeru da zaustavi izvršavanje na određenim mestima, prati tok programa i ispituje stanje podataka (vrednosti promenljivih, stanje memorije, stanje na pozivnom, programskom steku, itd).

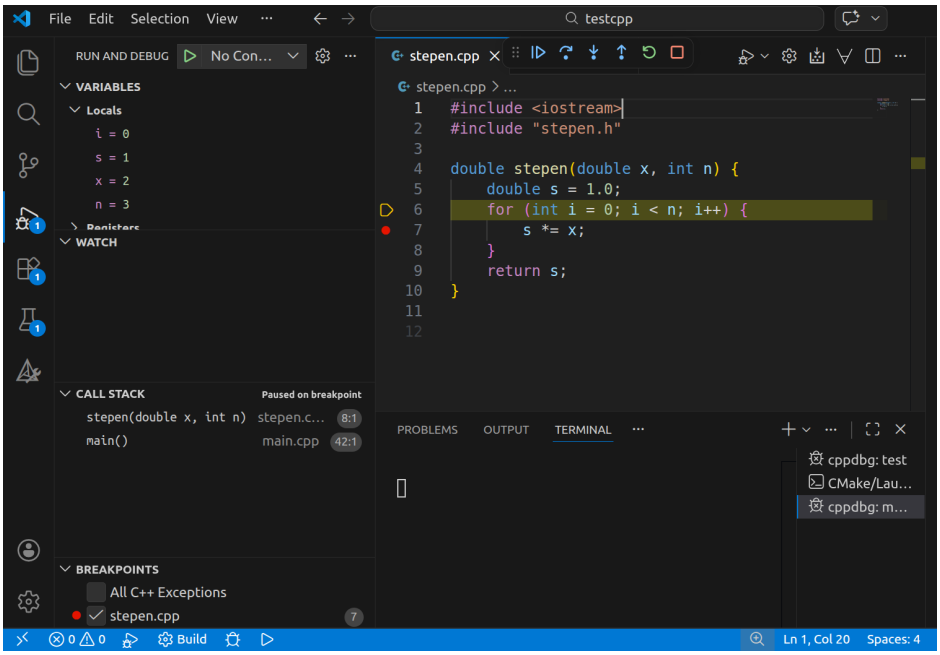
Osnovni pojmovi u radu sa debagerom su:

- *Breakpoint* (tačka prekida) – mesto u kodu na kome se izvršavanje programa automatski zaustavlja, kako bi programer mogao da ispita stanje programa.
- *Step over* – izvršava narednu liniju koda, ali bez ulaska u funkcije koje se u njoj pozivaju.
- *Step into* – ulazi u funkciju koja se poziva u tekućoj liniji, kako bi se pratilo njeno izvršavanje korak po korak.
- *Step out* – izvršava ostatak tekuće funkcije i vraća se u pozivajući kontekst.
- *Locals* – prikazuje sve lokalne promenljive u trenutnoj funkciji i njihove vrednosti.
- *Watch* – omogućava praćenje vrednosti izabranih izraza ili promenljivih tokom izvršavanja.
- *Call Stack* (pozivni stek) – prikazuje niz aktivnih poziva funkcija, što pomaže da se razume kako je program došao do trenutne tačke izvršavanja.

Posmatranjem ovih informacija programer može da uoči gde dolazi do neočekivanog ponašanja i lakše pronađe uzrok greške.

Da bi se program debagovao, potrebno je kompajlirati ga u debag režimu (npr. u C++-u pomoću opcije -g). Nakon toga se program pokreće u debageru (na primer, gdb ili njegov grafički interfejs kdbg).

Savremena razvojna okruženja, kao što su Visual Studio i Visual Studio Code (Slika 1.1), imaju ugrađene debagere koji omogućavaju jednostavno postavljanje tačaka prekida, praćenje promenljivih i kontrolu izvršavanja programa, što debagovanje čini znatno jednostavnijim.



Slika 1.1: Ilustracija rada debagera u okruženju VS Code

1.4 Statičko ispitivanje ispravnosti programa

Statičko ispitivanje ispravnosti programa, (tj. statička verifikacija) podrazumeva analizu izvornog koda programa (bez njegovog izvršavanja). Takve analize mogu da sprovode i ljudi samostalno ("na papiru"), ali ih obično sprovode ljudi uz pomoć namenskih alata ili namenski programi, potpuno automatski.

Proces verifikacije može biti neformalan i formalan. Neformalnu verifikaciju sprovode programeri, analizom koda, posebno nekih kritičnih delova. Formalna verifikacija zasniva se na precizno definisanoj semantici programskog jezika i strogom logičkom okviru u kojem se dokazi korektnosti izvode. Formalno dokazana ispravnost vodi do najvišeg mogućeg nivoa pouzdanosti programa. Formalno dokazivanje ispravnosti je obično veoma zahtevno, te se ono retko primenjuje, obično samo za bezbednosno kritične programe (kao što je, na primer, program za upravljanje metroom).

Ne postoji algoritam koji za proizvoljan algoritam može da dokaže da zadovoljava svoju specifikaciju (baš kao što ne postoji ni algoritam koji može da ispita da li se proizvoljni program zaustavlja). Najveći problemi u tome su zaustavljanje i analiza petlji. Ono što jeste moguće je postojanje algoritama koji u nekim slučajevima i sa nekim pojednostavljivanjima mogu da dokažu ispravnost zadatog programa.

1.4.1 Neformalno ispitivanje ispravnosti algoritama

Potpuno precizna, formalna verifikacija programa zahteva poznavanje svih detalja semantike programskih jezika. Jedan od izazova u tome predstavlja činjenica da se semantika uobičajenih tipova podataka i operacija u programima razlikuje od uobičajene semantike matematičkih operacija nad celim i realnim brojevima (iako velike sličnosti postoje). Na primer, iako tip `int` podseća na skup celih brojeva, a operacija sabiranja dva podatka tipa `int` na sabiranje dva cela broja, razlike su evidentne — domen tipa `int` je konačan (fiksne "širine"), a operacija se vrši "po modulu" tj. u nekim slučajevima dolazi do prekoračenja. Mnoga pravila koja važe za cele brojeve ne važe za podatke tipa `int`. Na primer, $x \geq 0 \wedge y \geq 0 \Rightarrow x + y \geq 0$ važi ako su x i y celi brojevi, ali ne važi ako su podaci tipa `int`. U nekim slučajevima, prilikom verifikacije ovakve razlike se apstrahuju i zanemaruju. Time se, naravno, gubi potpuno precizna karakterizacija ponašanja programa i "dokazi" korektnosti prestaju da budu dokazi korektnosti u opštem slučaju. Ipak, ovim se značajno olakšava proces verifikacije i u većini slučajeva ovakve aproksimacije ipak mogu značajno da podignu stepen pouzdanosti programa.

U nastavku teksta, ovakve aproksimacije će biti često vršene. Dakle, u narednom tekstu će fokus biti stavljen na ispitivanje korektnosti algoritama, a ne njihove konkretne implementacije koja zavisi od tehničkih detalja programskog jezika u kom su oni implementirani (na primer, pitanja reprezentacije podataka i prekoračenja).

1.4.2 *Induktivno-rekurzivna konstrukcija*

Ključna ideja u konstrukciji algoritama je to da je konstrukcija algoritama veoma tesno povezana sa dokazivanjem teorema *matematičkom indukcijom*. Cilj ovog poglavlja (ali i ostatka ove knjige) je da pokažemo da je matematička indukcija osnovni alat za analizu svih vrsta programa (i iterativnih i rekurzivni), ali i više od toga – ona je osnovni alat za konstrukciju algoritama. U tom svetlu se precizna analiza korektnosti nekog algoritma ne vrši nakon što je algoritam konstruisan, već je ona prisutna sve vreme tokom same konstrukcije algoritma. Dokaz teoreme kokreknosti algoritma i sam algoritam su neraskidivo povezani.

Matematička indukcija, u svom osnovnom obliku, je sledeći način dokazivanja osobina prirodnih brojeva. Neka je P proizvoljno svojstvo koje se može formulisati za prirodne brojeve. Tada važi

$$(P(0) \wedge (\forall n)(P(n) \Rightarrow P(n+1))) \Rightarrow (\forall n)(P(n))$$

Dakle, da bismo dokazali da svaki prirodan broj ima neko svojstvo P (tj. da bismo dokazali $(\forall n)(P(n))$), dovoljno je da dokažemo da nula ima to svojstvo (tj. $P(0)$) i da dokažemo da čim neki broj ima to svojstvo, ima ga i njegov sledbenik (tj. da dokažemo $(\forall n)(P(n) \Rightarrow P(n+1))$). Prvo tvrđenje se naziva *baza indukcije*, a drugo *induktivni korak*. Prinicip matematičke indukcije je prilično jasan – na osnovu baze znamo da 0 ima svojstvo P, na osnovu koraka da njen sledbenik tj. 1 ima svojstvo P, na osnovu koraka da njegov sledbenik tj. 2 ima svojstvo P itd. Intuitivno nam je jasno da na ovaj način možemo stići do bilo kog prirodnog broja, koji sigurno mora imati svojstvo P. Baza se može formulisati i za veće vrednosti od nule, ali onda možemo da tvrdimo samo da elementi koji su veći ili jednaki od baze imaju svojstvo P.

Osnovni pristup konstrukcije algoritama je tzv. *induktivni* tj. *rekurzivni* pristup. U ovom pristupu cilj je od rešenja problema zadatog oblika ali manje dimenzije dobiti rešenje tog problema veće dimenzije (u naprednijim oblicima je moguće i da se rešava veći broj problema manje dimenzije). Pritom, za početne dimenzije problema (koje zovemo *bazni slučajevi*) rešenje mora da se izračuna neposredno, bez daljeg svođenja na probleme manje dimenzije. Ako se prilikom svođenja dimenzija problema uvek smanjuje, konstruisani algoritmi će se uvek zaustavljati (jer dimenzija mora da bude nenegativna).

- Implementacija algoritma može biti takva da promenljive unutar petlje iterativno ažuriraju svoje vrednosti krenuvši od vrednosti za bazne slučajeve, pa

do krajnjih vrednosti koje predstavljaju rešenja zadatog problema. Pošto je ovo prilično slično principu matematičke indukcije, kažemo da je algoritam definisan *induktivno*.

- Implementacija može biti takva da funkcija koja rešava polazni problem samu sebe poziva da bi rešila problem istog oblika, ali manje dimenzije (osim u baznim slučajevima, koji se neposredno rešavaju) i tada kažemo da je algoritam definisan *rekurzivno*.

Induktivna konstrukcija leži u osnovni praktično svih iterativnih algoritama koje smo do sada razmatrali. Na primer, prikazani algoritam stepenovanja počiva na tome da znamo da izračunamo nulti stepen broja (to je 1) i da ako znamo da izračunamo x^k, tada umemo da izračunamo i x^{k+1} (to radimo tako što taj poznati stepen x^k pomnožimo sa x).

```
double stepen(double x, int n) {
    double s = 1.0;
    for (int i = 0; i < n; i++)
        s = s * x;
    return s;
}
```

Dakle, i u ovom algoritmu imamo induktivnu bazu (koja odgovara inicijalizaciji promenljive pre ulaska u petlju) i induktivni korak (koji odgovara telu petlje, u kom se ažurira vrednost rezultujuće promenljive, u ovom slučaju stepena). Baza može odgovarati i slučaju prvog (a ne obavezno nultog) stepena, ali tada ne možemo da garantujemo da će algoritam ispravno izračunavati nulti stepen.

Rekurzivna implementacija izračunavanja stepena može biti sledeća (u njoj se prilikom rešavanja problema dimenzije $n > 0$ eksplicitno zahteva rešavanje problema dimenzije $n - 1$).

```
double stepen(double x, int n) {
    if (n == 0)
       return 1.0;
    else
```

```
        return stepen(x, n-1) * x;
}
```

Definisanje algoritama induktivno-rekurzivom konstrukcijom je u veoma tesnoj vezi sa dokazivanjem njihove korektnosti. Iako postoje formalni okviri za dokazivanje korektnosti imperativnih programa (pre svega *Horova logika*), u ovom poglavlju ćemo se baviti isključivo neformalnim dokazima i veza između logike u kojoj vršimo dokazivanje i (imperativnog) programskog jezika u kojem se program izražava biće prilično neformalna.

Kao što smo već nagovestili, prilikom dokazivanja korektnosti programa obično ćemo ignorisati ograničenja zapisa brojeva u računaru i podrazumevaćemo da je opseg brojeva neograničen i da se realni brojevi zapisuju sa maksimalnom preciznošću. Dakle, obično ćemo ignorisati greške koje mogu nastati usled prekoračenja ili potkoračenja vrednosti tokom izvođenja aritmetičkih operacija. Ipak, prekoračenja ili potkoračenja mogu biti uzrok bitnih grešaka u nekim programima i tada u ispitivanju ispravnosti koristimo bogatije modelovanje zapisa brojeva.

1.4.3 *Dokazivanje ispravnosti rekurzivnih funkcija*

U principu, dokazivanje korektnosti najjednostavnije je za programe koji su sačinjeni od rekurzivno definisanih funkcija koje ne koriste elemente imperativnog programiranja (kao što su dodele vrednosti promenljivama, petlje i slično). Takve programe zovemo *funkcionalni programi*. Glavni razlog za jednostavnije dokazivanje ispravnosti ovakvih programa je to što se njihove funkcije mogu jednostavno modelovati matematičkim funkcijama (koje za iste argumente uvek daju iste vrednosti). Naime, u funkcionalnim programima funkcije za iste ulazne argumente takođe uvek vraćaju istu vrednost. To je tako zbog pretpostavke da nema eksplicitne dodele vrednosti promenljivama, kao i da kontekst poziva i globalne promenljive ne utiču na izvršavanje funkcija. Dokazivanje korektnosti rekurzivnih funkcija teče nekim oblikom matematičke indukcije.

Problem: Definisati funkciju koja za dati prirodan broj $n \geq 0$ i dat realan broj x (različit od 0 ako je $n = 0$) izračunava x^n.

Kao što smo videli, algoritam se veoma jednostavno konstruiše induktivno-rekurzivnom konstrukcijom. Dimenzija problema u ovom primeru je izložilac n.

Rekurzivna implementacija, kao što smo videli je sledeća.

```
double stepen(double x, int n) {
    if (n == 0)
       return 1.0;
    else
       return stepen(x, n-1) * x;
}
```

Korektnost se može formulisati u obliku sledeće teoreme (koju dokazujemo zanemarujući pitanja reprezentacije brojeva u računaru).

Teorema 1.4.1. *Za svaki prirodan broj $n \geq 0$ i realan broj x (različit od 0, ako je $n = 0$), rekurzivna funkcija* `stepen` *vraća vrednost x^n tj. važi stepen$(x, n) = x^n$.*

Dokaz. Teoremu možemo dokazati indukcijom.

- Bazu indukcije predstavlja slučaj $n = 0$, tj. poziv `stepen(x, 0)`. Na osnovu definicije funkcije `stepen` rezultat je 1, što je ujedno i vrednost x^0 (jer je po specifikaciji tada $x \neq 0$).
- Induktivna hipoteza je tvrđenje: poziv `stepen(x, n-1)` vraća vrednost x^{n-1}. Iz te pretpostavke potrebno je da dokažemo da poziv `stepen(x, n)` vraća vrednost x^n. Na osnovu definicije funkcije `stepen`, poziv `stepen(x, n)` će vratiti proizvod `stepen(x, n-1) * x`. Na osnovu induktivne hipoteze znamo da poziv `stepen(x, n-1)` vraća x^{n-1}, pa je stoga rezultat $x^{n-1} \cdot x$, što je upravo x^n. □

Primećujemo ogromnu sličnost između rekurzivne konstrukcije algoritma i induktivnog dokaza njegove korektnosti. Stoga slobodno možemo da kažemo da su rekurzija i indukcija "dve strane iste medalje" (indukciju koristimo kao tehniku dokazivanja, a rekurziju kao tehniku definisanja funkcija tj. konstrukcije algoritama).

Primetimo i da ovaj oblik korišćenja matematičke indukcije nije onaj uobičejeni, jer se ne koristi indukcija neposredno po prirodnim brojevima, već se koristi indukcija po strukturi rekurzivne funkcije u kojoj se, iz pretpostavke da svaki rekurzivni poziv vraća korektan rezultat, dokazuje da funkcija vraća korektan rezultat. Takva forma matematičke indukcije se može dokazati na osnovu uobičajene indukcije po broju rekurzivnih poziva, pod pretpostavkom da se dokaže da se rekurzivna funkcija uvek zaustavlja.

Isti problem stepenovanja se može rešiti i na sledeći način, efikasnije. Ovaj algoritam poznat je pod nazivom **brzo stepenovanje**. Na primer, umesto da 2^{10} izračunavamo kao $2^9 \cdot 2$, brže ga možemo izračunati kao $(2^2)^5$. Rekurzivna definicija tada izgleda ovako.

$$x^n = \begin{cases} 1 & \text{ako } n = 0, \\ (x^2)^{n/2} & \text{ako je } n \text{ paran}, \\ x \cdot x^{n-1} & \text{ako je } n \text{ neparan}. \end{cases}$$

```
float stepen_brzo(float x, unsigned n)
{
  if (n == 0)
    return 1.0f;
  else if (n % 2 == 0)
    return stepen_brzo(x * x, n / 2);
  else
    return x * stepen_brzo(x, n - 1);
}
```

Vrednost 2^{10} se ovom funkcijom izračunava na sledeći način (označimo, kratkoće radi, funkciju `stepen_brzo` sa s):

$$\begin{aligned} s(2,10) &= s(4,5) = 4 \cdot s(4,4) = 4 \cdot s(16,2) = 4 \cdot s(256,1) = \\ &= 4 \cdot 256 \cdot s(256,0) = 4 \cdot 256 \cdot 1 = 1024. \end{aligned}$$

Teorema 1.4.2. *Za svaki prirodan broj $n \geq 0$ i realan broj x (različit od 0, ako je $n = 0$), rekurzivna funkcija* `stepen_brzo` *vraća vrednost x^n tj. važi stepen_brzo$(x, n) = x^n$.*

Dokaz. Dokažimo ispravnost navedene funkcije. Kako je u ovom slučaju funkcija definisana opštom rekurzijom, dokaz će pratiti sledeću shemu: "Da bi se pokazalo da vrednost *stepen_brzo*(x, n) zadovoljava neko svojstvo, može se pretpostaviti da za $n > 0$ i n parno vrednost *stepen_brzo*$(x \cdot x, n/2)$ zadovoljava to svojstvo, kao i da za $n > 0$ i n neparno vrednost *stepen_brzo*$(x, n-1)$ zadovoljava to svojstvo,

i onda tvrđenje treba dokazati pod tim pretpostavkama". Slične sheme indukcije se mogu dokazati i za druge funkcije definisane opštom rekurzijom i, u principu, one dozvoljavaju da se prilikom dokazivanja korektnosti dodatno pretpostavi da svaki rekurzivni poziv vraća korektan rezultat.[1]

Pređimo na sâm dokaz činjenice da je *stepen_brzo*$(x, n) = x^n$ (za nenegativnu vrednost n).

- **Slučaj** $n = 0$**:** Tada je *stepen_brzo*(x, n) = *stepen_brzo*$(x, 0) = 1 = x^0$.
- **Slučaj** $n > 0$**:** Tada je n ili paran ili neparan.
  - **Slučaj** n **je paran:** Tada je *stepen_brzo*(x, n) = *stepen_brzo*$(x \cdot x, n/2)$. Na osnovu prvog dela induktivne pretpostavke, *stepen_brzo*$(x \cdot x, n/2) = (x \cdot x)^{n/2}$. Dalje, elementarnim aritmetičkim transformacijama sledi da je *stepen_brzo*$(x, n) = (x \cdot x)^{n/2} = (x^2)^{n/2} = x^n$.
  - **Slučaj** n **je neparan:** Tada je *stepen_brzo*$(x, n) = x \cdot$*stepen_brzo*$(x, n-1)$. Na osnovu drugog dela induktivne pretpostavke, *stepen_brzo*$(x, n-1) = x^{n-1}$. Dalje, elementarnim aritmetičkim transformacijama sledi da je *stepen_brzo*$(x, n) = x \cdot x^{n-1} = x^n$. □

1.4.4 *Dokazivanje ispravnosti iterativnih algoritama i invarijante petlji*

U slučaju imperativnih programa (programa koji sadrže naredbu dodele i petlje), aparat koji se koristi za dokazivanje korektnosti mora biti znatno složeniji. Semantiku imperativnih konstrukata znatno je teže opisati u odnosu na (jednostavnu jednakosnu) semantiku čisto funkcionalnih programa. Sve vreme dokazivanja mora se imati u vidu tekući kontekst tj. *stanje programa* koje obuhvata tekuće vrednosti svih promenljivih koje se javljaju u programu. Program implicitno predstavlja relaciju prelaska između stanja i dokazivanje korektnosti zahteva dokazivanje da će program na kraju stići u neko stanje u kojem su zadovoljeni uslovi zadati specifikacijom. Dodatnu otežavajuću okolnost čine propratni efekti dodela, kao i činjenica

[1] Ova teorema se može dokazati i principom jake indukcije za prirodne brojeve, u kom se iz induktivne pretpostavke da svi brojevi manji ili jednaki k imaju neko svojstvo dokazuje da i broj $k+1$ ima to svojstvo. Međutim, princip indukcije koji prati definiciju rekurzivne funkcije je opštiji, jer se može primeniti i na funkcije čiji argumenti nisu prirodni brojevi ili jesu prirodni brojevi, ali se ne smanjuju monotono tokom rekurzivnih poziva. Stoga ćemo u svim dokazima koristiti ovaj opštiji princip, koji važi za sve rekurzivne funkcije koje se zaustavljaju za sve vrednosti svojih argumenata.

da pozivi funkcija mogu da vrate različite vrednosti za iste prosleđene ulazne parametre (u zavisnosti od globalnog konteksta tj. stanja u kojem se poziv izvršio). Zbog toga je dokaz korektnosti složenog programa teže razložiti na elementarne dokaze korektnosti pojedinih funkcija.

Kao najkompleksniji programski konstrukt, petlje predstavljaju jedan od najvećih izazova u verifikaciji. Umesto pojedinačnog razmatranja svakog stanja kroz koje se prolazi prilikom izvršavanja petlje, obično se formulišu uslovi (*invarijante petlji*) koji precizno karakterišu taj skup stanja. *Invarijanta petlje* je logička formula koja uključuje vrednosti promenljivih koje se javljaju u nekoj petlji i koja važi pri svakom ispitivanju uslova petlje (tj. neposredno pre, nakon svakog izvršavanja tela petlje, kao i neposredno nakon izvršavanja petlje). Invarijante suštinski opisuju značenje svih promenljivih unutar petlje. Ilustrujmo pojam invarijante na primeru iterativne funkcije za izračunavanje stepena.

```
double stepen(double x, int n) {
   double s = 1.0;
   for (int i = 0; i < n; i++)
       s = s * x;
   return s;
}

int main() {
    cout << stepen(2.0, 10) << endl;
}
```

Suština dokaza korektnosti leži u sledećem razmatranju. Nakon inicijalizacije promenljiva s sadrži vrednost $1 = x^0 = x^i$. U svakom koraku petlje promenljiva s se množi sa x, a vrednost i uvećava za 1, pa i dalje važi $s = x^i$. Dakle, uslov $s = x^i$ važi i pre, i tokom i nakon izvršavanja petlje. Pošto je nakon izvršavanja petlje $i = n$, na kraju promenljiva s sadrži vrednost x^n, pa, pošto je ta vrednost povratna vrednost funkcije, funkcija ispravno izračunava x^n.

Ovu skicu dokaza možemo dodatno precizirati i dokazati je matematičkom indukcijom.

Teorema 1.4.3. *U svakom koraku petlje (i na njenom početku, neposredno nakon provere uslova, ali i na njenom kraju, neposredno nakon izvršavanja tela), kao i nakon*

izvršavanja cele petlje važi da je $0 \leq i \leq n$ *i da je* $s = x^i$ *(gde je* i *tekuća vrednost promenljive* `i`*, a* s *tekuća vrednost promenljive* `s`*).*

Dokaz. Ovo tvrđenje možemo dokazati indukcijom i to po broju izvršavanja tela petlje (obeležimo taj broj sa k). Napomenimo samo da ćemo petlju `for` smatrati samo skraćenicom za petlju `while`, tako da ćemo inicijalizaciju petlje smatrati za kôd koji se izvršava pre petlje, dok ćemo korak petlje smatrati kao poslednju naredbu tela petlje.

```
double s = 1.0;
int i = 0;
while (i < n) {
    s = s * x;
    i++;
}
```

Obeležimo sa $s_0, s_1, \ldots, s_k, \ldots$ vrednosti promenljive `s`, a sa $i_0, i_1, \ldots, i_k, \ldots$ vrednost promenljive `i` nakon $0, 1, \ldots, k, \ldots$ izvršavanja tela petlje. Pošto promenljiva `n` ne menja svoju vrednost, njenu vrednost ćemo označiti sa n.

- Bazu indukcije čini slučaj $k = 0$ tj. slučaj kada se telo petlje nije još izvršilo. Pre ulaska u petlju promenljiva `i` se inicijalizuje na 0 (važi $i_0 = 0$). Pošto je $n \geq 0$, važi $0 \leq i = i_0 = 0 \leq n$. Promenljiva `s` se inicijalizuje na vrednost 1 (važi $s_0 = 1$), pa je zaista $s_0 = x^{i_0}$. Dakle, uslovi teoreme su zadovoljeni pre prvog izvršavanja tela petlje.

- Pretpostavimo sada kao induktivnu hipotezu da tvrđenje važi nakon k izvršavanja tela petlje. Dakle, pretpostavljamo da uslovi važe za vrednosti s_k i i_k (sa s_k i i_k obeležavamo vrednosti promenljivih nakon k izvršavanja tela petlje) tj. da je $0 \leq i_k \leq n$ i da je $s_k = x^{i_k}$. Ako je uslov petlje ispunjen, to će ujedno biti i vrednosti promenljivih na početku tela petlje, pre njenog $k+1$-vog izvršavanja.

  Iz induktivne hipoteze i pretpostavke da je uslov petlje `i < n` ispunjen (tj. da je $i_k < n$) dokažimo da nakon $k+1$ izvršavanja tela petlje uslovi teoreme važe i za vrednosti s_{k+1} i s_{k+1} (sa s_{k+1} i i_{k+1} obeležavamo vrednosti promenljivih nakon $k+1$ izvršavanja tela petlje). Vrednosti s_{k+1} i i_{k+1} se mogu lako odrediti na osnovu vrednosti s_k i i_k, analizom jednog izvršavanja tela petlje.

Na osnovu definicije funkcije važi da je $s_{k+1} = s_k \cdot x$ i $i_{k+1} = i_k + 1$. Zato, pošto je $0 \leq i_k < n$, važi i da je $0 \leq i_{k+1} \leq n$, pa je uslov koji se odnosi na raspon vrednosti promenljive `i` očuvan. Na osnovu induktivne hipoteze znamo da je $s_k = x^{i_k}$. Zato je $s_{k+1} = x^{i_k} \cdot x = x^{i_k+1} = x^{i_{k+1}}$.

Neka su i i s vrednosti promenljivih `i` i `s` nakon izvršavanja petlje. Na osnovu dokazanog tvrđenja znamo da uslovi navedeni u njemu važe i nakon završetka petlje. Kada se petlja završi, važi da je $i = n$ (jer na osnovu prvog uslova znamo da je $0 \leq i \leq n$, a uslov petlje `i < n` nije ispunjen). Na osnovu drugog uslova znamo da je $s = x^i = x^n$.

Zaustavljanje se dokazuje jednostavno tako što se dokaže da se u svakom koraku petlje nenegativna vrednost $n - i$ smanjuje za po 1, dok ne postane 0. □

Ako razmotrimo strukturu prethodnog razmatranja, možemo ustanoviti da smo identifikovali logičke uslove koji su ispunjeni neposredno pre i neposredno nakon svakog izvršavanja tela petlje. Takvi uslovi se nazivaju *invarijante petlje*. Da bismo dokazali da je neki uslov invarijanta petlje, dovoljno je da dokažemo:

(1) da taj uslov važi pre prvog ulaska u petlju i

(2) da iz pretpostavke da taj uslov važi pre nekog izvršavanja tela petlje i da je uslov petlje ispunjen dokažemo da taj uslov važi i nakon izvršavanja tela petlje.

Te dve činjenice nam, na osnovu induktivnog argumenta, garantuju da će uslov biti ispunjen pre i posle svake iteracije petlje, kao i nakon izvršavanja cele petlje (ako se ona ikada zaustavi), tj. da će taj uslov biti invarijanta petlje (taj dokaz se može sprovesti klasičnom matematičkom indukcijom na osnovu broja izvršavanja tela petlje). Primetimo da prvi korak odgovara dokazivanju baze indukcije, a drugi dokazivanju induktivnog koraka.

Svaka petlja ima puno invarijanti, pri čemu su neki uslovi "preslabi" a neki "prejaki" tj. ne objašnjavaju ponašanje programa. Na primer, bilo koja valjana formula (na primer, $x \cdot 0 = 0$ ili $(x \geq y) \vee (y \geq x)$) je uvek invarijanta petlje. Od interesa su nam samo one invarijante koje u kombinaciji sa uslovom prekida petlje (pod pretpostavkom da petlja nije prekinuta naredbom `break`) impliciraju uslov koji nam je potreban nakon petlje. Ako je petlja jedina u nekom algoritmu, obično je to onda uslov korektnosti samog algoritma. Dakle, nakon dokaza leme koja čini osnovu dokaza da je neki uslov invarijanta petlje, potrebno je da dokažemo i

(3) da iz toga da invarijanta važi nakon završetka petlje i da uslov petlje nije ispunjen sledi korektnost algoritma.

Dakle, opšta struktura analize programa korišćenjem invarijanti se može opisati na sledeći način.

```
<incijalizacija>
// ovde vazi <invarijanta>
while (<uslov>)
   // ovde vaze i <uslov> i <invarijanta>
   <telo>
   // ovde vazi <invarijanta>
// ovde ne vazi <uslov>, a vazi <invarijanta>
```

Prikažimo sada kraću verziju prethodnog dokaza korektnosti funkcije stepenovanja, u kom ćemo koristiti pojam invarijante i nećemo se direktno pozivati na matematičku indukciju.

Lema 1.4.1. *Za svaki prirodni broj $n \geq 0$ i realni broj x (različit od 0 kada je $n = 0$), u svakom koraku petlje važi da je $0 \leq i \leq n$ i $s = x^i$ (ovo je jedna invarijanta petlje).*

Dokaz. Dokažimo da je uslov invarijanta.

- Pokažimo da invarijanta važi pre ulaska u petlju. Pre ulaska u petlju promenljive imaju vrednosti $s = 1$ i $i = 0$, te invarijanta, trivijalno, važi.

- Pokažimo da invarijanta ostaje održana nakon svakog izvršavanja tela petlje. Obeležimo sa s' i s'' i sa i' i i'' vrednosti promenljivih `s` i `i` pre i posle izvršavanja tela i koraka petlje. Pošto se promenljive `x` i `n` ne menjaju, njihove vrednosti obeležimo sa x i n.

  Pretpostavimo da invarijanta važi pri ulasku u petlju tj. da važi $s' = x^{i'}$ i $0 \leq i' \leq n$. Pošto je uslov petlje ispunjen, važi i $i' < n$.

  Nakon izvršavanja tela petlje, promenljive imaju vrednosti $s'' = s' \cdot x$ i $i'' = i' + 1$. Potrebno je pokazati da ove nove vrednosti zadovoljavaju invarijantu, tj. da važi $s'' = x^{i''}$ i $0 \leq i'' \leq n$. Zaista, $s' \cdot x = x^{i'+1}$ što je tačno

na osnovu pretpostavke. Takođe, pošto važi da je $0 \leq i' < n$, važi i da je $0 \leq i'' \leq n$.

Dakle, ako su s, i, n i x tekuće vrednosti promenljivih, tada su $s = x^i$ i $0 \leq i \leq n$ invarijante petlje. □

Pokažimo da dokazana invarijanta obezbeđuje korektnost.

Teorema 1.4.4. *Na kraju izvršavanja algoritma važi da je* $s = x^n$.

Dokaz. Kada se izađe iz petlje, važi $i = n$. Zaista, na osnovu leme znamo da važi invarijanta $0 \leq i \leq n$, pa pošto ne važi uslov petlje $i < n$, po izlasku iz petlje mora da važi da je $i = n$. Kombinovanjem sa invarijantom $x = s^i$, dobija se da tada važi $s = x^n$, što je i trebalo dokazati. □

Iterativno brzo stepenovanje

Dokažimo sada korektnost naredne iterativne implementacije algoritma brzog stepenovanja. Ovakvu implementaciju dobijamo prateći promenu promenljivih tokom izvršavanja rekurzivnih poziva. Novouvedena promenljiva s akumulira konačnu vrednost stepena.

```
double stepen_brzo(double x, int n) {
    double s = 1.0;
    while (n > 0) {
        if (n % 2 == 1) {
            s *= x;
            n--;
        } else {
            x = x * x;
            n /= 2;
        }
    }
    return s;
}
```

Razmotrimo kako se menjaju vrednosti promenljivih dok se izračunava stepen 2^{10}.

x	n	s
2	10	1
4	5	1
4	4	4
16	2	4
256	1	4
256	0	1024

Ili malo opštije, ako je inicijalna vrednost promenljive x jednaka nekoj vrednosti c.

x	n	s
c	10	1
c^2	5	1
c^2	4	c^2
c^4	2	c^2
c^8	1	c^2
c^8	0	c^{10}

Lema 1.4.2. *Neka je $n \geq 0$ i $x \neq 0$ ako je $n = 0$. Neka je n_0 početna vrednost promenljive n, a x_0 promenljive x. Tokom izvršavanja petlje važi invarijanta da je $n \geq 0$ i $s \cdot x^n = x_0^{n_0}$.*

Dokaz. Razmotrimo inicijalizaciju i održanje invarijante.

- Nakon inicijalizacije važi da je $x = x_0$, $n = n_0$ i $s = 1$, pa invarijanta trivijalno važi.
- Pretpostavimo da invarijanta važi pri ulasku u telo petlje tj. da je $s \cdot x^n = x_0^{n_0}$. Tada važi i uslov petlje $n > 0$.
  - Ako je n neparan broj, tada je $n' = n - 1$, $s' = s \cdot x$ i $x' = x$ pa je $s' \cdot {x'}^{n'} = s \cdot x \cdot x^{n-1} = s \cdot x^n = x_0^{n_0}$. Pošto je $n > 0$, važi da je $n' \geq 0$.
  - Ako je n paran broj, tada je $n' = n/2$, $x' = x^2$ i $s' = s$. Tada je $s' \cdot {x'}^{n'} = s \cdot (x^2)^{n/2} = s \cdot x^n = x_0^{n_0}$. Pošto je $n > 0$, važi da je $n' \geq 0$. □

Iz ove invarijante sledi korektnost.

Teorema 1.4.5. *Neka je* $n \geq 0$ *i* $x \neq 0$ *ako je* $n = 0$. *Tada je stepen_brzo*$(x, n) = x^n$.

Dokaz. Kada se petlja završi važi invarijanta $n \geq 0$, $s \cdot x^n = x_0^{n_0}$, a uslov petlje $n > 0$ nije ispunjen. Zato je $n = 0$, pa važi $s \cdot x^0 = x_0^{n_0}$ tj. $s = x_0^{n_0}$. Dakle, povratna vrednost funkcije sadržana u promenljivoj s je upravo jednaka stepenu x^n, za zadate početne vrednosti promenljivih x i n. □

Za vežbu pokažite i da je naredna, malo jednostavnija implementacija takođe korektna.

```
double stepen_brzo(double x, int n) {
    double s = 1.0;
    while (n > 0) {
        if (n % 2 == 1)
            s *= x;
        x *= x;
        n /= 2;
    }
    return s;
}
```

Još nekoliko primera ovako preciznih dokaza korektnosti dato je u dodatku 1.A.2. U nastavku ovog poglavlja videćemo još nekoliko primera primene tehnike invarijante petlje. Mora se priznati da kada se tehnika koristi potpuno formalno, da bi se dokazala korektnost već napisanog programskog koda, to ne deluje naročito inspirišuće (pogotovo, ako su programi jednostavni i ako je jednostavno intuitivno razumeti razloge njihove korektnosti). Retko kada se u praktičnom programiranju korektnost zaista dokazuje potpuno formalno (osim u slučaju softvera koji može da ugrozi veliki broj života, poput, na primer, softvera koji upravlja metro-sistemom u Parizu, koji jeste u potpunosti formalno verifikovan). Međutim, argumente i invarijante na kojima korektnost počiva programer često "provrti po glavi". Videćemo i da se tehnika invarijanti može upotrebiti i pre nego što je program napisan u cilju izvođenja programskog koda iz specifikacije. Jasne invarijante često jednoznačno ukazuju na to kako programski kôd treba da izgleda i na taj način pomažu u procesu programiranja.

Zadatak: Trobojka

Napisati program koji učitava niz celih brojeva a zatim ga transformiše tako da elementi budu podeljeni u tri dela u zavisnosti od zadatih vrednosti A i B. U prvom delu su elementi manji od zadate vrednosti A (vrednosti iz intervala $[-\infty, A)$), u drugom elementi veći ili jednaki zadatoj vrednosti A i manji ili jednaki zadatoj vrednosti B (vrednosti iz intervala $[A, B]$), a u trećem elementi veći od zadate vrednosti B (vrednosti iz intervala $(B, +\infty)$). Nije bitno u kom se redosledu nalaze elementi unutar delova. Učitati elemente u niz, a zatim reorganizovati redosled elemenata u tom nizu (ne koristiti pomoćne nizove).

Opis ulaza

U jednoj liniji standardnog ulaza nalazi se broj elemenata niza, N, a zatim se, u narednoj liniji nalaze elementi niza razdvojeni razmacima. U poslednje dve linije se nalaze celi brojevi A i B odvojeni prazninom, i pri tome je $A < B$.

Opis izlaza

Ispisati elemente rezultujućeg niza na standardni izlaz (moguće je ispisati elemente svake od tri grupe u posebnom redu, razdvojene razmacima, a moguće je ispisati i ceo niz u jednom redu ili u više redova).

Primer

Ulaz	*Izlaz*
`10`	`1 2`
`1 3 5 4 8 5 7 2 3 6`	`5 4 3 5 3`
`3`	`7 6 8`
`5`	

Rešenje

Dva prolaza kroz niz

Jedan pristup je da se do rešenja dođe u dve faze. U prvoj fazi bi se na početak niza doveli svi elementi koji su manji od broja A, a iza njih bi se postavili svi elementi koji su veći ili jednaki broju A. Nakon toga, u drugoj fazi obrađuje se samo deo niza, i on se ponovo istim postupkom deli na elemente koji su manji ili jednaki od broja B (to će biti tačno elementi iz intervala $[A, B]$) i elemente koji su veći od B. Podelu možemo realizovati zasebnom funkcijom, koja prima deo niza koji se reorganizuje i granicu na osnovu koje se vrši podela, a koja vraća poziciju na kojoj počinje drugi deo reorganizovanog niza.

Jedan prolaz kroz niz

Zadatak možemo rešiti pomoću samo jednog prolaza kroz niz i to "u mestu" tj. bez korišćenja pomoćnog niza. Algoritam u nastavku poznat je pod nazivom "Holandska zastava trobojka" (engl. Dutch national flag, slika 1.2) i pripisuje se čuvenom informatičaru Dajkstri (engl. Edsger W. Dijkstra).

Slika 1.2: Holandska trobojka po kojoj je Dajkstra nazvao ovaj problem

Održavaćemo tri promenljive `l`, `d` i `i` i tokom petlje nametnućemo sledeće uslove koji će biti invarijante petlje. Pretpostavavićemo da tekuće vrednosti l, d i i ovih promenljivih zadovoljavaju $0 \leq l \leq i \leq d \leq n$ i da važi:

- u intervalu pozicija $[0, l)$ nalaziće se elementi manji od A tj. brojevi iz intervala $(-\infty, A)$,
- u intervalu pozicija $[l, i)$ nalaziće se elementi iz intervala $[A, B]$,
- u intervalu pozicija $[i, d)$ nalaziće se elementi koji još nisu ispitani,
- u intervalu pozicija $[d, n)$ nalaziće se elementi koji su veći od B tj. elementi iz intervala $(B, +\infty)$.

Dakle, održavamo raspored `<<<===???>>>`, gde su sa `<` obeleženi elementi prve grupe, sa `=` elementi druge, sa `?` elementi treće grupe, a sa `>` elementi četvrte grupe.

Da bi invarijanta važila pre ulaska u petlju, jasno je da mora da važi da je $i = 0$ i $d = n$ (jer su svi elementi iz intervala $[i, d) = [0, n)$ neispitani). Takođe, da bismo bili sigurni da su u intervalu $[0, l)$ svi elementi manji od A, taj interval mora biti prazan i mora da važi da je $l = 0$. Nakon ovakve inicijalizacije i interval $[l, i) = [0, 0)$ i interval $[d, n) = [n, n)$ je prazan, pa zadovoljava nametnuti uslov.

Petlja će se izvršavati dok god ima neispitanih elemenata, a to je dok je $i < d$. Razmotrimo kako treba da izgleda telo petlje, da bi uslovi bili održani.

- Ako je element na poziciji i manji od broja A tada ćemo ga zameniti sa elementom na poziciji l (prvim elementom iz intervala $[A, B]$), nakon čega možemo uvećati i i i l.

- Inače, ako je element na poziciji i manji ili jednak od B on pripada intervalu $[A, B]$ i već je na svom dopuštenom mestu, pa samo možemo uvećati vrednost i.
- Inače, element je veći od B i tada možemo smanjiti vrednost d i razmeniti element na poziciji i sa elementom na (umanjenoj) poziciji d, ne menjajući vrednost i (da bi se element koji je upravo doveden na poziciju i mogao ispitati u narednoj iteraciji).

Na kraju petlje važi da je $i = d$. Uz ostale nametnute uslove tvrđenje odatle sledi (elementi iz intervala pozicija $[0, l)$ su manji od A, elementi iz intervala pozicija $[l, i) = [l, d)$ su između A i B, interval nepregledanih elemenata $[i, d)$ je prazan, dok su elementi iz intervala $[d, n)$ veći od B). Dakle, niz je razbijen na nadovezane segmente $[0, l)$, $[l, d)$ i $[d, n)$ i u svakom segmentu se nalaze odgovarajući elementi.

```
// funkcija organizuje elemente vektora u tri dela:
// prvo elementi iz intervala (-Inf, A), zatim
// elementi iz intervala [A, B] i na kraju
// elementi iz intervala (B, Inf)
void podelaNiza(vector<int>& niz, int A, int B) {
  // - na pozicijama [0, l) su elementi iz intervala (-Inf, A)
  // - na pozicijama [l, i) su elementi iz intervala [A, B]
  // - na pozicijama [i, d) su jos neispitani elementi
  // - na pozicijama [d, n) su elementi iz intervala (B, Inf)
  int l = 0, i = 0, d = niz.size();
  // dok god postoje neispitani elementi
  while (i < d) {
    if (niz[i] < A)
      // menjamo tekuci element sa prvim elementom iz [A, B]
      swap (niz[i++], niz[l++]);
    else if (niz[i] <= B)
      // tekuci element ostaje na svom mestu
      i++;
    else
      // menjamo tekuci element sa poslednjim neispitanim
      swap(niz[i], niz[--d]);
```

```
    }
  }
```

Primer 1.4.1. *Razmotrimo rad algoritma na jednom primeru. Neka je* $A = 4$, $B = 7$ *i neka niz ima sadržaj* `5 1 8 6 3 9 4 2`. *U nastavku ćemo prikazati stanje niza tokom izvođenja algoritma.*

Nakon inicijalizacije promenljivih stanje je sledeće.

l							*d*
5	*1*	*8*	*6*	*3*	*9*	*4*	*2*
i							

Element 5 *na poziciji* i *pripada intervalu* $[A, B]$, *pa se samo uvećava pokazivač* i.

l							*d*
5	*1*	*8*	*6*	*3*	*9*	*4*	*2*
	i						

Element 1 *na poziciji* i *pripada intervalu* $(-\infty, A)$, *pa se razmenjuje sa elementom* 5 *na poziciji* l *i oba pokazivača* i *i* l *se uvećavju.*

	l						*d*
1	*5*	*8*	*6*	*3*	*9*	*4*	*2*
		i					

Element 8 *na poziciji* i *pripada intervalu* $(B, +\infty)$, *pa se menja sa prvim elementom ispred pozicije* d *(što je element* 2*) i pokazivač* d *se umanjuje.*

	l						*d*
1	*5*	*2*	*6*	*3*	*9*	*4*	*8*
		i					

Element 2 *na poziciji* i *pripada intervalu* $(-\infty, A)$, *pa se razmenjuje sa elementom* 5 *na poziciji* l *i oba pokazivača* i *i* l *se uvećavju.*

		l					d
1	2	5	6	3	9	4	8
			i				

Element 6 *na poziciji* i *pripada intervalu* $[A, B]$*, pa se samo uvećava pokazivač* i*.*

		l					d
1	2	5	6	3	9	4	8
				i			

Element 3 *na poziciji* i *pripada intervalu* $(-\infty, A)$*, pa se razmenjuje sa elementom* 5 *na poziciji* l *i oba pokazivača* i *i* l *se uvećavju.*

			l				d
1	2	3	6	5	9	4	8
					i		

Element 9 *na poziciji* i *pripada intervalu* $(B, +\infty)$*, pa se menja sa prvim elementom ispred pozicije* d *(što je element* 4*) i pokazivač* d *se umanjuje.*

			l			d	
1	2	3	6	5	4	9	8
					i		

Element 4 *na poziciji* i *pripada intervalu* $[A, B]$*, pa se samo uvećava pokazivač* i*.*

			l			d	
1	2	3	6	5	4	9	8
						i	

Pošto i *dostiže vrednost* d*, algoritam se završava. Elementi na pozicijama* $[l, d)$ *su iz intervala* $[A, B]$*, levo od njih su manji elementi, a desno veći.*

Zadatak: Najmanji broj koji nije zbir elemenata skupa

Uz terazije stoji skup tegova celobrojnih masa. Odrediti koja je najmanja celobrojna masa koja se ne može izmeriti pomoću tih tegova (svaki teg se može upotrebiti samo jednom).

Napomena: masa tela se uz pomoć terazija meri tako što se na jedan tas stavi to telo, a na drugi tegovi koje imamo na raspolaganju, tako da terazije budu u ravnoteži.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 10^3$), a zatim u narednom redu sortiran niz od n prirodnih brojeva manjih od 10^4, koji predstavljaju skup masa tegova.

Opis izlaza

Na standardni izlaz ispisati traženi najmanji prirodan broj koji nije zbir nekih elemenata tog skupa.

Primer

Ulaz	*Izlaz*
`8` `1 2 4 7 15 32 35 48`	`30`

Rešenje

Pomoću tegova mase $m_0, \ldots, m_i$ sigurno ne možemo da izmerimo mase veće od $Z_i = m_0 + \ldots + m_i$. Pitanje je da li možemo izmeriti sve mase iz intervala $[0, Z_i]$.

Činjenica da su elementi sortirani olakšava rešenje zadatka. Obrađivaćemo element po element i prilikom dodavanja svakog novog tega mase m_i proveravaćemo da li u intervalu $[0, Z_i]$ postoji neka masa koja se ne može izmeriti ili na snazi ostaje invarijanta da je moguće izmeriti sve mase iz $[0, Z_i]$.

Pretpostavimo da se tegovima $m_0, \ldots, m_i$ mogla izmeriti svaka masa iz intervala $[0, Z_i]$. Tada se dodavanjem tega m_{i+1} može izmeriti svaka masa iz intervala $[0 + m_{i+1}, Z_i + m_{i+1}] = [m_{i+1}, Z_{i+1}]$ (dodavanjem tega m_{i+1} na prethodno odabrane tegove). Ako je novi učitani element $m_{i+1} > Z_i + 1$, onda se $Z_i + 1$ ne može izmeriti (jer je niz sortiran, pa su mase svih preostalih tegova veće ili jednake m_{i+1}), i to je tražena vrednost. U suprotnom, unija intervala $[0, Z_i] \cup [m_{i+1}, Z_{i+1}]$ pokriva interval $[0, Z_{i+1}]$, pa se sve mase iz tog intervala mogu izmeriti i invarijanta ostaje na snazi.

Primer 1.4.2. *Na primer, neka je dat niz* $1, 2, 3, 5, 14, 20, 27$.

- *0 možemo dobiti kao zbir praznog skupa* $\{\}$.
- *1 možemo dobiti kao zbir skupa* $\{1\}$.
- *Kada u prethodne skupove uključimo i* 2*, možemo dobiti sve brojeve zaključno sa* 3 *(*2 *kao* $\{2\}$ *i* 3 *kao* $\{1, 2\}$*).*
- *Kada u prethodne skupove uključimo i* 3*, možemo dobiti sve brojeve zaključno sa* 6 *(*4 *kao* $\{1, 3\}$*,* 5 *kao* $\{2, 3\}$ *i* 6 *kao* $\{1, 2, 3\}$*).*
- *Kada u prethodne skupove uključimo* 5 *možemo dobiti sve brojeve zaključno sa* 11 *(*7 *kao* $\{2, 5\}$*,* 8 *kao* $\{1, 2, 5\}$ *i* 9 *kao* $\{1, 3, 5\}$*,* 10 *kao* $\{2, 3, 5\}$ *i* 11 *kao* $\{1, 2, 3, 5\}$*).*
- *Pošto je naredni broj* 14*, jasno je da se broj* 12 *ne može nikako dobiti.*

```
int n;
cin >> n;
// invarijanta: sabiranjem elemenata trenutnog obradjenog
// skupa tegova mogu se dobiti sve mase iz intervala [0, zbir]
int zbir = 0;
for (int i = 0; i < n; i++) {
  int m; cin >> m;
  if (m > zbir + 1)
    break;
  zbir += m;
}
cout << zbir + 1 << endl;
```

Dokažimo korektnost ovog algoritma.

Lema 1.4.3. *Neka je* Z *tekuća vrednost promenljive* `zbir`*. Invarijanta petlje je da je* $0 \leq i \leq n$*, da je* Z *zbir prvih* i *elemenata niza i da se svaki broj iz intervala* $[0, Z]$ *može dobiti kao zbir nekog podskupa prvih* i *elemenata niza.*

Dokaz. Pre ulaska u petlju je $i = 0$ i $Z = 0$. Zbir prvih $i = 0$ elemenata niza je po definiciji nula (tj. Z). Broj 0 je jedini element intervala $[0, Z] = [0, 0]$ i on se može dobiti kao zbir praznog podskupa (tj. 0 elemenata polaznog niza).

Obeležimo sa Z i i vrednosti promenljivih `zbir` i `i` pre ulaska u petlju, a sa Z' i i' vrednosti nakon izvršavanja tela i koraka petlje. Pretpostavimo da invarijanta važi pre ulaska u petlju.

- Ako je $m_i > Z + 1$, tvrdimo da je $Z + 1$ traženi najmanji broj. Na osnovu invarijante znamo da su svi brojevi iz intervala $[0, Z]$ pokriveni, tako da manji broj od $Z + 1$ ne može biti rešenje. Dokažimo da broj $Z + 1$ ne može biti zbir nekog podskupa tegova. Pošto je niz sortiran, važi $Z + 1 < m_i \leq m_{i+1} \leq ... \leq m_{n-1}$. Dakle, ni jedan od tih elemenata ne sme biti uključen u podskup jer bi njihovim uključivanjem zbir već premašio $Z+1$. Podskup se mora sastojati samo od elemenata m_0 do m_{i-1}, međutim, pošto je Z njihov zbir, zbir svakog njihovog podskupa je manji ili jednak Z. Dakle, $Z + 1$ se ne može izmeriti i on je traženo rešenje.

- Ako je $m_i \leq Z + 1$, tada je $Z' = Z + m_i$, $i' = i + 1$ i tvrdimo da je Z' zbir svih elemenata m_0, ..., m_i i da se svaki broj iz intervala $[0, Z']$ može predstaviti kao zbir nekog podskupa prvih i' elemenata niza. Prva tvrdnja je prilično očigledna, jer je po pretpostavci Z zbir svih elemenata m_0, ..., m_{i-1}, a $Z' = Z + m_i$. Na osnovu pretpostavke znamo da svi brojevi iz $[0, Z]$ mogu biti zbirovi podskupova prvih i elemenata niza. Slično i svi brojevi iz intervala $[m_i + 0, m_i + Z]$ se mogu dobiti kao zbir nekog podskupa prvih $i' = i + 1$ elementa niza. Naime, taj podskup će biti unija elementa m_i i onog podskupa prvih i elemenata niza čiji je zbir jednak razlici između tog broja i broja m_i — on je iz intervala $[0, Z]$, pa na osnovu pretpostavke takav podskup postoji. Pošto je $m_i \leq Z + 1$ unija intervala $[0, Z]$ i $[m_i, m_i + Z]$ je $[0, m_i + Z] = [0, Z']$. Zato je svaki element iz $[0, Z']$ jednak zbiru nekog podskupa prvih i' elemenata niza, pa invarijanta ostaje očuvana. □

Teorema 1.4.6. *Kada se program završi, vrednost $Z + 1$ je najmanji prirodni broj koji se ne može predstaviti kao zbir unetih brojeva.*

Dokaz. Slučaj kada se petlja završi prekidom, jer je $m_i > Z + 1$ je već razmotren. Kada se petlja završi, važi da je $i = n$. Na osnovu invarijante Z je zbir svih elemenata niza, i svaki broj iz $[0, Z]$ jeste zbir nekog podskupa prvih $i = n$ elemenata niza, tj. celog niza. Zato je $Z + 1$ najmanji element koji nije moguće dobiti (jer se uključivanjem svih elemenata dobija najviše Z) i prikazano rešenje je ispravno. □

1.4.5 *Ispitivanje zaustavljanja programa*

Ne postoji opšti postupak kojim se za proizvoljni zadati program može utvrditi da li se on zaustavlja za zadate vrednosti argumenata[2]. Ipak, za mnoge konkretne programe, može se utvrditi da li se zaustavljaju ili ne. Kako ne postoji opšti postupak koji bi se primenio na sve programe, zaustavljanje svakog programa mora se ispitivati zasebno i koristeći specifičnosti tog programa.

U programima u kojima su petlje jedine naredbe koje mogu dovesti do nezaustavljanja potrebno je dokazati zaustavljanje svake pojedinačne petlje. Ovo se obično radi tako što se definiše dobro zasnovana relacija[3] takva da su susedna stanja kroz koje se prolazi tokom izvršavanja petlje međusobno u relaciji. Kod elementarnih algoritama ovo se obično radi tako što se izvrši neko preslikavanje skupa stanja u skup prirodnih brojeva i pokaže da se svako susedno stanje preslikava u manji prirodan broj.[4] Pošto je relacija $>$ na skupu prirodnih brojeva dobro zasnovana, i ovako definisana relacija na skupu stanja biće dobro zasnovana. Veličina koja se menja (smanjuje) tokom izvršavanja petlje naziva se *varijanta petlje*.

Razmotrimo nekoliko primera.

Stepenovanje

Iterativni algoritam koji vrši stepenovanje uzastopnim množenjem se zaustavlja. Zaista, u svakom koraku petlje vrednost $n - i$ je prirodan broj (jer invarijanta kaže da je $0 \leq i \leq n$). Ova vrednost opada kroz svaki korak petlje (jer se n ne menja, a i raste), pa u jednom trenutku mora da dostigne vrednost 0.

Brzo stepenovanje

Rekurzivna funkcija koja vrši brzo stepenovanje se zaustavlja. Zaista, u svakon narednom rekurzivnom pozivu vrednost n je strogo manja od trenutne. Zaista, rekurzivni pozivi se vrše samo kada je $n > 0$. Ako je n parno, vrednost $n/2$ je strogo manja od n (jer je tada $n \geq 2$), a ako je neparna, vrednost $n - 1$ je strogo manja od n. Pošto je n prirodan broj on mora u nekom trenutku dostići vrednost 0, kada se izlazi iz rekurzije.

Kolacova hipoteza

[2] Ovo je čuveni halting-problem čiju je neodlučivost prvi dokazao Alan Tjuring.

[3] Za relaciju $\succ$ se kaže da je *dobro zasnovana* (engl. well founded) ako ne postoji beskonačan opadajući lanac elemenata $a_1 \succ a_2 \succ \ldots$.

[4] Smatramo da i nula pripada skupu prirodnih brojeva.

Nije poznato da li se naredna funkcija zaustavlja za proizvoljnu ulaznu vrednost n.[5]

```
void f(unsigned n)
{
  while (n > 1) {
    if (n % 2)
      n = 3*n+1;
    else
      n = n/2;
  }
}
```

Opšte uverenje je da se funkcija zaustavlja za svaku ulaznu vrednost n (to tvrdi još uvek nepotvrđena *Kolacova (Collatz) hipoteza* iz 1937). Navedeni primer pokazuje kako pitanje zaustavljanja čak i za neke veoma jednostavne programe može da bude ekstremno komplikovano.

1.5 Formalno ispitivanje korektnosti programa

Sva dosadašnja razmatranja o korektnosti programa vršena su zapravo poluformalno, tj. nije postojao precizno opisan formalni sistem u kojem se vrši dokazivanje korektnosti imperativnih programa. Jedan od najznačajnijih formalnih sistema ovog tipa opisao je Toni Hor (engl. Tony Hoare).

Semantika određenog programskog koda može se zapisati trojkom oblika

$$\{\varphi\}\mathtt{P}\{\psi\}$$

gde je `P` niz naredbi, a $\{\varphi\}$ i $\{\psi\}$ su logičke formule koje opisuju veze između promenljivih koje se javljaju u tim naredbama. Trojku $(\varphi, \mathtt{P}, \psi)$ nazivamo *Horova trojka*. Interpretacija trojke je sledeća: "Ako izvršenje niza naredbi `P` počinje sa vrednostima ulaznih promenljivih (u stanju) koje zadovoljavaju uslov $\{\varphi\}$ i ako `P`

[5] Pošto je opseg tipa `unsigned` ograničen, u ovom konkretnom primeru se mogu testirati sve moguće vrednosti za n, međutim, u opštem slučaju, ako razmotrimo bilo koji mogući prirodan broj n, tada je zaustavljanje nepoznato.

završi rad u konačnom broju koraka, tada vrednosti programskih promenljivih (stanje) zadovoljavaju uslov $\{\psi\}$". Uslov $\{\varphi\}$ naziva se *preduslov*, a uslov $\{\psi\}$ naziva se *postuslov (posleuslov)*.

Na primer, trojka $\{x = 1\}$`y:=x`$\{y = 1\}$[6], opisuje dejstvo naredbe dodele i kaže da, ako je vrednost promenljive x bila jednaka 1 pre izvršavanja naredbe dodele, i ako se naredba dodele izvrši, tada će vrednost promenljive y biti jednaka 1. Ova trojka je tačna. S druge strane, trojka $\{x = 1\}$`y:=x`$\{y = 2\}$ govori da će nakon dodele vrednost promenljive y biti jednaka 2 i ona nije tačna. Specifikacija programa se može zadati u obliku Horove trojke. Na primer, specifikacija funkcije stepenovanja se može zadati u sledećem obliku:

$$\{n \geq 0 \wedge (n = 0 \Rightarrow x \neq 0)\}\texttt{s := stepen(x, n)}\{s = x^n\}$$

Horovom logikom se definišu pravila kojima se od jednostavnijih Horovih trojki izvode složenije. Opis ovih pravila i primer njihove primene na dokaz korektnosti funkcije stepenovanja dati su u poglavlju 1.A.3.

Formalni dokazi (u jasno preciziranom logičkom okviru) su važni jer mogu da se generišu automatski uz pomoć računara ili barem interaktivno u saradnji čoveka sa računarom. U oba slučaja, formalni dokaz može da se proveri automatski (dok automatska provera neformalnog dokaza nije moguća). Softver čija je ispravnost dokazana i proverena automatski od strane namenskih programa je najpouzdaniji softver i zahtevi za takvim nivoom pouzdanosti postavljaju se za neke bezbednosno kritične aplikacije (kao što je, na primer, kontrolni softver za metro).

Kada se program i dokaz njegove korektnosti istovremeno razvijaju, programer bolje razume sam program i njegova svojstva. Metodologija formalnog ispitivanja ispravnosti utiče i na preciznost, konzistentnost i kompletnost specifikacije, na jasnoću implementacije i sklad implementacije i specifikacije. Zahvaljujući tome dobija se pouzdaniji softver, čak i onda kada se formalni dokaz ne izvede eksplicitno.

1.6 Proba podsekcije

Ovo je proba.

[6] Prilikom opisa Horove logike, umesto C-ovske, biće korišćena sintaksa slična sintaksi korišćenoj u originalnom Horovom radu, gde se dodela umesto operatorom = označava operatorom :=.

1.6.1 Naslov u podsekciji

Zadatak: Najvredniji poklon

U prodavnici se nalazi n poklona, poređanih po ceni u rastućem redosledu. Svaki kupac želi da kupi što vredniji poklon, ali tako da ne prekorači svoj budžet.

Za svakog kupca potrebno je odrediti cenu najskupljeg poklona koji može da priušti.

Opis ulaza

- U prvom redu nalazi se ceo broj n $(1 \leq n \leq 10^5)$ — broj poklona.
- U drugom redu nalazi se n celih brojeva $c_1, c_2, \ldots, c_n$ $(1 \leq c_i \leq 10^9)$, koji predstavljaju cene poklona.
- U trećem redu nalazi se ceo broj q $(1 \leq q \leq 10^5)$ — broj kupaca.
- U četvrtom redu nalazi se q celih brojeva $b_1, b_2, \ldots, b_q$ $(1 \leq b_i \leq 10^9)$, gde b_i predstavlja budžet i-tog kupca.

Opis izlaza

Za svakog kupca ispisati po jedan ceo broj u zasebnom redu – cenu najskupljeg poklona koji je manji ili jednak od x_i.

Ukoliko takav poklon ne postoji, ispisati -1.

Primer

Ulaz	*Izlaz*
`5`	`-1`
`20 10 70 35 50`	`35`
`3`	`70`
`5 40 70`	

Rešenje

Nakon što se niz cena sortira, zadatak se veoma efikasno može rešiti *binarnom pretragom*, primenjenu iznova na svaki budžet b_i. Krenimo od sledeće invarijante. Tokom petlje održavaćemo dva pokazivača `l` i `d`, tako da su sve cene levo od pokazivača `l` takve da se poklon može kupiti za dati budžet, a sve cene desno od pokazivača `d` takve da se poklon ne može kupiti za dati budžet, dok su na pozicijama iz intervala $[l, d]$ cene koje algoritam još nije ispitao. Dakle, pretpostavimo da je stanje niza sledeće:

			l				d			
$\leq$	$\leq$	$\leq$	?	?	?	?	?	>	>	>

U početku su sve cene nepregledane, pa interval $[l, d]$ treba da se poklopi sa celim nizom, tj. l treba inicijalizovati na 0, a d na $n - 1$.

U svakom koraku petlje određujemo središnju poziciju s u intervalu $[l, d]$. Da bi se smanjila mogućnost prekoračenja, tu poziciju umesto pomoću $\lfloor \frac{l+d}{2} \rfloor$, možemo izračunati pomoću $l + \lfloor \frac{d-l}{2} \rfloor$.

- Ako se element na poziciji s uklapa u dati budžet tj. ako je cena $c_s \leq b_i$, tada se, zbog sortiranosti, mogu priuštiti i svi pokloni na pozicijama levo od s, pa invarijanta ostaje na snazi ako se l pomeri na poziciju $s + 1$.
- U suprotnom, ako je $b_i < c_s$, tada se, zbog sortiranosti, ne može prišutiti ni jedan poklon na pozicijama desno od s, pa invarijanta ostaje na snazi ako se l pomeri na poziciju $s - 1$.

Petlja se izvršava dok god ima nepregledanih cena, tj. dok je interval $[l, d]$ neprazan. Kada se pretraga završi, stanje u nizu je sledeće.

				d	l					
$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	>	>	>	>	>	>

Na osnovu invarijante znamo da se svaki element levo od l može priuštiti, a da se ni jedan desno od d ne može, pa se stoga traženi najvredniji poklon nalazi na poziciji $d = l - 1$. Ako su svi pokloni preskupi, krajnje vrednosti će biti $d = -1$ i $l = 0$, pa se vraćanjem vrednosti d i u tom slučaju dobija korektan rezultat.

Funkcija se zaustavlja jer se u svakom koraku širina intervala $[l, d]$ striktno smanjuje (zahvaljujući zaokruživanju naniže prilikom određivanja središnje pozicije s).

```
// vraca poziciju u sortiranom nizu cene najvrednijeg poklona
// koji se moze kupiti za dati budzet ili
// -1 ako su svi pokloni preskupi
int najvredniji_poklon(const vector<int>& cene, int budzet) {
  int n = cene.size();
```

```
  int l = 0, d = n - 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (cene[s] <= budzet)
      l = s + 1;
    else
      d = s - 1;
  }
  return d;
}
```

1.A Dodatak: ispravnost

1.A.1 Primeri softverskih grešaka

Evo nekih od najpoznatijih i najzanimljivijih (pokušajte da na internetu pronađete još ovakvih primera):

- "Internet crv"[7] Moris (engl. Morris) raširio se, koristeći propuste i greške u nekoliko sistemskih programa, putem interneta 1988. godine (kao jedan od prvih takvih programa) i privukao pažnju mnogih svetskih medija. Autor crva, student Robert Moris, nije nameravao da crv bude destruktivan, već samo da se replicira i širi preko mreže, ali stepen replikacije bio je takav da su se računari "inficirali" mnogo puta, do nivoa da više nisu mogli da funkcionišu. Na taj način, nekoliko hiljada računara prestalo je sa normalnim funkcionisanjem i bilo je neoperativno nekoliko dana. Ukupna šteta je u to vreme procenjena na nekoliko miliona dolara. Suđenje autoru crva bilo je jedno od prvih takvih, zatvorska kazna je na kraju bila uslovna, uz novčanu kaznu i društveno koristan rad.

- Eksplozija rakete *Ariane* (fr. Ariane 5) 1996. uzrokovana konverzijom broja iz šezdesetčetvorobitnog realnog u šesnaestobitni celobrojni zapis koja je dovela do prekoračenja.

[7] Osnovna razlika između računarskog virusa i računarskog crva je u tome što virus mora biti aktiviran nekakvom akcijom na računaru na kojem se nalazi. S druge strane, crv je samostalni program koji može da se replicira i širi čim dopre do nekog računarskog sistema.

- Greška u *Patriot* raketnom sistemu tokom Zalivskog rata 1991. bila je posledica akumulacije numeričke greške u proračunu vremena koristeći pokretni zarez ograničene preciznosti. Interni sat sistema merio je vreme u desetinkama sekunde, a pri konverziji u sekunde dolazilo je do malog zaokruživanja koje se vremenom sabiralo. Nakon dugotrajnog rada sistema bez restartovanja, ova greška je postala dovoljno velika da uzrokuje značajno odstupanje u predviđanju položaja cilja. Kao posledica, sistem nije uspeo da presretne dolazeći projektil, što je dovelo do ljudskih žrtava. Ovaj incident jasno ilustruje kako i male numeričke greške mogu imati ozbiljne posledice u sistemima koji rade dugo i u realnom vremenu.

- Greška u numeričkom koprocesoru procesora *Pentium* 1994.
uzrokovana pogrešnim indeksima u `for` petlji u okviru softvera koji je radio dizajn čipa,

- Pad orbitera poslatog na Mars 1999. uzrokovan činjenicom da je deo softvera koristio metričke, a deo softvera engleske jedinice.

- U slučaju *Therac-25*, medicinskog uređaja za terapiju zračenjem, softverska greška imala je fatalne posledice po pacijente. Dve konkurentne rutine upravljale su istim delovima memorije bez odgovarajuće sinhronizacije. U određenim okolnostima sistem bi pogrešno procenio konfiguraciju uređaja i isporučio višestruko veću dozu zračenja nego što je predviđeno. Pošto su ranije verzije sistema imale hardverske zaštite koje su uklonjene u ovoj verziji, softver je ostao jedina linija odbrane. Kombinacija lošeg dizajna, nedovoljnog testiranja i izostanka bezbednosnih mehanizama dovela je do više ozbiljnih povreda i smrtnih ishoda.

- U Los Anđelesu je 14. septembra 2004. godine više od četiristo aviona u blizini aerodroma istovremeno izgubilo vezu sa kontrolom leta. Na sreću, zahvaljujući rezervnoj opremi unutar samih aviona, do nesreće ipak nije došlo. Uzrok gubitka veze bila je greška prekoračenja u brojaču milisekundi u okviru sistema za komunikaciju sa avionima. Da ironija bude veća, ova greška je bila otkrivena ranije, ali pošto je do otkrića došlo kada je već sistem bio isporučen i instaliran na nekoliko aerodroma, njegova jednostavna popravka i zamena nije bila moguća. Umesto toga, preporučeno je da se sistem resetuje svakih 30 dana kako do prekoračenja ne bi došlo. Procedura nije ispoštovana i greška se javila posle tačno 2^{32} milisekundi, odnosno 49.7 dana od uključivanja sistema.

- Pad satelita *Kriosat* (engl. *Cryosat*)} 2005. godine koštao je Evropsku Uniju oko 135 miliona evra. Pad je uzrokovan greškom u softveru zbog koje nije na vreme došlo do razdvajanja satelita i rakete koja ga je nosila.

- Ranljivost poznata kao *Heartbleed* nastala je u implementaciji proširenja u biblioteci OpenSSL, gde nije adekvatno proveravana dužina podataka koje klijent prijavljuje da šalje. Napadač je mogao da pošalje mali paket uz lažno veliku deklarisanu dužinu, nakon čega bi server u odgovoru vratio i deo svoje radne memorije. Time su mogli biti otkriveni osetljivi podaci poput korisničkih lozinki, sesijskih kolačića, pa čak i privatnih kriptografskih ključeva. Problem je bio posebno opasan jer nije ostavljao tragove u logovima i zahvatio je veliki broj servera širom interneta, što je dovelo do masovnog resetovanja lozinki i zamene sertifikata.

- Više od pet procenata penzionera i primalaca socijalne pomoći u Nemačkoj je privremeno ostalo bez svog novca kada je 2005. godine uveden novi računarski sistem. Greška je nastala zbog toga što je sistem, koji je zahtevao desetocifreni zapis svih brojeva računa, kod starijih računa koji su imali osam ili devet cifara brojeve dopunjavao nulama, ali sa desne umesto sa leve strane kako je trebalo.

- Incident sa *XZ Utils* 2024. godine predstavlja jedan od najozbiljnijih primera napada na lanac razvoja otvorenog softvera. Napadač je tokom dužeg vremenskog perioda stekao poverenje održavaoca projekta i postepeno ubacio zlonameran kôd u biblioteku `liblzma`, koja se široko koristi u Linux sistemima. Taj kôd je bio pažljivo prikriven i aktivirao se samo u specifičnim uslovima, omogućavajući potencijalno neautorizovan pristup sistemima preko SSH servera, što bi, imajući u vidu široku rasprostranjenost Linux servera imalo katastrofalne posledice. Ranljivost je otkrivena relativno rano, pre nego što je dospela u stabilne verzije većine distribucija, zahvaljujući neobičnom ponašanju performansi koje je privuklo pažnju istraživača. Ovaj slučaj je pokazao koliko su projekti otvorenog koda ranjivi i koliko je važno pažljivo upravljanje doprinosima i revizija koda.

- Kompanije Dell i Apple morale su tokom 2006. godine da korisnicima zamene više od pet miliona laptop računara zbog greške u dizajnu baterije kompanije Sony koja je uzrokovala da se nekoliko računara zapali.

- Ne naročito opasan, ali veoma zanimljiv primer greške je greška u programu *Microsoft Excell 2007* koji, zbog greške u algoritmu formatiranja brojeva pre prikazivanja, rezultat izračunavanja izraza $77.1 \cdot 850$ prikazuje kao $100,000$ (iako je interno korektno sačuvan).

- U poslednje vreme veoma su popularne kriptovalute, zasnovane na tzv. blokčejn tehnologiji (npr. Bitcoin, Ethereum). S obzirom na ogromna finansijska sredstva koja kontroliše ovaj softver (procenjuje se da je krajem 2024. godine vrednost svih kritpovaluta bila oko $3 \cdot 10^{12}$ američkih dolara), svaka greška može dovesti do ogromnih finansijskih gubitaka, Na primer, u čuvenom napadu na investicioni fond DAO tokom 2018. godine ukradeno je oko 150 miliona američkih dolara (u to doba oko 14% ukupnog iznosa kriptovalute ETH na mreži Ethereum na kojoj je taj fond radio). Sredstva su vraćena tako što su sporne transakcije poništene, što je u suprotnosti sa filozofijom kritpovaluta i dovelo je do cepanja mreže (tzv. hard fork). Procenjuje se da je do kraja 2024. ukupan iznos kriptovaluta koje su prenete na osnovu softverskih grešaka oko 7 milijardi dolara.

- Američka kompanija CrowdStrike je 19. jula 2024. godine distribuirala neispravnu verziju svog bezbednosnog softvera Falcon Sensor za računare sa sistemom Microsoft Windows, što je dovelo do pada oko 8,5 miliona računara širom sveta, dovelo do otkazivanja preko 5000 letova, poremetilo živote miliona ljudi i uzrokovalo ogromnu materijalnu štetu.

1.A.2 Primeri dokazivanja korektnosti

U ovom poglavlju prikazaćemo još nekoliko primera dokaza korektnosti, što rekurzivnih, što iterativnih funkcija.

1.A.2.1 Minimum niza

Problem: Definisati funkciju koja određuje minimum nepraznog niza brojeva i dokazati njenu korektnost.

Algoritam se veoma jednostavno konstruiše induktivno-rekurzivnom konstrukcijom. Dimenzija problema u ovom primeru je broj elemenata niza.

Baza: Ako niz ima samo jedan element, tada je taj element ujedno i minimum.

Korak: U suprotnom, pretpostavimo da nekako umemo da rešimo problem za manju dimenziju i na osnovu toga pokušajmo da dobijemo rešenje za ceo niz. Dakle, pretpostavimo da je dužina niza $n > 1$ i da umemo da nađemo broj m koji predstavlja minimum prvih $n - 1$ elemenata niza. Minimum celog niza dužine n je manji od brojeva m i preostalog, n-tog elementa niza (ako brojanje kreće od 0, to je element a_{n-1}).

Na osnovu ovoga možemo definisati rekurzivnu funkciju.

```
int minNiza(int a[], int n) {
    if (n == 1)
       return a[0];
    else {
       int m = minNiza(a, n-1);
       return min(m, a[n-1]);
    }
}

int main() {
    int a[] = {3, 5, 4, 1, 6, 2, 7};
    int n = 7;
    cout << minNiza(a, n) << endl;
}
```

Korektnost prethodnog algoritma se može formulisati u obliku sledeće teoreme.

Teorema 1.A.1. *Za svaki neprazan niz a (niz za koji je dužina $|a| \geq 1$) i za svako $1 \leq n \leq |a|$ poziv `minNiza(a, n)` vraća najmanji među prvih n elemenata niza a (sa a i n su obeležene vrednosti niza `a` i promenljive `n`, a sa $|a|$ dužina niza `a`).*

Dokaz. Teoremu možemo dokazati indukcijom.

- Bazu indukcije predstavlja slučaj $n = 1$, tj. poziv `minNiza(a, 1)`. Na osnovu definicije funkcije `minNiza` rezultat je `a[0]` tj. prvi član niza a_0 i tada tvrđenje trivijalno važi (jer je on ujedno najmanji među prvih 1 elemenata niza).

- Induktivna hipoteza je tvrđenje: ako važi $1 \leq n-1 < |a|$, tada poziv `minNiza(a, n-1)` vraća najmanji od prvih $n-1$ elemenata niza a. Iz te pretpostavke potrebno je da dokažemo da poziv `minNiza(a, n)` vraća najmanji od prvih n elemenata niza a (pri tom je a neprazan niz). Na osnovu definicije funkcije `minNiza`, poziv `minNiza(a, n)` će vratiti minimum brojeva m (koji predstavlja rezultat poziva `minNiza(a, n-1)`) i a_{n-1}. Pošto su uslovi induktivne hipoteze zadovoljeni, na osnovu induktivne hipoteze znamo da će m biti najmanji među prvih $n-1$ elemenata niza a. Zato će minimum broja m i n-tog elementa niza (elementa a_{n-1}) biti najmanji među prvih n elemenata niza a. □

Dokažimo sada korektnost sledeće iterativne implementacije, korišćenjem tehnike invarijante petlje.

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

int minNiza(int[] a, int n) {
    int m = a[0];
    for (int i = 1; i < n; i++)
        m = min(m, a[i]);
    return m;
}

int main() {
    int a[] = {3, 5, 4, 1, 6, 2, 7};
    int n = 7;
    cout << minNiza(a, n) << endl;
}
```

U svakom koraku petlje, deo niza čiji minimum znamo postaje duži za po jedan element. Algoritam kreće od prefiksa niza dužine 1 i postavlja promenljivu `m` na vrednost prvog elementa niza a_0. U svakom koraku petlje, pretpostavljamo da promenljiva `m` sadrži vrednost minimuma prvih i elemenata niza, a onda u telu petlje obrađeni deo niza proširujemo dodajući $i+1$-vi element niza, na poziciji i. Minimum proširenog niza se izračunava kao minimum minimuma prvih i elemenata

niza (čija je vrednost smeštena u promenljivoj `m`) i dodatnog elementa niza a_i. Nakon izvršavanja tela petlje, deo niza čiji minimum je poznat je proširen na $i + 1$ element. Na kraju petlje je i jednako dužini niza, pa promenljiva `m` sadrži minimum celog niza.

Pre nego što pređemo na precizan dokaz prethodog razmatranja, skrenimo još jednom pažnju na to da imenovane veličine u matematici (tačnije algebri) i u programiranju imaju različite osobine. Naime, imenovane veličine u matematici (parametri, nepoznate) označavaju jednu vrednost, dok u (imperativnom) programiranju imenovane veličine imaju dinamički karakter i menjaju svoje vrednosti tokom izvršavanja programa po pravilima zadatim samim programom. Na primer, brojačka promenljiva `i` u nekoj petlji može redom imati vrednosti 1, 2 i 3. Prirodno je da tekuća vrednost promenljive `i` bude označena prosto sa i. Međutim, nekada je važno da razlikujemo staru i novu vrednost promenljive `i` (vrednost pre i vrednost posle izvršavanja nekog koda), i tada ćemo koristiti oznake i' i i''. Ako želimo da naglasimo da je promenljiva redom uzimala neku seriju vrednosti, koristićemo oznake i_0 (početna vrednost promenljive `i`), i_1, i_2, … Vrednosti promenljivih koje koje se ne menjaju tokom izvršavanja programa ćemo označavati prosto imenom promenljive (npr. vrednost promenljive `n` iz prethodnog programa ćemo uvek označavati sa n, a elemente niza `a` sa a_0, a_1, …, a_{n-1}).

Sledeću teoremu možemo strogo dokazati.

Teorema 1.A.2. *Ako je niz a dužine $n \geq 1$, neposredno pre početka petlje, u svakom koraku petlje (i na njenom početku, neposredno nakon provere uslova, ali i na njenom kraju, neposredno nakon izvršavanja tela), kao i nakon izvršavanja cele petlje važi da je $1 \leq i \leq n$ i da je m minimum prvih i elemenata niza (gde je i tekuća vrednost promenljive `i`, a m tekuća vrednost promenljive `m`).*

Dokaz. Ovo tvrđenje možemo dokazati indukcijom i to po broju izvršavanja tela petlje (obeležimo taj broj sa k). Napomenimo samo da ćemo petlju `for` smatrati samo skraćenicom za petlju `while`, tako da ćemo inicijalizaciju petlje smatrati za kôd koji se izvršava pre petlje, dok ćemo korak petlje smatrati kao poslednju naredbu tela petlje.

```
    int m = a[0];
    int i = 1;
    while (i < n) {
        m = min(m, a[i]);
```

```
        i++;
    }
```

Takođe, implicitno ćemo podrazumevati da se tokom izvršavanja petlje niz ni u jednom trenutku ne menja (i to se eksplicitno može dokazati indukcijom). Ni promenljiva `n` ne menja svoju vrednost.

Obeležimo sa $m_0, m_1, \ldots, m_k, \ldots$ vrednosti promenljive `m`, a sa $i_0, i_1, \ldots, i_k, \ldots$ vrednost promenljive `i` nakon $0, 1, \ldots, k, \ldots$ izvršavanja tela petlje. Pošto promenljiva `n` ne menja svoju vrednost, njenu vrednost ćemo označiti sa n.

- Bazu indukcije čini slučaj $k = 0$ tj. slučaj kada se telo petlje nije još izvršilo. Pre ulaska u petlju promenljiva `i` se inicijalizuje na 1 (važi $i_0 = 1$). Pošto pretpostavljamo da je niz neprazan, važi da je $1 \leq i = i_0 = 1 \leq n$. Promenljiva `m` se inicijalizuje na vrednost `a[0]` (važi $m_0 = a_0$), što je zaista minimum jednočlanog prefiksa niza a. Dakle, uslovi su zadovoljeni pre prvog izvršavanja tela petlje.

- Pretpostavimo sada kao induktivnu hipotezu da tvrđenje važi nakon k izvršavanja tela petlje. Dakle, pretpostavljamo da uslovi teoreme važe za vrednosti m_k i i_k tj. da je $1 \leq i_k \leq n$ i da je m_k jednako minimumu prvih i_k elemenata niza (sa i_k i m_k obeležavamo vrednosti promenljivih nakon k izvršavanja tela petlje). Ako je uslov petlje ispunjen, to će ujedno biti i vrednosti promenljivih na početku tela petlje, pre njenog $k+1$-vog izvršavanja. Nakon k izvršavanja tela petlje važi da je $i_k = k + 1$, jer je promenljiva `i` imala početnu vrednost 1 i tačno k puta je uvećana za 1 (i ovo bi se formalno moglo dokazati indukcijom).

  Iz induktivne hipoteze i pretpostavke da je uslov petlje `i < n` ispunjen (tj. da je $i_k < n$) dokažimo da nakon $k + 1$ izvršavanja tela petlje uslovi teoreme važe i za vrednosti m_{k+1} i i_{k+1} (sa m_{k+1} i i_{k+1} obeležavamo vrednosti promenljivih nakon $k+1$ izvršavanja tela petlje). Vrednosti m_{k+1} i i_{k+1} se mogu lako odrediti na osnovu vrednosti m_k i i_k, analizom jednog izvršavanja tela petlje. Važi da je $i_{k+1} = i_k+1 = k+2$. Zato, pošto je $1 \leq i_k = k+1 < n$, važi i da je $1 \leq i_{k+1} = k + 2 \leq n$, pa je uslov koji se odnosi na raspon vrednosti promenljive `i` očuvan. Dokažimo i da je m_{k+1} minimum prvih i_{k+1} elementa niza. Važi da je m_{k+1} minimum vrednosti m_k i elementa a_{i_k}, tj. a_{k+1}. Na osnovu induktivne hipoteze znamo da je m_k minimum prvih

$i_k = k + 1$ elemenata niza. Zato će m_{k+1} biti minimum prvih $k + 2$ elemenata niza (zaključno sa elementom a_{k+1}), što je tačno i_{k+1} elemenata niza, pa i drugi uslov ostaje očuvan.

Neka su i i m vrednosti promenljivih `i` i `m` nakon izvršavanja petlje. Na osnovu dokazanog tvrđenja znamo da uslovi navedeni u njemu važe i nakon završetka petlje. Kada se petlja završi, važi da je $i = n$ (jer na osnovu prvog uslova znamo da je $1 \leq i \leq n$, a uslov petlje `i < n` nije ispunjen). Na osnovu drugog uslova znamo da je m minimum n članova niza (što je zapravo ceo niz, jer je n njegova dužina), tj. da promenljiva `m` sadrži traženu vrednost, čime je dokazana parcijalna korektnost. Zaustavljanje se dokazuje jednostavno tako što se dokaže da se u svakom koraku petlje nenegativna vrednost $n - i$ smanjuje za po 1, dok ne postane 0. □

Izolujmo ključne delove prethodnog dokaza i prikažimo ih u formatu koji ćemo i ubuduće koristiti prilikom dokazivanja invarijanti petlji (indukcija će u tim dokazima biti samo implicitna).

Lema 1.A.1. *Ako je niz a dužine $n \geq 1$, uslov da je $1 \leq i \leq n$ i da je m minimum prvih i elemenata niza je invarijanta petlje (gde sa i obeležavamo tekuću vrednost promenljive `i`, a sa m tekuću vrednost promenljive `m`).*

Dokaz. Dokažimo da invarijanta važi nakon inicijalizacije, a zatim i da se održava nakon izvršenja tela petlje.

- Pre ulaska u petlju promenljiva `i` se inicijalizuje na 1 (važi $i = 1$). Pošto pretpostavljamo da je niz neprazan, važi da je $1 \leq i \leq n$. Promenljiva `m` se inicijalizuje na vrednost `a[0]` (važi $m = a_0$), što je zaista minimum jednočlanog prefiksa niza a.

- Obeležimo sa i' i m' vrednosti promenljivih `i` i `m` pre, sa i'' i m'' vrednosti posle izvršavanja tela i koraka petlje.

  Pretpostavimo da invarijanta važi nakon ulaska u petlju tj. da je vrednost m' promenljive `m` jednaka minimumu prvih i' članova niza, da je $1 \leq i' \leq n$, kao i da je uslov petlje ispunjen tj. da je $i' < n$.

  Pošto je nakon izvršavanja tela petlje vrednost promenljive `i` uvećana za jedan, važi da je $i'' = i' + 1$. Pošto je važi da je $i' < n$ i $1 \leq i' \leq n$, nakon izvršavanja tela petlje, važiće da je $1 \leq i'' \leq n$.

Nova vrednost m'' promenljive `m` biće jednaka manjoj od vrednosti m' i a_i. Na osnovu pretpostavke važi da je m' jednako minimumu prvih i elemenata niza, tj. minimumu brojeva a_0, …, a_{i-1}, pa je m'' jednako minimumu brojeva a_0, …, a_i, što je upravo minimum prvih $i + 1$ elemenata niza, pa je zaista m'' minimum prvih i'' elemenata niza.

□

Teorema 1.A.3. *Nakon izvršavanja petlje, promenljiva `m` sadrži minimum celog niza.*

Dokaz. Na osnovu invarijante važi da je $1 \leq i \leq n$, a pošto po završetku petlje njen uslov nije ispunjen (ne važi $i < n$), važi da je $i = n$. Na osnovu invarijante važi i da promenljiva `m` sadrži minimum prvih i elemenata niza, a pošto je $i = n$, gde je n broj članova niza, to je zapravo minimum celog niza. □

Zadatak: Binarni zapis

Napiši program koji na osnovu neoznačenog celog broja n formira i ispisuje njegov 32-bitni binarni zapis.

Opis ulaza

Sa standardnog ulaza se unosi broj n ($0 \leq n \leq 2^{32} - 1$).

Opis izlaza

Na standardni izlaz ispisati 32-bitni binarni zapis broja n.

Primer 1

Ulaz	*Izlaz*
`123456`	`00000000000000011110001001000000`

Primer 2

Ulaz	*Izlaz*
`16777215`	`00000000111111111111111111111111`

Rešenje

Neka je niz od 32 logičke vrednosti popunjen vrednošću `false`. Binarni zapis određujemo tako što određujemo jednu po jednu binarnu cifru broja, zdesna nalevo. U svakom koraku petlje određujemo ostatak pri deljenju broja n sa 2, i na naredno mesto u nizu (u koraku i na mesto i) upisujemo `true` ako je taj ostatak jednak 1. Na kraju petlje, ispisujemo sadržaj niza unazad.

Primer 1.A.1. *Prikažimo izvršavanje algoritma na primeru prevođenja broja 38 u binarni zapis. Prikazaćemo samo korake koji se izvode dok n ne postane 0 (od tog trenutka nadalje niz se samo popunjava nulama).*

```
n      niz b
38
19       0
9       10
4      110
2     0110
1    00110
0   100110
```

Implementacija se može napraviti na sledeći način.

```
// broj koji se prevodi
unsigned long n;
cin >> n;

// niz binarnih cifara, redom, od cifre najmanje do
// cifre najvece tezine
bool binarneCifre[32] = {false};
// prevodjenje
for (int i = 0; n > 0; i++, n /= 2)
  binarneCifre[i] = n % 2;

// ispisujemo rezultat (od cifre najmanje tezine
for (int i = 31; i >= 0; i--)
  cout << (binarneCifre[i] ? '1' : '0');
cout << endl;
```

Dokažimo korektnost ovog algoritma.

Da bismo lakše odredili invarijantu proširimo primer izvršavanja programa vrednošću binarnog broja trenutno zapisanog u nizu b i odgovarajućim stepenom dvojke.

```
n      niz b    b     2^i
38              0     1
19       0      0     2
```

```
9       10      2     4
4      110      6     8
2     0110      6    16
1    00110      6    32
0   100110     38    64
```

Sada se lako može primetiti da u svakom redu važi da je $2^i \cdot n + b = 38$ (zaista, važi da je $1 \cdot 38 + 0 = 2 \cdot 19 + 0 = 4 \cdot 9 + 2 = 8 \cdot 4 + 6 = 16 \cdot 2 + 6 = 32 \cdot 1 + 6 = 64 \cdot 0 + 38 = 38$.

Lema 1.A.2. *Uslov $2^i \cdot n + b = n_0$ je invarijanta petlje, gde je b broj trenutno kodiran nizom binarnih cifara (ako logička vrednost na poziciji k u nizu odgovara cifri b_k, neka je $b = \sum_{k=0}^{31} b_k 2^k$), gde je i tekuća vrednost promenljive `i`, dok je n_0 početna, a n tekuća vrednost neoznačenog broja `n`.*

Dokaz. Dokažimo da je navedeni uslov invarijanta.

- Zaista na početku je $n = n_0$, $i = 0$ i $b = 0$ pa tvrđenje važi.
- Pretpostavimo da tvrđenje važi pri ulasku u petlju. Promenljive se tokom izvršavanja tela i koraka petlje menjaju na sledeći način. $n' = n \operatorname{div} 2$, $b' = b + 2^i \cdot (n \bmod 2)$ i $i' = i + 1$. Tada je $2^{i'} \cdot n' + b' = 2^{i+1} \cdot (n \operatorname{div} 2) + b + 2^i \cdot n \bmod 2 = 2^i \cdot (2 \cdot (n \operatorname{div} 2) + n \bmod 2) + b$. Na osnovu definicije celobrojnog deljenja važi da je $2 \cdot (n \operatorname{div} 2) + n \bmod 2 = n$, pa je vrednost prethodnog izraza jednaka $2^i \cdot n + b$, a na osnovu pretpostavke o tome da invarijanta važi na ulasku u telo petlje znamo da je to jednako n_0. □

Teorema 1.A.4. *Po završetku algoritma niz sadrži binarni zapis neoznačenog broja `n`.*

Dokaz. Kako je po izlasku iz petlje $n = 0$, na osnovu invarijante važi da je $b = n_0$ tj. da niz sadrži binarni zapis polaznog broja. □

1.A.3 Horova logika

Formalna specifikacija programa može se zadati u obliku Horove trojke. U tom slučaju preduslov opisuje uslove koji važe za ulazne promenljive, dok postuslov opisuje uslove koje bi trebalo da zadovolje rezultati izračunavanja. Na primer, program `P` za

množenje brojeva x i y, koji rezultat smešta u promenljivu z bi trebalo da zadovolji trojku $\{\top\}\mathtt{P}\{z = x \cdot y\}$ ($\top$ označava iskaznu konstantu *tačno*, i ovom primeru znači da nema preduslova koje vrednosti x i y treba da zadovoljavaju). Ako se zadovoljimo time da program može da množi samo nenegativne brojeve, specifikacija se može oslabiti u $\{x \geq 0 \ \wedge \ y \geq 0\}\mathtt{P}\{z = x \cdot y\}$.

Jedno od ključnih pitanja za verifikaciju je pitanje da li je neka Horova trojka tačna (tj. da li program zadovoljava datu specifikaciju). Hor je dao formalni sistem (aksiome i pravila izvođenja) koji omogućava da se tačnost Horovih trojki dokaže na formalan način (slika 1.3). Za svaku sintaksnu konstrukciju programskog jezika koji se razmatra formiraju se aksiome i pravila izvođenja koji daju rigorozan opis semantike odgovarajućeg konstrukta programskog jezika.[8]

Opis aksiome i pravila Horove logike:

- *Aksioma dodele* Ova aksioma definiše semantiku naredbe dodele. Izraz $\varphi[x \ \rightarrow \ E]$ označava logičku formulu koja se dobije kada se u formuli φ sva slobodna pojavljivanja promenljive x zamene izrazom E. Na primer, jedna od instanci ove sheme je i $\{x + 1 = 2\}\mathtt{y} := \mathtt{x} + 1\{y = 2\}$. Zaista, preduslov $x + 1 = 2$ se može dobiti tako što se sva pojavljivanja promenljive kojoj se dodeljuje (u ovom slučaju y) u izrazu postuslova $y = 2$ zamene izrazom koji se dodeljuje (u ovom slučaju $x + 1$). Nakon primene pravila posledice, moguće je izvesti trojku $\{x = 1\}\mathtt{y} := \mathtt{x} + 1\{y = 2\}$. Potrebno je naglasiti da se podrazumeva da izvršavanje dodele i sračunavanje izraza na desnoj strani ne proizvodi nikakve propratne efekte do samog efekta dodele (tj. izmene vrednosti promenljive sa leve strane dodele) koji je implicitno i opisan navedenom aksiomom.

- *Pravilo posledice* Ovo pravilo govori da je moguće ojačati preduslov i oslabiti postuslov svake trojke. Na primer, od tačne trojke $\{x = 1\}\mathtt{y} := \mathtt{x}\{y = 1\}$, moguće je dobiti tačnu trojku $\{x = 1\}\mathtt{y} := \mathtt{x}\{y > 0\}$. Slično, na primer, ako program P zadovoljava $\{\top\}\mathtt{P}\{z = x \cdot y\}$, tada će zadovoljavati i trojku $\{x \geq 0 \wedge y \geq 0\}\mathtt{P}\{z = x \cdot y\}$.

- *Pravilo kompozicije* Pravilom kompozicije opisuje se semantika sekvencijalnog izvršavanja dve naredbe. Kako bi trojka $\{\varphi\}P_1; P_2\{\psi\}$ bila

[8] U svom originalnom radu, Hor je razmatrao veoma jednostavan programski jezik (koji ima samo naredbu dodele, naredbu grananja, jednu petlju i sekvencijalnu kompoziciju naredbi), ali u nizu radova drugih autora Horov originalni sistem je proširen pravilima za složenije konstrukte programskih jezika (funkcije, pokazivače, itd.).

Aksioma dodele (assAx):

$$\{\varphi[x \to E]\}\mathtt{x} := E\{\varphi\}$$

Pravilo posledice (Cons):

$$\frac{\varphi' \Rightarrow \varphi \quad \{\varphi\}\mathtt{P}\{\psi\} \quad \psi \Rightarrow \psi'}{\{\varphi'\}\mathtt{P}\{\psi'\}}$$

Pravilo kompozicije (Comp):

$$\frac{\{\varphi\}\mathtt{P}_1\{\mu\} \quad \{\mu\}\mathtt{P}_2\{\psi\}}{\{\varphi\}\mathtt{P}_1\mathtt{;}\ \mathtt{P}_2\{\psi\}}$$

Pravilo grananja (if):

$$\frac{\{\varphi \wedge c\}\mathtt{P}_1\{\psi\} \quad \{\varphi \wedge \neg c\}\mathtt{P}_2\{\psi\}}{\{\varphi\}\mathtt{if\ c\ then\ P}_1\ \mathtt{else\ P}_2\{\psi\}}$$

Pravilo petlje (while):

$$\frac{\{\varphi \wedge c\}\mathtt{P}\{\varphi\}}{\{\varphi\}\mathtt{while\ c\ do\ P}\{\neg c \wedge \varphi\}}$$

Slika 1.3: Horov formalni sistem

tačna, dovoljno je da postoji formula μ koja je postuslov programa P_1 za preduslov φ i preduslov programa P_2 za postuslov ψ. Na primer, iz trojki $\{x = 1\}\mathtt{y} := \mathtt{x} + \mathtt{1}\{y = 2\}$ i $\{y = 2\}\mathtt{z} := \mathtt{y} + \mathtt{2}\{z = 4\}$, može da se zaključi $\{x = 1\}\mathtt{y} := \mathtt{x} + \mathtt{1}; \mathtt{z} := \mathtt{y} + \mathtt{2}\{z = 4\}$.

- *Pravilo grananja* Pravilom grananja definiše se semantika `if-then-else` naredbe. Korektnost ove naredbe se svodi na ispitivanje korektnosti njene `then` grane (uz mogućnost korišćenja uslova grananja u okviru preduslova) i korektnosti njene `else` grane (uz mogućnost korišćenja negiranog uslova grananja u okviru preduslova). Opravdanje za ovo, naravno, dolazi iz činjenice da ukoliko se izvršava `then` grana uslov grananja je ispunjen, dok, ukoliko se izvršava `else` grana, uslov grananja nije ispunjen.

- *Pravilo petlje* Pravilom petlje definiše se semantika `while` naredbe. Uslov φ u okviru pravila predstavlja invarijantu petlje. Kako bi se dokazalo da je inva-

rijanta zadovoljena nakon izvršavanja petlje (pod pretpostavkom da se petlja zaustavlja), dovoljno je pokazati da telo petlje održava invarijantu (uz mogućnost korišćenja uslova ulaska u petlju u okviru preduslova). Opravdanje za ovo je, naravno, činjenica da se telo petlje izvršava samo ako je uslov ulaska u petlju ispunjen.

Dokazi ispravnosti programa u okviru Horove logike koriste instance aksioma (aksiome primenjene na neke konkretne naredbe) i iz njih, primenom pravila, izvode nove zaključke. Dokazi se mogu pogodno prikazati u vidu stabla, u čijem su listovima primene aksioma.

1.A.3.1 *Stepenovanje*

Dokažimo korektnost algoritma stepenovanja. Potrebno je dokazati sledeću trojku.

$$\{x > 0 \wedge k \geq 0\}$$
$$\texttt{i := 0; s := 1; while (i < k) do begin s := s * x; i := i + 1 end}$$
$$\{s = x^k\}$$

Na osnovu pravila dodele primenjenog na dodele `i:=0` i `m:=0` i dva puta primenjenog pravila kompozicije važi trojka

$$\{x > 0 \wedge k \geq 0\}\texttt{i := 0; s := 1}\{x > 0 \wedge k \geq 0 \wedge i = 0 \wedge s = 1\}$$

Dovoljno je, dakle, da dokažemo trojku

$$\{x > 0 \wedge k \geq 0 \wedge i = 0 \wedge s = 1\}$$
$$\texttt{while (i < k) do begin s := s * x; i := i + 1 end}$$
$$\{s = x^k\}$$

Primenimo pravilo petlje na ovu petlju uz invarijantu $\varphi \equiv x > 0 \wedge k \geq 0 \wedge 0 \leq i \leq k \wedge m = x^i$. Potrebno je dokazati da telo petlje čuva invarijantu tj. dokazati trojku:

$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i \leq k \wedge m = x^i \wedge i < k\}$$
$$\texttt{m := m * x; i := i + 1}$$
$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i \leq k \wedge m = x^i\}$$

tj.

$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i < k \wedge m = x^i\}$$
$$\texttt{m} := \texttt{m} * \texttt{x};\ \texttt{i} := \texttt{i} + \texttt{1}$$
$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i \leq k \wedge m = x^i\}$$

Nakon prve dodele, na osnovu pravila dodele, važi trojka

$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i < k \wedge m = x^i\}$$
$$\texttt{m} := \texttt{m} * \texttt{x}$$
$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i < k \wedge m = x^{i+1}\}$$

Zaista, zamenom promenljive m izrazom $m \cdot x$ u postuslovu i skraćivanjem vrednosti x (što je dopušteno, jer važi $x > 0$) dobija se tačno preduslov.

Nakon druge dodele, na osnovu pravila dodele važi trojka

$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i < k \wedge m = x^{i+1}\}$$
$$\texttt{i} := \texttt{i} + \texttt{1}$$
$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i \leq k \wedge m = x^i\}$$

Zaista, zamenom vrednosti i sa $i + 1$ u postuslovu dobija se uslov $x > 0 \wedge k \geq 0 \wedge 0 \leq i + 1 \leq k \wedge m = x^{i+1}$, koji je logička posledica preduslova. Dakle, kombinovanjem pravila posledice i pravila kompozicije dobija se trojka koja garantuje da telo petlje zadovoljava invarijantu, odakle, na osnovu pravila petlje, sledi trojka

$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i \leq k \wedge m = x^i\}$$
$$\texttt{while}\ (\texttt{i} < \texttt{k})\ \texttt{do}\ \texttt{begin}\ \texttt{m} := \texttt{m} * \texttt{x};\ \texttt{i} := \texttt{i} + \texttt{1}\ \texttt{end}$$
$$\{x > 0 \wedge k \geq 0 \wedge 0 \leq i \leq k \wedge m = x^i \wedge i \geq k\}$$

Postuslov implicira $i = k$, pa zato i željeni uslov $m = x^k$. Ostalo je još da se prethodna trojka spoji sa trojkom

$$\{x > 0 \wedge k \geq 0\}\texttt{i} := \texttt{0};\ \texttt{m} := \texttt{1}\{x > 0 \wedge k \geq 0 \wedge i = 0 \wedge m = 1\}$$

koju smo ranije dokazali. Za to je dovoljno primetiti da uslov $x > 0 \wedge k \geq 0 \wedge i = 0 \wedge m = 1$ povlači uslov invarijante $x > 0 \wedge k \geq 0 \wedge 0 \leq i \leq k \wedge m = x^i$. Zato se trojka koja zaključak pravila petlje, na osnovu pravila posledice, može zameniti željenom trojkom:

$\{x > 0 \wedge k \geq 0 \wedge i = 0 \wedge m = 1\}$
`while` $(\texttt{i} < \texttt{k})$ `do begin` $\texttt{m} := \texttt{m} * \texttt{x};\ \texttt{i} := \texttt{i} + 1$ `end`
$\{m = x^k\}$

Sada na osnovu pravila kompozicije sledi korektnost celog algoritma tj. trojka:

$\{x > 0 \wedge k \geq 0\}$
$\texttt{i} := 0;\ \texttt{m} := 1;$ `while` $(\texttt{i} < \texttt{k})$ `do begin` $\texttt{m} := \texttt{m} * \texttt{x};\ \texttt{i} := \texttt{i} + 1$ `end`
$\{m = x^k\}$

1.A.4 *Softver za dokazivanje korektnosti programa*

Verifikacija softvera je razvijena oblast i postoji određen broj softverskih sistema koji se koriste za verifikaciju softvera. Samo ilustrativno, da bi se stekao neki osećaj kako se ovaj alat koristi navešćemo dva primera.

Softver **Dafny** (https://dafny.org/) je razvijen u kompaniji Microsoft. Program se piše na posebnom jeziku koji pored naredbi sadrži i specifikaciju (preduslove, postuslove, invarijante). Ako automatski dokazivač uspe da dokaže sve uslove koje je programer naveo, moguće je izvesti izvorni kod u nekom od klasičnih programskih jezika (Java, C#, ...).

U narednom kodu je prikazana funkcija za izračunavanje maksimuma niza. Kao preduslov (klauzula `requires`) navedeno je da niz mora biti neprazan, a kao postusplov (klauzule `ensures`) navedeno je da je povratna vrednost `max` zaista maksimum niza (veća je ili jednaka od svih članova niza i javlja se u nizu). Uz petlju je navedena i invarijanta koja tvrdi da se brojač `i` petlje uvek nalazi u granicama $[1, n]$, gde je n dužina niza i da u svakom koraku petlje promenljiva `max` sadrži maksimum prvih `i` članova niza. Da bi se automatski moglo dokazati zaustavljanje, formulisana je i varijanta koja kaže da se u svakom koraku petlje vrednost $n - i$ smanjuje. Ova specifikacija je dovoljna da bi softver Dafny dokazao korektnost algoritma određivanja maksimuma niza i njegove implementacije.

```
// izračunava najveći element datog nepraznog niza arr
method Max(arr: array<int>) returns (max: int)
  // maksimum računamo samo za neprazne nizove
  requires arr.Length > 0
```

```
  // max je najveći element niza
  ensures forall j :: 0 <= j < arr.Length ==> max >= arr[j]
  ensures exists j :: 0 <= j < arr.Length && max == arr[j]
{
  max := arr[0];
  var i: int := 1;
  while (i < arr.Length)
     // granice promenljive i
     // (nakon petlje i će biti jednako dužini niza arr)
     invariant 1 <= i <= arr.Length
     // promenljiva max sadrži najveću vrednost prvih i
     // elemenata niza
     invariant forall j :: j >= 0 && j < i ==> max >= arr[j]
     invariant exists j :: j >= 0 && j < i && max == arr[j]
     // naredna varijanta obezbeđuje zaustavljanje
     // (ova razlika se smanjuje sve dok ne dođe do nule)
     decreases arr.Length - i
  {
    if (arr[i] >= max)
    {
      max := arr[i];
    }
    i := i + 1;
  }
}
```

2. Efikasnost programa i složenost algoritama

Pored svojstva ispravnosti programa, važno pitanje za praktičnu primenu je koliko resursa program zahteva za svoje izvršavanje. Najvažniji resursi su vreme potrebno za izvršavanje programa i količina memorije zauzete tokom izvršavanja (mada se mogu analizirati i drugi resursi, na primer, kod mobilnih uređaja važan resurs je utrošena energija). U skladu sa ovim, razmatraju se:

- vremenska složenost;
- prostorna (memorijska) složenost.

Prostorna složenost odnosi se i na prostor koji zahvataju ulazni podaci. Kada se govori o potrebnom memorijskom prostoru ne računajući prostor koji zahvataju ulazni podaci, onda se govori o *dodatnoj prostornoj složenosti*.

Iako konkretno vreme izvršavanja zavisi od računara i sistema na kom se program izvršava, relevantne procene utrošenog vremena mogu se dati razmatrajući samo algoritam koji se koristi. Loš algoritam za neki težak problem praktično je neupotrebljiv čak iako bi se koristili računari koji su za nekoliko redova veličina efikasniji nego ovi današnji. Umesto o *efikasnosti programa* obično se govori o *složenosti algoritama*. Ovi termini koriste se sinonimno: algoritmi velike složenosti dovode do neefikasnih programa, dok algoritmi male složenosti dovode do efikasnih programa.

Zadatak programera je često da napravi balans između potrošnje različitih vrsta resursa. Neki programi zahtevaju više memorije kako bi se brže izvršavali. Na primer, ako jedan program vrši potrebno izračunavanje za 10 sekundi, a drugi za dva i po minuta, jasno je da je prvi program praktično primenjiviji. Međutim, ako prvi program za svoje izvršavanje zahteva preko 10 gigabajta memorije, drugi oko 1 gigabajt, a mi imamo računar sa 4 gigabajta memorije, prvi program nam je praktično neupotrebljiv (iako radi mnogo brže od drugog). Ipak, s obzirom na to da

savremeni računarski sistemi imaju prilično veliku količinu memorije (desetine gigabajta), vreme je ograničavajući faktor češće nego memorija i u nastavku ćemo se više baviti analizom vremenske nego memorijske složenosti algoritama. Sa druge strane, treba imati u vidu da se programi izvršavaju i na nekim specijalizovanim platformama (uređajima sa ugrađenim računarom) i da je moguće da oni imaju mnogo manje memorije nego klasični računari i mobilni uređaji, pa ni prostornu složenost ne treba zanemariti.

Koliko brzo program treba da radi da bismo ga smatrali efikasnim zavisi od konkretne primene. Na primer, ako program uspe da za pola dana reši neki do tada nerešen matematički problem, koji ljudi godinama nisu mogli da reše, on je svakako koristan i možemo ga smatrati veoma efikasnim. Sa druge strane, ako program ugrađen u automobil kontroliše kočnice prilikom proklizavanja, njemu i nekoliko stotinki za izračunavanje može biti previše, jer za to vreme automobil može da nekontrolisano sleti sa puta.

Kod programa koji obrađuju samo male količine podataka, vremenska i prostorna složenost nisu mnogo važne (jer se takvi zadaci izvršavaju brzo čak i ako se koriste naivni algoritmi, i ne zahtevaju mnogo memorije). Međutim, u mnogim situacijama se sasvim prirodno javlja potreba za obradom velikih količina podataka i tada su neefikasni programi praktično neupotrebljivi (na primer, digitalna slika veličine hiljadu puta hiljadu piksela sadrži milion piksela, pa svi algoritmi obrade slika barataju sa ogromnom količinom podataka i moraju koristiti veoma efikasne algoritme).

Iako su spori, naivni programi praktično neupotrebljivi, u razvoju softvera, posebno u nekim oblastima, ponekad se preporučuje da je poželjno prvo kreirati najjednostavniji program koji obavlja dati zadatak, a onda ga modifikovati ako je potrebno da se uklopi u zadata vremenska ili prostorna ograničenja. Naime, naivni algoritmi se često jednostavno implementiraju, lako se razumeju i njihova ispravnost se lako dokazuje, a tokom njihove implementacije programer može steći neke uvide o problemu koji se rešava, dobiti ideje za efikasnija rešenja, generisati testove za testiranje efikasnijih rešenja, uvideti da se neki delovi koda jako retko izvršavaju (pa nema potrebe gubiti vreme na njihovu optimzaciju) i slično.

Ocena potrebnih resursa se obično vrši:

- u terminima konkretnog vremena/prostora utrošenog za neke konkretne ulazne podatke;
- u terminima asimptotskog ponašanja vremena/prostora kada veličina ulaza raste.

2.1 Efikasnost programa za konkretne ulazne podatke

U nekim situacijama nas zanima ponašanje programa za konkretne ulazne podatke koje imamo zadatak da obradimo na konkretnom računaru i tada se analiza programa može izvršiti i eksperimentalno, pokretanjem programa i merenjem utrošenih resursa.

2.1.1 Merenje utrošenog vremena

Najjednostavnija mera vremenske efikasnosti programa (ili nekog njegovog dela) je njegovo vreme izvršavanja za neke konkretne vrednosti na konkretnom računaru.

U jeziku C++, pogodan način za merenje utrošenog vremena pružaju funkcije deklarisane u zaglavlju `<chrono>`. Naredni primer ilustruje kako se može dobiti utrošeno vreme izraženo u mikrosekundama[1]. Ovako dobijeno vreme nije vreme utrošeno za tekući proces nego apsolutno vreme koje je proteklo između dva merenja. Ako je vreme izvršavanja funkcije kratko, savetuje se da se ono izmeri više puta, pa da se izračuna prosečno utrošeno vreme.

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

void f(void) {  ... }

int main() {
  const int BROJ_POZIVA = 1000;
  auto pocetak = chrono::high_resolution_clock::now();
  for (int i = 0; i < BROJ_POZIVA; i++)
    f();
  auto kraj = chrono::high_resolution_clock::now();
  auto trajanje =
    chrono::duration_cast<chrono::microseconds>(kraj-pocetak);

  cout << "Procena vremena rada jednog poziva funkcije f: "
       << trajanje.count() / BROJ_POZIVA << " mikrosekundi"
```

[1] Mikrosekunda je milioniti deo sekunde, tj. hiljaditi deo milisekunde i označava se sa μs.

```
        << endl;
  return 0;
}
```

Postoje i druge funkcije koje se koriste za merenje utrošenog vremena. U programima na programskom jeziku C++, merenje vremena može da se vrši i korišćenjem funkcija iz jezika C.

2.1.2 *Profajliranje*

Kada se ustanovi da neki veći program zahteva previše vremena, postavlja se pitanje šta je tačno uzrok tome, odnosno koji deo programa troši najviše vremena i treba da se optimizuje. Informacije ovog tipa nam daju *profajleri* (engl. profiler). Njihova osnovna uloga je da pruže podatke o tome koliko puta je koja funkcija pozvana tokom (nekog konkretnog) izvršavanja, koliko je utrošila vremena i slično. Ukoliko se razvijeni program ne izvršava željeno brzo, potrebno je unaprediti neke njegove delove. Prvi kandidati za izmenu su delovi koji troše najviše vremena.

Za operativni sistem Linux, popularan je sistem `valgring` koji objedinjuje mnoštvo alatki za dinamičku analizu rada programa, uključujući profajler `callgrind`. Profajler `callgrind` se poziva na sledeći način:

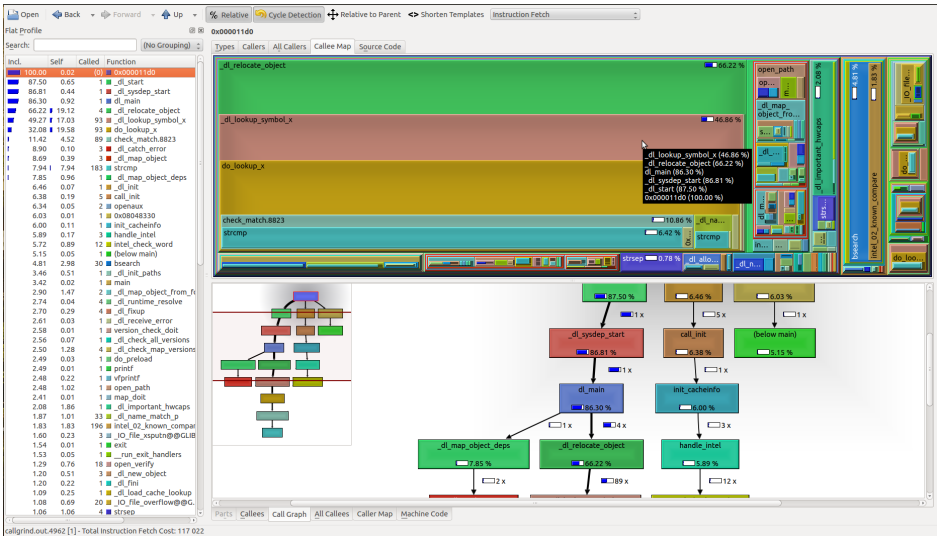
```
valgring --tool=callgrind mojprogram argumenti
```

gde `mojprogram` označava izvršivi program koji se analizira, a `argumenti` njegove argumente (komandne linije). Program `callgrind` izvršava zadati program `mojprogram` sa argumentima `argumenti` i registruje informacije o tome koja funkcija je pozivala koje funkcije (uključujući sistemske funkcije i funkcije iz standardne biblioteke), koliko je koja funkcija utrošila vremena itd. Detaljniji podaci mogu se dobiti ako je program preveden u debag režimu (`gcc` kompilatorom, debag verzija se dobija korišćenjem opcije `-g`). Prikupljene informacije program `callgrind` čuva u datoteci sa imenom, na primer, `callgrind.out.4873`. Ove podatke može da na pregledan način prikaže, na primer, program `kcachegrind`:

```
kcachegrind callgrind.out.4873
```

Slika 2.1 ilustruje rad programa `kcachegrind`. Uvidom u prikazane podatke, programer može da uoči funkcije koje troše najviše vremena i da pokuša da ih unapredi i slično.

Manje precizan, ali dosta brži je profajler `gprof`.



Slika 2.1: Ilustracija prikaza u programu `kcachegrind` podataka dobijenih profajliranjem

Za merenje utrošene količine memorije mogu se koristiti programi `memcheck` i `massif`, koji su takođe deo sistema `valgrind`.

2.1.3 *Procenjivanje potrebnog vremena*

Vreme izvršavanja programa ili nekih njegovih delova na nekom konkretnom računaru može i da se *proceni*, na osnovu analize teksta programa tj. operacija koje program izvršava. Na takvu procene utiču procene vremena izvršavanja pojedačnih operacija, ali i operativni sistema pod kojim računar radi, brzina ulazno-izlaznih hardverskih uređaja (na primer, diska) ako im program pristupa, od jezika i od kompilatora kojim je napravljen izvršivi program za testiranje, itd. Na primer, na jednom današnjem prosečnom računaru, koji radi pod operativnim sistemom Linux, operacija množenja dve vrednosti tipa `int` troši oko jednu nanosekundu tj. za jednu sekundu se na tom računaru može izvršiti oko milijardu množenja celih brojeva. Procene vremena izvršavanja programa može da pruži grubu sliku o trajanju izvršavanja programa, ali treba ih uzimati sa rezervom, jer možda ne uzimaju u obzir sve procese koji se odigravaju tokom izvršavanja programa, kao ni optimizacije koje su primenjene u kreiranju izvršivog programa. Ipak, procene su jako važne, jer programer i pre pisanja koda treba da ima neki grubi osećaj koliko resursa će program

trošiti i da li će odabrani algoritam biti dovoljno efikasan.

2.2 Asimptotsko ponašanje i red složenosti algoritma

Ponašanje programa (pa i količina utrošenih resursa), naravno, zavisi od njegovih ulaznih parametara. Jasno je, na primer, da će program brže izračunati prosečnu ocenu dvadesetak učenika jednog odeljenja, nego prosečnu ocenu nekoliko desetina hiljada učenika koji polažu državnu maturu.

Za veličinu ulaza može se uzeti broj ulaznih elemenata koje treba obraditi (na primer, dužina niza) ili broj bitova potrebnih za zapisivanje ulaza ili vrednost elemenata koje treba obraditi. Da bi se izbegla zabuna, uvek je potrebno eksplicitno navesti u odnosu na koju veličinu ulazne vrednosti se razmatra složenost.

Ponašanje programa često zavisi samo od ukupne količine podataka a ne i od konkretnih vrednosti koje obrađuje. U navedenom primeru, ponašanje programa zavisi samo od broja učenika, a ne i od konkretnih ocena koje su učenici dobili. Zato složenost algoritma često izražavamo u funkciji *veličine (dimenzije) njegovih ulaznih parametara*, a ne samih vrednosti parametara.

Mnogi algoritmi se ne izvršavaju isto za sve ulaze iste veličine, pa je potrebno naći način za opisivanje efikasnosti algoritma za razne moguće ulaze iste veličine.

- **Analiza najgoreg slučaja** zasniva procenu složenosti algoritma na najgorem slučaju (na slučaju za koji se algoritam najduže izvršava — u analizi vremenske složenosti, ili na slučaju za koji algoritam koristi najviše memorije — u analizi prostorne složenosti). Ta procena može da bude varljiva, tj. previše pesimistična. Na primer, ako se program u 99,9% slučajeva izvršava ispod sekunde, dok se samo u 0,1% slučajeva izvršava za 10 sekundi, analiza najgoreg slučaja daje zaključak samo o izvršavanju za 10 sekundi. Sa druge strane, analiza najgoreg slučaja nam daje garancije da ako je program u najgorem slučaju dovoljno efikasan, onda u svim mogućim slučajevima može da se izvrši sa raspoloživim resursima. Analiza najgoreg slučaja je najčešći oblik izražavanja složenosti algoritma i ako se ne naglasi drugačije, pod složenošću algoritma se podrazumeva upravo složenost najgoreg slučaja.

- U nekim situacijama moguće je izvršiti **analizu prosečnog slučaja** i izračunati prosečno vreme izvršavanja algoritma, ali da bi se to uradilo, potrebno je poznavati prostor dopuštenih ulaznih vrednosti i verovatnoću da se svaka

dopuštena ulazna vrednost pojavi na ulazu programa. U slučajevima kada je bitna garancija efikasnosti svakog pojedinačnog izvršavanja programa, procena prosečnog slučaja može biti varljiva, previše optimistična, i može da se desi da u nekim situacijama program ne može da se izvrši sa raspoloživim resursima. Na primer, analiza prosečnog slučaja bi za pomenuti program prijavila da se u proseku izvršava ispod jedne sekunde, međutim, za neke ulaze on se može izvršavati i preko deset sekundi.

- Analiza najboljeg slučaja je previše optimistična i najčešće nema smisla. Njeno poznavanje može biti korisno kao poznavanje donje granice izvršavanja.

Nekada se analiza vrši tako da se proceni ukupno vreme potrebno da se izvrši određen broj srodnih operacija. Taj oblik analize naziva se **amortizovana analiza** i u tim situacijama nam nije bitno vreme izvršavanja pojedinačnih operacija, već samo zbirno vreme izvršavanja svih operacija. Ovaj oblik analize je posebno pogodan u slučajevima kada vreme izvršavanja pojedinačnih operacija u nekoj seriji operacija može da varira i kada se dešava da vreme potrošeno za neko izvršavanje operacije omogućava da narednih nekoliko izvršavanja te operacije bude veoma brzo. Tipičan primer je dodavanje elemenata na kraj niza koji se dinamički širi (operacija `push_back` na vektorima u jeziku C++). Prilikom prvog dodavanja elemenata alocira se određena količina memorije (što je vremenski zahtevno), da bi se u narednim operacijama elementi samo upisivali u ranije alociranu memoriju (što je vremenski veoma efikasno). Kada se alocirani prostor popuni, pre dodavanja elementa potrebno je realocirati memoriju što je ponovo vremenski zahtevno. Ako se obezbedi da se proširivanje niza i realokacija memorije dešavaju dovoljno retko, ovakve strukture podataka imaju nisku amortizovanu složenost.

Vremenska i prostorna složenost algoritma određuju njegovu praktičnu upotrebljivost tj. najveće ulazne vrednosti za koje je moguće da će se algoritam izvršiti u nekom razumnom vremenu.

2.2.1 *Zavisnost između vremena izvršavanja i veličine ulaza*

Neka je funkcija $T(n)$ predstavlja meru vremena potrebnog za izvršavanje algoritma za ulaz veličine n (u najgorem slučaju, ako se ne naglasi drugačije). Pod pretpostavkom da se svaka instrukcija izvršava približno isto vreme, ova funkcija može se dobiti i na osnovu funkcije kojom se izražava broj instrukcija koje algoritam izvršava za ulaz veličine n.

2.2.1.1 *Elementarne funkcije*

U analizi mnogih algoritama, funkcija $T(n)$ izražava se kao neka kombinacija logaritamske funkcije $\log(n)$, korene funkcije $\sqrt{n}$, polinomskih funkcija n, n^2, n^3, ..., eksponencijalnih funkcija 2^n, 3^n, ..., faktorijelne funkcije $n!$ i slično. Njihova osnovna matematička svojstva (pre svega brzina rasta) rezimirani su u dodatku 2.A.1.

Tabela 2.1 prikazuje potrebno vreme izvršavanja algoritma ako se pretpostavi da jedna instrukcija traje jednu nanosekundu (10^{-9} sekundi). Ova procena je gruba, ali nije previše pogrešna i daje dobru procenu realnih vremena na današnjim računarima.

Tabela 2.1: Vreme izračunavanja.

$n/T(n)$	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3
10	0,003 μs	0,003 μs	0,01 μs	0,033 μs	0,1 μs	1 μs
100	0,007 μs	0,010 μs	0,1 μs	0,644 μs	10 μs	1 ms
1 000	0,010 μs	0,032 μs	1,0 μs	9,966 μs	1 ms	1 s
10 000	0,013 μs	0,1 μs	10 μs	130 μs	0,1 s	16,7 min
100 000	0,017 μs	0,316 μs	100 μs	1,67 ms	10 s	11,57 dan
1 000 000	0,020 μs	1 μs	1 ms	19,93 ms	16,7 min	31,7 god
10 000 000	0,023 μs	3,16 μs	10 ms	0,23 s	1,16 dan	3×10^5 god
100 000 000	0,027 μs	10 μs	0,1 s	2,66 s	115,7 dan	
1 000 000 000	0,030 μs	31,62 μs	1 s	29,9 s	31,7 god	

U narednom primeru je ilustrovano kako su vrednosti u ovoj tabeli izračunate.

Primer 2.2.1. *Neka je* $n = 10\,000$ *i* $T(n) = n^2$. *Broj izvršenih instrukcija je onda* $T(n) = T(10\,000) = 10\,000^2 = (10^4)^2 = 10^8$, *a vreme je* $10^8 \cdot 10^{-9}\text{s} = 0,1\text{s}$.

Neka je $n = 1\,000\,000 = 10^6$ *i* $T(n) = n \log_2 n$. *Važi da je* $\log_2(10^6) = 2\log_2(10^3) \approx 20$ *(jer je* $2^{10} = 1024 \approx 1000$*). Zato je* $T(n) = 10^6 \cdot 20 = 2 \cdot 10^7$, *a vreme je približno jednako* $2 \cdot 10^7 \cdot 10^{-9}\text{s} = 20 \cdot 10^{-3} s = 20\text{ms}$.

Algoritmi čija je složenost odozdo ograničena eksponencijalnom ili faktorijelskom funkcijom se smatraju neefikasnim.

$n/T(n)$	2^n	$n!$
10	1 μs	3,63 ms

$n/T(n)$	2^n	$n!$
20	1 ms	77,1 god
30	1 s	$8,4 \times 10^{15}$ god
40	18,3 min	
50	13 dan	
100	4×10^{13} god	

Možemo postaviti i pitanje koja dimenzija ulaza se otprilike može obraditi za određeno vreme. Odgovor je dat u narednoj tabeli.

t	n	$n \log n$	n^2	n^3	2^n	$n!$
1 ms	10^6	63 000	1 000	100	20	9
10 ms	$10 \cdot 10^6$	530 000	3 200	215	23	10
100 ms	$100 \cdot 10^6$	$4,5 \cdot 10^6$	10 000	465	27	11
1 s	10^9	$40 \cdot 10^6$	32 000	1 000	30	12
1 min	$60 \cdot 10^9$	$1,9 \cdot 10^9$	245 000	3 900	36	14

2.2.1.2 *Grafičko predstavljanje elementarnih funkcija*

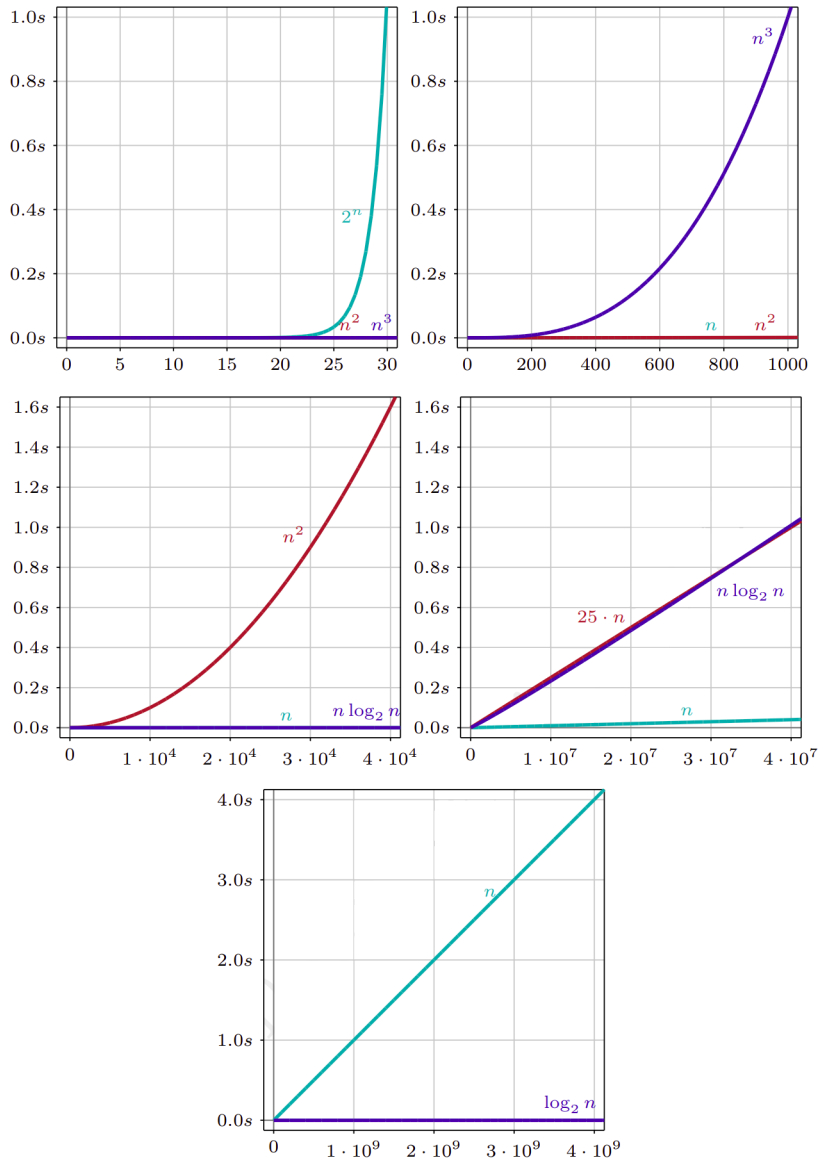
Pokušajmo sada da podatke iz datih tabela predstavimo grafički (slika 2.2). Usled jako velike razlike u brzinama rasta analiziraćemo samo "susedne" funkcije.

Na prvom grafiku na slici 2.2 prikazan je odnos vremena izvršavanja eksponencijalnih i polinomskih algoritama. Već za dimenziju 30 eksponencijalni algoritam zahteva oko jedne sekunde, dok je vreme izvršavanja algoritama polinomske složenosti praktično zanemarivo (čak i u slučaju polinoma većeg stepena, poput n^3).

Na drugom grafiku prikazan je odnos brzina rasta polinomskih algoritama. Povećanjem stepena prati jako veliki porast vremena. Algoritam koji zahteva n^3 instrukcija je mnogo sporiji nego onaj koji zahteva n^2 instrukcija (kubnom je već za problem dimenzije oko 1 000 potrebno oko jedne sekunde, dok kvadratni i linearni na tim dimenzijama posao obavljaju praktično momentalno).

Na trećem grafiku prikazan je odnos tri funkcije veoma česte u analizi algoritama: n^2, $n \log n$ i n. Sa grafika se može uočiti da na dimenzijama na kojima su kvadratnom algoritmu potrebne već sekunde, nema velike razlike između algoritama kojima je potrebno $n \log n$ i n instrukcija – oba skoro trenutno završavaju posao.

Da bi se razlika između takvih algoritama opazila, potrebno je da se dimenzija ulaza znatno poveća, što je i prikazano na četvrtom grafiku. Tek kada dimenzija ulaza



Slika 2.2: Grafička ilustracija vremena izvršavanja: na x-osi je dimenzija problema, a na y-osi je vreme u sekundama

dostigne milione, vidi se da je algoritam koji zahteva $n \log n$ instrukcija nešto sporiji. Sa grafika se vidi da svojim oblikom ta funkcija jako liči na linearnu (otuda i naziv "kvazilinearna" funkcija). Sa grafika se može videti i da razlika između linearne i kvazilinearne funkcije nije "drastična", jer se povećavanjem konstantnog faktora uz linearnu funkciju (faktora 25 na ovom grafiku) može desiti da na ulazima razmatrane dimenzije broj koraka bude veći nego kod osnovne kvazilinearne funkcije.

Na petom grafiku se vidi da je vreme izvršavanja algoritama kod kojih broj koraka logaritamski zavisi od dimenzije ulaza praktično zanemarivo (čak i za ogromne ulaze od milijardu elemenata). Treba imati na umu da je samo za učitavanje tolikog ulaza potrebno linearno vreme, pa prednost algoritama logaritamske složenosti dolazi tek kod problema kod kojih se nakon učitavanja podaci obrađuju veliki broj puta ili kod algoritama kod kojih nema potrebe za učitavanjem svih podataka.

Zaključak koji sledi iz proučavanja prikazanih grafika je da izmena algoritma takva da se broj koraka umesto nekom funkcijom iz našeg razmatranog niza funkcija izražava prethodnom u tom nizu, donosi drastično smanjenje vremena izvršavanja i mogućnost obrade mnogo većih ulaza. Izuzetak delom predstavljaju funkcije $n \log n$ i n, koje su veoma bliske i njihova razlika dolazi do izražaja tek kod jako velikih ulaza (ovo je jasno kada se pogleda količnik svake dve susedne funkcije – kod ove dve funkcije on je ubedljivo najmanji).

2.2.1.3 *Kombinacija elementarnih funkcija*

Prirodno se postavlja pitanje šta ako funkcija $T(n)$ koja određuje zavisnost između broja potrebnih koraka tj. vremena izračunavanja i dimenzije ulaza nije ovako jednostavna.

Primer 2.2.2. *Na primer, neka je* $T(n) = 2n^2+10n+8$*? Koliko je vreme potrebno za* $n = 10^5$*? Broj potrebnih koraka je* $2 \cdot (10^5)^2 + 10 \cdot 10^5 + 8$*, a potrebno vreme je* $2 \cdot 10^{10} \cdot 10^{-9}\text{s} + 10 \cdot 10^5 \cdot 10^{-9}\text{s} + 8 \cdot 10^{-9}\text{s} = 20\text{s} + 1\text{ms} + 8\text{ns} \approx 20\text{s}$*. Dakle, faktor* $2n^2$ *daje vreme* 20s*, faktor* $10n$ *daje vreme* 1ms*, a faktor* 8 *daje vreme* 8ns*. Očigledno je da je udeo vremena koje dolazi od članova* $10n$ *i* 8 *zanemariv u ukupnom zbiru. Nema, dakle, za velike vrednosti* n *nikakve značajne razlike između funkcija* $T(n) = 2n^2$*,* $T(n) = 2n^2 + 10n + 8$*, pa čak ni* $T(n) = 2n^2 + 1000n + 5000$*.*

Dakle, vreme izvršavanja algoritma je za velike ulaze praktično potpuno određeno dominantnim članom u funkciji koja opisuje zavisnost broja koraka od dimenzije ulaza.

Još jedno pitanje koje se nameće je koliko je značajan vodeći koeficijent uz taj dominantni član.

Primer 2.2.3. *Videli smo da nema praktično nikakve razlike između* $2n^2 + 10n + 8$ *i* $2n^2$. *Između* n^2, $2n^2$ *i* $5n^2$ *razlike ima, ali je ona mnogo manje značajna od razlike između, na primer,* n^2 *i* $1000n$. *Naime, već za* $n > 1000$ *algoritam koji zahteva* n^2 *koraka će biti sporiji od onog koji zahteva* $1000n$ *koraka i ta razlika će se povećavati sa porastom* n. *Za* $n = 10^6$, *prvi algoritam će biti 1000 puta sporiji nego drugi drugi.*

Dakle, red veličine vodećeg člana je mnogo značajniji za opis efikasnosti (za velike vrednosti n) nego što je koeficijent uz vodeći član. Razlika između, na primer, $2n^2$ i $5n^2$ se može nadomestiti nekim manjim optimizacijama ili boljim hardverom, dok se razlika između n^2 i n ne može nikako nadomestiti osim zamenom algoritma.

2.2.2 *Asimptotske oznake* O, Ω *i* Θ

Videli smo da nam je prilikom analize složenosti algoritama potrebno da grubo procenimo brzinu rasta funkcija koje opisuju zavisnost potrebnog vremena i memorije u odnosu na veličinu ulaza, tj. samo da odredimo dominantni član te funkcije, zanemarujući ostale članove i koeficijent uz dominantni član. Takva gruba procena nam može dati informaciju o tome da li se za neku veličinu ulaza vreme meri milisekundama, sekundama, satima, danima, godinama itd. Matematičke oznake O, Ω i Θ koriste se da bi se opisala brzina rasta funkcija[2].

- Oznaka $f(n) = O(g(n))$ se koristi da bi se iskazalo da funkcija $f(n)$ ne raste brže od funkcije $g(n)$, tj. da je $g(n)$ asimptotski *gornje ograničenje* brzine rasta funkcije $f(n)$.
- Oznaka $f(n) = \Omega(g(n))$ se koristi da bi se iskazalo da funkcija $f(n)$ ne raste sporije od funkcije $g(n)$, tj. da je $g(n)$ asimptotski *donje ograničenje* brzine rasta funkcije $f(n)$.
- Oznaka $f(n) = \Theta(g(n))$ se koristi da bi se iskazalo da funkcije $f(n)$ i $g(n)$ rastu istom brzinom.

Razmotrimo kako možemo definisati pojam $f(n) = O(g(n))$. Želimo da iskažemo da će počevši od neke veličine ulaza vrednosti funkcije f biti manje od vrednosti funkcije g. Pošto želimo da zanemarimo vrednost vodećeg koeficijenta ispred

[2]Pojmovi "veliko o", "veliko omega" i "veliko teta" mogu da se uvedu i za funkcije nad realnim brojevima, ali za potrebe analize složenosti izračunavanja dovoljne su verzije za funkcije nad prirodnim brojevima.

dominantnog člana, dopustićemo da se funkcija g skalira proizvoljnom konstantnom c tj. da se dopusti da vrednosti funkcije f budu manje od vrednosti funkcije $c \cdot g$. Takođe, pošto nas odnos funkcija f i $c \cdot g$ zanima samo za velike vrednosti n, zahtevaćemo da važi $f(n) \leq c \cdot g(n)$ za dovoljno velike vrednosti n, tj. za svako n veće od neke proizvoljne vrednosti n_0. Dakle, pojam $f(n) = O(g(n))$ se formalno može definisati na sledeći način.

Definicija 2.2.1. *Ako postoje pozitivna realna konstanta c i prirodan broj n_0 takvi da za funkcije f i g nad prirodnim brojevima važi*

$$f(n) \leq c \cdot g(n),$$

za svaki prirodan broj n veći od n_0 onda pišemo $f(n) = O(g(n))$ i čitamo "f je veliko o od g".

Naglasimo da $O(g(n))$ ne označava neku konkretnu funkciju, već klasu funkcija i uobičajeni zapis $f(n) = O(g(n))$ zapravo znači $f(n) \in O(g(n))$. Pored toga, kako O predstavlja relaciju, odnos između dve zadate funkcije f i g, a ne odnos između njihovih konkretnih vrednosti, pod zapisom $f(n) \in O(g(n))$, zapravo se misli $f \in O(g)$ (jer su f i g funkcije, a $f(n)$ i $g(n)$ njihove vrednosti).

Definicija 2.2.2. *Ako je $T(n)$ vreme izvršavanja algoritma A (čiji ulaz karakteriše prirodan broj n) i ako važi $T(n) = O(f(n))$, onda kažemo da je algoritam A* složenosti *ili* reda $O(f(n))$ *ili da algoritam A* pripada klasi $O(f(n))$.

Primer 2.2.4. *Može se dokazati da važi:*

- $7n^2 = O(n^2)$
  Tvrđenje važi jer za $c = 7$ (ali i za veće vrednosti c), važi $7n^2 \leq c \cdot n^2$ za sve vrednosti n.
- $7n^2 + 8n + 9 = O(n^2)$
  Za $c = 16$ važi $7n^2 + 8n + 9 \leq c \cdot n^2$ za sve vrednosti n veće od 2 (jer za takve vrednosti n važi $7n^2 \leq 7n^2$ i $8n \leq 8n^2$ i $9 \leq n^2$).
- $7n^2 + 8n + 9 = O(n^3)$
  Za $c = 1$, važi $7n^2 + 8n + 9 \leq c \cdot n^3$ za sve vrednosti n veće od 8.
- $2^n + n^2 = O(2^n)$
  Za $c = 2$ važi $2^n + n^2 \leq c \cdot 2^n$ za sve vrednosti n veće od 3 (jer za takve vrednosti n važi $2^n \leq 2^n$ i $n^2 \leq 2^n$; da važi $n^2 \leq 2^n$ za $n > 3$ može se dokazati, na primer, matematičkom indukcijom).

- $5 \cdot 3^n + 7 \cdot 2^n = O(3^n)$
  Za $c = 12$ *važi* $5 \cdot 3^n + 7 \cdot 2^n \leq c \cdot 3^n$ *za sve vrednosti* n *veće od* 0 *(jer za takve vrednosti* n *važi* $5 \cdot 3^n \leq 5 \cdot 3^n$ *i* $7 \cdot 2^n \leq 7 \cdot 3^n$*; da važi* $2^n \leq 3^n$ *za* $n > 0$ *može se dokazati, na primer, matematičkom indukcijom).*

Teorema 2.2.1. *Relacija* O *ima sledeća svojstva.*

- *Ako su* a *i* b *realni brojevi i* $a > 0$*, onda važi* $af(n) + b = O(f(n))$ *(tj. multiplikativne i aditivne konstante ne utiču na klasu kojoj funkcija pripada).*
- *Ako važi* $f_1(n) = O(g_1(n))$ *i* $f_2(n) = O(g_2(n))$*, onda važi i* $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$ *i* $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

Relacije Ω i Θ se definišu slično relaciji O.

Definicija 2.2.3. *Ako postoje pozitivna realna konstanta* c *i prirodan broj* n_0 *takvi da za funkcije* f *i* g *nad prirodnim brojevima važi*

$$c \cdot g(n) \leq f(n),$$

za svaki prirodan broj n *veći od* n_0*, onda pišemo* $f(n) = \Omega(g(n))$ *i čitamo "*f *je veliko omega od* g*".*

Definicija 2.2.4. *Ako postoje pozitivne realne konstante* c_1 *i* c_2 *i prirodan broj* n_0 *takvi da za funkcije* f *i* g *nad prirodnim brojevima važi*

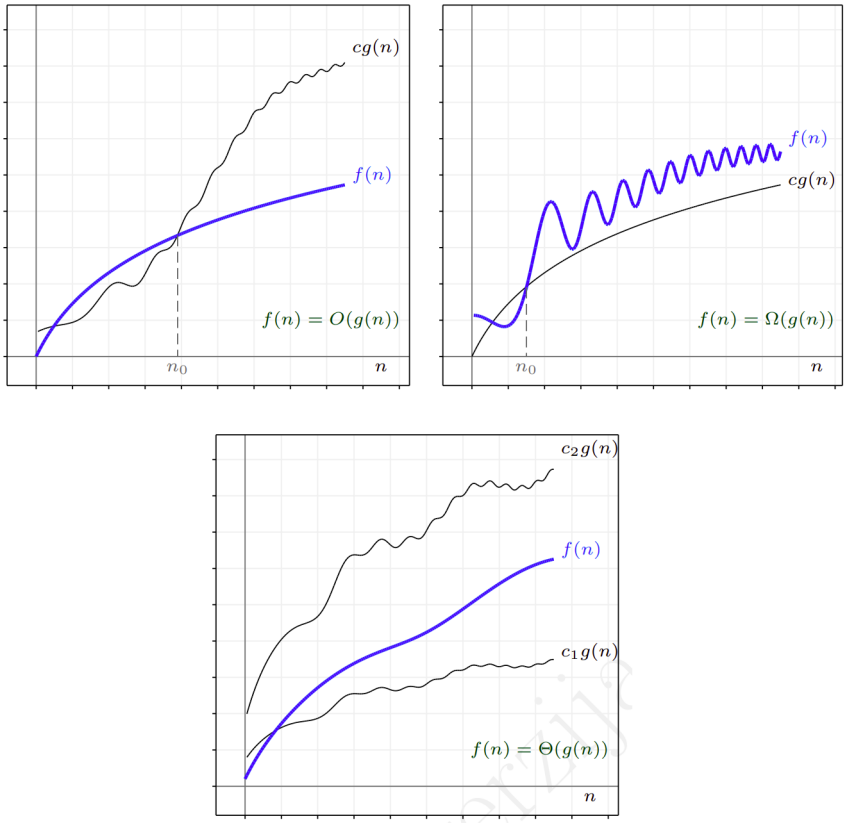
$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),$$

za svaki prirodan broj n *veći od* n_0*, onda pišemo* $f(n) = \Theta(g(n))$ *i čitamo "*f *je veliko teta od* g*".*

Analogno definiciji 2.2.2 definiše se kada algoritam A pripada klasi $\Omega(g(n))$ i kada pripada klasi $\Theta(g(n))$. Složenost algoritama najčešće se izražava u terminima O (što važi i za nastavak ove knjige).

Pojmovi *veliko o* (O), *veliko omega* (Ω) i *veliko teta* (Θ) ilustrovani su na slici 2.3.

Primer 2.2.5. *Može se dokazati da važi:*



Slika 2.3: Ilustracija pojmova “veliko o”, “veliko omega” i “veliko teta”

- $5 \cdot 2^n + 9 = \Theta(2^n)$

  Za $c_1 = 5$*, važi* $c_1 \cdot 2^n \leq 5 \cdot 2^n + 9$ *za sve vrednosti* n *veće od* 0*. Za* $c_2 = 6$*, važi* $5 \cdot 2^n + 9 \leq c_2 2^n$ *za sve vrednosti* n *veće od* 3*. Iz navedena dva tvrđenja sledi zadato tvrđenje.*

- $2^n + 2^n n = \Theta(2^n n)$

  Za $c_1 = 1$*, važi* $c_1 \cdot 2^n n \leq 2^n + 2^n n$ *za sve vrednosti* n *veće od* 0*. Za* $c_2 = 2$*, važi* $2^n + 2^n n \leq c_2 2^n n$ *za sve vrednosti* n *veće od* 0*. Iz navedena dva tvrđenja sledi zadato tvrđenje.*

Teorema 2.2.2. *Relacije* O*,* Ω *i* Θ *imaju sledeća svojstva.*

- *Ako važi* $f(n) = O(g(n))$ *i* $g(n) = O(h(n))$*, onda važi i* $f(n) = O(h(n))$ *(ovakva tranzitivnost važi i za* Θ *i* Ω*).*
- *Važi* $f(n) = \Theta(g(n))$ *akko važi* $f(n) = O(g(n))$ *i* $f(n) = \Omega(g(n))$*.*
- *Ako važi* $f(n) = O(g(n))$*, onda važi i* $g(n) = \Omega(f(n))$*.*
- *Ako važi* $f(n) = \Theta(g(n))$*, onda važi i* $f(n) = O(g(n))$ *i* $g(n) = O(f(n))$*.*
- *Ako važi* $f(n) = \Theta(g(n))$*, onda važi i* $g(n) = \Theta(f(n))$*.*

Informacija o složenosti algoritma u terminima Θ (koja daje i gornju i donju granicu) preciznija je nego informacija u terminima O (koja daje samo gornju granicu) ili informacija u terminima Ω (koja daje samo donju granicu). Međutim, obično je složenost algoritma jednostavnije iskazati u terminima O nego u terminima Θ. Štaviše, za neke algoritme složenost se ne može lako iskazati u terminima Θ. Na primer, ako za neke ulaze algoritam troši n, a za neke n^2 vremenskih jedinica, za taj algoritam se ne može reći ni da je reda $\Theta(n)$ ni reda $\Theta(n^2)$, ali jeste reda $O(n^2)$ (pa i, na primer, reda $O(n^3)$). Kada se kaže da algoritam pripada klasi $O(g(n))$ obično se podrazumeva da je g najmanja takva klasa (ili makar — najmanja za koju se to može dokazati). I O i Θ notacija se koriste i u analizi najgoreg slučaja i u analizi prosečnog slučaja.

Složenost algoritama u terminima Ω razmatra se ređe nego složenost u terminima Θ i O, ali ponekad takođe može da bude veoma važna. Složenost u terminima O daje gornju granicu za neku funkciju, a složenost u terminima Ω daje donju granicu. To se može razumeti i ovako: složenost u terminima O govori nam koliko je neki algoritam dobar ("asimptotski nije lošiji nego…"), a složenost u terminima Ω govori nam koliko je neki algoritam loš ("asimptotski nije bolji nego…"). Na primer, može

se pokazati da je prosečno vreme izvršavanja nekog algoritma za sortiranje reda $\Omega(n^2)$ i to govori da je taj algoritam loš (jer postoje algoritmi čije vreme izvršavanje pripada klasi $O(n \log n)$.

Lako se dokazuje da funkcija koja je konstantna (koliko god da je ta konstanta velika) pripada klasama $O(1)$, $\Omega(1)$ i $\Theta(1)$. Štaviše, svaka funkcija koja je ograničena odozgo nekom konstantom pripada klasama $O(1)$ i $\Theta(1)$.

Za algoritme složenosti $O(1)$ kažemo da imaju *konstantnu* složenost, za algoritme složenosti $O(n)$ kažemo da imaju *linearnu*, za $O(n \log n)$ da imaju *kvazilinearnu*, za $O(n^2)$ *kvadratnu*, za $O(n^3)$ *kubnu*, za $O(n^k)$ za neko k *polinomsku* (negde se kaže i *polinomijalnu*), za $O(a^n)$ za neko a *eksponencijalnu*, a za $O(\log n)$ *logaritamsku složenost*.

Priroda parametra klase složenosti zavisi od samog algoritma. Složenost izračunavanja neke funkcije može da zavisi i od više parametara: na primer, klasa $O(n)$ ima parametar n, dok klasa $O(2^{m+k})$ ima parametre m i k. Na primer, algoritam koji za n ulaznih tačaka proverava da li pripadaju unutrašnjosti nekog od m zadatih trouglova, koji kombinuje svaku tačku sa svakim trouglom, ima složenost $O(mn)$. Složenost funkcije koja sabira dva broja fiksne širine je konstantna tj. pripada klasi $O(1)$, a brojeva proizvoljne širine je $O(m+n)$, gde je m broj cifara prvog, a n broj cifara drugog broja. Pošto broj cifara broja odgovara veličini ulaza (tj. broju bitova potrebnih za zapis), složenost sabiranja je linearna u odnosu na broj bitova potrebnih za zapis ulaza. Sa druge strane, ako se složenost izrazi u odnosu na vrednosti brojeva koji se sabiraju, ona će biti logaritamska (jer broj cifara logaritamski zavisi od veličine brojeva). Kao što je već rečeno, potrebno je uvek eksplicitno navesti u odnosu na koju veličinu ili koje veličine se razmatra složenost algoritma.

Složenost izračunavanja je izuzetno važna i iz praktične i iz teorijske perspektive. Klasa složenosti P je klasa problema za koje postoje algoritmi koji su polinomske složenosti (u odnosu na ulaznu veličinu). Problem *SAT* (od engleskog *satisfiability*) je problem ispitivanja zadovoljivosti iskazne formule u konjunktivnoj normalnoj formi — za ulaznu formulu treba dati odgovor **da** ako je formula zadovoljiva, a odgovor **ne** inače. Trenutno nije poznato da li za ovaj problem postoji rešenje koje radi u polinomskom vremenu u odnosu na dužinu zadate formule. Nijedan trenutno poznat algoritam za SAT nema polinomsku složenost, svi imaju eksponencijalnu složenost. Pitanje da li SAT pripada klasi P jedno je od najvažnijih otvorenih pitanja u računarstvu. Pored klase P, izučavaju se i mnoge druge klase problema i algoritama na osnovu njihove prostorne i vremenske složenosti.

2.2.3 *Određivanje složenosti*

Određivanje (vremenske i prostorne) složenosti algoritama zasniva se na određivanju funkcije $T(n)$ tj. određivanje tačnog ili približnog broja instrukcija koje se izvršavaju i memorijskih jedinica koje se koriste. Tačno određivanje tih vrednosti najčešće je veoma teško ili nemoguće, te se obično koriste razna pojednostavljivanja. Na primer, često se smatra da sve pojedinačne instrukcije (osim poziva funkcija) troše jednako vremena. Ono što je važno je da takva pojednostavljivanja ne utiču na klasu složenosti kojoj algoritam pripada (jer, kao što je rečeno, konstantni aditivni i multiplikativni faktori ne utiču na red algoritma).

- Ukoliko se deo programa sastoji od nekoliko instrukcija bez grananja i petlji, onda se procenjuje da je njegovo vreme izvršavanja uvek isto, konstantno, te da pripada klasi $O(1)$.
- Ukoliko program ima dva dela, vremenske složenosti[3] $O(f)$ i $O(g)$, koji se izvršavaju jedan za drugim, ukupna složenost je[4] $O(f+g)$.

  To važi i u slučaju kada se u tim delovima javljaju rekurzivni pozivi (tada, na primer, vremenska složenost izražena u terminima vrednosti $f(n)$ može da zavisi od vremenske složenosti izražene u terminima vrednosti $f(n-1)$), ali tada efektivno izračunavanje složenosti zahteva korišćenje dodatnih tehnika (videti poglavlje 2.2.3.3).
- Ukoliko deo programa sadrži jedno grananje i ukoliko vreme izvršavanja jedne grane pripada klasi $O(f)$ a druge grane pripada klasi $O(g)$, onda je ukupno vreme izvršavanja tog dela programa ograničeno vremenom koje zahteva složenija grana, pa pripada klasi $O(\max(f,g))$, a ona je jednaka klasi $O(f+g)$.
- Ukoliko deo programa sadrži petlju koja se izvršava n puta, a vreme izvršavanja tela petlje je konstantno, onda ukupno vreme izvršavanja tog dela programa pripada klasi $O(n)$.

  Ukoliko deo programa sadrži petlju koja se izvršava za vrednosti i od 1 do n, a vreme izvršavanja tela petlje je $O(f(i))$, onda ukupno vreme izvršavanja tog dela programa pripada klasi $O(f(1)+f(2)+\ldots+f(n))$.

[3] Gde su f i g funkcije koje zavise od jednog ili više parametara programa.

[4] Na primer, ako je prvi deo složenosti $O(n)$ a drugi složenosti $O(n^2)$, onda je ukupna vremenska složenost $O(n+n^2)$, što je opet $O(n^2)$. Tada kažemo da drugi deo programa dominira u vremenu izvršavanja.

Ukoliko deo programa sadrži dvostruku petlju – jednu koja se izvršava m puta i, unutar nje, drugu koja se izvršava n puta i ukoliko je vreme izvršavanja tela unutrašnje petlje konstantno, onda ukupno vreme izvršavanja tog dela programa pripada klasi $O(m \cdot n)$. Ova pravila mogu se dalje uopštiti.

Analogno se računa vremenska složenost za druge vrste kombinovanja linearnog koda, grananja i petlji.

Analiza prostorne složenosti se vrši slično.

- Ukoliko deo programa ne sadrži pozive funkcija, dinamičku alokaciju, niti deklaracije objekata čija veličina zavisi od nekih parametera, onda je prostorna složenost tog dela programa konstanta tj. pripada klasi $O(1)$.
- Ukoliko neki deo programa vrši dinamičku alokaciju nekih n objekata veličine $O(1)$ – onda to doprinosi njegovoj prostornoj složenosti $O(n)$.
- Ukoliko program ima dva dela, prostorne složenosti $O(f)$ i $O(g)$, koji se izvršavaju jedan za drugim, ukupna složenost je $O(f + g)$.[5]
- Ukoliko se u programu javljaju pozivi funkcija, svaki poziv zauzima neki fiksni prostor na programskom steku (veličina tog prostora može biti ograničena jednom konstantom zajedničkom za sve funkcije) kao i dodatni prostor koji zauzimaju lokalne promenljive te funkcije (koje zauzimaju prostor na pozivnom steku ili su delom, kao na primer, elementi vektora, smešteni u hipu).

  Ukoliko se javljaju rekurzivni pozivi, onda u prostornu složenost ulazi maksimalni broj stek okvira koji se mogu naći na programskom steku tokom izvršavanja i čija je veličina konstantna,[6] ali i elementi lokalnih promenljivih koji se čuvaju u hip segmentu.

[5] Primetimo da ovo važi i ako prvi deo oslobađa prostor koji je zauzeo. Naime, $O(f + g)$ daje prostornu složenost u najgorem slučaju koja istovremeno ograničava odozgo i $O(f)$ i $O(g)$. U ovakvoj situaciji prostorna složenost ponaša se drugačije od vremenske: ako postoje dva dela programa koja se izvršavaju jedan za drugim, ukupno vreme izvršavanja (ne asimptotsko ograničenje nego konkretno utrošeno vreme) jednako je zbiru dva vremena izvršavanja, a ukupni zauzeti prostor jednak je maksimumu dva zauzeta prostora: jedan deo programa je koristio i oslobodio neki prostor a drugi deo programa je onda delom koristio isti taj prostor (na primer, na programskom steku).

[6] Treba imati na umu da je veličina stek okvira za svaku funkciju konstanta, ali ona može biti veoma velika, na primer – ako je u funkciji deklarisan niz velike dimenzije. Dodatno, treba imati na umu da stek okviri funkcija zauzimaju prostor na programskom steku, a prostor za njega je obično daleko manji od memorije u ostalim segmentima.

Ukoliko deo programa prostorne složenosti $O(f)$ poziva funkciju prostorne složenosti $O(g)$, onda ukupna prostorna složenost tog dela programa pripada klasi $O(f+g)$. Analogno se računa prostorna složenost za druge vrste kombinovanja koda.

2.2.3.1 *Iterativni algoritmi*

Prikažimo sada kroz nekoliko primera analizu složenosti iterativno implementiranih algoritama. Da bi mogla da se analizira složenost programa potrebno je vladati određenim matematičkim aparatom (na primer, izračunavanjem ili procenom određenih suma, postavljanjem i rešavanjem rekurentnih jednačina i slično). Neke matematičke tehnike koje su potrebne za analizu složenosti rezimirane su u dodatku 2.A.

Primer 2.2.6. *Izračunajmo vremensku složenost naredne funkcije koja ispisuje trougaoni deo tablice množenja:*

```
void mnozenje(int n)
{
  int i, j;
  if (n == 0)
    return;
  else {
    for (i = 1; i <= n; i++) {
      for (j = i; j <= n; j++)
        cout << i << "*" << j << " = " << i*j << "\t";
      cout << endl;
    }
  }
}
```

Za n jednako 5*, funkcija daje izlaz:*

```
1*1 = 1     1*2 = 2     1*3 = 3     1*4 = 4     1*5 = 5
2*2 = 4     2*3 = 6     2*4 = 8     2*5 = 10
3*3 = 9     3*4 = 12    3*5 = 15
4*4 = 16    4*5 = 20
5*5 = 25
```

Smatraćemo da se telo unutrašnje petlje u `else` *grani izvršava konstantno vreme* c*. Unutrašnja petlja ima* $n-i+1$ *iteracija, te je njeno vreme izvršavanja* $(n-i+1)c$*. Spoljašnja petlja ima* n *iteracija, a* i*-ta iteracija ima vreme izvršavanja* $(n-i+1)c$*. Ukupno vreme izvršavanja spoljašnje petlje je*

$$\sum_{i=1}^{n}(n-i+1)c = \sum_{i=1}^{n} ic = \frac{n(n+1)c}{2}.$$

Grana `if` *izvršava se konstantno vreme* c'*, pa je ukupna složenost funkcije* `mnozenje` $O\left(c' + \frac{n(n+1)c}{2}\right) = O(n(n+1)) = O(n^2)$.

Funkcija `mnozenje` *nema poziva drugih funkcija i ne koristi dinamičku alokaciju, niti objekte čija veličina zavisi od ulaznih parametara, te je njena prostorna složenost konstantna, tj. pripada redu* $O(1)$.

U narednim primerima ćemo se potruditi da pokrijemo oblike petlji koji se javljaju u velikom broju konkretnih algoritama i rešenja konkretnih zadataka. U primerima petlji koji slede, pretpostavlja se da kôd u telu petlji koji nije prikazan ne utiče na brojačke promenljive i ne menja granice petlji. Za razliku od prethodnog zadatka nećemo detaljno izračunavati broj instrukcija, već ćemo ga samo procenjivati na osnovu oblika petlji.

Primer 2.2.7. *Razmotrimo nekoliko čestih oblika petlje* `for`*.*

```
for (int i = 0; i < n; i++)
   // kod slozenosti O(1)
```

Složenost prethodne petlje je $O(n)$.

```
for (int i = m; i < n; i++)
   // kod slozenosti O(1)
```

Složenost prethodne petlje je $O(n-m)$.

```
for (int i = 0; i < n; i += 2)
   // kod slozenosti O(1)
```

Složenost prethodne petlje je $O(n)$. Pošto se petlja izvršava za parne vrednosti brojačke promenljive, telo petlje se izvršava oko $\frac{n}{2}$ puta i konstantni faktor je $\frac{1}{2}$, ali je složenost i dalje linearna.

```
for (int i = 0, j = n-1; i < j; i++, j--)
   // kod slozenosti O(1)
```

Složenost prethodne petlje je $O(n)$. Pokazivači se susreću približno na sredini opsega, a telo petlje se izvršava oko $\frac{n}{2}$ puta.

Moglo bi se pomisliti da je složenost svake petlje linearna, ali to nije uvek slučaj.

```
for (int i = 0; i < 10; i++)
   // kod slozenosti O(1)
```

Složenost prethodnog koda je $O(1)$. Iako je prisutna petlja, broj njenih izvršavanja je uvek 10 i ne zavisi ni od jednog parametara, pa ga možemo smatrati malom konstantom.

```
for (int i = 1; i*i <= n; i++)
   // kod slozenosti O(1)
```

Iako sadrži jednu petlju, složenost prethodnog koda nije $O(n)$, već $O(\sqrt{n})$. Naime, petlja se izvršava sve dok je $i^2 \leq n$ tj. dok je $i \leq \sqrt{n}$.

```
for (int i = 1; i < n; i *= 2)
   // kod slozenosti O(1)
```

Iako prethodni kod sadrži petlju, njegova složenost je $O(\log n)$, jer se vrednost promenljive `i` duplira u svakom koraku, sve dok ne prestigne graničnu vrednost n.

Primer 2.2.8. *Razmotrimo sada primere programa u kojima se javlja nekoliko petlji.*

```
for (int i = 0; i < m; i++)
    // kod slozenosti O(1)
```

```
for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Složenost prethodnih petlji je $O(m+n)$. Naime, telo prve petlja se izvrši m puta, a zatim telo druge petlje n puta, pa se tela obe petlje ukupno izvrše $m+n$ puta.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
       // kod slozenosti O(1)
```

Složenost prethodnih petlji je $O(n^2)$. Zaista, spoljašnja petlja se izvršava n, a u njenom telu se unutrašnja petlja izvršava n puta, pa se telo unutrašnje petlje izvrši tačno n^2 puta.

```
for (int i = 0; i < m; i++)
   for (int j = 0; j < n; j++)
       // kod slozenosti O(1)
```

Složenost prethodnih petlji je $O(mn)$. Zaista, spoljašnja petlja se izvršava m, a u njenom telu se unutrašnja petlja izvršava n puta, pa se telo unutrašnje petlje izvrši tačno mn puta.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
       // kod slozenosti O(1)
```

Složenost prethodnih petlji je $O(n^2)$. Broj izvršavanja tela unutrašnje petlje je $(n-1)+(n-2)+\ldots+2+1$, što je jednako $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$. Konstantni faktor je $\frac{1}{2}$, ali je složenost kvadratna. Do istog rezultata možemo doći ako shvatimo da u svakom koraku unutrašnje petlje par brojača određuje jednu kombinaciju brojeva od 0 do $n-1$. Zato broj izvršavanja tela odgovara broju dvočlanih kombinacija skupa od n elemenata, što je jednako $\binom{n}{2} = \frac{n(n-1)}{2}$.

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
       for (int k = 0; k < n; k++)
          // kod slozenosti O(1)
```

Složenost prethodnih petlji je prilično očigledno $O(n^3)$.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
       for (int k = j+1; k < n; k++)
          // kod slozenosti O(1)
```

Složenost prethodnih petlji je $O(n^3)$. *Najlakši način da se ovo zaključi je da se primeti da svakom izvršavanju tela odgovara jedna tročlana kombinacija elemenata skupa* 0, ..., $n-1$. *Pošto tročlanih kombinacija ima* $\binom{n}{3} = \frac{n(n-1)(n-2)}{3 \cdot 2 \cdot 1}$, *složenost je kubna, a konstantni faktor je* $\frac{1}{6}$.

```
for (int i = 0; i < n; i++)
   for (int j = i+1; j < n; j++)
       // kod slozenosti O(1)

for (int i = 0; i < n; i++)
    // kod slozenosti O(1)
```

Složenost prethodnih petlji je $O(n^2)$. *Naime, telo ugnežđenih petlji se izvrši* $\frac{n(n-1)}{2}$ *puta, a zatim telo druge petlje* n *puta, što je zapravo zanemarivo malo u odnosu na broj izvršavanja tela ugnežđenih petlji. Dakle, prvi deo koda dominira vremenom izvršavanja i složenost je* $O(n^2)$.

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j < i; j *= 2)
          // kod slozenosti O(1)
```

Složenost prethodnog koda je $O(n \log n)$. *Složenost unutrašnje petlje, za svako konkretno* `i` *je* $O(\log i)$, *pa je ukupna složenost otprilike jednaka* $\log 1 + \log 2 + \ldots + \log n$,

a za ovo se može pokazati da je $O(n \log n)$ (jasno je da je izraz manji ili jednak $n \log n$ jer je svaki sabirak manji ili jednak $\log n$, međutim, može se pokazati i da je zbir veći ili jednak od $\frac{n}{2} \log \frac{n}{2}$ (što je takođe $\Theta(n \log n)$), zanemarivanjem prvih $\frac{n}{2}$ sabiraka, nakon čega ostaju samo sabirci koji su veći ili jednaki $\log \frac{n}{2}$).

```
for (int i = n; i >= 1; i /= 2)
    for (int j = 1; j < i; j++)
          // kod slozenosti O(1)
```

Složenost prethodnog koda je $O(n)$. Naime, broj izvršavanja unutrašnje petlje je $n + \frac{n}{2} + \frac{n}{4} + \ldots$, za šta se lako može pokazati da je odozgo ograničeno sa $2n$.

```
int j = 0;
for (int i = 0; i < n; i++) {
    while (j < n && P[j]) {
       // kod slozenosti O(1)
       j++;
    }
   // kod slozenosti O(1)
}
```

Iako postoje ugnežđene petlje, složenost prethodne petlje je $O(n)$. Ključni detalj je to što unutrašnja petlja nema inicijalizaciju promenljive `j` na nulu i brojač `j` u unutrašnjoj petlji se sve vreme samo uvećava (isto kao i brojač `i` u spoljašnjoj petlji). Ukupan broj koraka je stoga ograničen sa $2n$.

```
int l = 0, d = n-1;
while (l < d) {
  do l++; while (l < d && P(a[l]));
  do d--; while (l < d && !P(a[d]));
  if (l < d)
     // kod slozenosti O(1)
}
```

Sličan kôd se, na primer, može sresti u algoritmu particionisanja niza tako da se elementi razdvoje na one koji ne zadovljavaju svojstvo `P` i na one koji zadovoljavaju

svojstvo P. I prethodni algoritam je složenosti $O(n)$ iako i on sadrži ugnežđene petlje. To ponovo možemo utvrditi amortizovanom analizom (jer ne znamo pojedinačni broj izvršavanja unutrašnjih petlji, ali možemo lako proceniti ukupan broj koraka koji se u njima napravi). Naime, promenljiva `l` se samo uvećava krenuvši od početka, a `d` se samo smanjuje krenuvši od kraja niza, dok se ne susretnu, što će se desiti u najviše n koraka.

Dakle, iako nam u većini slučajeva ugnežđenost petlji opisuje složenost algoritma, treba biti obazriv i kod analizirati pažljivije, da se složenost ne bi precenila.

2.2.3.2 Skrivena složenost

Često procenu složenosti grubo vršimo tako što analiziramo strukturu petlji u programu, zanemarući ostale operacije (obično za sve sem petlji smatramo da je $O(1)$). To može biti prilično varljivo, jer se u kodu mogu pozivati funkcije (bilo korisnički definisane, bilo bibliotečke) koje nisu konstantne složenosti. Još gore, i neki operatori mogu biti nekonstantne složenosti (obično linearne).

```
string rez = "";
for (char c : s)
   rez = c + rez;
```

Iako ima samo jednu petlju, prethodni fragment može biti složenosti $O(n^2)$, gde je n dužina niske `s`. Naime, dodavanje karaktera na početak niske u jeziku C++ može biti linearne složenosti $O(n)$, gde je n dužina niske (jer zahteva pomeranje narednih karaktera nadesno).

2.2.3.3 Rekurzivne funkcije

U narednom primeru analiziramo složenost rekurzivno definisane funkcije (mnogo više reči o ovome biće dato u delu udžbenika posvećenom rekurzivnim pristupima rešavanja problema).

Primer 2.2.9. *Izračunajmo vremensku složenost naredne rekurzivne funkcije koja izračunava faktorijel svog argumenta:*

```
unsigned faktorijel(unsigned n)
{
  if (n == 0)
```

```
    return 1;
  else
    return n * faktorijel(n-1);
}
```

Neka $T(n)$ označava broj instrukcija koje zahteva poziv funkcije `faktorijel` *za ulaznu vrednost n. Za $n = 0$, važi $T(n) = a$, gde je a neka konstanta. Za $n > 0$, važi $T(n) = b + T(n-1)$, gde je b neka konstanta (u tom slučaju izvršava se jedno poređenje, jedno oduzimanje, broj instrukcija koje zahteva funkcija* `faktorijel` *za argument $n-1$, jedno množenje i jedna naredba* `return`*). Dakle, $T(n) = b + T(n-1) = b + b + T(n-2) = ... = \underbrace{b + b + ... + b}_{n} + T(0) = bn + a$ Dakle, navedena funkcija ima linearnu vremensku složenost. Skrenimo pažnju i na to da vrednost faktorijela jako brzo raste i da zato veoma brzo dolazi do prekoračenja.*

Razmotrimo i prostornu složenost $S(n)$ funkcije `faktorijel`*. Jedna instanca funkcije* `faktorijel` *zauzima (na programskom steku) konstantan prostor c. Najviše prostora je zauzeto kada se izvršava rekurzivni poziv, te za prostornu složenost važi:*

$$S(0) = c$$

$$S(n) = S(n-1) + c$$

odakle se dobija da $S(n)$ pripada klasi $O(n)$.

Za izračunavanje složenosti komplikovanijih rekurzivnih funkcija potreban je matematički aparat za rešavanje rekurentnih jednačina*, u poglavlju 2.A.3.*

2.3 Popravljanje efikasnosti programa

Ključni savet za poboljšanje složenosti je to da računar radi samo ono što je neophodno da bi se dobio konačan rezultat. Kada se ta ideja malo detaljnije razradi, dobijamo sledeći niz veoma jednostavnih saveta koji nas često dovode do algoritama manje složenosti:

- Ne treba terati računar da vrši dugotrajna izračunavanja koja se jednostavno mogu izvršiti i “peške”, primenom matematičih uvida.

- Ne treba terati računar da više puta izračunava jedno te isto – rezultate izračunavanja moguće je upamtiti u memoriji, da se ne bi računali više puta.
- Ne treba terati računar da izračunava stvari koje nisu potrebne za dobijanje konačnog rešenja problema.
- Ne treba terati računar da ispituje slučajeve za koje se unapred može zaključiti da ne mogu voditi do traženog rešenja problema.
- Ako je to moguće, treba pripremiti ulazne podatke tako da se kasnije mogu efikasnije obraditi.
- Treba koristiti što efikasnije strukture podataka tj. podatke treba predstaviti na način koji je pogodan za problem koji se rešava.
- …

Ova lista saveta, naravno, nije iscrpna, ali iznenađujuće veliki broj značajnih efikasnih algoritama se suštinski zasniva na primeni baš ovih saveta.

Ukoliko performanse programa nisu zadovoljavajuće, pre bilo čega drugog treba razmotriti zamenu ključnih algoritama – algoritama koji dominantno utiču na složenost. Postoji niz važnih i elementarnih i naprednih tehnika koje mogu pomoći da se snizi složenost algoritama i njima ćemo se baviti u narednim poglavljima.

Ukoliko zamena algoritama efikasnijim ne uspeva tj. ukoliko se smatra da je asimptotsko ponašanje najbolje moguće, preostaje da se pokuša popravljanje efikasnosti snižavanjem konstantnih faktora u funkciji koja opisuje složenost (time se ne menja asimptotsko ponašanje, ali se ponekad može donekle smanjiti ukupno utrošeno vreme – na primer, program se može modifikovati tako da izvršava $6n + 3$ instrukcija umesto $7n + 2$). Ubrzavanje programa često zahteva specifična rešenja, ali postoje i neke ideje koje su primenljive u velikom broju slučajeva.

2.3.1 *Kompilatorske optimizacije*

Moderni kompilatori omogućavaju dodatno podešavanje za generisanje efikasnijeg koda. Iako značajno poboljšavaju performanse (često konstantno, ali nekada i asimptotski), napredne optimizacije nose određene izazove:

- **Otežana analiza:** Transformacija izvornog koda u asembler onemogućava efikasno korišćenje debagera i profajlera.
- **Varijacije u brzini:** Iako su ubrzanja često značajna, postoji rizik da neke tehnike u specifičnim slučajevima zapravo uspore program.

- **Vreme kompilacije:** Proces je znatno sporiji, pa se preporučuje tek u završnim fazama razvoja uz obavezno ponovno testiranje.

Kompilatori obično imaju mogućnost da se eksplicitno izabere neka tehnika optimizacije ili nivo optimizovanja (što uključuje ili ne uključuje više pojedinačnih tehnika). Na primer, za kompilator `gcc` nivo optimizacije se bira korišćenjem opcije `-O` iza koje može biti navedena oznaka stepena optimizacije:

Opcija	Opis optimizacije
`-O0`	Podrazumevani režim; koristi samo bazične tehnike bez značajnog uticaja na kod.
`-O1`	Balansira veličinu i brzinu; izbegava tehnike koje drastično produžavaju kompilaciju.
`-O2`	Fokus na brzini izvršavanja; ne dozvoljava povećanje memorijskog otiska programa.
`-O3`	Najagresivnija brzina; dozvoljava veći memorijski prostor zarad boljih performansi.
`-Os`	Optimizacija isključivo za smanjenje veličine izvršive datoteke.

Moderni kompilatori su, dakle, u stanju da samostalno primene izmene koda slične onima koje ćemo opisati u nastavku. Zato treba imati na umu pre svega opšti duh navedenih primera i način razmišljanja koji ih prati. Pored toga, nije dobro uvek se oslanjati na to da će kompilator sve optimizacije izvršiti automatski. Zadatak programera je da proveri da li se to događa i ako se ne događa, da program samostalno optimizuje, ako je to moguće. Takođe, treba imati u vidu da će se program u nekim situacijama prevoditi na drugoj platformi, pomoću drugačijeg prevodioca, za koji nije sigurno kakve će optimizacije vršiti. Zato, opšti savet može biti da u delovima programa za koje programer očekuje da mogu predstavljati usko grlo, programer od početnih faza piše program tako da izbegne neefikasnosti, ukoliko taj način pisanja programa ne dovodi do komplikovanog i nerazumljivog koda. Nakon inicijalnog razvoja korektnog programa, trebalo bi pažljivo izmeriti vreme izvršavanja programa i njegovih pojedinih delova (profilisanje), utvrditi koji kritični delovi koda nisu automatski optimizovani dovoljno kvalitetno i zatim pokušati njihovu optimizaciju samostalno.

2.3.2 *Optimizacija samo bitnih delova programa*

Programeri često pogrešno procenjuju koji delovi programa troše najviše resursa. Neproverene optimizacije mogu nepotrebno iskomplikovati kôd i otežati održavanje, a da pritom ne donesu realno poboljšanje. Zato je ključno:

- Koristiti profajler za precizno identifikovanje „uskih grla“.
- Meriti vreme izvršavanja na relevantnim test-primerima.
- Odbaciti optimizacije koje empirijski ne doprinose efikasnosti.

Uvek se treba fokusirati na delove programa koji zaista troše najviše vremena. Ako je uticaj nekog dela na ukupne performanse zanemarljiv, bolje je da on ostane jednostavan i lako razumljiv.

Čuvene su reči Donalda Knuta: „Programeri troše enormne količine vremena razmišljajući ili brinući o brzini nekritičnih delova svojih programa i to zapravo stvara jak negativan uticaj u fazama debagovanja i održavanja. Treba da zaboravimo na male efikasnosti, recimo 97% vremena dok programiramo: prerana optimizacija je koren svih zala. Ipak, ne treba da propustimo mogućnosti u preostalih kritičnih 3%.“

Prilikom optimizacije programa, ključno je razumeti kolika su realna očekivanja od ubrzanja pojedinačnih delova koda. Amdalov zakon nam pomaže da kvantifikujemo maksimalno moguće ubrzanje celog sistema na osnovu poboljšanja samo jednog njegovog dela.

Pretpostavimo da program ima segment koji možemo tehnički unaprediti. Definišimo sledeće parametre:

- p – udeo vremena izvršavanja koji se može ubrzati (procenat);
- $1 - p$ – udeo vremena izvršavanja koji se ne može ubrzati;
- s – faktor ubrzanja samog optimizovanog dela.

Ako je ukupno vreme izvršavanja pre optimizacije T, ono se može predstaviti kao zbir vremena koje troši deo koji se ne menja i deo koji se optimizuje:

$$T = (1 - p)T + pT$$

Nakon primene optimizacije, vreme izvršavanja dela p se smanjuje za faktor s, pa novo vreme T' iznosi:

$$T' = (1-p)T + \frac{pT}{s}$$

Ukupno ubrzanje programa (S) definiše se kao odnos starog i novog vremena izvršavanja:

$$S = \frac{T}{T'} = \frac{T}{(1-p)T + \frac{pT}{s}}$$

Skraćivanjem vrednosti T, dobijamo konačnu formulu **Amdalovog zakona**:

$$S = \frac{1}{(1-p) + \frac{p}{s}}$$

Da bismo bolje razumeli limite optimizacije, pogledajmo dva scenarija gde određenu funkciju ubrzavamo tačno 10 puta ($s = 10$).

Primer 2.3.1. *Pretpostavimo da određena funkcija zauzima samo 10% ukupnog vremena izvršavanja programa ($p = 0,1$). Ako tu funkciju ubrzamo 10 puta, ukupno ubrzanje će biti:*

$$S = \frac{1}{(1-0,1) + \frac{0,1}{10}} = \frac{1}{0,9 + 0,01} \approx 1,1$$

Zaključak: *Iako smo funkciju ubrzali drastično (10 puta), ceo program je postao brži samo za oko* ***10%****.*

Primer 2.3.2. *Pretpostavimo da ista ta funkcija zauzima čak 90% vremena izvršavanja ($p = 0,9$). Uz isto ubrzanje od 10 puta, rezultat je:*

$$S = \frac{1}{(1-0,9) + \frac{0,9}{10}} = \frac{1}{0,1 + 0,09} \approx 5,3$$

Zaključak: *U ovom slučaju, ubrzanje čitavog programa iznosi oko* ***5,3 puta****. Primetite da čak i kada ubrzamo 10 puta deo koji dominira izvršavanjem, ukupno ubrzanje je daleko manje od 10 puta.*

Iz ovih primera možemo izvući neke ključne zaključke.

1. **Prioriteti:** Optimizacija delova programa koji troše mali procenat vremena je praktično zanemariva.
2. **Fokus na "uska grla":** Optimizacija je najefikasnija kada se primeni na delove koda koji zauzimaju najveći deo vremena izvršavanja.
3. **Teorijski limit:** Čak i ako deo p ubrzate do beskonačnosti ($s \to \infty$), ukupno ubrzanje nikada ne može preći vrednost $1/(1-p)$. To znači da deo koji se ne može optimizovati predstavlja fiksnu barijeru za performanse.

Primer 2.3.3. *Ilustrujmo, na kraju, optimizaciju jednim Lojdovim*[7] *problemom: "Ribolovac je sakupio 1kg crva. Sakupljeni crvi imali su* 1% *suve materije i* 99% *vode. Sutra, nakon sušenja, crvi su imali* 95% *vode u sebi. Kolika je tada bila ukupna masa crva?" Na početku, suva materija činila je 1% od 1kg, tj. 10gr. Sutradan, vode je bilo 19 puta više od suve materije, tj. 190gr, pa je ukupna masa crva bila 200 grama.*

Ako bi program činile funkcije f i g i ako bi f pokrivala 99% *a funkcija g 1% vremena izvršavanja, glavni kandidat za optimizovanje bila bi, naravno, funkcija f. Ukoliko bi njen udeo u konačnom vremenu pao sa* 99% *na* 95%*, onda bi ukupno vreme izvršavanja programa bilo svedeno na samo* 20% *početnog.*

2.3.3 *Izbegavanje ponovljenih izračunavanja*

Identično skupo izračunavanje ne bi trebalo ponavljati. Umesto direktnog pozivanja zahtevnih funkcija više puta, bolje je sačuvati njihove vrednosti u pomoćnim promenljivama.

Primer 2.3.4. *Uvođenjem pomoćnih promenljivih u narednom primeru se smanjuje broj poziva trigonometrijskih funkcija (koje se izvršavaju relativno sporo).*

```
// Manje efikasno: cos i sin se racunaju po dva puta
x = x0*cos(0.01) - y0*sin(0.01);
y = x0*sin(0.01) + y0*cos(0.01);

// Efikasnije:
cs = cos(0.01); sn = sin(0.01);
x = x0*cs - y0*sn;
y = x0*sn + y0*cs;
```

[7] Samuel Loyd, 1841-1911

Moderni kompilatori (npr. `gcc` uz `-O1`) često sami vrše ovu optimizaciju. Međutim, to nije uvek trivijalno jer kompilator mora analizom potvrditi da funkcija nema propratnih efekata i da uvek vraća istu vrednost za isti argument.

Svako izračunavanje čiji se rezultat ne menja tokom iteracija treba izneti ispred petlje.

Primer 2.3.5. *Veoma slično prethodnom primeru u narednom kodu je još značajnije izračunavanje vrednosti trigonometrijskih funkcija izdvojiti van petlje i sačuvati u pomoćnim promenljivama.*

```
for (i=0; i < N; i++) {
  x = x0*cos(0.01) - y0*sin(0.01);
  y = x0*sin(0.01) + y0*cos(0.01);
  ...
}
```

Ovo je posebno važno kada se radi o funkcijama čija složenost zavisi od veličine podataka.

Primer 2.3.6. *U sledećem primeru, pozivanje `strlen(s)` za C-ovsku niska karatkera unutar uslova petlje drastično narušava performanse:*

```
// Losa praksa: slozenost O(n^2) jer se strlen poziva u
// svakoj iteraciji
for (i = 0; i < strlen(s); i++) { ... }

// Bolja praksa: slozenost O(n)
int len = strlen(s);
for (i = 0; i < len; i++) { ... }
      ...
```

Pošto `strlen` ima linearnu složenost $O(n)$, njenim pozivanjem unutar petlje ukupno vreme izvršavanja postaje kvadratno $O(n^2)$ (ovo je još jedan primer skrivene složenosti). Ovo je primer optimizacije koja poboljšava asimptotsku složenost programa.

Za iteraciju kroz nisku u jeziku C, najbolje je koristiti proveru terminirajućeg karaktera `\0`, čime se potpuno izbegava pozivanje funkcije `strlen`:

```
for (i = 0; s[i] != '\0'; i++)
  if (s[i] == c)
    ...
```

2.3.4 *Slabljenje operacija*

Ova tehnika podrazumeva zamenu računski zahtevnih („skupih“) operacija matematički ekvivalentnim, ali hardverski „jeftinijim“ operacijama koje se izvršavaju u manjem broju ciklusa procesora. Na primer, ukoliko je moguće dobro je izbegavati skupe trigonometrijske funkcije, korenovanje i slično.

Primer 2.3.7. *Čest primer je poređenje rastojanja između tačaka. Umesto direktnog poređenja* $\sqrt{x_1^2 + y_1^2} > \sqrt{x_2^2 + y_2^2}$*, daleko je efikasnije porediti kvadrate rastojanja:* $x_1^2 + y_1^2 > x_2^2 + y_2^2$*. Time se u potpunosti eliminiše poziv funkcije* `sqrt`*, koja je interno složena i zahteva mnogo procesorskog vremena.*

Slično, površinu trougla je mnogo bolje računati pomoću vektorskog proizvoda i determinante, nego pomoću Heronovog obrasca (koji uključuje korenovanje).

I operacije nad celim brojevima se mogu oslabiti. Na primer, množenja ili deljenja stepenom dvojke pomoću bitovskog šiftovanja (na primer, `x * 8` je isto što i `x << 3`, što je operacija koju procesor izvršava trenutno). Ipak, pošto kompilatori ovakve transformacije veoma često vrše u procesu optimizacije, ovakvim transformacijama treba pristupiti s rezervom, jer mogu narušiti čitljivost koda bez ikakvog doprinosa na brzinu izvršavanja.

Pored zamene skupih funkcija jefinijim, umesto “većih” tipova dobro je koristiti manje, poželjno celobrojne. Procesori su najbrži kada rade sa celobrojnim vrednostima (`int`). Prelazak sa decimalnih (`double`) na celobrojne tipove, gde god logika programa to dozvoljava, donosi značajna ubrzanja. Takođe, manji tipovi podataka bolje koriste keš memoriju, što smanjuje broj sporih pristupa glavnoj memoriji (RAM).

2.3.5 *Pretprocesiranje i upotreba unapred izračunatih vrednosti*

Osnovna ideja pretprocesiranja je prebacivanje tereta izračunavanja iz faze izvršavanja u fazu kompilacije ili inicijalizacije. Ako se u programu često koriste vrednosti iz konačnog i relativno malog skupa, efikasnije je izračunati ih unapred.

Na primer, ako nam je u nekom programu često potrebno da koristimo vrednosti korena brojeva od 1 do 100, umesto da ih stalno iznova izračunamo, možemo ih smestiti u konstantni niz (tabelu) u izvornom kodu i izračunati prilikom inicijalizacije programa. Ovo direktno menja složenu funkciju običnim pristupom memoriji, što je znatno brže. Ovaj pristup je klasičan primer kompromisa između vremena i prostora (engl. *time-memory tradeoff*). Žrtvujemo malu količinu memorije (prostor za niz) kako bismo drastično uštedeli na vremenu izvršavanja.

U jeziku C++, na primer, koristi se ključna reč `constexpr` koja primorava kompilator da izračuna vrednost nekog izraza već tokom samog prevođenja programa, tako da izvršivi program već sadrži gotov rezultat.

2.3.6 *Odabir programskog jezika*

Izbor programskog jezika je jedna od najvažnijih odluka u fazi dizajna softvera. Taj izbor direktno utiče na balans između brzine razvoja (vreme programera) i brzine izvršavanja (vreme procesora).

Danas se većina aplikacija piše u jezicima visokog nivoa kao što je Python. Idealan je za prototipove, analizu podataka, veštačku inteligenciju i skripte gde je bitno da se kôd napiše brzo i da bude čitljiv. Međutim, Python je interpretiran jezik, što ga čini znatno sporijim od kompiliranih jezika. Međutim, on često koristi biblioteke (poput `NumPy` ili `TensorFlow`) koje su interno napisane u C-u, čime se dobija "najbolje od oba sveta".

Kada su resursi ograničeni ili je brzina kritična, programeri se okreću jezicima C i C++. Koriste se za razvoj operativnih sistema, video-igara, sistema za upravalje bazama podataka i veb-servera. Ovi jezici omogućavaju direktno upravljanje memorijom i blisku interakciju sa hardverom, ali zahtevaju mnogo pažljivije programiranje jer greške često dovode do rušenja celog programa.

Iako se danas teži ka višim nivoima apstrakcije, pitanje jezika u kojem se piše kritični deo koda ostaje relevantno za sistemsko programiranje i razvoj softvera visokih performansi.

Iako savremeni kompilatori za C i C++ generišu izvanredan mašinski kôd koji retko koji čovek može nadmašiti, asembler i dalje ima svoje mesto. Nekada je pisanje kritičnih delova na asembleru bilo standard za izvlačenje maksimuma iz procesora. Danas, zahvaljujući naprednim optimizacijama kompilatora ova praksa se sve ređe koristi. Ipak, asembler ostaje neophodan u sistemskom programiranju, pisanju draj-

vera ili kada je potrebno iskoristiti specifične vektorske instrukcije procesora (npr. SIMD) koje kompilator možda ne koristi optimalno.

2.3.7 *Paralelizacija*

Od kako se hardverski razvoj više ne oslanja na drastično povećanje radne frekvencije jednog jezgra, već na dodavanje novih jezgara, paralelizacija je postala ključna za performanse. Savremeni procesori imaju 4, 8, 16 ili više jezgara. Ako je vaš algoritam sekvencijalan, on koristi samo delić snage računara. Paralelizacija omogućava da se veliki zadatak podeli na manje delove koji se izvršavaju istovremeno. Iako pisanje programa koji se mogu izvršavati na taj način zahteva specifične programerske veštine i poznavanje specijalizovanih biblioteka, ono je sve češće opcija.

2.3.8 *Popravljanje prostorne složenosti*

Za razliku od nekadašnjih računara, na savremenim računarima memorija obično nije kritični resurs. Optimizacije se obično usredsređuju na štednju vremena ili energije, a ne prostora. Ipak, postoje situacije u kojima je potrebno štedeti memoriju, na primer, onda kada program barata ogromnim količinama podataka, kada je s^am kôd programa veliki i zauzima značajan deo radne memorije (što je danas veoma retko) ili kada se program izvršava na nekom specifičnom uređaju koji ima malo memorije.

Naravno, ključni put ka optimizaciji je ponovo optimizacija asimptotske memorijske složenosti programa tj. zamena algoritama i struktura podataka. Ipak, u nekim slučajevima popravljanje konstantnog faktora može biti značajno (na primer, razlika samo u faktoru 2 može činiti razliku da li će izračunavanje biti uspešno ili neuspešno). Često ušteda memorije zahteva specifična rešenja, ali postoje i neke ideje koje su primenljive u velikom broju slučajeva:

- **Koristiti najmanje moguće tipove.** Za celobrojne podatke, umesto tipa `int` često je dovoljan tip `short` ili čak `char`. Za predstavljanje realnih brojeva, ukoliko preciznost nije kritična, može se, umesto tipa `double` koristiti tip `float`. Za predstavljanje logičkih vrednosti dovoljan je jedan bit a više takvih vrednosti može da se čuva u jednom bajtu (i da im se pristupa koristeći bitovske operatore).
- **Ne čuvati ono što može da se lako izračuna.** U prethodnom delu je, u slučaju da je kritična brzina, a ne prostor, dobro da se vrednosti koje se često

koriste u programu izračunaju unapred i uključe u izvorni kod programa. Ukoliko je kritična memorija, treba uraditi upravo suprotno i ne čuvati nikakve vrednosti koje se mogu izračunati u fazi izvršavanja.

Smanjenje prostorne složenosti često se postiže povećavanjem vremenske i obratno, ali postoje i mnoge situacije u kojima je dobrim rešenjem moguće popraviti istovremeno i vremensku i prostornu složenosti.

2.A Dodatak: matematičke osnove izračunavanja složenosti

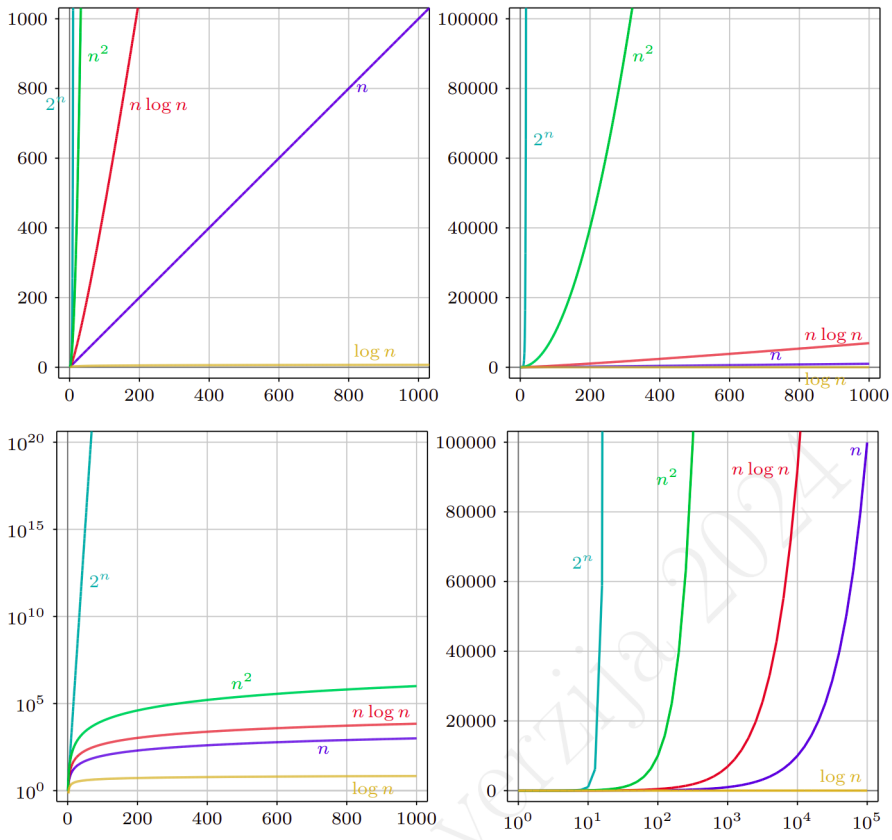
Da bismo mogla da se analizira složenost programa potrebno je vladati određenim matematičkim aparatom. U nastavku ćemo rezimirati osnovne matematičke pojmove koje ćemo koristiti u analizi složenosti algoritama.

2.A.1 Asimptotsko ponašanje elementarnih funkcija

Podsetimo se matematičkih svojstava elementarnih funkcija.

Priroda funkcija u matematici lakše se proučava i razume kada su dati njihovi grafici. U zavisnosti od relevantnih veličina ulaznih vrednosti ili od vrednosti funkcija (vrednosti funkcija mogu davati broj instrukcija ili vreme izvršavanja), nekad je pogodno za prikaz funkcija umesto uobičajenog Dekartovog sistema koristiti neku modifikovanu verziju. Na primer, možemo birati različite raspone x i y koordinata koje će biti prikazane na grafiku (u slučaju da je raspon neke od koordinata mnogo veći od one druge, odnos jediničnih podeoka se podešava tako da grafik i dalje zadrži skoro kvadratni oblik). Drugo, jedna od osa (pa i obe) može biti logaritamski transformisana — tada se od jedne do druge istaknute tačke skale ne dolazi sabiranjem sa određenom konstantom (1 u uobičajenom Dekartovom sistemu), već množenjem sa određenom konstantom (na primer, 2 ili 10). Na slici 2.4 prikazani su grafici funkcija koje se javljaju često u analizi složenosti.

Sa svih grafika jasno je uočljiv relativni odnos brzina rasta ovih funkcija. Nesporno je da među prikazanim funkcijama eksponencijalna funkcija raste najbrže, da je naredna po brzini rasta funkcija n^2, zatim $n \log n$, potom n i na kraju $\log n$ koja raste jako sporo. Sa druge strane, može se primetiti da se zaključci o međusobnom odnosu brzina rasta moraju izvoditi veoma pažljivo, jer mogu biti različiti u zavisnosti od odabranog raspona x i y koordinata. Na primer, na prvom grafiku i x i y su odabrani tako da idu do 1000, a na drugom je odabrano da x ide do 1000, a y do 100000. Sa



Slika 2.4: Grafici funkcija koji se često javljaju u analizi složenosti, prikazani na različitim skalama

prvog grafika se može steći utisak da je funkcija $n \log n$ po brzini rasta negde tačno između n i n^2, dok se sa drugog već taj odnos može preciznije proceniti i vidi se da funkcija n^2 raste mnogo brže nego $n \log n$ i n. Prvi prikazani grafik zapravo je samo mali deo drugog (dobija se tako što se y osa "saseče" pri samom dnu, na vrednosti 1000). Stoga prilikom analize algoritama grafici treba da se podese tako da x-osa pokazuje dimenzije ulaza zaista relevantne za problem koji se rešava, a da se na y osi prikazuje realna procena broja instrukcija tj. vremena izvršavanja koje se u realnom kontekstu dopuštaju algoritmu. Ako prikazane funkcije mere broj instrukcija, onda je drugi grafik (u tom smislu) mnogo bolji od prvog, jer se na prvom ne prikazuju izvršavanja nakon 1000 instrukcija, što je za današnje računare zanemarivo malo. Možemo reći da bi se još bolji grafik dobio kada bi se broj instrukcija povećao na milione, pa čak i milijarde, jer su današnji računari u stanju da izvrše milijarde instrukcija u nekoliko sekundi. Grafici sa logaritamskim skalama mnogo bolje mogu da predstave funkcije i njihove odnose, ali treba steći dobar osećaj za to kako ih treba čitati.

2.A.2 *Sumiranje*

Tokom analize algoritama često imamo potrebu da izračunamo određene konačne sume. Rezimirajmo ih kroz nekoliko najznačajnijih primera.

2.A.2.1 *Aritmetički niz*

Gausu se pripisuje da je još kao dete izračunao da je

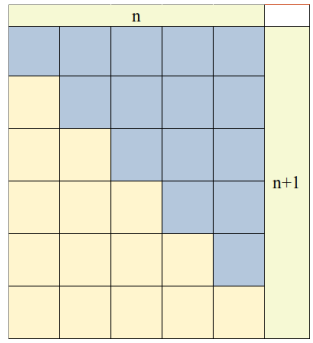
$$1 + 2 + ... + n = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}.$$

Ako je n paran broj, u ovom zbiru se krije $n/2$ parova čiji je zbir $n + 1$: prvi i poslednji, drugi i pretposlednji, itd. U istom duhu, ako označimo traženi zbir sa S, onda je $2S = S + S = (1 + 2 + ... + n) + (n + (n - 1) + ... + 1) = n \cdot (n + 1)$.

Nekada slika govori više od reči (videti sliku 2.5).

Na osnovu prethodnog, jednostavno se izvodi da je zbir prvih n članova aritmetičkog niza čiji je prvi član a, a razlika između svaka dva susedna člana jednaka r jednaka

$$a+(a+r)+(a+2r)+...+(a+(n-1)\cdot r) = \sum_{k=0}^{n-1}(a+k\cdot r) = n\cdot a + r\frac{n(n-1)}{2}.$$



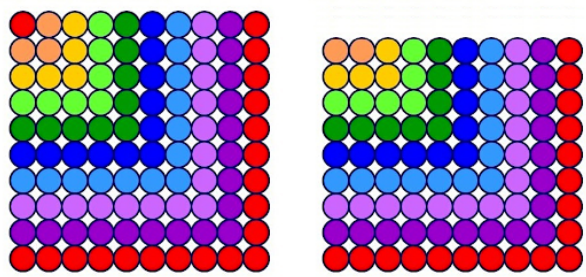
Slika 2.5: Gausova formula

Intuicija nam opet govori da se ovde krije $\frac{n}{2}$ parova (ako je n paran) čiji je zbir $a_0 + a_{n-1}$, što opet dovodi do formule $\frac{n}{2}(2a + (n-1) \cdot r)$.

Jedan važan aritmetički niz je niz neparnih brojeva. Važi da je $1 + 3 + 5 + ... + (2k-1) = \frac{k}{2} \cdot (1 + (2k-1)) = k^2$.

Izračunavanje zbira uzastopnih parnih brojeva se lako svodi na zbir uzastopnih brojeva. $2 + 4 + 6 + ... 2k = 2(1 + 2 + 3 + ... k) = k(k+1)$.

Ponovo slika govori više od reči (videti sliku 2.6).



Slika 2.6: Zbir neparnih brojeva od 1 do $2k-1$ i zbir parnih brojeva od 2 do $2k$

2.A.2.2 Geometrijski niz i red

Izvedimo formulu za zbir prvih n članova geometrijskog niza kome je prvi član a a količnik svaka dva uzastopna člana $q \neq 1$. Obeležimo traženu sumu sa S.

$$S = a + a \cdot q^1 + a \cdot q^2 + ... + a \cdot q^{n-2} + a \cdot q^{n-1} = \sum_{k=0}^{n-1} a \cdot q^k.$$

Ako levu i desnu stranu prethodne jednakosti pomnožimo sa $1 - q$ dobijamo jednakost:

$S \cdot (1 - q) = a \cdot (1 - q) + a \cdot q \cdot (1 - q) + a \cdot q^2 \cdot (1 - q) + ... + a \cdot q^{n-2} \cdot (1 - q) + a \cdot q^{n-1} \cdot (1 - q)$

Izvršimo množenja na desnoj strani jednakosti:

$$S \cdot (1-q) = a - a \cdot q + a \cdot q - a \cdot q^2 + a \cdot q^2 - a \cdot q^3 + ... + a \cdot q^{n-2} - a \cdot q^{n-1} + a \cdot q^{n-1} - a \cdot q^n$$

Sređivanjem poslednjeg izraza dobijamo $S \cdot (1 - q) = a - a \cdot q^n$. Prema tome, pošto je $q \neq 1$, važi

$$S = a \cdot \frac{1 - q^n}{1 - q}.$$

Za $|q| < 1$ geometrijski red konvergira i suma mu je $\frac{a}{1-q}$.

Nama će često biti korisni slučajevi $q = 2$ i $q = 1/2$.

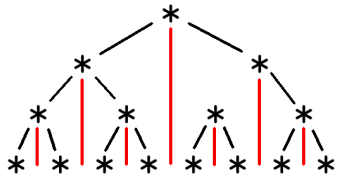
Na osnovu prethodne formule, za $a = 1$ i $q = 2$, važi da je $1 + 2 + ... + 2^{n-1} = 2^n - 1$. Ova formula ima interesantno tumačenje. Suma sa leve strane predstavlja ukupan broj čvorova na prvih n nivoa potpunog binarnog drveta (videti sliku 2.7), dok je izraz sa desne strane za jedan manji od broja čvorova na narednom nivou $n + 1$. Dakle, na svakom narednom nivou binarnog drveta ima jedan čvor više nego što je čvorova na svim prethodnim nivoima drveta.

Za $a = 1$ i $q = 1/2$ dobijamo da je

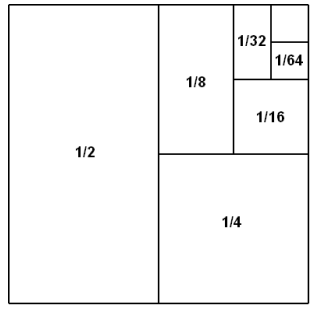
$$1 + 1/2 + ... + (1/2)^{n-1} = \frac{1 - (1/2)^n}{1 - 1/2} = 2 - (1/2)^{n-1}.$$

Sa porastom n ova vrednost se približava vrednosti 2 (svakako je njome ograničena odozgo).

Opet slika govori više od reči (videti sliku 2.8).



Slika 2.7: Broj čvorova na najnižem nivou binarnog drveta za jedan je veći od ukupnog broja čvorova na prethodnim nivoima



Slika 2.8: Zbir geometrijskog reda za $a = 1/2$, $q = 1/2$

2.A.2.3 *Stepene sume*

Prikažimo kako možemo izračunati sumu kvadrata svih prirodnih brojeva od 1 do n. Važi da je

$$(k+1)^3 - k^3 = k^3 + 3k^2 + 3k + 1 - k^3 = 3k^2 + 3k + 1.$$

Zato je

$$\begin{aligned} 2^3 - 1^3 &= 3 \cdot 1^2 + 3 \cdot 1 + 1 \\ 3^3 - 2^3 &= 3 \cdot 2^2 + 3 \cdot 2 + 1 \\ &\ldots \\ (n+1)^3 - n^3 &= 3 \cdot n^2 + 3 \cdot n + 1 \end{aligned}$$

Sabiranjem prethodnih jednakosti dobijamo

$$(n+1)^3 - 1 = 3 \cdot (1^2 + \ldots + n^2) + 3 \cdot (1 + \ldots + n) + (1 + \ldots + 1)$$

tj.

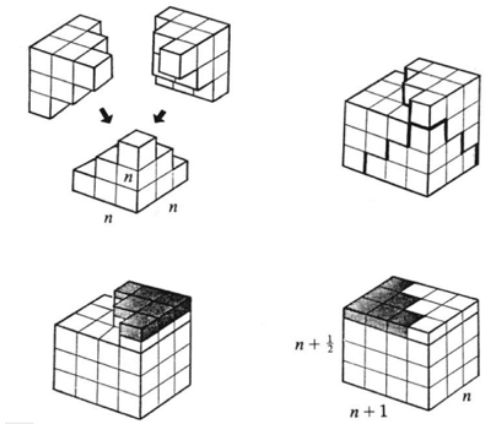
$$3 \cdot \sum_{k=1}^{n} k^2 = (n+1)^3 - 1 - 3\sum_{k=1}^{n} k - \sum_{k=1}^{n} 1.$$

Na osnovu ranije izvedenih formula za zbir aritmetičkog niza, sledi da je

$$\begin{aligned} \sum_{k=1}^{n} k^2 &= \frac{1}{3} \cdot \left(n^3 + 3n^2 + 3n - 3\frac{n(n+1)}{2} - n \right) \\ &= \frac{n \cdot (2n+1) \cdot (n+1)}{6} \\ &= \frac{1}{3} \left(n \cdot (n + \frac{1}{2}) \cdot (n+1) \right). \end{aligned}$$

Dakle, suma kvadrata prirodnih brojeva od 1 do n se asimptotski ponaša kao $\frac{n^3}{3}$.

Opet slika govori više od reči (videti sliku 2.9).



Slika 2.9: Zbir kvadrata svih prirodnih brojeva od 1 do n

Potpuno analogno, sumiranjem izraza $(k+1)^4 - k^4$ od 1 do n i primenom do sada izvedenih formula može se pokazati da je

$$1^3 + 2^3 + ... + n^3 = \sum_{k=1}^{n} k^3 = \frac{(n(n+1))^2}{4}.$$

Ovo tvrđenje, poznato kao Nikomahova teorema pokazuje da je zbir kubova svih prirodnih brojeva od 1 do n jednak kvadratu zbira svih prirodnih brojeva od 1 do n i asimptotski se ponaša kao $\frac{n^4}{4}$.

Opet slika govori više od reči (videti sliku 2.10).

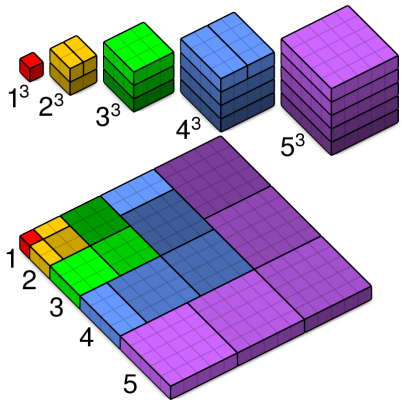
Pošto će nas u analizi algoritama najčešće zanimati samo asimptotsko ponašanje funkcija, najvažnije je zapamtiti da se suma n p-tih stepena svih prirodnih brojeva od 1 do n asimptotski ponaša kao $\frac{n^{p+1}}{p+1}$.

2.A.2.4 Suma logaritama

Procenimo asimptotsko ponašanje sume logaritama $\log 1 + \log 2 + ... + \log n$.

Pošto je logaritam rastuća funkcija, svaki od n članova ovog zbira ograničen je odozgo vrednošću $\log n$. Zato važi da ovaj zbir pripada klasi $O(n \log n)$. Dokažimo da ovo ograničenje nije previše grubo. Važi da je

$$\log 1 + \log 2 + ... + \log n \geq \log (n/2) + \log (n/2 + 1) + ... + \log n,$$



Slika 2.10: Zbir kubova svih prirodnih brojeva od 1 do n

jer je prvih $n/2$ članova koji su iz sume izostavljeni sigurno nenegativno. Pošto je logaritam rastuća funkcija (za osnovu veću od 1), svi sabirci u ovom zbiru su veći ili jednaki $\log(n/2)$, pa je

$$\log(n/2)+\log(n/2+1)+\ldots+\log n \geq \log(n/2)+\log(n/2)+\ldots+\log(n/2).$$

Zbir na desnoj strani ima $n/2$ istih sabiraka i jednak je $(n/2)\cdot\log(n/2)$. Stoga je početni zbir logaritama ograničen i odozdo i odozgo funkcijama koje su $\Theta(n\log n)$, pa je i sam $\Theta(n\log n)$.

Još jedan način da se ovo pokaže koji se često sreće u literaturi je sledeći. Zbir logaritama, jednak je logaritmu proizvoda, pa zapravo ovde računamo vrednost $\log 1\cdot\ldots\cdot n = \log n!$. Po Stirlingovoj formuli $n!$ se ponaša kao $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$. Zato se $\log n!$ ponaša kao $n\log n - n + O(\log n)$, pa je ukupan zbir $\Theta(n\log n)$.

Pokušajmo da uopštimo nekoliko prethodnih rezultata. Često se asimptotska ocena složenosti svodi na izračunavanje nekog zbira $T(n) = c+f(1)+f(2)+\ldots+f(n)$, tako da se određivanje asimptotskog ponašanja svodi na sumiranje.

- Zbir n sabiraka reda $1+2+\ldots+n$ ima vrednost $n(n+1)/2$, koja je reda $\Theta(n^2)$. To je samo duplo manje od vrednosti zbira $n+\ldots+n$, koji se sastoji od n sabiraka i ima vrednost n^2.

- Zbir n sabiraka $\log 1 + ... + \log n$ ima vrednost koja se ponaša kao $n \log n - n$, što je asimptotski isto kao vrednost zbira $\log n + ... + \log n$ koji ima n sabiraka i vrednost $n \log n$.
- Slično, zbir $1^2 + ... + n^2$ ima vrednost $n(n+1)(2n+1)/6$, što je $\Theta(n^3)$ i samo je oko tri puta manje od zbira $n^2 + ... + n^2$ koji ima n sabiraka i vrednost n^3.

Iako ovakve generalizacije prete da budu neprecizne, sa malom dozom rezerve se može proceniti da algoritmi u kojima se n puta primenjuje neka operacija složenosti $\Theta(f(k))$ imaju složenost $\Theta(n \cdot f(n))$, čak i kada se operacija u svakom koraku primenjuje nad podacima koji su se za $O(1)$ povećali u odnosu na prethodni korak i samo u krajnjoj instanci imamo $\Theta(n)$ podataka.

2.A.2.5 Primena diferencijalnog i integralnog računa u izračunavanju i oceni suma

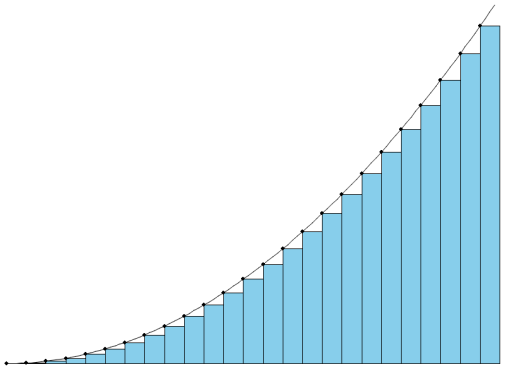
Za izračunavanje i ocenu suma mogu se koristiti i izvodi i integrali. Primetimo da je neodređeni integral funkcije x^p funkcija $\frac{x^{p+1}}{p+1}$, a da se zbir p-tih stepena svih prirodnih brojeva od 1 do n asimptotski ponaša upravo kao $\frac{n^{p+1}}{p+1}$. To nije slučajnost. Razmotrimo monotono rastuću funkciju f (takva je funkcija $f(x) = x^n$) na domenu $x \geq 0$. Suma $\sum_{k=0}^{n-1} f(k)$ se može predstaviti kao površina n pravougaonika (svakome je širina 1, a visina k-tog je $f(k)$). Na slici je prikazana suma prvih 25 potpunih kvadrata. Sa slike je vidi da je ta suma (zbir površina pravougaonika) bliska površini ispod krive $f(x) = x^2$ koja se može izračunati (tj. čije se asimptotsko ponašanje može proceniti) primenom određenih integrala.

Ilustrujmo ovo i malo preciznije. Površina ispod krive $f(x)$ za $k \leq x \leq k+1$ jednaka je određenom integralu $\int_k^{k+1} f(x)\, dx$. Pošto je funkcija rastuća, ta površina je veća od površine pravougaonika čija je visina $f(k)$ i širina 1, a manja od površine pravougaonika čija je visina $f(k+1)$ i širina 1 tj. važi

$$f(k) \leq \int_k^{k+1} f(x)\, dx \leq f(k+1).$$

Zato je

$$\sum_{k=0}^{n-1} f(k) \leq \int_0^n f(x)\, dx \leq \sum_{k=0}^{n-1} f(k+1).$$



Slika 2.11: Procena sume određenim integralom

Gornju granicu sume možemo direktno pročitati iz prve nejednakosti. Pošto je

$$\sum_{k=0}^{n-1} f(k+1) = \sum_{k=0}^{n-1} f(k) - f(0) + f(n),$$

iz druge nejednakosti sledi i donja granica, pa važi:

$$\int_0^n f(x)\,dx + f(0) - f(n) \leq \sum_{k=0}^{n-1} f(k) \leq \int_0^n f(x)\,dx.$$

Slučaj kada je $f(x)$ monotono opadajuća funkcija za $x \geq 0$ se obrađuje analogno (samo je potrebno umesto $\leq$ koristiti $\geq$).

Na primer, ponašanje sume $\sum_{k=0}^{n-1} ka^k = a + 2a^2 + 3a^3 + ... + (n-1)a^{n-1}$, možemo proceniti izračunavanjem određenog integrala $\int_0^n xa^x\,dx$.

On se jednostavno može izračunati parcijalnom integracijom za $u = x$ (pa je $du = dx$) i $dv = a^x\,dx$, odakle je $v = \frac{a^x}{\ln a}$. Zato je

$$\int_0^n xa^x\,dx = \int_0^n u\,dv = (uv)\Big|_0^n - \int_0^n v\,du = \frac{na^n}{\ln a} - \frac{\int_0^n a^x\,dx}{\ln a} = \frac{na^n}{\ln a} - \frac{(a^n-1)}{\ln^2 a},$$

pa se ova funkcija asimptotski ponaša kao na^n.

Ovu sumu je moguće izračunati i egzaktno, primenom diferenciranja. Naime, važi da je

$$\sum_{k=0}^{n-1} x^k = 1 + x + ... + x^{n-1} = \frac{x^n - 1}{x - 1}.$$

Diferenciranjem obe strane po x dobijamo

$$1 + 2x + 3x^2 + ... + (n-1)x^{n-2} = \frac{nx^{n-1}(x-1) - (x^n - 1)}{(x-1)^2}.$$

Množenjem sa x dobijamo

$$\sum_{k=0}^{n-1} kx^k = x + 2x^2 + 3x^3 + ... + (n-1)x^{n-1} = x\frac{nx^{n-1}(x-1) - (x^n - 1)}{(x-1)^2}.$$

Na primer, za $x = 2$ dobijamo da je $\sum_{k=0}^{n-1} k2^k = (n-2) \cdot 2^n + 2$.

Napomenimo i da se ta suma veoma jednostavno može izračunati i sasvim elementarnim tehnikama. Neka je $S_x(n) = \sum_{k=0}^{n-1} kx^k$. Tada množenjem ove jednakosti sa x, oduzimanjem dobijenog rezultata od polazne jednakosti i primenom formule za zbir geometrijskog niza dobijamo

$$\begin{aligned}
S_x(n) &= x^1 + 2x^2 + 3x^3 + ... + (n-2)x^{n-2} + (n-1)x^{n-1} \\
x \cdot S_x(n) &= x^2 + 2x^3 + 3x^4 + ... + (n-2)x^{n-1} + (n-1)x^n \\
S_x(n) - xS_x(n) &= x^1 + x^2 + x^3 + ... + x^{n-1} + (n-1)x^n \\
(1-x)S_x(n) &= x(x^0 + ... + x^{n-2}) + (n-1)x^n = x\frac{1 - x^{n-1}}{1 - x} + (n-1)x^n
\end{aligned}$$

Odatle sledi da je

$$S_x(n) = \frac{x - x^n}{(1-x)^2} + (n-1)x^n$$

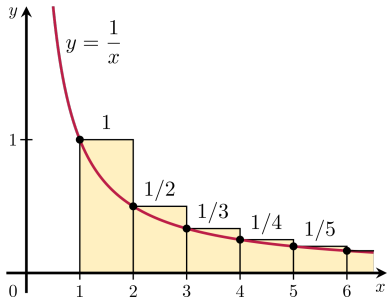
Lako se pokazuje da su dobijeni izrazi i izraz dobijen primenom diferenciranja jednaki. Zamenom vrednosti $x = 2$ ponovo dobijamo da je $S_2(n) = \sum_{k=0}^{n-1} k2^k = (n-2) \cdot 2^n + 2$.

2.A.2.6 *Harmonijski red*

Razmotrimo zbir $H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n}$. Ovaj zbir možemo elegantno a prilično dobro približno proceniti.

Prvo pokažimo da $H(n)$ teži beskonačnosti kako n teži beskonačnosti. Važi da je $H(n) > 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + ...$ Naime, važi da je $\frac{1}{3} > \frac{1}{4}$, zatim da je $\frac{1}{5} > \frac{1}{8}$, $\frac{1}{6} > \frac{1}{8}$ i $\frac{1}{7} > \frac{1}{8}$ itd. Međutim, važi da je $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$, da je $\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} = \frac{1}{2}$ itd. Zato je desni zbir jednak $1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + ...$, a ovaj zbir jasno divergira tj. teži beskonačnosti sa povećanjem broja sabiraka.

Slično bi se moglo dokazati i korišćenjem procene sume određenim integralom. Broj $H(n)$ se može proceniti kao $\int_1^{n+1} \frac{1}{x} \mathrm{d}x$, koji je jednak $(\ln x)\Big|_1^{n+1}$ tj. $\ln(n+1)$.

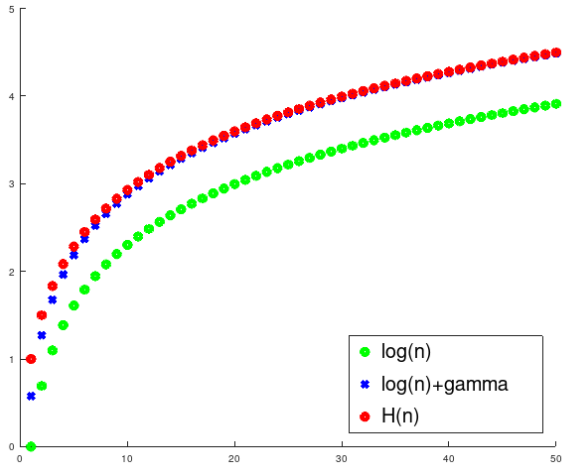


Slika 2.12: Procena zbira harmonijskog reda $H(n)$ određenim integralom

Na grafiku funkcije $H(n)$ može se uočiti da ona veoma sporo teži beskonačnosti i može se primetiti veoma sličan oblik grafiku funkcije $\ln n$. Zaista, dokazuje se da razlika između funkcija $H(n)$ i $\ln n$ veoma brzo konvergira i teži konstanti γ koja je poznata pod imenom Ojler-Maskeronijeva konstanta i čija je vrednost približno jednaka $\gamma = 0.57722$. Ništa se ne bi promenilo i da se posmatra odnos $H(n)$ sa funkcijom $\ln(n+1)$ (jer sa povećanjem n razlika između $\ln(n+1)$ i $\ln n$ veoma brzo teži nuli), osim što bi procena bila malo preciznija za male vrednosti n.

2.A.3 *Rekurentne jednačine*

Kod rekurzivnih funkcija, vreme $T(n)$ potrebno za izračunavanje vrednosti funkcije za ulaz veličine n može se izraziti kao zbir vremena izračunavanja za sve rekurzivne pozive i vremena potrebnog za pripremu rekurzivnih poziva i objedinjavanje rezultata. Tako se, obično jednostavno, može zapisati *linearna rekurentna relacija* oblika:



Slika 2.13: Zbir harmonijskog reda $H(n)$ i odnos sa logaritamskom funkcijom ln n)

$$T(n) = a_1 T(n-1) + ... + a_k T(n-k) + r(n), \quad n \geq k,$$

gde rekurzivna funkcija za ulaz veličine n vrši po nekoliko (a_i su prirodni brojevi) rekurzivnih poziva za ulaze veličina $n-1$, ..., $n-k$, dok je $r(n)$ vreme potrebno za pripremu poziva i objedinjavanje rezultata. Ovakvom rekurentnom vezom i početnim članovima niza $T(0), T(1), ..., T(k-1)$ u potpunosti je određen niz T.

U nekim slučajevima, iz rekurentne relacije i početnih elemenata može se eksplicitno izračunati nepoznati opšti član niza $T(n)$. U nekim slučajevima eksplicitno rešavanje jednačine nije moguće, ali se može izračunati asimptotsko ponašanje niza $T(n)$.

Navedimo nekoliko najčešćih oblika primene rekurentnih jednačina na analizu rekurzivnih algoritama.

U prvoj grupi se problem svodi na problem dimenzije koja je tačno za jedan manja od dimenzije polaznog problema.

- *Jednačina*: $T(n) = T(n-1) + O(1), T(0) = O(1)$. *Primer*: Traženje minimuma niza. *Rešenje*: $O(n)$.
- *Jednačina*: $T(n) = T(n-1) + O(\log n), T(0) = O(1)$. *Primer*: Formiranje balansiranog binarnog drveta. *Rešenje*: $O(n \log n)$.

- *Jednačina*: $T(n) = T(n-1) + O(n), T(0) = O(1)$. *Primer*: Sortiranje selekcijom. *Rešenje*: $O(n^2)$.

U drugoj grupi se problem svodi na dva (ili više) problema čija je dimenzija za jedan ili dva manja od dimenzije polaznog problema. To obično dovodi do eksponencijalne složenosti.

- *Jednačina*: $T(n) = 2T(n-1) + O(1), T(0) = O(1)$. *Primer*: Hanojske kule. *Rešenje*: $O(2^n)$
- *Jednačina*: $T(n) = T(n-1) + T(n-2) + O(1), T(0) = O(1)$. *Primer*: Fibonačijevi brojevi. *Rešenje*: $O(2^n)$

U narednoj grupi se problem svodi na jedan (ili više) potproblema koji su značajno manje dimenzije od polaznog (obično su bar duplo manji). Ovo dovodi do polinomijalne složenosti, pa često i do veoma efikasnih rešenja.

- *Jednačina*: $T(n) = T(n/2) + O(1), T(0) = O(1)$. *Primer*: Binarna pretraga sortiranog niza. *Rešenje*: $O(\log n)$.
- *Jednačina*: $T(n) = T(n/2) + O(n), T(0) = O(1)$. *Primer*: Pronalaženje medijane (središnjeg elementa) niza. *Rešenje*: $O(n)$.
- *Jednačina*: $T(n) = 2T(n/2) + O(1), T(0) = O(1)$. *Primer*: Obilazak potpunog binarnog drveta. *Rešenje*: $O(n)$.
- *Jednačina*: $T(n) = 2T(n/2) + O(n), T(0) = O(1)$. *Primer*: Sortiranje objedinjavanjem. *Rešenje*: $O(n \log n)$.

Ako su granice u samim jednačinama egzaktne, skoro u svim prethodno nabrojanim jednačinama dato rešenje nije samo gornje ograničenje, već je asimptotski egzaktno. Na primer, rešenje jednačine $T(n) = 2T(n/2) + \Theta(n), T(1) = \Theta(1)$ ima rešenje $T(n) = \Theta(n \log n)$. Izuzetak je primer Fibonačijevog niza gde ponašanje jeste eksponencijalno, ali osnova nije 2, već zlatni presek $(1 + \sqrt{5})/2$.

2.A.3.1 *Homogena rekurentna jednačina prvog reda*

Razmotrimo jednačinu oblika

$$T(n) = aT(n-1),$$

za $n > 0$, pri čemu je data vrednost $T(0) = c$. Jednostavno se pokazuje da je rešenje ove jednačine geometrijski niz $T(n) = ca^n$.

Ovo rešenje govori da je složenost rekurzivne funkcije za parametar n koja vrši više od jednog rekurzivnog poziva dimenzije $n - 1$, čak i kada se ostale operacije zanemare, eksponencijalna (na primer, ako je $T(n) = 2T(n - 1)$, onda $T(n)$ pripada klasi $O(2^n)$). Zato takva rešenja treba izbegavati kada god je to moguće.

2.A.3.2 ***Homogena rekurentna jednačina drugog reda***

Razmotrimo jednačinu oblika

$$T(n) = aT(n - 1) + bT(n - 2),$$

za $n > 1$, pri čemu su date vrednosti za $T(0) = c_0$ i $T(1) = c_1$.

Ukoliko nisu navedeni početni uslovi, jednačina ima više rešenja. Zaista, ukoliko nizovi $T_1(n)$ i $T_2(n)$ zadovoljavaju jednačinu, tada jednačinu zadovoljava i njihova proizvoljna linearna kombinacija $T(n) = \alpha T_1(n) + \beta T_2(n)$:

$$\begin{aligned} T(n) &= \alpha T_1(n) + \beta T_2(n) \\ &= \alpha(aT_1(n-1) + bT_1(n-2)) + \beta(aT_2(n-1) + bT_2(n-2)) \\ &= a(\alpha T_1(n-1) + \beta T_2(n-1)) + b(\alpha T_1(n-2) + \beta T_2(n-2)) \\ &= aT(n-1) + bT(n-2)\ . \end{aligned}$$

S obzirom na to da i nula niz (niz čiji su svi elementi nule) trivijalno zadovoljava jednačinu, skup rešenja čini vektorski prostor.

Razmotrimo funkcije oblika t^n i pokušajmo da proverimo da li postoji broj t takav da t^n bude rešenje date jednačine. Za takvu vrednost bi važilo:

$$t^n = a \cdot t^{n-1} + b \cdot t^{n-2},$$

odnosno, posle množenja sa t^2 i deljenja sa t^n:

$$t^2 = at + b.$$

Dakle, da bi t^n bilo rešenje jednačine, potrebno je da t bude koren navedene kvadratne jednačine, koja se naziva *karakteristična jednačina* za *homogenu rekurentnu jednačinu drugog reda*.

Ako su t_1 i t_2 različiti koreni ove jednačine (oni mogu biti i kompleksne vrednosti), može se dokazati da opšte rešenje $T(n)$ može biti izraženo kao linearna kombinacija baznih funkcija t_1^n i t_2^n, tj. da je oblika

$$T(n) = \alpha \cdot t_1^n + \beta \cdot t_2^n \ ,$$

tj. da ove dve funkcije čine bazu pomenutog vektorskog prostora rešenja. Ako se želi pronaći ono rešenje koje zadovoljava zadate početne uslove (tj. zadovoljava date vrednosti $T(0) = c_0$ i $T(1) = c_1$), onda se vrednosti koeficijenata α i β mogu dobiti rešavanjem sistema dobijenog za $n = 0$ i $n = 1$, tj. rešavanjem sistema jednačina $c_0 = \alpha + \beta$, $c_1 = \alpha \cdot t_1 + \beta \cdot t_2$.

U slučaju da je t_1 dvostruko rešenje karakteristične jednačine, može se dokazati da opšte rešenje $T(n)$ može biti izraženo kao linearna kombinacija baznih funkcija t_1^n i $n \cdot t_1^n$, tj. da je oblika

$$T(n) = \alpha \cdot t_1^n + \beta \cdot n \cdot t_1^n \ .$$

Koeficijenti α i β koji određuju partikularno rešenje koje zadovoljava početne uslove, takođe se dobijaju rešavanjem sistema za $n = 0$ i $n = 1$.

Iz prethodnog sledi da je složenost rekurzivnih funkcija koje dovode do rekurentnih jednačina drugog reda eksponencijalna, pa je često dovoljno odrediti osnovu te eksponecijalne funkcije (rešavanjem karakteristične jednačine), dok se određivanje konkretnih vrednosti koeficijenata α i β manje značajno. Takve rekurzivne funkcije je dobro izbegavati i rešenja formulisati, ukoliko je moguće, tako da im složenost bude polinomska (dobar primer kako se ovo može uraditi čini Fibonačijev niz). Postoje, međutim, i rekurzivni algoritmi eksponencijalne složenosti koji rešavaju probleme za koje se ne zna da li imaju rešenja polinomske složenosti.

Primer 2.A.1. *Neka za vreme izvršavanja $T(n)$ algoritma A (gde n određuje ulaznu vrednost za algoritam) važi $T(n+2) = 2T(n+1) + 3T(n)$ (za $n \geq 1$) i $T(1) = 5, T(2) = 19$. Složenost algoritma A može se izračunati na sledeći način. Karakteristična jednačina za navedenu homogenu rekurentnu vezu je*

$$t^2 = 2t + 3,$$

i njeni koreni su $t_1 = 3$ *i* $t_2 = -1$. *Opšti član niza* $T(n)$ *može biti izražen u obliku*

$$T(n) = \alpha \cdot t_1^n + \beta \cdot t_2^n \ .$$

tj.

$$T(n) = \alpha \cdot 3^n + \beta \cdot (-1)^n \ .$$

Iz $T(1) = 5, T(2) = 19$ *dobija se sistem:*

$$\begin{aligned} \alpha \cdot 3 + \beta \cdot (-1) &= 5 \\ \alpha \cdot 9 + \beta \cdot 1 &= 19 \end{aligned}$$

čije je rešenje $(\alpha, \beta) = (2, 1)$, *pa je* $T(n) = 2 \cdot 3^n + (-1)^n$, *odakle sledi da je* $T(n) = O(3^n)$.

Primetimo da je za računanje asimptotske složenosti bilo dovoljno izračunati korene karakteristične jednačine. Veći od njih je 3 *i to je dovoljno da se zaključi da složenost pripada klasi* $O(3^n)$.

Primer 2.A.2. *Neka za vreme izvršavanja* $T(n)$ *algoritma* A *(gde* n *određuje ulaznu vrednost za algoritam) važi* $T(n+2) = 4T(n+1) - 4T(n)$ *(za* $n \geq 1$*) i* $T(1) = 6, T(2) = 20$. *Složenost algoritma* A *može se izračunati na sledeći način. Karakteristična jednačina za navedenu homogenu rekurentnu vezu je*

$$t^2 = 4t - r$$

i njen dvostruki koren je $t_1 = 2$. *Opšti član niza* $T(n)$ *može biti izražen u obliku*

$$T(n) = \alpha \cdot 2^n + \beta \cdot n \cdot 2^n \ .$$

Iz $T(1) = 6, T(2) = 20$ *dobija se sistem*

$$\begin{aligned} \alpha \cdot 2 + \beta \cdot 2 &= 6 \\ \alpha \cdot 4 + \beta \cdot 8 &= 20 \end{aligned}$$

čije je rešenje $(\alpha, \beta) = (1, 2)$*, pa je* $T(n) = 2^n + 2 \cdot n \cdot 2^n$*, odakle sledi da je* $T(n) = O(n2^n)$.

U ovom primeru se u rekurentnoj jednačini javlja i jedan negativan koeficijent (koeficijent -4 *uz vrednost* $T(n)$*) ali, kako smo videli, to ne utiče na opšti način rešavanja. Iako se jednačine sa negativnim koeficijentima ne mogu javiti direktnim modelovanjem rekurzivnih funkcija (jer se u njima ukupno vreme dobija sabiranjem vremena koje se utroši na svaki rekurzivni poziv), one mogu nastati tokom procesa rešavanja i uvođenjem raznih smena u originalne jednačine.*

2.A.3.3 *Homogena rekurentna jednačina reda* k

Homogena rekurentna jednačina reda k (gde k može da bude i veće od 2) je jednačina oblika:

$$T(n) = a_1 \cdot T(n-1) + a_2 \cdot T(n-2) + ... + a_k \cdot T(n-k),$$

za $n > k-1$, pri čemu su date vrednosti za $T(0) = c_0, T(1) = c_1, ..., T(k-1) = c_{k-1}$.

Tehnike prikazane na homogenoj jednačini drugog reda, lako se uopštavaju na jednačinu proizvoljnog reda k. Karakteristična jednačina navedene jednačine je:

$$t^k = a_1 \cdot t^{k-1} + a_2 \cdot t^{k-2} + ... + a_k.$$

Ako su rešenja t_1, t_2, ..., t_k sva različita, onda je opšte rešenje polazne jednačine oblika:

$$T(n) = \alpha_1 \cdot t_1^n + \alpha_2 \cdot t_2^n + ... + \alpha_k \cdot t_k^n,$$

pri čemu se koeficijenti α_i mogu dobiti iz početnih uslova (kada se u navedeno opšte rešenje za n uvrste vrednosti 0, 1, ..., $k-1$).

Ukoliko je neko rešenje t_1 dvostruko, onda u opštem rešenju figurišu bazne funkcije t_1^n i $n \cdot t_1^n$. Ukoliko je neko rešenje t_1 trostruko, onda u opštem rešenju figurišu bazne funkcije t_1^n, $n \cdot t_1^n$, $n^2 \cdot t_1^n$, itd.

2.A.3.4 *Nehomogena rekurentna jednačina prvog reda*

Razmotrimo jednačinu oblika

$$T(n) = aT(n-1) + b,$$

za $n > 0$, pri čemu je data vrednost $T(0) = c$. Jedan način rešavanja ovog tipa jednačina je svođenje na homogenu jednačinu drugog reda. Iz $T(1) = aT(0) + b$, sledi da je $T(1) = ac + b$. Iz

$$T(n) = aT(n-1) + b$$

$T(n+1) = aT(n) + b$,

sledi $T(n+1) - T(n) = (aT(n) + b) - (aT(n-1) + b) = aT(n) - aT(n-1)$ i, dalje, $T(n+1) = (a+1)T(n) - aT(n-1)$, za $n > 0$. Rešenje novodobijene homogene jednačine se može dobiti na gore opisani način (jer su poznate i početne vrednosti $T(0) = c$ i $T(1) = ac + b$).

Primer 2.A.3. *Razmotrimo nehomogenu rekurentnu jednačina prvog reda* $T(0) = 0$ *i* $T(n) = 2T(n-1)+1$ *(i* $T(1) = 2T(0)+1 = 1$ *(ova jednačina se dobija kada se analizira broj prebacivanja diskova u igri Hanojske kule). Iz* $T(n) - 2T(n-1) = 1 = T(n-1) - 2T(n-2)$ *(za* $n > 1$*) sledi* $T(n) = 3T(n-1) - 2T(n-2)$. *Ova jednačina je homogena jednačina drugog reda i ona može biti rešena na ranije opisan način. Karakteristična jednačina je* $t^2 = 3t - 2$ *i njeni koreni su* 2 *i* 1. *Iz sistema*

$$\begin{aligned} \alpha \cdot 1 + \beta \cdot 1 &= 0 \\ \alpha \cdot 2 + \beta \cdot 1 &= 1 \end{aligned}$$

sledi $\alpha = 1$ *i* $\beta = -1$, *pa je*

$$T(n) = 1 \cdot 2^n + (-1) \cdot 1^n = 2^n - 1\ .$$

2.A.3.5 *Nehomogena rekurentna jednačina reda* k

Nehomogena rekurentna jednačina reda k ($k > 0$) oblika:

$$T(n) = a_1 \cdot T(n-1) + a_2 \cdot T(n-2) + ... + a_k \cdot T(n-k) + c,$$

za $n > k-1$, pri čemu su date vrednosti za $T(0) = c_0, T(1) = c_1, ..., T(k-1) = c_{k-1}$, može se rešiti svođenjem na homogenu rekurentnu jednačinu reda $k+1$, analogno gore opisanom slučaju za $k = 1$.

Primer 2.A.4. *Razmotrimo jednačinu* $T(0) = T(1) = c_1$, *gde je* c_1 *neka konstanta i* $T(n) = T(n-1) + T(n-2) + c_2$, *gde je* c_2 *neka konstanta (ova jednačina se dobija prilikom analize složenosti izračunavanja elemenata Fibonačijevog niza). Iz* $T(n) = T(n-1) + T(n-2) + c_2$ *i* $T(n+1) = T(n) + T(n-1) + c_2$, *sledi* $T(n+1) = 2T(n) - T(n-2)$. *Karakteristična jednačina ove jednačine je* $t^3 = 2t^2 - 1$ *i njeni koreni su* 1, $\frac{1+\sqrt{5}}{2}$ *i* $\frac{1-\sqrt{5}}{2}$, *pa je opšte rešenje oblika*

$$T(n) = a \cdot 1^n + b \cdot \left(\tfrac{1+\sqrt{5}}{2}\right)^n + c \cdot \left(\tfrac{1-\sqrt{5}}{2}\right)^n,$$

odakle sledi da je $T(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)$.

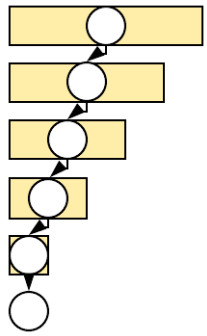
2.A.3.6 *Najznačajniji primeri*

Jednačina $T(n) = T(n-1) + O(1)$**,** $T(0) = O(1)$

Ova jednačina se javlja, na primer, kod rekurzivnog određivanja zbira serije brojeva, određivanja minimuma/maksimuma, linearne pretrage i slično.

Jednačina je dovoljno jednostavna da bi se rešila odmotavanjem. Preciznosti radi, predstavimo je u obliku $T(n) = T(n-1) + c$ i $T(0) = d$, nakon odmotavanja dobijamo da važi:

$$\begin{aligned} T(n) &= T(n-1) + c \\ &= T(n-2) + c + c \\ &= T(n-3) + c + c + c \\ &= ... \\ &= T(0) + n \cdot c \\ &= d + n \cdot c \\ &= O(n). \end{aligned}$$



Slika 2.14: Drvo poziva u slučaju $T(n) = T(n-1) + O(1)$, $T(0) = O(1)$ za $n = 5$. Pravougaonik označava dimenziju ulaza, a elipsa količinu posla koji se obavlja u tom čvoru.

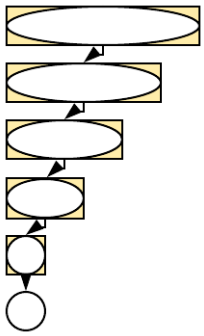
Jednačina $T(n) = T(n-1) + O(n)$**,** $T(0) = O(1)$

Ova jednačina se javlja, na primer, kod naivnih algoritama sortiranja niza, poput sortiranja selekcijom ili sortiranja umetanjem. Na primer, kod sortiranja selekcijom se pronalazi pozicija najmanjeg elementa u nizu (za šta je potrebno $O(n)$ koraka), on se dovodi na prvom mesto u nizu i zatim se prelazi na sortiranje ostatka niza, koji sadrži $n-1$ elemenata.

Kod jednačine $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$ slično dobijamo n sabiraka koji su svi $O(n)$ tako da je ukupna suma $O(n^2)$. Postavlja se pitanje da li je ova granica egzaktna tj. da li je moguće da je složenost manja od izvedenog gornjeg ograničenja. Pretpostavimo da je $T(n) = T(n-1) + cn$ i da je $T(0) = d$. Tada je

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + c(n-1) + cn \\ &= ... \\ &= T(0) + c(1 + ... + n) \\ &= d + cn(n+1)/2, \end{aligned}$$

tako da je $T(n) = \Theta(n^2)$.



Slika 2.15: Drvo poziva u slučaju $T(n) = T(n-1) + O(n)$, $T(0) = O(1)$ za $n = 5$

2.A.3.7 *Jednačina* $T(n) = T(n-1) + O(\log n)$, $T(0) = O(1)$

Pokažimo još i šta se dešava sa jednačinom $T(n) = T(n-1) + c \log n$, $T(0) = d$. Odmotavanjem dobijamo

$$\begin{aligned} T(n) &= T(n-1) + c \log n \\ &= T(n-2) + c \log (n-1) + c \log(n) \\ &= \ldots \\ &= d + c(\log 1 + \ldots + \log n). \end{aligned}$$

Ranije smo pokazali da se zbir logaritama ponaša kao $\Theta(n \log n)$, pa je to i rešenje ove jednačine.

Jednačina $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$

Jednačine u kojima se rekurzivni pozivi dimenzije $n-1$ ponavljaju više puta imaju eksponencijalna rešenja.

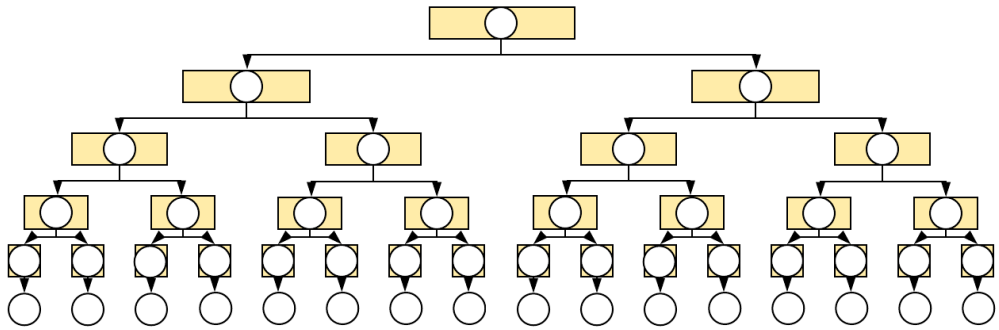
U pitanju je nehomogena jednačina prvog reda i ona se može rešiti svođenjem na homogenu jednačinu. Pretpostavimo da je jednačina $T(n) = 2T(n-1) + c$, $T(0) = d$. Važi $T(n+1) = 2T(n) + c$, pa je $T(n+1) - T(n) = 2T(n) - T(n-1)$, tj. $T(n+1) = 3T(n) - 2T(n-1)$. Karakteristična jednačina je $t^2 = 3t - 2$, čija su rešenja $t = 2$ i $t = 1$. Opšte rešenje je, dakle, oblika

$T(n) = \alpha 2^n + \beta$. Važi i $T(0) = d$ i $T(1) = 2T(0) + c = 2d + c$, odakle se mogu izračunati i konkretne vrednosti koeficijenata α i β, jer je $\alpha + \beta = d$ i $2\alpha + beta = 2d + c$, pa je $\alpha = d + c$, a $\beta = -c$, tj. rešenje je $(c + d)2^n - c$.

Ova jednačina se još jednostavnije može rešiti odmodavanjem.

$$\begin{aligned} T(n) &= 2T(n-1) + c \\ &= 2(2T(n-2) + c) + c = 4T(n-2) + 2c + c \\ &= 4(2T(n-3) + c) + 2c + c \\ &= 8T(n-3) + 4c + 2c + c \\ &= ... \\ &= 2^n T(0) + (2^{n-1} + ... + 2 + 1) \cdot c \\ &= 2^n \cdot d + (2^n - 1) \cdot c = O(2^n). \end{aligned}$$

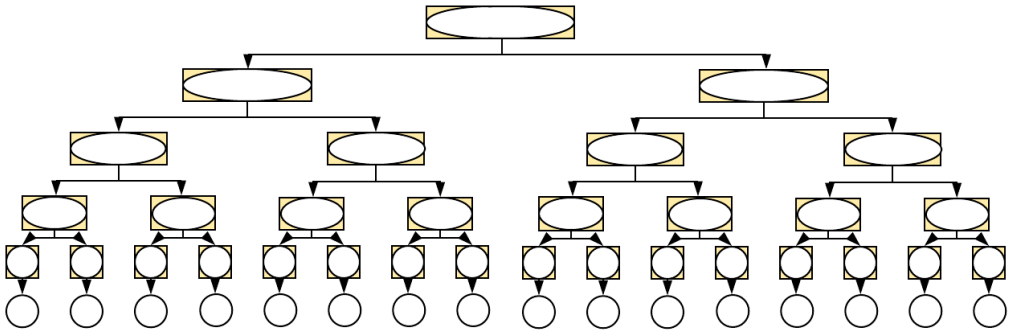
Dakle, iako se u svakom rekurzivnom pozivu radi malo posla, rekurzivnih poziva ima eksponencijalno mnogo, što dovodi do izrazito neefikasnog algoritma.



Slika 2.16: Drvo poziva u slučaju $T(n) = 2T(n-1) + O(1)$, $T(0) = O(1)$ za $n = 5$

Jednačina $T(n) = 2T(n-1) + O(n)$**,** $T(0) = O(1)$

Funkcije koje zadovoljavaju jednačinu $T(n) = 2T(n-1) + O(n), T(0) = O(1)$ takođe pokazuju eksponencijalnu složenost.



Slika 2.17: Drvo poziva u slučaju $T(n) = 2T(n-1) + O(n), T(0) = O(1)$ za $n = 5$

Odmotavanjem jednačine $T(n) = 2T(n-1) + c \cdot n, T(0) = d$ dobijamo

$$\begin{aligned}
T(n) &= 2T(n-1) + c \cdot n \\
&= 2(2T(n-2) + c \cdot (n-1)) + c \cdot n = 4T(n-2) + c \cdot (2(n-1) + n) \\
&= 4(2T(n-3) + c \cdot (n-2)) + c \cdot (2(n-1) + n) \\
&= 8T(n-3) + c \cdot (4(n-2) + 2(n-1) + n) \\
&= ... \\
&= 2^n T(0) + c(2^{n-1} \cdot 1 + 2^{n-2} \cdot 2 + ... + 2(n-1) + n) \cdot O(1) \\
&= 2^n \cdot d + c(\sum_{k=0}^{n} 2^k (n-k)).
\end{aligned}$$

Korišćenjem ranije izvedenih formula možemo jednostavno izračunati i sumu

$$\sum_{k=0}^{n} 2^k (n-k) = n + 2(n-1) + 2^2(n-2) + ... + 2^{n-1} \cdot 1.$$

Važi da je

$$\begin{aligned}\sum_{k=0}^{n} 2^k(n-k) &= n\sum_{k=0}^{n} 2^k - \sum_{k=0}^{n} k2^k \\ &= n(2^{n+1}-1) - ((n-1)2^{n+1}+2) \\ &= 2^{n+1} - n - 2\end{aligned}$$

Zato je $T(n) = 2^n \cdot d + c(2^{n+1} - n - 2)$. Dakle, i u ovom slučaju funkcija pokazuje eksponencijalno ponašanje $O(2^n)$.

2.A.3.8 Master teorema

Jednačine zasnovane na dekompoziciji problema na nekoliko manjih potproblema koje su oblika $T(n) = aT(n/b) + O(n^k)$, $T(0) = O(1)$ se rešavaju na osnovu **master teoreme**.

Teorema: Rešenje rekurentne relacije $T(n) = aT(n/b) + cn^k$, gde su a i b celobrojne konstante takve da važi $a \geq 1$ i $b \geq 1$, i c i k su pozitivne realne konstante je

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) , & \text{ako je } \log_b a > k, \text{tj.} a > b^k \\ \Theta(n^k \log n) , & \text{ako je } \log_b a = k, \text{tj.} a = b^k \\ \Theta(n^k) , & \text{ako je } \log_b a < k, \text{tj.} a < b^k \end{cases}$$

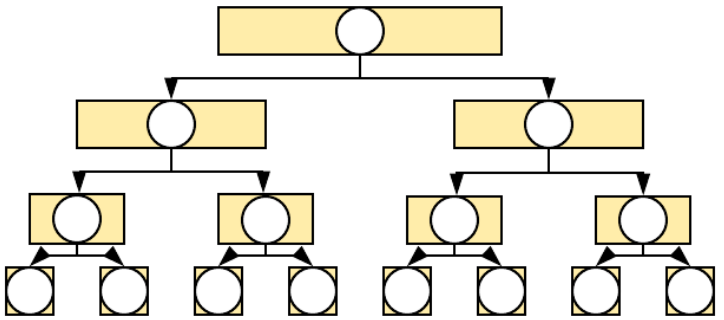
Pre nego što damo dokaz ove teoreme, pokušajmo da kroz niz primera damo intuitivno objašnjenje zašto ova teorema važi, u sva tri svoja slučaja.

Jednačina $T(n) = 2 \cdot T(n/2) + O(1)$, $T(1) = O(1)$

U prvom slučaju se dobija drvo rekurzivnih poziva čiji broj čvorova dominira poslom koji se radi u svakom čvoru. Razmotrimo, na primer, jednačinu $T(n) = 2 \cdot T(n/2) + O(1)$, $T(1) = O(1)$. Drvo će sadržati $O(n)$ čvorova, a u svakom čvoru će se vršiti posao koji zahteva $O(1)$ operacija. Odmotavanjem rekurentne jednačine, dobijamo

$$\begin{aligned}T(n) &= 2 \cdot T(n/2) + O(1) \\ &= 4 \cdot T(n/4) + 2 \cdot O(1) + O(1) \\ &= 8 \cdot T(n/8) + 4 \cdot O(1) + 2 \cdot O(1) + O(1) \\ &= 2^k \cdot T(n/2^k) + (2^{k-1} + ... + 2 + 1) \cdot O(1).\end{aligned}$$

Ako je $n = 2^k$ dobijamo da je $n/2^k = 1$, pa pošto je na osnovu formule za zbir geometrijskog niza $2^{k-1} + ... + 2 + 1 = 2^k - 1$, složenost je $\Theta(n)$. I kada n nije stepen dvojke, dobija se isto asimptotsko ponašanje (što se može dokazati ograničavanjem odozgo i odozdo stepenima dvojke).



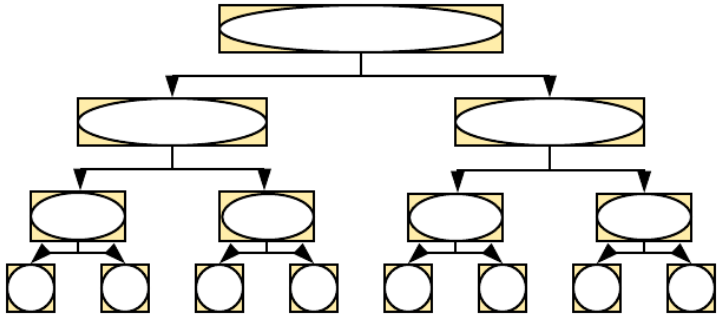
Slika 2.18: Drvo poziva u slučaju $T(n) = 2T(n/2) + O(1)$, $T(1) = O(1)$ za $n = 8$

Jednačina $T(n) = 2 \cdot T(n/2) + c \cdot n$**,** $T(1) = O(1)$

U drugom slučaju su broj čvorova i posao koji se radi na neki način uravnoteženi. Razmotrimo, na primer, jednačinu $T(n) = 2 \cdot T(n/2) + c \cdot n$, $T(1) = O(1)$ i ponovo pokušajmo da je odmotamo.

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + c \cdot n \\ &= 2 \cdot (2 \cdot T(n/4) + c \cdot n/2) + c \cdot n \\ &= 4T(n/4) + c \cdot n + c \cdot n \\ &= 4(2T(n/8) + c \cdot n/4) + 2 \cdot c \cdot n \\ &= 8T(n/8) + 3 \cdot c \cdot n \\ &= ... \\ &= 2^k \cdot T(n/2^k) + k \cdot c \cdot n. \end{aligned}$$

Ako je $n = 2^k$ posle $k = \log_2 n$ koraka $n/2^k$ će dostići vrednost 1 tako da će zbir biti reda veličine $n \cdot O(1) + \log_2 n \cdot c \cdot n = \Theta(n \log n)$. Isto važi i kada n nije stepen dvojke.



Slika 2.19: Drvo poziva u slučaju $T(n) = 2T(n/2) + O(n)$, $T(1) = O(1)$ za $n = 8$

Jednačina $T(n) = T(n/2) + cn$**,** $T(1) = O(1)$

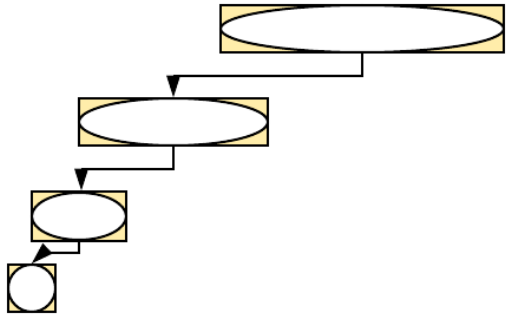
U trećem slučaju posao koji se radi u čvorovima dominira brojem čvorova. Razmotrimo jednačinu $T(n) = T(n/2) + cn$, $T(1) = O(1)$. Njenim odmotavanjem dobijamo da je

$$\begin{aligned} T(n) &= T(n/2) + cn \\ &= T(n/4) + cn/2 + cn \\ &= T(n/8) + cn/4 + cn/2 + cn \\ &= \ldots \\ &= T(n/2^k) + cn(1/2^{k-1} + \ldots + 1/2 + 1). \end{aligned}$$

Ponovo, ako je $n = 2^k$, tada je prvi član jednak $O(1)$ i pošto je na osnovu formule za zbir geometrijskog niza $1/2^{k-1} + \ldots + 1/2 + 1 = (1-(1/2)^k)/(1-(1/2)) = 2 - 2/n$ zbir je jednak $O(1) + cn(2 - 2/n) = \Theta(n)$.

Dokaz master teoreme

Pretpostavimo, jednostavnosti radi, da je $n = b^s$ (ovo znači da će se na nekom nivou rekurzije doći do toga da su svi pozivi za $n = 1$, tj. da će nakon s rekurzivnih poziva u svima njima doći do izlaza iz rekurzije, jer je $n/b^s = 1$). Razmotajmo jednačinu, uopštavajući je na oblik: $T(n) = aT(n/b) + f(n)$, $T(1) = d$:



Slika 2.20: Drvo poziva u slučaju $T(n) = T(n/2) + O(n)$, $T(1) = O(1)$ za $n = 8$

$$
\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\
&= a\left(aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right)\right) + f(n) = a^2T\left(\frac{n}{b^2}\right) + \left(af\left(\frac{n}{b}\right) + f(n)\right) = \\
&\ldots \\
&= a^sT\left(\frac{n}{b^s}\right) + \left(a^{s-1}f\left(\frac{n}{b^{s-1}}\right) + \ldots + af\left(\frac{n}{b}\right) + f(n)\right) \\
&= a^sT(1) + \left(a^{s-1}f\left(\frac{n}{b^{s-1}}\right) + \ldots + af\left(\frac{n}{b}\right) + f(n)\right)
\end{aligned}
$$

Primetimo da rešenje zavisi od dva sabirka. Vrednost a^s je broj listova drveta rekurzivnih poziva, a $a^s \cdot T(1)$ je vreme koje se utroši na obradu baznih slučajeva tj. obradu izlazaka iz rekurzije. Drugi sabirak daje ukupno vreme potrebno za pripremu svih rekurzivnih poziva i obradu njihovih rezultata. Na prvom nivou postoji jedan poziv u kom se obrađuje ulaz dimenzije n (vrednost $f(n)$), na drugom a poziva u kom se obrađuju ulazi dimenzije n/b (vrednost $af(n/b)$), na trećem a^2 poziva u kom se obrađuju ulazi dimenzije n/b^2 (vrednost $a^2f(n/b^2)$) i tako dalje. Postoji tačno s nivoa rekurzije (na poslednjem je izlaz iz rekurzije).

Iz $n = b^s$, sledi da je $s = \log_b n$, pa važi sledeće:

$$
a^s = a^{\log_b n} = a^{\frac{\log_a n}{\log_a b}} = a^{\log_a n \cdot \log_b a} = \left(a^{\log_a n}\right)^{\log_b a} = n^{\log_b a}.
$$

Zato pod pretpostavkom da izlaz iz rekurzije zahteva konstantno vreme, za prvi sabirak važi $a^s \cdot T(1) = \Theta\left(n^{\log_b a}\right)$. U zavisnosti od funkcije f i odnosa vrednosti a i b, zavisi da li će vremenom dominirati prvi ili drugi sabirak. Naime, vrednost $\log_b a$ se naziva kritična vrednost i rezultat zavisi od toga da li drugi sabirak ima asimptotsku složenost manju, veću ili jednaku od $n^{\log_b a}$.

U slučaju polazne jednačine $T(n) = aT(n/b) + cn^k$, $T(1) = d$, važi da je $f(n) = cn^k$, pa dobijamo:

$$T(n) = d \cdot a^s + c \cdot n^k \cdot \left(1 + \frac{a}{b^k} + ... + \left(\frac{a}{b^k}\right)^{s-1}\right)$$

Ako je $a \neq b^k$, tada je drugi sabirak moguće uprostiti primenom formule za zbir geometrijskog niza.

$$T(n) = d \cdot a^s + c \cdot n^k \cdot \left(\frac{\left(\frac{a}{b^k}\right)^s - 1}{\frac{a}{b^k} - 1}\right).$$

Slučaj $a < b^k$

Ako je $a < b^k$, tada član $\left(\frac{a}{b^k}\right)^s$ teži nuli sa porastom n tj. s, pa je zbir geometrijskog niza odozgo ograničen vrednošću $\frac{1}{1-\frac{a}{b^k}} = \Theta(1)$. Zato je asimptotsko ponašanje vremena izvršavanja jednako $T(n) = \Theta(n^{\log_b a}) + \Theta(n^k)$. Pošto je $a < b^k$, važi i da je $\log_b a < k$, pa je $a^s = n^{\log_b a} = O(n^k)$ i konačno asimptotsko ponašanje rešenja je $\Theta(n^k)$.

Primetimo da je u ovom slučaju rekurzivni postupak takav da je posla u listovima (prilikom izlaska iz rekurzije) manje nego što ima posla na pripremi rekurzivnih poziva i obradi njihovih rezultata. Ovo se obično dešava kod slabo razgranatih drveta i kod drveta kod kojih je priprema rekurzivnih poziva i obrada dobijenih rezultata skupa operacija.

Slučaj $a > b^k$

U ovom slučaju vrednost geometrijskog niza teži beskonačnosti, brzinom vodećeg sabirka $\Theta\left(\left(\frac{a}{b^k}\right)^s\right)$. Zato važi

$$\begin{aligned} T(n) &= d \cdot a^s + c \cdot n^k \cdot \left(\frac{\left(\frac{a}{b^k}\right)^s - 1}{\frac{a}{b^k} - 1} \right) \\ &= \Theta(a^s) + c \cdot n^k \cdot \Theta\left(\left(\frac{a}{b^k}\right)^s \right) \\ &= \Theta(a^s) + c \cdot n^k \cdot \Theta\left(\frac{a^s}{(b^s)^k} \right) \\ &= \Theta(n^{\log_b a}) + c \cdot n^k \cdot \Theta\left(\frac{n^{\log_b a}}{n^k} \right) \\ &= \Theta(n^{\log_b a}) + \Theta(n^{\log_b a}) = \Theta(n^{\log_b a}). \end{aligned}$$

Primećujemo da u ovom slučaju dominiraju listovi drveta, tj. vreme potrebno za pripremu svih rekurzivnih poziva i obradu rezultata je asimptotski jednako vremenu koje se potroši u listovima tj. pri izlasku iz rekurzije. Ovo se obično dešava kod veoma razgranatih drveta, kod kojih je priprema rekurzivnih poziva i obrada rezultata brza.

Slučaj $a = b^k$

U ovom slučaju ne možemo primeniti formulu za zbir geometrijskog niza, međutim, svaki njegov član je jednak 1. Zato je $T(n) = d \cdot a^s + c \cdot n^k \cdot s$. Važi da je $c \cdot n^k \cdot s = c \cdot \log_b n \cdot n^k = \Theta(n^k \log n)$. Pošto je i sada $a^s = n^{\log_b a} = n^k$, ukupno vreme je $T(n) = \Theta(n^k \log n)$.

Primetimo da je u ovom slučaju vreme potrošeno na svakom nivou rekurzije (uključujući i izlaz iz rekurzije) ujednačeno i jednako $\Theta(n^k) = \Theta\left(n^{\log_b a}\right)$. Ukupan broj nivoa rekurzije je $s = \Theta(\log n)$.

Ostali tipovi jednačina

Prokomentarišimo da se u nekim problemima dobijaju jednačine koje nisu baš u svakom rekurzivnom pozivu identične ovim navedenim. Na primer, prilikom analize algoritma QuickSort, ako je pivot tačno na sredini niza, važi da je $T(n) = 2T(n/2) + O(n)$ i $T(1) = O(1)$. Kada bi se to stalno događalo, rešenje bi bilo $T(n) = O(n \log n)$, međutim, verovatnoća da se to dogodi je strašno mala, jer u većini slučajeva pivot ne deli niz na dva dela potpuno iste dimenzije i zato treba biti obazriv. Ako bi se desilo da pivot stalno završavao na jednom kraju niza, jednačina

bi bila $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$, što bi dovelo do složenosti $O(n^2)$, što i jeste složenost najgoreg slučaja. Analizom koju ćemo prikazati kasnije se može utvrditi da je prosečna složenost $O(n \log n)$ tj. da iako pivot nije stalno na sredini, da je u dovoljnom procentu slučajeva negde blizu nje (recimo između 25% i 75% dužine niza). Slična analiza važi i za problem pronalaženja medijane.

Međutim, postoje i algoritmi kod kojih stvari stoje drugačije. Prilikom obilaska binarnog drveta, balansiranost nema uticaja. Naime, ako je drvo potpuno, tada je jednačina $T(n) = 2T(n/2) + O(1)$, $T(1) = O(1)$, čije je rešenje $O(n)$. Međutim, čak i kada je drvo izdegenerisano u listu, jednačina je $T(n) = T(n-1) + O(1)$, $T(1) = O(1)$, čije je rešenje opet $O(n)$. Kakav god da je odnos broja čvorova u levom i desnom poddrvetu rešenje će biti $O(n)$. To se može opisati jednačinom $T(n) = T(k) + T(n-k-1) + O(1)$, $T(1) = O(1)$, za $0 \leq k \leq n-1$, čije će rešenje biti $O(n)$, bez obzira na to kakvo se k pojavljuje u raznim rekurzivnim pozivima.

3. Elementarne tehnike poboljšanja složenosti

U nastavku ovog poglavlja prikazaćemo niz zadataka koje ćemo rešiti različitim algoritmima, analiziraćemo njihovu asimptotsku složenost najgoreg slučaja i prikazaćemo kako se na bazi prikazanih saveta mogu izgraditi značajno efikasniji algoritmi.

3.1 Zamena iteracije formulom

Jedan od važnih principa optimizacije algoritama je taj da se izbegne da računar dugotrajnim iterativnim postupkom (u linearnoj ili višoj složenosti) računa nešto što se unapred može izračunati primenom matematičkih formula (u konstantnoj složenosti).

Na primer, računanje zbira prvih n prirodnih brojeva u programu ne bi trebalo da bude linearne složenosti, već konstantne, jer se zna da je taj zbir jednak $1 + 2 + \ldots + n = n(n + 1)/2$. Slične optimizacije se mogu primeniti i kada je potrebno izračunati n-ti element ili zbir aritmetičkog ili zbir geometrijskog niza ili bilo kog drugog niza koji možemo sabrati primenom pogodne matematičke formule.

Dalje, često se dešava da nam je potrebno da izračunamo koliko ima nekih objekata. Tada se često bolje rešenje dobija primenom kombinatornih formula nego procedurom koja generiše sve takve objekte (ili, još gore, u nekom širem skupu proveravala koji od objekata zadovoljavaju tražena svojstva). Na primer, n-tocifreni brojevi koji se zapisuju samo pomoću dve različite cifre se mogu lako prebrojati primenom kombinatorike i veoma loše rešenje je da se u programu generišu i analiziraju svi n-tocifreni brojevi.

I u optimizacionim problemima je ponekad moguće jednoznačno odrediti karakterizaciju maksimalne tj. minimalne vrednosti i takva rešenja su mnogo bolja nego isprobavanje velikog broja kandidata.

S druge strane, često se javljaju i problemi za koje ne postoji unapred poznata matematička formula kojom se izračunava tražena vrednost. U takvim situacijama neophodno je upotrebiti računarsku snagu da bi se do rešenja došlo iterativnim postupcima (sabiranjem puno sabiraka, analizom puno kandidata za rešenje i slično).

3.1.1 *Euklidov algoritam*

Kao što je već rečeno, i neki čuveni algoritmi i njihove optimizacije su zasnovane na jednostavnim tehnikama koje u ovom poglavlju opisujemo. Jedan od njih je i Euklidov algoritam. Originalna varijanta Euklidovog algoritma za pronalaženje NZD dva broja pronalazi NZD tako što od većeg pozitivnog prirodnog broja oduzima manji sve dok manji od njih ne postane nula.

```
int nzd(int a, int b) {
   while (b != 0) {
      if (a > b)
          a -= b;
      else
          b -= a;
   }
   return a;
}
```

Program ispravno radi kada su a i b pozitivni, ispravno radi i kada je $b = 0$, ali se ne zaustavlja kada je $a = 0$ (što se lako može ispraviti ispitivanjem tog specijalnog slučaja i vraćanjem vrednosti b na početku funkcije ili tako što se uslov petlje promeni da se program zaustavlja kada bilo koja promenljiva postane 0, a kao rezultat vrati vrednost one druge promenljive).

U slučaju kada je jedan od ovih brojeva dosta manji od drugog, ovo je veoma sporo. Na primer, za $a = 135$ i $b = 12$ dobija se sledeći niz vrednosti:

a	135	123	111	99	87	75	63	51	39	27	15	3	3	3	3	3
b	12	12	12	12	12	12	12	12	12	12	12	12	9	6	3	0

Ceo dugački iterativni niz izračunavanja kojim se vrednost 135 smanjuje do 3 se može ukloniti kada se primeti da će se broj 12 oduzimati od veće vrednosti sve dok

se broj a ne smanji do vrednosti ostatka pri deljenju polaznog broja 145 brojem 12, a to je 3. Zatim će se ta vrednost 3 oduzimati iterativno od broja 12 sve dok se broj b ne smanji do vrednosti ostatka pri deljenju broja 12 brojem 3 a to je 0. Dakle, umesto da u svakom koraku iterativno smanjujemo broj oduzimanjem, ovaj dugačak postupak možemo zameniti jednim korakom izračunavanja ostatka pri deljenju. Tako dobijamo sledeći algoritam.

```
int nzd(int a, int b) {
   while (b != 0) {
       int ostatak = a % b;
       a = b;
       b = ostatak;
   }
   return a;
}
```

Na ovaj način dobijamo sledeću tabelu izvršavanja za brojeve $a = 145$ i $b = 12$.

a	135	12	3
b	12	3	0

Primetimo da u varijanti sa deljenjem nije potrebno porediti vrednosti a i b. Ako je vrednost a manja od b, tada je $a \bmod b = a$, pa se od para (a, b) dobija par $(b, a \bmod b) = (b, a)$ tj. u prvom koraku se vrednosti razmenjuju. Pošto je $a \bmod b < b$, nadalje važi da će vrednost a biti veća od vrednosti b.

Složenost najgoreg slučaja polazne varijante algoritma – varijante sa oduzimanjem je $O(\max(a, b))$, jer najgori slučaj nastupa kada je vrednost manjeg od dva broja jednaka 1. U novom algoritmu – u varijanti sa deljenjem, asimptotska složenost značajno je redukovana. Naime, u svaka dva koraka algoritma vrednost broja a smanji bar upola. Zaista, u prvom koraku se od para brojeva (a, b) dobija par brojeva $(b, a \bmod b)$, a u drugom se dobija par brojeva $(a \bmod b, b \bmod (a \bmod b))$. Tvrdimo da je $a \bmod b \leq \frac{a}{2}$. Zaista, ako je $b \leq \frac{a}{2}$, tada je $a \bmod b < b \leq \frac{a}{2}$. Ako je $b > \frac{a}{2}$, tada je $a \bmod b = a - b < a - \frac{a}{2} = \frac{a}{2}$. Zato je dvostruki broj koraka koji se može sprovesti u najgorem slučaju $\log_2(\max(a, b))$, pa je složenost ove varijante algoritma jednaka $O(\log(\max(a, b)))$.

Primetimo da smo složenost iskazali u terminima ulaznih vrednosti a i b. Ako se složenost iskazuje u terminima veličine zapisa ulaza, onda je potrebno izraziti ju je u terminima broja cifara vrednosti a i b, a koji logaritamski zavisi od njihovih vrednosti. Dakle, u tom slučaju varijanta sa oduzimanjem ima eksponencijalnu, a varijanta sa deljenjem linearnu složenost u odnosu na veličinu ulaza.

Ilustrujmo tehniku optimizacije zamenom iteracije formulom kroz još nekoliko jednostavnih problema.

Zadatak: Broj podniski koje počinju i završavaju sa 1

Data je binarna niska (niska karaktera koja se sastoji od karaktera 0 i 1). Napisati program kojim se određuje broj segmenata (podniski uzastopnih elemenata) u datoj niski koji su dužine najmanje 2, a koji počinju i završavaju sa 1.

Opis ulaza

Prva i jedina linija standardnog ulaza sadrži binarnu nisku (sastavljenu od 0 i 1).

Opis izlaza

Na standardnom izlazu prikazati traženi broj segmenata.

Primer

Ulaz	*Izlaz*
`010001001`	`3`

Objašnjenje

To su podniske `10001`, `10001001` i `1001`.

Rešenje

Analiza svih segmenata

Broj svih segmenata koji počinju i završavaju sa 1 možemo jednostavno odrediti analizirajući sve segmente. U spoljašnjoj petlji analiziramo jedan po jedan karakter. Svaku jedinicu na koju naiđemo (za svako i takvo da je s_i jednako 1), razmatramo kao početak segmenta i u unutrašnjoj petlji (brojačem j od $i + 1$ do kraja niske) tražimo jedinicu kojom se segment završava. Za svaku jedinicu pronađenu u unutrašnjoj petlji (za svako j takvo da je s_j jednako 1) uvećavamo broj segmenata.

```
int broj1x1Podniski(const string& s) {
  int n = s.length();
  int br = 0;
```

```
  for (int i = 0; i < n - 1; i++)
    if (s[i] == '1')
      for (int j = i + 1; j < n; j++)
        if (s[j] == '1')
          br++;
  return br;
}
```

Primetimo da na ovaj način iste karaktere niske nepotrebno analiziramo veliki broj puta. Složenost algoritma odgovara ukupnom broju svih segmenata i jednaka je $O(n^2)$.

Brojanje jedinica

Svaki segment koji počinje i koji se završava jedinicom definisan je pozicijama dve jedinice u niski, pa je ukupan broj traženih segmenata jednak broju načina da se izaberu dve različite jedinice u niski. Ako je ukupan broj jedinica u niski jednak b onda dve jedinice možemo izabrati na $\frac{b\cdot(b-1)}{2}$ načina.

```
int broj1x1Podniski(const string& s) {
  int brojJedinica = 0;
  for (char c : s)
    if (c == '1')
       brojJedinica++;
  return brojJedinica * (brojJedinica - 1) / 2;
}
```

Pošto jedinice prebrojavamo samo jednim prolaskom kroz nisku, složenost ovog algoritma je $O(n)$.

Ovaj problem se lako uopštava na, na primer, brojanje svih podniski koje počinju i završavaju se istim karakterom u datom tekstu. Ovakve statistike teksta mogu biti korisne u analizi podataka, bioinformatici (na primer, analizi DNK sekvenci) i slično.

Zadatak: Pitagorine trojke

Napisati program koji ispisuje sve trojke prirodnih brojeva a, b, c, takve da je $a^2 + b^2 = c^2$, a u kojima važi $a \leq b \leq c \leq n$.

Opis ulaza

Sa standardnog ulaza se učitava prirodan broj $n \leq 5 \cdot 10^5$.

Opis izlaza

Na standardni izlaz ispisati u leksikografskiom redosledu tražene trojke brojeva, svaku u posebnom redu, sa po jednim razmakom između brojeva.

Primer

Ulaz	*Izlaz*
17	3 4 5 5 12 13 6 8 10 8 15 17 9 12 15

Rešenje

Gruba sila

Zadatak možemo da rešimo formiranjem svih trojki (a, b, c), u kojima je $a \leq b \leq c \leq n$ i ispisivanjem onih koje ispunjavaju Pitagorin uslov $a^2 + b^2 = c^2$.

```
for (int a = 1; a <= n; a++)
  for (int b = a; b <= n; b++)
    for (int c = b; c <= n; c++)
      if (a*a + b*b == c*c)
        cout << a << " " << b << " " << c << " " << endl;
```

Broj ispitivanja uslova u trostrukoj petlji je srazmeran sa n^3, pa je složenost ovog rešenja $O(n^3)$.

Izračunavanje hipotenuze

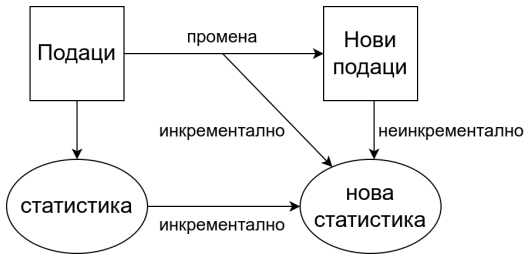
Primetimo da, kada su vrednosti kateta a, b fiksirane, hipotenuzu možemo da izračunamo kao $\sqrt{a^2 + b^2}$. Tako, umesto da za vrednost hipotenuze iterativno isprobavamo sve vrednosti od b do n, dovoljno je da ispitamo da li je izračunata vrednost celobrojna. Time zadatak rešavamo pomoću dvostruke, umesto trostruke petlje, čime rešenje postaje značajno brže. Složenost ovog rešenja je $O(n^2)$.

```
for (int a = 1; a <= n; a++)
  for (int b = a; a*a + b*b <= n*n; b++) {
    double cr = sqrt(a*a + b*b);
    int c = round(cr);
    if (c == cr)
      cout << a << " " << b << " " << c << " " << endl;
  }
```

Ubrzanje smo ponovo dobili tako što smo izbegli iteraciju na osnovu primene matematičke formule tj. tako što smo izračunali jedinstvenu vrednost koja ima šanse da zadovolji traženi uslov. Česta greška programera je da previde da u nekom skupu postoji jedinstven kandidat za rešenje zadatka i da se taj kandidat može eksplicitno izračunati, umesto da se iterativno pretražuje.

3.2 Inkrementalnost

Jedna od osnovnih tehnika za izbegavanje loše složenosti algoritama je da se izbegne izračunavanje istih ili povezanih stvari više puta u istom programu. U računarstvu često imamo potrebu za izračunavanjem određene funkcije nekog skupa podataka (reći ćemo da izračunavamo neku statistiku tih podataka). Kada se podaci promene, menja se i vrednost statistike. Čest je slučaj da se nakon male promene podataka (na primer, proširenja skupa novim podatkom) statistika ne mora računati iz početka, već se može efikasnije izračunati na osnovu poznate vrednosti statistike originalnih podataka i promene koje su se desile nad podacima. Ovo je ilustrovano na slici 3.1.



Slika 3.1: U neinkrementalnom pristupu nova vrednost statistike se iznova izračunava na osnovu novih vrednosti podataka. U inkrementalnom pristupu se nova vrednost statistike izračunava na osnovu stare vrednosti statistike i promene u podacima.

Veoma jednostavan primer principa inkrementalnosti je izračunavanje zbirova prefiksa (tzv. parcijalnih zbirova) elemenata nekog niza. Na primer, ako je dat niz $1, 2, 3, 4, 5$, njegovi parcijalni zbirovi su redom $0, 1, 3, 6, 10, 15$ (o značaju prefiksnih zbirova biće više reči u poglavlju 3.4). Veoma jednostavno se primećuje da se izračunavanje narednog parcijalnog zbira ne mora vršiti sabiranjem svih elemenata od početka, već se može dobiti sabiranjem prethodnog parcijalnog zbira sa tekućim elementom niza (na primer, zbir $1+2+3+4 = 10$, se dobija sabiranjem prethodnog zbira $1 + 2 + 3 = 6$ i tekućeg elementa 4). Ako zbir prvih k elemenata niza označimo sa Z_k, tada važi:

$$Z_0 = 0, \quad Z_{k+1} = Z_k + a_k, k > 0.$$

Ovim smo dobili seriju brojeva u kojoj se naredni element izračunava na osnovu prethodnog (ili nekoliko prethodnih). Za takve serije kažemo da su *rekurentne serije*. Svaki naredni član se izračunava u složenosti $O(1)$, pa se izračunavanje svih parcijalnih zbirova niza dužine n vrši u složenosti $O(n)$. Kada bi se svaki parcijalni zbir računao sabiranjem elemenata niza iz početka, tada bi izračunavanje k-tog zbira bilo složenosti $O(k)$, a izračunavanje svih zbirova složenosti $O(n^2)$.

Princip inkrementalnosti je u tesnoj vezi sa induktivno/rekurzivnom konstrukcijom algoritama i leži u osnovi velikog broja osnovnih algoritama. Izračunavanje zbira svih elemenata niza zapravo počiva na postepenom, inkrementalnom izračunavanju zbirova prefiksa, sve dok se ne izračuna zbir svih elemenata niza. Slično je i sa izračunavanjem minumuma, maksimuma, linearnom pretragom i drugim fundamentalnim algoritmima. U svim ovim primerima krećemo od neke početne vrednosti u nizu rezultata, a zatim narednu vrednost u tom nizu izračunavamo na osnovu prethodne ili nekoliko prethodnih, što direktno odgovara induktivnom postupku izračunavanja. Slična tehnika (dobijanja narednih rezultata na osnovu prethodnih) primenjuje se u sklopu tehnike dinamičkog programiranja naviše, o čemu će više reči biti u poglavlju **??**.

Pored parcijalnih zbirova, inkrementalno se mogu izračunavati i parcijalni proizvodi, parcijalni minimumi i maksimumi i slično, ali i mnoge druge, naprednije statistike. Ilustrujmo ovu tehniku kroz nekoliko primera.

Zadatak: Suma reda

Želimo da obezbedimo anonimnost sistema za elektronsko glasanje tako što će se nakon glasanja izvršiti permutovanje glasova i to tako da nijedan glas ne ostane uz

osobu koja ga je dala. Permutacija je *deranžman* (engl. derangement) ako se nijedan element ne nalazi na svojoj originalnoj poziciji (na primer, permutacija 4321 jeste deranžman za elemente 1234, dok permutacija 3241 nije deranžman, jer se element 2 nalazi na poziciji 2). Verovatnoća da je nasumično izabrana permutacija dužine n deranžman (tj. da anonimnost glasanja n glasača bude obezbeđena nasumično odabranom permutacijom) može se izračunati na osnovu formule $\sum_{k=0}^{n} \frac{(-1)^k}{k!} = 1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} \ldots + \frac{(-1)^n}{n!}$. Napisati program koji izračunava ovu verovatnoću.

Napomena: Pošto važi

$$e^x = \sum_{k=0}^{+\infty} \frac{x^k}{k!},$$

sa povećanjem broja n ova verovatnoća teži vrednosti $e^{-1} = 1/e \approx 36{,}79\% \ldots$.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($2 \leq n \leq 20$).

Opis izlaza

Na standardni izlaz ispisati traženu verovatnoću, zaokruženu na 14 decimala.

Primer 1

Ulaz	*Izlaz*
`2`	`0.500000000000000`

Primer 2

Ulaz	*Izlaz*
`10`	`0.367879464285714`

Rešenje

Izračunavanje svakog sabirka zasebno

Direktan način da se izračuna tražena verovatnoća je da se izračuna zbir serije brojeva koja se dobija tako što se za svako k od 0 do n izračuna vrednost $\frac{(-1)^k}{k!}$. Stepen $(-1)^k$ se može izračunati funkcijom `pow` ili se može odrediti grananjem, pošto je $(-1)^k = 1$ za parne vrednosti k i $(-1)^k = -1$ za neparne vrednosti k. Faktorijel $k!$ se može izračunati množenjem serije brojeva od 1 do k.

```
// faktorijel broja n
double faktorijel(int n) {
  double p = 1.0;
  for (int i = 2; i <= n; i++)
```

```
    p *= i;
  return p;
}

// zbir 1 - 1/1! + 1/2! + ... + (-1)^n/n!
double verovatnoca(int n) {
  double p = 0.0;
  for (int k = 0; k <= n; k++)
    p += pow(-1, k) / faktorijel(k);
  return p;
}
```

Izračunavanje k-tog sabirka zahteva $O(k)$ operacija, pa je za sabiranje n sabiraka potrebno vreme $O(n^2)$.

Problem sa ovim pristupom nije samo vremenska složenost. Pošto faktorijeli veoma brzo rastu, postoji opasnost da za veće vrednosti k nastupi prekoračenje prilikom izračunavanja $k!$ (čak i kada se koristi tip podataka `double`). Slično, ako se računa $\frac{x^k}{k!}$, za $x > 1$, tada i brojilac raste veoma brzo, pa i tu preti opasnost od prekoračenja, iako je vrednost razlomka sve bliža i bliža nuli.

Inkrementalno izračunavanje sabiraka

Efikasnije rešenje se može dobiti ako se uoči da brojevi $1 = \frac{(-1)^0}{0!}$, $-1 = \frac{(-1)^1}{1!}$, $\frac{1}{2} = \frac{(-1)^2}{2!}$, $-\frac{1}{6} = \frac{(-1)^3}{3!}$ itd. čine veoma pravilnu seriju u kojoj se svaki naredni član može dobiti množenjem prethodnog člana vrednošću $-\frac{1}{k}$. Dakle, u ovom zadatku se koristi kako inkrementalnost serije parcijalnih zbirova (u sklopu algoritma sabiranja), tako i inkrementalnost serije samih sabiraka, koji su zapravo parcijalni proizvodi pravilne serije $-\frac{1}{1}, -\frac{1}{2}, -\frac{1}{3}, -\frac{1}{4}, \ldots$ Ako sa x_k obeležimo sabirak k, tada je $x_0 = \frac{(-1)^0}{0!} = 1$, dok je $x_{k+1} = -\frac{1}{k} \cdot x_k$.

Tokom implementacije ćemo održavati dve promenljive. Prva će da predstavlja zbir do sada sabranih sabiraka, a druga tekući sabirak. Zbir i tekući član inicijalizovaćemo na vrednost 1 (to je vrednost $\frac{(-1)^0}{0!}$). U svakom koraku petlje u kojoj k uzima vrednosti od 1 do n tekući član ćemo ažurirati množenjem sa vrednošću $-\frac{1}{k}$ i dodavaćemo ga na zbir. Nakon završetka petlje, zbir će sadržati traženu verovatnoću koju ćemo ispisati sa traženim brojem decimala.

Dobar savet prilikom izračunavanja zbirova ovog tipa je da se izračuna količnik dva susedna sabirka i da se proveri da li se on možda veoma jednostavno izračunava u funkciji od k (u ovom slučaju taj količnik je $\frac{-1}{k}$). Umesto količnika, u nekim zadacima je pogodnije razmatrati razliku dva uzastopna sabirka.

```
// verovatnoca je jednaka zbiru
// 1 - 1/1! + 1/2! + ... + (-1)^n/n!
double verovatnoca(int n) {
  double p = 1.0;
  // tekuci element zbira (-1)^k/k!
  double xk = 1.0;
  for (int k = 1; k <= n; k++) {
    // izracunavamo sledeci clan mnozenjem prethodnog sa -1/k
    xk *= -1.0/k;
    // dodajemo ga na z
    p += xk;
  }
  return p;
}
```

Izračunavanje k-tog sabirka na osnovu prethodnog zahteva samo $O(1)$ operacija, pa se izračunavanje zbira n sabiraka vrši u vremenu $O(n)$. Doduše, u ovom zadatku n je veoma mali broj, pa se na ovaj način ne postiže značajno ubrzanje, ali u drugim sličnim zadacima optimizacija na osnovu inrementalnosti može biti veoma značajna.

Zadatak: Ruter

Duž jedne ulice su ravnomerno raspoređene zgrade (rastojanje između svake dve susedne je jednako). Za svaku zgradu je poznat broj stanova koje novi dobavljač interneta treba da poveže. Svaki stan se povezuje posebnim optičkim kablom sa ruterom. Odrediti u koju od zgrada treba postaviti ruter tako da ukupna dužina optičkih kablova kojim se svaki od stanova povezuje sa ruterom bude minimalna (računati samo dužinu kablova od zgrade do zgrade i zanemariti dužine unutar zgrada).

Opis ulaza

U prvom redu standardnog ulaza nalazi se broj n ($1 \leq n \leq 10^5$), a u narednom n prirodnih brojeva razdvojenih razmacima koji predstavljaju broj stanova u svakoj od n zgrada.

Opis izlaza

Na standardni izlaz ispisati minimalnu dužinu kablova.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`6` `3 5 1 6 2 4`	`30`	Ruter treba postaviti u četvrtu zgradu sleva i dužina kablova je tada jednaka $3 \cdot 3 + 2 \cdot 5 + 1 \cdot 1 + 1 \cdot 2 + 2 \cdot 4 = 30$.

Rešenje

Gruba sila

Direktno rešenje bi podrazumevalo da se izračuna dužina kablova za svaku moguću poziciju rutera i da se odabere najmanji. Da bismo izračunali dužinu kablova, ako je ruter u zgradi na poziciji k, računamo zapravo zbir

$$\sum_{i=0}^{n-1} |k - i| \cdot a_i,$$

gde je a_i broj stanova u zgradi i.

```
long long minDuzinaKablova(const vector<int>& broj_stanara) {
  // broj zgrada
  int n = broj_stanara.size();
  // minimalna duzina kablova
  long long min_duzina_kablova =
     numeric_limits<long long>::max();
  // obrađujemo sve zgrade od 1 do n-1
  for (int k = 0; k < n; k++) {
    // duzina kablova ako je ruter u zgradi broj k
    long long duzina_kablova = 0;
    for (int i = 0; i < k; i++)
      duzina_kablova += (k - i) * broj_stanara[i];
    for (int i = k+1; i < n; i++)
      duzina_kablova += (i - k) * broj_stanara[i];

    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
```

```
    }

    return min_duzina_kablova;
}
```

Svaki težinski zbir možemo izračunati u vremenu $O(n)$, pa pošto se ispituje n pozicija, algoritam je složenosti $O(n^2)$.

Rešenje na osnovu principa inkrementalnosti

Mnogo bolje rešenje i algoritam linearne složenosti možemo dobiti ako primenimo princip inkrementalnosti i izbegnemo računanje u svakom koraku iz početka. Razmotrimo kako se dužina kablova menja kada se ruter pomera sa zgrade k na zgradu $k+1$.

Dužinu kablova za ruter u zgradi $k+1$ dobijamo od dužine kablova za ruter u zgradi k tako što tu dužinu uvećamo za ukupan broj stanova zaključno sa zgradom k i umanjimo je za ukupan broj stanova počevši od zgrade $k+1$. To je intuitivno jasno i bez strogog matematičkog izvođenja. Pomeranjem rutera u narednu zgradu, svakom stanu zaključno do zgrade k dužina kabla se povećala za jedno rastojanje između zgrada, a svim stanovima od zgrade $k+1$ nadesno se ta dužina smanjuje za jedno rastojanje između zgrada.

Formalno, matematički, to se može pokazati na sledeći način. Ako je ruter na poziciji k, tada je dužina kablova jednaka

$$d_k = \sum_{i=0}^{k-1} (k-i) \cdot a_i + \sum_{i=k+1}^{n-1} (i-k) \cdot a_i.$$

Ako je ruter na poziciji $k+1$, tada je dužina kablova jednaka

$$d_{k+1} = \sum_{i=0}^{k} (k+1-i) \cdot a_i + \sum_{i=k+2}^{n-1} (i-k-1) \cdot a_i.$$

Razlika između te dve sume jednaka je

$$
\begin{aligned}
d_{k+1} - d_k &= \left(\sum_{i=0}^{k} (k+1-i) \cdot a_i - \sum_{i=0}^{k-1} (k-i) \cdot a_i \right) + \\
&\quad \left(\sum_{i=k+2}^{n-1} (i-k-1) \cdot a_i - \sum_{i=k+1}^{n-1} (i-k) \cdot a_i \right) \\
&= \left(\sum_{i=0}^{k-1} ((k+1-i) - (k-i)) \cdot a_i \right) + a_k \\
&\quad -a_{k+1} + \left(\sum_{i=k+2}^{n-1} ((i-k-1) - (i-k)) \cdot a_i \right) \\
&= \sum_{i=0}^{k-1} a_i + a_k - a_{k-1} - \sum_{i=k+2}^{n-1} a_i \\
&= \sum_{i=0}^{k} a_i - \sum_{i=k+1}^{n-1} a_i
\end{aligned}
$$

Ukupne brojeve stanova pre i posle date zgrade možemo takođe računati inkrementalno (pri prelasku na narednu zgradu, prvi broj se uvećava, a drugi umanjuje za broj stanova u tekućoj zgradi).

Dakle, u programu možemo da pamtimo tri stvari: dužinu kablova d_k ako je ruter na poziciji k, ukupan broj stanova pre_k pre zgrade k (ne uključujući nju) i ukupan broj stanova $posle_k$ od zgrade k (uključujući nju) do kraja. Na početku, kada je $k = 0$, prvi broj d_0 moramo eksplicitno izračunati kao $\sum_{i=1}^{n-1} i \cdot a_i$, drugi broj treba inicijalizovati na nulu $pre_0 = 0$, a treći na ukupan broj svih stanova $posle_k = \sum_{i=0}^{n-1} a_i$. Zatim za svako k od 1 do $n-1$ računamo $pre_k = pre_{k-1} + a_{k-1}$, $posle_k = posle_{k-1} - a_{k-1}$ i zatim $d_k = d_{k-1} + pre_k - posle_k$.

Primer 3.2.1. *Ilustrujmo izvršavanje ovog algoritma na primeru zgrada u kojima živi redom* $3, 5, 1, 6, 2, 4$ *stanara.*

k	d_k	pre_k	$posle_k$
0	*53*	*0*	*21*

k	d_k	pre_k	$posle_k$
1	38	3	18
2	33	8	13
3	30	9	12
4	39	15	6
5	52	17	4

```
long long minDuzinaKablova(const vector<int>& broj_stanova) {
  // broj zgrada
  int n = broj_stanova.size();
  // krećemo od zgrade 0
  // ukupna dužina kablova ako je ruter u tekućoj zgradi
  long long duzina_kablova = 0;
  for (int i = 0; i < n; i++)
    duzina_kablova += broj_stanova[i] * i;
  // broj stanova pre tekuće zgrade
  long long stanova_pre = 0;
  // broj stanova od tekuće zgrade do kraja
  long long stanova_posle = 0;
  for (int i = 0; i < n; i++)
    stanova_posle += broj_stanova[i];

  // minimalna duzina kablova
  long long min_duzina_kablova = duzina_kablova;

  // obrađujemo sve zgrade od 1 do n-1
  for (int k = 1; k < n; k++) {
    // ažuriramo brojeve stanova
    stanova_pre += broj_stanova[k-1];
    stanova_posle -= broj_stanova[k-1];
    // ažuriramo duzinu kablova
    duzina_kablova += stanova_pre - stanova_posle;
    // ažuriramo minimum ako je potrebno
    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
```

```
    }

    return min_duzina_kablova;
}
```

Primetimo da je moguće izvršiti i malu optimizaciju (doduše koja neće popraviti asimptotsku složenost) na osnovu monotonosti niza d_k i petlju prekinuti čim se broj d_k prvi put poveća.

Pošto je i za jednu i za drugu fazu potrebno vreme $O(n)$, to je ujedno složenost ovog algoritma.

3.3 Odsecanje u pretrazi

Jedan od osnovnih principa za dobijanje efikasnijih algoritama i programa je da računar ne treba da izračunava stvari za koje se unapred može proceniti da nisu potrebne za dobijanje konačnog rešenja problema. Važan primer ovog principa se javlja kod algoritama pretrage. Pretragu elemenata ne treba eksplicitno vršiti među elementima za koje se može unapred utvrditi da ne mogu da zadovolje uslov pretrage. Kada preskočimo proveru takvih elemenata, kažemo da smo učinili *odsecanje u pretrazi*. Sličan princip se primenjuje i kada se vrši optimizacija tj. traži najveći (odnosno najmanji) element. Tada se može preskočiti eksplicitna provera elemenata za koje se unapred može dokazati da su manji (odnosno veći) od traženog maksimuma (odnosno minimuma).

Da bi se osigurala korektnost algoritama u kojima se vrši odsecanje, potrebno je nesumnjivo utvrditi da je odsecanje opravdano i da se u delu prostora pretrage koji se ne ispituje zaista ne može nalaziti rešenje problema.

U nastavku ovog poglavlja ćemo kroz određen broj primera prikazati kako se odsecanjem postiže asimptotski efikasniji algoritam. Jedan od najznačajnijih primera odsecanja predstavlja *binarna pretraga*, koja će, zbog svog značaja biti analizirana u posebnom poglavlju 3.6. Odsecanje se primenjuje i u drugim oblicima pretrage (bektreking pretrazi, pretrazi u dubinu, pretrazi u širinu), o čemu će više biti reči u kasnijim poglavljima.

Zadatak: Prost broj

Napiši program koji ispituje da li je uneti prirodan broj prost (veći je od 1 i nema drugih delilaca osim 1 i samog sebe).

Opis ulaza

Sa standardnog ulaza se unosi prirodan broj n ($1 \leq n \leq 10^9$).

Opis izlaza

Na standardni izlaz ispisati `DA` ako je broj n prost tj. `NE` ako nije.

Primer 1		**Primer 2**	
Ulaz	*Izlaz*	*Ulaz*	*Izlaz*
`17`	`DA`	`903543481`	`NE`

Rešenje

Linearna pretraga svih potencijalnih delilaca

Prirodan broj je prost ako je veći od 1 i ako nije deljiv ni sa jednim brojem osim sa 1 i sa samim sobom. Po definiciji broj 1 nije prost. Dakle, broj veći od 1 je prost ako nema ni jednog pravog delioca. Potrebno je dakle proveriti da li neki element skupa potencijalnih delilaca stvarno deli broj n (brojeva od 2 do $n-1$). Implementacija se zasniva na algoritmu linearne pretrage. Naivna implementacija proverava sve elemente skupa brojeva od 2 do $n-1$.

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;
  for (int i = 2; i < n; i++)
    if (n % i == 0)
      return false;
  return true;
}
```

Pošto se provera svakog delioca izvršava izračunavanjem jednog ostatka pri deljenju, u složenosti $O(1)$, složenost ovog pristupa odgovara broju delilaca i jednaka je $O(n)$.

Odsecanje u pretrazi

Delioci broja se uvek javljaju u paru. Na primer, delioci broja 100 organizovani po parovima su (1, 100), (2, 50), (4, 25) (5, 20) i (10, 10). Ako je i delilac broja n, delilac je i broj $\frac{n}{i}$. Pri tom, ako je $i \geq \sqrt{n}$, tada je $\frac{n}{i} \leq \sqrt{n}$. Dakle, važi sledeća teorema.

Teorema. Prirodan broj $n \geq 2$ ima prave delioce koji su veći ili jednaki vrednosti $\sqrt{n}$ ako i samo ako ima delioce koji su manji ili jednaki vrednosti $\sqrt{n}$.

Ova teorema nam daje mogućnost da pretragu potencijalnih delilaca redukujemo samo na interval $[2, \sqrt{n}]$, jer ako broj nema delilaca manjih ili jednakih vrednosti $\sqrt{n}$, onda ne može da ima delilaca većih ili jednakih toj vrednosti, tj. nema pravih delilaca i prost je. Ovo je primer algoritma u kom se efikasnost značajno popravlja tako što je eliminisan (odsečen) značajan deo prostora pretrage za koji smo uspeli da dokažemo da ga nije neophodno proveravati.

Sama implementacija je jednostavna i zasniva se ponovo na algoritmu linearne pretrage. U posebnoj funkciji na početku proveravamo specijalan slučaj broja 1. Nakon toga, u petlji proveravamo potencijalne delioce od 2 do $\sqrt{n}$. Jedan način da odredimo gornju granicu je da upotrebimo bibliotečku funkciju `sqrt`. Međutim, rad sa realnim brojevima je moguće u potpunosti izbeći tako što se umesto uslova $i \leq \sqrt{n}$ upotrebi uslov $i \cdot i \leq n$. Za svaku vrednost i proverava se da li je delilac broja i (izračunavanjem ostatka pri deljenju). Čim se utvrdi da je i delilac broja n funkcija može da vrati `false` (time se ujedno prekida izvršavanje petlje). Na kraju petlje, funkcija može da vrati `true`, jer nije pronađen nijedan delilac manji ili jednak od $\sqrt{n}$, pa na osnovu teoreme koje smo dokazali ne može postojati ni jedan delilac iznad te vrednosti i broj je prost.

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;
  for (int i = 2; i*i <= n; i++)
    if (n % i == 0)
      return false;
  return true;
}
```

Složenost ovog algoritma je $O(\sqrt{n})$. Ovo skraćivanje intervala pretrage veoma je značajno (ako je najveći broj oko 10^9 tj. oko milijardu, umesto milijardu delilaca potrebno je proveravati samo njih koren iz milijardu, što je tek nešto iznad trideset hiljada).

Naravno, isti algoritam se može implementirati i na drugačije načine.

Provera samo neparnih brojeva

Još jedna moguća optimizacija zasnovana na dodatnom odsecanju je da se na početku proveri da li je broj paran a da se nakon toga proveravaju samo neparni delioci (jer neparan broj ne može imati parne delioce).

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1) return false;     // broj 1 nije prost
  if (n == 2) return true;      // broj 2 jeste prost
  if (n % 2 == 0) return false; // ostali parni nisu prosti
  // proveravamo neparne delioce od 3 do korena iz n
  for (int i = 3; i*i <= n; i += 2)
    if (n % i == 0)
      return false;
  // nismo nasli delioca - broj jeste prost
  return true;
}
```

Složenost ovog algoritma je $O(\sqrt{n})$, ali se proverom samo neparnih brojeva konstantni faktor smanjio dva puta. Dakle, ova optimizacija ne donosi previše. Kada je n prost broj oko milijardu, obilazak do korena smanjuje broj potencijalih kandidata sa milijarde na tek tridesetak hiljada, a provera samo neparnih delilaca taj broj smanjuje na petnaestak hiljada, što je znatno manja ušteda.

Provera samo brojeva oblika $6k-1$ ***i*** $6k+1$

Program se još malo može ubrzati ako se primeti da su svi prosti brojevi veći od 2 i 3 oblika $6k-1$ ili $6k+1$, za $k \geq 1$ (naravno, obratno ne važi). Zaista, brojevi oblika $6k$, $6k+2$ i $6k+4$ su sigurno parni tj. deljivi sa 2, brojevi oblika $6k+3$ su deljivi sa 3, tako da su jedini preostali $6k+1$ i $6k+5$, pri čemu su ovi drugi sigurno oblika $6k'-1$ (za $k'=k+1$). Dakle, umesto da proveravamo deljivost sa svim neparnim brojevima manjim od korena, možemo proveravati deljivost sa svim brojevima oblika $6k-1$ ili $6k+1$, čime izbegavamo proveru sa jednim na svaka tri neparna broja i program ubrzamo shodno tome.

Primetimo da se petlja zaustavlja kada je $6k-1 > \sqrt{n}$ (tada sigurno važi i $6k+1 > \sqrt{n}$).

Složenost ovog algoritma je $O(\sqrt{n})$, ali se proverom samo brojeva oblika 6k-1 i 6k+1 konstantni faktor smanjio tri puta u odnosu na prvi algoritam ove složenosti.

Moguće su još neke optimizacije konstantnih faktora u prethodnim kodovima. Na primer, umesto da se u uslovu petlje množenjem izračunava kvadrat broja, on se može izračunati inkrementalno, uvećavajući kvadrat prethodnog broja. Naime, važi da je $(i+1)^2 = i^2 + 2i + 1$, pa se $(i+1)^2$ može dobiti uvećavanjem i^2 za $2i+1$, a ta vrednost se može izračunati bez množenja (pomeranjem bitova i sabiranjem). Međutim, ubedljivo najznačajnije ubrzanje je ono asimptotsko, nastalo odsecanjem u pretrazi i njime je program za brojeve reda veličine 10^9 ubrzan nekoliko desetina hiljada puta, dok su sve naredne opisane optimizacije ubrzavaju program te po nekoliko puta.

Ako je potrebno za više brojeva odjednom proveriti da li su prosti, umesto proveravanja svakog pojedinačnog, bolje je upotrebiti Eratostenovo sito.

Zadatak: Eratostenovo sito

Napisati program koji određuje broj prostih brojeva u intervalu $[a, b]$ i njihov zbir (ako zbir ima više od 6 cifara, ispisati samo ostatak pri deljenju sa $1\,000\,000$).

Opis ulaza

Sa standardnog ulaza unose se brojevi a i b ($1 \leq a \leq b \leq 10^7$).

Opis izlaza

Na standardnom izlazu prikazati broj prostih brojeva iz intervala $[a, b]$ i traženi zbir.

Primer

Ulaz	*Izlaz*
`1`	`168 76127`
`1000`	

Rešenje

Pojedinačne provere prostih brojeva

Očigledan algoritam za određivanje svih prostih brojeva iz nekog intervala jeste da se za svaki broj iz tog intervala pojedinačno proveri da li je prost.

Na osnovu specifikacije zadatka potrebno je odrediti najviše 6 poslednjih cifara zbira svih prostih brojeva iz intervala $[a, b]$, što, je ekvivalentno određivanju zbira tih brojeva po modulu 10^6. Naime, važi $(a+b) \bmod m = (a \bmod m + b \bmod m) \bmod m$.

U petlji prolazimo kroz sve brojeve od a do b, vršimo filtriranje na osnovu uslova da je broj prost i vršimo brojanje i sabiranje dobijene filtrirane serije.

Napomenimo da se zbir računa tako što se na početku inicijalizuje na nulu, a zatim se u svakom koraku izračunava sabiranje zbira i tekućeg prostog broja po modulu 10^6. Pošto će u svakom koraku zbir biti manji od 10^6, i pošto ne postoji opasnost od prekoračenja kada se u obzir uzme maksimalna vrednost prostih brojeva koji se sabiraju (pretpostavljajući da tip `int` može da predstavi brojeve bar do 10^9), sabiranje možemo vršiti naredbom `zbir = (zbir + p) % 1000000`.

Ako se provera da li je dati broj k prost vrši u složenosti $O(\sqrt{k})$, tada je ovaj algoritam složenosti $O((b-a)\sqrt{b})$. Ako je interval oblika $[0, n]$, složenost je $O(n\sqrt{n})$.

Eratostenovo sito

Bolji rezultat od ispitivanja za svaki broj pojedinačno da li je prost može se dobiti primenom algoritma poznatog kao *Eratostenovo sito*. Osnovna ideja algoritma je da se prvo napišu svi brojevi od 1 do datog broja n, zatim da se precrta broj 1 (jer on po definiciji nije prost), nakon njega svi umnošci broja 2 osim broja 2 (oni nisu prosti zato što su deljivi sa 2, dok broj 2 ostaje neprecrtan jer je on prost), zatim svi umnošci broja 3 osim broja 3 (oni nisu prosti jer su deljivi brojem 3), zatim umnošci broja 5 osim broja 5 (oni nisu prosti zato što su deljivi brojem 5) i tako dalje.

Efikasna implementacija ovog algoritma podrazumeva nekoliko odsecanja (kojima se izbegava ponavljanje istih operacija više puta i asimptotski ubrzava algoritam).

Prvo i najvažnije, umnoške složenih brojeva nema potrebe posebno precrtavati jer su oni već precrtani tokom precrtavanja umnožaka nekog od njihovih prostih faktora (na primer, nema potrebe posebno precrtavati umnoške broja 4 jer su oni već precrtani tokom precrtavanja umnožaka broja 2). Dakle, kada naiđemo na precrtan broj, njegove umnoške ne precrtavamo.

Drugo, prilikom precrtavanja umnožaka broja d dovoljno je krenuti od $d \cdot d$ jer su manji umnošci već precrtani ranije (svi imaju prave faktore manje od d). Zato je potrebno je da se postupak ponavlja samo dok se ne precrtaju umnošci svih onih prostih brojeva koji nisu veći od korena broja n. Za brojeve veće od korena od n precrtavanje bi krenulo od njihovog kvadrata koji je veći od n, pa je jasno da se ni za jedan od njih ništa dodatno ne bi precrtalo.

Brojevi koji su ostali neprecrtani su prosti (jer znamo da nemaju pravih delilaca manjih ili jednakih korenu od n, pa samim tim i manjih ili jednakih svom korenu, a

pošto nemaju delilaca ispod vrednosti korena, nemaju pravih delilaca ni iznad vrednosti korena).

Primer 3.3.1. *Prikažimo kako se ovim algoritmom određuju svi prosti brojevi od 2 do 50. Krećemo od pune tabele u kojoj su upisani svi brojevi od 2 do 50.*

```
 .  2  3  4  5  6  7  8  9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
```

U prvom koraku precrtavamo sve umnoške broja 2 (osim samog broja 2).

```
 .  2  3  .  5  .  7  .  9  .
11  . 13  . 15  . 17  . 19  .
21  . 23  . 25  . 27  . 29  .
31  . 33  . 35  . 37  . 39  .
41  . 43  . 45  . 47  . 49  .
```

U narednom koraku precrtavamo sve umnoške broja 3*, krenuvši od njegovog kvadrata tj. od* 9 *(broj* 6 *je već precrtan kao umnožak broja* 2*).*

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  . 25  .  .  . 29  .
31  .  .  . 35  . 37  .  .  .
41  . 43  .  .  . 47  . 49  .
```

Umnoške broja 4 *ne precrtavamo, jer je on precrtan (pa samim tim i svi njegovi umnošci).*

U narednom koraku precrtavamo sve umnoške broja 5*, krenuvši od njegovog kvadrata tj. broja* 25 *(umnošci* $2 \cdot 5$*,* $3 \cdot 5$ *i* $4 \cdot 5$ *su već precrtani).*

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  .  .  .  .  . 29  .
31  .  .  .  .  . 37  .  .  .
41  . 43  .  .  . 47  . 49  .
```

Umnoške broja 6 *ne precrtavamo, jer je on precrtan (pa samim tim i svi njegovi umnošci).*

Precrtavamo umnoške broja 7*, krenuvši od njegovog kvadrata tj. broja* 49*.*

```
 .  2  3  .  5  .  7  .  .  .
11  . 13  .  .  . 17  . 19  .
 .  . 23  .  .  .  .  . 29  .
31  .  .  .  .  . 37  .  .  .
41  . 43  .  .  . 47  .  .  .
```

Precrtavanje svih narednih neprecrtanih brojeva bi krenulo od njihovog kvadrata, međutim, ti kvadrati su već van tabele (jer su veći od 50), pa se postupak može završiti. Brojevi koji su ostali neprecrtani su prosti.

Precrtavanje brojeva modelovaćemo nizom (ili vektorom) koji sadrži logičke vrednosti (vrednosti tipa `bool`) i precrtane brojeve označavaćemo sa `false`, a neprecrtane sa `true`. Određivanje prostih brojeva (pomoću pomenutog niza tj. vektora) realizovaćemo u zasebnoj funkciji, jer ta funkcija može biti korisna i u mnogim narednim zadacima.

Recimo i da je bez obzira na to što su nama potrebni samo brojevi iz intervala od a do b, u Eratostenovom situ potrebno vršiti analizu svih brojeva iz intervala od 0 do b (jer se precrtavanje mora vršiti i brojevima manjim od a).

```
// funkcija koja popunjava logicki niz podacima o prostim
// brojevima iz intervala [0, n]
void Eratosten(vector<bool>& prost, int n) {
  // alociramo potreban prostor
  prost.resize(n + 1, true);
  prost[0] = prost[1] = false; // 0 i 1 nisu prosti
  // brojevi ciji se umnosci precrtavaju
  for (int i = 2; i * i <= n; i++)
    // nema potrebe precrtavati umnoske slozenih brojeva
    if (prost[i]) {
      // precrtavamo umnoske broja i i to krenuvsi od i*i
      for (int j = i * i; j <= n; j += i)
        prost[j] = false;
    }
}

// funkcija odredjuje broj i zbir po modulu 1000000 prostih
// brojeva iz intervala [a, b]
```

```
void prostiUIntevalu(int a, int b, int& broj, int& zbir) {
  // odredjujemo proste brojeve u intervalu [0, b]
  vector<bool> prost;
  Eratosten(prost, b);

  // analiziramo jedan po jedan broj u intervalu
  zbir = 0; broj = 0;
  for (int i = a; i <= b; i++)
    if (prost[i]) {
      zbir = (zbir + i) % 1000000;
      broj++;
    }
}
```

Analiza složenosti je komplikovanija i zahteva određeno poznavanje teorije brojeva. Procenimo broj izvršavanja tela unutrašnje petlje. U početnom koraku spoljne petlje precrtava se oko $\frac{n}{2}$ elemenata. U narednom, oko $\frac{n}{3}$. U narednom koraku je broj 4 već precrtan, pa se ne precrtava ništa. U narednom se precrtava oko $\frac{n}{5}$, nakon toga opet ništa, zatim $\frac{n}{7}$ itd. U poslednjem koraku se precrtava oko $\frac{n}{\sqrt{n}}$ elemenata. Dakle, broj precrtavanja je najviše

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \ldots + \frac{n}{\sqrt{n}} = n \cdot \left(\sum_{\substack{d \text{ prost}, \\ d \leq \sqrt{n}}} \frac{1}{d} \right)$$

Broj je zapravo i manji, jer prilikom precrtavanja u unutrašnjoj petlji precrtavanje ne krećemo od d, već od d^2, ali za potrebe lakšeg određivanja gornje granice složenosti koristićemo prethodnu ocenu.

Zbir $H(m) = 1 + 1/2 + 1/3 + \ldots + 1/m = \sum_{d \leq m} \frac{1}{d}$ (takozvani harmonijski zbir) se asimptotski ponaša slično funkciji $\log m$ (razlika između ove dve funkcije teži takozvanoj Ojler-Maskeronijevoj konstanti $\gamma \approx 0.5772156649$), pa samim tim znamo da taj zbir divergira. Kada se sabiranje vrši samo po prostim brojevima, tada se zbir ponaša kao logaritam harmonijskog zbira, tj. kao $\log \log m$ (pa je i on divergentan). Dakle, u našem primeru možemo zaključiti da je broj precrtavanja jednak $n \cdot \log \log \sqrt{n}$. Pošto je $\log \log \sqrt{n} = \log \log n^{\frac{1}{2}} = \log \left(\frac{1}{2} \log n \right) =$

$\log \frac{1}{2} + \log \log n$, pod pretpostavkom da je sabiranje brojeva (koje se koristi u implementaciji petlji) konstantne složenosti, važi da je složenost Eratostenovog sita $O(n \cdot \log \log n)$. Iako nije linearna, funkcija $\log \log n$ toliko sporo raste, da se za sve praktične potrebe Eratostanovo sito može smatrati linearnim u odnosu na n (što je dosta sporije samo od ispitivanja da li je broj n prost, što ima složenost $O(\sqrt{n})$, ali je brže od proveravanja svakog broja pojedinačno koje je složenosti $O(n\sqrt{n})$).

Zadatak: Maksimalni zbir segmenta

Napiši program koji određuje najveći zbir nekog segmenta (podniza uzastopnih elemenata) datog niza.

Opis ulaza

Sa standardnog ulaza se unosi broj n ($1 \leq n \leq 50\,000$), a zatim n celih brojeva između -10 i 10, razdvojenih razmakom.

Opis izlaza

Na standardni izlaz ispiši traženi zbir.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`6` `2 -3 4 -1 3 -2`	`6`	Segment najvećeg zbira je $4, -1, 3$.

Rešenje

Gruba sila

Najdirektniji mogući način da se zadatak reši je da se izračuna zbir svakog segmenta. Zbir svakog segmenta možemo izračunavati zasebno (u petlji ili bibliotečkom funkcijom). Efikasnije rešenje dobijamo ako segmente nabrajamo redom (ugnežđenim petljama, gde spoljašnja petlja nabraja redom leve, a unutrašnja desne krajeve segmenata) i zbir narednog segmenta izračunavamo inkrementalno, na osnovu zbira prethodnog segmenta.

U nastavku je prikazana implementacija algoritma u kom se zbir izračunava inkrementalno.

```
int maksZbirSegmenta(const vector<int>& a) {
  int n = a.size();
  int max = 0;
```

```
  for (int i = 0; i < n; i++) {
    int z = 0;
    for (int j = i; j < n; j++) {
      z += a[j];
      if (z > max)
        max = z;
    }
  }
  return max;
}
```

Ako zbir svakog segmenta računamo nezavisno, sabiranjem njegovih elemenata (bilo u petlji, bilo pomoću bibliotečke funkcije), složenost rešenja je $O(n^3)$. Ako zbirove segmenata računamo inkrementalno, dobijamo algoritam složenosti $O(n^2)$. U oba slučaja elemente učitavamo u niz i memorijska složenost je $O(n)$.

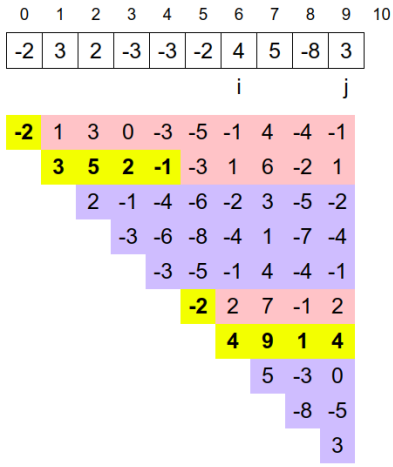
Odsecanje nepotrebnih provera

Algoritam zasnovan na proveri svih segmenata se slobodno može nazvati trivijalnim, jer se do njega dolazi prilično direktno i veoma jednostavno mu se i dokazuje korektnost i analizira složenost. Međutim, on je prilično neefikasan za rešavanje ovog problema, čak i kada se zbirovi računaju inkrementalno. Značajno unapređenje možemo dobiti kada primetimo da veliki broj segmenata uopšte ne moramo da obrađujemo, jer iz nekih drugih razloga znamo da njihov zbir ne može biti maksimalan.

Posmatrajmo niz `-2 3 2 -3 -3 4 -2 5 -8 3` i zbirove svih njegovih nepraznih segmenata.

Prekid posle negativnog zbira

Razmotrimo bilo koji niz koji počinje negativnim brojem. Nijedan segment koji počinje tim brojem, ne može biti segment maksimalnog zbira, pošto se izostavljanjam tog broja dobija segment većeg zbira. Ovo svojstvo je i opštije. Ukoliko segment počinje prefiksom negativnog zbira, on ne može biti segment maksimalnog zbira, jer se izostavljanjem tog prefiksa dobija segment većeg zbira. Otud, pri inkrementalnom proširivanju intervala udesno, čim se ustanovi da je tekući zbir negativan, moguće je prekinuti dalje proširivanje i odmah preći na narednu narednu početnu poziciju (tj. narednu vrstu imajući u vidu sliku 3.2).



Slika 3.2: Maksimalni zbir segmenta. Žutom bojom su obeleženi zbirovi koji se proveravaju, crvenom oni koji se mogu preskočiti jer im prethodni negativni zbir, a plavom oni koji se mogu preskočiti jer se iznad početka njihove vrste nalazi pozitivan zbir.

Na primer, čim vidimo da je prvi element prvog segmenta `-2`, možemo prekinuti dalju obradu elemenata prve vrste, jer će svi elementi druge vrste sigurno biti za dva veći nego odgovarajući elementi prve vrste (`3` je veće od `1`, `5` je veće od `3`, `2` je veće od `0` itd.).

Slično, kada se prilikom proširivanja segmenta koji počinje na poziciji `1` (od elementa `3`) dođe do toga da je parcijalni zbir `-1` (što se dešava kada se izračuna zbir $3+2-3-3=-1$, možemo prekinuti sa obradom daljih segmenata koji počinju na toj poziciji, jer smo sigurni da će za svaki od njih kasnije veći biti onaj koji se dobija izostavljanje prefiksa `3 2 -3 -3` (čiji je zbir `-1`). Zaista, od preostalih zbirova `3 1 6 -2 1` u drugoj vrsti za jedan su veći zbirovi `-2 2 7 -1 2` u šestoj vrsti koji su dobijeni izostavljanjem tog prefiksa. Obratimo pažnju na to da prekid unutrašnje petlje na ovaj način uzrokuje da se maksimalna vrednost u tekućoj vrsti ne mora uopšte naći. Petlja koja obrađuje drugu vrstu će biti prekinuta čim se naiđe na zbir `-1`, kada je tekuća vrednost maksimuma `5` iako je maksimum te vrste `6`. Sigurni smo da će u nekoj narednoj vrsti postojati veća vrednost od te najveće (zaista, u šestoj vrsti se javlja `7`), pa nam nalaženje stvarnog maksimuma u tekućoj vrsti uopšte nije neophodno.

Iako se na ovaj način može preskočiti razmatranje nekih segmenata, u najgorem

slučaju složenost nije smanjena. Na primer, u slučaju da su elementi niza strogo pozitivni, zbir nikad ne postaje negativan i složenost nakon ovog isecanja je i dalje kvadratne složenosti tj. $O(n^2)$.

Odsecanje provere početaka unutar pozitivnog segmenta

Ako su svi elementi polaznog niza pozitivni, maksimalan zbir biva nađen za $i = 0$ i $j = n - 1$. Nakon toga se, uvećavanjem indeksa i, zbir smanjuje pošto se svakim skraćivanjem segmenta sleva izostavlja neki pozitivan broj koji doprinosi zbiru. I ovo zapažanje se može uopštiti. Ne samo što je nepoželjno skratiti interval sleva za neki pozitivan broj, već je nepoželjno skratiti ga za bilo koji prefiks čiji je zbir pozitivan. Pitanje je dokle takvi prefiksi sežu? Bar do elementa čijim obuhvatanjem dobijamo prvi negativan prefiks. Otud segment maksimalnog zbira ne može počinjati ni na jednoj poziciji između tekuće početne pozicije i prve pozicije na kojoj zbir postaje negativan.

U navedenom primeru, maksimalni segment ne može počinjati na poziciji `2`, jer se proširivanjem nalevo i dodavanjem elementa `3` sa pozicije `1` dobijaju sigurno zbirovi koji su veći za tri. Dakle, svi elementi druge vrste (koja odgovara poziciji `1` u nizu) su za `3` veći od odgovarajućih elemenata treće vrste (koja odgovara poziciji `2` u nizu). Zaista, `5` je veće od `2`, `2` od `-1` itd. Slično, ti elementi su za `5` veći od odgovarajućih elemenata četvrte vrste (koja odgovara poziciji `3` u nizu). Zaista, `2` je veće od `-3`, `-1` od `-6` itd. Oni su za `2` veći od odgovarajući elemenata pete vrste (koja odgovara poziciji `4` u nizu). Zaista, `-1` je veće od `-3`, `-3` je veće od `-5` itd. Zato te tri vrste uopšte nema potrebe razmatrati.

Zahvaljujući ovom zapažanju, pri završetku obrade jedne vrste i prelasku na narednu, nije neophodno uvećavati promenljivu i za jedan, već je moguće nastaviti iza elementa čijim je uključivanjem zbir postao negativan.

Pošto se svaki element obrađuje samo jednom, prilikom implementacije nije neophodno sve elemente pamtiti u nizu.

```
int maksZbirSegmenta(const vector<int>& a) {
  int n = a.size();
  int max = 0;
  int i = 0;
  while (i < n) {
    int z = 0;
    int j;
```

```
    for (j = i; j < n; j++) {
      z += a[j];
      if (z < 0)
        break;
      if (z > max)
        max = z;
    }
    i = j + 1;
  }
  return max;
}
```

Primer 3.3.2. *Na slici 3.2 je ilustrovan rad algoritma na primeru niza* `-2 3 2 -3 -3 4 -2 5 -8 3`.

- *Na početku se promenljiva i inicijalizuje na 0 i kreće se sa obradom prve vrste. Zbir z se inicijalizuje na 0, a promenljiva j na vrednost promenljive i, što je 0. Pošto već u prvom koraku unutrašnje petlje vrednost z postaje negativna (-2), unutrašnja petlja se prekida, i nakon toga se vrednost promenljive i postavlja na $j + 1 = 1$ (dakle, prelazi se na drugu vrstu). Ostali zbirovi prve vrste se ne izračunavaju (ovo odsecanje je opravdano jer oni odgovaraju segemntima koji počinjinju negativnom vrednošću -2, pa se veći zbirovi mogu dobiti odbacivanjem te vrednosti).*

- *Obrada elemenata druge vrste ($i = 1$) počinje od pozicije $j = i = 1$. Promenljiva z se ponovo inicijalizuje na nulu, a zatim se uvećava za jedan po jedan element niza, sve dok joj vrednost ne postane negativna (što se prvi put dešava kada je $j = 4$ i tada je $z = -1$). Pri tom se u svakom koraku ažurira vrednost maksimuma z_{max} i on dostiže vrednost 5. Prekidom unutrašnje petlje preskočeno je računanje i analiziranje elemenata druge vrste iza vrednosti -1 (što je opravdano, jer oni odgovaraju segmentima koji počinju prefiksom* `3 2 -3 -3` *čiji je zbir -1 negativan). Nakon prekida unutrašnje petlje, vrednost promenljive i se postavlja na $j + 1 = 5$. To znači da se naredne tri vrste mogu preskočiti (u njima se nalaze zbirovi segmenata koji se od segmenta čiji se zbirovi nalaze u tekućoj vrsti dobijanjem izbacivanjem početnog pozitivnog prefiksa, pa su sigurno manji od njih).*

- *Obrada elemenata šeste vrste ($i = 5$) počinje od vrednosti $j = i = 5$. Promenljiva z se ponovo inicijalizuje na 0, i već nakon obrade prvog elementa ($j = 5$) postaje negativna (-2). Nakon prekida unutrašnje petlje, promenljiva i se postavlja na vrednost $j + 1 = 6$.*

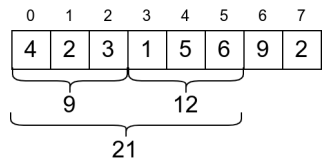
- *Na kraju se obrađuje sedma vrsta ($i = 6$). Promenljiva z se postavlja na nulu, a j na $i = 6$. Zbir z se zatim uvećava za jedan po jedan element i pošto ni u jednom trenutku ne postaje negativan stiže se do kraja niza ($j = 10$). Tokom unutrašnje petlje ažurira se maksimum i dostiže vrednost 9. Po završetku unutrašnje petlje vrednost promenljive i se postavlja na $j+1 = 11$ i spoljašnja petlja se završava. Izračunavanje vrednosti i analiza elemenata u poslednje tri vrste je opravdano preskočena.*

Pošto obe promenljive prolaze kroz raspon od 0 do n i kreću se samo u jednom smeru (vrednost im se samo povećava i nikada ne smanjuje), složenost ovog rešenja je linearna tj. $O(n)$. U prikazanoj implementaciji elementi se čuvaju u nizu pa je i memorijska složenost linearna tj. $O(n)$, međutim, pošto se svaki element analizira samo jednom, za tim nema potrebe i moguće je napraviti i implementaciju konstantne memorijske složenosti.

3.4 Zbirovi prefksa, razlike susednih elemenata

Zbir svih elemenata niza na pozicijama u intervalu $[a, b]$ se može izračunati kao razlika između zbira svih elemenata na pozicijama u intervalu $[0, b]$ i zbira elemenata na pozicijama u intervalu $[0, a-1]$. Dakle, zbir bilo kog segmenta se može izračunati kao razlika dva zbira prefiksa.

Primer 3.4.1. *Na primer, razmotrimo kako da izračunamo zbir elemenata na pozicijama iz intervala [3, 5] (tj. na pozicijama 3, 4 i 5) u nizu 4, 2, 3, 1, 5, 6, 9, 2. Na tim pozicijama se nalaze elementi 1, 5 i 6 i zbir im je $1 + 5 + 6 = 12$. Zbir svih elemenata na pozicijama iz intevala $[0, 5]$ je $4 + 2 + 3 + 1 + 5 + 6 = 21$, dok je zbir svih elemenata na pozicijama iz intervala $[0, 2]$ jednak $4 + 2 + 3 = 9$. Razlika $21 - 9$ upravo je jednaka 12.*



Ova naizgled veoma jednostavna osobina sabiranja može značajno pomoći ubrzanju raznih algoritama u kojima su nam potrebni zbirovi segmenata tj. zbirovi uzastopnih elemenata niza. Naime, ako znamo *niz zbirova svih prefiksa* niza tj. zbirove na svim intervalima $[0, k)$, za $k = 0$ do n (a njih možemo izračunati tokom faze pretprocesiranja, inkrementalno, u linearnoj složenosti), tada u konstantnoj složenosti (jednim oduzimanjem) možemo izračunati zbir proizvoljnog segmenta niza.

Da rezimiramo, niz zbirova prefiksa $Z_k = \sum_{i=0}^{k-1} a_k$ izračunavamo na osnovu veza $Z_0 = 0, \quad Z_{k+1} = Z_k + a_k, 0 \leq k < n.$

Tada zbir segmenta $Z_{ab} = \sum_{i=a}^{b} a_k$, računamo efikasno na osnovu veze $Z_{ab} = Z_{b+1} - Z_a$.

Implementacija ove tehnike je veoma jednostavna.

```
vector<int> a(n);
...
// izracunavanje niza prefiksni zbirova
vector<int> ps(n+1);
ps[0] = 0;
for (int i = 0; i < n; i++)
   ps[i+1] = ps[i] + a[i];

// izracunavanje zbira segmenta odredjenog pozicijama [a, b]
int zbirSegmenta = ps[b+1] - ps[a];
```

U jeziku C++ parcijalne zbirove je moguće izračunati i korišćenjem bibliotečke funkcije `partial_sum`, koja, naravno, radi u linearnoj složenosti. Funkciji se prosleđuju dva iteratora na deo niza koji se sabira, kao i iterator na početak niza u koji se smeštaju rezultati (pošto se unapred zna koliko će elemenata biti, taj niz se unapred alocira).

```
// izracunavanje niza prefiksni zbirova
vector<int> ps(n+1);
partial_sum(begin(a), end(a), begin(ps));
```

Ipak, direktno inkrementalno izračunavanje niza zbirova prefiksa je toliko jednostavno da se ova funkcija ne koristi često.

Niz zbirova prefiksa datog niza možemo izračunati u linearnoj složenosti, ali važi i obratno. Od niza zbirova prefiksa, u linearnoj složenosti možemo izračunati elemente originalnog niza. Važi čak i jače tvrđenje od toga, jer svaki konkretni element niza možemo naći u konstantnoj složenosti, oduzimanjem dva susedna zbira prefiksa (za svako $0 \leq k < n$, važi $a_k = Z_{k+1} - Z_k$). Zato prelazak sa niza na zbirove njegovih prefiksa možemo smatrati promenom reprezentacije podataka čuvanjem istih podataka u efikasnijoj strukturi podataka (često nema smisla čuvati i jedno i drugo istovremeno u memoriji).

Primetimo ogromnu sličnost sa integralnim i diferencijalnim računom. Izračunavanje zbirova prefiksa odgovara određenom integraljenju, razlika zbirova prefiksa odgovara Njutn-Lajbnicovoj formuli, dok izračunavanje razlike susednih elemenata odgovara diferenciranju. Integraljenje i diferenciranje su međusobno inverzne operacije.

Dualan pristup zbirovima prefiksa je promena reprezentacije u kojoj umesto niza čuvamo niz razlika susednih elemenata.

$$R_0 = a_0, \quad R_k = a_k - a_{k-1}, 1 \leq k \leq n.$$

Povratak na orginalni niz se onda može izvršiti u linearnoj složenosti tako što izračunamo zbirove prefiksa niza razlika. Ova reprezentacija nam omogućava da veoma efikasno menjamo segmente niza tako što sve elemente iz nekog zadatog segmenta uvećamo ili umanjimo za neku fiksnu vrednost, što može biti veoma korisna operacija u nekim primenama (koje ćemo ilustrovati kroz zadatke).

Na ideji niza prefiksnih zbirova možemo izgraditi i strukturu podataka koja nam omogućava da se brzo izračuna koliko proizvoljni segment niza ima elemenata koji zadovoljavaju neki dati uslov. Dovoljno je izračunati niz zbirova prefiksa niza koji ima jedinice na mestima na kojima se u originalnom nizu nalazi element koji zadovoljava taj uslov i nule na ostalim mestima.

Primer 3.4.2. *Ako želimo da izračunamo koliko ima parnih brojeva u bilo kom segmentu niza* $3, 2, 4, 8, 1, 5, 7, 6$*, dovoljno je da formiramo niz* $0, 1, 1, 1, 0, 0, 0, 1$*, a zatim da izračunamo njegove prefiksne zbirove* $0, 0, 1, 2, 3, 3, 3, 3, 4$*. Tada, na primer, broj parnih brojeva između pozicija* 2 *i* 7 *(tj. između elemenata* 4 *i* 7*) u originalnom nizu možemo izračunati kao* $3 - 1 = 2$*. Zaista, zaključno sa elementom* 7 *postoje 3 parna broja (to su* 2*,* 4 *i* 8*), a zaključno sa elementom* 2 *postoji 1 paran broj (to je* 2*).*

Slično nizu zbirova prefiksa, možemo čuvati i niz vrednosti neke druge statistike. Ako se statistika izračunava operacijom koja ima inverznu operaciju (kao što je oduzimanje inverzna operacija operaciji sabiranja), onda je možemo koristiti na potpuno isti način kao niz zbirova prefiksa. Na primer, ako znamo da su svi elementi niza različiti od nule, tada možemo čuvati niz proizvoda prefiksa. Proizvod bilo kog segmenta možemo ostvariti deljenjem proizvoda dva prefiksa (kao što se zbir segmenta određuje oduzimanjem dva zbira prefiksa). Međutim, ako niz sadrži nulu, stvari se komplikuju jer će proizvod svakog prefiksa nakon te nule biti jednak nuli, pa ne možemo izvršiti deljenje. U tom slučaju je pored niza proizvoda prefiksa potrebno čuvati i niz u kome čuvamo broj nula do tekuće pozicije u originalnom nizu.

Ako operacija nema inverznu (poput operacija minimuma, maksimuma, NZD, NZS), tada njeno efikasno računanje za proizvoljni segment niza zahteva kompleksnije strukture podataka (na primer, segmentno drvo koje će detaljno biti opisano u narednim kursevima). Ponekad se traži računanje statistike niza koji se dobija izbacivanjem jednog ili više uzastopnih elemenata niza (tj. izbacivanjem nekog segmenta). Tu nam može pomoći istovremeno poznavanje niza statistika prefiksa i niza statistika sufiksa. Nizovi statistika prefiksa i sufiksa se mogu izračunati u vremenu $O(n)$, da bi se nakon toga statistike niza bez izbačenih pojedinačnih segmenata mogle računati u vremenu $O(1)$.

Primer 3.4.3. *Neka je dat niz* $3, 2, 4, 8, 1, 5, 7, 6$. *Tada je niz minimuma (nepraznih) prefiksa ovog niza jednak* $3, 2, 2, 2, 1, 1, 1, 1$, *a niz minimuma (nepraznih) sufiksa jednak* $1, 1, 1, 1, 1, 5, 6, 6$. *Sada efikasno možemo izračunati minimum niza dobijenog od polaznog kada se izbaci segment* $8, 1, 5$. *Minimum prefiksa pre elementa* 8 *je vrednost upisana na poziciji* 2 *u nizu prefiksa i to je* 2, *dok je minimum sufiksa posle elementa* 5 *upisan na poziciji* 6 *u nizu sufiksa i to je* 6. *Tražena vrednost je manja od te dve vrednosti* $\min(2, 6) = 2$.

Zadatak: Maksimalni zbir segmenta

Ovaj zadatak je ponovljen u cilju ilustrovanja različitih tehnika rešavanja.

Tekst zadatka.

Opis ulaza

Sa standardnog ulaza se unosi broj n ($1 \leq n \leq 50\,000$), a zatim n celih brojeva između -10 i 10, razdvojenih razmakom.

Opis izlaza

Na standardni izlaz ispiši traženi zbir.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
6 2 -3 4 -1 3 -2	6	Segment najvećeg zbira je $4, -1, 3$.

Rešenje

Zbirovi prefiksa

Analiziraćemo svaku poziciju u nizu i razmatraćemo sve segmente koji se na toj poziciji završavaju. Zbir elemenata svakog od njih se može izračunati kao razlika između zbira prefiksa do te tekuće pozicije i odgovrajućeg prefiksa pre te pozicije. Da bi razlika bila što veća, umanjilac mora biti što manji. Dakle, od svih segmenata koji se završavaju na tekućoj poziciji, najveći zbir se dobija oduzimanjem minimalnog zbira prefiksa pre tekuće pozicije od zbira prefiksa do tekuće pozicije. Minimalni zbir prefiksa niza ne moramo da određujemo svaki put iz početka, već i njega možemo inkrementalno da ažuriramo. Analiziramo jednu po jednu poziciju, za svaku određujemo najveći zbir segmenta koji se završava na toj poziciji i ako je ona veća od globalnog maksimuma, ažuriramo maksimum. Pošto se maksimalni segment sigurno završava na nekoj poziciji i najveći je od svih takvih, a pošto eksplicitno proveravamo sve pozicije, sigurni smo da će predloženi algoritam zaista pronaći maksimalni segment.

Složenost ovog rešenja je $O(n)$, pa je ovo rešenje optimalne složenosti.

```
int zbir_prefiksa = 0;
int min_zbir_prefiksa = zbir_prefiksa;
int max_zbir_segmenta = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  zbir_prefiksa += x;
  int zbir = zbir_prefiksa - min_zbir_prefiksa;
  if (zbir > max_zbir_segmenta)
    max_zbir_segmenta = zbir;
  if (zbir_prefiksa < min_zbir_prefiksa)
```

```
    min_zbir_prefiksa = zbir_prefiksa;
}
cout << max_zbir_segmenta << endl;
```

3.5 Primene sortiranja

Sortiranje niza je zadatak koji je sam po sebi interesantan ali i izuzetno važan jer ima mnogobrojne primene. Na primer, često se podaci sortiraju prilikom rangiranja (na primer, prilikom upisa na fakultet, da bi se odredio redosled kandidata). Pored toga, sortiranje je i izrazito značajna tehnika pretprocesiranja podataka, koja omogućava njihovu efikasniju obradu. Najznačajniji dobitak koji pruža sortiranje je to što je pretraga sortiranog niza neuporedivo brža nego kada niz nije sortiran, o čemu će više reči biti u poglavljima 3.6 i 3.7. U ovom poglavlju razmotrićemo još neke dodatne koristi od sortiranja. Na primer, u sortiranom nizu se elementi koji su bliski po vrednosti nalaze na bliskim pozicijama. Jednaki elementi su međusobno susedni. Ovo omogućava da se obrada duplikata vrši brzo a i da se veoma bliske vrednosti u nizu nalaze brzo. Sortirani niz predstavlja i jednu kanonsku reprezentaciju multiskupa podataka koje sadrži, pa se, na primer, provera da li dva niza sadrže iste elemente može efikasno izvršiti ako se oni najpre sortiraju.

Sortiranje je veoma jednostavna, a toliko korisna tehnika da se često savetuje da se prilikom konstrukcije algoritama pre bilo čega drugoga pokuša sa sortiranjem.

Ako nemamo nikakve posebne pretpostavke o sadržaju nizova, i ako pretpostavimo da se dva elementa niza mogu uporediti i razmeniti u vremenu $O(1)$, tada se niz može sortirati u vremenu $O(n \log n)$, gde je n broj elemenata niza. Postoji nekoliko efikasnih algoritama kojima se ovo može postići i o njima će više reči biti u narednim poglavljima. Biblioteke svih savremenih programskih jezika nude bibliotečke funkcije za efikasno sortiranje, koje implementiraju ove algoritme i niz sortiraju u vremenu $O(n \log n)$.

Čest je slučaj da neka obrada nesortiranog niza zahteva vreme $O(n^2)$ (na primer, potrebno je obraditi sve parove elemenata niza), a da obrada sortiranog niza zahteva vreme $O(\log n)$, $O(n)$ ili $O(n \log n)$ i tada se isplati sortirati niz pre obrade. Na primer, prebrojavanje različitih elemenata nesortiranog niza se mora vršiti u vremenu $O(n^2)$, a sortiranog niza može se izvršiti u vremenu $O(n)$, te se za prebrojavanje različitih elemenata isplati prvo sortirati niz.

S druge strane, ako neka obrada nesortiranog niza zahteva vreme $O(n)$, tada se sortiranje ne isplati, čak i ako se obrada sortiranog niza vrši u vremenu $O(1)$. Na primer, minimum i maksimum nesortiranog niza se mogu odrediti u vremenu $O(n)$, a sortiranog niza u vremenu $O(1)$, pa se za pronalaženje minimuma i maksimuma ne isplati sortirati niz (jer je za to potrebno vreme $O(n \log n)$.

Situacija se menja ako je obradu potrebno izvršiti više puta. Pretpostavimo da je, na primer, za više vrednosti potrebno proveriti da li su sadržane u nizu. Proveru da li je element sadržan u nesortiranom nizu možemo izvršiti u vremenu $O(n)$, a proveru da li je sadržan u sortiranom nizu u vremenu $O(\log n)$ (algoritmom binarne pretrage, prikazanom u poglavlju 3.6). Ako se vrši samo jedan upit tj. ako samo za jedan element proveravamo da li je sadržan u nizu, bolje je izvršiti linearnu pretragu niza u vremenu $O(n)$, nego najpre sortirati niz u vremenu $O(n \log n)$. Međutim, ako se vrši k upita, tada je bez sortiranja potrebno vreme $O(kn)$, a sa sortiranjem $O(n \log n) + O(k \log n)$. Kada je k reda veličine n tada se radi o razlici između kvadratnog vremena (u pristupu bez sortiranja) i kvazilinearnog vremena (u pristupu sa sortiranjem), što daje ogromnu razliku za velike vrednosti n.

U nekim problemima nepoželjno je da se tokom neke analize podataka podaci transformišu, jer to može da spreči neke naredne analize podataka. Na primer, ako sortiramo niz, dalje analize u kojima je važan redosled podataka neće biti moguće, jer se gubi informacija o polaznom redosledu. U takvim problemima je pre sortiranja neophodno napraviti kopiju niza.

U nastavku ćemo prikazati nekoliko tipičnih zadataka koji prikazuju kako se primenom sortiranja neki problemi rešavaju efikasnije.

3.5.1 Obrada duplikata

Kao što smo već najavili, sortiranje može pomoći prilikom obrade duplikata u nizu.

Zadatak: Duplikati

Pretpostavimo da su internet adrese predstavljene prirodnim brojevima (IP adrese se, na primer, čuvaju u obliku neoznačenih 32-bitnih brojeva). Pretraživač čuva spisak svih adresa koje je korisnik posetio tokom nekog prethodnog perioda. Korisnik je mnoge adrese posećivao i više puta. Napiši program koji određuje broj različitih adresa koje je korisnik posetio.

Opis ulaza

Sa standardnog ulaza se unosi broj n ($1 \leq n \leq 10^5$), a zatim i n prirodnih brojeva

(manjih od 2^{32}), svaki u posebnom redu.

Opis izlaza

Na standardni izlaz ispisati broj različitih adresa koje je korisnik posetio.

Primer

Ulaz	*Izlaz*
8 123456789 234567890 345678901 234567890 456789012 234567890 456789012 234567890	4

Rešenje

Linearna pretraga

Naivan način da se detektuju duplikati se može zasnovati na algoritmu linearne pretrage. Brojaćemo samo one članove niza koji se prvi put pojavljuju (kada se pojavi neki element koji se već pojavio ranije nećemo ga brojati). Proveru da li se element niza na poziciji i (od nula do $n-1$) ranije već pojavio vršićemo tako što ćemo proveriti da li se taj element javlja na nekoj poziciji j od 0 do $i-1$. To možemo uraditi algoritmom linearne pretrage (linearna pretraga se može vršiti i bibliotečkom funkcijom `find`).

Ovo rešenje je neefikasno i složenost mu je $O(n^2)$, gde je n broj elemenata niza.

```
// broj razlicitih elemenata niza a
int brojRazlicitih(const vector<unsigned>& a) {
  int broj = 0;
  // za svaki element niza a
  for (int i = 0; i < a.size(); i++) {
    // proveravamo da li se a[i] javlja pre pozicije i
    bool sadrzi = false;
    for (int j = 0; j < i && !sadrzi; j++)
      if (a[i] == a[j])
```

```
        sadrzi = true;
    // ako se ne pojavljuje, uracunavamo ga
    if (!sadrzi)
      broj++;
  }
  return broj;
}
```

Sortiranje

Jedan od najčešćih načina uklanjanja duplikata iz niza je zasnovan na sortiranju, jer se nakon sortiranja duplikati nađu jedan do drugog. Sortiranje možemo najbolje uraditi pozivom bibliotečke funkcije. Nakon sortiranja prolazimo redom kroz niz i brojimo prvi element, a zatim i sve elemente koji su različiti od njima prethodnog (to su prva pojavljivanja elemenata u sortiranom nizu).

Složenošću ovog pristupa dominira složenost postupka sortiranja. Prolaz nakon sortiranja je linearne složenosti, a sortiranje se može ostvariti u složenosti $O(n \log n)$, gde je n broj elemenata niza.

```
// broj razlicitih elemenata niza a
int brojRazlicitih(const vector<unsigned>& a) {
  // pravimo kopiju, da bi originalni niz ostao nepromenjen
  auto as = a;
  // sortiramo niz
  sort(as.begin(), as.end());
  // brojimo prvi element i sve elemente koje su razliciti od
  // svojih prethodnika
  int broj = 1;
  for (int i = 1; i < as.size(); i++)
    if (as[i] != as[i-1])
      broj++;
  return broj;
}
```

3.5.2 Grupisanje bliskih vrednosti

Nakon sortiranja bliske vrednosti se nalaze na bliskim pozicijama.

Zadatak: Najbliže sobe

Dva gosta su došla u hotel i žele da odsednu u slobodnim sobama koje su što bliže jedna drugoj, da bi tokom večeri mogli da zajedno rade u jednoj od tih soba. Ako postoji više takvih soba, oni biraju da budu što dalje od recepcije, tj. u sobama sa što većim rednim brojevima, kako im buka ne bi smetala. Napiši program koji određuje brojeve soba koje gosti treba da dobiju, pretpostavljajući da je poznat spisak slobodnih soba u tom trenutku.

Opis ulaza

U prvoj liniji standardnog ulaza nalazi se broj n ($1 \leq n \leq 10^5$), a zatim se nalaze brojevi slobodnih soba - svi brojevi su različiti, ali je njihov redosled proizvoljan.

Opis izlaza

Na standardni izlaz ispisati brojeve soba gostiju (prvo manji broj, pa veći), razdvojene jednim razmakom.

Primer

Ulaz	*Izlaz*
`7` `18 6 25 11 4 1 16`	`16 18`

Rešenje

Gruba sila

Zadatak može da se reši naivno tako što se izračunaju rastojanja između svake dve sobe i što se pronađe par sa najmanjim rastojanjima.

Pošto parova ima $\frac{n(n-1)}{2}$, složenost ovog pristupa je $O(n^2)$.

Sortiranje

Bolje rešenje se može dobiti ako se niz najpre sortira. Naime, najbliži element svakom elementu u sortiranom nizu je jedan od njemu susednih. Dakle, ako broj a_i učestvuje u paru najbližih soba, onda drugi element tog para može biti ili broj a_{i-1} koji je neposredno ispred a_i u sortiranom redosledu ili broj a_{i+1} koji je neposredno iza njega (naravno, ne postoji element ispred prvog, niti element iza poslednjeg elementa

niza). Zato je nakon sortiranja dovoljno proveriti sve razlike između susednih elemenata i odrediti najmanju od njih (ako ima više istih, određujemo poslednju). Za ovo koristimo algoritam određivanja najmanjeg elementa, dok sortiranje možemo najlakše izvršiti bibliotečkom funkcijom.

Sortiranje bibliotečkom funkcijom ima složenost $O(n \log n)$ operacija, dok je traženje minimuma složenosti $O(n)$, tako da je ukupno vreme opisanog postupka $O(n \log n)$.

```
void najblizeSobe(const vector<int>& a,
                  int& soba1, int& soba2) {
  // pravimo duplikat niza koji cemo da sortiramo
  auto b = a;
  sort(begin(b), end(b));
  int min = 1;
  for (int i = 2; i < b.size(); i++)
    if (b[i] - b[i-1] <= b[min] - b[min-1])
      min = i;
  soba1 = b[min-1]; soba2 = b[min];
}
```

3.5.3 *Kanonski oblik niza (provera jednakosti multiskupova)*

Da bi se proverilo da li su multiskupovi elemenata koji se nalaze u dva niza jednaka tj. da li je jedan niz permutacija drugog, možemo upotrebiti sortiranje.

Zadatak: Provera permutacija

Napiši program koji učitava dva niza brojeva i proverava da li je drugi niz permutacija prvog tj. da li se mogao dobiti od prvog samo promenom redosleda njegovih elemenata.

Opis ulaza

Sa standardnog ulaza se unose dva niza prirodnih brojeva. Za svaki niz se unosi broj elemenata (najviše 50000), a zatim i elementi razdvojeni po jednim razmakom.

Opis izlaza

Na standardni izlaz ispiši reč `da` ako je drugi niz dobijen mešanjem prvog, tj. `ne` ako nije.

Primer

Ulaz	*Izlaz*
5 1 3 2 4 3 5 4 3 2 3 1	da

Rešenje

Sortiranje

Jedan od načina da proverimo da li je jedan niz permutacija drugog je da oba niza dovedemo u neku "kanonsku" formu, a onda da proverimo da li su dobijene kanonske forme jednake. Najjednostavnija kanonska forma se dobija kada se nizovi sortiraju po veličini (na primer, neopadajuće).

Poređenje jednakosti dva vektora se može ostvariti (bibliotečkim) operatorom == (u vremenskoj složenosti $O(n)$). Ako bi elementi bili smešteni u nizove, onda bi jednakost bilo potrebno proveriti ili ručno implementiranim poređenjem jednog po jednog elementa (linearnom pretragom bi se ispitivalo da li postoji element koji im se razlikuje) ili bibliotečkom funkcijom `equal` koja prima iterator na početak i iza kraja prvog niza i na početak drugog niza.

Ako želimo da održimo redosled elemenata u vektorima, pre provere permutacija bismo morali da napravimo kopije, koje ćemo zatim sortirati (ovim se povećava dodatna memorijska složenost).

Nizovi od n elemenata se bibliotečkom funkcijom porede u vremenu $O(n \log n)$, dok se njihova jednakost proverava u vremenu $O(n)$. Proverom, dakle, dominira vreme sortiranja i algoritam je složenosti $O(n \log n)$.

```
bool jePermutacija(vector<int>& a, vector<int>& b) {
  if (a.size() != b.size())
    return false;
  sort(a.begin(), a.end());
  sort(b.begin(), b.end());
  return a == b;
}
```

3.5.4 *Sortiranje intervala*

U mnogim realnim primenama programiranja se razmatraju intervali. To mogu biti prostorni intervali, ali i vremenski intervali. Na primer, prilikom raspoređivanja časova ili nekih drugih aktivnosti, svaki čas koji se raspoređuje se predstavlja intervalom određenim vremenom početka i vremenom kraja. Intervali mogu biti jednodimenzionalni, dvodimenzionalni (tada su u pitanju pravougaonici), pa i višedimenzionalni. Efikasni algoritmi za obrade intervala se obično dobijaju tako što se intervali obilaze u nekom sortiranom redosledu: to je u jednodimenzionom slučaju obično ili redosled levih krajeva ili redosled desnih krajava, a ponekad se istovremeno razmatraju i sortiraju sve tačke (i levi i desni krajevi). Prikažimo ovo kroz nekoliko zadataka.

Zadatak: Najbrojniji presek intervala

Poznat je raspored časova i za svaki čas je poznato vreme početka i završetka. Pretpostavićemo da su časovi intervali oblika $[a, b)$, tj. da čas traje u trenutku svog početka a, a da ne traje više u trenutku svog završetka b. Koliko je učionica potrebno da bi svi časovi mogli da se održe?

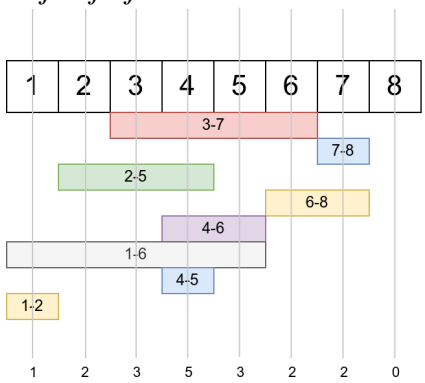
Opis ulaza

Sa standardnog ulaza se učitava broj časova n ($1 \leq n \leq 50000$), a zatim u narednih n redova vreme početka i vreme završetka svakog časa (merenje je veoma precizno, pa se vreme predstavlja prirodnim brojevima manjim od milijarde), odvojene sa po jednim razmakom.

Opis izlaza

Na standardni izlaz ispisati maksimalni broj časova koji se održavaju u istom trenutku.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
8 3 7 7 8 2 5 6 8 4 6 1 6 4 5 1 2	5	

U trenutku 4 raspoređeno je 5 časova.

Rešenje

Brojač za svaki pojedinačan trenutak

Direktno rešenje bi podrazumevalo da se održava niz u kome se za svaki trenutak pamti broj časova koji se u tom trenutku održavaju. Pošto ne znamo koliko trenutaka postoji umesto niza možemo upotrebiti mapu (u jeziku C++ možemo upotrebiti `unordered_map`).

Ako je trenutaka i časova puno, ova metoda će biti veoma neefikasna. Složenost najgoreg slučaja je $O(n \cdot m)$, gde je n broj trenutaka, a m broj časova.

Sortirani niz karakterističnih trenutaka (početaka i krajeva časova)

Broj potrebnih učionica se menja samo u trenucima kada neki čas počne ili se završi. Da bi se odredio najveći broj učionica dovoljno je razmotriti samo te karakteristične trenutke. Veoma prirodno je da te karakteristične trenutke obrađujemo hronološki, u rastućem redosledu vremena. Možemo kreirati niz koji sadrži sve karakteristične trenutke (vremena početaka i krajeva časova) i za svaki trenutak beležiti da li je početak ili kraj. Taj niz možemo sortirati i zatim obrađivati redom, izračunavajući za svaki trenutak broj časova koji u tom trenutku traju inkrementalno, na osnovu broja časova u prethodnom karakterističnom trenutku. Ako u nekom vremenskom trenutku više časova počinje ili se završava, broj učionica treba upoređivati sa maksimumom tek kada obradimo sve časove koji su počeli ili su se završili u tom trenutku. To se može rešiti tako što u unutrašnjoj petlji obrađujemo sve časove koji su počeli

ili su se završili u tekućem trenutku.

Međutim, unutrašnja petlja nije potrebna. Pošto se sortiranje parova radi leksikografski (prvo po vremenu, a onda po oznaci 1 tj. -1), tako da su svi događaji koji su se desili u istom trenutku sortirani tako da prvo idu završeci časova (oznaka -1), pa onda počeci (oznaka 1), ne moramo da imamo unutrašnju petlju, jer će se broj prvo smanjivati, pa onda rasti i neće biti moguće da se dobije pogrešan rezultat zato što su dodati neki časovi pre nego što je konstatovano da su se neki časovi završili.

Ako postoji n časova, postoji $2n$ karakterističnih trenutaka za čije je sortiranje potrebno $O(n \log n)$ koraka. Nakon sortiranja, niz trenutaka se obrađuje jednim prolaskom kroz niz u linearnom vremenu. Dakle, složenošću dominira sortiranje i složenost ovog rešenja je $O(n \log n)$.

```
int n;
cin >> n;

// niz karakterističnih trenutaka
vector<pair<int, int>> promene;
promene.reserve(2*n);
for (int i = 0; i < n; i++) {
  int dosao, otisao;
  cin >> dosao >> otisao;
  promene.emplace_back(dosao, 1);
  promene.emplace_back(otisao, -1);
}

sort(begin(promene), end(promene));

int trenutnoUcionica = 0;
int maksUcionica = 0;
for (int i = 0; i < 2*n; i++) {
  trenutnoUcionica += promene[i].second;
  if (trenutnoUcionica > maksUcionica)
    maksUcionica = trenutnoUcionica;
}

cout << maksPrisutno << endl;
```

3.6 Binarna pretraga

Binarno pretraživanje ili *binarna pretraga* je algoritam pretrage uređene (sortirane) serije elemenata. Najčešće se (ali ne uvek) proverava da li niz sadrži datu vrednosti). Nakon poređenja tražene vrednosti sa središnjim elementom niza, zahvaljujući sortiranosti niza, može se izvršiti odbacivanje (tj. odsecanje) jedne polovine niza i pretraga se nastavlja u drugoj polovini. Ovo polovljenje dužine niza u svakom koraku dovodi do veoma efikasnog postupka (pokazaćemo da je broj koraka $O(\log n)$, gde je n broj elemenata serije).

U osnovnoj varijanti, binarna pretraga služi da se proveri da li sortirani niz elemenata sadrži neku datu vrednost. Pored ovoga, binarna pretraga se može upotrebiti da se pronađe prvi element u sortiranom nizu koji je (bilo strogo, bilo nestrogo) veći ili manji od date vrednosti. U svom najopštijem obliku binarna pretraga se koristi da se u nizu pronađe tzv. prelomna tačka: ako se zna da su elementi uređeni tako da prvo idu oni koji ne zadovoljavaju neko svojstvo P, a zatim oni koji zadovoljavaju svojstvo P, moguće je efikasno pronaći prvi element koji to svojstvo zadovoljava ili poslednji element koji ne zadovoljava to svojstvo.

Standardna biblioteka jezika C++ nudi nekoliko funkcija koje sprovode algoritam binarne pretrage.

- Funkcija `binary_search` proverava da li dati segment elemenata (zadat pomoću dva iteratora) sortiranog niza (tj. vektora)[1] sadrži zadatu vrednost (funkcija vraća `true` ako i samo ako se tražena vrednost nalazi unutar zadatog segmenta). Provera da li se data vrednost `x` nalazi unutar sortiranog niza (ili vektora) `a` može se izvršiti pomoću `binary_search(begin(a), end(a), x)`.
- Ako je potrebno u sortiranom nizu (tj. vektoru) pronaći sve elemente jednake datom elementu, možemo koristiti funkciju `equal_range` (sa istim parametrima kao `binary_search`). Ona vraća par iteratora koji ograničavaju segment elemenata jednakih datom (prvi iterator ukazuje na prvi element jednak traženoj vrednosti, a drugi na poziciju neposredno iza poslednjeg elementa jednakog traženoj vrednosti).

[1] I bibliotečke funkcije i ručna implementacija rade na potpuno isti način bilo da se koristi vektor, bilo da se koristi statički niz ili neka treća sekvencijalna kolekcija koja daje mogućnost efikasnog indeksnog pristupa. Bez gubitka na opštosti, u nastavku ćemo uglavnom koristiti vektore.

- Funkcija `lower_bound` vraća prvi od ta dva iteratora (tj. prvi element sortiranog niza (tj. vektora) tj. nekog njegovog segmenta koji je veći ili jednak od tražene vrednosti).
- Funkcija `upper_bound` vraća drugi od njih (tj. prvi element sortiranog niza (tj. vektora) tj. nekog njegovog segmenta koji je strogo veći od tražene vrednosti).

Ako se ne zada drugačije, ove funkcije podrazumevaju se da je niz sortiran u odnosu na podrazumevani poredak elemenata (neopadajući numerički ako su brojevi u pitanju, tj. neopadajući abecedni leksikografski ako su niske u pitanju). Poredak se može zadati ili promeniti na sličan način kao kod funkcija za sortiranje.

I ručna implementacija algoritma binarne pretrage je prilično jednostavna, a potrebno ju je koristiti kada se pretražuje serija brojeva koji nisu smešteni u nizu (na primer, kada se binarnom pretragom traži optimalna vrednost neke funkcije, o čemu će biti reči u poglavljima 3.6.3 i 3.6.4) ili kada je potrebno algoritam prilagoditi nekom problemu. Stoga ćemo u nastavku prikazati i ručne implementacije ovog algoritma.

3.6.1 Traženje vrednosti u nizu

Opišimo prvo osnovnu varijantu algoritma u kojoj se u sortiranom nizu (tj. vektoru) traži pozicija na kojoj se pojavljuje neka zadata vrednost.

Provera da li neopadajuće sortiran niz sadrži datu vrednost se može lako izvršiti bibliotečkom funkcijom `binary_search`.

```
vector<int> a = {1, 8, 13, 15, 15, 23, 38, 38, 38, 42};
int x = 15;
if (binary_search(begin(a), end(a), x))
   cout << "Sadrzi" << endl;
else
   cout << "Ne sadrzi" << endl;
```

Opišimo sada i kako binarna pretraga može da se implementira. Ako sortirani niz u kojem tražimo zadati element ima konačni broj elemenata, binarno pretraživanje se vrši na sledeći način: pronalazi se središnji element (element na središnjoj poziciji) dela niza koji se pretražuje, proverava se da li je on jednak zadatoj vrednosti i ako

jeste – vraća se njegov indeks, a ako nije – pretraživanje se nastavlja nad delom niza levo od središnjeg elementa, u kojem su svi manji elementi (ako je središnji element veći od zadate vrednosti) ili u delu niza desno od središnjeg elementa, u kojem su svi veći elemenati (ako je središnji element manji od zadate vrednosti). Dakle, u svakom koraku, sve dok se ne pronađe tražena vrednost, niz se deli se na dva dela i pretraga se nastavlja samo u jednom njegovom delu — odbacuje se deo koji sigurno ne sadrži traženu vrednost. Binarno pretraživanje je, stoga, primer pristupa *podeli i vladaj* (engl. divide and conquer), koji će detaljnije biti opisan u poglavlju **??**. Pošto se jedna polovina elemenata eliminiše, ponekad se ovaj pristup naziva i *smanji i vladaj* (engl. decrease and conquer).

Primer 3.6.1. *Binarno pretraživanje može se koristiti u igri pogađanja zamišljenog prirodnog broja iz zadatog intervala. Jedan igrač treba da zamisli jedan broj iz tog intervala, a drugi igrač treba da pogodi taj broj, na osnovu što manjeg broja pitanja na koje prvi igrač odgovara samo sa* da *ili* ne*. Ako pretpostavimo da interval čine brojevi od* 1 *do* 16 *i ako je prvi igrač zamislio broj* 11*, onda igra može da se odvija na sledeći način:*

- Da li je zamišljeni broj veći od 8? *da**
- Da li je zamišljeni broj veći od 12? ne
- Da li je zamišljeni broj veći od 10? da
- Da li je zamišljeni broj veći od 11? ne

Na osnovu dobijenih odgovora, drugi igrač može da zaključi da je zamišljeni broj 11.

Generalno, broj potrebnih pitanja tj. koraka binarne pretrage je reda $O(\log n)$, gde je n širina polaznog intervala koji se pretražuje. Naime, posle prvog pitanja širina sa n opada na $\frac{n}{2}$, posle sledećeg na $\frac{n}{4}$ itd. Posle k pitanja širina intervala opada na $\frac{n}{2^k}$. Pošto se pretraga vrši sve dok interval ne postane jednočlan ili prazan, važi da je $\frac{n}{2^k} \leq 1$, tj. da se odgovor dobija kada k dostigne vrednost približno jednaku $\log_2 n$.

Binarno pretraživanje je moguće primeniti i kada nije unapred poznata dužina sortiranog niza koji se pretražuje. Tada se u prvoj fazi određuje gornja granica dela niza u kom bi traženi element mogao da se nađe (pronalazi se prva vrednost veća ili jednaka od tražene), da bi se u drugoj fazi primenila klasična binarna pretraga.

Primer 3.6.2. *Ukoliko u prethodnoj igri nije zadata gornja granica intervala, najpre treba odrediti jedan broj koji je veći od zamišljenog broja i onda primeniti binarno pretraživanje. Ako pretpostavimo da je prvi igrač zamislio broj* 11*, onda igra može da se odvija na sledeći način:*

- Da li je zamišljeni broj veći od 1? da
- Da li je zamišljeni broj veći od 2? da
- Da li je zamišljeni broj veći od 4? da
- Da li je zamišljeni broj veći od 8? da
- Da li je zamišljeni broj veći od 16? ne

Na osnovu dobijenih odgovora, drugi igrač može da zaključi da je zamišljeni broj u intervalu od 9 *do* 16 *i da primeni binarno pretraživanje na taj interval.*

Broj pitanja potrebnih za određivanje intervala pretrage je $O(\log n)$*, gde je* n *zamišljeni broj a ukupna složenost pogađanja ponovo je* $O(\log n)$.

Binarno pretraživanje daleko je efikasnije nego linearno, ali zahteva da su podaci koji se pretražuju uređeni. To je i jedan od glavnih razloga da se u rečnicima, enciklopedijama, štamapnim telefonskim imenicima kakvi su se koristili u doba fiksne telefonije i slično odrednice sortiraju. Ovakve knjige obično se pretražuju postupkom koji odgovara varijantama binarne pretrage[2]. Odnos složenosti postaje još očigledniji ukoliko se zamisli koliko bi komplikovano bilo sekvencijalno pretraživanje reči u nesortiranom rečniku.

Binarno pretraživanje može se implementirati iterativno ili rekurzivno.

Naredna implementacija binarnog pretraživanja poziva pomoćnu, rekurzivnu funkciju `binarna_pretraga` koja rešava nešto opštiji zadatak – vraća indeks elementa niza `a` između indeksa `l` i indeksa `d` (uključujući i njih) koji je jednak zadatoj vrednosti `x` ako takav postoji, a vraća `-1` inače.

```
int binarna_pretraga(const vector<int>& a, int l, int d, int x)
{
  int s;
  if (l > d)
    return -1;
  s = l + (d - l)/2;
  if (x < a[s])
    return binarna_pretraga(a, l, s-1, x);
```

[2]Postupak se naziva *interpolaciona pretraga* i podrazumeva da se knjiga ne otvara uvek na sredini, već se tačka otvaranja određuje otprilike na osnovu položaja slova u abecedi (na primer, ako se traži reč na slovo B, knjiga se otvara mnogo bliže početku, a ako se traži reč na slovo U, knjiga se otvara mnogo bliže kraju).

```
  else if (x > a[s])
    return binarna_pretraga(a, s+1, d, x);
  else /* if (x == a[s]) */
    return s;
}

int binarna_pretraga(const vector<int>& a, int x) {
  int n = a.size();
  return binarna_pretraga(a, 0, n-1, x);
}
```

Primetimo da se, umesto izraza `l + (d - l)/2`, za određivanje središnjeg indeksa može koristiti i kraći izraz `(l + d) / 2`. Ipak, upotreba prvog izraza je preporučena kako bi se smanjila mogućnost nastanka prekoračenja. Ovo je jedan od mnogih primera gde izrazi koji su matematički jednaki, imaju različita svojstva u aritmetici fiksne širine.

Složenost navedene funkcije je $O(\log n)$, a njena korektnost dokazuje se jednostavno (indukcijom), pri čemu se pretpostavlja da je niz `a` sortiran.

Lema 3.6.1 (Korektnost rekurzivne binarne pretrage)**.** *Razmotrimo poziv* `binarna_pretraga(a, l, d, x)`*, pri čemu važi* $0 \leq l \leq n$ *i važi* $-1 \leq d < n$*. Ovaj poziv vraća ili poziciju* s *takvu da je* $l \leq s \leq d$ *i* $a_s = x$ *ili vrednost* -1*, ako su svi elementi niza* a *na pozicijama iz intervala* $[l, d]$ *različiti od* x*.*

Dokaz. Tvrđenje dokazujemo indukcijom.

- Bazu indukcije čini slučaj $l > d$. Tada funkcija vraća -1, dok je interval $[l, d]$ prazan, pa je tvrđenje važi.
- Razmotrimo slučaj $l \leq d$. Tada je $0 \leq l \leq d < n$. Za $s = l + (d - l)/2$ važi da je $l \leq s \leq d$, pa je i ono u granicama niza.
  - Ako je $a_s = x$, funkcija vraća vrednost s i tvrđenje važi.
  - Ako je $x < a_s$, tada funkcija vraća rezultat poziva `binarna_pretraga(a, l, s-1, x)`. Uslovi za primenu induktivne hipoteze su ispunjeni (nova desna granica $s - 1$ se nalazi u intervalu $[-1, n)$). Na osnovu induktivne hipoteze važi da ako ako taj poziv vrati neku poziciju s' tada je

$l \leq s' \leq s-1$ i $a_{s'} = x$, međutim, važi da je $s-1 \leq d$, pa je $l \leq s' \leq d$ i tvrđenje važi. Ako rekurzivni poziv vrati -1, tada nijedan element niza a na pozicijama iz intervala $[l, s-1]$ nije jednak x. Međutim, važi i $a_s > x$ i, pošto je niz sortiran, svi elementi iz intervala $[s, d]$ su takođe strogo veći od x. Zato i poziv `binarna_pretraga(a, l, d, x)` ispravno vraća vrednost -1 (jer, zaista, ni jedan element niza na pozicijama iz intervala $[l, d]$ nije jednak x).

– Slučaj $x > a_s$ se rešava analogno prethodnom. □

Rekurzivna funkcija se zaustavlja jer vrednost $d - l$ opada u svakom rekurzivnom pozivu, a važi $d - l \geq -1$, pa se to opadanje mora u nekom trenutku završiti.

U glavnom pozivu funkcije su ispunjeni uslovi prethodnog tvrđenja (indeksi 0 i $n-1$ su u zahtevanim granicama), pa je i glavna funkcija korektna.

Umesto rekurzivne, implementacija može biti i iterativna.[3]

```
int binarna_pretraga(const vector<int>& a, int x) {
  int l, d, s;
  l = 0; d = a.size() - 1;
  while (l <= d) {
    s = l + (d - l)/2;
    if (x < a[s])
      d = s - 1;
    else if (x > a[s])
      l = s + 1;
    else if (x == a[s])
      return s;
  }
  return -1;
}
```

Njen dokaz korektnosti (ostavljen čitaocu za vežbu) se zasniva na invarijanti da su elementi levo od pozicije l strogo manji od x, a desno od pozicije d strogo veći od x.

[3] Oba rekurzivna poziva u navedenoj implementaciji su repno-rekurzivna, tako da se mogu jednostavno eliminisati (postpuci eliminisanja rekurzije su sistematično opisani u poglavlju **??**).

3.6.2 *Traženje prelomne tačke*

Osnovna varijanta binarne pretrage, čija je implementacija prikazana, pronalazi datu vrednost u sortiranom nizu. Kao što je već nagoveštено, binarna pretraga može se upotrebiti i za rešavanje nešto opštijih problema. Pretpostavimo da je niz uređen tako da svi njegovi početni elementi zadovoljavaju neki uslov P, a da posle njih idu elementi koji ne zadovoljavaju taj uslov. Takav niz možemo neformalno predstaviti nizom pluseva (koji označavaju elemente koji zadovoljavaju svojstvo P), a zatim minusa (koji označavaju elemente koji ne zadovoljavaju svojstvo P). Na primer, `+++++----`. Potrebno pronaći mesto u nizu gde se ta promena dešava (tj. potrebno je pronaći poslednji element koji zadovoljava uslov P tj. poslednji `+` ili prvi element koji ga ne zadovoljava tj. prvi `-`). Na primer, može biti poznato da se u nizu nalaze prvo parni, a zatim neparni elementi i potrebno je pronaći koliko postoji svakih. Slično, može se razmatrati sortiran niz i za uslov P uzeti uslov da je element niza strogo manji od neke date vrednosti x (u sortiranom nizu, prvo su svi elementi koji su strogo manji od x, a iza njih su elementi koji su veći od ili jednaki x).

Problem traženja prelomne tačke opštiji je od problema traženja zadate vrednosti u sortiranom nizu, jer se u ovom drugom problemu podrazumeva postojanje relacije poretka na osnovu koje su vrednosti u nizu sortirane, dok se u prvom problemu podrazumevamo samo tzv. svojstvo *monotonosti*, koje podrazumeva da se u nizu prvo javljaju elementi koji zadovoljavaju neko svojstvo P, pa onda oni koji ga ne zadovoljavaju. Da bi se dokazalo da niz zadovoljava svojstvo monotonosti, dovoljno je dokazati bilo koji od sledeća dva (ekvivalentna) uslova:

- Za svake dve pozicije $0 \leq i < j < 0$, ako element na poziciji j ima svojstvo P, onda i element na poziciji i ima svojstvo P.
- Za svake dve pozicije $0 \leq i < j < 0$, ako element na poziciji i nema svojstvo P, onda i element na poziciji j nema svojstvo P.

Kao ilustraciju varijante binarne pretrage u kojoj se pronalazi takva prelomna tačka, u nastavku je prikazana funkcija koja pronalazi poziciju prvog elementa u sortiranom nizu koji je veći od ili jednak datoj vrednosti x (ili vraća dužinu niza ako takav element ne postoji). Ovo je tačno ono što radi bibliotečke funkcija `lower_bound`. Na sličan način može se odrediti i pozicija prvog elementa koji je strogo veći od zadatog broja, poslednjeg elementa koji je manji od ili jednak datom broju ili poslednjeg elementa koji strogo manji od datog broja.

```
vector<int> a = {1, 8, 13, 15, 15, 23, 38, 38, 38, 42};
int x = 16;
// prvi element veci ili jednak od 16 je 23 (na poziciji 6)
auto it = lower_bound(begin(a), end(a), x);
cout << distance(begin(a), it) << endl;
```

Binarna pretraga prelomne tačke može se jednostavno iskoristi i za proveru da li niz sadrži dati broj. Kada se nađe pozicija prvog elementa koji je veći od ili jednak od traženog, jednostavno se može proveriti da li se na toj poziciji nalazi upravo taj element (ako postoji u nizu, on mora biti na toj poziciji). Ako vrednost postoji u nizu, to će biti tražena pozicija, a ako ne postoji, onda će iterator koji funkcija `lower_bound` vraća biti ili van granica niza (ako su svi elementi niza manji od tražene vrednosti) ili će ukazivati na element koji je strogo veći od tražene vrednosti.

```
vector<int> a = {1, 8, 13, 15, 15, 23, 38, 38, 38, 42};
int x = 15;
auto it = lower_bound(begin(a), end(a), x);
if (it < end(a) && *it == x)
   cout << "Na poziciji: " << distance(begin(a), it) << endl;
else
   cout << "Ne sadrzi" << endl;
```

Korišćenjem funkcije `lower_bound` može se, na primer, odrediti i broj elemenata koji su veći od ili jednaki datom elementu u nekom sortiranom nizu (u opštem slučaju može se pronaći broj elemenata koji zadovoljavaju i broj elemenata koji ne zadovoljavaju uslov P).

```
int broj_vecih_ili_jednakih(const vector<int>& a, int x)  {
  auto it = lower_bound(begin(a), end(a), x);
  return  distance(it, end(a));
}
```

Broj jednakih elemenata jednakih datoj vrednosti `x` se može pronaći funkcijom `equal_range` ili kombinacijom funkcija `lower_bound` i `upper_bound` tako što se nađe pozicija prvog elementa većeg ili jednakog od `x` i pozicija prvog elementa

strogo većeg od x. Razlika te dve pozicije daje traženi broj pojavljivanja vrednosti x.

```
int broj_pojavljivanja(const vector<int>& a, int x)
{
   auto lb = lower_bound(begin(a), end(a), x);
   auto ub = upper_bound(begin(a), end(a), x);
   return distance(lb, ub);
}
```

Pređimo sada na ručnu implementaciju pretrage prelomne tačke. Umesto da prikažemo implementaciju ove funkcije, a zatim da dokažemo njenu korektnost, pokušajmo da funkciju izvedemo iz specifikacije tj. nametnute invarijante. Uvedimo promenljive l i d i osigurajmo, kao invarijantu, da sve vreme tokom pretrage važe sledeći uslovi:

- Elementi niza a na pozicijama iz intervala $[0, l)$ manji od vrednosti x (zadovoljavaju uslov P),
- da su elementi na pozicijama iz intervala (d, n) veći od ili jednaki x (ne zadovoljavaju uslov P).

Elementima na pozicijama iz intervala $[l, d]$ status još nije poznat (ovo formalno nije deo invarijante, mada važi tokom pretrage).

Raspored je, dakle, sledeći:

0					l				d					n
$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	?	?	?	?	?	$>$	$>$	$>$	$>$	$>$

- Ovi uslovi će biti na početku ispunjeni, ako se promenljiva l inicijalizuje na nulu (tada je interval $[0, l) = [0, 0)$ prazan), a promenljiva d na vrednost $n - 1$ (tada je i interval $(d, n) = (n - 1, n)$ prazan).

- Neka s predstavlja sredinu intervala $[l, d]$.

- Ako je element niza a na poziciji s manji od vrednosti x (zadovoljava uslov P) takvi su i svi elementi iz intervala $[l, s]$. Zato se vrednost l može postaviti na $s + 1$ i invarijanta će ostati da važi (zaista, ako je $l' = s + 1$, svi elementi sa pozicija iz intervala $[0, l')$ će biti ili sa pozicija iz intervala $[0, l)$ za koje se zna da su manji od x (zadovoljavaju uslov P) ili su iz sa pozicija iz intervala $[l, s]$, za koje je upravo utvrđeno da su manji od x tj. da zadovoljavaju uslov P).
- Ako je element niza na poziciji s veći od ili jednak x (ne zadovoljava uslov P), tada su i svi elementi na pozicijama iz intervala $[s, d]$ veći od ili jednaki x (i ne zadovoljavaju uslov P). Zato se vrednost d može postaviti na $s-1$ i invarijanta će ostati da važi (zaista, ako je $d' = s-1$, tada su svi elementi iz intervala pozicija (d', n) ili iz intervala $[s, d]$ za koje je upravo utvrđeno da će biti veći od ili jednaki x ili iz intervala (d, n) za koje se od ranije zna da su veći od ili jednaki x).

Pretraga se vrši sve dok postoje elementi nepoznatog statusa, tj. sve dok je interval $[l, d]$ neprazan, odnosno dok je $l \leq d$. U trenutku kada važi $l > d$, na osnovu invarijante sledi:

- Svi elementi levo od l zadovoljavaju uslov P, pa se poslednji takav element mora nalaziti na poziciji $l - 1 = d$.
- Svi elementi desno od d ne zadovoljavaju uslov P, pa se prvi takav element nalazi na poziciji $d + 1 = l$. Dakle, prvi element veći ili jednak od datog broja x se nalazi na poziciji l.

```
int prvi_veci_ili_jednak(const vector<int>& a, int n, int x)  {
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (a[s] < x) {
      l = s + 1;
    } else
      d = s - 1;
  }
```

```
  return l;
}
```

Funkciju smo mogli zasnovati i na invarijanti da se u intervalu $[0, l)$ nalaze elementi koji su manji od x (zadovoljavaju svojstvo P), da se u intervalu $[d, n)$ nalaze elementi koji su veći od ili jednaki x (ne zadovoljavaju svojstvo P), a da se u intervalu $[l, d)$ nalaze elementi čiji status još nije poznat. Tada bi vrednost l bila inicijalizovana na 0, a d na n. Pretraga se bi se vršila dok je $l < d$. Nakon pronalaženja sredine s intervala $[l, d]$, ako je $a[s] < x$, tada se l može postaviti na $s+1$ (jer su svi elementi na pozicijama levo od s zaključno sa njom manji od x), a u suprotnom se d može postaviti na s (jer su svi elementi od pozicije s, pa naviše veći od ili jednaki x).

Kada je kôd korektan, dokaz korektnosti je često neinformativan. Zahvaljujući njemu znamo da je kôd ispravan, ali ne mnogo više od toga. Mnogo interesantnija situacija se dešava u slučaju kada nam formalno rezonovanje o kodu pomaže da detektujemo i ispravimo greške u programu (tzv. bagove). Pogledajmo naredni pokušaj implementacije algoritma.

```
int prvi_veci_ili_jednak(const vector<int>& a, int n, int x) {
  int l = 0, d = n;
  while (l < d) {
    int s = l + (d - l) / 2;
    if (a[s] < x)
      l = s+1;
    else
      d = s-1;
  }
  return d+1;
}
```

Na osnovu incijalizacije deluje da pokušavamo da pretražimo poluzatvoreni interval $[l, d)$. Pošto je u pitanju binarna pretraga, izgleda da se nameće invarijanta da je $0 \leq l \leq d \leq n$ i da važe sledeći uslovi.

- Svi elementi na pozicijama iz $[0, l)$ su manji od x tj. zadovoljavaju uslov P,
- Svi elementi na pozicijama iz intervala $[d, n)$ su veći ili jednaki x tj. ne zadovoljavaju uslov P.

Na početku su oba ta intervala prazna, pa invarijanta za sada dobro funkcioniše. Ako pogledamo uslov petlje, deluje da petlja radi dok se interval nepoznatih elemenata $[l, d)$ ne isprazni (zaista, kada je $l \geq d$, taj interval je prazan). Za sada sve radi kako treba. Pokušamo sada da proverimo da li izvršavanje tela petlje održava invarijantu.

- Ako je $a_s < x$, tada se promenljiva `l` postavlja na vrednost $l' = s + 1$. Na osnovu invarijante treba da važi da su svi elementi na pozicijama u intervalu $[0, l')$ manji od x, međutim, to će ovde biti ispunjeno, jer je $a_s < x$, biće manji i svi elementi ispred njega. Dakle, u ovom slučaju je kôd korektan i invarijanta ostaje održana.
- Ako je $a_s \geq x$, tada se promenljiva `d` postavlja na vrednost $d' = s - 1$. Na osnovu invarijante treba da važi da su svi elementi u intervalu $[d', n)$ veći ili jednaki od x. Međutim, mi to ne znamo, jer samo znamo da je $a_s \geq x$, ali ne znamo da li je $a_{s-1} \geq x$. Dakle, ovde se sigurno krije greška u kodu. Ako dodelu `d = s-1` zamenimo sa `d = s`, tada će invarijanta biti održana (jer znamo da je $a_s \geq x$ i takvi su i svi elementi na pozicijama desno od s).

Na kraju, kada se petlja završi možemo zaključiti da važi da je $l = d$ (jer sve vreme važi da je $l \leq d$, a nakon petlje ne važi da je $l < d$). U kodu se za poziciju prvog elementa koji je veći ili jednak x proglašava pozicija $d + 1$. Iako je u originalnoj varijanti koda `l` moglo bez problema da se zameni sa `d+1`, u ovoj varijanti to nije moguće. Naime, mi na osnovu invarijante ovog koda znamo da se na poziciji $l = d$ nalazi element koji je veći ili jednak x, a da se na poziciji $l - 1$ nalazi element koji je manji od x (osim kada je $l = 0$ i tada nema elemenata manjih od x). Zato krajnji rezultat nije korektan i potrebno ga je zameniti sa `d`, jer se prvi element koji je veći ili jednak x nalazi na poziciji d (osim kada su svi elementi veći ili jednaki x, kada je $d = n$, no i tada je d ispravna povratna vrednost). Dakle, formalnom analizom smo otkrili i ispravili dve greške.

Neki programeri program ispravljaju tako što nasumice pokušavaju da pomere indekse za 1 levo ili desno, da zamene relaciju '<' relacijom '<=' i slično. Već na ovako kratkim programima se vidi da je prostor mogućih takvih malih izmena ogroman, a da je mogućnost za grešku prilikom takvog eksperimentalnog pristupa veoma velika. Stoga je uvek bolje zastati, formalno analizirati šta je potrebno da kôd radi i ispraviti ga na osnovu rezultata formalne analize.

Na kraju, skrenimo pažnju na još jedan detalj ispravljenog programa. Parcijalna korektnost je jasna na osnovu analize koju smo sproveli, međutim, zaustavljanje može

biti dovedeno u pitanje, s obzirom na naredbu `d = s`. Zaustavljanje dokazujemo tako što pokazujemo da se u svakom koraju smanjuje broj nepoznatih elemenata, tj. da dužina intervala $[l, d)$ koja je jednaka $d - l$ u svakom koraku petlje opada. Pošto je $l \leq d$ invarijanta, smanjivanje ne može trajati zauvek, pa se u nekom trenutku program zaustavlja. Postavlja se pitanje da li se $d-l$ smanjuje i u izmenjenom kodu u kome se javlja naredba `d=s`. Odgovor je potrvrdan, a obrazloženje je suptilno. Prvo, na osnovu uslova petlje važi da je $l < d$. Dalje, vrednost s se izračunava naredbom `s = l + (d - l) / 2` što nam da je $s = \left\lfloor \frac{l+d}{2} \right\rfloor$. Zbog zaokruživanja naniže, važi da je $s < d$ i zato se nakon određivanja $d' = s$, $l' = l$ vrednost $d' - l'$ smanjuje u odnosu na $d-l$. Važi i da je $l \leq s$, ali pošto je u drugoj grani $l' = s+1$ i $d' = d$, vrednost $d' - l'$ se opet smanjuje u odnosu na $d - l$. Da je zaokruživanje kojim slučajem vršeno naviše (npr. `s = l + (d - l + 1) / 2`), program bi mogao upasti u beskonačnu petlju.

Prikažimo primene ovog oblika binarne pretrage kroz nekoliko zadataka.

Zadatak: Broj studenata iznad praga

Komisija za upis na fakultet treba da odredi prag za upis kandidata. Komisiju stalno pitaju koji bi broj studenata bio upisan kada bi prag prolaznosti bio zadati broj poena (upisuju se svi kandidati čiji je broj poena veći ili jednak pragu). Potrebno je napisati program koji daje odgovore na ta pitanja.

Opis ulaza

Sa standardnog ulaza učitava se broj kandidata n ($0 \leq n \leq 10^5$), a zatim i njihovi takmičara (celi brojevi), zadati u sortiranom redosledu od najvećeg do najmanjeg. Nakon toga se učitava broj m ($1 \leq m \leq 50000$) koji predstavlja broj pitanja na koja treba da se odgovori, a zatim i m brojeva za koje je potrebno dati odgovor koliko bi se studenata upisalo kada bi se taj broj poena uzeo za prag.

Opis izlaza

Na standardni izlaz ispisati tražene brojeve upisanih studenata, svaki u posebnom redu.

Primer

Ulaz	*Izlaz*
`5`	`0`
`89 73 73 56 23`	`4`
`4`	`3`
`95 50 70 0`	`5`

Rešenje

U zadatku je potrebno efikasno odrediti broj elemenata sortiranog niza koji su veći ili jednaki od datog broja. Ako nađemo poziciju prvog elementa koji je veći ili jednak od datog broja, tada broj takvih elemenata možemo odrediti tako što izračunamo razliku između ukupnog broja članova niza i te pozicije.

Linearna pretraga

Najjednostavniji način da nađemo poziciju prvog elementa koji je veći ili jednak od datog broja je da primenimo linearnu pretragu i da redom proveravamo jedan po jedan element sve dok ne dođemo ili do kraja niza ili do tražene pozicije.

Složenost ovakve pretrage je $O(n)$, pa je ukupna složenost rešenja $O(m \cdot n)$, što je previše imajući u vidu ograničenja data u zadatku.

Binarna pretraga

Pozicija se efikasno može pronaći primenom algoritma binarne pretrage. Najjednostavnije je upotrebiti bibliotečku implementaciju. U jeziku C++ možemo upotrebiti funkciju `lower_bound`.

Složenost jedne binarne pretrage je $O(\log n)$, pa je ukupna složenost algoritma $O(m \log n)$.

```
int m;
cin >> m;
for (int i = 0; i < m; i++) {
 int prag;
 cin >> prag;
 auto it = lower_bound(begin(poeni), end(poeni), prag);
 int broj = distance(it, end(poeni));
 cout << broj << endl;
}
```

Zadatak: Minimum rotiranog sortiranog niza

Sortirani niz celih brojeva u kome su svi elementi različiti je rotiran za k mesta ulevo i time je dobijen ciklični niz koji zadovoljava uslov da je $x_k < x_{k+1} < \ldots < x_{n-1} < x_0 < \ldots < x_{k-1}$. Jedan takav niz je, na primer, `11 13 15 19 24 1 3 8`

9. Napisati program koji pronalazi najmanji element takvog niza. Potrudi se da se nakon učitavanja elemenata minimum pronađe u vremenskoj složenosti $O(\log n)$.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 50000$), a zatim n elemenata niza (n celih brojeva razvojenih sa po jednim razmakom).

Opis izlaza

Na standardni izlaz ispisati najmanji element niza.

Primer

Ulaz	*Izlaz*
`9` `11 13 15 19 24 1 3 8 9`	`1`

Rešenje

Linearna pretraga

Zadatak možemo rešiti uobičajenim algoritmom ili bibliotečkom funkcijom za pronalaženja minimuma. U jeziku C++ možemo upotrebiti funkciju `min_element`.

Ovaj algoritam zahteva prolazak kroz sve elemente niza, pa je složenosti $O(n)$.

Binarna pretraga

****Poređenje sa prvim elementom niza****

Bolje rešenje se može dobiti binarnom pretragom. Nakon rotacije svi elementi u početnom delu niza su strogo veći od početnog, a onda u završnom delu niza idu svi elementi koji su strogo manji od početnog. Najmanji element koji tražimo je prvi element u nizu koji je strogo manji od početnog i njega možemo naći binarnom pretragom. Treba obratiti pažnju na specijalni slučaj u kome je niz rotiran za 0 mesta i tada ne postoji element koji je strogo manji od početnog. Binarna pretraga će tada vratiti poziciju iza kraja niza i u tom slučaju najmanji element u nizu je upravo prvi element niza.

Dakle, ponovo tražimo prvi element koji zadovoljava neki dati uslov. Održavamo pozicije l i d, i invarijanta petlje je da su:

- svi elementi ispred pozicije l (elementi na pozicijama iz intervala $[0, l)$) veći ili jednaki početnom i sortirani su,

- svi elementi iza pozicije d (elementi na pozicijama iz intervala (d, n)) strogo manji od početnog i sortirani su.

Pretraga se završava u trenutku kada je $l = d + 1$ i tada važi da su svi elementi iza pozicije d tj. svi elementi od pozicije $l = d + 1$ pa do kraja niza strogo manji od početnog elementa niza, dok su elementi od početka niza levo od pozicije l veći ili jednaki od početnog elementa niza. Ako postoje elementi od pozicije l do kraja niza, tj. ako je $l < n$, tada su oni sigurno manji od elemenata ispred pozicije l, a pošto su sortirani, najmanji je prvi od njih, tj. element na poziciji l. U suprotnom, ako je $l = n$, tada postoji samo levi deo niza, tj. svi elementi u nizu su veći ili jednaki početnom elementu, a pošto je taj deo niza sortiran, najmanji element je početni.

Složenost binarne pretrage je $O(\log n)$. Naglasimo da ako se razmatra ceo program, a ne samo funkcija pretrage, dominira učitavanje elemenata u niz, koje je složenosti $O(n)$, pa se prednosti binarne pretrage ne mogu efektivno izmeriti.

Poređenje se poslednjim elementom niza

Provera specijalnog slučaja nakon pretrage se može izbeći ako se umesto odnosa sa prvim, gleda odnos sa poslednjim elementom u nizu: tražimo prvi element koji je strogo manji od poslednjeg.

Invarijanta ovog algoritma je da su:

- svi elementi ispred pozicije l (elementi na pozicijama iz intervala $[0, l)$) strogo veći od poslednjeg elementa niza i sortirani su,
- svi elemeni iza pozicije d (elementi na pozicijama iz intervala (d, n)) manji ili jednaki od poslednjeg elementa niza i sortirani su.

Kada se petlja završi važi da je $l = d + 1$. Zato su svi elementi iza pozicije l strogo veći od elemenata na poziciji l. Pošto je deo od pozicije l do kraja sortiran, minimum se nalazi na poziciji l, jer je taj deo uvek neprazan. Zaista, mora da važi da je $l < n$, jer bi u suprotnom poslednji element bio levo od pozicije l, što je nemoguće, jer se levo od pozicije l nalaze elementi koji su strogo veći od poslednjeg.

```
int minRotiranogSortiranog(const vector<int>& a) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
```

```
    int s = l + (d-l)/2;
    if (a[s] < a[n-1])
      d = s-1;
    else
      l = s+1;
  }
  return a[l];
}
```

3.6.3 *Optimizacija binarnom pretragom (pretraga po rešenju)*

Binarna pretraga se može upotrebiti i u procesu optimizacije, ako se problem može formulisati kao problem pronalaženja prelomne tačke. Ovaj oblik pretrage se ponekad naziva *binarna pretraga po rešenju*, jer se prostor kome može pripadati vrednost rešenja problema binarno pretražuje. Ideja je da se problem optimizacije "naći najmanju vrednost koja zadovoljava određeni uslov", svede na problem odlučivanja "da li data vrednost zadovoljava određeni uslov". Binarnu pretragu je moguće primeniti ako problem zadovoljava svojstvo monotonosti, koje zahteva da ako neka vrednost zadovoljava uslov, onda uslov zadovoljavaju i sve vrednosti veće od nje, a ako ne zadovoljava, onda uslov ne zadovoljavaju ni vrednosti manje od nje. Naravno, sasvim sličan je zadatak pronalaženja najveće vrednosti koja ne zadovoljava uslov. Karakteristično za ovu upotrebu binarne pretrage je to što vrednosti o kojima je reč obično nisu indeksi elemenata niza, a često se vrši optimizacija i nad neprekidnim skupom vrednosti (do na određenu tačnost). Takođe, provera ispunjenja uslova za svaku konkretnu vrednost je obično spora i želimo da smanjimo broj provera ispunjenja uslova koliko je moguće. Stoga se u ovakvim situacijama umesto korišćenja bibliotečkih funkcija, binarna pretraga implementira ručno.

Ilustrujmo ovu tehniku kroz sledeći problem.

Zadatak: Drva

Drvoseča treba da naseče određenu količinu drveta i ima testeru koju može da podešava da seče na bilo kojoj celobrojnoj visini (u metrima). Pošto testera seče samo drvo iznad visine na koju je postavljena, što je testera više, naseći će se manje drveta. Pošto drvoseča brine o okolini, on ne želi da naseče više drveta nego što mu je potrebno. Napiši program koji određuje najvišu moguću celobrojnu visinu testere, tako da drvoseča dobije dovoljno drveta (pretpostavi da uvek postoji dovoljno drveta).

Opis ulaza

Sa standardnog ulaza se učitava broj drveća u šumi n ($1 \leq n \leq 10^5$), a zatim niz visina svakog drveta (niz prirodnih brojeva između 1 i 10000, razdvojenih sa po jednim razmakom). Nakon toga učitava se i količina nasečenog drveta (pošto su sva debla iste debljine, količina se meri u metrima visine isečenih stabala).

Opis izlaza

Na standardni izlaz ispisati traženu maksimalnu celobrojnu visinu testere.

Primer

Ulaz	*Izlaz*	*Objašnjenje*

`5`
`24 21 19 14 22`
`14`
	`18`	Postavljanjem testere na 18 metara, od prvog drveta ćemo odseći 6 metara, od drugog 3, od trećeg 1, od četvrtog ništa, a od petog 4 metra. To je ukupno 14 metara, što je tačno onoliko koliko mu je potrebno.

Rešenje

Optimizacija binarnom pretragom

Jedno rešenje problema se može zasnovati na binarnoj pretrazi po rešenju tj. po traženju optimalne vrednosti korišćenjem binarne pretrage. Postavljanjem testere na visinu h, kod svih drva koja su viša od h biće odsečeno $h_i - h$ metara, dok od ostalih drva neće biti isečeno ništa. Na osnovu toga, za fiksiranu visinu testere grubom silom (ispitivanjem svakog drveta zasebno) u vremenu $O(n)$ možemo izračunati ukupnu količinu nasečenog drveta. Binarna pretraga je primenljiva jer znamo da je do određenih visina testere drveta dovoljno, a da je od određene visine testere drveta premalo, tako da zapravo tražimo prelomnu tačku, tj. najveću visinu testere za koju je drveta dovoljno tj. poslednji element niza koji zadovoljava uslov. Primetimo da u ovom slučaju nemamo vrednosti smeštene u niz, već ih računamo po potrebi. Zato binarnu pretragu implementiramo ručno.

Ako je maksimalna visina drveta M, tada je složenost ovog pristupa $O(n \log M)$. Naime, binarnom pretragom se pretražuje interval $[0, M]$, pa se provera da li je nasečeno dovoljno drveta poziva $\log M$ puta. Izračunavanje količine nasečenog drveta i provera da li je ona dovoljna vrši se jednim prolazak kroz niz drveta i složenosti je $O(n)$.

```
int testera(const vector<int>& visine, int potrebno) {
  int od_visina = 0;
  int do_visina = *max_element(begin(visine), end(visine));
  while (od_visina <= do_visina) {
    int visina = od_visina + (do_visina - od_visina) / 2;
    long long naseceno = 0;
    for (int v : visine)
      if (v >= visina)
        naseceno += v - visina;

    if (naseceno >= potrebno)
      od_visina = visina + 1;
    else
      do_visina = visina - 1;
  }
  return do_visina;
}
```

3.6.4 *Određivanje nulu, minumum ili maksimum realne funkcije*

Binarnom pretragom možemo odrediti neke značajne tačke realne funkcije (nule, minimum, maksimum), do na određenu tačnost. Na primer, jednačinu $\cos(x) = x$, možemo rešiti sa tačnošću 10^{-5} narednim postupkom.

```
double l = 0, d = 1;
while (abs(d - l) >= 1e-5) {
  double s = (l + d) / 2;
  if (cos(s) > s)
      l = s;
  else
      d = s;
}
cout << (l + d) / 2 << endl;
```

3.7 Tehnika dva pokazivača, tehnika pokretnog prozora

Ugnežđene petlje obično podrazumevaju postojanje dve brojačke promenljive od kojih spoljašnja samo uvećava svoju vrednost tokom iteracije, dok se vrednost brojačke petlje u unutrašnjoj uvećava do neke gornje granice, zatim se ponovo vraća na neku donju granicu i ponovo uvećava i to se ponavlja više puta, sve dok spoljašnja brojačka promenljiva ne dostigne svoju maksimalnu vrednost. Ovo po pravilu dovodi do kvadratne složenosti (tj. složenosti višeg stepena u slučaju ugnežđavanja većeg broja petlji).

Tehnika dva pokazivača obuhvata široku klasu efikasnih algoritama koje takođe karakteriše postojanje dve ili više brojačkih promenljivih, koje se kreću kroz elemente nekog niza (često sortiranog). Međutim, ono što je karakteristično za njih je to što se, za razliku od unutrašnjih promenljivih u ugnežđenim petljama, ove promenljive stalno "kreću u istom smeru", tj. vrednost im se ili stalno povećava ili stalno smanjuje (a česta je i kombinacija gde se "niz obilazi sa dva kraja", gde se jedna promenljiva stalno povećava, a druga stalno smanjuje). Tehnička realizacija može biti bilo pomoću jedne petlje koja kontroliše vrednosti obe promenljive, bilo pomoću ugnežđenih petlji, ali tako da se nakon završetka tela unutrašnje petlje, spoljašnja promenljiva uvećava do mesta gde se unutrašnja petlja završila. Pošto se svaka promenljiva može promeniti najviše n puta (gde je n neko gornje ograničenje njihove vrednosti, obično dužina niza), broj promena (pa samim tim i izvršavanja tela petlje) je najviše $2n$ i linearan je po n tj. složenost mu je $O(n)$.

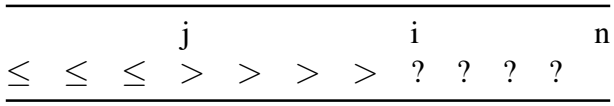
Algoritmi zasnovani na tehnici dva pokazivača obično mogu da se izvedu korišćenjem odsecanja primenjenih na ugnežđene petlje, pa je, kao i kod svake primene odsecanja, potrebno pažljivo obrazložiti njihovu korektnost.

Pokažimo sada nekoliko najkarakterističnijih primena ove tehnike (nakon čega ćemo još nekoliko primena ilustrovati i kroz zadatke).

3.7.1 Particionisanje niza

Ilustrujmo ovu tehniku na jednostavnom problemu preraspodeljivanja elemenata niza tako da se nakon preraspodele u nizu prvo nalaze elementi niza koji su manji ili jednaki datoj vrednosti x (u proizvoljnom redosledu), a zatim nalaze elementi niza koji su veći od date vrednosti x (u proizvoljnom redosledu). Videćemo u poglavlju **??** da je rešavanje ovog problema ključno za algoritam brzog sortiranja niza. Primetimo da za dati ulazni niz ne postoji jedinstveno rešenje (jer međusobni redosled elemenata u prvoj i u drugoj grupi može biti proizvoljan).

Jedno moguće rešenje može biti takvo da j pokazivač i ukazuje na naredni element niza koji treba obraditi, a da pokazivač j, takav da važi $j \leq i$ određuje granicu između elemenata manjih ili jednakih x i elemenata većih od x. Preciznije, pretpostavićemo invarijantu koja tvrdi da su elementi na pozicijama u intervalu $[0, j)$ manji ili jednaki x, elementi na pozicijama u intervalu $[j, i)$ veći od x, dok su elementi na pozicijama u intervalu $[i, n)$ nepoznatog statusa.

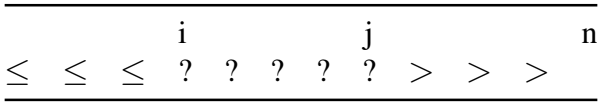


- Na početku inicijalizujemo $i = j = 0$ i invarijanta važi.
- Prilikom analize elementa na poziciji i mogu nastupiti dva slučaja.
  - Ako je $a_i \leq x$, dovoljno je razmeniti elemente na pozicijama j i i i uvećati vrednosti oba pokazivača i i j.
  - Ako je $a_i > x$, dovoljno je samo uvećati pokazivač i.

```
int j = 0;
for (int i = 0; i < a.size(); i++)
   if (a[i] <= x)
       swap(a[i], a[j++]);
```

Sasvim je jasno da je složenost ovog algoritma $O(n)$.

Još jedno rešenje datog problema se može zasnovati na invarijanti da su svi elementi u intervalu $[0, i)$ manji ili jednaki x, da su elementi u intervalu (j, n) veći od x, dok su elementi u intervalu $[i, j]$ još neobrađeni.




```
int n = a.size();
int i = 0, j = n-1;
while (i < j) {
   while (i < j && a[i] <= x)
```

```
        i++;
    while (i < j && a[j] > x)
        j--;
    swap(a[i], a[j]);
}
```

Na prvi pogled možda deluje da ova varijanta algoritma ima kvadratnu složenost, jer sadrži ugnježđene petlje, međutim, to nije tačno. Naime, pošto se oba pokazivača kreću stalno u istom smeru (promenljiva `i` se samo uvećava, a promenljiva `j` umanjuje), kao i uvek u slučaju tehnike dva pokazivača, ukupan broj operacija je ograničen odozgo sa $2n$ i složenost ovog algoritma je $O(n)$.

3.7.2 *Objedinjavanje sortiranih nizova*

Naivan način da se zadatak reši je da se elementi oba učitana niza prekopiraju u treći (bilo pomoću petlje, bilo bibliotečkom funkcijom `copy`) i da se onda sortiraju, najbolje bibliotečkom funkcijom `sort`.

```
// objedinjavanje dva sortirana niza u treci
vector<int> objedini(const vector<int>& a,
                     const vector<int>& b) {
  // kopiramo dva niza u treci, jedan iza drugog
  vector<int> c(a.size() + b.size());
  copy(a.begin(), a.end(), c.begin());
  copy(b.begin(), b.end(), next(c.begin(), a.size()));
  // sortiramo treci niz
  sort(c.begin(), c.end());
  return c;
}
```

Kopiranje nizova dužine m i n zahteva $m + n$ operacija, a sortiranje $O((m + n) \cdot \log(m + n))$. Tehnički, mogli smo odmah elemente učitavati u rezultujući niz i tako uštedeti memoriju i vreme potrebno za kopiranje, ali dominatni faktor, a to je vreme potrebno za sortiranje bi ostao. Primetimo da u ovom rešenju nismo uopšte upotrebili činjenicu da su polazni elementi već sortirani.

Iako ovo rešenje po vremenu izvršavanja ne zaostaje puno u odnosu na optimalno (njegovo vreme izvršavanja je kvazilinearno tj. $O((m + n) \cdot \log(m + n))$, a op-

timalno vreme je linearno tj. $O(m + n)$), ono je dosta komplikovanije nego što je potrebno. To se u ovoj implementaciji ne vidi, jer je upotrebljena bibliotečka funkcija sortiranja, međutim, implementacija efikasnog algoritma sortiranja zahteva napredne tehnike programiranja. Interesantno, jedan od popularnih algoritama sortiranja je sortiranje objedinjavanjem (engl. merge sort), opisan u poglavlju **??**, u svom osnovnom koraku zahteva objedinjavanje dva sortirana niza u treći. Samim tim, donekle je besmisleno problem objedinjavanja rešavati svođenjem na komplikovaniji problem sortiranja.

Zadatak možemo rešiti efikasnim algoritmom, zasnovanom na tehnici dva pokazivača. *Algoritam objedinjavanja* (engl. *merge*) podrazumeva da su nizovi koji se objedinjavaju sortirani. Ako je jedan od nizova prazan, rezultat objedinjavanja je drugi niz i njegove elemente je potrebno jednostavno prekopirati u rezultat. Ako nizovi nisu prazni, pošto su sortirani, prvi element niza je ujedno najmanji u njemu. Manji od dva početna elementa je manji (ili jednak) od početnog elementa drugog niza, pa je manji ili jednak svim elementima u oba niza i samim tim je najmanji element od svih i treba da bude prvi u rezultatu. Kada se taj element ukloni iz niza, dobijamo problem istog tipa kao i polazni, koji se onda rešava na isti način. Implementacija može biti rekurzivna, međutim, rekurzija je repna i lako se eliminiše.

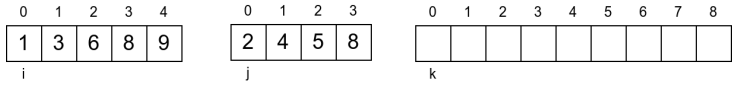
Tokom iterativne implementacije održavaju se dva pokazivača: promenljiva i koja ukazuje na poziciju tekućeg elementa prvog i j koja ukazuje na tekući element drugog niza. Dok su obe ove promenljive manje od dužine niza po kojem se kreću, poredimo elemente na tim pozicijama. Ako je element na poziciji i u prvom nizu manji (ili jednak) elementu na poziciji j u drugom nizu, tada taj element prepisujemo u treći niz (na poziciju k koju inicijalizujemo na nulu i uvećavamo prilikom dodavanja svakog novog elementa) i uvećavamo i za 1. U suprotnom, u treći niz prepisujemo element iz drugog niza sa pozicije j i uvećavamo j. Kada bar jedna od promenljivih dostigne dužinu odgovarajućeg niza, tada elemente preostalog niza prepisujemo u treći niz. Ne moramo eksplicitno proveravati da li u nekom od ovih nizova ima preostalih elemenata, već možemo u jednoj petlji kopirati preostale elemente prvog, a u drugoj petlji kopirati preostale elemente drugog niza (jedna od ovih petlji će biti prazna).

```
// objedinjava sortirani niz a sa n elemenata i sortirani niz b
// sa m elemenata smestajuci rezultat u sortirani niz c
vector<int> objedini(const vector<int>& a,
                     const vector<int>& b) {
```

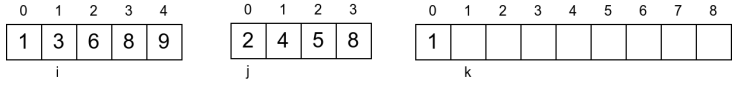
```
  int m = a.size(), n = b.size();
  vector<int> c(m + n);
  int i = 0, j = 0, k = 0;
  while (i < n && j < m)
    c[k++] = a[i] <= b[j] ? a[i++] : b[j++];
  while (i < n)
    c[k++] = a[i++];
  while (j < m)
    c[k++] = b[j++];
  return c;
}
```

Primer 3.7.1. *Prikažimo rad ovog algoritma i na jednom primeru.*

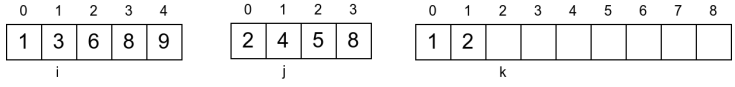
- *Pretpostavimo da je potrebno objediniti naredna dva niza.*



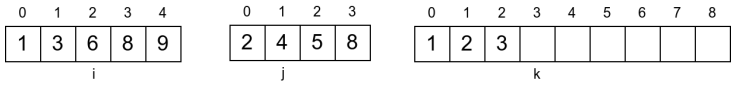
- *U prvom koraku je* $i = 0$ *i* $j = 0$*, pa se porede elementi na pozicijama* 0*, tj. elementi* 1 *i* 2*. Pošto je* 1 *manji, on se prepisuje u rezultujući niz i uvećava se levi pokazivač.*



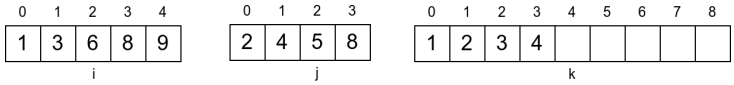
- *Sada je* $i = 1$ *i* $j = 0$*, pa se porede elementi na pozicijama* 1 *i* 0*, tj.* 3 *i* 2*. Pošto je* 2 *manji, on se prepisuje u rezultujući niz i uvećava se desni pokazivač.*



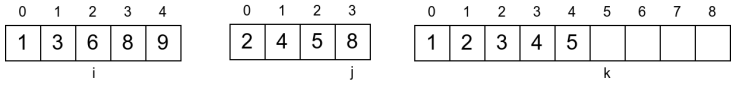
- *Sada je* $i = 1$ *i* $j = 1$*, pa se porede elementi na pozicijama* 1 *i* 1*, tj.* 3 *i* 4*. Pošto je* 3 *manji, on se prepisuje u rezultujući niz i uvećava se levi pokazivač.*



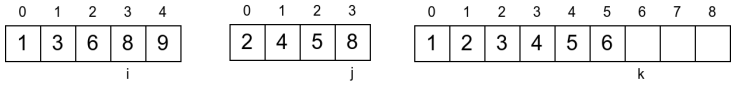
- *Sada je* $i = 2$ *i* $j = 1$*, pa se porede elementi na pozicijama* 2 *i* 1*, tj.* 6 *i* 4*. Pošto je* 4 *manji, on se prepisuje u rezultujući niz i uvećava se desni pokazivač.*



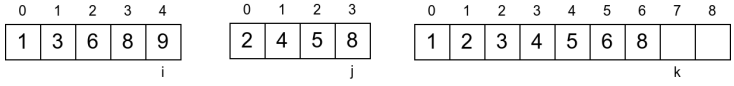
- *Sada je* $i = 2$ *i* $j = 2$*, pa se porede elementi na pozicijama* 2 *i* 2*, tj.* 6 *i* 5*. Pošto je* 5 *manji, on se prepisuje u rezultujući niz i uvećava se ponovo desni pokazivač.*



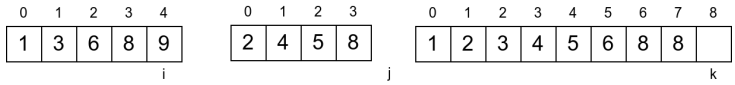
- *Sada je* $i = 2$ *i* $j = 3$*, pa se porede elementi na pozicijama* 2 *i* 3*, tj.* 6 *i* 8*. Pošto je* 6 *manji, on se prepisuje u rezultujući niz i uvećava se levi pokazivač.*



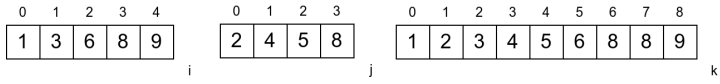
- *Sada je* $i = 3$ *i* $j = 3$*, pa se porede elementi na pozicijama* 3 *i* 3*, tj.* 8 *i* 8*. Pošto su jednaki, bilo koji od njih (na primer levi) može biti prepisan u rezultujući niz i odgovarajući pokazivač se uvećava.*



- *Sada je* $i = 4$ *i* $j = 3$*, pa se porede elementi na pozicijama* 4 *i* 3*, tj.* 9 *i* 8*. Pošto je* 8 *manji, on se prepisuje u rezultujući niz i uvećava se ponovo desni pokazivač.*



- *Pošto je $j = 4$, desni niz se ispraznio, pa u rezultujući niz prepisujemo jedini preostali element iz levog niza.*



Svaki pokazivač prolazi kroz jedan od dva niza i ukupan broj koraka je $m + n$, pa je složenost ovog algoritma $O(m + n)$.

Postupak možemo opisati i rekurzivno. Izlaz iz rekurzije čini slučaj kada je jedan od nizova prazan i rezultat u tom slučaju čine elementi drugog niza (koji su sortirani). Ako su nizovi neprazni, bira se minimalni element oba niza, stavlja se na početak rezultata, izbacuje se iz svog niza i postupak se rekurzivno primenjuje na rep tog niza (deo niza dobijen izbacivanjem početnog elementa) i čitav drugi niz. Pošto su nizovi sortirani, minimalni element oba niza se dobija poređenjem njihovih početnih elemenata (pošto je on na početku sortiranog niza u kom se nalazi manji je ili jednak od svih elemenata u tom nizu, a pošto je manji ili jednak od početnog elementa drugog, sortiranog niza, on je manji ili jednak i od svih elenata u tom nizu). Korektnost dalje jednostavno sledi indukcijom.

3.7.3 *Filtriranje niza*

Filtriranje podrazumeva da se iz niza izdvoje samo elementi koji zadovoljavaju neko svojstvo. Te elemente možemo premestiti u neki drugi niz, a možemo ih i zadržati u istom nizu, tako što ćemo ih pomeriti ka početku niza. Pokazivač `i` se može kretati redom kroz elemente niza, dok će pokazivač `k` označavati prvo trenutno slobodno mesto u nizu tj. broj elemenata koji su prebačeni na početak niza.

```
void izdvojParne(vector<int>& a) {
  int k = 0;
  for (int i = 0; i < a.size(); i++)
     if (a[i] % 2 == 0)
        a[k++] = a[i];
  a.resize(k);
```

```
}
```

Složenost prikazane funkcije je, očigledno, $O(n)$, gde je n dužina niza.

U nastavku ćemo prikazati još nekoliko primera koji se efikasno rešavaju primenom tehnike dva pokazivača.

Zadatak: Broj parova datog zbira

Dat je ceo broj s i niz različitih celih brojeva. Napisati program kojim se određuje broj parova u nizu koji imaju zbir jednak datom broju s.

Opis ulaza

U prvoj liniji standardnog ulaza nalazi se ceo broj s (broj iz intervala $[0, 10^6]$), u drugoj liniji nalazi se broj elemenata niza n ($1 \leq n \leq 50000$), a u trećoj liniji se nalaze redom elementi niza (brojevi iz intervala $[0, 10^6]$).

Opis izlaza

Na standardnom izlazu prikazati broj parova različitih elemenata niza čiji je zbir jednak broju s.

Primer

Ulaz	*Izlaz*	*Objašnjenje*

`5`
`6`
`1 4 3 6 -1 5`
	`2`	To su parovi $(1, 4)$ i $(6, -1)$.

Rešenje

Iterativni obilazak sa dva kraja niza pomoću dva pokazivača

Zadatak možemo rešiti tako što sortiramo niz neopadajuće (pošto su svi elementi različiti, on će zapravo biti sortiran strogo rastuće) i primenimo tehniku dva pokazivača, implementiranu iterativno.

Članove datog zbira možemo tražiti polazeći sa oba kraja niza. Obilazimo niz sa oba kraja: levog ($l = 0$) i desnog ($d = n - 1$). Uporedimo $a_l + a_d$ sa s.

- Ako je $a_l + a_d > s$ potrebno je smanjiti zbir para elemenata, što postižemo zamenom elementa manjim, pa pošto je niz sortiran u rastućem poretku, prelazimo na sledeći element u desnom delu niza (d umanjujemo za 1).

- Ako je $a_l + a_d < s$ potrebno je povećati zbir para elemanata, što postižemo zamenom elementa većim, pa pošto je niz sortiran u rastućem poretku, prelazimo na sledeći element u levom delu niza (l uvećavamo za 1).
- Ako je $a_l + a_d = s$ uvećamo broj traženih parova, i prelazimo na sledeći element u levom delu niza (l uvećavamo za 1) i na sledeći element u desnom delu niza (d umanjujemo za 1).

Proces nastavljamo dok ne obiđemo ceo niz, to jest dok je $l < d$.

Primer 3.7.2. *Prikažimo ovo na primeru pronalaženja elemenata čiji je zbir* 14 *u sortiranom nizu* $1, 2, 5, 7, 9, 11, 13, 14$. *Krećemo od para* $(1, 14)$. *Pošto je zbir veći od traženog, pomeramo desni kraj ulevo i analiziramo par* $(1, 13)$, *koji ima traženi zbir. Zato prelazimo na* $(2, 11)$. *Pošto je zbir sada manji, pomeramo levi kraj udesno i analiziramo par* $(5, 11)$. *Zbir je preveliki i pomeramo desni kraj ulevo i analiziramo par* $(5, 9)$. *On ima traženi zbir, pa prelazimo na* $(7, 7)$, *no taj par ne analiziramo, jer su se pokazivači susreli.*

```
int brojParovaDatogZbira(const vector<int>& a, int s) {
  // pravimo sortiranu kopiju niza a
  auto as = a;
  sort(begin(as), end(as));

  // brojimo parove pomocu dva pokazivaca
  int brojParova = 0;
  int levo = 0, desno = as.size() - 1;
  while (levo < desno)
    if (as[levo] + as[desno] > s)
      desno--;
    else if (as[levo] + as[desno] < s)
      levo++;
    else {
      brojParova++;
      levo++;
      desno--;
    }
```

```
    return brojParova;
}
```

Pošto se u svakom koraku razlika između d i l smanji bar za 1 (nekada i za 2), ukupan broj koraka ne može biti veći od n, pa je složenost ovog dela algoritma $O(n)$. Naravno, složenošću dominira prvobitno sortiranje čija je složenost $O(n \log n)$.

3.8 Primena efikasnih struktura podataka

Izbor strukture podataka može u velikoj meri uticati na efikasnost. U nastavku ćemo prikazati nekoliko primera koji ovo ilustruju.

3.8.1 Primena skupova i mapa

Većina savremenih programskih jezika nudi gotove tipove podataka koji implementiraju skupove i mape. U jeziku C++ su to `set` i `map` koji omogućavaju osnovne operacije (umetanje, pretraga, brisanje) u složenosti $O(\log n)$, gde je n broj elemenata skupa tj. ključeva u mapi i `unordered_set` i `unordered_map` čija je amortizovana složenot osnovnih operacija $O(1)$, ali je složenost najgoreg slučaja $O(n)$. Iako imaju malo lošiju prosečnu složenost operacija, uređeni skupovi nude neke dodatne operacije – pronalaženje najmanjeg ili najvećeg elemenata (pomoću `begin` i `end`), obilazak elemenata u uređenom redosledu i pronalaženje najmanjeg elementa koji je veći ili jednak datoj vrednosti (metodom `lower_bound`) ili najmanjeg elementa koji je strogo veći od date vrednosti (metodom `upper_bound`). Pošto ove metode vrše binarnu pretragu, njihova složenost je takođe $O(\log n)$.

Ako dopuštamo da skup sadrži duplikate, možemo koristiti `multiset` i `unordered_multiset`, koji su iste efikasnosti kao `set` i `unordered_set`.

Ilustrujmo na primeru sortiranja niza kako izbor pogodne strukture podataka čini algoritam efikasnijim. Algoritam sortiranja umetanjem (engl. insertion sort) radi tako što se jedan po jedan element ubacuje na svoje mesto u sortiranom nizu. Kada se koristi niz, umetanje elementa je linearne složenosti (jer se svi elementi iza njega moraju pomeriti za jedno mesto udesno). Ako se umesto niza elementi ubacuju u uređeni skup, isti algoritam radi mnogo efikasnije. Naime, umetanje svakog elementa je složenost $O(\log k)$, pa je ukupna složenost umetanja svih n elemenata složenosti $O(n \log n)$. Nakon toga se ispis svih elemenata uređenog multiskupa vrši u linearnom vremenu $O(n)$.

```
int n;
cin >> n;
multiset<int> a;
for (int i = 0; i < n; i++) {
   int x;
   cin >> x;
   a.insert(x);
}
for (int x : a)
   cout << x << endl;
```

Dakle, uz korišćenje pogodne strukture podataka isti algoritam može biti mnogo efikasniji. Ilustrujmo ovo još jednim primerom.

Zadatak: Kupovina računara

U prodavnici laptop računara se često pojavljuju novi modeli, dok se stari modeli rasprodaju. Poznata je cena svakog računara. Kupci dolaze sa željom da kupe računare koji su u okviru njihovog budžeta. Prodavnica radi veoma pošteno, tako da skuplji računari uvek imaju i bolje performanse. Zato svaki korisnik želi da kupi što skuplji računar koji može da priušti.

Opis ulaza

Svaka linija standardnog ulaza počinje karakterom `i`, `e` ili `f`.

- Linije koje počinju sa `i` označavaju da je novi računar stigao u prodavnicu i nakon slova `i` navedena je njegova cena (prirodan broj manji od 10^9).
- Linije koje počinju sa `e` označavaju da je neki računar prodat i nakon slova `e` navedena je cena tog računara.
- Linije koje počinju sa `f` označavaju da kupac želi da kupi najskuplji računar trenutno u prodavnici čija je cena manja ili jednaka od iznosa budžeta navedenog nakon slova `f`.

Opis izlaza

Na standardni izlaz ispisati redom rezultate svih upita koje su organizatori postavili (ako nijedan računr ne može da se kupi za navedeni iznos, ispisati `-`).

Primer

Ulaz	*Izlaz*
`i 38000`	`-`
`i 50000`	`50000`
`i 50000`	`-`
`i 83299`	`50000`
`f 30000`	`38000`
`f 55000`	`83299`
`e 50000`	
`f 10000`	
`f 60000`	
`e 50000`	
`f 60000`	
`f 90000`	

Rešenje

Gruba sila

Rešenje grubom silom podrazumeva da se cene svih računara u prodavnici čuvaju u nizu. Dodavanje je moguće vršiti na kraj niza, brisanje elementa sa prve pozicije na kojoj se nalazi, dok se pronalazak najboljeg računara koji kupac može da priušti može vršiti jednim prolaskom kroz niz i analizom svih elemenata.

Dodavanje elementa na kraj niza je složenosti $O(1)$, ali su brisanje i prebrojavanje složensti $O(n)$, tako da je ukupna složenost $O(q^2)$, gde je q broj linija (najgori slučaj može biti kada se prvo učita $q/2$ linija kojima se elementi unose u niz, a zatim $q/2$ linija kojima se pretražuju elementi niza).

Uređen skup

Da se vrši samo umetanje i jedno prebrojavanje na kraju, niz bi mogao da se sortira i da se binarnom pretragom pronađe traženi računar. Pošto se u ovom zadatku vrši više prebrojavanja a elementi mogu i da se brišu, u efikasnom rešenju nije moguće koristiti običan niz. Umesto toga, brojave ćemo čuvati u uređenom (multi)skupu koji omogućava efikasno dodavanje, brisanje i binarnu pretragu.

Uređen multiskup je implementiran kroz bibliotečku kolekciju `multiset` a za njegovu binarnu pretragu koristimo metodu `upper_bound` (ona vraća iterator na poziciju prve vrednosti koja je strogo veća od date, pa se tražena vrednost nalazi na prethodnoj poziciji). Za brisanje koristimo metodu `erase`, kojoj moramo da predamo

iterator koji ukazuje na element koji želimo da obrišemo (ako navedemo vrednost, biće obrisana sva pojavljivanja te vrednosti). Pretragu vršimo metodom `find` (koja takođe binarno pretražuje drvo).

Svaka od navedenih operacija sa uređenim multiskupom se vrši u vremenu $O(\log n)$, gde je n trenutni broj elemenata u multiskupu. Ukupno izvršavanje q upita je $O(q \log q)$.

```
multiset<int> cene;
char c;
while (cin >> c) {
  if (c == 'i') {
    int cena;
    cin >> cena >> ws;
    cene.insert(cena);
  } else if (c == 'e') {
    int cena;
    cin >> cena >> ws;
    auto it = cene.find(cena);
    if (it != a.end())
      cene.erase(it);
  } else if (c == 'f') {
    int budzet;
    cin >> budzet >> ws;
    // pozicija najmanje cene strogo vece od budzeta
    auto it = cene.upper_bound(budzet);
    if (it != cene.begin()) {
      // trazimo prethodnu cenu
      --it;
      cout << *it << '\n';
    } else
      cout << "-" << '\n';
  }
}
```

3.8.2 *Primena stekova i redova*

Stek i red su strukture podataka koje se prirodno javljaju i koriste u mnogim primenama. U ovom poglavlju ćemo prikazati kako se mogu upotrebiti i da se snizi složenost nekih algoritama.

Zadatak: Brisanje parova uzastopnih jednakih karaktera

Niska se skraćuje dokle god je to moguće tako što joj se briše prvi par jednakih uzastopnih karaktera. Napiši program koji određuje skraćenu nisku.

Opis ulaza

Sa standardnog ulaza se učitava niska sastavljena od malih slova engleske abecede, dužine n ($1 \leq n \leq 10^6$).

Opis izlaza

Na standardni izlaz ispisati skraćenu nisku.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`babccbddabbcaa`	`bc`	Skraćivanje teče sledećim redosledom `babccbddabbcaa`, `babbddabbcaa`, `baddabbcaa`, `baabbcaa`, `bbbcaa`, `bcaa`, `bc`.

Rešenje

Brisanje karaktera sa početka ili iz sredine niske zahteva pomeranje ostalih karaktera, što dovodi do neefikasnog programa. Umesto toga, moguće je kreirati novu nisku obrađujući jedan po jedan karakter polazne niske. Pretpostavićemo da smo obradili prvih k karaktera niske i da smo brisanjem svih pojavljivanja uzastopnih jednakih karaktera dobili nisku t.

- Ako je niska t prazna ili ako je poslednji karakter niske t različit od tekućeg karaktera s_k niske s (onog na poziciji k), karakter s_k treba dodati na t,
- U suprotnom (ako je poslednji karakter niske t jednak karakteru s_k) treba ukloniti poslednji karakter niske t i time (u oba slučaja) dobijamo rezultat obrade prvih $k + 1$ karaktera niske s.

Primetimo da se niska t ponaša kao stek (karakteri joj se dodaju i uklanjaju sa desnog kraja). Tip `string` u jeziku C++ podržava metode `empty`, `back`, `push_back` i

`pop_back` koji se izvršavaju u složenosti $O(1)$, pa se taj tip može koristiti kao stek karaktera.

Pošto su sve operacije za rad sa stekom složenosti $O(1)$ i svaki karakter se najviše jednom može dodati i jednom može ukloniti sa steka, složenost ovog algoritma je $O(n)$. I memorijska složenost je takođe $O(n)$.

```
char c;
string t;
while (cin >> c && c != '\n')
  if (t.empty() || c != t.back())
    t.push_back(c);
  else
    t.pop_back();
cout << t << endl;
```

Zadatak: Maksimalna bijekcija

Filmski producent organizuje večeru na koju želi da pozove glumce. Da bi se glumci osećali prijatno na večeri, producent želi da obezbedi da je svaki glumac prisutan na večeri omiljen glumac nekog drugog glumca prisutnog na večeri. Svaki od n glumaca, potencijalnih gostiju, odabrao je svog omiljenog glumca iz tog skupa glumaca (pri čemu nije isključeno i da je neki glumac odabrao sam sebe). Napiši program koji određuje najveći podskup tog skupa glumaca koji sadrži glumce koje producent može pozvati na večeru.

Opis ulaza

Sa standardnog ulaza unosi se broj n ($1 \leq n \leq 50000$) koji predstavlja broj glumaca koji su glasali, a zatim i redom redni brojevi omiljenog glumca svakog glumca (svi brojevi su između 0 i $n-1$).

Opis izlaza

Na standardni izlaz ispiši najveći broj glumaca koji mogu prisustvovati večeri.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`7` `2 0 0 4 4 3 5`	`3`	Na večeru mogu biti pozvani glumci sa brojevima $0, 2, 4$. Glumac 0 je omiljeni glumac glumca 2, glumac 2 je omiljeni glumac glumca 0, dok je glumac 4 sam sebi omiljen.

Rešenje

Glasovi glumaca određuju funkciju f definisanu na skupu $\{0, 1, \ldots, n-1\}$. Neka je skup S skup glumaca koji su pozvani na večeru. Da bi svaki glumac bio omiljen glumac nekom drugom glumcu iz skupa S, potrebno je da restrikcija funkcije f na skup S bude "na" (tj. da za svaku sliku postoji original koji se slika u tu sliku). Pošto je skup S konačan, na osnovu Dirihleovog principa, ona će ujedno biti i "1-1" tj. bijekcija (za svaku sliku će postojati tačno jedan original koji se u nju slika). Zaista, ako bi neki glumac na večeri bio omiljen za dva različita glumca, nekom glumcu na večeri bi nedostajao glumac kome bi on bio omiljen.

Direktan, ali veoma neefikasan način da se ovaj zadatak reši je da se nabroje svi mogući podskupovi i da se za svaki od njih proveri da li je restrikcija f na taj podskup bijekcija.

Eliminacija elemenata

Efikasan algoritam možemo napraviti ako pronađemo potreban uslov da element bude deo skupa S. Naime, svaki element skupa S mora biti slika tačno jednog elementa skupa S. Ako je svaki element skupa X (domena funkcije f) slika tačno jednog elementa skupa X, tada je f bijekcija na skupu X. U suprotnom mora da postoji element koji nije slika ni jednog elementa skupa X i taj element ne može biti deo skupa S. Kada taj element uklonimo (zajedno sa njegovom slikom), dobijamo skup koji je za jedan element manji i na koji možemo primeniti isti postupak (suštinski, imamo opisan induktivni tj. rekurzivni postupak).

Ovaj pristup se može jednostavno implementirati tako što za svaki element skupa X izračunamo broj elemenata koji se slikaju u njega. To možemo uraditi korišćenjem jednostavnog, asocijativnog niza.

Možemo održavati radnu listu (red) elemenata u koje se ne slika ni jedan element (nakon izračunavanja broja elemenata koji se slikaju u svaki od elemenata skupa X, sve brojeve za koje je vrednost u asocijativnom nizu nula, ubacujemo u radnu listu). Nakon toga, sve dok se radna lista ne isprazni, uzimamo jedan po jedan element iz radne liste, izbacujemo ga iz skupa X i zato smanjujemo broj elemenata koji se slikaju u sliku tog izbačenog elementa (umanjujemo vrednost u asocijativnom nizu). Ako se ustanovi da se nakon toga vrednost slike u asocijativnom nizu smanjila na nulu, tada se slika ubacuje u radnu listu. Iako redosled uzimanja elemenata iz radne liste može biti potpuno proizvoljan, za implementaciju se najčešće koristi red (jer daje nekakav osećaj pravičnosti). Rešenje u kom bi se umesto reda koristio stek bilo

bi takođe ispravno.

```
// izračunavamo ulazni stepen svakog elementa
vector<int> ulazniStepen(n, 0);
for (int i = 0; i < n; i++)
  ulazniStepen[f[i]]++;
// red u kom se čuvaju elementi ulaznog stepena 0
// oni ne mogu biti deo domena bijekcije
queue<int> q;
for (int i = 0; i < n; i++)
  if (ulazniStepen[i] == 0)
    q.push(i);
// izbacujemo jedan po jedan element iz reda
int broj_elemenata = n;
while (!q.empty()) {
  int i = q.front(); q.pop();
  broj_elemenata--;
  // uklanjamo sliku tog elementa i ako nakon toga
  // slika dobije ulazni stepen nula ubacujemo je u red
  if (--ulazniStepen[f[i]] == 0)
    q.push(f[i]);
}
cout << broj_elemenata << endl;
```

3.8.3 *Primena redova sa prioritetom*

Red sa prioritetom je vrsta reda u kome elementi imaju na neki način pridružen prioritet, dodaju se u red jedan po jedan, a uvek se iz reda uklanja onaj element koji ima najveći prioritet od svih elemenata u redu. Podsetimo se, u jeziku C++ red sa prioritetom se realizuje klasom `priority_queue<T>`, gde je `T` tip elemenata u redu. Red sa prioritetom podržava sledeće metode:

- `push` - dodaje dati element u red
- `pop` - uklanja element sa najvećim prioritetom iz reda (pod pretpostavkom da red nije prazan). Naglasimo da je ova metoda tipa `void` i da ne vraća uklonjeni element.

- `top` - očitava element sa najvećim prioritetom (pod pretpostavkom da red nije prazan)
- `empty` - proverava da li je red prazan
- `size` - vraća broj elemenata u redu

Operacije `push` i `pop` su složenosti $O(\log k)$, gde je k broj elemenata u redu, dok su ostale operacije složenosti $O(1)$.

Ilustrujmo kako se korišćenjem redova sa prioritetom može poboljšati složenost algoritama sortiranja. Koristi se algoritam sortiranja uz pomoć hipa (engl. heap sort) koji je varijacija algoritma sortiranja selekcijom (engl. selection sort) u kojem se, podsetimo se, u svakom koraku najmanji element dovodi na početak niza. Hip je struktura podataka koja se najčešće koristi za implementaciju reda sa prioritetom. Algoritam hip sort koristi činjenicu da je određivanje i uklanjanje najmanjeg elementa iz reda sa prioritetom prilično efikasna operacija. Stoga se sortiranje može realizovati tako što se svi elementi umetnu u red sa prioritetom (implementiran pomoću strukture hip), iz koga se zatim pronalazi i uklanja jedan po jedan najmanji element.

I ubacivanje elemenata u red sa prioritetom i izbacivanje elemenata iz reda sa prioritetom obično je složenosti $O(\log k)$, gde je k broj elemenata u redu sa prioritetom. Stoga je ukupna složenost ovog algoritma sortiranja $O(n \log n)$.

```
// ovo je način da se u C++-u definiše red sa prioritetom u
// kome su elementi poređani u opadajućem redosledu prioriteta
// (ovde, vrednosti)
priority_queue<int, vector<int>, greater<int>> Q;

// učitavamo sve elemente niza i ubacujemo ih u red
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int ai;
  cin >> ai;
  Q.push(ai);
}
// vadimo jedan po jedan element iz reda i ispisujemo ga
while (!Q.empty()) {
```

```
  cout << Q.top() << endl;
  Q.pop();
}
```

Zadatak: Zbir k najboljih

Student je radio n zadataka i za svaki zadatak je dobio određeni broj poena. Odrediti zbir poena na k zadataka koje je najbolje uradio.

Opis ulaza

U prvoj liniji standardnog ulaza dat je prirodan broj n ($1 \leq n \leq 10^6$) – broj zadataka koje je učenik radio, u drugoj prirodan broj k ($1 \leq k \leq n$) – broj zadataka koje je najbolje uradio, a u trećoj n brojeva broj poena koje je dobio na zadacima.

Opis izlaza

Ukupan broj poena koje je osvojio na k najbolje ocenjenih zadataka.

Primer

Ulaz	*Izlaz*

`10`
`3`
`15 80 25 60 10 20 50 45 40 30`
	`190`

Rešenje

Red sa prioritetom

Najvećih k do sada viđenih elemenata niza možemo čuvati u strukturi podataka koja nam omogućava da pronađemo najmanji element u njoj i da ga eventualno zamenimo onim koji je trenutno učitan (ako je trenutno učitani element veći od njega). Idealna struktura za to je hip tj. red sa prioritetom.

Red sa prioritetom u jeziku C++ možemo dobiti pomoću `priority_queue`. Elemente u red možemo ubaciti metodom `push`. Element koji je najmanji možemo očitati metodom `top` i izbaciti metodom `pop`.

Na početku red popunjavamo sa k prvih učitanih elemenata, a zatim svaki naredni učitani element poredimo sa najmanjim u redu i ako je veći od njega, najmanji izbacujemo, a učitani element ubacujemo.

Pošto je složenost metoda za ubacivanje i izbacivanje iz reda sa prioritetom logaritamska, a metode za očitavanje najmanjeg elementa konstantna, vremenska složenost ovog algoritma je $O(n \cdot \log(k))$, dok je prostorna složenost $O(k)$.

Primetimo da je ovaj algoritam donekle sličan algoritmu sortiranja uz pomoć hipa tj. algoritma Hip-sort (HeapSort).

```
int n, k;
cin >> n >> k;

// red sa prioritetom koji cuva k najvecih elemenata koristi
// se min-hip, koji omogucava brzo uklanjanje najmanjeg
// elementa
priority_queue<int, vector<int>, greater<int>> pq;

// ucitavamo prvih k elemenata i ubacujemo ih u red
for (int i = 0; i < k; i++) {
  int x;
  cin >> x;
  pq.push(x);
}

// ucitavamo preostale elemente
for (int i = k; i < n; i++) {
  int x;
  cin >> x;
  // ako je ucitani element veci od najmanjeg trenutno u
  // redu izbacujemo taj najmanji i menjamo ga ucitanim
  if (x > pq.top()) {
    pq.pop();
    pq.push(x);
  }
}

// izbacujemo elemente iz reda racunajuci njihov zbir i
// ispisujemo ga
int s = 0;
while (!pq.empty()) {
  s += pq.top();
  pq.pop();
```

```
}
cout << s << endl;
```

3.A Dodatak: elementarne tehnike poboljšanja složenosti

U nastavku ćemo prikazati još neke primere koji ilustruju primenu opisanih tehnika poboljšanja složenosti algoritama.

3.A.1 Zamena iteracije formulom

Zadatak: Broj deljivih u intervalu

Napiši program koji određuje koliko u intervalu $[a, b]$ postoji brojeva deljivih brojem k.

Opis ulaza

Sa standardnog ulaza unose se tri cela broja, svaki u posebnom redu: a ($0 \leq a \leq 10^9$), b ($a \leq b \leq 10^9$), k ($1 \leq k \leq 10^9$).

Opis izlaza

Na standardni izlaz ispisati traženi ceo broj.

Primer

Ulaz	*Izlaz*	*Objašnjenje*

`30`
`53`
`5`
	`5`	Brojevi su 30, 35, 40, 45 i 50.

Rešenje

Linearna pretraga

Moguće je napraviti rešenje zasnovano na pretrazi grubom silom, i koje redom prolazi kroz sve brojeve od a do b, proverava za svaki da li je deljiv brojem k i za svaki koji jeste, uvećava brojač pronađenih brojeva. To rešenje je najjednostavnije za razumevanje i implementaciju, međutim može biti neefikasno.

```
// broj brojeva u intervalu [a, b] deljivih brojem k
int brojDeljivih(int a, int b, int k) {
  int broj = 0;
```

```
  for (int i = a; i <= b; i++)
    if (i % k == 0)
      broj++;
  return broj;
}
```

Složenost ovog rešenja je linearna u odnosu na broj elemenata u intrevalu tj. $O(b - a)$.

Izračunavanje broja deljivih brojeva

Da bi broj x bio deljiv brojem k potrebno je da postoji neko n tako da je $x = n \cdot k$. Pošto x mora biti u intervalu $[a, b]$, mora da važi da je $a \leq n \cdot k$ i $n \cdot k \leq b$. Najmanje n koje zadovoljava prvu nejednačinu jednako je $n_l = \lceil \frac{a}{k} \rceil$, najveće n koje zadovoljava drugu nejednačinu jednako je $n_d = \lfloor \frac{b}{k} \rfloor$. Bilo koji broj iz intervala $[n_l, n_d]$ zadovoljava obe nejednakosti i predstavlja količnik nekog broja iz intervala $[a, b]$ brojem k. Slično, bilo koji broj iz intervala $[a, b]$ deljiv brojem k daje neki količnik iz intervala $[n_l, n_d]$. Zato je traženi broj brojeva iz intervala $[a, b]$ koji su deljivi brojem k jednak broju brojeva u intervalu $[n_l, n_d]$ a to je $n_d - n_l + 1$ ako je $n_d \geq n_l$, tj. 0 ako je taj interval prazan tj. ako je $n_d < n_l$. Brojevi n_l i n_d se mogu odrediti, na primer, grananjem.

```
// broj brojeva u intervalu [a, b] deljivih brojem k
int brojDeljivih(int a, int b, int k) {
  int nl = a % k == 0 ? a/k : a/k + 1; // ceil(a/k)
  int nd = b / k;                      // floor(b/k)
  int broj = nd >= nl ? nd-nl+1 : 0;
  return broj;
}
```

Složenost ovog rešenja je, očigledno, konstantna tj. $O(1)$.

Zadatak: Maksimalna površina nakon produženja stranica pravougaonika

Dat je pravougaonik dimenzije $a \times b$, gde su brojevi a i b celi. Kolika je maksimalna površina pravougaonika koji se može dobiti produžavanjem stranica tog pravougaonika za ukupnu dužinu c, gde je c ceo broj i dužine stranica novog pravougaonika su takođe celi brojevi?

Opis ulaza

Sa standardnog ulaza se učitavaju prirodni brojevi $a, b, c \leq 10^9$, razdvojeni sa po jednim razmakom.

Opis izlaza

Na standardni izlaz ispisati maksimalnu površinu nakon produženja stranica.

Primer 1

Ulaz	*Izlaz*	*Objašnjenje*
`5 10 3`	`80`	Dimenzija nakon proširenja će biti 8×10.

Primer 2

Ulaz	*Izlaz*	*Objašnjenje*
`9 10 4`	`132`	Dimenzija nakon proširenja će biti 11×12.

Primer 3

Ulaz	*Izlaz*
`14 17 5`	`324`

Rešenje

Gruba sila

Zadatak može biti rešen grubom silom, tj. isprobavanjem svih mogućnosti raspodele dodatne dužine. Za svaku dužinu i između 0 i c, dodajemo i stranici a i $c-i$ stranici b, izračunavamo površinu i tražimo maksimum tako dobijenih površina.

```
long long maksPovrsina(long long a, long long b, long long c) {
  long long maks = a * (b + c);
  for (long long i = 1; i <= c; i++)
    maks = max(maks, (a+i)*(b+c-i));
  return maks;
}
```

Složenost ovog rešenja je $O(c)$.

Izračunavanje maksimalnog prinosa

Dokažimo da od svih pravougaonika datog fiksiranog obima, najveću površinu ima kvadrat. Neka je obim pravougaonika jednak $2(x + y) = 2s$. Njegova površina jednaka je $x \cdot y = x \cdot (s - x) = x \cdot s - x \cdot x = s^2/4 - (x - s/2)^2$. Pošto je

$(x - s/2)^2 \geq 0$, površina ne može biti veća od $s^2/4$, a jednaka je toj vrednosti kada je $x = y = s/2$. Dakle, u zadatom problemu, uvećanje treba napraviti tako da se dobije oblik koji je što sličniji kvadratu (dobijanje kvadrata u nekim slučajevima nije moguće).

Neka je $a \leq b$. Ako je $a + c \leq b$, tada celokupan iznos uvećanja c treba dodati na manju stranicu a. U suprotnom, prvo se kraća stranica a produži tako da postane jednaka dužoj stranici b, a zatim se preostali iznos uvećanja $(c - (b - a))$ podeli što ravnomernije moguće (ako je to paran broj može se dobiti kvadrat, a ako nije, tada se dobija pravougaonik kod kojeg je jedna stranica za jedan duža od druge). U implementaciji dužine novih stranica možemo izračunati tako što dužu stranicu b uvećamo za $\left\lfloor \frac{c-(b-a)}{2} \right\rfloor$ i za $\left\lceil \frac{c-(b-a)}{2} \right\rceil = \left\lfloor \frac{c-(b-a)+1}{2} \right\rfloor$.

```
long long maksPovrsina(long long a, long long b, long long c) {
  if (a > b) swap(a, b);
  if (c <= b - a)
    a += c;
  else {
    long long preostalo = c - (b - a);
    a = b + preostalo / 2;
    b = b + (preostalo + 1) / 2;
  }
  return a*b;
}
```

Složenost ovog rešenja je $O(1)$.

Primetimo da smo u ovom optimizacionom problemu uspeli da damo preciznu matematičku karakterizaciju traženog maksimuma (što je bilo moguće jer je funkcija koja se optimizovala bila jednostavna, u ovom slučaju - kvadratna) i time izbegli iterativnu pretragu. Generalno, kada se rešavaju optimizacioni problemi uvek ima smisla razmisliti da li je funkcija koja se optimizuje možda takva da se maksimum može izračunati matematičkim metodama da bi se u potpunosti izbegla pretraga ili bar okarakterisati na neki način koji omogućava da se značajno redukuje broj kandidata koje treba proveriti.

Zadatak: Aritmetički kvadrat

Brojevi od 0 do $n^2 - 1$ređaju se u kvadrat. Na primer, za $n = 5$ dobija se sledeći kvadrat:

```
0   1  2  3  4
5   6  7  8  9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

Napiši program koji izračunava zbir brojeva u datoj vrsti i datoj koloni kvadrata (i njih ćemo brojati od nule).

Opis ulaza

Sa standardnog ulaza učitavaju se sledeći celi brojevi (svaki u posebnom redu).

- n - dimenzija kvadrata ($1 < n \leq 10^9$)
- i - broj vrste i kolone čiji se zbir traži ($0 \leq i < n$)

Opis izlaza

Na standardni izlaz ispisati dva cela broja, svaki u posebnom redu:

- zbir brojeva u i-toj vrsti kvadrata i
- zbir brojeva u i-toj koloni kvadrata

Ako su ovi zbirovi veći ili jednaki od 10^9, ispisati njihovih poslednjih 9 cifara.

Primer

Ulaz	*Izlaz*
5	35
1	55

Rešenje

Iteracija

Rešenje se može dobiti upotrebom petlji i algoritma za sabiranje serije brojeva, pri čemu je svako sabiranje potrebno vršiti po modulu 10^9 (da ne bi došlo do prekoračenja i da bi se kod velikih brojeva dobilo samo poslednjih 9 cifara).

Početak vrste i izračunavamo tako što broj n saberemo i puta (početak svake naredne vrste veći je za n od prethodne, dok prva vrsta počinje nulom). Naravno, jasno je da će se dobiti vrednost $i \cdot n$, pa bi ova petlja lako mogla da se izbegne. Zbir vrste izračunavamo tako što sabiramo jedan po jedan element te vrste, pri čemu je svaki element za jedan veći od prethodnog (ako je početak vrste p, sabiramo brojeve p, $p+1$, …, $p+n-1$).

Prvi element kolone sa rednim brojem i je i. Svaki naredni element te kolone za n je veći od prethodnog. Na taj način izračunavamo svih n elemenata te kolone, dodajući ih jedan po jedan na zbir.

Složenost iterativnog algoritma za izračunavanje zbira svake vrste i svake kolone je $O(n)$.

```
typedef long long ll;

int main() {
  ll n, i;
  cin >> n >> i;

  const ll MOD = 1e9;

  ll pocetakVrste = 0;
  for (ll k = 0; k < i; k++)
    pocetakVrste = (pocetakVrste + n) % MOD;
  ll zbirVrste = 0;
  for (int k = pocetakVrste; k < pocetakVrste + n; k++)
    zbirVrste = (zbirVrste + k) % MOD;
  cout << zbirVrste << endl;

  ll elementKolone = i;
  ll zbirKolone = i;
  for (ll k = 1; k < n; k++) {
    elementKolone = (elementKolone + n) % MOD;
    zbirKolone = (zbirKolone + elementKolone) % MOD;
  }
  cout << zbirKolone << endl;
```

```
    return 0;
}
```

Formula za zbir aritmetičkog niza

Kvadrat sadrži sledeće brojeve:

$$\begin{array}{ccccccc}
0 & 1 & \dots & i & \dots & n-1 \\
n & n+1 & \dots & n+i & \dots & n+n-1 \\
\dots & \dots & \dots & \dots & \dots & \dots \\
i \cdot n & i \cdot n+1 & \dots & i \cdot n+i & \dots & i \cdot n+n-1 \\
\dots & \dots & \dots & \dots & \dots & \dots \\
(n-1) \cdot n & (n-1) \cdot n+1 & \dots & (n-1) \cdot n+i & \dots & (n-1) \cdot n+n-1
\end{array}$$

Elementi i-te vrste čine aritmetički niz. Prvi član je $a_1 = i \cdot n$, a razlika između susedna dva člana $d = 1$. Primenom formule za zbir n elemenata aritmetičkog niza $S_n = a_1 + ... + a_n = n \cdot a_1 + d \cdot \frac{n(n-1)}{2}$ dobijamo da je zbir elemenata u i-toj vrsti jednak

$$i \cdot n^2 + \frac{n(n-1)}{2}.$$

Slično, elementi i-te koloni takođe čine aritmetički niz u kome je prvi član $a_0 = i$, a razlika između susedna dva člana jednaka je n. Zato je njen zbir jednak

$$n \cdot i + n \cdot \frac{n(n-1)}{2}.$$

Pošto je bar jedan od brojeva n ili $n-1$ paran, rezultat je uvek ceo broj i moguće je sve operacije izvoditi nad celim brojevima (uključujući i celobrojno deljenje sa 2). Implementaciju je moguće napraviti direktno na osnovu ovih formula, vodeći računa o tome da se sve operacije izvode po modulu 10^9, da ne bi došlo do prekoračenja i da bi se odredilo poslednjih 9 cifara dužih brojeva.

Složenost ovog programa je, jasno, $O(1)$. U implementaciji bi mogao biti smanjen broj primene izračunavanja ostatka (ovo je skupa operacija), međutim, pošto se ona primenjuje samo konstantan broj puta, time se ne bi dobilo značajno ubrzanje, a kod bi postao teže razumljiv i lakše bi moglo doći do greške.

```
typedef long long ll;

const long long MOD = 1e9;

ll pm(ll a, ll b) {
  return ((a % MOD) * (b % MOD)) % MOD;
}

ll pm(ll a, ll b, ll c) {
  return pm(pm(a, b), c);
}

ll zm(ll a, ll b) {
  return (a % MOD + b % MOD) % MOD;
}

int main() {
  ll n, i;
  cin >> n >> i;
  // i*n*n + n*(n-1)/2
  ll zbirVrste =
   zm(pm(i, n, n), n%2 == 0 ? pm(n/2, n-1) : pm(n, (n-1)/2));
  cout << zbirVrste << endl;
  // n*i + n*n*(n-1)/2
  ll zbirKolone =
   zm(pm(n, i), n%2 == 0 ? pm(n/2, n, n-1) : pm(n, n, (n-1)/2));
  cout << zbirKolone << endl;
  return 0;
}
```

Moguće je i definisati funkciju za izračunavanje zbira elemenata aritmetičkog niza i dva puta je upotrebiti. I u ovom rešenju je potrebno sve aritmetičke operacije izvoditi po modulu 10^9.

```
ll sumaAritmetickog(ll a, ll d, ll n) {
  // n*a + d*n*(n-1)/2
```

```
  return zm(pm(n, a),
            n%2 == 0 ? pm(d, n/2, n-1) : pm(d, n, (n-1)/2));
}

int main() {
  ll n, i;
  cin >> n >> i;
  cout << sumaAritmetickog(i*n, 1, n) << endl;
  cout << sumaAritmetickog(i, n, n) << endl;
  return 0;
}
```

Zadatak: Rastavljanja na zbir uzastopnih

Napiši program koji određuje na koliko se načina dati prirodni broj n može predstaviti kao zbir dva ili više uzastopnih prirodnih brojeva (većih ili jednakih 1).

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 10^9$).

Opis izlaza

Na standardni izlaz ispisati traženi broj načina.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
15	3	Važi: $15 = 1 + 2 + 3 + 4 + 5 = 4 + 5 + 6 = 7 + 8$.

Rešenje

Isprobavanje svih mogućnosti za prvi član, pa za dužinu

Prvi pokušaj može biti rešavanje problema grubom silom, tj. isprobavanje svih mogućih prvih članova zbira. Najmanji mogući prvi član je $a_0 = 1$. Pošto zbir mora da bude bar dvočlan, najveći mogući prvi član je onaj broj a_0 takav da je $a_0 + (a_0 + 1) \leq n$. Kada smo odredili prvi član, određujemo koliko sabiraka treba uzeti da bi se dobio zbir n. Krećemo od dvočlanog niza i zatim dodajemo jedan po jedan naredni sabirak sve dok zbir ne dostigne ili ne prestigne zbir n. Brojač uvećavamo ako je nakon petlje zbir jednak vrednosti n (tada je uspešno nađeno jedno rešenje).

```
int brojNacina(int n) {
  int broj = 0;
  for (int a0 = 1; a0 + (a0+1) <= n; a0++) {
    int zbir = a0 + (a0+1);
    for (int ai = a0 + 2; zbir < n; ai++)
      zbir += ai;
    if (zbir == n)
      broj++;
  }
  return broj;
}
```

Spoljašnja petlja se izvršava oko $\frac{n}{2}$ puta. Broj izvršavanja unutrašnje petlje je teže proceniti. Pitamo se koji je broj sabiraka m potreban, tako da je $a_0 + (a_0 + 1) + \ldots + (a_0 + (m-1)) \geq n$. Ako primenimo formulu za zbir aritmetičkog niza, vidimo da je taj zbir jednak $m \cdot a_0 + \frac{m(m-1)}{2}$. Veoma gruba procena kada je a_0 malo daje nam procenu za m oko $\sqrt{2n}$. Mada, čim a_0 krene da raste, ovaj broj krene da opada. Veoma grubo, složenost možemo ograničiti odozgo kao $O(n\sqrt{n})$.

Isprobavanje svih mogućnosti za dužinu, pa za prvi član

Redosled petlji može biti drugačiji. Spoljašnjom petljom možemo određivati broj sabiraka m, a unutrašnjom isprobavati vrednosti početnog sabirka. Krećemo od dva sabirka. Najveći mogući broj sabiraka nastupa kada je $a_0 = 1$, i pošto je $a_0 + (a_0 + 1) + \ldots + (a_0 + (m-1)) = m \cdot a_0 + \frac{m(m-1)}{2}$, da bi zbir mogao da eventualno bude n mora da važi da je $m + \frac{m(m-1)}{2} \leq n$.

```
int brojNacina(int n) {
  int broj = 0;
  for (int m = 2; m + m*(m-1)/2 <= n; m++) {
    int a0 = 1;
    int zbir = a0*m + m*(m-1)/2;
    while (zbir < n) {
      a0++;
      zbir = a0*m + m*(m-1)/2;
    }
```

```
    if (zbir == n)
      broj++;
  }
  return broj;
}
```

Složenost je identična kao u prethodnom pristupu i može se grubo proceniti sa $O(n\sqrt{n})$.

Isprobavanje svih mogućnosti za dužinu i izračunavanje prvog člana

Ključna optimizacija nastupa kada uvidimo da nam unutrašnja petlja nije potrebna. Naime, nema potrebe isprobavati različite vrednosti a_0, već se a_0 može izračunati na osnovu m i n. Ako je $m \cdot a_0 + \frac{m(m-1)}{2} = n$, tada je $a_0 = \frac{n - \frac{m(m-1)}{2}}{m}$. Zbir sa m sabiraka postoji ako i samo ako je ovo ceo broj (što se može proveriti ispitivanjem ostatka pri deljenju brojeva $n - \frac{m(m-1)}{2}$ i m). Uslov $m + \frac{m(m-1)}{2} \leq n$ garantuje da je $a_0 \geq 1$.

```
int brojNacina(int n) {
  int broj = 0;
  for (int m = 2; m + m*(m-1)/2 <= n; m++)
    if ((n - m*(m-1)/2) % m == 0)
      broj++;
  return broj;
}
```

Složenost jedine petlje, pa i celog programa se može grubo oceniti sa $O(\sqrt{n})$ (u njenom telu se provera postojanja broja a_0 vrši u složenosti $O(1)$).

Primetimo da se nakon primene formule za zbir aritmetičkog niza zadatak sveo na pronalaženje celobrojnih rešenja jedne jednačine sa dve nepoznate (a_0 i m)

$$m \cdot a_0 + \frac{m(m-1)}{2} = n.$$

Pošto broj celobrojnih rešenja ne možemo unapred odrediti, koristimo iterativni postupak da proverimo razne kandidate.

Do efikasnog rešenja ovog zadatka smo došli tako što smo umesto provere raznih parova vrednosti, uvideli da se nakon fiksiranja jedne vrednosti ona druga može eksplicitno izračunati. Redosled provera je veoma važan, jer je jednačina linearna po a_0, a kvadratna po m, tako da je za fiksirano m, a_0 prilično jednostavno izračunati, dok za fiksirano a_0 nije jednostavno izračunati m. Dakle, u ovom zadatku vidimo odličan spoj računarskog i matematičkog pristupa: pošto jednačinu sa dve promenljive ne možemo da rešimo matematički, koristimo iterativni postupak i isprobavamo razne vrednosti m. Nakon toga dobija se jednačina koja se može rešiti matematički i tada izbegavamo korišćenje iterativnog postupka koji bi isprobavao razne vrednosti a_0.

3.A.2 *Inkrementalnost*

Zadatak: Pangrami

Pangrami su reči koje sadrže bar jedno pojavljivanje svakog slova abecede ili azbuke (slova se mogu pojavljivati i više puta). Čuveni pangram u engleskom jeziku je niska `"the quick brown fox jumps over a lazy dog"`. Napiši program koji proverava da li se u datom tekstu nalazi neki podtekst (niz uzastopnih karaktera) dužine k koji je pangram.

Opis ulaza

Prva linija standardnog ulaza sadrži nisku sastavljenu samo od malih slova engleske abecede, dužine najviše 10^5 karaktera. Naredni red sadrži prirodan broj k ($1 \leq k \leq 10^5$).

Opis izlaza

Na standardni izlaz ispisati `da` ako u unetom tekstu postoji pangram dužine k, odnosno `ne` u suprotnom.

Primer 1

Ulaz	*Izlaz*
`xxxabcdefghijklmnopqrstuvwxyzxxx` `26`	`da`

Primer 2

Ulaz	*Izlaz*
`xxxabcdefghijklmxxxnopqrstuvwxyzxxx` `28`	`ne`

Primer 3

Ulaz	*Izlaz*
`xxxabcdefghijklmxxxnopqrstuvwxyzxxx` `29`	`da`

Rešenje

Gruba sila

Rešenje grubom silom podrazumeva da se za svaku podnisku dužine k proveri da li sadrži sve karaktere abecede. Za svaku podnisku i svako slovo abecede možemo linearnom pretragom utvrditi da li podniska sadrži to slovo (linearna pretraga može biti realizovana bilo bibliotečkom funkcijom, bilo ručno implementirana). Ako su sva slova pronađena, podniska je pangram i tada možemo prekinuti dalju analizu, dok u suprotnom prelazimo na analizu naredne podniske.

Provera da li je podniska pangram zahteva 26 linearnih pretraga dužine k. U principu, možemo smatrati da je složenost $O(k)$, iako je konstanta 26 prilično velika. Podniski ima ukupno $n-k$, pa u najgorem slučaju (kada je k oko pola dužine niza), dobijamo algoritam kvadratne složenosti $O(n^2)$.

```
// provera da li dati tekst sadrzi pangram duzine k
bool sadrzi_pangram(const string& tekst, int k) {
  // proveravamo sve podniske duzine k
  for (int i = 0; i + k < tekst.length(); i++) {
    // da li je podniska pangram
    bool pangram = true;
    // proveravamo da li podniska [i, i+k) sadrzi sva slova
    for (char c = 'a';  c <= 'z'; c++) {
      bool sadrzi = false;
      for (int j = i; j < i + k; j++)
        if (tekst[j] == c) {
          sadrzi = true;
          break;
        }
      // ako nema nekog slova, podniska nije pangram
      if (!sadrzi) {
        pangram = false;
        break;
```

```
      }
    }
    // sva slova su pronadjena, podniska jeste pangram
    if (pangram)
      return true;
  }
  return false;
}
```

Inkrementalno rešenje

Efikasnije rešenje možemo dobiti zahvaljujući principu inkrementalnosti. Da bismo proverili da li je neka reč pangram, dovoljno je da znamo skup karaktera koji se u njoj javljaju i da proverimo da li taj skup sadrži 26 elemenata. Prilikom prelaska sa jedne na drugu podnisku, većina karaktera se poklapa. Razlikuju se samo prvi karakter prve podniske i poslednji karakter druge podniske. Zato prilikom prelaska sa podniske na podnisku treba iz skupa ukloniti prvi, a dodati poslednji karakter. Međutim, pošto se karakteri javljaju više puta, ovo može biti pogrešno, pa zapravo umesto skupova treba razmatrati multiskupove. Najjednostavniji način je da uvedemo preslikavanje koje svakom karakteru dodeljuje njegov broj pojavljivanja u podniski (ono može biti realizovano pomoću niza brojača ili, jednostavnije, preko mape tj. rečnika). Prilikom prelaska na narednu podnisku umanjujemo brojač prvog karaktera (i ako dostigne nulu, uklanjamo ga iz mape) i uvećavamo brojač poslednjeg karaktera. Nakon toga, proveravamo da li je broj karaktera u mapi jednak 26 i ako jeste, tekuća podniska predstavlja pangram.

Pošto se u svakom trenutku održavaju podaci o nekom segmentu originalnog niza kraktera dužine k i taj segment se polako pomera sleva nadesno, nekada se kaže da se primenjuje tehnika *pokretnog prozora* i taj prozor se ažurira inkrementalno.

U petlji se analizira svaki karakter niske. Ako pretpostavimo da su operacije sa mapom tj. rečnikom konstantne složenosti (što je prilično opravdano, jer postoji samo 26 različitih ključeva), složenost celog algoritma se može oceniti sa $O(n)$, gde je n dužina niske koja se proverava.

```
// provera da li dati tekst sadrzi pangram duzine k
bool sadrzi_pangram(const string& tekst, int k) {
  // broj pojavljivanja karaktera u trenutnoj podniski duzine k
```

```
  map<char, int> karakteri;
  // prolazimo sve karaktere u tekstu
  for (int i = 0; i < tekst.length(); i++) {
    // prelazimo na narednu podnisku,
    // inkrementalno azurirajuci njen broj karaktera
    if (i >= k)
      // izbacujemo karakter na poziciji i-k
      if (--karakteri[tekst[i-k]] == 0)
        karakteri.erase(tekst[i-k]);
    // dodajemo karakter na poziciji i
    karakteri[tekst[i]]++;
    // ako se svi karakteri javljaju, pangram je pronadjen
    if (karakteri.size() == 26)
      return true;
  }
  return false;
}
```

3.A.3 *Odsecanje u pretrazi*

Zadatak: Najduža serija uzastopnih nula

Neki blokovi memorije su zauzeti, a neki slobodni. Da bi se u memoriju mogao smestiti dugačak niz podataka, potrebno je pronaći što duži niz uzastopnih slobodnih blokova.

Opis ulaza

Sa standardnog ulaza se unosi prirodan broj N ($5 \leq N \leq 50000$), a zatim i N brojeva 1 (što označava da je blok zauzet) ili 0 (što označava da je blok slobodan).

Opis izlaza

Na standardni izlaz ispisati jedan prirodan broj koji predstavlja traženu dužinu najduže serije uzastopnih slobodnih blokova.

Primer

Ulaz	*Izlaz*
8 0 0 1 0 0 0 1 1	3

Rešenje

Gruba sila

Zadatak zahteva da se odredi dužina najduže serije uzastopnih nula.

U naivnom pristupu rešavanju ovog problema ulazni podaci se smeštaju i čuvaju u nizu. To nije neophodno, ali ćemo najpre prikazati i takva rešenja, da bismo sistematično ilustrovali neke tehnike postupne optimizacije koda.

Jedno veoma naivno rešenje je da analiziramo sve moguće segmente niza određene svim mogućim vrednostima promenljivih $0 \leq i \leq j < n$. Njih možemo nabrojati ugnežđenim petljama. Za svaki segment možemo primenom linearne pretrage proveriti da li sadrži samo nule i ako sadrži, ažurirati maksimum u skladu sa tim.

```
// izracunava duzinu najduze serije uzastopnih nula
int najduzaSerijaNula(const vector<int>& a) {
  // broj podataka
  int N = a.size();

  // duzina najduze serije uzastopnih nula
  int maxDuzina = 0;
  // analiziramo sve segmente a[i, j]
  for (int i = 0; i < N; i++) {
    for (int j = i; j < N; j++) {
      // proveravamo da li su u segmentu a[i, j] samo nule
      bool samo_nule = true;
      for (int k = i; k <= j; k++)
        if (a[k] != 0) {
          samo_nule = false;
          break;
        }
      // ako jesu, azuriramo maksimum u odnosu na duzinu
      // segmenta [i, j]
      if (samo_nule)
        maxDuzina = max(maxDuzina, j - i + 1);
    }
  }
```

```
  return maxDuzina;
}
```

Pošto se podaci smeštaju u pomoćni niz, memorijska složenost je $O(n)$. Ovo rešenje je izrazito neefikasno, jer mu je vremenska složenost čak kubna tj. $O(n^3)$. Sa druge strane korektnost ovog rešenja je zaista trivijalno obrazložiti (ono se potpuno direktno izvodi iz same formulacije zadatka).

Najduža serija za svaki levi kraj

Malo bolji pristup je da za svaku poziciju i odredimo najdužu seriju uzastopnih nula koja počinje na toj poziciji. To se može uraditi tako što se serija koji počinje na poziciji i proširuje od pozicije i nadesno, sve dok se u njoj nalaze samo nule. Važna ušteda na ovom mestu je to što ako znamo da su u segmentu $[i, j]$ sve nule i da se nula nalazi i na poziciji $j + 1$, onda znamo i da su sve nule i u segmentu $[i, j + 1]$ (kažemo da proveru vršimo inkrementalno).

Kada se naiđe na broj različit od nule (ili na kraj niza), nađena je najduža serija uzastopnih nula koja počinje na poziciji i, jer će sva produžavanja te serije nadesno, ako ih ima, sadržati i ovu vrednost različitu od nule. Dakle, na ovom mestu vršimo odsecanje pretrage preskačući mnoge segmente niza za koje se unapred zna da ne mogu zadovoljiti nametnuti uslov. Takođe, poređenje sa maksimalnom dužinom vršimo tek kada maksimalno proširimo tekući segment, jer unapred znamo da su svi podsegmenti tog maksimalno proširenog segmenta kraći od njega (i ovde zapravo vršimo određeno odsecanje).

```
// izracunava duzinu najduze serije uzastopnih nula
int najduzaSerijaNula(const vector<int>& a) {
  // broj podataka
  int N = a.size();

  // duzina najduze serije uzastopnih nula
  int maxDuzina = 0;
  // za svaku poziciju i odredjujemo duzinu najduze serije
  // uzastopnih nula koja pocinje na poziciji i
  for (int i = 0; i < N; i++) {
    int duzina = 0;
    for (int j = i; j < N && a[j] == 0; j++)
```

```
      duzina++;
    // ako je duzina serije koja pocinje na poziciji i veca od
    // maksimalne do tada vidjene duzine, azuriramo maksimalnu
    // duzinu
    if (duzina > maxDuzina)
      maxDuzina = duzina;
  }

  return maxDuzina;
}
```

Ovo rešenje je složenosti $O(n^2)$, što je bolje od prvog, međutim i dalje suboptimalno. Pošto se podaci smeštaju u pomoćni niz, memorijska složenost je $O(n)$.

Dalja odsecanja nepotrebnih izračunavanja

Program se može dodatno značajno ubrzati daljim odsecanjem. Jednom kada odredimo da je najduži segment koji sadrži uzastopne nule i počinje na poziciji i segment $[i, j]$, vreme značajno možemo uštedeti tako što primetimo da ni jedan segment koji sadrži uzastopne nule i počinje na pozicijama nakon i, a zaključno sa j ne može biti duži od segmenta koji počinje sa i (jer ako pozicija j nije poslednja u nizu, na poziciji iza nje se ne nalazi nula i ti segmenti se ne mogu proširiti dalje od pozicije j). Zato je nakon širenja segmenta koji počinje na poziciji i nadesno i određivanja serije nula koja počinje na poziciji i moguće direktno preći na izračunavanje najdužeg segmenta uzastopnih nula koji počinje na poziciji $j + 1$ (ako takva postoji). Ovo zapravo odgovara tome da ceo niz izdelimo na segmente uzastopnih nula koji se nadovezuju (presečeni segmentima uzastopnih jedinica). Složenost takvog pristupa je $O(n)$, jer se granice segmenata samo uvećavaju i nikada ne smanjuju.

Ovaj algoritam je zapravo i vrlo intuitivan i verovatno je prvi algoritam koji bi programer sa malo iskustva implementirao: krećemo od početka, pronalazimo seriju uzastopnih nula koji počinje na početku, nakon toga tražimo seriju uzastopnih nula nakon te prve serije, zatim seriju uzastopnih nula nakon te druge i tako dalje. Dakle, ceo niz delimo na manje segmente koji sadrže jednake elemente i nadovezuju se jedan iza drugog, pri čemu je podela takva da je svaki od tih segmenata optimalan u smislu da ga nije moguće produžiti (ni na levo, ni na desno).

Možemo primetiti da nam tokom implementacije nije više neophodno da pamtimo sve rezultate u nizu istovremeno. U jednoj petlji ćemo čitati redom elemente niza i

u svakom trenutku održavati dužinu tekuće i dužinu najduže do tada obrađene serije (segmenta) uzastopnih nula. Pošto na početku nismo videli još ni jedan element niza, obe promenljive inicijalizujemo na nulu. Ako učitamo nulu, tada se tekuća seriju uzastopnih nula produžava i njenu dužinu uvećavamo za jedan. Ako nije nula, tada se tekuća serija prekida i započinje nova, koja ima dužinu 0 (jer je za sada prazna i ne sadrži ni jednu nulu).

Nakon završetka čitanja svake serije uzastopnih nula potrebno je ažurirati dužinu najduže serije. To se dešava ili kada se naiđe na podatak različit od nule ili nakon petlje, kada je poslednja eventualna serija nula završena. Treba biti veoma obazriv da se ne zaboravi na poslednju seriju, tj. da se ne zaboravi poređenje tekuće i najduže serije nakon završetka petlje. Alternativa je da se maksimum ažurira prilikom svakog uvećanja dužine tekuće serije uzastopnih nula (na taj način se zapravo određuje dužina najduže serije uzastopnih nula koja se završava na tekućoj poziciji).

```
// broj blokova
int N;
cin >> N;
// duzina tekuce serije uzastopnih nula
int duzinaTekuce = 0;
// duzina najduze do sada vidjene serije uzastopnih nula
int duzinaNajduze = 0;
// ucitavamo podatke
for (int i = 0; i < N; i++) {
  // naredni broj
  int x;
  cin >> x;
  if (x == 0) {
    // nula produzava tekucu seriju
    duzinaTekuce++;
  } else {
    // ako je upravo prekinuta serija duza od najduze,
    // azuriramo duzinu najduze
    if (duzinaTekuce > duzinaNajduze)
      duzinaNajduze = duzinaTekuce;
    // broj razlicit od nule prekida seriju i zapocinje se nova
    // u kojoj jos ne postoji nijedna nula
```

```
    duzinaTekuce = 0;
  }
}
// vrsimo proveru i za poslednju seriju
if (duzinaTekuce > duzinaNajduze)
  duzinaNajduze = duzinaTekuce;

// ispisujemo konacan rezultat
cout << duzinaNajduze << endl;
```

Složenost ovog algoritma je $O(n)$. Memorijska složenost je $O(1)$.

3.A.4 *Zbirovi prefiksa i razlike susednih elemenata niza*

Zadatak: Broj rastućih segmenata

Dat je niz a celih brojeva, dužine n. Napisati program kojim se određuje na koliko načina možemo izabrati rastuće segmente u nizu. Rastući segment čine uzastopni elementi niza $a_p < a_{p+1} < ... < a_q$, za $0 \leq p < q < n$.

Opis ulaza

Prva linija standardnog ulaza sadrži prirodan broj n ($2 \leq n \leq 10000$), broj elemenata niza. U svakoj od n narednih linija standardnog ulaza, nalazi po jedan član niza.

Opis izlaza

Na standardnom izlazu prikazati broj rastućih segmenata datog niza.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
5 1 3 4 -2 10	4	To su segmenti $[1, 3]$, $[1, 3, 4]$, $[3, 4]$, $[-2, 10]$.

Rešenje

Gruba sila

Zadatak možemo rešiti analizirajući sve segmente datog niza a i za svaki segment $a_i, a_{i+1}, ..., a_j$ gde je $0 \leq i < j < n$ proveriti da li je rastući i u skladu sa tim uvećati brojač rastućih segmenata. Pošto segmenata ima $O(n^2)$ i svakom se provera monotonosti vrši u linearnoj složenosti, ovo rešenje je složenosti $O(n^3)$ (naravno, nisu svi segmenti iste dužine i precizna analiza bi trebalo da u obzir uzme dužine svih segmenata, međutim, i na taj način bi se izračunalo da je algoritam kubne složenosti). Elementi su smešteni u niz, pa je memorijska složenost $O(n)$.

Broj rastućih segmenata za svaki levi kraj

Efikasnije rešenje se može dobiti ako se provera monotonosti vrši inkrementalno (da bi se proverilo da je segment $[i, j]$ rastući dovoljno je da je $a_j > a_{j-1}$ i da je segment $[i, j-1]$ rastući ili jednočlan).

Vreme izvršavanja možemo unaprediti i odsecanjem. Primetimo da ako segment koji čine elementi na pozicijama $[i, j]$ nije rastući, onda nisu rastući ni segmenti $[i, j']$ za $j \leq j' < n$, pa za te segmente ne treba vršiti proveru, što znači da je brojanje rastućih segmenata koji počinju na poziciji i moguće prekinuti čim se pronađe neki segment koji počinje na toj poziciji i nije rastući.

Dakle, za svaku poziciju i u nizu (za svako $0 \leq i < n-1$) analiziramo jedan po jedan segment $[i, j]$ koji na toj poziciji počinje sve dok su ti segmenti rastući i za svaki rastući segment uvećavamo brojač rastućih segmenata za 1. Čim naiđemo na segment koji nije rastući (tj. na element koji je manji od prethodnog), prelazimo na narednu poziciju i.

Primer 3.A.1. *Na primer u nizu* $[1, 3, 4, 5, 2, 6]$ *analiziramo segmente koji počinju prvim elementom niza tj. elementom* a_0 *sve dok su segmenti rastući, pri tome prebrojimo rastuće segmente* $[1, 3]$*;* $[1, 3, 4]$ *i* $[1, 3, 4, 5]$*. Slično polazeći od drugog elementa niza prebrojimo rastuće segmente* $[3, 4]$ *i* $[3, 4, 5]$*. Nastavljajući isti postupak za ostale elemente niza prebrojimo i rastući segment* $[4, 5]$*, a zatim i* $[2, 6]$.

Složenost ovog algoritma je $O(n^2)$ i njime se najefikasnije moguće eksplicitno nabrajaju svi rastući segmenti. Memorijska složenost je $O(n)$.

```
long long brojRastucihSegmenata(const vector<int>& a) {
  // velicina niza
  int n = a.size();
  // ukupan broj rastucih serija
  long long brojRastucih = 0;
  // za svaku poziciju u nizu
  for (int i = 0; i < n - 1; i++) {
    // pronalazimo sve rastuce serije koje pocinju na toj
    // poziciji
    // proveru da li je serija odredjena pozicijama [i, j]
    // rastuca odredjujemo inkrementalno
    // postupak prekidamo cim se naidje na seriju koja nije
    // rastuca
    for (int j = i + 1; j < n; j++)
      if (a[j] > a[j-1])
        brojRastucih++;
      else
        break;
  }
  return brojRastucih;
}
```

Maksimalni rastući segmenti

U prethodno rešenju se nabrajaju svi rastući segmenti, međutim, u zadatku je potrebno izračunati samo njihov broj (a ne i nabrojati ih eksplicitno), a to se može uraditi i efikasnije.

Primetimo da u prethodnom primeru analizirajući rastući segment koji počinje od elementa a_0 prolazimo i po rastućim segmentima niza koji počinju sa a_1 i a_2. Ako je segment $[a_i, a_{i+1}, ..., a_j]$ rastući, onda unapred znamo da su rastući i segmenti $[a_i, a_{i+1}]$, $[a_i, a_{i+1}, a_{i+2}]$, ..., $[a_i, a_{i+1}, \dots, a_j]$, zatim $[a_{i+1}, a_{i+2}]$, ..., $[a_{i+1}, a_{i+2}, \dots, a_j]$, pa sve do $[a_{j-1}, a_j]$, a da segmenti $[a_i, a_{i+1}, \dots, a_{j+1}]$, ..., $[a_i, a_{i+1}, \dots, a_{n-1}]$, zatim $[a_{i+1}, a_{i+2}, \dots, a_{j+1}]$, ..., $[a_{i+1}, a_{i+2}, \dots, a_{n-1}]$ itd., zaključno sa $[a_j, \dots, a_{n-1}]$ nisu rastući. Dakle, za svaku poziciju iz intervala $[i, j]$ tačno znamo sve rastuće segmente koji na njoj počinju.

Rastući segment $[a_i, a_{i+1}, \dots, a_{i+k-1}]$ dužine k u sebi sadrži:

- $k-1$ rastućih segmenata koji počinju sa a_i
- $k-2$ rastućih segmenata koji počinju sa a_{i-1}
- ...
- 1 rastući segmenat koji počinje sa a_{i+k-2}

ukupno $(k-1)+(k-2)+...+1$ rastućih segmenata što iznosi $\frac{k\cdot(k-1)}{2}$.

Do istog zaključka možemo doći i na sledeći način: svaki podsegment $a_p, a_{p+1}, ..., a_q$ gde je $i \leq p < q \leq i+k-1$ rastućeg segmenta $a_i, a_{i+1}, ..., a_{i+k-1}$ je rastući, početak p i kraj q podsegmenta možemo izabrati na $\frac{k\cdot(k-1)}{2}$ načina.

Dakle, potrebno je pronaći dužine maksimalnih rastućih segmenata (onih koji se ne mogu produžiti dodatnim elementom tako da i dalje ostaju rastući), a zatim broj njihovih rastućih podsegmenata umesto iteracijom, izračunati formulom. Naime, nakon nalaženja nekog maksimalnog rastućeg segmenta $[i, j]$, možemo neposretno da izračunamo broj rastućih segmenata koji počinju na svim pozicijama između i i j.

Prema tome, u petlji analiziramo niz član po član počev od prvog člana ($i = 0$) i određujemo dužinu t tekućeg rastućeg segmenta (ako je $a_i < a_{i+1}$ uvećavamo t za 1). Kada dođemo do kraja tekućeg rastućeg segmenta, uvećavamo ukupan broj rastućih segmenata br za $\frac{t\cdot(t-1)}{2}$ i počinjemo analizu sledećeg rastućeg segmenta ($t = 1$). Do kraja maksimalnog rastućeg segmenta se može stići na dva načina: ili kada je $a_i \geq a_{i+1}$ ili kada se dođe do kraja niza. Napomenimo da za taj poslednji rastući segment uvećavamo ukupan broj rastućih segmenata izvan petlje (česta greška je da se to zaboravi).

U svakom trenutku je dovoljno porediti samo susedna člana niza, tako da nije neophodno ceo niz pamtiti u memoriji, već samo dva susedna člana niza (prethodni i tekući).

Na prethodno opisan način dobijamo rešenje jednim prolaskom po nizu i vremenska složenost mu je $O(n)$. Memorijska složenost je $O(1)$.

```
int n;
cin >> n;
int prethodni;
cin >> prethodni;
// ukupan broj rastucih serija
```

```
long long brojRastucih = 0;
// duzina tekuce rastuce serije
long long duzinaTekuceRastuce = 1;
for(int i = 1; i < n; i++) {
  int tekuci;
  cin >> tekuci;
  if (tekuci > prethodni)
    // tekuci element produzava tekucu rastucu seriju
    duzinaTekuceRastuce++;
  else {
    // tekuci element zapocinje novu rastucu seriju
    // dodajemo sve rastuce serije koje su podserije rastuce
    // serije koja se zavrsila sa prethodnim elementom
    brojRastucih += (duzinaTekuceRastuce - 1) *
                    duzinaTekuceRastuce / 2;
    duzinaTekuceRastuce = 1;
  }
  prethodni = tekuci;
}
// dodajemo sve rastuce serije koje su podserije poslednje
// rastuce serije
brojRastucih += (duzinaTekuceRastuce - 1) *
                duzinaTekuceRastuce / 2;
cout << brojRastucih << endl;
```

Zadatak: Zbirovi segmenata

Čuvena veb-platforma želi da napravi sistem za analizu posete svom veb-sajtu. Sakupljen je broj poseta tokom svakog minuta u prethodnih nekoliko godina (najviše 10). Napisati program koji omogućava korisniku da za zadate vremenske raspone (određene minutima od početka brojanja) izračuna broj poseta tokom tih vremenskih raspona.

Opis ulaza

Sa standardnog ulaza se unosi broj minuta n ($1 \leq n \leq 60 \cdot 24 \cdot 365 \cdot 10$), a zatim u narednom redu n celih brojeva između 0 i 100, razdvojenih sa po jednim razmakom, koji predstavljaju broj posetilaca tokom svakog od tih n minuta. Nakon

toga se unosi broj upita m ($1 \leq m \leq 100000$) i u narednih m redova se unose vremenski periodi određeni rednim brojem početnog minuta a i krajnjeg minuta b ($0 \leq a \leq b < n$).

Opis izlaza

Na standardni izlaz ispisati m celih brojeva koji predstavljaju ukupan broj posetilaca u svakom od tih m perioda.

Primer

Ulaz	*Izlaz*
5	15
1 2 3 4 5	9
3	3
0 4	
1 3	
2 2	

Objašnjenje

Učitan je broj posetilaca tokom 5 minuta.

- Prvo se traži broj posetilaca između minuta 0 i minuta 4 i to je $1 + 2 + 3 + 4 + 5 = 15$.
- Zatim se traži broj posetilaca između minuta 1 i minuta 3 i to je $2+3+4 = 9$.
- Na kraju se traži broj posetilaca u minutu 2 i to je 3.

Rešenje

Direktno rešenje

Direktno rešenje podrazumeva da sve brojeve učitamo u niz, a zatim da za svaki upit iznova računamo zbir odgovarajućeg segmenta niza.

```
// ucitavamo niz
int n;
cin >> n;
vector<int> brojevi(n);
for (int i = 0; i < n; i++)
  cin >> brojevi[i];

// ucitavamo granice segmenata i izracunavamo i ispisujemo
```

```
// njihove zbirove
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int a, b;
  cin >> a >> b;
  int zbir = 0;
  for (int j = a; j <= b; j++)
    zbir += brojevi[j];
  cout << zbir << endl;
}
```

Složenost ovakvog pristupa je $O(nm)$. Pošto se faza učitavanja i ispisa podataka prepliću, učitavanje bi i ispis podataka bi trebalo dodatno ubrzati, no to ne može popraviti neefikasnost ovog naivnog algoritma.

Zbirovi prefiksa

Jednostavno efikasno rešenje je zasnovano na narednoj ideji: umesto čuvanja elemenata niza, možemo čuvati niz zbirova prefiksa niza. Zbir svakog segmenta $[l, d]$ možemo razložiti na razliku zbira prefiksa do elementa d i prefiksa do elementa $l-1$. Ako koristimo oznaku $\sum_{k=m}^{n} a_k$ koja označava zbir $a_m + a_{m+1} + ... + a_n$, možemo zapisati da je

$$\sum_{k=l}^{d} a_k = \sum_{k=0}^{d} a_k - \sum_{k=0}^{l-1} a_k.$$

Zbirovi svih prefiksa se mogu izračunati i smestiti u dodatni (a ako je ušteda memorije bitna, onda čak i u originalni) niz. Dakle, tokom učitavanja elemenata možemo formirati niz zbirova prefiksa (računaćemo ih inkrementalno, jer se svaki naredni zbir prefiksa dobija uvećavanjem prethodnog zbira prefiksa za tekući element niza). Neka z_i označava zbir elemenata prefiksa određenog pozicijama iz intervala $[0, i)$. Formiramo, dakle, niz $z_i = \sum_{k=0}^{i-1} a_k$ (pri čemu je $z_0 = 0$, zbir praznog prefiksa). Tada zbir elemenata u segmentu pozicija $[l, d]$ izračunavamo kao $z_{d+1} - z_l$.

```
// ucitavamo brojeve i izracunavamo zbirove prefiksa
int n;
cin >> n;
vector<int> zbirovi_prefiksa(n+1);
zbirovi_prefiksa[0] = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  zbirovi_prefiksa[i+1] = zbirovi_prefiksa[i] + x;
}

// ucitavamo granice segmenata i izracunavamo i ispisujemo
// njihove zbirove
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int a, b;
  cin >> a >> b;
  cout << zbirovi_prefiksa[b+1] - zbirovi_prefiksa[a] << '\n';
}
```

Za učitavanje brojeva i formiranje niza zbirova prefiksa potrebno nam je $O(n)$ koraka. Nakon ovakvog pretprocesiranja, zbir svakog segmenta se može izračunati u vremenu $O(1)$, pa je ukupna složenost $O(n + m)$.

Pošto se u ovom zadatku prepliću faza učitavanja i faza ispisa podataka na standardni ulaz i izlaz, potrebno je obratiti pažnju na neefikasnost koja nastaje zbog čestog pražnjenja izlaznog bafera. Potrebno je razvezati `cin` i `cout` korišćenjem `cin.tie(0)` i umesto pomoću `endl` u novi red prelaziti pomoću `\n`. Naravno, ovo ima smisla samo u slučaju automatske primene programa na velike ulaze i izlaze - ovim izmenama program prestaje da radi korektno u interaktivnom režimu.

Zadatak: Broj segmenata čiji je zbir deljiv sa k

Dat je niz a prirodnih brojeva dužine n i prirodan broj k. Napisati program koji određuje broj segmenta niza a (nepraznih podnizova uzastopnih elemenata) čiji je zbir deljiv sa k.

Opis ulaza

U prvoj liniji standardnog ulaza nalazi se prirodan broj k ($k \leq 10^5$). Druga linija standardnog ulaza sadrži prirodan broj n ($n \leq 10^5$). U sledećoj liniji se nalazi n prirodnih brojeva (ti brojevi predstavljaju redom elemente niza a), razdvojenih sa po jednim razmakom.

Opis izlaza

Na standarnom izlazu prikazati koliko postoji segmenata niza a čiji je zbir deljiv sa k.

Primer

Ulaz	*Izlaz*
3 5 1 8 2 3 4	4

Objašnjenje

To su segmenti $(1, 8)$, (3), $(2, 3, 4)$ i $(1, 8, 2, 3, 4)$.

Rešenje

Gruba sila

Direktan način da se zadatak reši je da se ugnežđenim petljama nabroje svi segmenti, da se za svaki izračuna suma i da se proveri da li je deljiva sa k, brojeći usput takve segmente. Broj je deljiv sa k ako i samo ako daje ostatak 0 pri deljenju sa k, a znamo da je ostatak pri deljenju zbira sa brojem k zapravo jednak zbiru ostataka tj. da važi da je $(a + b) \bmod k = (a \bmod k + b \bmod k) \bmod k$.

Dakle, za svaki segment dovoljno je izračunati zbir po modulu k, pri čemu za fiksirani levi kraj segmenta, zbir svakog narednog segmenta $[i, j]$ dobijamo inkrementalno, od zbira segmenta $[i, j-1]$, dodajući na njega broj a_j i izračunavajući ostatak dobijenog zbira pri deljenju sa k.

```
// izracunava broj segmenata niza a ciji je zbir deljiv sa k
int brojSegmenataZbiraDeljivogSaK(const vector<int>& a,
                                  int k) {
  // duzina niza
  int n = a.size();
  // broj segmenata deljivih sa k
```

```
  int broj = 0;
  // obradjujemo sve segmente [i, j]
  for (int i = 0; i < n; i++) {
    // zbir po modulu k segmenta [i, j] inicijalizujemo na
    // nulu
    int s = 0;
    for (int j = i; j < n; j++) {
      // azuriramo zbir po modulu k segmenta [i, j] na osnovu
      // zbira [i, j-1]
      s = (s + a[j]) % k;
      // ako je zbir po modulu k jednak 0, zbir je deljiv sa k
      if (s == 0)
        broj++;
    }
  }
  return broj;
}
```

Uz ovako inkrementalno izračunavanje zbira, složenost algoritma jednaka je broju segmenata što je $O(n^2)$.

Ostaci zbirova prefiksa

Tražimo broj segmenata $a_p, a_{p+1}, \ldots, a_q$, za $0 \leq p \leq q < n$, takvih da je zbir $S_{pq} = a_p + a_{p+1} + \ldots + a_q$ deljiv sa k.

Obeležimo sa $S_0 = 0$, $S_1 = a_0$, $S_2 = a_0 + a_1$, itd., tj. obeležimo sa S_i, $0 < i \leq n$ zbir prvih i elemenata niza ($S_i = a_0 + a_1 + \ldots + a_{i-1}$). Zbir S_{pq} možemo izraziti kao $S_{q+1} - S_p$.

Na osnovu osobina operacija po modulu važi da je $S_{pq} \bmod k = (S_{q+1} \bmod k - S_p \bmod k + k) \bmod k$.

Prema tome zbir S_{pq} je deljiv sa k (tj. važi $S_{pq} \bmod k = 0$) akko zbirovi S_{q+1} i S_p imaju isti ostatak pri deljenju sa k tj. ako je $S_{q+1} \bmod k = S_p \bmod k$.

Obeležimo sa b_r broj zbirova S_i (za $0 \leq i \leq n$) koji pri deljenju sa k daju ostatak r (za svako $0 \leq r < k$). Svaki par (različitih) zbirova prefiksa koji daju isti ostatak r određuje tačno jedan segment čiji je zbir elemenata deljiv sa k. U skupu od m različitih elemenata postoji tačno $\frac{m(m-1)}{2}$ različitih parova. Zato je za svako r broj

segmenata koji se dobija kombinujući dva zbira koji daju ostatak r jednak $\frac{b_r \cdot (b_r - 1)}{2}$. Prema tome ukupan broj segmenata deljivih sa k je $\sum_{r=0}^{k-1} \frac{b_r \cdot (b_r - 1)}{2}$.

Ostaje još samo pitanje kako izbrojati zbirove prefiksa za svaki dati ostatak tj. kako izračunati sve brojeve b_r. Nizom b dužine k pamtimo broj prefiksa čiji zbirovi elemenata daju ostatke redom $0, 1, 2, \ldots, k-1$ tako da je b_r jednak broju prefiksa čiji zbir elemenata pri deljenju sa k daje ostatak r (što je moguće, s obzirom na datu gornju granicu broja k). Zbirove prefiksa, naravno, izračunavamo inkrementalno.

Polazni zbir je $S_0 = 0$, tako da sve elemente niza b inicijalizujemo na 0, osim vrednosti na poziciji 0 koju inicijalizujemo na 1. Zbir tekućeg prefiksa održavamo u promenljivoj S koju inicijalizujemo na nulu. Učitavamo član po član niza x, i pri tome ažuriramo zbir prefiksa (S postavljamo na $(S + x) \bmod k$), uvećavajući odgovarajući brojač (vrednost b_S uvećavamo za 1).

Primer 3.A.2. *Prikažimo rad ovog algoritma na primeru određivanja broja segmenata niza* $1, 8, 2, 3, 4$ *koji su deljivi sa* 3. *Mogući ostaci su* 0, 1 *i* 2.

i	ai	Si	b0	b1	b2
		0	1	0	0
0	1	1	1	1	0
1	8	0	2	1	0
2	2	2	2	1	1
3	3	2	2	1	2
4	4	0	3	1	2

Zato je konačan rezultat $\frac{b_0(b_0-1)}{2} + \frac{b_1(b_1-1)}{2} + \frac{b_1(b_1-1)}{2} = 3 + 0 + 1 = 4$.

```
// izracunava broj segmenata niza a ciji je zbir deljiv sa k
int brojSegmenataZbiraDeljivogSaK(const vector<int>& a,
                                  int k) {
  // duzina niza
  int n = a.size();

  // na mestu i u nizu br čuvamo broj segmenata čiji zbir pri
  // deljenju sa k daje ostatak i
  vector<int> br(k, 0);
  br[0] = 1;
```

```
  // zbir tekućeg segmenata
  int s = 0;
  for (int i = 0; i < n; i++) {
    // ažuriramo zbir elemenata tekućeg segmenta po modulu k
    s = (s + a[i]) % k;
    br[s]++;
  }

  // izračunavamo ukupan broj segmenata deljivih sa k
  int broj = 0;
  for(int i = 0; i < k; i++)
    broj += br[i]*(br[i]-1)/2;

  return broj;
}
```

Vremenska složenost ovog algoritma je, jasno, $O(n)$. Primetimo da u ovom rešenju nije bilo potrebno koristiti niz za brojeve koje unosimo, jer pri unosu obradimo svaki element, međutim, koristimo pomoćni niz dužine k, pa je memorijska složenost $O(k)$. Obratimo pažnju na to da ovo može biti ograničavajući faktor, ako k može biti jako veliki broj (što u ovom zadatku nije slučaj).

Zadatak: Uvećavanje segmenata

Kamion prevozi teret tokom N kilometara puta. Na put kreće prazan i tokom puta utovaruje i istovaruje pakete. Ako se za svaki paket zna na kom je kilometru puta utovaren, na kom je kilometru puta istovaren i kolika mu je masa, napiši program koji određuje koliko je opterećenje kamiona na svakom kilometru puta. Smatrati da se predmet utovaruje na početku, a istovaruje na kraju datog kilometra.

Opis ulaza

Sa standardnog ulaza se unosi broj kilometara N ($10 \leq N10000$), zatim, u narednom redu, broj predmeta M ($0 \leq M \leq 10000$), a nakon toga, u narednih M redova po tri cela broja razdvojena razmacima koji predstavljaju broj kilometra na čijem je početku utovaren predmet (ceo broj između 0 i $N-1$), broj kilometra na čijem kraju je istovaren (ceo broj između 0 i $N-1$) i na kraju masa predmeta (ceo broj između 1 i 10).

Opis izlaza

Na standardni izlaz ispisati masu tereta u kilogramima na svakom kilometru puta (iza svake mase napisati po jedan razmak).

Primer

Ulaz *Izlaz*

```
10      0 10 25 35 35 35 25 25 15 0
3
1 5 10
3 7 10
2 8 15
```

Objašnjenje

```
        km   0   1  2  3  4  5  9  7  8  9
             0   0  0  0  0  0  0  0  0  0
1 5 10       0  10 10 10 10 10  0  0  0  0
3 7 10       0  10 10 20 20 20 10 10  0  0
2 8 15       0  10 25 35 35 35 25 25 15  0
```

Rešenje

Direktno rešenje

Direktan način je da se održava niz M u kojem se pamti masa na kamionu tokom svakog kilometra puta. Nakon učitavanja svakog podatka o predmetu (početnog kilometra a, završnog kilometra b i mase m), sve vrednosti u nizu M na pozicijama od a do b (uključujući i njih) se uvećavaju za m.

Problem sa ovim rešenjem je to što predmeti mogu putovati veliki broj kilometara pa se u svakom koraku vrši ažuriranje velikog broja članova niza (složenost je u najgorem slučaju $O(n \cdot m)$).

```
int n;
cin >> n;
vector<int> mase(n, 0);
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int km_od, km_do, masa;
  cin >> km_od >> km_do >> masa;
```

```
  for (int km = km_od; km <= km_do; km++)
    mase[km] += masa;
}

for (int masa : mase)
  cout << masa << " ";
```

Razlike susednih elemenata niza

Zadatak možemo rešiti efikasnije ako umesto da u nizu M održavamo masu u kamionu u kilometru i, održavamo razliku između mase u kilometru i i $i-1$ (na poziciji 0 se čuva masa u kamionu u nultom kilometru). Dakle, uvodimo niz R_i takav da je $R_0 = M_0$, a $R_i = M_i - M_{i-1}$, za $1 \leq i < n$. Posmatrajmo šta se dešava sa nizom R kada se u nizu M svi elementi na pozicijama a do b uvećaju za m.

- Vrednost R_a jednaka je razlici $M_a - M_{a-1}$ (ili eventualno M_0 ako je $a = 0$) i ona se uvećava za m, jer je M_a uvećan za m, dok se M_{a-1} ne menja.
- Sve vrednosti od R_{a+1} do R_b ostaju ne promenjene. Naime, za sve njih važi da je $R_i = M_i - M_{i-1}$, a da su i M_i i M_{i-1} uvećani za m.
- Na kraju, vrednost R_{b+1} se umanjuje za m. Naime važi da je $R_{b+1} = M_{b+1} - M_b$, da se M_b uvećava za m, dok se M_{b+1} ne menja.

Recimo da ako je $b = n - 1$, tada ne moramo razmatrati vrednost $R_{b+1} = R_n$ (mada, uniformnosti radi, možemo, što zahteva da niz R sadrži $n + 1$ element). Dakle, prilikom svakog učitavanja brojeva a, b i m potrebno je samo da uvećavamo element R_a za m, a da element R_{b+1} umanjimo za m.

Kada znamo elemente niza R elemente niza M možemo jednostavno rekonstruisati sabiranjem elemenata niza R. Naime, važi da je $M_0 = R_0$, dok je $M_i = M_{i-1} + R_i$, tako da svaki naredni element niza M možemo izračunati kao zbir prethodnog elementa niza M i njemu odgovarajućeg elementa niza R. Primetimo da je zapravo element M_i jednak zbiru svih elemenata od R_0 do R_i, jer je $R_0 + R_1 + ... + R_i = M_0 + (M_1 - M_0) + ... + (M_i - M_{i-1}) = M_i$.

Ukupna složenost ovog pristupa je $O(n + m)$.

```
int n;
cin >> n;
vector<int> razlika(n+1, 0);
int m;
cin >> m;
for (int i = 0; i < m; i++) {
  int km_od, km_do, masa;
  cin >> km_od >> km_do >> masa;
  razlika[km_od] += masa;
  razlika[km_do+1] -= masa;
}

int masa_km = 0;
for (int km = 0; km < n; km++) {
  masa_km += razlika[km];
  cout << masa_km << " ";
}
```

Ideja korišćena u ovom zadatku donekle je slična (zapravo inverzna) tehnici određivanja zbira segmenata kao razlike dva zbira prefiksa. Može se primetiti da se rekonstrukcija niza vrši zapravo izračunavanjem prefiksnih zbirova niza razlika susednih elemenata, što ukazuje na duboku vezu između ove dve tehnike. Zapravo, razlike susednih elemenata predstavljaju određeni diskretni analogon izvoda funkcije, dok prefiksni zbirovi predstavljaju analogiju određenog integrala. Izračunavanje zbira segmenta kao razlike dva zbira prefiksa odgovara Njutn-Lajbnicovoj formuli.

3.A.5 *Primena sortiranja*

Zadatak: Ravnomerna podela poslova

Poznato je n računskih poslova koje treba izvršiti i za svaki od njih je poznato koliko je vremena potrebno procesoru da ga izvrši. Svakom od k identičnih procesora treba dodeliti tačno po jedan posao, pri čemu je cilj da svi procesori budu što ravnomernije opterećeni. Kolika je najmanja moguća razlika između vremena izvršavanja dva posla dodeljena tim procesorima?

Opis ulaza

Sa standardnog ulaza se unosi prirodan broj n ($1 \leq n \leq 50000$) a zatim i n prirod-

nih brojeva (između 1 i 10^6, razdvojenih po jednim razmakom) koji predstavljaju vreme potrebno za izvršavanje svakog od poslova. U poslednjem redu se unosi broj procesora k ($1 \leq k \leq n$).

Opis izlaza

Na standardni izlaz ispisati vrednost najmanje razlike između dva posla dodeljena ovim procesorima.

Primer

Ulaz	*Izlaz*
8 3 8 1 17 4 7 12 9 4	5

Objašnjenje

Najmanja razlika se dobija ako procesori izvršavaju poslove koji traju 8, 4, 7 i 9 jedinica vremena.

Rešenje

Sortiranje

Najdirektniji način da se reši zadatak je da se ispitaju svi podskupovi od k elemenata skupa od n elemenata i da se među njima odabere najbolji. Ovo rešenje je relativno komplikovano implementirati, a uz to je i veoma neefikasno (broj podskupova je $\binom{n}{k}$, što je $O(n^k)$).

Bolje i efikasnije rešenje se zasniva na sortiranju. Naime, kada se polazni poslovi sortiraju po vremenu izvršavanja, procesorima treba dodeliti nekih uzastopnih k poslova. Pretpostavimo da nakon sortiranja imamo niz $a_0, a_1, \ldots, a_{n-1}$. Procesorima treba dodeliti redom poslove od a_i, do a_{i+k-1}, za neko $0 \leq i \leq n-k$.

Dokažimo prethodnu činjenicu. Pretpostavimo suprotno, da poslovi koji daju najmanji raspon ne čine uzastopan niz i da svaki uzastopni niz poslova dužine k ima strogo veći raspon od raspona skupa uzetih paketa. Neka je prvi uzeti posao a_i. Tada sigurno postoji neki posao a_j za $i < j < i + k$ koji nije uzet, a umesto njega je uzet neki paket $a_{j'}$ za neko $i + k \leq j' < n$. Neka je j' poslednji paket koji je uzet. Međutim, kada bi procesor koji izvršava posao $a_{j'}$ zamenio taj posao za a_j, raspon bi se sigurno smanjio ili bar ostao isti (jer bi poslednji uzeti paket tada bio neki paket $a_{j''}$, za $j'' < j'$, a pošto je niz sortiran neopadajuće, važi da je $a_{j''} \leq a_j$, pa i $a_{j''} - a_i \leq a_{j'} - a_i$. Daljim zamenema istog tipa možemo doći do toga da

su svi uzeti paketi uzastopni, a da je raspon manji ili jednak polaznom, što je u kontradikciji sa tim da je raspon polaznog skupa uzetih paketa strogo manji od raspona bilo kojeg niza k uzastopnih paketa.

Razmatranje prethodnog tipa je karakteristično za takozvane gramzive (tj. pohlepne) algoritme, o kojima će više reči biti kasnije.

Na osnovu prethodnog, jasno je da niz trajanja poslova treba najpre sortirati i zatim odrediti minimum razlika vrednosti $a_{i+k-1} - a_i$, za $0 \leq i \leq n - k$, korišćenjem uobičajenog algoritma za nalaženje minimuma.

Složenošću ovog algoritma dominira složenost koraka sortiranja, a ona je $O(n \log n)$, ako se koriste bibliotečke implementacije. Nakon sortiranja, minimum se određuje u $n - k$ koraka, tj. u linearnoj složenosti $O(n)$ (kada je k malo, broj koraka može biti veoma blizak n).

```
// odredjuje najmanju razliku u trajanju poslova
// ako se bira k poslova medju poslovima iz niza a
int minRazlika(vector<int>& a, int k) {
  // broj paketa
  int n = a.size();
  // sortiramo pakete po trajanju poslova
  sort(begin(a), end(a));
  // odredjujemo i najmanju mogucu razliku
  int min = numeric_limits<int>::max();
  for (int i = 0; i + k - 1 < n; i++) {
    int razlika = a[i + k - 1] - a[i];
    if (razlika < min)
      min = razlika;
  }
  return min;
}
```

Zadatak: Anagrami

Dve reči su anagrami ako se jedna može dobiti od druge samo promenom redosleda slova (na primer, trava i vatra). Napiši program koji u datom skupu reči pronalazi najveći podskup reči koje su međusobni anagrami.

Opis ulaza

Sa standardnog ulaza se unosi broj reči n ($0 \leq n \leq 50000$), a zatim u n narednih linija po jedna reč polaznog skupa (sve reči se sastoje samo od malih slova engleskog alfabeta i imaju najviše 200 karaktera).

Opis izlaza

Na standardni izlaz ispisati veličinu najvećeg podskupa polaznog skupa reči, u kome su sve reči jedna drugoj anagrami.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`10`	`4`	Najbrojniju skup anagrama čine reči
`tommarvoloriddle`		`viviandarkbloom`
`viviandarkbloom`		`vladimirnabokov`
`iamlordvoldemort`		`bladvakvinomori`
`danabnormal`		`dorianvivalkomb`
`normdanabal`		
`vladimirnabokov`		
`vladimirkoborov`		
`bladvakvinomori`		
`damonalbarn`		
`dorianvivalkomb`		

Rešenje

Provera da li su dve reči anagrami može se zasnivati na sortiranju slova dve reči i zatim poređenju da li su dobijene reči iste. Alternativno, provera da li su dve reči anagrami može se uraditi tako što se izračunaju brojevi pojavljivanja svakog od 26 karaktera engleske abecede i zatim se porede ti brojevi za dve zadate reči. U oba pristupa, svaku reč *kanonizujemo* i umesto poređenja polaznih reči, možemo da poredimo njihove kanonske oblike. Kanonizovanjem dobijamo niz kanonskih predstavnika i želimo da u tom nizu pronađemo skup ekvivalentnih elemenata (jednakih kanonskih predstavnika) koji je najveće kardinalnosti od svih takvih podskupova.

Sortiranje slova, pa sortiranje reči

Jedno moguće rešenje je da sortiramo karaktere svake učitane reči (abecedno), a zatim da sortiramo tako dobijeni niz reči (leksikografski abecedno), da bi se identične

sortirane reči našle jedna iza druge. Na tako dobijen niz primenjujemo pronalaženje najduže serije jednakih uzastopnih elemenata.

Složenost sortiranja slova svih pojedinačnih reči je $O(n \cdot m \log m)$, gde je n broj reči, a m maksimalni broj slova u reči. Složenost sortiranja niza svih reči se može proceniti na $O(n \log n)$, jer je realno očekivati da će se leksikografsko poređenje dve reči vršiti veoma efikasno (reči su kratke, a realno je očekivati i da su "šarenolike", tj. da će se njihov poredak moći odrediti već nakon poređenja malog broja početnih karaktera). Pošto je broj slova m veoma mali, možemo ga smatrati konstantim i složenost algoritma grubo oceniti sa $O(n \log n)$.

```
// sortiramo karaktere svake reci
for (int i = 0; i < n; i++)
  sort(begin(reci[i]), end(reci[i]));
// sortiramo ceo niz reci leksikografski
sort(begin(reci), end(reci));

// odredjujemo duzinu najduze serije jednakih reci
int maxDuzinaSerije = 1;
int tekucaDuzinaSerije = 1;
for (int i = 1; i < n; i++) {
  if (reci[i] == reci[i-1])
    tekucaDuzinaSerije++;
  else
    tekucaDuzinaSerije = 1;
  if (tekucaDuzinaSerije > maxDuzinaSerije)
    maxDuzinaSerije = tekucaDuzinaSerije;
}

cout << maxDuzinaSerije << endl;
```

Zadatak: D-permutacija

Dat je niz koji sadrži prirodne brojeve i ceo broj D. Napiši program koji određuje koji se od nekoliko datih nizova može od polaznog dobiti razmenama elemenata koji su tačno na rastojanju D.

Opis ulaza

Sa standardnog ulaza se učitava ceo broj d ($1 \leq d \leq k$), zatim ceo broj k ($5 \leq k \leq 10^5$), zatim ceo broj n ($1 \leq n \leq 10$) i nakon toga n nizova dužine k čiji su elementi razdvojeni razmacima.

Opis izlaza

Na standardni izlaz za svaki niz od drugog do poslednjeg ispisati `da` ako se može transformisati u prvi razmenama elemenata na rastojanju d, tj. `ne` u suprotnom.

Primer

Ulaz *Izlaz*

```
2                 da
7                 ne
6                 da
1 2 3 4 5 6 7     da
3 4 1 2 7 6 5     ne
2 1 4 3 6 5 7
1 4 7 2 3 6 5
1 4 7 6 3 2 5
5 4 7 6 3 1 2
```

Objašnjenje

```
3 4 1 2 7 6 5   - razmena 3 i 1, 4 i 2, i 7 i 5
2 1 4 3 6 5 7   - ne može
1 4 7 2 3 6 5   - razmena 4 i 2, 7 i 3 i 7 i 5
1 4 7 6 3 2 5   - razmena 6 i 2, 4 i 2, 7 i 3 i 7 i 5
5 4 7 6 3 1 2   - ne može
```

Rešenje

Sortiranje

Primetimo da se element na poziciji 0, nakon permutacije može naći samo na poziciji d, $2d$, $3d$ itd. Element na poziciji 1 se može naći samo na pozicijama $d+1$, $2d+1$, $3d+1$ itd. Na kraju, element na poziciji $d-1$ se može naći samo na pozicijama $d+d-1$, $2d+d-1$, $3d+d-1$ itd.

Primer 3.A.3. *Na primer, u nizu,* 1, 2, 3, 4, 5, 6, 7, 8, 9, *ako je rastojanje* d *jednako* 3, *možemo razmenjivati samo elemente* 1, 4 *i* 7, *zatim elemente* 2, 5 *i* 8 *i na kraju* 3, 6 *i* 9. *Ne postoji nikakav način da se, na primer, element* 2 *ili element* 9 *nađu na poziciji elementa* 1 *ili elementa* 7.

Dakle, niz je suštinski izdeljen na d particija, tako što su dva elementa u istoj particiji ako i samo ako su na rastojanju $i \cdot d$, za neko celobrojno i. Particije su međusobno nezavisne u smislu da se razmenama mogu permutovati samo oni elementi koji se nalaze u istoj particiji. Jedan niz se može transformisati u drugi razmenama elemenata na rastojanju d, ako i samo ako se svaka particija može transformisati u odgovarajuću particiju drugog niza, razmenama susednih elemenata u okviru te particije.

Primer 3.A.4. *Na primer, niz* $1, 2, 3, 4, 5, 6, 7, 8, 9$ *se može transformisati u niz* $7, 2, 9, 1, 8, 6, 4, 5, 3$*, zato što se particija* $(1, 4, 7)$ *može transformisati u* $(7, 1, 4)$*, particija* $(2, 5, 8)$ *se može transformisati u* $(2, 8, 5)$*, a particija* $(3, 6, 9)$ *se može transformisati u particiju* $(9, 6, 3)$*. Pošto svaka particija čini niz za sebe, ostaje da se reši pitanje: da li se dati niz brojeva može transformisati u drugi niz operacijama razmene susednih elemenata?*

Postoje razni načini da se na ovo pitanje odgovori, a jedan od najefikasnijih se zasniva na činjenici da se svaki niz može sortirati isključivo razmenama uzastopnih elemenata (na primer, Bubble Sort algoritam vrši upravo takvo sortiranje). To znači da se jedan niz transformacijama uzastopnih elemenata može transformisati u drugi niz ako i samo ako se sortiranjem i jednog i drugog dobija isti niz. Pri tom, sortiranje ne mora biti vršeno samo razmenama uzastopnih elemenata već i mnogo efikasnijim algoritmima (jer je rezultat sortiranja jedinstveno određen i ako znamo da se nekim efikasnim algoritmom sortiranja od početnog niza dobija neki sortiran niz, taj isti niz bi se dobio i da se sortiranje vrši samo razmenama uzastopnih elemenata). Jedan od načina da prethodna razmatranja iskombinujemo u rešenje je da uvedemo operaciju pronalaženja kanonskog predstavnika za svaki niz u odnosu na dati parametar d, tako što ćemo svaku njegovu particiju kanonizovati tj. sortirati.

Primer 3.A.5. *Na primer, za* $d = 3$ *niz* $2, 16, 41, 15, 12, 31, 11, 9$ *se deli na particije* $(2, 15, 11)$*,* $(16, 12, 9)$*,* $(41, 31)$*, svaka se particija se nezavisno sortira i time se dobija* $(2, 11, 15)$*,* $(9, 12, 16)$*, i* $(31, 41)$*, čime se dobija kanonski predstavnik* $2, 9, 31, 11, 12, 41, 15, 16$*. Dva niza će se moći d-permutovati jedan u drugi ako i samo ako su njihovi ovako definisani kanonski predstavnici jednaki.*

Pošto u okviru kanonizacije koristimo bibliotečki algoritam sortiranja možemo pretpostaviti da će njegova složenost za niz dužine m biti $O(m \cdot \log(m))$. Pošto se sortira d particija, a svaka je dužine oko k/d, i vrši se prebacivanje elemenata u pomoćni niz i nazad, složenost najgoreg slučaja kanonizacije se može proceniti kao

$O(d \cdot (k/d \cdot \log(k/d) + 2(k/d)))$ tj. $O(k \cdot \log(k/d))$. Zato izračunavanje kanonskih predstavnika svih nizova može da se proceni na $O(n \cdot k \cdot \log(k/d))$. S obzirom na to da se poređenje jednakosti dva kanonska predstavnika vrši u vremenu $O(k)$, a ovo poređenje se vrši $n - 1$ put, imamo dodatno još $O(nk)$ operacija. Za ukupnu složenost algoritma se zato može uzeti $O(n \cdot k \cdot \log(k/d))$, tj. $O(k \cdot \log(k/d))$ jer n, za razliku od k ima veoma malu vrednost (praktično se može smatrati konstantom).

Napomenimo da je nakon kanonizacije svake particije niza T_i bilo moguće proveravati jednakost sa odgovarajućom particijom niza S, čime bi se malo brže moglo konstatovati da se niz ne može transformisati, ali bi se kod donekle zakomplikovao. Slično, korišćenje pomoćnog niza nije bilo neophodno za čuvanje particija, već je bilo moguće kanonizaciju vršiti u okviru niza S, razmenama njegovih elemenata iz istih particija, ali tada ne bi bilo moguće koristiti bibliotečku funkciju sortiranja, što bi takođe dosta zakomplikovalo kod. Asimptotska složenost najgoreg slučaja bi ostala ista.

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

using namespace std;

// određivanje kanonskog predstavnika za niz s
void kanonizuj(vector<int>& s, int d) {
  int duzina = s.size();
  // pomocni vektor koji ce cuvati particiju po particiju
  vector<int> particija(duzina / d + 1);

  // kanonizovacemo svaku od d particija
  for (int i = 0; i < d; i++) {
    int m;

    // prebacujemo elemente i-te particije u pomocni vektor
    m = 0;
    for (int j = i; j < duzina; j += d)
      particija[m++] = s[j];
```

```
    // sortiramo elemente particije unutar pomocnog vektora
    sort(particija.begin(), particija.begin() + m);

    // vracamo elemente particije iz pomocnog vektora nazad u niz s
    m = 0;
    for (int j = i; j < duzina; j += d)
      s[j] = particija[m++];
  }
}


int main() {
  // iskljucujemo sinhronizaciju da bi ucitavanje teklo brze
  ios::sync_with_stdio(false);

  // ucitavamo rastojanje d
  int d;
  cin >> d;
  // ucitavamo dužinu nizova k
  int k;
  cin >> k;
  // ucitavamo broj nizova
  int n;
  cin >> n;

  // ucitavamo niz S
  vector<int> s(k);
  for (int i = 0; i < k; i++)
    cin >> s[i];

  // odredjujemo kanonskog predstavnika niza s
  kanonizuj(s, d);

  // obradjujemo preostalih n-1 nizova Ti
  for (int j = 1; j < n; j++) {
    // ucitvamo niz Ti
```

```
    vector<int> t(k);
    for (int i = 0; i < k; i++)
      cin >> t[i];

    // odredjujemo kanonskog predstavnika niza Ti
    kanonizuj(t, d);

    // odredjujemo da li se niz S moze transformisati u niz Ti
    // (moze ako i samo ako su im kanonski predstavnici jednaki)
    // i ispisujemo rezultat u trazenom formatu
    cout << (s == t ? "da" : "ne") << endl;
  }

  return 0;
}
```

Zadatak: Pokrivanje prave zatvorenim intervalima

Napisati program koji za niz zatvorenih intervala $[a_i, b_i], i = 0, 1, \cdots, n-1$ određuje ukupnu dužinu delova prave koje pokrivaju i minimalni broj intervala kojim se može postići pokrivanje istog skupa tačaka prave (taj skup intervala može se dobiti ukrupnjivanjem polaznih intervala, tj. objedinjavanjem polaznih intervala koji se seku).

Opis ulaza

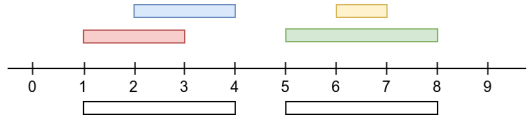
U prvoj liniji standardnog ulaza učitava se broj intervala n ($1 \leq n \leq 50000$), a u narednih n linija parovi celih brojeva za koje važi $-10^6 \leq L_i < R_i \leq 10^6$ koji predstavljaju levi i desni kraj intervala.

Opis izlaza

Dva broja: u prvom redu standardnog izlaza broj koji predstavlja dužinu dela prave koju pokrivaju učitani intervali, u sledećem redu broj intervala formiranih ukrupnjavanjem međusobno povezanih intervala.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
4 1 3 5 8 2 4 6 7	6 2	Intervali $[1, 3]$ i $[2, 4]$ se mogu ukrupniti u interval $[1, 4]$, a intervali $[5, 8]$ i $[6, 7]$ u interval $[5, 8]$.



Originalni intervali su prikazani iznad, a ukrupnjeni ispod prave.

Rešenje

Lista ukrupnjenih intervala

Zadatak možemo rešiti tako što jedan po jedan interval dodajemo u skup ukrupnjenih intervala formiranih na osnovu prethodno obrađenih intervala. Dodavanje novog intervala u kolekciju ukrupnjenih može se realizovati tako što se svi ukrupnjeni intervali koji se seku sa intervalom koji se trenutno obrađuje uklone iz kolekcije ukrupnjenih intervala, zatim se napravi njihova unija sa intervalom koji se dodaje i da se tako dobijen ukrupnjeni interval doda u trenutnu kolekciju ukrupnjenih intervala. Za ovo nam je potrebno da implementiramo proveru da li se dva intervala seku i izračunavanje unije dva intervala (što možemo uraditi korišćenjem minimuma i maksimuma). S obzirom na to da nam je potrebno da iz kolekcije ukrupnjenih intervala (efikasno) uklanjamo elemente dok je obilazimo, pogodno je da je predstavimo povezanom listom. U jeziku C++ možemo upotrebiti kolekciju `list`.

Uklanjanje elementa liste vrši se u vremenu $O(1)$. Pošto se prilikom dodavanja svakog novog intervala obilaze svi prethodno ukrupnjeni intervali (a njih može biti isto onoliko koliko i polaznih intervala, ako su intervali disjunktni), složenost u najgorem slučaju je $O(n^2)$.

```
typedef pair<int, int> Interval;

// provera da li se dva intervala seku
bool sekuSe(Interval i1, Interval i2) {
  return max(i1.first, i2.first) <= min(i1.second, i2.second);
}

// pokrivac dva data intervala
```

```
Interval pokrivac(Interval i1, Interval i2) {
  return make_pair(min(i1.first, i2.first),
                   max(i1.second, i2.second));
}

int main() {
  // učitavamo podatke o intervalima
  int n;
  cin >> n;
  vector<Interval> intervali(n);
  for (int i = 0; i < n; i++)
    cin >> intervali[i].first >> intervali[i].second;

  // lista svih ukrupnjenih ranije obradjenih intervala
  list<Interval> ukrupnjeni;
  // analiziramo jedan po jedan dati interval
  for (Interval interval : intervali) {
    // uniramo ga sa svim ranije ukrupnjenim intervalima sa
    // kojima se sece, uklanjajuci ih pritom iz liste
    Interval novi = interval;
    auto it = ukrupnjeni.begin();
    while (it != ukrupnjeni.end()) {
      if (sekuSe(interval, *it)) {
        novi = pokrivac(novi, *it);
        ukrupnjeni.erase(it++);
      } else
        it++;
    }
    // dodajemo u listu novi ukrupnjenih interval
    ukrupnjeni.push_back(novi);
  }

  // izracunavamo i ispisujemo broj ukrupnjenih intervala i
  // njihovu ukupnu duzinu
  int broj_ukrupnjenih = ukrupnjeni.size();
  int duzina_pokrivaca = 0;
```

```
  for (auto interval : ukrupnjeni)
    duzina_pokrivaca += interval.second - interval.first;

  cout << duzina_pokrivaca << endl
       << broj_ukrupnjenih << endl;

  return 0;
}
```

Sortiranje intervala prema levim krajevima

Efikasan algoritam možemo dobiti ako prilikom dodavanja novog intervala nekako izbegnemo potrebu da se obilaze svi do tada ukrupnjeni intervali. Osnovna ideja je da intervale obilazimo u neopadajućem poretku koordinata levih krajeva. Ako intervale predstavimo uređenim parovima, bibliotečke funkcije sortiranja će ih sortirati upravo na taj način. S obzirom na takav redosled obilaska, svaki novi interval se može seći samo sa poslednjim ukrupnjenim intervalom (i stoga se može preskočiti analiza ostalih ukrupnjenih intervala).

S obzirom na to da nas ne zanimaju sami ukrupnjeni intervali, već samo njihov broj i dužina, tokom izvršavanja petlje u kojoj obrađujemo jedan po jedan interval održavaćemo samo tri podatka: dužinu dela prave koju pokrivaju do tada obrađeni intervali, broj ukrupnjenih intervala koji pokrivaju taj deo prave (pokrivač tog dela prave) i poziciju D_{max} desnog kraja poslednjeg ukrupnjenog intervala (to je ujedno najdešnja tačka svih do sada obrađenih intervala).

Inicijalizaciju možemo izvršiti na osnovu prvog intervala (on sam pokriva deo prave čija je dužina jednaka njegovoj, trenutno je formiran jedan ukrupnjeni interval i desni kraj je jednak desnom kraju tog prvog intervala).

U svakom koraku proširujemo dosadašnji pokrivač intervalom $[L_i, D_i]$, pri čemu je poslednji ukrupnjeni interval $[L, D_{max}]$. Moguća su sledeća 3 slučaja:

1. $D_{max} < L_i$ — interval $[L_i, D_i]$ se ne može priključiti ni jednom već postojećem ukrupnjenom intervalu, niti se to može uraditi ni sa jednim narednim intervalom (jer su zbog sortiranosti svi oni desno od tekućeg). Tekući interval zato započinje novi ukrupnjeni interval, pa uvećavamo broj ukrupnjenih intervala za 1, dužinu pokrivenog dela prave povećavamo za dužinu tekućeg intervala, dok se D_{max} koriguje na D_i.

2. $L \leq L_i \leq D_{max} \leq D_i$ — dosadašnji pokrivač tj. njegov poslednji ukrupnjeni interval $[L, D_{max}]$ se seče sa intervalom $[L_i, D_i]$ pa se proširuje za dužinu $D_i - D_{max}$, a kraj D_{max} se koriguje na D_i, pri čemu se broj ukrupnjenih intervala ne menja.

3. $L \leq L_i \leq D_i \leq D_{max}$ — tekući interval $[L_i, D_i]$, je kompletno sadržan u nekom ukrupnjenom intervalu $[L, D_{max}]$ i može se preskočiti bez ažuriranja pokrivača koji se konstruiše.

Sortiranje n intervala vrši se u vremenu $O(n \log n)$. Nakon toga, obrada intervala vrši se jednim prolaskom kroz niz i složenost te faze je $O(n)$. Ukupnom složenošću, dakle, dominira sortiranje i ona iznosi $O(n \log n)$.

```
// sortiramo intervale na osnovu njihovog levog kraja
sort(begin(intervali), end(intervali));

// prvi interval ukljucujemo u pokrivac
int duzina_pokrivaca = intervali[0].second -
                       intervali[0].first;
// desni kraj poslednjeg do sada ukrupnjenog intervala
int desni_kraj = intervali[0].second;
// broj ukrupnjenih intervala
int broj_ukrupnjenih = 1;
for (int i = 1; i < n; i++)
  // trenutni interval se ne sece sa trenutno poslednjim
  // ukrupnjenim intervalom (leži desno od njega)
  if (intervali[i].first > desni_kraj) {
    // zapocinjemo novi ukrupnjeni interval
    broj_ukrupnjenih++;
    duzina_pokrivaca += intervali[i].second -
                        intervali[i].first;
    desni_kraj = intervali[i].second;
  } else if (intervali[i].second > desni_kraj) {
    // trenutni interval prosiruje trenutno poslednji
    // ukrupnjeni interval
    duzina_pokrivaca += intervali[i].second - desni_kraj;
    desni_kraj = intervali[i].second;
```

```
}
```

3.A.6 Binarna pretraga

Zadatak: Najvredniji poklon

Ovaj zadatak je ponovljen u cilju ilustrovanja različitih tehnika rešavanja.

Tekst zadatka.

Opis ulaza

- U prvom redu nalazi se ceo broj n $(1 \leq n \leq 10^5)$ — broj poklona.
- U drugom redu nalazi se n celih brojeva $c_1, c_2, \ldots, c_n$ $(1 \leq c_i \leq 10^9)$, koji predstavljaju cene poklona.
- U trećem redu nalazi se ceo broj q $(1 \leq q \leq 10^5)$ — broj kupaca.
- U četvrtom redu nalazi se q celih brojeva $b_1, b_2, \ldots, b_q$ $(1 \leq b_i \leq 10^9)$, gde b_i predstavlja budžet i-tog kupca.

Opis izlaza

Za svakog kupca ispisati po jedan ceo broj u zasebnom redu – cenu najskupljeg poklona koji je manji ili jednak od x_i.

Ukoliko takav poklon ne postoji, ispisati -1.

Primer

Ulaz	*Izlaz*
`5`	`-1`
`20 10 70 35 50`	`35`
`3`	`70`
`5 40 70`	

Rešenje

Nakon što se niz cena sortira, zadatak se veoma efikasno može rešiti *binarnom pretragom*, primenjenu iznova na svaki budžet b_i. Krenimo od sledeće invarijante. Tokom petlje održavaćemo dva pokazivača `l` i `d`, tako da su sve cene levo od pokazivača `l` takve da se poklon može kupiti za dati budžet, a sve cene desno od pokazivača `d` takve da se poklon ne može kupiti za dati budžet, dok su na pozicijama iz intervala

$[l, d]$ cene koje algoritam još nije ispitao. Dakle, pretpostavimo da je stanje niza sledeće:

			l				d			
$\leq$	$\leq$	$\leq$	?	?	?	?	?	>	>	>

U početku su sve cene nepregledane, pa interval $[l, d]$ treba da se poklopi sa celim nizom, tj. l treba inicijalizovati na 0, a d na $n - 1$.

U svakom koraku petlje određujemo središnju poziciju s u intervalu $[l, d]$. Da bi se smanjila mogućnost prekoračenja, tu poziciju umesto pomoću $\lfloor \frac{l+d}{2} \rfloor$, možemo izračunati pomoću $l + \lfloor \frac{d-l}{2} \rfloor$.

- Ako se element na poziciji s uklapa u dati budžet tj. ako je cena $c_s \leq b_i$, tada se, zbog sortiranosti, mogu priuštiti i svi pokloni na pozicijama levo od s, pa invarijanta ostaje na snazi ako se l pomeri na poziciju $s + 1$.
- U suprotnom, ako je $b_i < c_s$, tada se, zbog sortiranosti, ne može prišutiti ni jedan poklon na pozicijama desno od s, pa invarijanta ostaje na snazi ako se l pomeri na poziciju $s - 1$.

Petlja se izvršava dok god ima nepregledanih cena, tj. dok je interval $[l, d]$ neprazan. Kada se pretraga završi, stanje u nizu je sledeće.

				d	l					
$\leq$	$\leq$	$\leq$	$\leq$	$\leq$	>	>	>	>	>	>

Na osnovu invarijante znamo da se svaki element levo od l može priuštiti, a da se ni jedan desno od d ne može, pa se stoga traženi najvredniji poklon nalazi na poziciji $d = l - 1$. Ako su svi pokloni preskupi, krajnje vrednosti će biti $d = -1$ i $l = 0$, pa se vraćanjem vrednosti d i u tom slučaju dobija korektan rezultat.

Funkcija se zaustavlja jer se u svakom koraku širina intervala $[l, d]$ striktno smanjuje (zahvaljujući zaokruživanju naniže prilikom određivanja središnje pozicije s).

```
// vraca poziciju u sortiranom nizu cene najvrednijeg poklona
// koji se moze kupiti za dati budzet ili
// -1 ako su svi pokloni preskupi
int najvredniji_poklon(const vector<int>& cene, int budzet) {
  int n = cene.size();
  int l = 0, d = n - 1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (cene[s] <= budzet)
      l = s + 1;
    else
      d = s - 1;
  }
  return d;
}
```

Zadatak: i-ti na mestu i

Napisati program koji proverava da li u strogo rastućem nizu elemenata postoji pozicija i takva da se na poziciji i nalazi vrednost i tj. da važi da je $a_i = i$ (pozicije se broje od nule).

Opis ulaza

Sa standardnog ulaza se unosi broj n ($0 \leq n \leq 10^5$), a zatim i strogo rastući niz od n celih brojeva (razdvojenih razmacima).

Opis izlaza

Na standardni izlaz ispisati indeks i takav da je $a_i = i$ ili tekst `nema` ako takav indeks ne postoji u nizu. Ako u nizu postoji više takvih indeksa ispisati najmanji od njih.

Primer

Ulaz	*Izlaz*
`6` `-3 -1 1 3 5 7`	`3`

Rešenje

Linearna pretraga

Direktan način da se zadatak reši je da se upotrebi linearna pretraga i da se pozicije proveravaju redom, od 0 do $n-1$ sve dok se ne pronađe prva pozicija koja zadovoljava uslov ili dok se ne dođe do kraja niza.

Složenost linearne pretrage je $O(n)$.

```
int iti_na_mestu_i(const vector<int>& a) {
  for (int i = 0; i < a.size(); i++)
    if (a[i] == i)
      return i;
  return -1;
}
```

Binarna pretraga transformisanog niza

Razmotrimo niz `-10 -4 1 3 4 9 11`. Element `-10` je manji od svoje pozicije 0 za 10. Element `-4` je manji od svoje pozicije `1` za `5`, element `1` je manji od svoje pozicije 2 za 1. Elementi `3` i `4` su jednaki svojim pozicijama. Element `9` je veći od svoje pozicije `5` za `4` dok je element `11` veći od svoje pozicije `6` za `5`. Primećujemo određenu monotonost u ovom nizu, što nije slučajno. Pokažimo da je niz $a_i - i$ neopadajući. Posmatarajmo dva elementa a_i i a_j na pozicijama na kojima je $0 \leq i < j$. Pošto je niz a strogo rastući, važi da je $a_{i+1} > a_i$, pa je $a_{i+1} \geq a_i + 1$. Slično je $a_{i+2} > a_{i+1}$, pa je $a_{i+2} \geq a_{i+1} + 1 \geq a_i + 2$. Nastavljanjem ovakvog zaključivanja dobijamo da važi da je $a_j \geq a_i + (j - i)$. Zato je $a_j - j \geq a_i - i$. Ako je $a_i = i$, tada je $a_i - i = 0$. Rešenje, dakle, možemo odrediti tako što binarnom pretragom proverimo da li neopadajući niz $a_i - i$ sadrži nulu i ako sadrži, tada je rešenje prva pozicija na kojoj se ta nula nalazi.

Jedan od najlakših načina da realizujemo binarnu pretragu je da upotrebimo bibliotečku funkciju. Pošto nam je potrebna prva pozicija nule u transformisanom nizu, ne možemo upotrebiti funkciju `binary_search`, već moramo upotrebiti funkciju `lower_bound`.

Složenost binarne pretrage je $O(\log n)$, međutim, vremenom dominira učitavanje i transformisanje niza koje zahteva $O(n)$ koraka.

```
int iti_na_mestu_i(const vector<int>& a) {
  int n = a.size();
  // pripremamo ga za pretragu a[i] = i akko je a[i] - i = 0,
  // zato umesto niza a, pretrazujemo neopadajuci niz a[i] - i
  vector<int> b(n);
  for (int i = 0; i < n; i++)
    b[i] = a[i] - i;

  // trazimo poziciju nule u transformisanom nizu
  // pronalazimo poziciju prvog elementa koji je >= 0
  auto it = lower_bound(b.begin(), b.end(), 0);

  // ako takav element postoji i ako je jednak nuli
  if (it != b.end() && *it == 0)
    // pronasli smo element i izracunavamo njegovo rastojanje od
    //  pocetka niza
    return distance(b.begin(), it);
  else
    // u suprotnom element ne postoji u nizu
    return -1;
}
```

Binarna pretraga bez transformisanja niza

Binarna pretraga se može prilagoditi i primeniti tako da se ne modifikuje niz.

Osnovna implementacija binarne pretrage

Jedan mogući pokušaj implementacije prati osnovnu varijantu binarne pretrage u kojoj se traži element unutar niza. Nakon nalaženja središnjeg elementa možemo proveriti sledeće uslove.

- Ako je $a_s < s$ (tada bi u transformisanom nizu važilo da je $a_s - s < 0$, pretragu je potrebno nastaviti desno od pozicije s),
- Ako je $a_s > s$ (tada bi u transformisanom nizu važilo da je $a_s - s > 0$), pretragu je potrebno nastaviti levo od pozicije s.
- Ako je $a_s = s$, tada je element pronađen.

Kada se pronađe neki element koji zadovoljava dati uslov, potrebno je proveriti da li je najmanji. Pošto drugi elementi koji zadovoljavaju dati uslov mogu biti samo neposredno uz pronađeni element, najmanji element koji zadovoljava uslov pronalazimo krećući se ulevo tj. menjajući tekući element njemu prethodnim sve dok taj prethodni element zadovoljava traženi uslov (ili dok eventualno ne dođemo do samog početka niza). Naglasimo da je ovakva implementacija binarne pretrage zapravo loša, jer u slučaju kada u nizu postoji veliki broj elemenata koji su na svom mestu, pomeranje ulevo može trajati dugo.

Složenost binarne pretrage je $O(\log n)$, međutim, dalja pretraga za prvim elementom koji zadovoljava uslov je u najgorem slučaju složenosti $O(n)$, što poništava prednosti binarne pretrage. Vremenom svakako dominira učitavanje i transformisanje niza koje zahteva $O(n)$ koraka.

```
int iti_na_mestu_i(const vector<int>& a) {
  // sprovodimo binarnu pretragu - ako trazeni element a[k] = k
  // postoji, on se nalazi u intervalu [l, d]
  int l = 0, d = a.size()-1;
  // dok interval [l, d] nije prazan
  while (l <= d) {
    // pronalazimo sredinu intervala
    int s = l + (d - l) / 2;
    if (a[s] < s)
      // trazeni element se moze nalaziti samo desno od s
      l = s + 1;
    else if (a[s] > s)
      // trazeni element se moze nalaziti samo levo od s
      d = s - 1;
    else {
      // pronasli smo element na poziciji s
      // mozda postoji i neki pre njega?
      // Opasnost - ovo može biti linearna pretraga
      while (s - 1 >= 0 && a[s - 1] == s - 1)
        s = s - 1;
      return s;
    }
  }
```

```
    // nismo nasli element
    return -1;
}
```

Zadatak: Prvi koji nije deljiv

Razmotrimo niz brojeva $210, 2310, 390, 30, 510, 66, 6, 138, 46, 106, 59, 17, 23$. On je interesantan iz nekoliko razloga. Na primer, prvih pet brojeva je deljivo sa 10, a posle nijedan broj nije deljiv sa 10. Prvih deset brojeva je parno, a posle su svi brojevi neparni. Prvih osam brojeva je deljivo sa 6, a posle nijedan broj nije deljiv sa 6. Prva dva broja su deljiva sa 210, a posle nijedan broj nije deljiv sa 210, itd. Pokušaj da pronađeš još ovakvih pravilnosti. Napiši program koji za svaki uneti delilac određuje koliko brojeva je deljivo njime. Smatrati da za svaki uneti delilac važi navedena pravilnost.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 10^5$), a zatim u narednom redu n prirodnih brojeva (manjih od 10^{18}) razdvojenih po jednim razmakom. Nakon toga se do kraja ulaza unose delioci (svaki u posebnom redu). Za svaki delilac se sigurno zna (i to nije potrebno proveravati) da se u nizu nalaze prvo brojevi koji jesu, a zatim brojevi koji nisu deljivi tim deliocem.

Opis izlaza

Za svaki uneti delilac u posebnom redu ispisati broj elemenata niza koji su njime deljivi.

Primer

Ulaz	*Izlaz*
13 210 2310 390 30 510 66 6 138 46 106 59 17 23 10 2 6 2 4 15	5 10 8 10 0 5

Rešenje

Linearna pretraga

Naivno rešenje može biti zasnovano na primeni linearne pretrage (brojanju elemenata filtrirane serije) da bi se odredilo koliko u nizu postoji elemenata deljivih sa učitanim deliocem.

Ako niz ima n elemenata, a postoji m delilaca, složenost ovog rešenja je $O(mn)$. Primetimo da ovo rešenje ni na koji način ne koristi osobinu niza da prvo idu elementi koji su deljivi, a onda elementi koji nisu deljivi datim deliocem.

Binarna pretraga

Zahvaljujući interesantnoj osobini niza, zadatak efikasno može biti rešen primenom algoritma binarne pretrage prelomne tačke. Zaista, po uslovu zadatka u nizu se nalaze prvo svi elementi zadovoljavaju uslov P (deljivi su sa tekućim brojem k), a zatim svi elementi koji ne zadovoljavaju uslov P (nisu deljivi sa tekućim brojem k). Stoga se može primeniti algoritam binarne pretrage prelomne tačke.

Pošto je složenost bibliotečke funkcije binarne pretrage $O(\log n)$, složenost odgovora na svih m upita je $O(m \log n)$.

```
int prviKojiNijeDeljiv(const vector<long long>& a, long long k) {
  int n = a.size();
  int l = 0, d = n-1;
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (a[s] % k != 0)
      d = s - 1;
    else
      l = s + 1;
  }
  return l;
}
```

Rešenje zasnovano na bibliotečkim funkcijama

Zadatak možemo rešiti i pomoću bibliotečkih funkcija za binarnu pretragu, međutim, potrebno je prilagoditi ih zadavanjem funkcije poređenja (kojom se kodira uslov P).

Da bismo našli prvi element koji zadovoljava neki uslov, u jeziku C++ možemo upotrebiti funkciju `upper_bound`, na malo neobičan način. U slučaju pretrage prelomne tačke bitni su nam samo elementi niza, a ne element koji se traži (jer zapravo ne tražimo nikakav konkretan element unutar niza). Zato kao element koji tražimo možemo navesti bilo šta (na primer, nulu). Ključno je definisati funkciju poređenja (koja se prosleđuje kao poslednji argument funkciji `upper_bound`), tako da vraća informaciju o tome da su elementi koji ne zadovoljavaju uslov manji od traženog, dok elementi koji zadovoljavaju uslov nisu manji od traženog. Funkcija poređenja, dakle, treba samo da analizira svoj drugi element i da vrati informaciju o tome da li on zadovoljava uslov (u našem primeru, tražimo prvi element koji nije deljiv brojem d i to je uslov koji se proverava u sklopu funkcije poređenja).

Mogli bismo upotrebiti i funkciju `lower_bound`, ali bi tada u funkciji poređenja bilo potrebno razmeniti redosled argumenata (njoj je tražena vrednost uvek drugi argument) i negirati uslov.

Pošto je složenost bibliotečke funkcije binarne pretrage $O(\log n)$, složenost odgovora na svih m upita je $O(m \log n)$.

```
int prviKojiNijeDeljiv(const vector<long long>& a, long long d) {
    auto it = upper_bound(begin(a), end(a), 0,
                          [d](long long _, long long x) {
                            return x % d != 0;
                          });
    return distance(begin(a), it);

}
```

Zadatak: Dopuna mejlova

Aplikacije za slanje mejlova čuvaju ranije korišćene mejl adrese a onda pomažu korisnicima tako što na osnovu njih dopunjavaju započet unos nove mejl adrese (tzv. opcija automatskog dopunjavanja, engl. aucomplete). Napisati program koji efikasno implementira ovu funkcionalnost.

Opis ulaza

Sa standardnog ulaza se učitava broj mejl adresa n ($1 \leq n \leq 10^5$), a zatim u n narednih redova po jedna mejl adresa. Nakon toga se unosi broj započetnih mejl

adresa koje treba dopuniti m ($1 \leq m \leq 10^5$), a nakon toga i tih m započetih mejl adresa.

Opis izlaza

Na stadardni izlaz za svaku započetu mejl adresu ispisati broj mejl adresa koje je dopunjavaju.

Primer

Ulaz	*Izlaz*	*Objašnjenje*

`9`
`pera@gmail.com`
`lana123@yahoo.com`
`andrija@yahoo.com`
`laza@gmail.com`
`ana.ilic@mail.ru`
`lazica@hotmail.com`
`milica@gmail.com`
`larisa@zoho.com`
`anaconda@python.org`
`3`
`laz`
`milica@`
`a`
	`2`
	`1`
	`3`
		Na primer, početak `laz` dopunjavaju adrese `laza@gmail.com` i `lazica@hotmail.com`.

Rešenje

Linearna pretraga

Zadatak se može rešiti tako što se za svaki prefiks adrese linearnom pretragom celog niza pronađu one adrese koje počinju tim prefiksom.

Ako postoji n elemenata niza adresa, složenost linearne pretrage je $O(n)$, pa ako se ispituje m prefiksa, ukupna složenost je $O(mn)$.

Binarna pretraga

Zadatak se efikasnije može rešiti tako što se adrese sortiraju, a zatim se za svaki prefiks binarnom pretragom pronalaze adrese koje počinju tim prefiksom. Prva takva adresa je najmanja adresa (u leksikografskom poretku) koja je veća ili jednaka od unetog prefiksa. Iako se može pomisliti da se nakon njenog pronalaska, ostale adrese mogu pronaći u petlji koja nastavlja dalje sve dok adrese počinju tim prefiksom, to

rešenje je potencijalno neefikasno (jer može biti puno takvih adresa). Zato je bolje novom binarnom pretragom odrediti prvu adresu koja ne počinje tim prefiksom, što se može uraditi tako što se poslednji karakter tog prefiksa uveća i pronađe pozicija adrese koja je veća ili jednaka od tako transformisanog prefiksa.

Ako postoji n elemenata niza adresa, složenost inicijalnog sortiranja je $O(n \log n)$, a složenost svake od ove dve binarne pretrage je $O(\log n)$, pa ako se ispituje m prefiksa, ukupna složenost je $O((n+m)\log n)$.

```
// za dati sortiran niz mejlova odredjuju se interval
// [donja_granica, gornja_granica) u kom se nalaze svi mejlovi
// koji pocinju datim prefiksom
pair<int, int> dopunePrefiksaMejla(const vector<string>& mejlovi,
                                   const string& prefiks) {
  // odredjujemo poziciju prvog mejla koji je
  // veci ili jednak od datog prefiksa
  auto lb1 = lower_bound(begin(mejlovi), end(mejlovi), prefiks);
  int donja_granica =  distance(begin(mejlovi), lb1);
  // odredjujemo sledeci prefiks uvecavanjem poslednjeg slova
  // datog prefiksa
  // (na primer, za dati prefiks abc sledeci prefiks je abd)
  string sledeci_prefiks = prefiks;
  sledeci_prefiks.back()++;
  // odredjujemo poziciju prvog mejla koji je veci ili jednak
  // od sledeceg prefiksa
  auto lb2 = lower_bound(begin(mejlovi), end(mejlovi),
                         sledeci_prefiks);
  int gornja_granica = distance(begin(mejlovi), lb2);
  return make_pair(donja_granica, gornja_granica);
}
```

Zadatak: Mucajući podniz

Ako je s niska, onda neka s^n označava nisku koja se dobija ako se svako slovo ponovi n puta (npr. $(xyz)^3$ je $xxxyyyzzz$). Napiši program koji određuje najveći broj n takav da je s^n podniz date niske t (to znači da se sva slova niske s^n javljaju u niski t, u istom redosledu kao u s^n, ali ne obavezno uzastopno).

Opis ulaza

U prvom redu standardnog ulaza nalazi se niska s, a u drugom niska t.

Opis izlaza

Na standardni izlaz napiši traženi broj n.

Primer

Ulaz	*Izlaz*
`xyz` `xaxxybyxyxzyzzb`	`3`

Rešenje

Definisaćemo funkciju kojom proveravamo da li je s^n podniz niske t. Jedan način je da eksplicitno formiramo nisku s^n i da primenimo algoritam provere podniske. Malo bolje rešenje je da se algoritam modifikuje tako da se izbegne efektivno kreiranje niske s^n. Funkcija provere prima nisku s, broj n i nisku t, a zatim svaki od karaktera iz s traži n puta unutar niske t.

Linearna pretraga

Kada na raspolaganju imamo funkciju za proveru, tada optimalnu vrednost n možemo naći linearnom pretragom. Ključna opaska je da ako s^n nije podniz t za neko n, onda s^k nije podniz t ni za jedno $k \geq n$. Zato ćemo stepen uvećavati krenuvši od 1 sve dok ne naiđemo na prvu vrednost n za koju s^n nije podniz t i tada ćemo znati da je optimalna vrednost $n - 1$.

Provera da li je niska s^n podniz niske t vrši se u linearnom vremenu u odnosu na zbir dužina niske. Ako je T dužina niske t, vreme za jednu proveru je $O(T)$. Provere se vrše sve dok se ne nađe optimalno n. Ono najviše može biti $\lfloor \frac{T}{S} \rfloor$, gde je S dužina niske s pa je složenost $O(\frac{T^2}{S})$.

```
bool jeMucajuciPodniz(const string& podniz, const string& niz, int n) {
  int i = 0;
  for (char c : podniz) {
    for (int k = 0; k < n; k++) {
      while (i < niz.size() && niz[i] != c)
        i++;
      if (i == niz.size())
```

```
        return false;
      i++;
    }
  }
  return true;
}

int najduziMucajuciPodniz(const string& podniz, const string& niz) {
   int d = 1;
   while (jeMucajuciPodniz(podniz, niz, d))
      d++;
   return d - 1;
}
```

Binarna pretraga

Činjenica da postoji određeni oblik monotonosti u problemu, nam omogućava da traženu optimalnu vrednost nađemo binarnom pretragom. Važi da ako s^n nije podniz t za neko n, onda s^k nije podniz t ni za jedno $k \geq n$, a da ako je s^n podniz t za neko n, onda je s^k podniz t za svako $k \leq n$. Zato se sa porastom n javljaju se prvo one vrednosti za koje uslov važi, nakon koji idu vrednosti za koje uslov ne važi.

Sigurni smo da se optimalna vrednost nalazi u intervalu od 0 pa do $\left\lfloor \frac{|t|}{|s|} \right\rfloor$. Binarnom pretragom pronalazimo prelomnu tačku, tj. najmanju vrednost n takvu da s^n nije podniz t. Primetimo da u ovom slučaju ne pretražujemo vrednosti u nekom nizu, već je niz potencijalnih vrednosti n koji se pretražuje samo implicitan, pa binarnu pretragu implementiramo ručno.

Provera da li je niska s^n podniz niske t vrši se u linearnom vremenu u odnosu na zbir dužina niske. Ako je T dužina niske t, vreme za jednu proveru je $O(T)$. Interval koji se pretražuje je $[0, \frac{T}{S}]$, gde je S dužina niske s, pa je složenost $O(T \log \frac{T}{S})$.

```
int najduziMucajuciPodniz(const string& podniz, const string& niz) {
  int l = 0, d = niz.size() / podniz.size();
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (jeMucajuciPodniz(podniz, niz, s))
      l = s + 1;
```

```
    else
      d = s - 1;
  }
  return d;
}
```

Zadatak: Najkraća podniska koja sadrži sve date karaktere

Grafički dizajner je preuredio nekoliko slova u jednom fontu i želi da svoje promene prikaže klijentu. U dugačkom tekstu je potrebno da odabere najkraći deo (segment uzastopnih slova) koji sadrži sva slova koja je promenio.

Opis ulaza

U prvoj liniji standardnog ulaza nalazi se tekst (jednostavnosti radi pretpostavimo da je sastavljen samo od malih slova engleskog alfabeta) čija je dužina najviše 10^5 karaktera. U drugoj liniji se nalazi skup slova (opet, pretpostavimo malih slova engleskog alfabeta) koje je dizajner promenio (slova su napisana jedno do drugog, bez razmaka i bez ponavljanja).

Opis izlaza

Na standardni izlaz ispisati jedan ceo broj koji predstavlja dužinu najkraćeg dela teksta koji sadrži sve karaktere datog skupa. Ako takav deo teksta ne postoji, ispisati `nema`.

Primer 1

Ulaz	*Izlaz*
`dobardansvimakakoste` `arnk`	`10`

Primer 2

Ulaz	*Izlaz*
`abababababab` `abc`	`nema`

Rešenje

Gruba sila - provera svih podniski

Najdirektniji algoritam bio bi da se razmatraju sve podniske (one su određene indeksima $0 \leq i \leq j < n$), da se za svaku od njih proveri da li je ispravna tj. da li sadrži sve karaktere iz skupa S i da se među svim ispravnim podniskama pronađe najkraća. Ovakvu pretragu grubom silom je veoma jednostavno implementirati.

```
// proverava da li niska na pozicijama [i, j] sadrzi karakter c
bool podniskaSadrziKarakter(const string& niska,
                            int i, int j, char c) {
  for (int k = i; k <= j; k++)
    if (niska[k] == c)
      return true;
  return false;
}

// provera da li niska na pozicijama [i, j] sadrzi sve karaktere
// iz niske S
bool podniskaSadrziSveKaraktere(const string& niska,
                                int i, int j, const string& S) {
  for (char c : S)
    if (!podniskaSadrziKarakter(niska, i, j, c))
      return false;
  return true;
}

// "beskonacno"
const int INF = numeric_limits<int>::max();

// pronalazi duzinu najkrace podniske koja sadrzi sve karaktere
// iz skupa S
int duzinaPodniske(const string& niska, const string& S) {
  // duzina najkrace do sada pronadjene niske koja sadrzi sve
  // karaktere iz S
  int min_duzina = INF;

  // obradjujemo sve pozicije pocetka podniske
  for (int i = 0; i < niska.size(); i++) {
    // preskacemo karaktere koji nisu u skupu S
    // jer najkraca podniska mora da pocne sa karakterom iz S
    if (S.find(niska[i]) == string::npos) continue;

    // obradjujemo sve pozicije kraja podniske koja pocinje na
```

```
    // poziciji i
    for (int j = i; j < niska.size(); j++)
      if (podniskaSadrziSveKaraktere(niska, i, j, S)) {
        int duzina = j - i + 1;
        if (duzina < min_duzina)
          min_duzina = duzina;
        break;
      }
  }
  return min_duzina;
}
```

Proverava se $O(n^2)$ podniski, i za svaku se vrži m linearnih provera, pa se može pokazati da je složenost je $O(n^3 \cdot m)$ gde je n dužina teksta, a m broj karaktera u skupu S. Parametar m ima malu vrednost, ali dimenzija n može biti velika, pa je ovaj algoritam veoma neefikasan.

Gruba sila - optimizacije

Kada podniska određena pozicijama i i j sadrži sve karaktere iz skupa S, onda i sve podniske određene većim vrednostima j takođe sadrže sve karaktere iz tog skupa, a za njih znamo da su duže i nema potrebe razmatrati ih (vršimo odsecanje u pretrazi).

Primer 3.A.6. *Na primer, ako se unutar niske* `xCxxBxxBxxAxxBxxCxxxxBxAxAxCxxxBx` *traži podniska u kome se javljaju slova iz skupa* `ABC`*. Nakon pronalaska niske* `CxxBxxBxxA`*, koja sadrži sve potrebne karaktere, nema potrebe razmatrati njena proširenja nadesno (na primer, podnisku* `CxxBxxBxxAxxB`*).*

Dakle prva pronađena ispravna podniska koja počinje na poziciji i ujedno je i najkraća ispravna podniska pronađena na toj poziciji. Zato je odmah nakon pronalaska prve ispravne pondiske i ažuriranja vrednosti najmanje dužine moguće prekinuti unutrašnju petlju (naredbom `break`).

Ova optimizacija ne popravlja asimptotsku složenost najgoreg slučaja (na primer, u slučajevima kada ne postoji tražena podniska), ali u mnogim slučajevima može dovesti do ubrzanja.

Razmatranje samo pozicija unutar niske na kojima su karakteri iz skupa

Primetimo da tražena najkraća podniska mora da počinje i da se završava karakterom iz skupa S (reći ćemo da su samo ti karakteri relevantni). U suprotnom bi karakteri na početku i na kraju podniske koji nisu u skupu S mogli da budu uklonjeni čime bi se dobila kraća podniska koja bi i dalje sadržala sve karaktere iz skupa S. Takođe, prilikom provere da li određena podniska sadrži sve karaktere iz S, potrebno je analizirati samo one njene pozicije na kojima su karakteri iz S. Zato ćemo u fazi pretprocesiranja samo izgraditi niz pozicija relevantnih karaktera u niski (reći ćemo da su to relevantne pozicije) i zatim ćemo razmatrati samo podniske koji počinju i završavaju se na pozicijama unutar tog niza (na taj način vršimo odsecanje u pretrazi).

Ako se provera da li je pozicija relevantna vrši linearnom pretragom niske `S`, Vreme potrebno za izgradnju niza relevantnih pozicija je $O(n \cdot m)$ (pošto je m veoma mali broj, ovo je praktično linearno tj. $O(n)$, a može se sniziti i na $O(m + n)$ ako bi se skup S predstavio svojom karakterističnom funkcijom – asocijativnim nizom 26 logičkih vrednosti).

Iako se ovim ne snižava složenost najgoreg slučaja (na primer, kada su svi karakteri niske u skupu S), u slučaju da postoji značajan broj karaktera u tekstu koji nisu u skupu S, pretraga se može ubrzati.

Skup relevantnih karaktera tekuće podniske

Proveru da li se svi karakteri iz skupa S javljaju unutar podniske između dve relevantne pozicije i i j možemo izvršiti tako što odredimo skup svih relevantnih karaktera na svim relevantnim pozicijama niske između i i j (uključujući i njih). Svi karakteri iz tako napravljenog skupa će biti u skupu S (jer razmatramo samo relevantne pozicije), pa je umesto ispitivanja da li je taj skup jednak skupu S, dovoljno ispitati samo da li ima isti broj elemenata kao S (a pošto niska S po uslovima zadatka nema ponovljenih karaktera, broj elemenata skupa S jednak je dužini niske S). Izgradnju skupa relevantnih karaktera koji se pojavljuju unutar tekuće niske S možemo vršiti inkrementalno, tako što prilikom svakog povećanja j, tj. prilikom prelaska na novu relevantnu poziciju j dodamo karakter na toj poziciji u taj skup, što zahteva samo konstantno vreme (ako skup karaktera predstavimo nizom ili bibliotečkim skupom, jer je ukupan broj mogućih karaktera samo 26).

Skup karaktera u jeziku C++ možemo predstaviti preko niza 26 logičkih vrednosti i dodatnog brojača koji predstavlja broj karaktera u skupu. Ipak, elegantnija

implementacija se dobija ako upotrebi bibliotečka implementacija skupa (`set` ili `unordered_set`).

```
// "beskonacno"
const int INF = numeric_limits<int>::max();

// pronalazi duzinu najkrace podniske koja sadrzi sve karaktere iz skupa
int duzinaPodniske(const string& niska, const string& S) {
  // pozicije unutar niske na kojima su karakteri iz skupa karakteri
  vector<int> relevantne_pozicije;
  for (int i = 0; i < niska.size(); i++)
    // ako se niska[i] nalazi u skupu karakteri
    if (S.find(niska[i]) != string::npos)
      // zapamti njegovu poziciju i
      relevantne_pozicije.push_back(i);

  int min_duzina = INF;
  for (auto i = relevantne_pozicije.begin(); i != relevantne_pozicije.en
    // relevantni karakteri koje sadrzi trenutna podniska
    set<char> karakteri_podniske;
    for (auto j = i; j != relevantne_pozicije.end(); j++) {
      karakteri_podniske.insert(niska[*j]);
      if (karakteri_podniske.size() == S.size()) {
        int duzina = *j - *i + 1;
        if (duzina < min_duzina)
          min_duzina = duzina;
        break;
      }
    }
  }
  return min_duzina;
}
```

Složenost algoritma koji za svaku poziciju i određuje najkraću ispravnu podnisku koja počinje na poziciji i uz sve navedene optimizacije je $O(n^2)$.

Binarna pretraga

Zahvaljujući inkrementalnosti, proveru da li za datu dužinu podniske k postoji neka podniska koja sadrži sve date karaktere možemo uraditi u linearnom vremenu $O(n)$, u jednom prolasku kroz niz. Koristićemo tehniku pokretnog prozora i prilikom prelaska sa jedne na narednu podnisku, iz skupa ćemo uklanjati prvi karakter tekuće podniske i dodavaćemo poslednji karakter nove podniske. Ovo dodatno možemo ubrzati (doduše ne asimptotski) ako je skup karaktera mali, tako što ćemo prolaziti samo kroz pozicije originalne niske na kojima znamo da se javljaju karakteri iz skupa.

Skup karaktera iz S unutar tekuće podniske ćemo predstaviti pomoću preslikavanja karaktera u njihov broj pojavljivanja. Najjednostavnije je upotrebiti bibliotečku implementaciju mape tj. rečnika, jer nam ona pruža mogućnost efikasnog određivanja broja karaktera iz S unutar podniske (podniska je ispravna tj. sadrži sve potrebne karaktere ako i samo ako je broj karaktere iz S koje sadrži jednak broju elemenata skupa S).

Nakon toga, najmanju dužinu k možemo naći tehnikom binarne pretrage tako što nađemo najmanju vrednost k za koju važi neki uslov. Bitno je naglasiti da problem zadovoljava svojstvo monotonosti tj. da ako za neko k ne postoji podniska dužine k koja sadrži sve karaktere skupa, onda takva podniska ne postoji ni za jedno manje k tj. da ako za neko k postoji takva podniska, onda ona postoji i za svako veće k.

```
// provera da li postoji podniska niske niska duzine k koja sadrzi sve
// karaktere iz S
bool postojiPodniska(const string& niska, const string& S, int k) {
  map<char, int> karakteri;
  for (int i = 0; i < niska.length(); i++) {
    if (i >= k && S.find(niska[i-k]) != string::npos)
      if (--karakteri[niska[i-k]] == 0)
        karakteri.erase(niska[i-k]);
    if (S.find(niska[i]) != string::npos) {
      karakteri[niska[i]]++;
      if (karakteri.size() == S.size())
        return true;
    }
  }
```

```
  return false;
}

// pronalazi duzinu najkrace podniske koja sadrzi sve date karaktere
int duzinaPodniske(const string& niska, const string& karakteri) {
  // binarnom pretragom odredjujemo najmanju duzinu k takvu da u T
  // postoji podniska duzine k koja sadrzi sve karaktere skupa S
  int l = 1, d = niska.length();
  while (l <= d) {
    int s = l + (d - l) / 2;
    if (postojiPodniska(niska, karakteri, s))
      d = s - 1;
    else
      l = s + 1;
  }
  return l;
}
```

Binarnom pretragom se pretražuje interval $[1, n]$, gde je n dužina niske, pa se provera da li postoji podniska neke fiksne koja sadrži sva potrebna slova vrši najviše $\log n$ puta. Ta provera se vrši jednim prolaskom kroz nisku. U svakom koraku vrši se linearna pretraga niske kojom je određen skup karaktera i vrše se operacije nad mapom tj. rečnikom u kom su ključevi karakteri iz skupa S. Ako pretpostavimo da se operacije nad mapom izvršavaju u logaritamskoj složenosti, složenost jedne pretrage je zato $n \cdot m \cdot \log m$, gde je m broj karaktera u S. Pošto je m veoma mali broj, faktor $m \log m$ možemo smatrati i konstantom i ukupnu složenost grubo oceniti sa $O(n \log n)$. Da je m veće, složenost bi se mogla popravljati time što bi se provera pripadnosti karaktera skupu S vršila na efikasniji način (na primer, predstavljanjem S pomoću bibilotečkih kolekcija za predstavljanje skupa ili pomoću niza logičkih vrednosti). Takođe je moguće unapred odrediti pozicije karaktera iz S unutar niske (što ne utiče na asimptotsku složenost najgoreg slučaja, jer svi karakteri u T mogu pripadati S, ali može dosta smanjiti faktor n u svakoj pojedinačnoj proveri).

3.A.7 *Dva pokazivača*

Zadatak: Najbliži par elemenata iz dva niza

Računarski sistem raspoređuje poslove na dva procesora. Za svaki od procesora su poznati poslovi koji se na njemu mogu rasporediti i za svaki od poslova je poznato očekivano vreme izvršavanja. U cilju dobre balansiranosti sistema potrebno je rasporediti ona dva posla čije je vreme izvršavanja što sličnije. Napisati program koji to radi.

Opis ulaza

Sa standardnog ulaza se učitavaju dva niza koji predstavljaju očekivana vremena izvršavanja poslova na prvom i na drugom procesoru. Za svaki niz je zadata prvo dužina n ($1 \leq m \leq 50000$), a zatim u drugom redu n celih brojeva (celi brojevi između 1 i $2 \cdot 10^9$ razdvojeni po jednim razmakom).

Opis izlaza

Na standardni izlaz ispisati najmanju vrednost razlike dva posla koja će biti raspoređena.

Primer

Ulaz	*Izlaz*

`5`
`4680 2120 7940 11530 17820`
`4`
`850 13420 5770 6300`
	`1090`

Objašnjenje
Najmanja razlika se postiže kada se na prvom procesoru pokrene proces čije je vreme izvršavanja 4680, a na drugom čije je vreme izvršavanja 5770.

Rešenje

Gruba sila

Jedan mogući pristup je da odredimo razliku (preciznije, apsolutnu vrednost razlike) u između svakog elementa prvog i svakog elementa drugog niza, pa od tih razlika nađemo najmanju.

Složenost odgovara broju parova i procenjuje se kao $O(m \cdot n)$, gde je m broj elemenata prvog, a n broj elemenata drugog niza.

```
// najmanja razlika a1[i] - a2[j]
int NajmanjaRazlika(const vector<int>& a1, const vector<int>& a2) {
  int minRazlika = numeric_limits<int>::max();
  for (int x1 : a1)
    for (int x2: a2)
      minRazlika = min(minRazlika, abs(x1 - x2));
  return minRazlika;
}
```

Uporedni prolaz kroz sortirane nizove

Efikasniji pristup je da se nizovi najpre sortiraju, a da se zatim istovremeno prolazi kroz oba niza, računajući razliku tekućih elemenata i napredujući u onom nizu u kojem je vrednost trenutno manja, kao kod klasičnog algoritma objedinjavanja dva sortirana niza. Usput se, naravno, po potrebi ažurira najmanja zabeležena razlika. Kada se stigne do kraja bilo kojeg niza, postupak je završen i najmanja zabeležena razlika je tada i ukupno najmanja.

Zaista, pošto su nizovi sortirani, kada se uporede početni elementi iz oba niza, onaj koji je manji od njih nema potrebe upoređivati sa ostalim elementima niza kome on ne pripada, jer će razlika moći biti samo veća (jer je taj niz sortiran). Na taj način vršimo odsecanje, čime dobijamo na efikasnosti. Taj element onda možemo izbaciti iz daljeg razmatranja tako što ćemo u nizu u kom se on nalazi preći na sledeći element. U specijalnom slučaju kada su početni elementi oba niza jednaki, razlika je jednaka nuli, što je najmanja moguća razlika, pa nema potrebe vršiti dalju analizu.

Primer 3.A.7. *Na primer, neka su nakon sortiranja vrednosti jednake sledećim.*

```
1 14 28 33 45
8 21 22 41 56 68
```

- *Prvo poredimo elemente 1 i 8. Razlika je 7. Razlika između broja 1 i svih daljih brojeva u donjem nizu je veća od 7, pa broj 1 ne moramo više analizirati.*
- *Nakon toga poredimo brojeve 14 i 8 i dobijamo razliku 6. Razlika između broja 8 i svih brojeva iza 14 je veća, pa sada ni 8 ne moramo više analizirati.*
- *Poredimo sada brojeve 14 i 21, razlika je 7, a 14 ne moramo više da analiziramo.*
- *I razlika između 28 i 21 je 7, a broj 21 ne moramo više da analiziramo.*

- *Razlika između 28 i 22 je 6, a 22 ne moramo da analiziramo dalje.*
- *Razlika između 28 i 41 je 13, a 28 ne moramo da analiziramo dalje.*
- *Razlika između 33 i 41 je 8, a 33 ne moramo da analiziramo dalje.*
- *Razlika između 45 i 41 je 4, a 41 ne moramo da analiziramo dalje.*
- *Razlika između 45 i 56 je 11, a 45 ne moramo da analiziramo dalje. Pošto nema više elemenata u gornjem nizu, postupak se završava.*

Možemo zaključiti da je najmanja moguća razlika jednaka 4 (za brojeve 41 i 45).

Složenošću dominira sortiranje, koje se izvršava u vremenu $O(m \log m + n \log n)$. Nakon sortiranja, prolazak sa dva pokazivača se izvršava u vremenu $O(m + n)$.

```
// najmanja razlika a1[i] - a2[j]
int NajmanjaRazlika(const vector<int>& a1, const vector<int>& a2) {
  // pravimo sortirane kopije dva niza
  auto a1s = a1;
  sort(begin(a1s), end(a1s));
  auto a2s = a2;
  sort(begin(a2s), end(a2s));
  // algoritmom objedinjavanja odredjujemo najmanju razliku
  int i1 = 0, i2 = 0;
  int minRazlika = numeric_limits<int>::max();
  while (i1 < a1s.size() && i2 < a2s.size())
    if (a1s[i1] <= a2s[i2]) {
      minRazlika = min(minRazlika, a2s[i2] - a1s[i1]);
      i1++;
    } else {
      minRazlika = min(minRazlika, a1s[i1] - a2s[i2]);
      i2++;
    }
  return minRazlika;
}
```

Zadatak: Broj parova date razlike

Napiši program koji određuje na koliko načina možemo da odaberemo dva elementa niza tako da im je razlika jednaka datom broju r.

Opis ulaza

Sa standardnog ulaza se unosi prvo pozitivan prirodan broj r, u narednom redu dužina niza n ($1 \leq n \leq 50000$), a nakon toga elementi niza.

Opis izlaza

Na standardni izlaz ispiši broj parova elemenata niza čija je razlika jednaka r.

Primer

Ulaz	*Izlaz*

`2350`
`5`
`15745 18095 15745 16234 13395`
	`4`

Objašnjenje

Moguće je napraviti parove od prvog i drugog, od prvog i petog, od drugog i trećeg i od trećeg i petog elementa niza.

Rešenje

Gruba sila

Naivan način da se zadatak reši je da se ispitaju svi uređeni parovi elemenata niza i da se prebroje oni čija je razlika jednaka traženoj. Pošto uređenih parova učenika ima n^2, složenost ovakvog algoritma je $O(n^2)$.

Tehnika dva pokazivača

Zadatak možemo rešiti tehnikom dva pokazivača. Niz mora biti sortiran, a oba pokazivača se kreću sleva nadesno. Jednostavnosti radi, pretpostavimo prvo da u nizu nema duplikata.

Primer 3.A.8. *Prikažimo kako bi algoritam radio na primeru određivanja broja elemenata čija je razlika 8 u narednom nizu.*

```
1 3 7 8 11 14 16 30 38
```

- *Krećemo od para `1 3`. Pošto je razlika manja od tražene element `3` ne može biti umanjenik, pa povećavamo umanjenik na `7`.*
- *Analiziramo par `1 7`. Situacija je opet ista, pa opet povećavamo umanjenik na `8`. Naime, pomeranjem umanjioca sa `1` na `3`, razlika bi se samo smanjila, pa `7` zaista ne može biti umanjenik.*

- *Analiziramo par* `1 8`. *I tu je situacija ista, pa opet povećavamo umanjenik na* `11` *(ponovo zaključujemo da se pomeranjem umanjioca sa* `1` *nadesno, na* `3` *ili* `7` *razlika smanjuje, pa* `8` *zaista ne može biti umanjenik).*

- *Analiziramo par* `1 11`. *Ovaj put je razlika veća od tražene. Stoga možemo zaključiti da* `1` *ne može biti umanjilac (daljim pomeranjem umanjenika nadesno, razlika bi se samo povećala). Stoga prelazimo na naredni umanjilac, a to je* `3`. *Ključna napomena je da su razlike svih umanjenika ispred* `11` *i broja* `3` *manje od tražene razlike* `8` *(jer su takve bile razike i kada je umanjilac bio manji)*

- *Analiziramo par* `3 11`. *To je prvi par koji ima datu razliku. Ako su elementi različiti, tada se za sve umanjenike posle* `11` *dobija veća razlika u odnosu na umanjilac* `3`, *pa možemo da pomerimo umanjilac na* `7`. *Slično, umanjenik* `11` *ne može da napravi ni jedan dalji par čija bi razlika bila jednaka traženoj, pa možemo da pomerimo umanjenik na* `14`.

- *Analiziramo par* `7 14`. *Razlika je manja od tražene, pa povećavamo umanjenik na* `16`.

- *Analiziramo par* `7 16` *razlika je veća od tražene, pa pomeramo umanjilac na* `8`.

- *Analiziramo par* `8 16` *čija je razlika jednaka traženoj. Nakon toga možemo povećati i umanjilac na* `11` *i umanjenik na* `17`.

- *Analiziramo par* `11 30` *i pošto mu je razlika veća od tražene, pomeramo umanjilac na* `14`.

- *Analiziramo par* `14 30` *i pošto mu je razlika veća od tražene, pomeramo umanjilac na* `16`.

- *Analiziramo par* `16 30` *i pošto mu je razlika veća od tražene, pomeramo umanjilac na* `30`.

- *Analiziramo par* `30 30` *kome je razlika manja od tražene, pa pomeramo umanjenik na* `38`.

- *Analiziramo par* `38 30` *kome je razlika jednaka traženoj, pa pomeramo i umanjilac i umanjenik. Pošto ne postoji veći umanjenik, algoritam se završava.*

Opišimo formalno prethodni postupak i dokažimo njegovu korektnost. Odredimo koliko parova date razlike r postoji u intervalu $[i, n)$, ako znamo da važi invarijanta da je je $a_{j-1} - a_i < r$. Vršimo inicijalizaciju $i = 0$ i $j = 1$, tako da određujemo broj parova i intervalu $[0, n)$, a invarijanta je zadovoljena jer je $a_{j-1} - a_i = a_0 - a_0 = 0 < r$.

- Ako je $j = n$, tada u intervalu $[i, n)$ ne postoji ni jedan par date razlike. Zaista, na osnovu invarijante važi da je $a_{n-1} - a_i < r$. Pošto je niz sortiran, i povećanjem i i smanjivanjem j razlika se smanjuje. Zato parovi brojeva unutar intervala $[i, n)$ imaju manju razliku od r.

- Ako je $a_j - a_i < r$, tada znamo da u intervalu $[i, j]$ ne postoji ni jedan par brojeva čija je razlika jednaka r (jer je razlika elemenata unutar intervala uvek manja neko razlika krajnjih elemenata). Invarijanta je zadovoljena za par $(i, j + 1)$ pa uvećavamo j i nastavljamo postupak.

- Ako je $a_j - a_i > r$, tada ni jedan par $a_{j'} - a_i$ za $i < j' < n$ nema razliku r. Na osnovu invarijante znamo da je $a_{j-1} - a_i$ manje od r, pa pošto je niz sortiran to važi i za sve elemente $i < j' < j$. Pošto je $a_j - a_i > r$ i pošto je niz sortiran povećanjem j se povećava razlika, pa su razlike za $j \leq j' < n$ veće od r. Zato se u intervalu $[i, n)$ svi eventualni parovi čija je razlika r nalaze u intervalu $[i + 1, n)$ i postupak nastavljamo tako što uvećavamo i za 1. Još moramo dokazati da tada invarijanta važi tj. da je $a_{j-1} - a_{i+1} < r$, međutim to važi jer je niz sortiran i važi $a_i < a_{i+1}$, a na osnovu invarijante je važilo da je $a_{j-1} - a_i < r$.

- Na kraju, ako je $a_j - a_i = r$, tada smo pronašli jedan par. Pošto smo pretpostavili da u nizu nema duplikata i da je niz sortiran, a_i ne može biti član ni jednog drugog para sa razlikom r u intervalu $[i, n)$ - pošto je niz sortiran pomeranjem umanjenika nalevo razlika se smanjuje, a pomeranjem nadesno, ona se povećava. Dakle, svi eventualni parovi čija je razlika r nalaze se u intervalu $[i + 1, n)$. Postupak se može nastaviti uvećavanjem i indeksa i i indeksa j za 1. Zaista, invarijanta je zadovoljena jer je $a_{j+1-1} - a_{i+1} = a_j - a_{i+1} < a_j - a_i = r$.

Složenost ovog pristupa je $O(n \log n)$ zahvaljujući početnom sortiranju, dok je složenost druge faze, nakon sortiranja linearna tj. $O(n)$. Zaista i umanjenik i umanjilac

se kreću u istom smeru (vrednost oba pokazivača se samo uvećava), pa se može napraviti najviše $2n$ koraka.

Pređimo sada na slučaj u kom se elementi u nizu mogu ponavljati.

Obrada duplikata čitanjem serije istih elemenata

Ako se elementi u nizu ponavljaju, onda u trenutku kada nađemo prvi par (i, j) takav da je $a_i - a_j = r$, određujemo broj pojavljivanja n_i elementa a_i i broj pojavljivanja n_j elementa a_j, broj parova uvećavamo za $n_i \cdot n_j$ (jer svako pojavljivanje vrednosti a_i možemo iskombinovati sa svakim pojavljivanjem vrednosti a_j) i nakon toga postupak nastavljamo od indeksa $(i + n_i, j + n_j)$.

```
// broj parova elemenata niza a kojima je razlika jednaka datom broju
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  // pravimo sortiranu kopiju niza a
  auto as = a;
  sort(begin(as), end(as));

  int broj = 0;
  int i = 0, j = 1;
  while (j < as.size()) {
    if (as[j] - as[i] < razlika)
      j++;
    else if (as[j] - as[i] > razlika)
      i++;
    else {
      // pronalazimo sve elemente jednake as[i]
      int ii;
      for (ii = i+1; ii < as.size() && as[ii] == as[i]; ii++)
        ;
      // odredjujemo koliko ih ima
      int broj_ai = ii - i;
      // preskacemo ih
      i = ii;

      // pronalazimo sve elemente jednake as[j]
      int jj;
```

```
      for (jj = j+1; jj < as.size() && as[jj] == as[j]; jj++)
        ;
      // odredjujemo koliko ih ima
      int broj_aj = jj - j;
      // preskacemo ih
      j = jj;

      // uvecavamo brojac za broj parova (as[i], as[j])
      broj += broj_ai * broj_aj;
    }
  }

  return broj;
}
```

Složenošću i dalje dominira sortiranje čija je složenost $O(n \log n)$, dok je složenost druge faze i dalje linearna tj. $O(n)$.

Obrada duplikata prebrojavanjem pojavljivanja

Još jedan način da se reši problem ponavljanja elemenata je da se u prvoj fazi niz vrednosti transformiše u niz parova koji sadrže vrednosti i njihov broj pojavljivanja.

```
// broj parova elemenata niza a kojima je razlika jednaka datom broju
int brojParovaDateRazlike(const vector<int>& a, int razlika) {
  // pravimo sortiranu kopiju niza
  auto as = a;
  sort(begin(as), end(as));

  // odredjujemo broj pojavljivanja svakog elementa
  vector<pair<int, int>> b;
  b.reserve(as.size());
  b.emplace_back(as[0], 1);
  for (int i = 1; i < as.size(); i++) {
    if (as[i] == b.back().first)
      b.back().second++;
    else
```

```
        b.emplace_back(as[i], 1);
    }

    // trazimo elemente cija je razlika jednaka datoj
    int broj = 0;
    int i = 0, j = 0;
    while (j < b.size()) {
      if (b[j].first - b[i].first < razlika)
        j++;
      else if (b[j].first - b[i].first > razlika)
        i++;
      else {
        broj += b[j].second * b[i].second;
        i++; j++;
      }
    }

    return broj;
}
```

Sortiranje je složenosti $O(n \log n)$. Prebrojavanje elemenata se nakon toga izvršava u složenosti $O(n)$, jednim prolaskom kroz niz, nakon čega se sa dva pokazivača prolazi kroz niz opet u složenosti $O(n)$. Ukupna složenost je, dakle, $O(n \log n)$. Implementacija koristi i pomoćni niz, ali memorijska složenost ostaje $O(n)$.

Zadatak: Segment datog zbira u nizu prirodnih brojeva

U datom nizu pozitivnih prirodnih brojeva naći sve segmente (njihov početak i kraj) čiji je zbir jednak datom pozitivnom broju (brojanje pozicija počinje od nule).

Opis ulaza

U prvoj liniji standardnog ulaza nalazi se zadati pozitivan prirodni broj z koji predstavlja dati zbir $0 < z < 10^6$, u drugoj broj elemenata niza, N ($2 \leq N \leq 50000$), a zatim, u narednoj liniji N elemenata niza (pozitivni prirodni brojevi manji od 200).

Opis izlaza

U svakoj liniji standardnog izlaza ispisuju se dva broja (celi brojevi) odvojena prazninom, koji predstavljaju indekse početka i kraja segmenta (brojano od nule). Ako

postoji više traženih segmenata njihove indekse ispisati sortirano na osnovu levog kraja.

Primer

Ulaz	*Izlaz*
`125`	`0 2`
`10 60 40 25 50 50 100 25 35 30 35`	`2 4`
	`5 6`
	`6 9`

Rešenje

Gruba sila

Naivno rešenje grubom silom pretpostavlja da se izračunaju zbirovih svih segmenata i da se proveri da li su jednaki datom broju. Čak i kada zbirove segmenata računamo inkrementalno, dobijamo neefikasno rešenje.

Postoji $O(n^2)$ segmenata, a zbir svakog segmenta na osnovu zbira prethodnog segmenta dobijamo u složenosti $O(1)$, pa je ukupna složenost $O(n^2)$.

Tehnika dva pokazivača

Pretragu segmenata grubom silom možemo korišćenjem odsecanja načiniti mnogo efikasnijom.

Primer 3.A.9. *Posmatrajmo primer pronalaženja prvog segmenta u nizu `1 2 3 5 15 1 2 5` koji ima zbir 21.*

Krećemo da ispitujemo zbirove segmenata koji počinju na poziciji 0*. Sve dok je zbir tekućeg segmenta strogo manji od 21, potrebno je da proširujemo segmente.*

`i`	`a[i]`	`zbir`
`0`	`1`	`1`
`1`	`2`	`3`
`2`	`3`	`6`
`3`	`5`	`11`
`4`	`15`	`26`

U trenutku kada je zbir postao strogo veći od tražene vrednosti 21, sigurni smo da ni jedan segment koji počinje na poziciji 0 ne može imati zbir 21. Naime, pošto su svi dalji elementi striktno pozitivni, njihovim uključivanjem bi se dobio samo još veći zbir. Zbog toga možemo da pređemo na segmente koji počinju na poziciji 1. Važna (i ne

baš trivijalna) opaska je to da svi segmenti koji počinju na poziciji 1 i završavaju se pre tekuće pozicije 4 imaju zbir strogo manji od 21 i stoga ih nije potrebno eksplicitno ispitivati. Naime, svi segmenti koji počinju na poziciji 0, a završavaju se pre tekuće pozicije su imali zbir manji od 21, pa se uklanjanjem elementa na poziciji 0 dobijaju segmenti čiji je zbir još manji. Dakle, prvi kandidat za zbir 21, je segment koji počinje na poziciji 1 i završava se na poziciji 4. Njegov zbir lako dobijamo oduzimanjem početne vrednosti 1 sa pozicije 0 od zbira tekućeg segmenta.

```
i   a[i]   zbir
1   2      2
2   3      5
3   5      10
4   15     25
```

Pošto i taj segment ima zbir veći od 21, takav zbir će imati i svi dalji segmenti koji počinju na poziciji 1, pa možemo preći na segmente koji počinju na poziciji 2. Ponovo je prvi kandidat onaj koji se završava na poziciji 4 (jer svi koji se ranije završavaju sigurno imaju zbir manji od 21).

```
i   a[i]   zbir
2   3      3
3   5      8
4   15     23
```

Ponovo je zbir preveliki, pa prelazimo na segmente koji počinju na poziciji 3. Prvi kandidat je segment koji se završava na poziciji 4.

```
i   a[i]   zbir
3   5      5
4   15     20
```

Ovaj put taj segment ima zbir manji od traženog, pa ga je potrebno proširiti nadesno. Dodavanjem narednog elementa dobijamo segment čiji je zbir jednak traženom.

```
i   a[i]   zbir
3   5      5
4   15     20
5   1      21
```

Nakon ovoga smo sigurni da nema više segmenata traženog zbira koji počinju na poziciji 3, pa prelazimo na poziciju 4 i postupak se po istom principu nastavlja dalje.

Dakle, održavamo tekući segment i njegov zbir. Dok je taj zbir manji od traženog

proširujemo segment nadesno (dok je to moguće), a kada zbir postane veći ili jednak traženom skraćujemo segment sa leve strane.

Dokažimo i formalno da je prethodni postupak korektan. Obeležimo sa $z_{ij} = \sum_{k=i}^{j} a_k$ zbir elemenata niza a čiji indeksi pripadaju segmentu $[i, j]$, a sa z traženi zbir elemenata. Pošto su svi elementi niza a pozitivni, zbirovi elemenata segmenta zadovoljavaju svojstvo monotonosti tj. važi da iz $i < i' \leq j$ sledi $z_{ij} > z_{i'j}$ i da iz $i \leq j < j' < n$ sledi $z_{ij} < z_{ij'}$.

Pretpostavimo da za neki interval $[i, j]$ znamo da za svako j' takvo da je $i \leq j' < j$ važi da je $z_{ij'} < z$. Postoje sledeći slučajevi za odnos z_{ij} i z.

- Prvo, ako je $z_{ij} < z$, tada ni za jedan interval koji počinje na poziciji i, a završava se najkasnije na poziciji j ne može važiti da mu je zbir elemenata z, i proveru je potrebno nastaviti od intervala $[i, j+1]$, uvećavajući j za 1. Ako takav interval ne postoji (ako je $j+1 = n$), onda se pretraga može završiti (jer je i za svako i' takvo da je $i < i' \leq j = n-1$ važi $z_{i'j} < z_{ij} < z$, a zato i za svako j' takvo da je $i' \leq j' < j = n-1$ važi da je $z_{i'j'} < z_{i'j} < z$, tako da za svaki interval $[i', j']$ takav da je $i \leq i' \leq j' < n$ važi da je $z_{i'j'} < z$).

- Drugo, pretpostavimo da je $z_{ij} \geq z$. Ako je $z_{ij} = z$, tada je pronađen jedan zadovoljavajući interval i potrebno je obraditi njegove granice i i j. To je jedini segment koji počinje na poziciji i sa zbirom z. Ako je $z_{ij} > z$, onda takvih segmenata nema. Naime, pošto su svi elementi niza a pozitivni, za svako j'' takvo da je $j < j'' < n$ važi da je $z \leq z_{ij} < z_{ij''}$. Dakle, pretragu možemo nastaviti uvećavajući vrednost i. Za sve vrednosti j' takve da je $i + 1 \leq j' < j$ važi da je $z_{(i+1)j'} < z$. Naime, pošto je $a_i > 0$ važi da je $z_{(i+1)j'} < z_{ij'} < z$. Dakle, na segment $[i + 1, j]$ može se primeniti analiza slučajeva istog oblika kao na interval $[i, j]$.

Još jedan način da obrazložimo korektnost ovog postupka je da posmatramo sve zbirove segmenta.

```
1 3 6 11 26 27 29 34
  2 5 10 25 26 28 33
    3  8 23 24 26 31
       5 20 21 23 28
         15 16 18 23
             1  3  8
```

```
2  7
   5
```

Pošto su elementi niza strogo pozitivni, svaka vrsta ove matrice je strogo rastuća, a svaka kolona je strogo opadajuća. Kada je tekući zbir manji od traženog i svi elementi njegove kolone ispod njega su manji od traženog i oni ne moraju biti razmatrani. Moguće je "precrtati" sve elemente u koloni ispod tekućeg i preći se na razmatranje narednog zbira u tekućoj vrsti (ako on postoji). Kada je tekući zbir veći od traženog, veći su i svi elementi u njegovoj vrsti desno od njega. Moguće je njih precrtati i preći na razmatranje narednog zbira u tekućoj koloni (ako on postoji). Elementi u toj narednoj vrsti levo od tekuće kolone su već ranije precrtani i nije potrebno vraćati se na njih.

Prilikom implementacije održavaćemo interval $[i, j]$ i njegov zbir ćemo izračunavati inkrementalno - prilikom povećanja broja j zbir ćemo uvećavati za a_j, a prilikom povećanja broja i zbir ćemo umanjivati za a_i.

Implementacija sa jednom petljom

Jedan način da se na osnovu prethodne analize napravi implementacija je da se u svakom koraku petlje održavaju dve promenljive i i j i promenljiva $zbir$. Obezbedićemo da pri svakom ulasku u telo petlje važi da promenljiva $zbir$ čuva tekuću vrednost z_{ij} i da je za svako $j' < j$ ispunjeno da je $z_{ij'} < z$. Ako se i i j inicijalizuju na nulu, tada se $zbir$ treba inicijalizovati na a_0, čime se zadovoljava prethodni uslov.

U telu petlje provervamo da li je $zbir$ manji od traženog i ako jeste, uvećavamo vrednost j. Ako j dostigne vrednost n, tada možemo prekinuti petlju i završiti pretragu. Ako je uvećano j manje od n, onda zbir uvećavamo za a_j i prelazimo na naredni korak petlje (uslov koji smo nametnuli da važi pri ulasku u petlju će biti ovim biti zadovoljen).

Ako vrednost promenljive $zbir$ nije manja od tražene vrednosti z proveravamo da li joj je jednaka. Ako jeste, prijavljujemo pronađeni interval $[i, j]$. Zatim prelazimo na obradu narednog intervala tako što zbir umanjujemo za vrednost a_i i i uvećavamo za 1 (uslov koji smo nametnuli da važi pri ulasku u petlju će ovim biti opet zadovoljen).

```
void ispisiSegmenteDatogZbira(const vector<int>& a, int trazeniZbir) {
  // granice segmenta
  int i = 0, j = 0;
```

```
  // zbir segmenta
  int zbir = a[0];
  while (true) {
    // na ovom mestu vazi da je zbir = sum(ai, ..., aj) i da
    // za svako i <= j' < j vazi da je sum(ai, ..., aj') < trazeniZbir

    if (zbir < trazeniZbir) {
      // prelazimo na interval [i, j+1]
      j++;
      // ako takav interval ne postoji, zavrsili smo pretragu
      if (j >= a.size())
        break;
      // izracunavamo zbir intervala [i, j+1] na osnovu
      // zbira intervala [i, j]
      zbir += a[j];
    } else {
      // ako je zbir jednak trazenom, vazi da je
      // sum(ai, ..., aj) = trazeniZbir, pa prijavljujemo interval
      if (zbir == trazeniZbir)
        cout << i << " " << j << endl;
      // prelazimo na interval [i+1, j]
      // izracunavamo zbir intervala [i+1, j] na osnovu
      // zbira intervala [i, j]
      zbir -= a[i];
      i++;
    }
  }
}
```

Implementacija sa ugnežđenim petljama

Još jedna moguća implementacija sadrži dve ugnežđene petlje. U prvoj desni kraj segmenta pomeramo nadesno sve dok ne stignemo do kraja ili dok je zbir tekućeg segmenta manji od traženog. U drugoj levi kraj segmenta pomeramo nadesno sve dok je zbir veći ili jednak od traženog (pri tom proveravajući da li je zbir jednak traženom i ako jeste, prijavljujući pronađeni interval).

```
void ispisiSegmenteDatogZbira(const vector<int>& a, int trazeniZbir) {
  int i = 0, j = 0; // granice segmenta
  int zbir = 0;     // zbir segmenta [i, j-1]
  while (j < n) {
    // prosirujemo segment nadesno dok god je zbir manji od trazenog
    while (j < n && zbir < trazeniZbir) {
      // dodajemo novi element
      zbir += a[j];
      j++;
    }

    // skracujemo interval sve dok je zbir veci od trazenog
    while (zbir >= trazeniZbir) {
      // ako je zbir intervala jednak trazenom ispisujemo
      // pronadjeni interval
      if (zbir == trazeniZbir)
        cout << i <<  " " <<  j - 1 << endl;
      // uklanjamo pocetni element
      zbir -= a[i];
      i++;
    }
  }
}
```

Iako sadrži ugnežđene petlje, ovo rešenje je linearne složenosti u odnosu na dužinu niza tj. složenosti je $O(n)$. Naime, promenljive i i j se u svakom koraku uvećavaju za jedan i nikada se ne umanjuju, tako da je ukupan broj koraka ograničen vrednošću $2n$.

Zadatak: Najkraća podniska koja sadrži sve date karaktere

Ovaj zadatak je ponovljen u cilju ilustrovanja različitih tehnika rešavanja.

Tekst zadatka.

Opis ulaza

U prvoj liniji standardnog ulaza nalazi se tekst (jednostavnosti radi pretpostavimo da je sastavljen samo od malih slova engleskog alfabeta) čija je dužina najviše 10^5

karaktera. U drugoj liniji se nalazi skup slova (opet, pretpostavimo malih slova engleskog alfabeta) koje je dizajner promenio (slova su napisana jedno do drugog, bez razmaka i bez ponavljanja).

Opis izlaza

Na standardni izlaz ispisati jedan ceo broj koji predstavlja dužinu najkraćeg dela teksta koji sadrži sve karaktere datog skupa. Ako takav deo teksta ne postoji, ispisati `nema`.

Primer 1

Ulaz	*Izlaz*
`dobardansvimakakoste` `arnk`	`10`

Primer 2

Ulaz	*Izlaz*
`abababababab` `abc`	`nema`

Rešenje

3.A.8 *Strukture*

Zadatak: Računi

Poreska inspekcija želi da proveri prevare tako što će proveravati bankovne račune na kojima se nalazi tačno neki specifičan iznos (na primer, tačno 100 hiljada dinara). U tom cilju potrebno je implementirati bankovni sistem. Sistem ima najviše k različitih korisnika. Svaki korisnik na početku ima račun sa početnim stanjem 0. Potrebno je podržati dve vrste operacija:

- `ime x` dodaje x dinara na račun osobe sa korisničkim imenom `ime` (x može biti i negativno)
- `provera x` određuje koliko postoji korisnika čiji račun sadrži tačno x dinara

Napisati program koji pordržava izvršavanje n ovakvih operacija.

Opis ulaza

Sa standardnog ulaza se unose brojevi n i k. Nakon toga se u n redova unosi po jedna operacija.

Opis izlaza

Za svaki upit (operaciju prvog tipa) ispisati odgovor, svaki u zasebnom redu.

Primer

Ulaz	*Izlaz*

`6 4`
`marko 2300`
`milan 5200`
`dragana 4000`
`provera 0`
`milan -1200`
`provera 4000`
	`1`
	`2`

Rešenje

Zadatak se jednostavno i lako može rešiti upotrebom dve mape tj. rečnika. Jedna se koristi da preslika ime korisnika u iznos na njegovom računu, a druga da preslika iznos na računu u broj računa koji sadrže taj iznos.

```
// broj upita i broj korisnika
int n, k;
cin >> n >> k;

// iznos na racunu
map<string, int> racun;
// broj korisnika sa datim iznosom novca
map<int, int> brPojavljivanja;

// na pocetku svih k korisnika ima 0 dinara
brPojavljivanja[0] = k;

// obradjujemo sve upite
for (int i = 0; i < n; i++) {
  string s;
  cin >> s >> x;

  if (s == "provera")
    cout << brPojavljivanja[x] << '\n';
  else {
    brPojavljivanja[racun[s]]--;
```

```
    racun[s] += x;
    brPojavljivanja[racun[s]]++;
  }
}
```

Zadatak: Segment datog zbira u nizu celih brojeva

Napiši program koji za dati niz celih brojeva određuje broj nepraznih segmenata uzastopnih elemenata niza čiji je zbir jednak datom broju.

Opis ulaza

Sa standardnog ulaza se u prvoj liniji unosi tražena vrednost zbira z (ceo broj -10000 i 10000), zatim, u narednoj liniji dimenzija niza n ($3 \leq n \leq 50000$) i zatim u narednoj liniji elementi niza (celi brojevi između -100 i 100, razdvojeni razmakom).

Opis izlaza

Na standardni izlaz ispiši broj segmenata čiji je zbir jednak z.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`11`	`7`	Objašnjenje: sledeći segmenti imaju zbir 11
`10`		`1 2 3 5`
`1 2 3 5 1 -1 1 5 3 2`		`1 2 3 5 1 -1`
		`2 3 5 1`
		`1 2 3 5 1 -1`
		`5 1 -1 1 5`
		`1 -1 1 5 3 2`
		`1 5 3 2`

Rešenje

Gruba sila

Direktno rešenje grubom silom podrazumevalo bi da se provere svi segmenti uzastopnih elemenata, da se za svaki izračuna zbir i da se proveri da li je taj zbir jednak traženom. Svi segmenti se mogu nabrojati ugnežđenim petljama, gde spoljna petlja prolazi kroz leve krajeve segmenta (i uzima vrednosti od 0 pa do $n-1$), a unutrašnja petlja prolazi kroz desne krajeve segmenata (od vrednosti i pa do $n-1$).

Složenost ovog pristupa je kubna u odnosu na dimenziju n tj. $O(n^3)$.

Inkrementalno računanje zbira segmenata

Algoritam grube sile koji posebno računa zbir svakog segmenta se može unaprediti ako se zbirovi računaju inkrementalno tj. ako se iskoristi činjenica da se zbir svakog segmenta koji se dobija proširivanjem prethodnog segmenta jednim elementom može lako izračunati na osnovu zbira prethodnog segmenta, tako što se zbir prethodnog segmenta uveća za tekući element niza.

Dakle, opet možemo nabrajati sve segmente ugnežđenim petljama, na početku tela spoljašnje petlje zbir inicijalizujemo na nulu, u unutrašnjoj petlji zbir uvećavamo za element a_j i, ako je on jednak traženom, ispisujemo indekse intervala $[i, j]$.

```
int brojSegmenataDatogZbira(const vector<int>& a,
                            int trazeniZbir) {
  // broj elemenata niza
  int n = a.size();
  // broj segmenata trazenog zbira
  int broj = 0;

  // prolazimo sve segmente
  for (int i = 0; i < n; i++) {
    // zbir segmenta [i, j]
    int zbir = 0;
    for (int j = i; j < n; j++) {
      // izracunavamo zbir segmenta [i, j] na osnovu zbira
      // segmenta [i, j-1]
      zbir += a[j];
      // proveravamo da li je zbir tog segmenta jednak
      // trazenom
      if (zbir == trazeniZbir)
        broj++;
    }
  }

  return broj;
}
```

Ovim je izbegnuta linearna složenost za izračunavanje zbira trenutnog intervala tj. da

se zbir svakog narednog intervala dobija u konstantnom vremenu, tako da je složenost celog algoritma redukovana na kvadratnu tj. $O(n^2)$.

Zadatak se, može rešiti i dosta efikasnije od ovoga.

Zbirovi prefiksa

Jedan elegantan i često primenjivan način da se dobiju zbirovi svih segmenata niza je da se zbir segmenta predstavi kao razlika zbira dva prefiksa niza. U pomoćni niz b (ili u sam niz a, ako je memorija kritičan resurs), inkrementalno, u linearnoj složenosti $O(n)$, na svaku poziciju k od 0 do n možemo smestiti zbir prvih k elemenata niza tj. zbir elemenata segmenta $[0, k)$.

Nakon toga možemo proveriti sve parove prefiksa i videti da li je njihova razlika jednaka traženom zbiru segmenata. Ako proveravamo svaki segment $[i, j]$ za $0 \leq i < j < n$ i za svaki zbir određujemo na osnovu razlike zbirova prefiksa (što možemo uraditi u konstantnom vremenu) dobijamo algoritam kvadratne složenosti $O(n^2)$.

Ovo nije efikasnije od inkrementalnog rešenja grubom silom, ali se uz korišćenje pogodne strukture podataka može unaprediti.

Efikasna pretraga zbirova prefiksa

Izražavanje zbira segmenta u obliku razlike zbira dva prefiksa nam daje mogućnost da stignemo do efikasnijeg rešenja. Problem možemo formulisati i ovako. Za svaki zbir b_{j+1} prefiksa $[0, j + 1)$ potrebno je da pronađemo da li postoji zbir b_i prefiksa $[0, i)$ za neko $i < j$ takva da je $b_{j+1} - b_i = z$, gde je z traženi zbir, tj. da se proveri da li se među zbirovima prethodnih prefiksa nalazi vrednost $b_i = b_{j+1} - z$. Ako se ta pretraga vrši linearno, dolazimo do implementacije veoma slične prethodnoj, koja ispituje svaki par elemenata $i < j$. Pošto među elementima niza može biti i negativnih, zbirovi prefiksa nisu sortirani i ne možemo primeniti ni binarnu pretragu. Ostaje nam, međutim, mogućnost da u nekoj strukturi podataka koja omogućava efikasno pretraživanje čuvamo sve zbirove prefiksa za indekse $i < j$. Ako algoritam organizujemo tako da j uvećavamo od 0 do $n - 1$, tada se na kraju svakog koraka u tu strukturu može dodati i zbir tekućeg segmenta (b_{j+1}). Struktura treba da realizuje pretragu po ključu, tako da je najbolje upotrebiti asocijativni niz (mapu tj. rečnik).

Pošto se u zadatku traži samo određivanje broja segmenata sa datim zbirom, ključevi mogu biti zbirovi prefiksa, a vrednost pridružena svakom ključu može biti broj prefiksa sa tim zbirom. Da se tražila samo provera da li postoji segment sa datim

zbirom, mogli smo umesto asocijativnog niza (mape, rečnika) čuvati samo skup ranije viđenih vrednosti zbirova prefiksa, a da su se eksplicitno tražili svi prefiksi, onda bismo svaki ključ preslikavali u niz vrednosti i takvih da je b_i jednako tom ključu.

Ako računamo da će pretraga biti realizovana u $O(\log n)$ (što je najčešće slučaj ako se koriste strukture podataka zasnovane na binarnim stablima), tada će ukupna složenost ove implementacije biti $O(n \log n)$. Napomenimo da smo dobitak na efikasnosti platili dodatnom memorijom koju smo angažovali, međutim, u ovom scenariju nije potrebno pamtiti učitan niz, tako da memorijska složenost neće biti značajno povećana.

```
// ucitavamo trazeni zbir
int trazeniZbir;
cin >> trazeniZbir;

// zbir prefiksa
int zbirPrefiksa = 0;

// broj segmenata sa trazenim zbirom
int broj = 0;

// broj pojavljivanja svakog vidjenog zbira prefiksa
map<int, int> zbiroviPrefiksa;
// zbir pocetnog praznog prefiksa je 0 i on se za sada pojavio
// jednom
zbiroviPrefiksa[0] = 1;

// ucitavamo elemente niza niz
int n;
cin >> n;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  // prosirujemo prefiks tekucim elementom
  zbirPrefiksa += x;

  // trazimo broj pojavljivanja vrednosti
```

```
  // zbirPrefiksa - trazeniZbir i azuriramo broj
  // pronadjenih segmenata
  auto it = zbiroviPrefiksa.find(zbirPrefiksa - trazeniZbir);
  if (it != zbiroviPrefiksa.end())
    broj += it->second;

  // povecavamo broj pojavljivanja trenutnog zbira
  zbiroviPrefiksa[zbirPrefiksa]++;
}

cout << broj << endl;
```

Zadatak: Josifov problem

Đaci sede u krugu obeleženi brojevima od 0 do $n-1$ i igraju se razbrajalice tako da u svakom brojanju jedan đak ispadne. Brojanje kreće od đaka 0 i svaki m-ti đak ispada. Napiši program koji određuje koji đak će ostatati poslednji.

Opis ulaza

U prvoj liniji standardnog ulaza nalazi se početni broj đaka n ($1 \leq n \leq 10^5$), a u drugom dužina brojalice m ($2 \leq m \leq n$).

Opis izlaza

Na standardni izlaz ispisati broj preostalog đaka.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
8	6	Đaci koji sede u krugu na početku i nakon svakog ispadanja su:
3		0 1 2 3 4 5 6 7
		0 1 3 4 5 6 7
		0 1 3 4 6 7
		1 3 4 6 7
		1 3 6 7
		3 6 7
		3 6
		6

Rešenje

Struktura podataka koja dopušta efikasno izbacivanje elemenata iz sredine je lista (npr. dvostruko povezana). Kružno kretanje po listi možemo ostvariti tako što nakon svakog pomeranja iteratora proverimo da li smo stigli do kraja liste i ako jesmo, iterator ručno ponovo postavimo na početak liste.

```
// efekat postfiksnog uvecavanja iteratora u kruznoj listi tj. efekat it++
// iterator se pomera na sledece mesto, ali se vraca polazna vrednost
list<int>::iterator uvecaj(list<int>& lista, list<int>::iterator& it) {
  list<int>::iterator polazni = it;
  it++;
  if (it == lista.end())
    it = lista.begin();
  return polazni;
}

int josif(int n, int m) {
  list<int> lista;
  for (int i = 0; i < n; i++)
    lista.emplace_back(i);

  auto it = lista.begin();
  while (lista.size() > 1) {
    for (int i = 0; i < m - 1; i++)
      uvecaj(lista, it);
    lista.erase(uvecaj(lista, it));
  }

  return *it;
}
```

Kružnu listu možemo jednostavno realizovati korišćenjem reda. U svakom koraku brojanja jednog đaka sa početka reda ćemo prebaciti na kraj reda. Nakon prebacivanja $m - 1$ učenika, onog koji je na početku reda trajno izbacujemo.

```
int josif(int n, int m) {
  queue<int> red;
  for (int i = 0; i < n; i++)
    red.push(i);
  while (red.size() > 1) {
    for (int i = 0; i < m - 1; i++) {
      red.push(red.front());
      red.pop();
    }
    red.pop();
  }
  return red.front();
}
```

Zadatak: K-ti najveći zbir para elemenata dva niza

Data su dva niza koja sadrže prirodne brojeve. Napiši program koji određuje k-ti najveći zbir koji se može dobiti kada se sabere jedan element prvog i jedan element drugog niza. Voditi računa o memorijskoj efikasnosti programa.

Opis ulaza

Sa standardnog ulaza se učitava broj m ($1 \leq m \leq 5000$), a zatim iz narednog reda m elemenata prvog niza razdvojenih razmakom. Iz narednog reda se učitava broj n ($1 \leq n \leq 5000$), a zatim iz narednog reda n elemenata drugog niza razdvojenih razmakom. Elementi oba niza su prirodni brojevi između 0 i 10^6. Na kraju se učitava broj k ($0 \leq k < mn$).

Opis izlaza

Na standardni izlaz ispisati zbir koji se nalazi na poziciji k u nizu koji bi se dobio kada bi se niz svih zbirova parova jednog elementa prvog i jednog elementa drugog niza sortirao nerastuće (pozicije se broje od nule).

Primer 1

Ulaz	*Izlaz*	*Objašnjenje*

`3`
`1 5 3`
`3`
`6 4 2`
`4`
	`7`	Zbirovi koji se mogu dobiti, poređani od najvećeg ka najmanjeg su $5+6=11, 5+4=9, 3+6=9, 5+2=7, 3+4=7, 1+6=7, 3+2=5, 1+4=5, 1+2=3$, pa se na poziciji 4 nalazi zbir 7.

Primer 2

Ulaz	*Izlaz*
5 5 3 8 6 1 6 1 10 9 7 12 2 9	15

Rešenje

Gruba sila

Rešenje grubom silom podrazumeva da se formira niz svih zbirova, da se sortira nerastuće i da se pročita element sa pozicije k.

Parova elemenata ima $m \cdot n$, pa je memorijska složenost kvadratna i iznosi $O(mn)$. Vremenskom složenošću dominira sortiranje i ona iznosi $O(mn \log (mn))$.

Umesto sortiranja, moguće je primeniti i malo efikasniji algoritam *QuickSelect*, koji pronalazi element na poziciji k i bez sortiranja niza, no time rešenje ne bi značajno bilo unapređeno.

```
// formiramo sve zbirove parova elemenata dva niza
vector<int> zbirovi;
zbirovi.reserve(m*n);
for (int i = 0; i < m; i++)
  for (int j = 0; j < n; j++)
    zbirovi.push_back(a[i] + b[j]);

// sortiramo niz zbirova nerastuce
sort(begin(zbirovi), end(zbirovi), greater<int>());

// ispisujemo k-ti element sortranog niza zbirova
cout << zbirovi[k] << endl;
```

Objedinjavanje sortiranih nizova

Problem se može svesti na problem objedinjavanja nekoliko sortiranih nizova, koji se ne moraju istovremeno čuvati u memoriji. Naime, ako sortiramo oba niza opadajuće, možemo razmatrati sledeće nizove:

$$a_0 + b_0, a_0 + b_1, \ldots, a_0 + b_{n-1},$$
$$a_1 + b_0, a_1 + b_1, \ldots, a_1 + b_{n-1},$$
$$\ldots$$
$$a_{m-1} + b_0, a_{m-1} + b_1, \ldots, a_{m-1} + b_{n-1}.$$

Svi oni su sortirani nerastuće i mogu se objediniti korišćenjem reda sa prioritetom. U početku u red ubacujemo prvi element svake liste tj. sve zbirove oblika $a_i + b_0$. U svakom koraku izbacujemo najveći zbir iz reda i dodajemo u red naredni element liste kojoj on pripada. Da bismo nakon vađenja elementa iz reda mogli dodati naredni element liste kojoj on pripada, pored vrednosti zbira $a_i + a_j$ (na osnovu kojih je red uređen) u redu moramo čuvati i indekse i i j. Zapravo dovoljno je čuvati samo indeks j, jer se na osnovu zbira $z = a_i + b_j$, zbir $a_i + b_{j+1}$ može dobiti kao $z - b_j + b_{j+1}$.

U redu se u svakom trenutku nalazi najviše m elemenata. Pošto se čuvaju i originalni nizovi (da bi se sortirali), memorijska složenost je $O(m + n)$. Ažuriranje reda vrši se k puta. Pošto je složenost inicijalnih sortiranja nizova $O(m \log m + n \log n)$, složenost jednog ažuriranja reda je $O(\log m)$, a važi $k < mn$, vremenska složenost je $O(m \log m + n \log n + mn \log m)$. Složenošću jasno dominira faza objedinjavanja, pa se vremenska složenost može proceniti i samo sa $O(mn \log m)$.

```
// sortiramo nizove opadajuce
sort(begin(a), end(a), greater<int>());
sort(begin(b), end(b), greater<int>());

// objedinjavamo sortirane nizove
// a[i] + b[0], ..., a[i] + b[n-1], za svako i od 0 do m-1
// u redu cuvamo zbirove a[i] + b[j] i pozicije j
priority_queue<pair<int, int>> pq;

// dodajemo u red prvi element svakog niza
for (int i = 0; i < m; i++)
  pq.emplace(a[i] + b[0], 0);

// k puta azuriramo red
for (int K = 0; K < k; K++) {
  // skidamo element sa vrha reda
```

```
  int j, z;
  tie(z, j) = pq.top(); pq.pop();
  // ako lista u kojoj je bio skinuti elemnt nije ispraznjena,
  // u red dodajemo njen naredni element
  if (j + 1 < n)
    pq.emplace(z - b[j] + b[j+1], j+1);
}

// element na poziciji k je trenutno na pocetku reda
cout << pq.top().first << endl;
```

Zadatak: Ažuriranje medijane

U zavodu za statistiku žele da što nepristrasnije procene koja je prosečna plata. Zaključili su da izračunavanje aritmetičke sredine može dati malo iskrivljenu sliku jer nekoliko ljudi sa veoma visokim platama mogu značajno povećati prosek. Zato su odlučili da umesto aritmetičke sredine izračunaju medijanu, koja se dobija tako što se sve plate poređaju u neopadajući niz i onda se uzme središnji element tog niza. Ako u nizu ima paran broj elemenata, onda se za medijanu uzima aritmetička sredina dva središnja elementa. Na primer, medijana niza brojeva 1, 2, 4, 7, 9 je 4 (jer je on središnji), a niza brojeva 1, 2, 4, 5, 7, 9 je 4.5 (jer je to aritmetička sredina brojeva 4 i 5 koji su središnji elementi). Podaci o platama pristižu u zavod, a softver mora da može da u svakom trenutku da podatak o medijani do tada unetih plata.

Opis ulaza

Sa standardnog ulaza se unose linije, sve do kraja standardnog ulaza. Linija ili sadrži slovo `d` i zatim iznos plate razdvojen razmakom (ceo broj), što znači da se unosi podatak o novoj plati ili sadrži slovo `m` što znači da je potrebno na standardni izlaz ispisati podatak o medijani do tada unetih plata. Prva linija sigurno sadrži `d`.

Opis izlaza

Na standardnom izlazu su ispisane tražene medijane, svaka u posebnom redu, zaokružene na jednu decimalu.

Primer

Ulaz	*Izlaz*
`d 5`	`6.0`
`d 7`	`6.5`
`d 6`	
`m`	
`d 8`	
`m`	

Rešenje

Izračunavanje medijane svaki put

Direktan način da se zadatak reši je taj da se sve učitane plate smeštaju u niz (ili još bolje vektor tj. listu, pošto ne znamo unapred koliko će plata biti učitano) i da se svaki put kada se zatraži izračunavanje medijane pozove funkcija koja računa medijanu niza. Ta funkcija može biti zasnovana na sortiranju i čitanju središnjeg elementa (tj. središnjih elemenata kada je niz parne dužine), a postoje i malo efikasnija rešenja od toga.

Složenost takvog rešenja zavisi od složenosti sortiranja koji je $O(n \log(n))$ kada se koristi bibilotečka funkcija sortiranja, tako da je ukupna složenost $O(n^2 \cdot \log(n))$, pri čemu pretpostavljamo da će se medijana računati $O(n)$ puta za niz koji ima $O(n)$ elemenata (tj. da su i broj dodavanja i broj upita za medijanom određeni brojem n).

Medijana se može naći i bez sortiranja celog niza, korišćenjem ideje particionisanja tj. algoritma brze selekcije (QuickSelect).

U jeziku C++ ovo je najlakše ostvariti pomoću funkcije `nth_element`. Recimo i da je u slučaju parne dimenzije funkciju `nth_element` potrebno pozvati dva puta, da bi se odredila dva središnja elementa.

Složenost pronalaženja medijane algoritmom brze selekcije je u najgorem slučaju $O(n)$, što dovodi do ukupnog rešenja složenosti $O(n^2)$.

Dva hipa

Efikasno rešenje se može dobiti ako u svakom trenutku u jednoj (reći ćemo levoj) kolekciji čuvamo sve elemente koji su manji od središnjeg, a u drugoj (reći ćemo desnoj) sve one koji su veći ili jednaki središnjem (ako postoji paran broj elemenata, te dve kolekcije treba da sadrže isti broj elemenata, a ako postoji neparan broj elemenata, desna kolekcija može da sadrži jedan element više). Ako ima neparan broj

elemenata, tada je medijana jednaka najmanjem elementu desne kolekcije, a u suprotnom je jednaka aritmetičkoj sredini između najvećeg elementa leve i najmanjeg elementa desne kolekcije. Svaki novi element se poredi sa najmanjim elementom desne kolekcije i ako je manji ili jednak njemu ubacuje se u levu kolekciju, a ako je veći od njega, ubacuje se u desnu kolekciju. Tada se proverava da li se sredina promenila. Ako se desilo da leva kolekcija ima više elemenata od desne (što ne dopuštamo), najveći element leve kolekcije treba da prebacimo u desnu. Ako se desilo da u desnoj kolekciji ima dva elementa više nego u levoj, tada najmanji element desne kolekcije prebacujemo u levu. Dakle, leva kolekcija treba da bude takva da lako možemo da pronađemo i izbacimo njen najveći element, a desna da bude takva da lako možemo da pronađemo i izbacimo njen najmanji element, pri čemu obe kolekcije moraju da podrže efikasno ubacivanje proizvoljnih elemenata. Jasno je da te kolekcije treba da budu hipovi (u jeziku C++ možemo upotrebiti `priority_queue`) u kojima se najmanja tj. najveća vrednost može očitati u konstantnom vremenu, ukloniti u logaritamskom, isto koliko je potrebno i da se umetne novi element.

Primer 3.A.10. *Prikažimo rad ovog algoritma na jednom primeru.*

- *Na početku su oba hipa prazna.*
- *Pretpostavimo da se na ulazu pojavljuje element 5. Njega ubacujemo u desni hip (jer on može da sadrži jedan element više). Stanje je sada sledeće:*

  levi : $[]$ *desni* : $[5]$

  Medijana je element na vrhu desnog hipa tj. 5.
- *Pretpostavimo sa se na ulazu sada pojavljuje element 6. Pošto je on veći od elementa 5 koji je na vrhu desnog hipa, element 5 prebacujemo u levi, a element 6 ubacujemo u desni hip. Stanje je sada sledeće:*

  levi : $[5]$ *desni* : $[6]$

  Medijana je prosek elemenata na vrhu oba hipa tj. 5,5.
- *Pretpostavimo da se sada na ulazu pojavljuje element 3. Pošto je on manji od elementa 5 na vrhu levog hipa element 5 prebacujemo u desni hip, a element 3 ubacujemo u levi. Stanje je sada sledeće:*

  levi : $[3]$ *desni* : $[5, 6]$

  Medijana je element na vrhu desnog hipa tj. 5.

- *Pretpostavimo da se sada na ulazu pojavljuje element 1. Pošto je on manji od elementa 5 na vrhu desnog hipa, dodajemo ga u levih hip.*

  Stanje je sada sledeće:

  levi : $[1, 3]$ *desni* : $[5, 6]$

  Medijana je prosek elemenata na vrhu oba hipa tj. 4.

- *Pretpostavimo da se sada na ulazu pojavljuje element 8. Pošto je on veći od elementa 5 na vrhu desnog hipa, dodajemo ga u desni hip.*

  Stanje je sada sledeće:

  levi : $[1, 3]$ *desni* : $[5, 6, 8]$

  Medijana je elemenat na vrhu desnog hipa, tj. 5.

Ideja da se umesto u jedne podaci čuvaju u dve strukture podataka često može dovesti do efikasnog rešenja.

```
priority_queue<int, vector<int>, greater<int>> desni;
priority_queue<int, vector<int>, less<int>> levi;

double medijana() {
  if (levi.size() == desni.size())
    return (levi.top() + desni.top()) / 2.0;
  else
    return desni.top();
}

void dodaj(int x) {
  if (desni.empty())
    desni.push(x);
  else {
    if (x <= desni.top())
      levi.push(x);
    else
      desni.push(x);
    if (levi.size() > desni.size()) {
      desni.push(levi.top());
```

```
      levi.pop();
    } else if (desni.size() > levi.size() + 1) {
      levi.push(desni.top());
      desni.pop();
    }
  }
}
```