Sadržaj

Predgovor

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

- "Internet crv"[1] 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 funkionisanjem i bilo je neoperativno nekoliko dana. Ukupna šteta je u to vreme procenjena na nekoliko miliona dolara. Suđenje autoru crva bilo je

[1] 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.

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.

- 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 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.

- 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.

- 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.

1.2 Specifikacija programa

Postupak pokazivanja da je program ispravan naziva se *verifikovanje programa*. U razvijanju tehnika verifikacije 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 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. Kada je poznata specifikacija, potrebno je verifikovati program, tj. dokazati da on zadovoljava specifikaciju.

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 parcijalno 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

- **Proveravanje graničnih vrednosti.** Veliki broj bagova 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.
- **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.

- **Proveravanje pre-uslova.** Da bi neko izračunavanje imalo smisla često je potrebno da važe neki pre-uslovi. Na primer, ako je u kodu potrebno izračunati prosečan broj poena n studenata, pitanje je šta raditi ukoliko je n jednako 0 i ponašanje programa treba da bude testirano u takvim situacijama. Ta ponašanja bi trebalo da budu određena specifikacijom programa. Jedno moguće ponašanje programa je, na primer, da se ako je n jednako 0, vrati 0 kao rezultat. Drugo moguće ponašanje programa je da se ne dozvoli izračunavanje proseka ako je n jednako 0. 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 povratnih vrednosti funkcija koje ukazuju na to da li je funkcija uspešno uradila ono što što je trebalo (na primer, funkcije za alokaciju memorije, 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. Opšti savet je da se uvek proverava povratna vrednost ovakvih funkcija. Takođe, 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. Neka tvrđenja o programu je moguće testirati, dok neka nije. Na primer, tvrđenje “program ima

prosečno vreme izvršavanja 0.5 sekundi" je (u principu) proverivo testovima, pa čak i tvrđenje "prosečno vreme između dva pada programa je najmanje 8 sati sa verovatnoćom 95%". Međutim, tvrđenje "prosečno vreme izvršavanja programa je dobro" suviše je neodređeno da bi moglo da bude testirano. Primetimo da je, na primer, tvrđenje "prosečno vreme između dva pada programa je najmanje 8 godina sa verovatnoćom 95%" u principu proverivo testovima ali nije praktično izvodivo.

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 netrivijalnih 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 ovom metodu 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

U ranim danima računarstva i programiranja, testiranje je često vršeno na naivan i nesistematičan način. U međuvremenu, testiranje, kao jedan od ključnih vidova za osiguranje kvaliteta softvera (eng. quality assurance), razvilo se u dobro utemeljenu i široko podržanu oblast računarstva. Jedan od pravaca unapređivanja procesa

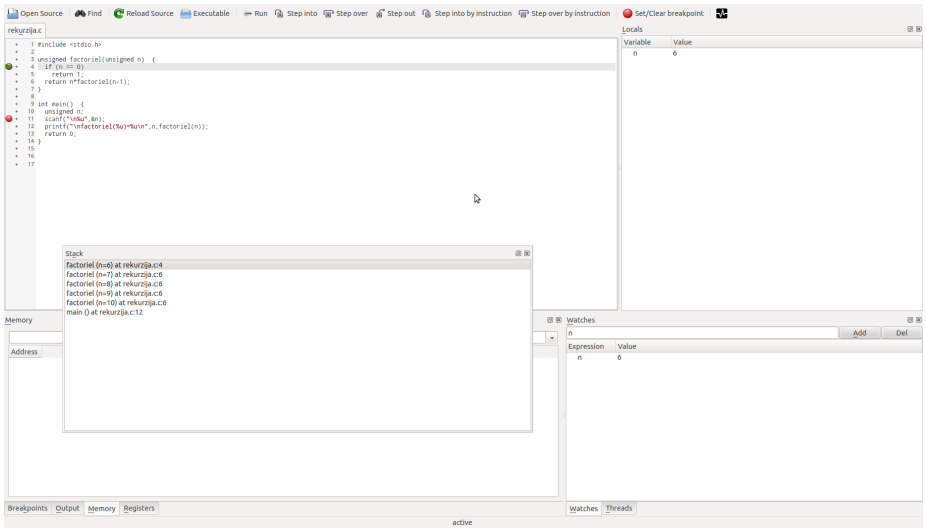
testiranja je njegova automatizacija. Rani vid testiranja je ručno testiranje: neka osoba (tester) sprovodi nizove akcija u okviru programa (bilo unapred precizno definisane, bilo sa nekim faktorom slučajnosti) – unosi podatke, bira opcije, pokreće obrade, itd, a zatim zapisuje sva svoja zapažanja. Ukoliko je u nekom delu programa otkrivena i ispravljena greška – onda će odgovarajući testovi morati da se iznova primenljuju. Štaviše, ukoliko je u programu napravljena neka izmena, takođe će mnogi testovi morati da se vrše iznova, kako bi se osiguralo da u program nije uveden neki bag. Ručno testiranje zato brzo postaje previše zahtevno za čoveka: često se monotono ponavlja i zahteva značajno vreme i koncentraciju. Umesto da testiranje sprovodi 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. Testiranje na početku može da bude ručno i iteracije u tom procesu mogu biti sačuvane za reprodukovanje i automatizaciju u kasnijim fazama. 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. I pored automatizacije, obično ostaje i važan prostor za ručno testiranje, posebno za aspekte korišćenja programa koje nije lako automatizovati (poput subjektivnog osećaja korisnika).

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.

1.3.3 Debagovanje

Pojednostavljeno rečeno, testiranje je proces proveravanja ispravnosti programa, sistematičan pokušaj da se u programu (za koji se pretpostavlja da je ispravan) pronađe greška. S druge strane, debagovanje se primenjuje kada se zna da program ima grešku. Debager je alat za praćenje izvršavanja programa radi otkrivanja konkretne greške (baga, engl. bug). To je program napravljen da olakša detektovanje, lociranje i ispravljanje grešaka u drugom programu. On omogućava programeru da ide korak po korak kroz izvršavanje programa, prati vrednosti promenljivih, stanje programskog steka, sadržaj memorije i druge elemente programa.



Slika 1.1: Ilustracija rada debagera `kdbg`

Slika 1.1 ilustruje rad debagera `kdbg`, koji nudi grafički korisnički interfejs za rad sa debagerom `gdb` koji se često koristi za razvoj programa u GNU/Linux okruženju. Uvidom u prikazane podatke, programer može da uoči traženu grešku u programu. Da bi se program debagovao, potrebno je da bude preveden za *debag* režim izvršavanja. Za to se, u kompilatoru `g++` koristi opcija `-g`. Ako je izvršivi program `mojprogram` dobijen na taj način, može se debagovati navođenjem naredbe:

```
kdbg mojprogram
```

Debageri su danas uglavnom tesno integrisani sa okruženjima za razvoj programa (na primer, orkuženje Visual Studio, ali i editor Visual Studio Code pružaju veoma udobnu podršku za debagovanje C++ programa).

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 verifi-

kacija 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 za 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, 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 samo možemo da tvrdimo 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 direktno, bez daljeg svođenja na probleme manje dimenzije. Ako se prilikom svođenja dimenzija problema uvek smanjuje, konstruisani algoritmi će se uvek zaustavljati.

- 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 direktno 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, algoritam izračunavanja zbira serije brojeva (na primer, zbira elemenata nekog niza) počiva na tome da znamo da izračunamo zbir prazne serije (to je 0) i da ako znamo zbir serije od k elemenata, tada umemo da izračunamo i zbir serije koja se dobija proširivanjem te serije dodatnim $k+1$-vim elementom (to radimo tako što dotadašnji zbir uvećamo za taj novi element).

```
    int zbir = 0;
    for (int i = 0; i < a.size(); i++)
        zbir = zbir + a[i];
```

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 zbira). Baza može odgovarati i slučaju jednočlanog (a ne obavezno praznog) niza, ali tada ne možemo da garantujemo da će algoritam raditi ispravno u slučaju praznog niza. Takvo bazno tvrđenje odgovara varijanti algoritma u kojoj zbir inicijalizujemo na prvi element niza, pa ga uvećavamo redom za jedan po jedan element od pozicije 1 nadalje.

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

```
int zbir(int a[], int n) {
    if (n == 0)
       return 0;
    else
```

```
        return zbir(a, n-1) + a[n-1];
}
```

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 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 1.4.1. *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 elementa 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. □

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 direktno indukcija 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 teorema indukcije se može dokazati na osnovu uobičajene matematičke indukcije po broju rekurzivnih poziva, pod pretpostavkom da se dokaže da se rekurzivna funkcija uvek zaustavlja.

Problem: Definisati funkciju koja vrši efikasno stepenovanje pomoću kvadriranja i dokazati njenu korektnost.

Efikasnu funkciju brzog stepenovanja možemo definisati rekurzivno, prateći sledeća svojstva stepenovanja:

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

```
float stepen_brzo(float x, unsigned k)
{
  if (k == 0)
    return 1.0f;
```

```
  else if (k % 2 == 0)
    return stepen_brzo(x * x, k / 2);
  else
    return x * stepen_brzo(x, k - 1);
}
```

Teorema 1.4.2. *Za svaki realan broj x i prirodan broj $k \geq 0$ važi stepen_brzo$(x, k) = x^k$.*

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, k) zadovoljava neko svojstvo, može se pretpostaviti da za $k \neq 0$ i k parno vrednost *stepen_brzo*$(x \cdot x, k/2)$ zadovoljava to svojstvo, kao i da za $k \neq 0$ i k neparno vrednost *stepen_brzo*$(x, k-1)$ 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.[2]

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

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

[2] 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.

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 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, za vreme i neposredno nakon izvršavanja petlje). Invarijante suštinski opisuju značenje svih promenljivih unutar petlje. Ilustrujmo pojam invarijante na jednom jednostavnom primeru.

Minimum

Razmotrimo sledeću jednostavnu implementaciju algoritma za određivanje minimuma nepraznog niza brojeva.

```
#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 1.4.3. *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. □

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>
```

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 1.4.1. *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.

- 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 1.4.4. *Nakon izvršavanja petlje, promenljiva `m` sadrži minimum celog niza.*

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.

Stepenovanje

Razmotrimo naredni algoritam stepenovanja, tj. kôd kojim se izračunava vrednost x^k i smešta se u promenljivu `m`:

```
int i = 0;
double m = 1.0;
while (i < k) {
  m *= x;
```

```
    i++;
}
```

Lema 1.4.2. *Za svaki prirodni broj $k \geq 0$ i realni broj k, u svakom koraku petlje važi da je $0 \leq i \leq k$ i $m = x^i$ (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 $m = 1$ i $i = 0$, te invarijanta, trivijalno, važi.
- Pokažimo da invarijanta ostaje održana nakon svakog izvršavanja tela petlje. Obeležimo sa m' i m'' i sa i' i i'' vrednosti promenljivih `m` i `i` pre i posle izvršavanja tela i koraka petlje. Pošto se promenljive `x` i `k` ne menjaju, njihove vrednosti obeležimo sa x i k.

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

  Nakon izvršavanja tela petlje, promenljive imaju vrednosti $m'' = m' \cdot x$ i $i'' = i' + 1$. Potrebno je pokazati da ove nove vrednosti zadovoljavaju invarijantu, tj. da važi $m'' = x^{i''}$ i $0 \leq i'' \leq k$. Zaista, $m' \cdot x = x^{i'+1}$ što je tačno na osnovu induktivne pretpostavke. Takođe, pošto važi da je $0 \leq i' < k$, važi i da je $0 \leq i'' \leq k$.

  Dakle, ako su m, i, k i x tekuće vrednosti promenljivih $m = x^i$ i $0 \leq i \leq k$ su invarijante petlje.

□

Pokažimo da dokazana invarijanta obezbeđuje korektnost.

Teorema 1.4.5. *Na kraju izvršavanja algoritma važi da je $m = x^k$.*

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

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) i pripisuje se čuvenom informatičaru Dajkstri (engl. Edsger W. Dijkstra).

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.

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.*

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

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

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

  l           d
1 5 2 6 3 9 4 8
    i

    l         d
1 2 5 6 3 9 4 8
      i

    l         d
1 2 5 6 3 9 4 8
        i

      l       d
1 2 3 6 5 9 4 8
          i

      l     d
1 2 3 6 5 4 9 8
          i

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

```
// funkcija organizuje elemente vektora tako da se prvo nalaze elementi
// za koje vazi da su iz intervala (-Inf, A), nakon toga dolaze
// elementi iz intervala [A, B], i nakon toga elementi iz intervala
// (B, Inf)
void podelaNiza(vector<int>& niz, int A, int B) {
  // - u intervalu pozicija [0, l) su elementi iz intervala (-Inf, A)
  // - u intervalu pozicija [l, i) su elementi iz intervala [A, B]
```

```
  // - u intervalu pozicija [i, d) su jos neispitani elementi
  // - u intervalu pozicija [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 intervala [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]);
  }
}
```

Zadatak: Najmanji broj koji nije zbir elemenata skupa

Dat je skup prirodnih brojeva (zadat u obliku sortiranog niza). Odrediti najmanji prirodan broj koji nije zbir nekih elemenata tog skupa (svaki element skupa može samo jednom učestvovati u zbiru).

Opis ulaza

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

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

Činjenica da su elementi sortirani olakšava rešenje zadatka. Obrađivaćemo element po element i održavaćemo granicu do koje smo sigurni da se svaki broj može predstaviti kao zbir nekog podskupa. Možda malo iznenađujuće, ta granica je u svakom koraku jednaka zbiru svih trenutno učitanih elemenata. Ako je novi učitani element strogo veći od zbira svih prethodnih elemenata uvećanog za jedan, onda se taj uvećani zbir ne može dobiti kao podskup. U suprotnom možemo biti sigurni da se svi brojevi od 0 pa do zbira svih elemenata (u koji je uključen i novi element) mogu dobiti kao zbir nekog podskupa. Naime, pošto je u prethodnom koraku bilo moguće dobiti sve brojeve od 1 do zbira svih elemenata bez tog novog, kada u sve te podskupove uključimo novi element dobićemo sve brojeve od tog novog elementa, pa do zbira svih elemenata sa tim novim elementom.

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;
// sabiranjem elemenata trenutnog skupa mogu se dobiti svi elementi
// iz intervala [0, mozeDo]
int mozeDo = 0;
for (int i = 0; i < n; i++) {
  int x; cin >> x;
  if (x > mozeDo + 1)
    break;
  mozeDo += x;
```

```
}
cout << mozeDo + 1 << endl;
```

Dokažimo korektnost ovog algoritma.

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

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

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

- Ako je $a_i > m + 1$, tvrdimo da je $m + 1$ traženi najmanji broj. Na osnovu invarijante znamo da su svi brojevi iz intervala $[0, m]$ pokriveni, tako da manji broj od $m + 1$ ne može biti rešenje. Dokažimo da broj $m + 1$ ne može biti zbir podskupa. Pošto je niz sortiran, svi elementi od a_i do a_{n-1} su strogo veći od $m + 1$. Dakle, ni jedan od tih elemenata ne sme biti uključen u podskup jer bi njihovim uključivanjem zbir već premašio $m + 1$. Podskup se mora sastojati samo od elemenata a_0 do a_{i-1}, međutim, pošto je m njihov zbir, zbir svakog njihovog podskupa je manji ili jednak m. Dakle, $m + 1$ se ne može postići i on je traženo rešenje.

- Ako je $a_i \leq m + 1$, tada je $m' = m + a_i$, $i' = i + 1$ i tvrdimo da je m' zbir svih elemenata $a_0, \ldots, a_i$ i da se svaki broj iz intervala $[0, m']$ može predstaviti kao zbir nekog podskupa prvih $i' = i + 1$ elemenata niza. Prva tvrdnja je prilično očigledna, jer je po pretpostavci m zbir svih elemenata $a_0, \ldots, a_{i-1}$, a $m' = m + a_i$. Na osnovu pretpostavke znamo da svi brojevi iz $[0, m]$ mogu biti zbirovi podskupova prvih i elemenata niza. Slično i svi brojevi iz intervala $[a_i, a_i + m]$ se mogu dobiti kao zbir nekog podskupa prvih $i' = i+1$ elementa niza. Naime, taj podskup će biti unija elementa a_i i onog podskupa prvih i elemenata niza čiji je zbir jednak razlici između tog broja i broja a_i – on je iz $[0, m]$, pa na osnovu pretpostavke takav podskup postoji. Pošto je $a_i \leq m + 1$ unija intervala $[0, m]$ i $[a_i, a_i + m]$ je

$[0, a_i + m] = [0, m']$. Zato je svaki element iz $[0, m']$ jednak zbiru nekog podskupa prvih i' elemenata niza, pa invarijanta ostaje očuvana.

□

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

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

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 1.4.3. *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 1.4.4. *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 1.4.7. *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. □

1.4.5 *Ispitivanje zaustavljanja programa*

Ne postoji opšti postupak kojim se za proizvoljni zadati program može utvrditi da li se on zaustavlja za zadate vrednosti argumenata[3]. 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[4] 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.[5] 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

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

Binarni zapis

Algoritam koji određuje cifre u binarnom zapisu se zaustavlja za svaki prirodni broj n. Zaista, u svakom koraku petlje vrednost n se menja vrednošću $n \operatorname{div} 10$. Kada je uslov petlje ispunjen, važi $n > 0$, pa je $n \operatorname{div} 10 < n$. Dakle, vrednost promenljive n opada tokom izvršavanja algoritma, a na osnovu invarijante je sve vreme nenegativna, pa u jednom trenutku mora dostići nulu. Tada uslov petlje neće biti ispunjen i algoritam se zaustavlja.

Kolacova hipoteza

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

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

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

Ukoliko se ne zna širina podatka tipa `unsigned int`, nije poznato da li se naredna funkcija zaustavlja za proizvoljnu ulaznu vrednost n:

```
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.

Naravno, ukoliko je poznata širina podatka `unsigned int`, i ukoliko se testiranjem za sve moguće ulazne vrednosti pokaže da se `f` zaustavlja, to bi dalo odgovor na pitanje u specijalnom slučaju.

1.5 Formalno ispitivanje korektnosti programa

1.5.1 Horova logika

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).

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.

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 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 $\{x = 1\}\mathtt{y:=x}\{y = 1\}$[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 $\{x = 1\}\mathtt{y:=x}\{y = 2\}$ govori da će nakon dodele vrednost promenljive y biti jednaka 2 i ona nije tačna.

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

[6] U ovom poglavlju, umesto C-ovske, biće korišćena sintaksa slična sintaksi korišćenoj u originalnom Horovom radu.

formalan način (slika 1.2). Za svaku sintaksnu konstrukciju programskog jezika koji se razmatra formiraju se aksiome i pravila izvođenja koji daju rigorozan opis semantike odgovarajućeg konstrukta programskog jezika.[7]

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 1.2: Horov formalni sistem

Opis aksiome i pravila Horove logike:

- *Aksioma dodele* Ova aksioma definiše semantiku naredbe dodele. Izraz $\varphi[x \to E]$ 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 $\{x + 1 = 2\}\mathtt{y} := \mathtt{x} + \mathtt{1}\{y = 2\}$. Zaista,

[7] 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.).

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 $\{x = 1\}\texttt{y} := \texttt{x} + \texttt{1}\{y = 2\}$. 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 $\{x = 1\}\texttt{y} := \texttt{x}\{y = 1\}$, moguće je dobiti tačnu trojku $\{x = 1\}\texttt{y} := \texttt{x}\{y > 0\}$. Slično, na primer, ako program `P` zadovoljava $\{\top\}\texttt{P}\{z = x \cdot y\}$, tada će zadovoljavati i trojku $\{x \geq 0 \wedge y \geq 0\}\texttt{P}\{z = x \cdot y\}$.

- *Pravilo kompozicije* Pravilom kompozicije opisuje se semantika sekvencijalnog izvršavanja dve naredbe. Kako bi trojka $\{\varphi\}P_1; P_2\{\psi\}$ bila 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 $\{x = 1\}\texttt{y} := \texttt{x} + \texttt{1}\{y = 2\}$ i $\{y = 2\}\texttt{z} := \texttt{y} + \texttt{2}\{z = 4\}$, može da se zaključi $\{x = 1\}\texttt{y} := \texttt{x} + \texttt{1}; \texttt{z} := \texttt{y} + \texttt{2}\{z = 4\}$.

- *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 invarijanta 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.

Stepenovanje

Dokažimo korektnost algoritma stepenovanja. Potrebno je dokazati sledeću trojku.

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

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

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

Dovoljno je, dakle, da dokažemo trojku

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

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:

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

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:

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

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

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

1.5.2 *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.

```
// metod Max 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
  // naredna dva svojstva obezbeđuju da je max najveći element niza arr
  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 (ovim se obezbedjuje da je nakon petlje
    // i jednako dužini niza arr
    invariant 1 <= i <= arr.Length
    // naredna invarijanta tvrdi da 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, videćemo da se procene ponašanja mogu dati razmatrajući samo algoritam koji se koristi. Loši algoritmi za neki težak problem praktično su neupotrebljivi čak iako bi se koristili računari koji su za nekoliko redova veličina efikasniji nego ovi današnji. Stoga se često umesto o efikasnosti programa govori o složenosti algoritama tj. ovi termini se koriste 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. Na primer, programi često moraju trošiti više memorije da 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 češće ograničavajući faktor i u nastavku ćemo se češće baviti analizom vremenske 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 stotina milisekundi izračunavanja 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, slika prosečne veličine hiljadu puta hiljadu piksela zapravo sadrži milion piksela, pa svi algoritmi obrade slika zapravo barataju sa ogromnom količinom podataka i moraju koristiti veoma efikasne algoritme).

Iako su spori programi praktično neupotrebljivi, u razvoju softvera se ponekad preporučuje da je ipak 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 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.

U nastavku teksta najčešće će se govoriti o vremenskoj složenosti algoritama, ali u većini situacija potpuno analogna razmatranja mogu se primeniti na prostornu

složenost.

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 ponašanja može izvršiti i eksperimentalnim putem tj. pokretanjem programa i merenjem utrošenih resursa.

2.1.1 Merenje i procenjivanje utrošenog vremena

Najjednostavnija mera vremenske efikasnosti programa (ili neke funkcije) 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. Ovako dobijeno vreme nije vreme utrošeno za tekući proces nego apsolutno vreme koje je proteklo između dva merenja.

```
#include <iostream>
#include <chrono>

#define BROJ_POZIVA 1000

using namespace std;

int f(void)
{
   ...
}
```

```
int main() {
  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() << " mikrosekundi " << 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. Nekoliko takvih opisano je u dodatku.

2.1.2 *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 velikom 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.

2.1.3 *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`:

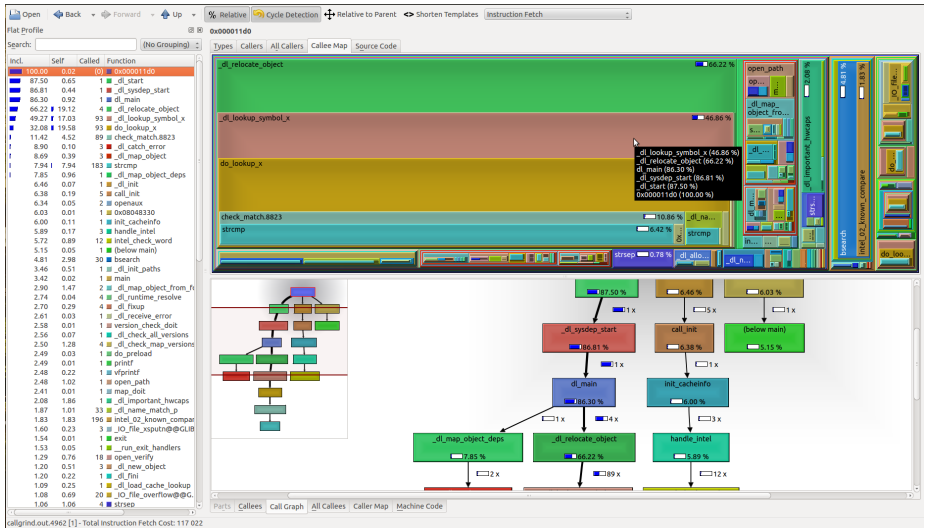
```
kcachegrind callgrind.out.4873
```

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.

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

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. Dodatno, ponašanje programa često ne zavisi od konkretnih vrednosti koje obrađuje, već samo od ukupne količine podataka. U navedenom primeru, ponašanje programa ne zavisi od konkretnih ocena koje su učenici dobili, već samo od broja učenika. Zato složenost algoritma



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

često izražavamo u funkciji *veličine (dimenzije) njegovih ulaznih parametara*, a ne samih vrednosti parametara.

Za veličinu ulaza može se uzeti broj ulaznih elemenata koje treba obraditi (na primer, dužina niza) ili broj bitova potrebnih za zapisivanje ulaza. Ponekad se za veličinu ulaza uzima i vrednost brojeva koje program treba da obradi (na primer, kod sabiranja velikih brojeva vreme potrebno za sabiranje zavisi od vrednosti brojeva koji se sabiraju), mada je pogodnije veličinom ulaza smatrati broj njihovih cifara (što je u direktnoj vezi sa brojem bitova koji su potrebni za njihov zapis). Da bi se izbegla zabuna, uvek je potrebno eksplicitno navesti u odnosu na koju veličinu ulazne vrednosti se razmatra složenost.

Neki 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 naj-

goreg 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 precizno 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.

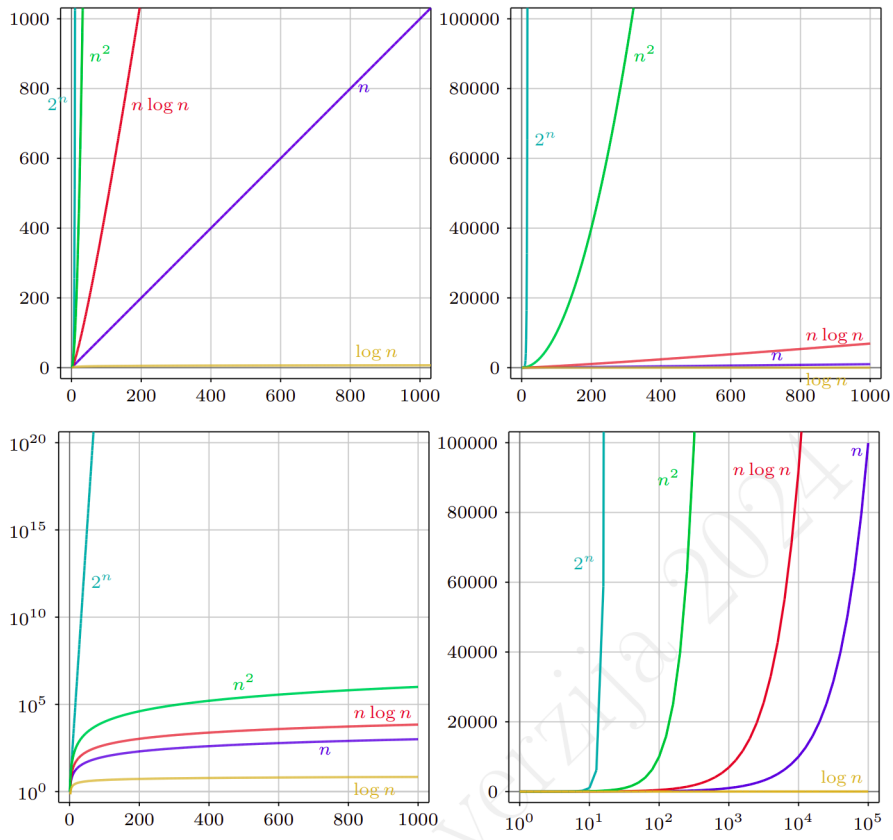
- Analiza najboljeg slučaja je, naravno, previše optimistična i nikada nema smisla.

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 serije operacija može da varira i kada se dešava da vreme potrošeno za prvo 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). Ako se obezbedi da se proširivanje niza i realokacija memorije dešavaju dovoljno retko, ovakve strukture podataka imaju povoljnu amortizovanu složenost.

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

2.2.1 *Zavisnost između vremena izvršavanja i veličine ulaza*

Neka je funkcija $f(n)$ 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. U analizi mnogih algoritama, funkcija $f(n)$ izražava se kao neka kombinacija logaritamske funkcije $\log(n)$, polinomskih funkcija n, n^2, n^3, ..., eksponencijalnih funkcija 2^n, 3^n, ..., faktorijelne funkcije $n!$ i slično. Podsetimo se, zato, osnovnih matematičkih svojstava ovih funkcija.



Slika 2.2: Grafici funkcija koji se često javljaju u analizi složenosti, prikazani na različitim skalama

Priroda funkcija u matematici lakše se proučava i razume kada su dati njihovi gra-

fici. 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 2.2 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 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.

Vratimo se sada na konktekst analize realnih algoritama na savremenim računarima. Tabela 2.1 prikazuje potrebno vreme izvršavanja algoritma ako se pretpostavi da jedna instrukcija traje jednu nanosekundu – 10^{-9} sekundi, tj. 0.001 mikrosekundi

(μs). Ova procena je gruba, ali nije previše pogrešna i daje dobru procenu realnih vremena na današnjim računarima.

Tabela 2.1: Vreme izračunavanja.

$n/f(n)$	$\log n$	$\sqrt{n}$	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	

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

$n/f(n)$	2^n	$n!$
10	1 μs	3,63 ms
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	$n!$
$1ms$	10^6	$63,000$	$1,000$	100	20	9
$10ms$	$10 \cdot 10^6$	$530,000$	$3,200$	215	23	10
$100ms$	$100 \cdot 10^6$	$4,5 \cdot 10^6$	$10,000$	465	27	11

t	n	$n \log n$	n^2	n^3	2^n	$n!$
$1s$	10^9	$40 \cdot 10^6$	$32,000$	$1,000$	30	12
$1min$	$60 \cdot 10^9$	$1,9 \cdot 10^9$	$245,000$	$3,900$	36	14

Pokušajmo sada da podatke iz ove tabele predstavimo grafički (slika 2.3). Usled jako velike razlike u brzinama rasta analiziraćemo samo "susedne" funkcije.

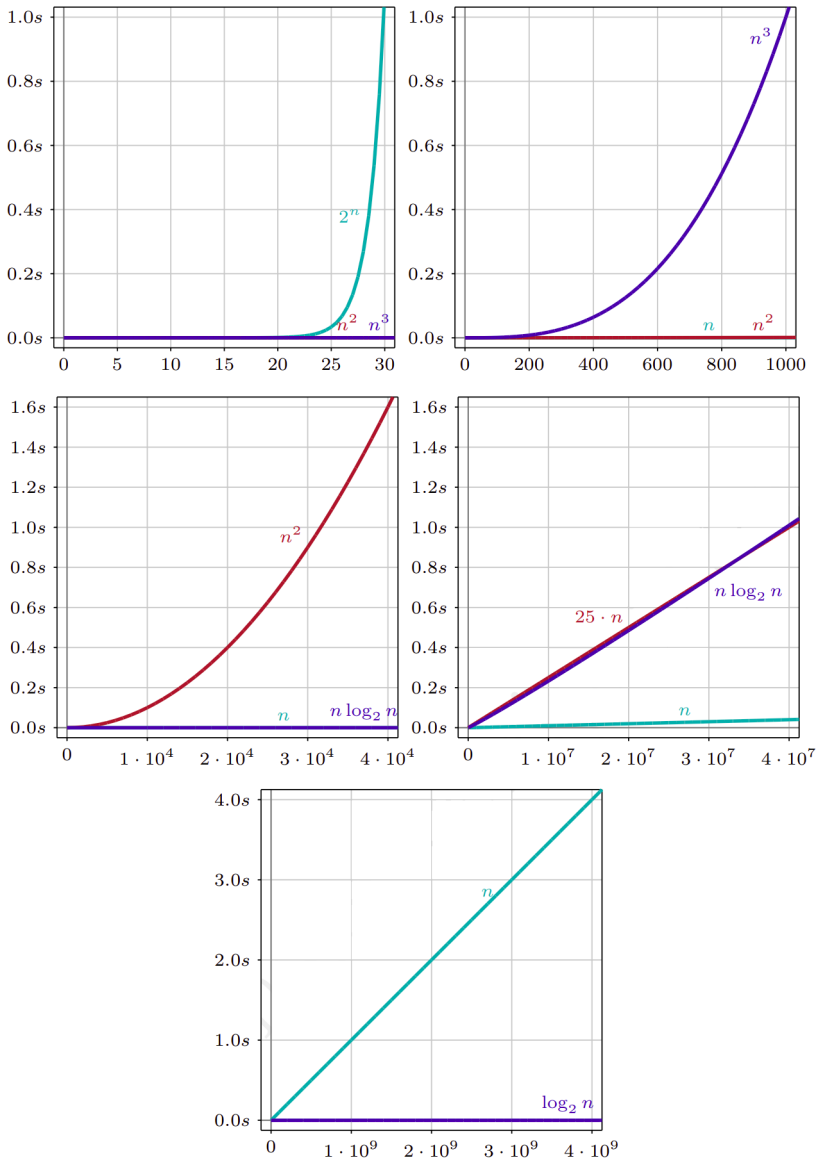
Na prvom grafiku na slici 2.3 prikazan je odnos vremena izvršavanja eksponencijalnih i polinomskih algoritama. Već za dimenziju 30 eksponencijalni algoritam zahteva oko jedne sekunde, dok je vreme izvršavanja polinomijalnih algoritama praktično zanemarivo (čak i u slučaju polinoma većeg stepena, poput n^3).

Na drugom grafiku prikazan je odnos brzina rasta polinomskih algoritama. Povećanjem stepena prati jako veliki porast vremena. ako je algoritam koji zahteva n^3 instrukcija mnogo sporiji nego onaj koji zahteva n^2 instrukcija (kubnom je već za problem dimenzije oko 1000 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 nikakve primetne razlike između algoritama kojima je potrebno $n \log n$ i n instrukcija – oba praktično momentalno 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 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.



Slika 2.3: Grafička ilustracija vremena izvršavanja: na x-osi je dimenzija problema, a na y-osi je vreme u sekundama

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).

2.2.2 *Asimptotske oznake O, Ω i Θ*

Prilikom analize složenosti algoritama obično nam je potrebno da grubo procenimo brzinu rasta funkcija koje opisuju zavisnost potrebnog vremena i memorije u odnosu na veličinu ulaza. 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.

U svim dosadašnjim analizama smo pretpostavili da je broj koraka tj. vreme izvršavanja tačno određeno nekom od funkcija prikazanih na graficima i u tabeli. Međutim, to je retko slučaj – broj koraka obično se izračunava kao neka kombinacija tih funkcija. Lako se može zaključiti da u tim slučajevima vodeći član u funkciji $f(n)$ određuje potrebno vreme izvršavanja. Tako, na primer, ako je broj instrukcija jednak $n^2 + 2n$, onda za ulaz veličine $1\,000\,000$, član n^2 odnosi 16,7 minuta dok član $2n$ odnosi samo dodatne dve milisekunde. Vremenska (a i prostorna) složenost je, dakle, skoro potpuno određena "vodećim" (ili "dominantnim") članom u izrazu koji određuje broj potrebnih instrukcija. Na upotrebljivost algoritma ne utiču mnogo ni multiplikativni i aditivni konstantni faktori u broju potrebnih instrukcija, koliko asimptotsko ponašanje broja instrukcija u zavisnosti od veličine ulaza. Za ovakav pojam složenosti veoma je važna sledeća definicija.[1]

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*

$$f(n) \leq c \cdot g(n), \quad \forall n \geq n_0$$

onda pišemo $f(n) = O(g(n))$ i čitamo "f je veliko o od g".

[1] Pojmovi "veliko o" 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.

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 odnos između funkcija, a ne njihovih konkretnih vrednosti, pod zapisom $f(n) \in O(g(n))$, zapravo podrazumevamo $f \in O(g)$ (jer su f i g funkcije, a $f(n)$ i $g(n)$ njihove vrednosti). O je zapravo najpreciznije smatrati relacijom između dve zadate funkcije f i g.

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

Lako se pokazuje da aditivne i multiplikativne konstante ne utiču na klasu kojoj funkcija pripada (na primer, u izrazu $5n^2+1$, za konstantu 1 kažemo da je aditivna, a za konstantu 5 da je multiplikativna). Drugim rečima, ako $f(n)$ pripada klasi $O(g(n))$, onda $a \cdot f(n) + b$ takođe pripada klasi $O(g(n))$. Zato se može reći da je neki algoritam složenosti $O(n^2)$, ali se obično ne govori da pripada, na primer, klasi $O(5n^2 + 1)$, jer aditivna konstanta 1 i multiplikativna 5 nisu od suštinske važnosti. Zaista, ako jedan algoritam zahteva $5n^2 + 1$ instrukcija, a drugi n^2, i ako se prvi algoritam izvršava na računaru koji je šest puta brži od drugog, on će biti brže izvršen za svaku veličinu ulaza. No, ako jedan algoritam zahteva n^2 instrukcija, a drugi n, ne postoji računar na kojem će prvi algoritam da se izvršava brže od drugog za svaku veličinu ulaza.

Primer 2.2.1. *Može se dokazati da važi:*

- $n^2 = O(n^2)$
  Tvrđenje važi jer $c = 1$ (ali i za veće vrednosti c), važi $n^2 \leq c \cdot n^2$ za sve vrednosti n veće od 0.
- $n^2 + 10 = O(n^2)$
  Za $c = 2$, važi $n^2 + 10 \leq c \cdot n^2$ za sve vrednosti n veće od 3 (jer za takve vrednosti n važi $n^2 \leq n^2$ i $10 \leq n^2$).
- $5n^2 + 10 = O(n^2)$
  Za $c = 6$, važi $5n^2 + 10 \leq c \cdot n^2$ za sve vrednosti n veće od 3 (jer za takve vrednosti n važi $5n^2 \leq 5n^2$ i $10 \leq n^2$).
- $7n^2 + 8n + 9 = O(n^2)$
  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 $9 \leq n^2$).

- $n^2 = O(n^3)$
  Za $c = 1$, *važi* $n^2 \leq c \cdot n^3$ *za sve vrednosti* n *veće od* 0.
- $7 \cdot 2^n + 8 = O(2^n)$
  Za $c = 8$, *važi* $7 \cdot 2^n + 8 \leq c \cdot 2^n$ *za sve vrednosti* n *veće od* 2 *(jer za takve vrednosti* n *važi* $7 \cdot 2^n \leq 7 \cdot 2^n$ *i* $8 \leq 2^n$*).*
- $2^n + n^2 = O(2^n)$
  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).*
- $2^n + 2^n n = O(2^n n)$
  Za $c = 2$, *važi* $2^n + 2^n n \leq c \cdot 2^n n$ *za sve vrednosti* n *veće od* 0 *(jer za takve vrednosti* n *važi* $2^n \leq 2^n n$ *i* $2^n n \leq 2^n n$*).*

Teorema 2.2.1.

- *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* $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

- *Ako važi* $f_1(n) = O(g_1(n))$ *i* $f_2(n) = O(g_2(n))$, *onda važi i* $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

Oznaka O nam daje gornje ograničenje brzine rasta funkcije (čitamo ga kao "ne raste brže od"). Oznaka Ω se koristi da bi se dalo donje ograničenje brzine rasta funkcije (čitamo ga "ne raste sporije od"), dok oznaka Θ označava da su brzine rasta dve funkcije jednake. Definišimo precizno i ove dve oznake.

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*

$$c \cdot g(n) \leq f(n), \quad \forall n > n_0$$

onda pišemo $f(n) = \Omega(g(n))$ *i čitamo "*f *je veliko Omega od* g*".*

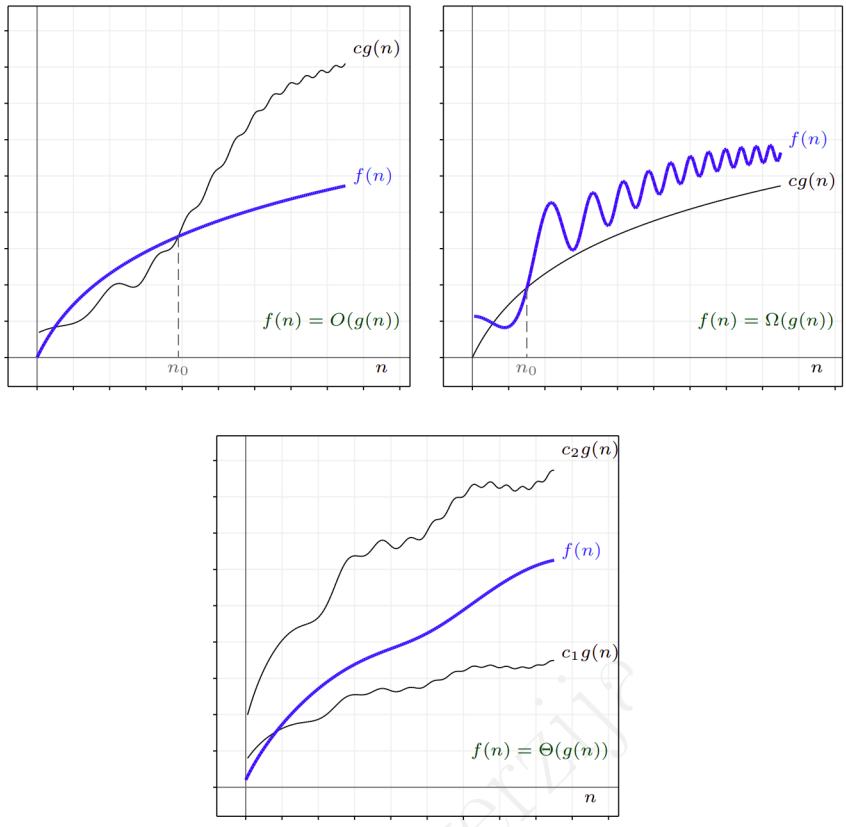
Definicija 2.2.4. *Ako postoje pozitivne realne konstante c_1 i c_2 i prirodan broj n_0 takvi da za funkcije f i g nad prirodnim brojevima važi*

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \quad \forall n > n_0$$

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 2.4.



Slika 2.4: Ilustracija pojmova "veliko o", "veliko omega" i "veliko teta"

Primer 2.2.2. *Može se dokazati da važi:*

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

  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.
- *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 pripada klasama $O(1)$, $\Omega(1)$ i $\Theta(1)$. Štaviše, svaka funkcija koja je ograničena odozgo nekom konstantom pripada klasama $O(1)$ i $\Theta(1)$. Treba imati na umu da i velike konstantne vrednosti kao, na primer, 1,000,000 ili 2^{32} pripadaju klasi $O(1)$.

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 (na primer, n u $O(n)$ ili m i k u $O(2^{m+k})$) zavisi od samog algoritma. Složenost algoritama zavisi od veličine ulaza koja se nekada izračunava brojem podataka koji se obrađuju, a nekad (posebno u teorijskim analizama složenosti) se brojem bitova potrebnih da se zapiše ulaz. Na primer, složenost funkcije koja računa prosek k ulaznih brojeva ne zavisi od vrednosti tih brojeva, već samo od toga koliko ih ima i jednaka je $O(k)$. Ipak, složenost se nekada izražava i u zavisnosti od same ulazne veličine. Na primer, složenost funkcije za izračunavanje faktorijela ulazne vrednosti m zavisi od m i jednaka je (za razumnu implentaciju) $O(m)$. Složenost izračunavanja neke funkcije može da zavisi i od više parametara. 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. 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). Dakle, potrebno je uvek eksplicitno navesti u odnosu na koju veličinu ili koje veličine se razmatra složenost algoritma.

Složenost izračunavanja je izuzetno važna i za praktične i za 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 od-

govor **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 *Izračunavanje složenosti*

Izračunavanje (vremenske i prostorne) složenosti algoritama zasniva se na određivanju 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, u ovom kontekstu pojam "jedinična instrukcija" se ponekad pojednostavljuje, pa se smatra da sve pojedinačne naredbe (bez 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). Zbog toga se u asimptotskoj analizi složenosti obično i ne broje pojedinačne naredbe, već se pretpostavlja da one troše konstantno vreme za izvršavanje.

- 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 $O(f)$ i $O(g)$[2], koji se izvršavaju jedan za drugim, ukupna složenost je $O(f+g)$[3].

  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 primer 2.2.6 i poglavlje 2.3.2).

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

[3] 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 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 $O(f(1) + f(2) + ... + f(n))$.

  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 $O(f + g)$.[4]

[4]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).

- 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 su smeštene na stek, a ne u registre procesora).

  Ukoliko se javljaju rekurzivni pozivi, onda u prostornu složenost ulazi maksimalni broj stek okvira koji se mogu naći na programskom steku tokom izvršavanja (više o tome u glavi 4) i čija je veličina konstantna.[5]

  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.

Prikažimo sada kroz nekoliko primera analizu složenosti iterativno implementiranih algoritama. Da bismo 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 ovog udžbenika.

Primer 2.2.3. *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;
```

[5] 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.

```
    }
  }
}
```

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 2.2.4. *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 2.2.5. *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; j++)
          // 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; j++)
          // 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 apsolutno 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.

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

Primer 2.2.6. *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,* $T(n) = b + T(n-1) = b + b + T(n-2) = \ldots = \underbrace{b + b + \ldots + b}_{n} + T(0) = bn + a$ *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 2.3.2.*

2.2.3.1 *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).

2.2.4 *Popravljanje vremenske efikasnosti programa*

Ukoliko performanse programa nisu zadovoljavajuće, 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 modifikovanti 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.

- **Koristiti optimizacije kompilatora.** Moderni kompilatori i u podrazumevanom režimu rada generišu veoma efikasan kôd ali mogu da se posebno podese tako da primene i dodatne tehnike optimizacije. Korišćenje ovih tehnika veoma značajno može da poboljša efikasnost programa (one najčešće utiču na konstantni faktor, ali postoje primeri kada se vremenska ili memorijska složenost mogu poboljšati i asimptotski). U najvećem broju slučajeva korišćenje naprednih optimizacija je poželjno, ali ono može dovesti i do određenih poteškoća.

  Nakon naprednih optimizacija izvorni k^od transformiše se u asembler na netrivijalan način, što narušava vezu između izvornog i izvršivog programa. Zbog toga je optimizovane programe teško ili nemoguće analizirati degaberom i profajlerom.

  Optimizacijama koje pruža kompilator mogu se dobiti ubrzanja i od nekoliko desetina procenata, ali treba imati na umu da postoji i mogućnost da optimizovani program radi sporije nego neoptimizovani (jer neka tehnika optimizacije u tom konkretnom slučaju ne daje dobre rezultate).

  Proces kompilacija sa intenzivnim optimizovanjem može biti znatno sporiji od osnovne kompilacije. Zato se preporučuje da se optimizovanje programa primenjuje tek u završnim fazama razvoja programa — nakon intenzivnog testiranja i otklanjanja svih otkrivenih grešaka, ali i da se testiranje ponovi sa optimizovanom verzijom izvršivog programa.

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:

- `-O0` je podrazumevani režim kompilacije, samo sa bazičnim tehnikama optimizacije.
- `-O1` kompilator pokušava da smanji i veličinu izvršivog programa i vreme izvršavanja, ali ne primenjuje tehnike optimizacije koje mogu da bitno uvećaju vreme kompilacije.
- `-O2` kompilator primenjuje tehnike optimizacije koje ne zahtevaju dodatni memorijski prostor u zamenu za veću brzinu (u fazi izvršavanja programa).
- `-O3` kompilator primenjuje i tehnike optimizacije koje zahtevaju dodatni memorijski prostor u zamenu za veću brzinu (u fazi izvršavanja programa).
- `-Oc` kompilator primenjuje tehnike optimizovanja koje smanjuju izvršivi program, a ne njegovo vreme izvršavanja.

- **Optimizovati samo bitne delove programa.** Programeri često nemaju jasnu predstavu koji delovi programa troše najviše resursa. Takođe, nešto što se na prvi pogled učini kao korisna optimizacija može zapravo biti potpuno beskorisna transformacija programa, koja program čini komplikovanijim, nerazumljivijim, težim za održavanje i podložnijim greškama. Stoga je tokom optimizovanja jako bitno koristiti merenje vremena izvršavanja na pažljivo odabranim test-primerima i koristiti profajler koji može da ukaže na delove programa koji ima najviše smisla optimizovati. Optimizacije za koje se empirijski ne pokaže da doprinose poboljšanju efikasnosti programa obično nema smisla primenjivati (pogotovo ako komplikuju program). Ukoliko je merenjem potvrđeno da neki deo programa neznatno utiče na njegovu efikasnost – ne vredi unapređivati ga, bolje je da on ostane u jednostavnom i lako razumljivom obliku. Uvek se treba usredsrediti na delove programa koji troše najveći udeo vremena.

  Ilustrujmo ovaj savet jednim Lojdovim (Samuel Loyd, 1841-1911) 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 gr. 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 20% početnog.

Gledano drugačije, ako se funkcija f ubrza tako da se izvršava dvostruko brže, vreme izvršavanja programa se sa t smanjuje na $49,5\%t + 1\%t = 50,5\%t$ tj. i ceo program se izvršava skoro dvostruko brže. S druge strane, ako se (samo) funkcija g ubrza tako da se izvršava dvostruko brže, vreme izvršavanja programa se sa t smanjuje na $99\%t + 0,5\%t = 99,5\%t$ tj. efekat ubrzanja na vreme izvršavanja celog programa je praktično zanemariv.

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%."

- **Izdvojiti izračunavanja koja se ponavljaju.**

  Identično skupo izračunavanje ne bi trebalo da se ponavlja. U sledećem primeru:

```
x = x0*cos(0.01) - y0*sin(0.01);
y = x0*sin(0.01) + y0*cos(0.01);
```

  vrednosti funkcije cos i sin se u originalnom programu izračunavaju po dva puta za istu vrednost argumenta. Ove funkcije su vremenski zahtevne i bolje je u pomoćnim promenljivama sačuvati njihove vrednosti pre korišćenja. Dakle, umesto navedenog, donekle je bolji naredni k^od:

```
cs = cos(0.01);
sn = sin(0.01);
x = x0*cs - y0*sn;
```

```
y = x0*sn + y0*cs;
```

Ipak, ovo je jedna od osnovnih tehnika optimizacije koje kompilator primenjuje i sasvim je realno očekivati da će kompilator ovu optimizaciju sam izvršiti. Štaviše, pošto se radi o konstantnim izrazima, njihova vrednost biće izračunata već u fazi kompilacije. Recimo i da ovakva optimizacija u opštem slučaju nije trivijalna, jer iako je `0.01` konstantni izraz, što se lako može primetiti, potrebna je napredna analiza kojom se može zaključiti da funkcije `sin` i `cos` nemaju propratnih efekata i da će vratiti uvek istu vrednost za isti argument. Takva provera može biti ekstremno teška ili nemoguća za neke funkcije.

Kompilator gcc vrši ove optimizacije već na prvom nivou optimizacije (`-O1`), što se može proveriti u generisanom asemblerskom programu. Ovakva optimizacija daje nezanemariv dobitak samo ukoliko se modifikovani k^od izvršava veliki broj puta.

- **Izdvajanje koda izvan petlje.** Ova preporuka je u istom duhu kao prethodna. Na primer, od koda:

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

bolji je naredni k^od:

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

```
}
```

Još jedan tipičan primer neefikasnosti ovog tipa dolazi iz programskog jezika `C`, u kom se može sresti k^od oblika:

```
for (i = 0; i < strlen(s); i++)
  if (s[i] == c)
    ...
```

Kroz petlju se prolazi n puta, gde je n dužina niske `s`. Međutim, u svakom prolasku kroz petlju se poziva funkcija `strlen` za argument `s` izračunava dužinu niske. Složenost te funkcije je linearna u odnosu na dužinu niske `s` (neka je to n). Zbog toga je složenost ove petlje (barem) $O(n^2)$.

Bolja verzija je sledeća:

```
len = strlen(s);
for (i = 0; i < len; i++)
  if (s[i] == c)
    ...
```

U ovj se verziji koda funkcija `strlen()` poziva se samo jednom i složenost tog poziva i petlje koja sledi zajedno može da bude $O(n)$. Dakle, ovo je primer optimizacije koja ne menja samo konstantni faktor, već snižava asimptotsku složenost programa.

Naprednom analizom bi se moglo utvrditi i da će niska `s` biti konstantna tokom izvšavanja petlje iz prve verzije koda, tako da je moguće da će savremeni kompilatori izdvojiti poziv funkcije `strlen` ispred petlje.

Recimo i da je najbolji način da se svi karakteri niske u jeziku C obrade pomoću petlje narednog oblika, u kojoj se uopšte ni ne poziva funkcija `strlen` (jer se sve niske završavaju specijalnim karakterom `'\0'`):

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

- **Zameniti skupe operacije jeftinim.** Poželjno je izraze zameniti izrazima koji su im ekvivalentni a koriste jeftinije operacije i efikasnije se izračunavaju. Na primer, uslov $\sqrt{x_1x_1 + y_1y_1} > \sqrt{x_2x_2 + y_2y_2}$ je ekvivalentan uslovu $x_1x_1 + y_1y_1 > x_2x_2 + y_2y_2$, ali je u programu daleko bolje umesto uslova `sqrt(x_1*x_1+y_1*y_1) > sqrt(x_2*x_2+y_2*y_2)` koristiti `x_1*x_1+y_1*y_1 > x_2*x_2+y_2*y_2`, jer se njime izbegava pozivanje veoma skupe funkcije `sqrt`. Slično, ukoliko je moguće dobro je izbegavati skupe trigonometrijske funkcije, umesto "većih" tipova dobro je koristiti manje, poželjno celobrojne, itd.

- **Ne vršiti u fazi izvršavanja izračunavanja koja se mogu obaviti ranije.** Ukoliko se tokom izvrša-va-nja programa koriste vrednosti iz malog skupa, uštedu može da donese njihovo izračunavanje unapred i uključivanje rezultata u izvorni k^od programa. Na primer, ako se u nekom programu koriste vrednosti kvadratnog korena od 1 do 100, te vrednosti se mogu izračunati unapred i njima se može inicijalizovati konstantni niz. Ovaj pristup prihvatljiv je ako je kritični resurs vreme a ne prostor.

- **Napisati kritične delove k^oda na asembleru.** Savremeni kompilatori generišu veoma kvalitetan k^od. Ukoliko se koriste i raspoložive optimizacije, k^od takvog kvaliteta može da napiše retko koji programer. Tradicionalno, u nekim situacijama, za neke vremenski kritične delove programa, jedna opcija je bila pisanje tih delova programa na asembleru (što, naravno, podrazumeva jako dobro poznavanje asemblerskog programiranja i specifičnosti hardvera na kom će se program izvršavati). U današnjem svetu kada se sve više programira na većim nivoima apstrakcije, u višim programskim jezicima i okruženjima koji se oslanjaju na bogate biblioteke gotovog koda, ova tehnika se sve ređe i ređe koristi (osim u nekim slučajevima sistemskog programiranja).

- **Izvršiti paralelizaciju izračunavanja.** U današnje vreme sasvim je realno da na raspolaganju imamo veliki broj procesora i računara na kojima je moguće paralelno vršiti neko izračunavanje. Iako pisanje programa koji se mogu izvršavati na taj način zahteva specifične programerske veštine i poznavanje specijalizovanih biblioteka, ono je sve češće opcija.

Naglasimo još jednom da su moderni kompilatori u stanju da samostalno primene izmene koda slične nabrojanim u mnogim situacijama. 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.

2.2.5 *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 k^od 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.

2.3 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.

2.3.1 Sumiranje

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

2.3.1.1 Aritmetički niz

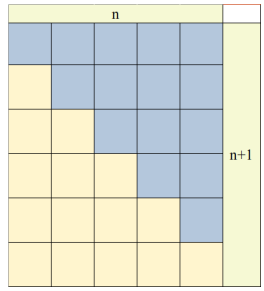
Gausu se pripisuje da je još kao dete izračunao da je

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

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 **??**).

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



Slika 2.5: Gausova formula

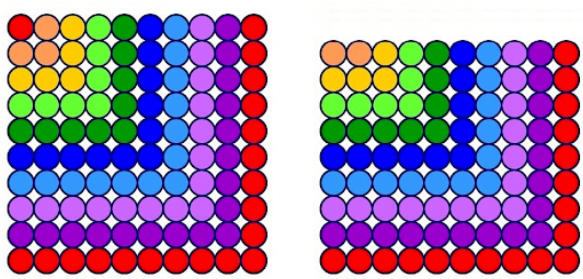
$$a+(a+r)+(a+2r)+...+(a+(n-1)\cdot r) = \sum_{k=0}^{n-1}(a+k\cdot r) = n\cdot a+r\frac{n(n-1)}{2}.$$

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 $2k$

2.3.1.2 *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 $q = 1/2$.

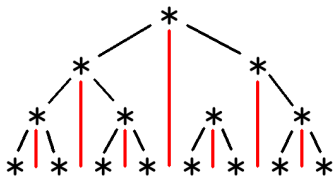
Na osnovu prethodne formule, za $a = 1$ i $q = 2$, važi da je $1 + 2 + ... + 2^{n-1} = 2^n - 1$. Ova formula ima interesantno tumačenje. Suma sa leve strane predstavlja ukupan broj čvorova na prvih n nivoa potpunog binarnog drveta (videti sliku 2.7), dok je izraz sa desne strane za jedan manji od broja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

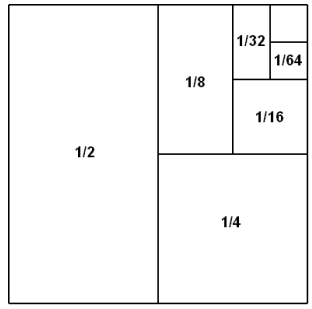
$$1 + 1/2 + ... + (1/2)^{n-1} = \frac{1 - (1/2)^n}{1 - 1/2} = 2 - (1/2)^{n-1}.$$

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 $a = 1/2$, $q = 1/2$

2.3.1.3 *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

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

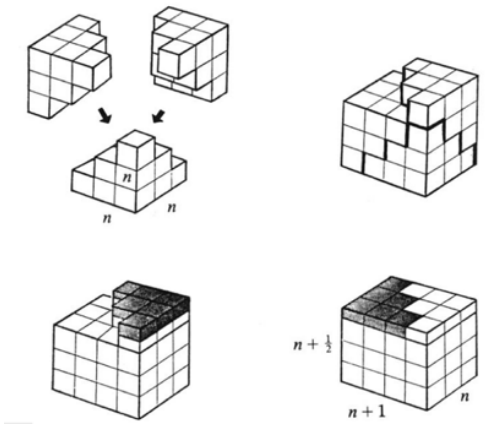
tj.

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

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).

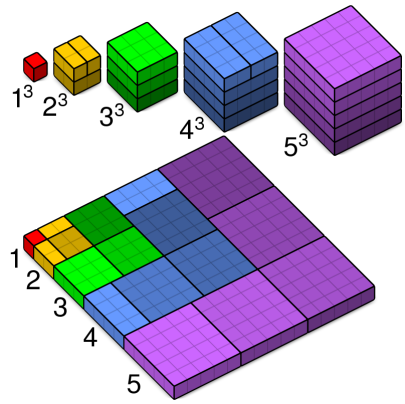
Pošto će nas u analizi algoritama najčešće zanimati samo asimptotsko ponašanje funkcija, najvažnije je zapamtiti da se suma n p-tih stepena svih prirodnih brojeva od 1 do n asimptotski ponaša kao $\frac{n^{p+1}}{p+1}$.

2.3.1.4 *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

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

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 $\log 1\cdot...\cdot n = \log n!$. 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 $T(n) = c+f(1)+f(2)+...+f(n)$, tako da se određivanje asimptotskog ponašanja svodi na sumiranje.

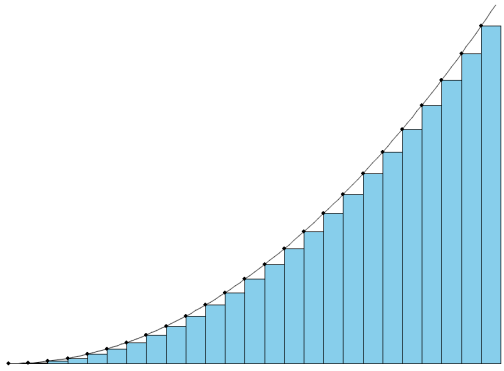
- Zbir n sabiraka reda $1+2+...+n$ ima vrednost $n(n+1)/2$, koja je reda $\Theta(n^2)$. To je samo duplo manje od vrednosti zbira $n+...+n$, koji se sastoji od n sabiraka i ima vrednost n^2.
- Zbir n sabiraka $\log 1+...+\log n$ ima vrednost koja se ponaša kao $n\log n-n$, što je asimptotski isto kao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

2.3.1.5 *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.



Procena sume određenim integralom

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

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

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

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

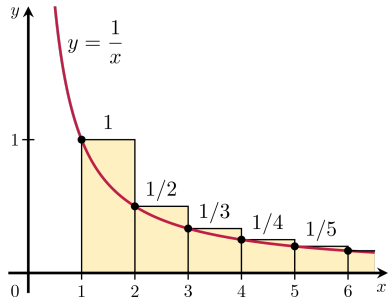
Lako se pokazuje da su dobijeni izrazi i izraz dobijen primenom diferenciranja jednaki. Zamenom vrednosti $x = 2$ ponovo dobijamo da je $S_2(n) = \sum_{k=0}^{n-1} k2^k = (n-2) \cdot 2^n + 2$.

2.3.1.6 *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. $\ln(n+1)$.

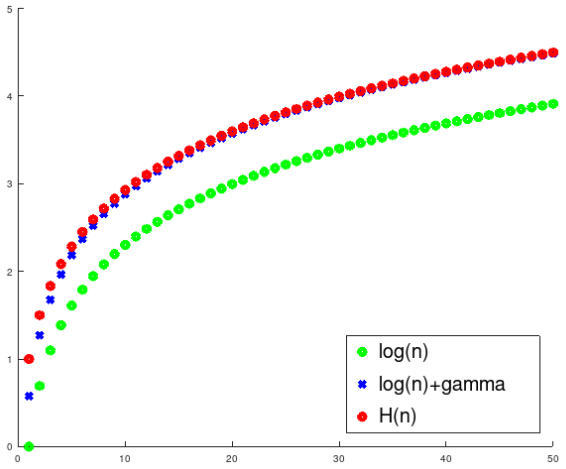


Procena zbira harmonijskog reda $H(n)$ određenim integralom

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.

2.3.2 *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



Zbir harmonijskog reda $H(n)$ i odnos sa logaritamskom funkcijom $\ln n)$

rezultata. Tako se, obično jednostavno, može zapisati *linearna rekurentna relacija* oblika:

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

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 $(1 + \sqrt{5})/2$.

2.3.2.1 ***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.

2.3.2.2 *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 2.3.1. *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 2.3.2. *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.*

2.3.2.3 *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.

2.3.2.4 *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 2.3.3. *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*

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

2.3.2.5 *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 2.3.4. *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 $T(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)$.

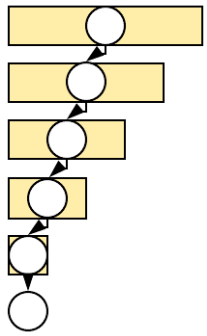
2.3.2.6 *Najznačajniji primeri*

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:

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



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.

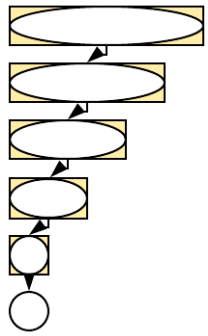
Jednačina $T(n) = T(n-1) + O(n)$**,** $T(0) = O(1)$

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 $T(n) = \Theta(n^2)$.



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

2.3.2.7 *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.

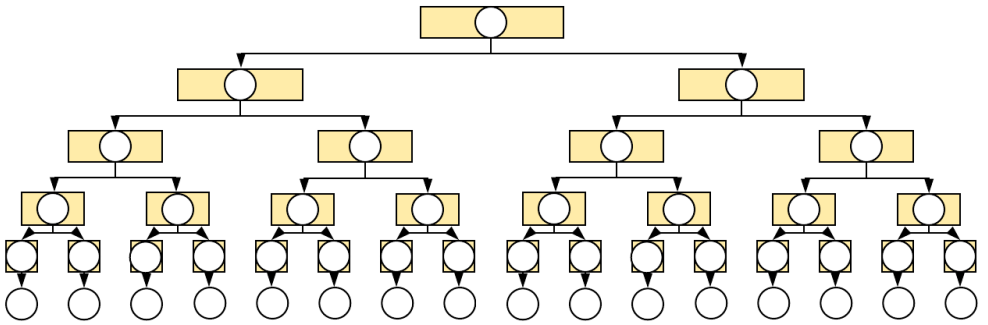
U pitanju je nehomogena jednačina prvog reda i ona se može rešiti svođenjem na homogenu jednačinu. Pretpostavimo da je jednačina $T(n) = 2T(n-1) + c$, $T(0) = d$. Važi $T(n+1) = 2T(n) + c$, pa je $T(n+1) - T(n) = 2T(n) - T(n-1)$, tj. $T(n+1) = 3T(n) - 2T(n-1)$. Karakteristična jednačina je $t^2 = 3t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

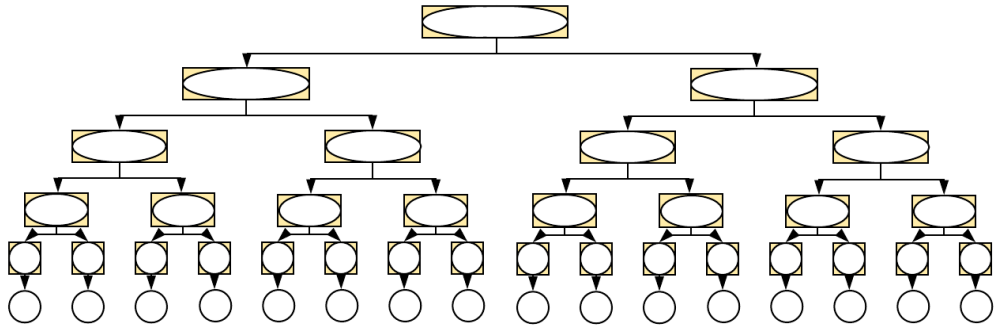


Drvo poziva u slučaju $T(n) = 2T(n-1) + O(1), T(0) = O(1)$ za $n = 5$

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.

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



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

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

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

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

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

2.3.2.8 *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

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

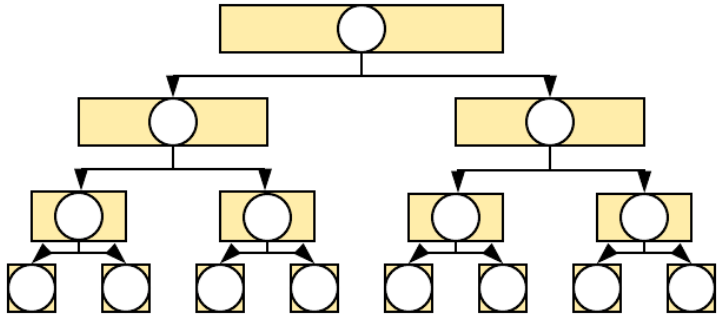
Pre nego što damo dokaz ove teoreme, pokušajmo da kroz niz primera damo intuitivno objašnjenje zašto ova teorema važi, u sva tri svoja slučaja.

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).



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

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

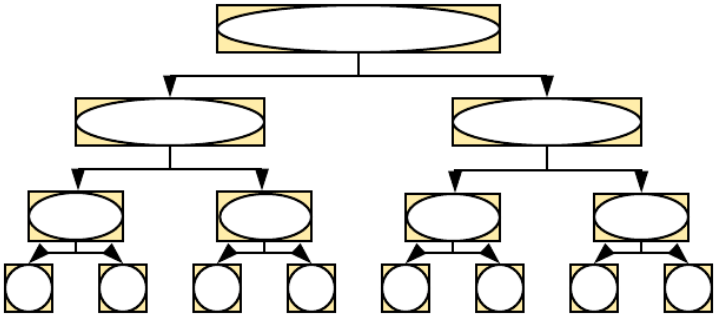
U drugom slučaju su broj čvorova i posao koji se radi na neki način uravnoteženi. Razmotrimo, na primer, jednačinu $T(n) = 2 \cdot T(n/2) + c \cdot n, T(1) = O(1)$ i ponovo pokušajmo da je odmotamo.

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

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.

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



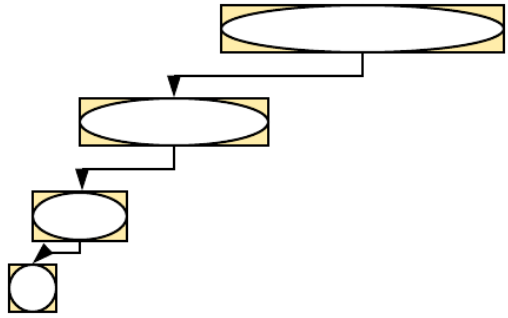
Drvo poziva u slučaju $T(n) = 2T(n/2) + O(n), T(1) = O(1)$ za $n = 8$

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

Ponovo, ako je $n = 2^k$, tada je prvi član jednak $O(1)$ i pošto je na osnovu formule za zbir geometrijskog niza $1/2^{k-1} + ... + 1/2 + 1 = (1-(1/2)^k)/(1-(1/2)) = 2 - 2/n$ 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: $T(n) = aT(n/b) + f(n), T(1) = d$:



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

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. 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 $a = 145$ i $b = 12$ dobija se sledeći niz vrednosti:

a	145	123	111	99	87	75	63	51	39	27	15	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 145 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	145	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 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: 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: 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 maksimalnaPovrsina(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 maksimalnaPovrsina(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: 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 prvi broj nije veći od drugog i sva tri broja su manja ili jednaka 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 $a^2 + b^2 = c^2$.

Granice za a, b, c možemo da odredimo i nešto preciznije. Pošto tražimo takve a, b, c da $a \leq b, a^2+b^2 = c^2 \leq n^2$, sledi da traženi brojevi moraju da ispunjavaju i $a^2 + a^2 \leq n^2$, odakle dalje sledi da je gornja granica za a jednaka $\sqrt{\frac{n^2}{2}}$.

Na sličan način dolazimo i do granica za b: $a \leq b \leq \sqrt{n^2 - a^2}$, dok je granice za c nešto jednostavnije odrediti: $b < c \leq n$.

```
for (int a = 1; 2*a*a <= n*n; a++)
  for (int b = a; a*a + b*b <= n*n; b++)
    for (int c = b+1; 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 $O(n^3)$. Pažljivo određivanje granica petlji utiče samo na smanjivanje konstante uz 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; 2*a*a <= n*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.

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 $a_0 + (a_0 + 1) + ... + (a_0 + (m-1)) \geq n$. 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 $a_0 + (a_0 + 1) + ... + (a_0 + (m-1)) = m \cdot a_0 + \frac{m(m-1)}{2}$, 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)

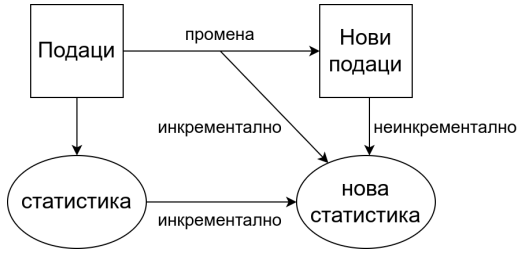
$$m \cdot a_0 + \frac{m(m-1)}{2} = n.$$

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.

3.2 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 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 **??**). 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 $\sum_{k=0}^{n} \frac{(-1)^k}{k!} = 1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} ... + \frac{(-1)^n}{n!}$. 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 $e^{-1} = 1/e \approx 36{,}79\% ....$

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		**Primer 2**	
Ulaz	*Izlaz*	*Ulaz*	*Izlaz*
`2`	`0.500000000000000`	`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;
}

// verovatnoca da nijedna devojcica nije uzela iste klizaljke:
// 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*

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. Naime, vrednosti u tom nizu će opadati do tražene minimalne vrednosti, nakon čega će krenuti da rastu. U prethodnom primeru, mogli smo zaključiti da je minimalna dužina kablova 30, čim se u narednom koraku ta vrednost povećala na 39.

```
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;
    if (duzina_kablova < min_duzina_kablova)
      min_duzina_kablova = duzina_kablova;
  }

  return min_duzina_kablova;
}
```

Pošto je i za jednu i za drugu fazu potrebno vreme $O(n)$, to je ujedno složenost ovog algoritma.

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;
}
```

3.3 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 (ako je n jednako 1, vraćamo vrednost `false`). 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 brojevi 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.

```
// funkcija koja proverava da li je dati broj prost
bool prost(int n) {
  if (n == 1 ||
      (n % 2 == 0 && n != 2) ||
      (n % 3 == 0 && n != 3))
    return false;
  for (int k = 1; (6*k - 1) * (6*k - 1) <= n; k++)
    if (n % (6 * k + 1) == 0 || n % (6 * k - 1) == 0)
      return false;
  return true;
}
```

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

Napiši 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` `1000`	`168 76127`

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 po definiciji 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 (doduše veoma elementarno) 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.

Ojler je dokazao da se zbir $H(m)=1+1/2+1/3+\ldots+1/m=\sum_{d\leq m}\frac{1}{d}$ (takozvani harmonijski zbir) 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. Takođe, otkrio je da 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: 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 $O(1)$.

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 3.3.2. *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 $a_p, a_{p+1}, \dots, a_q$ 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: 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

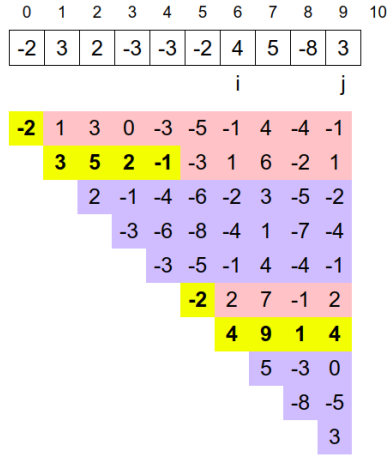
Algoritam zasnovan na proveri svih segmenata se slobodno može nazvati trivijalnim, jer se do njega dolazi prilično direktno i veoma jednostavno mu se i dokazuje korektnost i analizira složenost. Međutim, on je prilično neefikasan za rešavanje ovog problema, čak i kada se zbirovi računaju inkrementalno. 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).

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.).



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.

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 3.3.3. *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* $j + 1 = 6$*.*

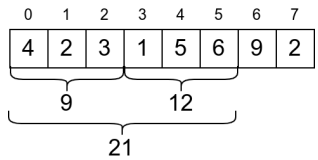
- *Na kraju se obrađuje sedma vrsta ($i = 6$). Promenljiva z se postavlja na nulu, a j na $i = 6$. Zbir z se zatim uvećava za jedan po jedan element i pošto ni u jednom trenutku ne postaje negativan stiže se do kraja niza ($j = 10$). Tokom unutrašnje petlje ažurira se maksimum i dostiže vrednost 9. Po 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: 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: Maksimalni zbir segmenta

Ovaj zadatak je ponovljen u cilju uvežbavanja različitih tehnika rešavanja. Vidi tekst zadatka.

Pokušaj da zadatak uradiš korišćenjem tehnika koje se izlažu u ovom poglavlju.

Rešenje

Zbirovi prefiksa

Jedan algoritam kojim možemo efikasno rešiti ovaj zadatak se zasniva na korišćenju zbirova prefiksa. Ako znamo zbir svakog prefiksa niza, onda zbir svakog segmenta možemo dobiti kao razliku zbirova dva prefiksa. Zbir elemenata segmenta $[l, d]$ jednak je razlici zbira elemenata segmenta $[0, d + 1)$ i zbira elemenata segmenta $[0, l)$, tj. važi da je

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

Računamo da je zbir praznog segmenta $[0, 0)$ po definiciji jednak nuli.

Za svaku poziciju u nizu određujemo maksimalni zbir sufiksa koji se na toj poziciji završava. Najveći od svih maksimalnih zbirova sufiksa je najveći zbir nekog segmenta u nizu (jer je svaki segment zapravo sufiks dela niza do one pozicije na kojoj se taj segment završava).

Koristimo induktivnou konstrukciju i niz proširujemo jednim po jednim elementom. Krećemo od praznog niza čiji je maksimalni zbir segmenta jednak nuli. Prilikom svakog proširivanja niza novim elementom, pretpostavljamo da znamo maksimum zbirova segmenta u neproširenom delu niza i da izračunamo maksimalni zbir sufiksa proširenog niza. Maksimum zbirova segmenta u proširenom nizu je veći od ta dva broja.

Maksimalni zbir sufiksa proširenog niza, na osnovu razlaganja na zbirove prefiksa, dobija se kao razlika zbira celog proširenog niza (tj. zbira prefiksa do tekuće pozicije) i zbira nekog prefiksa neproširenog niza (prazan sufiks ne moramo analizirati, jer je prazan segment već obrađen u sklopu inicijalizacije). Pošto je umanjenik konstantan, da bismo maksimizovali razliku potrebno da znamo najmanji mogući umanjilac, tj. da znamo najmanji zbir prefiksa koji se završava na nekoj poziciji ispred tekuće. I tekući i minimalni zbir prefiksa možemo održavati inkrementalno.

Kada god niz proširimo nekim elementom, zbir prefiksa uvećavamo za taj element, poredimo ga sa dotadašnjim minimalnim zbirom prefiksa i ako je manji, ažuriramo minimalni zbir prefiksa. Naravno, održavamo i globalni maksimalni zbir segmenta koji ažuriramo svaki put kada naiđemo na sufiks čiji je zbir veći od dotadašnjeg maksimuma.

Primetimo da je u ovom rešenju je induktivna hipoteza pojačana i pretpostavljamo da pored segmenta najvećeg zbira u obrađenom delu niza umemo da odredimo i minimalni zbir prefiksa obrađenog dela niza.

Složenost ovog rešenja je $O(n)$, pa je ovo rešenje optimalne složenosti.

```
int zbir_prefiksa = 0;
int min_zbir_prefiksa = zbir_prefiksa;
int max = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  zbir_prefiksa += x;
  int zbir_segmenta = zbir_prefiksa - min_zbir_prefiksa;
  if (zbir_segmenta > max)
    max = zbir_segmenta;
  if (zbir_prefiksa < min_zbir_prefiksa)
    min_zbir_prefiksa = zbir_prefiksa;
```

```
}
cout << max << endl;
```

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 3.4.4. *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 na-

rednom 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.

3.5 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 prebrojavanje različitih elemenata se 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 niza, 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 $O(n^2)$.

```
void najblizeSobe(const vector<int>& a, int& soba1, int& soba2) {
  // broj soba
  int n = a.size();
  // dve sobe sa najmanjim rastojanjem
  int min_i = a[0], min_j = a[1];
  // rastojanje izmedju njih
  int d_min = abs(min_i - min_j);
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++) {
      // rastojanje izmedju soba i i j
```

```
      int d_ij = abs(a[i] - a[j]);
      // ako je rastojanje manje od do sada najmanjeg ili je jednako,
      // ali su sobe dalje od recepcije
      if (d_ij < d_min ||
          (d_ij == d_min && min(min_i, min_j) < min(a[i], a[j]))) {
        // azuriramo najblize sobe i rastojanje izmedju njih
        min_i = a[i];
        min_j = a[j];
        d_min = d_ij;
      }
    }
  // vracamo sobe u uredjenom redosledu
  soba1 = min(min_i, min_j); soba2 = max(min_i, min_j);
}
```

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).

Zaista, u neopadajuće sortiranom nizu važi da iz $j' < j < i$ sledi $a_i - a_j \leq a_i - a_{j'}$, jer iz sortiranosti i $j' < j$ sledi da je $a'_j \leq a_j$. Zato za svako $j' < i-1$ važi da je $a_i - a_{i-1} \leq a_i - a_{j'}$, pa element na poziciji $j' < i-1$ nije bliži elementu a_i nego element a_{i-1}. Važi i da iz $i < j < j'$ sledi da je $a_j - a_i \leq a_{j'} - a_i$, pa za svako $j' > i+1$ važi da je $a_{i+1} - a_i \leq a_{j'} - a_i$, pa element na poziciji $j' > i+1$ nije bliži elementu a_i nego element a_{i+1}.

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 as = a;
  sort(begin(as), end(as));
  int min = 1;
  for (int i = 2; i < a.size(); i++)
    if (as[i] - as[i-1] <= as[min] - as[min-1])
      min = i;
  soba1 = as[min-1]; soba2 = as[min];
}
```

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 prirodnih 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*	*Objašnjenje*
8 3 8 1 17 4 7 12 9 4	5	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;
}
```

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;
}
```

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;
```

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: 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

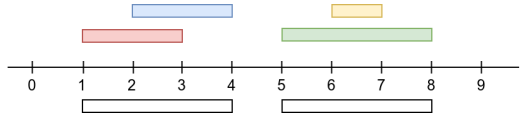
U prvoj liniji standardnog ulaza učitava se broj intervala n ($1 \leq n \leq 50000$), a u narednih n linija parovi celih brojeva za koje važi $-10^6 \leq L_i < R_i \leq 10^6$ koji predstavljaju levi i desni kraj intervala.

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 (ako se seku, to je njihova unija)
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 ukrupnjenih intervala
    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 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 $[L_i, D_i]$. 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 pocetka (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;
  }
```

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?

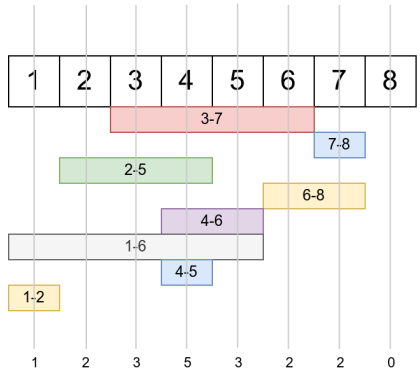
Opis ulaza

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

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 ćemo upoređivati sa maksimumom tek kada obradimo sve časove koji su počeli ili su se završili u tom trenutku (što rešavamo tako što u unutrašnjoj petlji obrađujemo sve časove koji su počeli ili su se završili u tekućem trenutku).

Ako postoji n ljudi koji su bili na bazenu, 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 (obratite pažnju na to da iako u implementaciji postoje dve ugnežđene petlje, promenljiva `i` samo uvećava svoju vrednost i ukupno se kroz obe petlje izvrši najviše $2n$ koraka). 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;
int i = 0;
while (i < 2*n) {
  int trenutak = promene[i].first;
  while (i < 2*n && promene[i].first == trenutak)
```

```
    trenutnoUcionica += promene[i++].second;
  if (trenutnoUcionica > maksUcionica)
    maksUcionica = trenutnoUcionica;
}

cout << maksUcionica << endl;
```

Eliminisanje unutrašnje petlje

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.

Složenost rešenja ostaje $O(n \log n)$, dok se u ovoj implementaciji još jasnije vidi da je faza nakon sortiranja niza linearne složenosti tj. $O(n)$.

```
int trenutnoUcionica = 0;
int maksUcionica = 0;
for (int i = 0; i < 2*n; i++) {
  trenutnoUcionica += promene[i].second;
  if (trenutnoUcionica > maksUcionica)
    maksUcionica = trenutnoUcionica;
}
```

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 datog. 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).

- 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

[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.

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 5. 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);
  else if (x > a[s])
    return binarna_pretraga(a, s+1, d, x);
  else /* if (x == a[s]) */
    return s;
}
```

[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).

```
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.

Oba rekurzivna poziva u navedenoj implementaciji su repno-rekurzivna, tako da se mogu jednostavno eliminisati (postpuci eliminisanja rekurzije su sistematično opisani u poglavlju 4.9). Time se dobija iterativna funkcija koja vraća indeks elementa niza `x` koji je jednak zadatoj vrednosti `x` ako takva postoji i `-1` inače.

```
int binarna_pretraga(const vector<int>& a, int n, int x)
{
  int l, d, s;
  l = 0; d = n-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;
}
```

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 datom elementu 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 i nalazi se na poziciji 6
```

```
auto it = lower_bound(begin(a), end(a), x);
cout << distance(begin(a), it) << endl;
```

Varijanta binarne pretrage u kojoj se pronalazi prelomna tačka 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 traženom, 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 << "Sadrzi 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. Da bismo ručno implementirali funkcionalnost funkcije `lower_bound`, 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).

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;
}
```

Napomenimo da smo funkciju 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 $a_s < x$ jeste deljiv sa k, 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: 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: 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.

```
int minRotiranogSortiranog(const vector<int>& a) {
  int l = 0, d = a.size()-1;
  while (l <= d) {
    int s = l + (d-l)/2;
    if (a[s] < a[0])
```

```
      d = s-1;
    else
      l = s+1;
  }

  int min = l < a.size() ? a[l] : a[0];
  return min;
}
```

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];
}
```

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	5
210 2310 390 30 510 66 6 138 46 106 59 17 23	10
10	8
2	10
6	0
2	5
4	
15	

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 početnih mejl adresa koje treba dopuniti m ($1 \leq m \leq 10^5$), a nakon toga i tih m početih 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);
}
```

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 vred-

nost 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 nekoliko problema.

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;
}
```

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;
}
```

3.6.4 *Određivanje minumuma ili maksimuma realne funkcije*

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 od date vrednosti x (u proizvoljnom redosledu), a zatim nalaze elementi niza koji su veći od date vrednosti x (u proizvoljnom redosledu). Videćemo u poglavlju 5.2 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 $[j, n)$ 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 ungežđene 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 optimalno 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 5.1, 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.

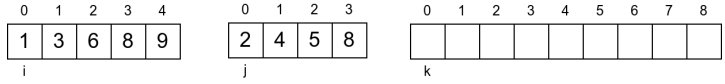
Tokom iterativne implementacije održavaju se dva pokazivača: promenljiva i koja ukazuje na poziciju tekućeg elementa prvog i j koja ukazuje na tekući element drugog niza. Dok su obe ove promenljive manje od dužine niza po kojem se kreću, poredimo elemente na tim pozicijama. Ako je element na poziciji 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).

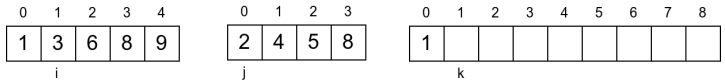
```
// objedinjava sortirani niz a od n elemenata i sortirani niz b od m
// elemenata smestajuci rezultat u sortirani niz c
vector<int> objedini(const vector<int>& a, const vector<int>& b) {
  int m = a.size(), n = b.size();
  vector<int> c(m + n);
  int i = 0, j = 0, k = 0;
  while (i < n && j < m)
    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.*

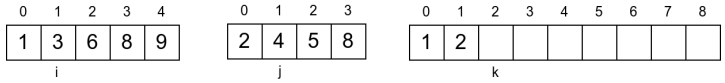
- *Pretpostavimo da je potrebno objediniti naredna dva niza.*



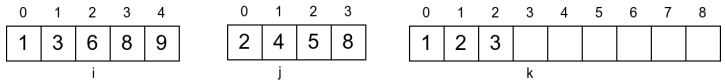
- *U prvom koraku je $i = 0$ i $j = 0$, pa se porede elementi na pozicijama 0, tj. elementi 1 i 2. Pošto je 1 manji, on se prepisuje u rezultujući niz i uvećava se levi pokazivač.*



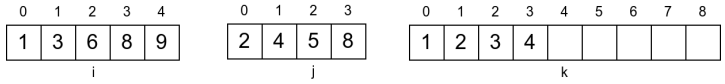
- *Sada je* $i = 1$ *i* $j = 0$*, pa se porede elementi na pozicijama* 1 *i* 0*, tj.* 3 *i* 2*. Pošto je* 2 *manji, on se prepisuje u rezultujući niz i uvećava se desni pokazivač.*



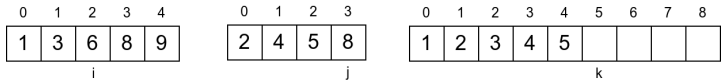
- *Sada je* $i = 1$ *i* $j = 1$*, pa se porede elementi na pozicijama* 1 *i* 1*, tj.* 3 *i* 4*. Pošto je* 3 *manji, on se prepisuje u rezultujući niz i uvećava se levi pokazivač.*



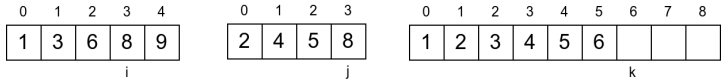
- *Sada je* $i = 2$ *i* $j = 1$*, pa se porede elementi na pozicijama* 2 *i* 1*, tj.* 6 *i* 4*. Pošto je* 4 *manji, on se prepisuje u rezultujući niz i uvećava se desni pokazivač.*



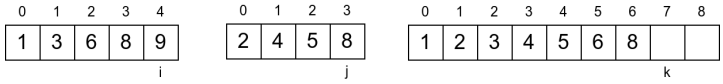
- *Sada je* $i = 2$ *i* $j = 2$*, pa se porede elementi na pozicijama* 2 *i* 2*, tj.* 6 *i* 5*. Pošto je* 5 *manji, on se prepisuje u rezultujući niz i uvećava se ponovo desni pokazivač.*



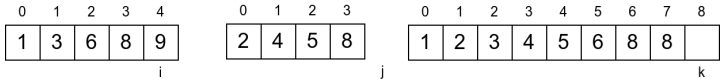
- *Sada je* $i = 2$ *i* $j = 3$*, pa se porede elementi na pozicijama* 2 *i* 3*, tj.* 6 *i* 8*. Pošto je* 6 *manji, on se prepisuje u rezultujući niz i uvećava se levi pokazivač.*



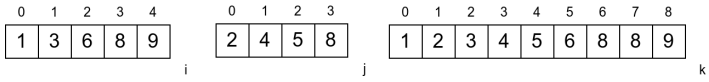
- *Sada je* $i = 3$ *i* $j = 3$*, pa se porede elementi na pozicijama* 3 *i* 3*, tj.* 8 *i* 8*. Pošto su jednaki, bilo koji od njih (na primer levi) može biti prepisan u rezultujući niz i odgovarajući pokazivač se uvećava.*



- *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.

Dokažimo i korektnost iterativne varijante. Invarijanta petlje je da se u nizu c na pozicijama $[0, k)$ nalaze objedinjeni delovi niza a sa pozicija $[0, i)$ i niza b sa

pozicija $[0, j)$ — multiskup elemenata tih podnizova se poklapa sa multiskupom elemenata na početku niza c (na pozicijama $[0, k)$), taj deo niza c je sortiran i svi elementi u tom delu niza c su manji ili jednaki od svih preostalih elemenata sortiranih nizova a i b.

- Invarijanta važi na početku jer su sva tri intervala prazna (pošto je $i = j = k = 0$).
- Kada su oba niza neprazna, telo prve petlje proširuje invarijantu. Na primer, kada se u niz c prenese element a_i niza a, tada se za jedan poveća vrednost i i vrednost k. Multiskup elemenata obrađenih delova nizova a i b se proširuje elementom a_i, isto kao i multiskup popunjenog dela niza c, pa pošto su ti multiskupovi bili jednaki pre proširenja, oni ostaju jednaki i nakon proširenja. Pošto su elementi u nizu c bili manji ili jednaki od svih preostalih elemenata u nizovima a i b, nakon prenošenja elementa a_i, niz c ostaje sortiran. Preostali elementi u nizovima a i b su veći ili jednaki od svih elemenata niza c (već su bili veći od svih osim ovog koji je trenutno dodat, a veći su i od njega, jer je on izabran kao minimalni element iz preostalih delova nizova a i b). Isto važi i kada se prebaci element niza b u niz c.
- Kada je preostali deo nekog od nizova a ili b prazan, rezultat se dobija prebacivanjem preostalih elemenata drugog niza u niz c. Na osnovu invarijante, niz c već sadrži sve elemente ispražnjenog niza i sve elemente drugog niza (osim tih koji su preostali), sortiran je i svi elementi u njemu su manji od preostalih elemenata drugog niza (koji su sortirani). Stoga se nakon prebacivanja tih elemenata u niz c dobija niz koji je sortiran i čiji je multiskup elemenata jednak uniji multiskupova elemenata nizova a i b.

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: 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 3.7.2. *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 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 3.7.3. *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;
}
```

Dokažimo korektnost prethodnog postupka. Posmatrajmo skup $S_{l,d}$ koji sadrži sve parove (a_i, a_j) takve da je $0 \leq i < l$ i da je $d < j < n$. Invarijanta prethodne petlje biće to da:

- promenljiva koja čuva tekući broj parova (označimo je sa b) čuva broj parova skupa $S_{l,d}$ tako da je $a_i + a_j = s$,
- za svako $0 \leq i < l$ važi da je $a_i + a_d < s$ i za svako $d < j < n$ važi da je $a_l + a_j > s$.

Pre ulaska u petlju važi da je $l = 0$ i $d = n - 1$. Tada nema indeksa i takvog da važi $0 \leq i < l$ niti indeksa j takvog da važi $d < j < n$, pa je $S_{l,d}$ prazan. Pošto je $b = 0$, oba dela invarijante važe.

Pretpostavimo da invarijanta važi pri ulasku u telo petlje i dokažimo da je izvršavanje tela petlje održava.

Pretpostavimo prvo da je $a_l + a_d > s$. Tada d umanjujemo za 1 ne menjajući pri tom broj parova b, tj. nakon izvršavanja tela petlje važi da je $l' = l$, $d' = d - 1$ i $b' = b$. Skup $S_{l',d'} = S_{l,d-1}$ se može razložiti na:

1. skup $S_{l,d}$ i
2. skup skup svih parova (a_i, a_j) takvih da je $0 \leq i < l$ i $j = d$.

Broj parova traženog zbira u skupu $S_{l,d}$ jednak je b (na osnovu prvog dela invarijante), dok se u drugom skupu ne nalazi nijedan takav par. Zaista, na osnovu drugog dela invarijante znamo da svako $0 \leq i < l$ važi da je $a_i + a_d < s$, pa među parovima drugog skupa ne može biti nijedan koji ima zbir jednak s. Pošto je $b' = b$, prvi deo invarijante ostaje očuvan. Potrebno je da pokažemo i da drugi deo invarijante ostaje očuvan. Prvo, treba da dokažemo da za svako $0 \leq i < l'$ važi da je $a_i + a_{d'} < s$. Pošto je $l' = l$, na osnovu drugog dela invarijante znamo da za sve takve indekse i važi da je $a_i + a_d < s$, a pošto je niz sortiran i pošto su mu elementi

različiti, važi da je $a_{d'} = a_{d-1} < a_d$, pa je $a_i + a_{d'} = a_i + a_{d-1} < a_i + a_d < s$. Treba da dokažemo i da za svako $d' < j < n$, važi da je $a_{l'} + a_j > s$. Na osnovu drugog dela invarijante to važi za sve $d < j < n$. Pošto je $l' = l$ i $d' = d - 1$, ostaje samo još da se dokaže da taj uslov važi za d, tj. samo da se dokaže da važi da je $a_l + a_d > s$, no to važi na osnovu pretpostavke (tj. grane koju trenutno analiziramo).

Veoma slično se pokazuje da se u slučaju $a_l + a_d < s$ povećanjem broja l i ne menjanjem broja d invarijanta održava.

Na kraju, ostaje slučaj kada je $a_l + a_d = s$. U tom slučaju se vrši uvećanje broja l, umanjenje broja d i uvećanje broja b, tj. važi da je $l' = l + 1$, $d' = d - 1$ i $b' = b + 1$. Skup $S_{l',d'} = S_{l+1,d-1}$ se može razložiti na:

1. skup $S_{l,d}$,
2. skup svih parova (a_i, a_j) takvih da je $0 \leq i < l$, $j = d$,
3. skup svih parova takvih da je $i = l$, $d < j < n$ i
4. skup koji sadrži samo par (a_l, a_d).

Na osnovu prvog dela invarijante važi da u skupu $S_{l,d}$ ima b parova čiji je zbir s. Pošto je $a_l + a_d = s$, (a_l, a_d) je još jedan traženi par. Ostaje još da pokažemo da u preostala dva skupa ne postoji nijedan par čiji je zbir s. Zaista, skup svih parova takvih da je $0 \leq i < l$, $j = d$ i $a_i + a_j = s$, je prazan, jer na osnovu drugog dela invarijanate znamo da je u svim tim parovima $a_i + a_j < s$. Analogno, na osnovu drugog dela invarijante dokazujemo da je prazan i skup svih parova takvih da je $i = l$, da je $d < j < n$ i $a_i + a_j = s$, jer za sve te parove važi da je $a_i + a_j > s$. Dakle, prvi deo invarijante ostaje očuvan. Potrebno je još dokazati da je očuvan i drugi deo invarijante. Potrebno je dokazati da je za za svako $0 \leq i < l'$ važi da je $a_i + a_{d'} < s$. Na osnovu drugog dela invarijante znamo da za svako $0 \leq i < l$ važi da je $a_i + a_d < s$. Pošto je niz sortiran i svi su mu elementi različiti i pošto je $d' = d - 1$, važi da je $a_{d'} < a_d$. Zato je $a_i + a_{d'} < a_i + a_d < s$. Pošto je $l' = l + 1$, ostaje da se to dokaže još da je $a_l + a_{d'} < s$. No znamo da je $a_l + a_{d'} < a_l + a_d = s$. Analogno se dokazuje još da za svako $d' < j < n$ važi $a_{l'} + a_j > s$.

Kraj petlje nastupa kada je $l \geq d$. Skup svih parova (a_i, a_j) takvih da je $0 \leq i < j < n$ se može razložiti na:

1. skup svih parova (a_i, a_j) takvih da je $j \leq d$,

2. skup $S_{l,d}$ tj. skup svih parova (a_i, a_j) takvih da je $0 \leq i < l$ i $d < j < n$ i
3. skup svih parova (a_i, a_j) takvih da je $l \leq i$.

U prvom i trećem skupu nema nijedan par čiji je zbir jednak s. Zaista, ako je $j \leq d$, tada važi da je $0 \leq i < j \leq d \leq l$. Na osnovu drugog dela invarijante znamo da tada važi da je $a_i + a_d < s$, a pošto je niz sortiran važi i da je $a_j \leq a_d$, pa je $a_i + a_j \leq a_i + a_d < s$. Ako važi da je $l \leq i$, tada važi i da je $d \leq l \leq i < j < n$. Na osnovu drugog dela invarijante znamo da je $a_l + a_j > s$, pa pošto je niz sortiran važi da je $a_l \leq a_i$ i zato je $a_i + a_j \geq a_l + a_j > s$. Na osnovu prvog dela invarijante znamo da je broj b jednak broju parova iz drugog skupa, što je na osnovu prethodnog ukupan broj traženih parova.

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 $O(n \log n)$.

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 3.7.4. *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 3.7.5. *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 izračuna-

vati 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 $2n$.

3.8 Primena efikasnih struktura podataka

Izbor strukture podataka koja će se koristiti u implementaciji algoritma 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. Podsetimo se, 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 od date 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 $O(\log n)$.

Nekada želimo da dopustimo da skup sadrži duplikate i tada možemo koristiti `multiset` i `unordered_multiset`, koji su iste efikasnosti kao `set` i `unordered_set`.

Ilustrujmo na primeru sortiranja niza brojeva kako izbor pogodne strukture podataka može da učini algoritam efikasnijim. Algoritam sortiranja umetanjem (engl. insertion sort) radi tako što se jedan po jedan element ubacuju na svoje mesto u sortiranom nizu brojeva (to je obično sortiran prefiks niza koji se sortira). 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 može izvršiti 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.

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`	`1`
`marko 2300`	`2`
`milan 5200`	
`dragana 4000`	
`provera 0`	
`milan -1200`	
`provera 4000`	

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]]++;
    }
  }

  return 0;
}
```

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: 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).

```
vector<int> cene;
char c;
while (cin >> c) {
  if (c == 'i') {
    int cena;
    cin >> cena >> ws;
    cene.push_back(cena);
  } else if (c == 'e') {
    int cena;
    cin >> cena >> ws;
    auto it = find(begin(cene), end(cene), cena);
    if (it != a.end())
      cene.erase(it);
  } else if (c == 'f') {
    int budzet;
    cin >> budzet >> ws;
    int maks_cena = -1;
    for (int cena : cene)
      if (cena <= budzet)
        if (maks_cena == -1 || cena > maks_cena)
          maks_cena = cena;

    if (maks_cena != -1)
      cout << maks_cena << '\n';
    else
```

```
      cout << "-" << '\n';
  }
}
```

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, dok 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" (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. Funkija koja je istovremeno "na" i "1-1" zove se bijekcija.

Provera svih podskupova

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. Generisanje svih podskupova možemo uraditi na način koji je opisan u zadatku *Svi podskupovi*, a proveru da li je f bijekcija možemo uraditi prebrojavanjem originala koji se slikaju u svaku od slika - ako je funkcija bijekcija tada se u svaku sliku slika tačno jedan original. Prebrojavanje možemo uraditi korišćenjem

asocijativnog niza i to ili niza brojača ili, još bolje, niza indikatora (istinitosnih vrednosti) koje ukazuju na to da li je za datu sliku već pronađen original.

Još bolje rešenje je da generišemo samo one podskupove za koje unapred znamo da je funkcija na njima bijekcija. Rekurzivnoj funkciji za generisanje podksupova prosleđujemo tekući element i, skup originala manjih od i koji su uključeni u podskup koji se gradi i skup njihovih slika (oba skupa možemo predstaviti nizom bitova). Elementi manji od i koji nisu u skupu originala izostavljeni su iz podskupa koji se gradi, dok je status elemenata većih ili jednakih od i nepoznat (oni potencijalno mogu biti i uključeni i isključeni iz skupa originala). Invarijanta koja se održava je da se nikoja dva originala ne slikaju ni u jednu sliku, da skup slika tačno predstavlja slike svih originala, kao i da su sve slike dosadašnjih originala koje su manje od i uključene u skup originala (one slike koje su veće ili jednake i biće uključene naknadno, ako to ne narušava injektivnost).

Izlaz iz rekurzije je slučaj $i = n$. Tada znamo da su sve slike skupa originala uključene u skup originala, pa se ta dva skupa poklapaju, i funkcija je bijekcija na tom skupu (jer znamo da je injektivna). Funkcija tada može da vrati kardinalnost tekućeg skupa originala.

U svakom pozivu rekurzivne funkcije proveravamo dve mogućnosti:

- Trenutni element i može da se uključi u skup originala. Ovo je moguće samo ako je slika elementa i nije izbačena iz skupa (što se dešava ako je $f(i) < i$, a $f(i)$ se ne nalazi u nizu originala), jer bi se u suprotnom narušio uslov da je svaka slika koja je manja od i uključena u niz originala. Takođe, slika ne sme biti već uključena u niz slika (tj. u nizu originala ne sme da se nalazi već neki drugi element koji ima istu sliku kao i), jer bi se tada dodavanjem elementa i narušila injektivnost.

- Druga mogućnost je da se i izostavi iz skupa originala. To je moguće samo ako i nije slika nekog manjeg elementa (ako jeste, onda i mora biti uključen u skup originala).

Pošto podskupova ima 2^n, ovaj algoritam je nedopustivo neefiksan (čak i u varijanti u kojoj se neki skupovi preskaču tokom generisanja).

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

using namespace std;

// do sada popunjeni skup originala (strogo manjih od i) i skup
// njihovih slika prosirujemo elementima vecim ili jednakim od i
// invarijanta je da je funkcija injektivna na skupu originala i
// da je slika svakog originala ukljucena u skup slika i da su slike
// svih elemenata manjih od i ukljucene u skup originala
int bijekcija(const vector<size_t>& f,
              vector<bool>& originali, vector<bool>& slike, size_t i) {
  // skup originala se ne moze vise prosiriti
  // na osnovu invarijante funkcija je bijekcija na skupu originala
  if (i == f.size())
    return count(begin(originali), end(originali), true);

  int rez = 0;
  // ako f[i] nije ranije oznacena kao izbacena iz skupa originala
  // i ako f[i] nije vec slika nekog elementa iz niza originala
  if ((f[i] >= i || originali[f[i]]) && !slike[f[i]]) {
    // ubacujemo i u skup originala i f[i] u skup slika
    originali[i] = true;
    slike[f[i]] = true;
    // prosirujemo vecim elementima i azuriramo rezultat
    int rez_sa_i = bijekcija(f, originali, slike, i+1);
    rez = rez_sa_i;
    // izbacujemo i iz skupa originala i f[i] iz skupa slika
    // da bismo se vratili u pocetnu poziciju
    originali[i] = false;
    slike[f[i]] = false;
  }
  // ako i nije slika nekog elementa, onda on moze biti preskocen
  if (!slike[i]) {
    // prosirujemo elementima vecim od i (i je preskocen)
```

```
    int rez_bez_i = bijekcija(f, originali, slike, i+1);
    // azuriramo rezultat ako je to potrebno
    if (rez_bez_i > rez)
      rez = rez_bez_i;
  }
  return rez;
}

int bijekcija(const vector<size_t>& f) {
  int n = f.size();
  vector<bool> originali(n, false), slike(n, false);
  return bijekcija(f, originali, slike, 0);
}

int main() {
  int n;
  cin >> n;
  vector<size_t> f(n);
  for (int i = 0; i < n; i++)
    cin >> f[i];

  cout << bijekcija(f) << endl;
  return 0;
}
```

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.

```
vector<int> ulazniStepen(n, 0);
for (int i = 0; i < n; i++)
  ulazniStepen[f[i]]++;
queue<int> q;
for (int i = 0; i < n; i++)
  if (ulazniStepen[i] == 0)
    q.push(i);
int broj_elemenata = n;
while (!q.empty()) {
  int i = q.front(); q.pop();
  broj_elemenata--;
  if (--ulazniStepen[f[i]] == 0)
    q.push(f[i]);
}
cout << broj_elemenata << endl;
```

Interesantno, u ovom zadatku se može primetiti da se nakon izbacivanja elementa iz reda u red uvek ubacuje najviše jedan element. To ukazuje na mogućnost rešenja u kome red nije ni potreban. Analiziramo redom jedan po jedan element koji ima ulazni stepen nula (u koji se ne slika ni jedan element). Svaki takav element izbacujemo i ako nakon njegovog izbacivanja stepen njegove slike postane nula isto radimo i sa njegovom slikom, sve dok se ne dogodi da nakon izbacivanja elementa stepen njegove slike ostane pozitivan.

```
vector<int> ulazniStepen(n, 0);
for (int i = 0; i < n; i++)
  ulazniStepen[f[i]]++;

int broj_elemenata = n;
for (int i = 0; i < n; i++) {
  if (ulazniStepen[i] == 0) {
    int m = i;
    while (ulazniStepen[m] == 0){
      broj_elemenata--;
      m = f[m];
      ulazniStepen[m]--;
      if (m > i)
        break;
    }
  }
}
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;
```

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:

$$\begin{gathered} a_0 + b_0, a_0 + b_1, \ldots, a_0 + b_{n-1}, \\ a_1 + b_0, a_1 + b_1, \ldots, a_1 + b_{n-1}, \\ \ldots \\ a_{m-1} + b_0, a_{m-1} + b_1, \ldots, a_{m-1} + b_{n-1}. \end{gathered}$$

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 oči-

tati u konstantnom vremenu, ukloniti u logaritamskom, isto koliko je potrebno i da se umetne novi element.

Primer 3.8.1. *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();
    }
  }
}
```

4. Induktivna i rekurzivna konstrukcija algoritama

U osnovni većine računarskih algoritama krije se ideja da se problemi rešavaju tako što se prethodno reše potproblemi istog oblika, ali manje dimenzije. Ovo je u veoma tesnoj vezi sa principom matematičke *indukcije* sa jedne i principom *rekurzije* sa druge strane. Videćemo da iako to na prvi pogled nije možda sasvim očigledno, ovaj princip leži u osnovi ne samo rekurzivno, već i iterativno implementiranih algoritama.

4.1 Matematička indukcija

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[1]. 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*. Prilično je jasno zašto je princip matematičke indukcije korektan — 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

[1] U nastavku pod skupom prirodnih brojeva podrazumevamo skup $\mathbb{N}_0$ tj. podrazumevamo da je nula najmanji prirodni broj.

prirodnog broja, koji sigurno mora imati svojstvo P. Baza se može formulisati i za veće vrednosti od nule, ali onda samo možemo da tvrdimo da elementi koji su veći ili jednaki od baze imaju svojstvo P.

Indukcija je tesno povezana sa skupom prirodnih brojeva, jer je sam skup prirodnih brojeva definisan induktivno, kao najmanji skup koji zadovoljava naredna induktivna pravila.

baza: nula je prirodan broj
korak: sledbenik svakog prirodnog broja je prirodan broj

Na sličan način, moguće je induktivno predstaviti i druge tipove (skupove) podataka.

Nisku (karaktera) moguće je definisati na sledeći način: (baza) prazna niska predstavlja nisku i (korak) dodavanjem karaktera na kraj neke niske dobija se niska.

Slično tome, dekadni zapis prirodnog je ili (baza) zapis dekadne cifre ili (korak) zapis prirodnog broja na koji je nadovezan zapis dekadne cifre sa desne strane.

I aritmetički izrazi (termovi) se mogu definisati induktivno.

baza: Bazni slučaj predstavljaju najjednostavniji izrazi: konstante i promenljive.
korak: Induktivni korak predviđa da se složeniji izrazi mogu dobiti od jednostavnijih primenom aritmetičkih operatora (+, *, …) ili zagrada.

I neke relacije mogu biti definisane induktivno. Na primer, relacija *predak* može se definisati na sledeći način:

baza: roditelj osobe je predak te osobe;
korak: roditelj bilo kog pretka neke osobe takođe je predak te osobe.

Relacija predak je onda najmanja relacija (u smislu inkluzije) koja zadovoljava ovako zadata pravila (predak nije nijedna osoba za koju se primenom ovih pravila ne može izvesti da je predak).

Indukcija, dakle, nije samo metod za dokazivanje teorema nad prirodnim brojevima, već opštiji princip koji nam daje mogućnost da od jednostavnijih objekata izgradimo složenije, krenuvši od nekih baznih i primenjujući zadata pravila proširenja.

4.2 Rekurzija

U matematici i računarstvu, *rekurzija* je veoma srodan (ali suprotno usmeren) pristup matematičkoj indukciji, u kojem se neka funkcija definiše, tj. kojim se daje postupak njenog izračunavanja na osnovu jednog ili više baznih slučajeva i na osnovu pravila koja složene slučajeve svode na jednostavnije. Većina savremenih programskih jezika dopušta definisanje rekurzivnih funkcija – funkcija može da pozove samu sebe, pri čemu korisnik treba da bude siguran da će se taj lanac rekurzivnih poziva u nekom trenutku zaustaviti (obično tako što će svaki naredni rekurzivni poziv biti za manju vrednost parametra).

Razmotrimo naredni primer.

Primer 4.2.1. *Vrednost faktorijela* $n!$ *se može opisati sledećom rekurzivnom definicijom:*

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

Vrednost $4!$ *se može izračunati primenom ovih pravila kao* $4! = 4 \cdot 3! = 4 \cdot 3 \cdot 2! = 4 \cdot 3 \cdot 2 \cdot 1! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 0! = 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 = 24$.

Implementacija u jeziku C++ direktno prati definiciju.

```
int faktorijel(int n)
{
  if (n == 0) return 1;
  else return n * faktorijel(n-1);
}
```

Izostanak bilo baznog koraka (koji obezbeđuje “zaustavljanje” primene definicije) bilo rekurzivnog koraka čini definiciju nepotpunom[2].

[2] Često se citira šala kako se u rečniku rekurzija može najilustrativnije objasniti tako što se pod stavkom *rekurzija* napiše: *vidi rekurzija*.

4.3 Odnos indukcije i rekurzije

Indukcija nam daje pravila kako se od jednostavnijih objekata dobijaju složeniji, dok nam rekurzije daje način kako da se izračunavanje nekog složenijeg objekta svede na izračunavanje nekih jednostavnijih. U induktivnoj definiciji tipa prirodnih brojeva koristila se činjenica da je *sledbenik* prirodnog broja prirodan broj, a u rekurzivnoj definiciji smo izračunavanje vrednosti faktorijela sveli na izračunavanje vrednosti faktorijela *prethodnika*.

Zbog izuzetne povezanosti pojma indukcije i rekurzije oni se ponekad koriste sinonimno i nekada se govori o *induktivno/rekurzivnoj* konstrukciji algoritama. Ona u svom osnovnom obliku podrazumeva da se rešenje problema veće dimenzije pronalazi tako što umemo da rešimo problem istog oblika, ali manje dimenzije i da od rešenja tog problema dobijemo rešenje problema veće dimenzije (u naprednijim oblicima je moguće i da se rešava veći broj problema manje dimenzije). Pritom za početne dimenzije problema rešenje moramo da izračunavamo direktno, bez daljeg svođenja na probleme manje dimenzije. Ako se prilikom svođenja dimenzija problema uvek smanjuje, konstruisani algoritmi će se uvek zaustavljati.

- Implementacija algoritma može biti takva da promenljive unutar petlje iterativno ažuriraju svoje vrednosti krenuvši od vrednosti koje predstavljaju rešenja elementarnih problema, 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 sama sebe poziva da bi rešila problem istog oblika, ali manje dimenzije (osim u slučaju elementarnih problema, koji se direktno rešavaju) i tada kažemo da je algoritam definisan *rekurzivno*.

Induktivna konstrukcija leži u osnovni praktično svih iterativnih algoritama koje smo do sada razmatrali. Razmotrimo jedan jednostavni primer.

Primer 4.3.1. *Algoritam izračunavanja zbira serije brojeva (na primer, zbira elemenata nekog vektora) počiva na tome da znamo da izračunamo zbir prazne serije (to je 0) i da ako znamo zbir serije od k elemenata, tada umemo da izračunamo i zbir serije koja se dobija proširivanjem te serije dodatnim $k+1$-vim elementom (to radimo tako što dotadašnji zbir uvećamo za taj novi element).*

```
int zbir = 0;
for (int i = 0; i < a.size(); i++)
    zbir = zbir + a[i];
```

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 zbira). Baza može odgovarati i slučaju jednočlanog (a ne obavezno praznog) niza, ali tada ne možemo da garantujemo da će algoritam raditi ispravno u slučaju praznog niza. To odgovara varijanti algoritma u kojoj zbir inicijalizujemo na prvi element niza, pa ga uvećavamo redom za jedan po jedan element od pozicije 1 nadalje.

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

```
int zbir(int a[], int n) {
    if (n == 0)
       return 0;
    else
       return zbir(a, n-1) + a[n-1];
}
```

Korektnost obe implementacije se dokazuje korišćenjem matematičke indukcije, pri čemu se ona u slučaju iterativne implementacije obično krije unutar koncepta invarijante petlje (videti poglavlje 1.4.4).

4.4 Primitivna i opšta rekurzija

Princip dokazivanja matematičkom indukcijom je u vezi sa induktivnom definicijom skupa prirodnih brojeva — skup prirodnih brojeva je najmanji skup koji sadrži nulu i zatvoren je za operaciju sledbenika. Iz ovoga proističe da je algoritme za rad sa prirodnim brojevima ponekad moguće definisati tako da u slučaju izlaska iz rekurzije razrešavaju slučaj nule, dok u slučaju nekog broja većeg od nule (sledbenika nekog broja) rekurzivno svode problem na slučaj njegovog prethodnika. Ovakav tip rekurzije, koji direktno odgovara induktivnoj definiciji tipa podataka, naziva se *primitivna rekurzija*.

Primer 4.4.1. *Naredna definicija stepenovanja je primitivno rekurzivna.*

$$x^n = \begin{cases} 1, & n = 0 \\ x \cdot x^{n-1}, & n > 0 \end{cases}$$

Na osnovu ove definicije, vrednost 2^4 *se vrši na sledeći način.*

$$2^4 = 2 \cdot 2^3 = 2 \cdot 2 \cdot 2^2 = 2 \cdot 2 \cdot 2 \cdot 2^1 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2^0 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 1 = 16.$$

Implementacija ove funkcije direktno prati definiciju.

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

Kada u obzir uzmemo induktivnu definiciju dekadnog zapisa prirodnog broja (u kojoj bazu čine jednocifreni brojevi, a višecifreni brojevi se dobijaju dopisivanjem cifre zdesna na već postojeći dekadni zapis), lako možemo definisati primitivno rekurzivne funkcije takve da izlaz iz rekurzije predstavljaju jednocifreni brojevi, dok se slučaj višecifrenog broja razrešava svođenjem na broj sa odsečenom poslednjom cifrom. Ako je broj n predstavljen podatkom celobrojnog tipa, poslednju cifru možemo odrediti izrazom `n%10`, a možemo je ukloniti izrazom `n/10`.

Primer 4.4.2. *Zbir cifara broja* n *može se izračunati na sledeći način.*

```
int zbir_cifara(int n)
{
  if (n < 10)
    return n;
  return zbir_cifara(n / 10) + n % 10;
}
```

Ako funkciju skraćeno obeležimo sa z, važi $z(123) = z(12)+3 = z(1)+2+3 = 1+2+3 = 6$.

Dualni pristup – razlaganje broja na prvu cifru i na ostale cifre – u ovom slučaju nije pogodan, zbog toga što ovakvo razlaganje nije moguće sprovesti jednostavnim matematičkim operacijama.

Funkcije koje obrađuju niske se takođe mogu definisati rekurzivno, tako da pri izlasku iz rekurzije obrađuju slučaj prazne niske, dok slučaj neprazne niske dužine n rešavaju tako što rekurzivno razreše njen prefiks dužine $n-1$ i onda rezultat iskombinuju sa poslednjim elementom niske.

Primer 4.4.3. *Naredna rekurzivna funkcija broji razmake (blanko karaktere) u datoj niski.*

```
int broj_razmaka(const string& s, int n)
{
   if (n == 0) return 0;
   if (s[n-1] == ' ')
      return 1 + broj_razmaka(s, n-1);
   else
      return broj_razmaka(s, n-1);
}

int broj_razmaka(const string& s)
{
   return broj_razmaka(s, s.length());
}
```

Primetimo da smo ovde, kao što je to čest slučaj sa rekurzivnim implementacijama, definisali i tzv. omotač funkciju (engl. wrapper function) čiji je jedini cilj da ispravno inicijalizuje argumente početnog rekuzivnog poziva i na taj način olaka krajnjem korisniku upotrebu rekurzivno definisane funkcije (korisnik ne mora ni da zna šta je parametar koji se prosleđuje i menja tokom rekurzivnih poziva).

Ako funkciju skraćeno označimo sa B, izračunavanje za nisku `"ab cd ef"` *suštinski teče na sledeći način:*

```
B("ab cd ef") = B("ab cd e") = B("ab cd ") =
1 + B("ab cd") = 1 + B("ab c") = 1 + B("ab ") =
1 + 1 + B("ab") = 1 + 1 + B("a") = 1 + 1 + B("") = 1 + 1 + 0 = 2
```

Međutim, da se niska ne bi menjala tokom rekurzivnih poziva (što bi bilo veoma neefikasno), u implementaciji se prosleđuje parametar n, koji se tokom rekurzivnih poziva smanjuje.

```
B("ab cd ef") = B("ab cd ef", 8) = B("ab cd ef", 7) =
B("ab cd ef", 6) = 1 + B("ab cd ef", 5) = 1 + B("ab cd ef", 4) =
1 + B("ab cd ef", 3) = 1 + 1 + B("ab cd ef", 2) =
1 + 1 + B("ab cd ef", 1) = 1 + 1 + B("ab cd ef", 0) =
1 + 1 + 0 = 2
```

Dualno razlaganje na prvi element i sufiks niske takođe je moguće, pri čemu je onda prilikom svakog rekurzivnog poziva potrebno, pored dužine niske, prosleđivati i poziciju početka sufiksa.

U nekim slučajevima, potrebno je koristiti i naprednije oblike indukcije, kakva je, na primer, *totalna indukcija*. U tom slučaju, nakon dokazivanja induktivne baze, u okviru induktivnog koraka moguće je pretpostaviti tvrđenje za sve brojeve manje od n i iz te pretpostavke dokazati tvrđenje za broj n. Slično, pored primitivno rekurzivnih funkcija, postoje i funkcije koje su *totalno (generalno) rekurzivne*. Tako je, na primer, prilikom razmatranja nekog broja n, dozvoljeno izvršiti rekurzivni poziv za bilo koji broj manji od njega (pa čak i više rekurzivnih poziva za različite prirodne brojeve manje od njega).

Primer 4.4.4. *Naredna definicija stepenovanja je efikasnija od one primitivno-rekurzivne.*

$$x^n = \begin{cases} 1, & n = 0 \\ (x \cdot x)^{\frac{n}{2}}, & n > 0,\ n \textit{ je paran} \\ x \cdot x^{n-1}, & n > 0,\ n \textit{ je neparan} \end{cases}$$

Na osnovu ove definicije, izračunavanje vrednosti 2^{10} *teče na sledeći način:*

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

I u ovom slučaju implementacija može direktno pratiti datu rekurzivnu definiciju.

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

Za razliku od prethodnih primera koji se trivijalno implementiraju i iterativno, iterativna implementacija brzog stepenovanja nije u potpunosti očigledna (i biće prikazana u poglavlju 4.9}).

U nekim slučajevima upotrebe totalne rekurzije, bazni slučaj mora da pokriva više od jedne vrednosti.

I prilikom implementacije algoritma koji radi sa nekim nizom, dozvoljeno je vršiti rekurzivne pozive za sve podnizove polaznog niza kraće od njega. Kao što će kasnije biti pokazano, obrada niza kao unije dva njegova dvostruko kraća podniza često vodi boljoj efikasnosti nego primitivno-rekurzivna obrada niza kao unije jednog elementa i podniza bez tog elementa.

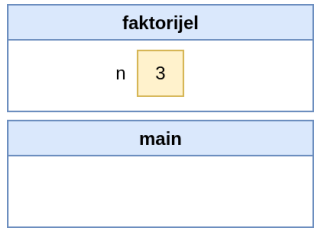
4.5 Realizacija rekurzije

U fazi izvršavanja, rekurzija se realizuje u skladu sa opštim mehanizmom izvršavanja funkcija: za svaki rekurzivni poziv stvara se novi stek okvir i u njemu se čuvaju informacije potrebne za izvršavanje funkcije ali i za nastavak izvršavanja programa. Ono što odlikuje rekurzivne pozive je to što ti stek okviri mogu da odgovaraju istoj funkciji. Dakle, za jednu istu funkciju na programskom steku može postojati više stek okvira, obično za različite vrednosti argumenata funkcije. Međutim, u memoriji računara postoji uvek samo jedna kopija kôda funkcije (koja se nalazi u segmentu koda). Naime, za različite instance iste funkcije (tj. za sve aktivne pozive funkcije) potrebno je pamtiti dokle je stalo njihovo izvšavanje ali nije potrebno imati nezavisne kopije kôda funkcije.

Primer 4.5.1. *Razmotrimo ponovo funkciju koja računa faktorijel broja:*

```
int faktorijel(int n)
{
  if (n == 0)
     return 1;
  else
     return n * faktorijel(n-1);
}
```

Pretpostavimo da je iz funkcije `main` *pozvana funkcija* `faktorijel` *sa argumentom* 3. *Tada se na programskom steku nalaze dva stek okvira, jedan za funkciju* `main`*, a jedan za funkciju* `faktorijel` *sa argumentom* 3.

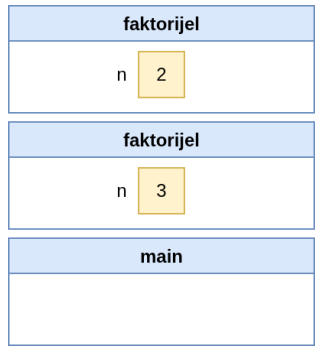


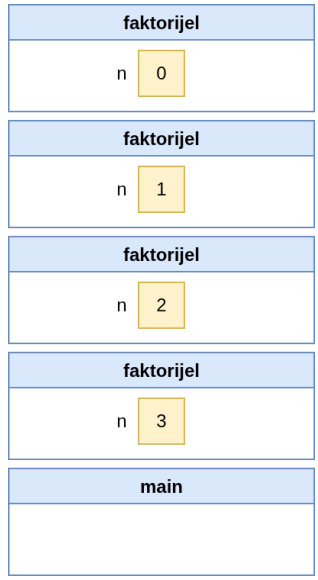
Stek okvir za `faktorijel` *čuva i mesto u kôdu (adresu instrukcije u kodu funkcije* `main`*) na koje se treba vratiti kada se završi izvršavanje funkcije* `faktorijel`*. Tokom izvršavanja funkcije* `faktorijel`*, utvrdi se da* n *nije jednako* 0*, te se prelazi na drugu granu grananja, poziva se funkcija* `faktorijel` *za* $n = 2$ *i stvara stek okvir za tu instancu funkcije* `faktorijel`*, gde će biti zapamćeno, između ostalog, i gde treba nastaviti izvršavanje funkcije za argument* 3.

Analogno se poziva funkcija `faktorijel` *za argument* 1 *i za argument* 0*:*

Kada se izvrši funkcija `faktorijel` *za argument* 0*, njen rezultat (vrednost* 1*) će biti poznat i vraćen pozivaocu (najčešće tako što se upiše u za to namenjen registar), biće obrisan stek okvir za poziv funkcije* `faktorijel` *za vrednost* 0 *i biće nastavljeno izvršavanje* `faktorijel` *za argument* 1 *(od mesta koje je bilo sačuvano u stek okviru za* 0*). Ona će zatim pomnožiti tu povratnu vrednost* 1 *sa vrednošću svoje promenljive* n *(takođe* 1*) i rezultat* 1 *će vratiti svom pozivaocu (što je poziv* `faktorijel` *za* $n = 2$*).*

Analogno će biti završeno i izvršavanje preostale instance funkcije `faktorijel` *i rezultat* 6 *instance za* $n = 3$ *biće vraćen funkciji* `main` *(najčešće preko registra).*


faktorijel
n
2
faktorijel
n
3
main


faktorijel
n
0
faktorijel
n
1
faktorijel
n
2
faktorijel
n
3
main

Pre nego što bude oslobođen stek okvir za `faktorijel` *za* $n = 3$ *iz njega će biti pročitano od koje instrukcije treba da se nastavi izvršavanje funkcije* `main`.

4.6 Primeri rekurzivnih funkcija

Prikažimo u nastavku još nekoliko primera rekurzivnih funkcija.

4.6.1 Fibonačijev niz

Jedan od primera koji se često koristi radi ilustracije dobrih i loših strana rekurzije je Fibonačijev niz (0,1,1,2,3,5,8,13,...) u kojem važi da je svaki naredni član jednak zbiru prethodna dva.[3] On se može definisati u vidu (totalno) rekurzivne funkcije *fib* (primetimo da bazni slučaj pokriva dve vrednosti (0 i 1), a da rekurzivni korak koriste dve prethodne vrednosti niza):

bazni slučaj $fib(0) = 0$ i $fib(1) = 1$ (tj. za $n = 0$ važi $fib(n) = 0$ i za $n = 1$ važi $fib(n) = 1$)
rekurzivni korak za $n > 1$ važi: $fib(n) = fib(n-1) + fib(n-2)$

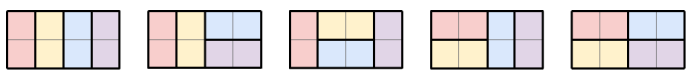
Funkcija za izračunavanje n-tog elementa Fibonačijevog niza može se definisati na sledeći način:

```
int fib(int n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

[3] Fibonačijev niz javlja se u mnogim pojavama u prirodi. Na primer, pčele se rađaju iz oplođenih, a trutovi iz neoplođenih jaja, pa trut ima samo jednu majku, ona ima dva roditelja (oca i majku), oni imaju tri roditelja (dve bake i deku), oni imaju pet roditelja (tri bake i dva deke) i tako dalje. Dakle, broj predaka trutova čini Fibonačijev niz. Fibonači je niz uveo kroz problem brojanja parova zečeva koji se pare tako da prvo potomstvo daju dva meseca posle rođenja i nakon toga svaki naredni mesec.

Ovaj pristup generisanju Fibonačijevih brojeva, iako veoma jednostavan za implementaciju, je veoma neefikasan i biće diskutovani mogući pravci poboljšanja. Primetimo da se za razliku od svih do sada prikazanih rekurzivnih funkcija, u sklopu jednog poziva funkcije vrše dva rekurzivna poziva. Ovo samo po sebi ne mora biti problematično, međutim, u ovom konkretnom slučaju se dešava da se rekurzivni pozivi ponavljaju za iste vrednosti parametara, što je jedan od osnovnih problema naivnih rekurzivnih implementacija i što dovodi do ogromne neefikasnosti. Zaista, neka $T(n)$ označava broj instrukcija koje zahteva poziv funkcije `fib` za ulaznu vrednost n. Za $n \leq 1$ važi $T(n) = c_1$, gde je c_1 neka konstanta. Za $n > 1$, važi $T(n) = T(n-1) + T(n-2) + c_2$, gde je c_2 neka konstanta (koja odgovara izvršavanju poređenja, dva oduzimanja, pripremu poziva funkcije za $n-1$ i $n-2$, jedno sabiranje i jedna naredba `return`). 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 $T(n) = a \cdot 1^n + b \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n + c \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n$, odakle sledi da je $T(n) = O(\left(\frac{1+\sqrt{5}}{2}\right)^n)$.

Rekurzivna definicija Fibonačijevog niza opšte je poznata i prethodna funkcija napisana je na osnovu te definicije. Međutim, funkcije sličnog tipa često se dobijaju kada se tehnika rekurzije primeni na rešavanje nekih problema prebrojavanja. Kao ilustraciju rekurzivnog pristupa rešavanju problema, razmotrimo broj načina da se pravougaona tabla dimenzije $2 \times k$ poploča pločicama dimenzije 1×2 i 2×1. Sva popločavanja table dimenzije 2×4 prikazana su na narednoj slici (ima ih 5).



Pokrivanje table dimenzije 4×2 pločicama dimenzije 2×1 i 1×2

Prvo polje na tabli (polje u gornjem levom uglu) mora biti prekriveno dominom. Ona mora biti postavljena ili vertikalno ili horizontalno.

- Ako dominu postavimo vertikalno tada preostaje da se pokrije deo table bez prve kolone. To je problem istog oblika, ali manje dimenzije, pa se može rešiti rekurzivno.
- Dominu možemo postaviti horizontalno samo ako tabla ima bar dve kolone. Pošto i drugo polje u prvoj koloni mora biti pokriveno, ispod prve moramo

postaviti još jednu horizontalnu dominu. Tada preostaje da se pokriju sve kolone table, osim prve dve, što je opet problem istog oblika, ali manje dimenzije, koji se rešava rekurzivno.

Izlaz iz ove rekurzije predstavlja tabla dimenzije 2×0, koja se može popločati samo na jedan način (ne stavljajući ni jednu dominu) i tabla dimenzije 2×1, koja se takođe može popločati samo na jedan način (stavljajući jednu vertikalnu dominu). Ako $F(k)$ označava broj popločavanja table $2 \times k$, važi da je $F(0) = F(1) = 1$ i da je $F(k) = F(k-1) + F(k-2)$, što je upravo jednačina koja definiše Fibonačijev niz.

4.6.2 *Grejovi kodovi*

Binarni brojevi sa n cifara obično seređaju po veličini. Na primer, trocifreni binarni brojevi bi bili poređani u niz 000, 001, 010, 011, 100, 101, 110, 111. Postoji alternativni načinređanja binarnih brojeva koji se, u čast svog izumitelja Frenka Greja, naziva *Grejov kôd*. Grejovi kodovi se koriste za minimalizaciju logičkih funkcija u sklopu metode Karnoovih mapa, a koriste se i za korekciju grešaka u digitalnoj komunikaciji (na primer, u digitalnoj i kablovskoj televiziji). Ključno svojstvo Grejovih kodova je da se svaka dva susedna broja razlikuju tačno na jednom bitu (pri čemu to svojstvo važi i za poslednji i prvi broj u nizu). Trocifreni Grejov kôd može biti 000, 001, 011, 010, 110, 111, 101, 100. Ovo nije jedini trocifreni Grejov kôd, ali se on ipak najčešće razmatra, jer je dobijen vrlo pravilnim, sistematičnim postupkom. Naime, prva četiri broja počinju nulom, dok naredna četiri broja počinju jedinicom. Kada se sa početka prva četiri broja izbriše nula, dobija se dvocifren Grejov kôd 00, 01, 11, 10, dok se brisanje jedinice sa početka naredna četiri broja dobija isti taj Grejov kôd, ali u obratnom poretku 10, 11, 01, 00. I ovaj dvocifreni Grejov kôd može biti dobijen na isti način. Prva dva broja počinju nulom, nakon koje se javlja jednocifreni kôd 0, 1, a druga dva broja počinju jedinicom iza koje se javlja obratan jednocifreni kôd 1, 0. Čak i za jednocifreni kôd možemo konstantovati isto. Prvi broj počinje nulom nakon koje ide Grejov kôd sa nula cifara (koji je prazan), dok drugi broj počinje jedinicom nakon koje ide isti taj kôd sa nula cifara (koji je prazan) u obratnom redosledu.

Definišimo funkciju koja određuje k-ti po redu zapis Grejovog koda sa n cifara. Taj kôd sadrži 2^n n-tocifrenih binarnih brojeva (podrazumevaćemo da je $0 \leq k < 2^n$). Nju je jednostavno definisati rekurzivno. Ako je n nula, rezultat je prazna niska. U suprotnom, treba izračunati neki element Grejovog koda sa $n-1$ cifara

i zatim ga dopuniti sleva nulom ili jedinicom. Treba razlikovati slučaj elemenata u prvoj i u drugoj polovini liste kodova. Pošto ukupno ima 2^n kodova, elementi u prvoj polovini su na pozicijama $0 \leq k < 2^{n-1}$, dok su elementi u drugoj polovini na pozicijama $2^{n-1} \leq k < 2^n$.

- Kada je $0 \leq k < 2^{n-1}$, tada se vraća k-ti element Grejovog koda sa $n-1$ cifara, dopunjen početnom nulom.
- Kada je $2^{n-1} \leq k < 2^n$ tada se vraća $2^n - 1 - k$-ti element Grejovog koda sa $n-1$ cifara, dopunjen početnom jedinicom. Izrazom $2^n - 1 - k$ se pozicija k svodi u raspon $[0, 2^{n-1})$ i ujedno se obrće redosled brojeva. Naime, operacijom $k - 2^{n-1}$ vršimo redukciju intervala $[2^{n-1}, 2^n)$ na interval $[0, 2^{n-1})$. Generalno, prilikom obrtanja redosleda elemenata, svaka pozicija p u intervalu $[0, m)$ se preslikava u poziciju $m - p - 1$ (pozicija 0 se slika u $m-1$, dok se $m-1$ slika u 0). Stoga se prilikom obrtanja intervala $[0, 2^{n-1})$ pozicija $k - 2^{n-1}$ slika u $2^{n-1} - (k - 2^{n-1}) - 1$, no to je jednako $2^n - 1 - k$.

Izračunavanje stepena dvojke najjednostavnije se vrši bitovskim operacijama, pri čemu treba obratiti pažnju na potencijalna prekoračenja. Šiftovanje ulevo za jednu poziciju je ekvivalentno množenju broja sa 2, tako da je šiftovanje broja 1 za n pozicija ulevo ekvivalentno sa n uzastopnih množenja sa 2 i izračunava tačno vrednost 2^n (pod pretpostavkom da ne dođe do prekoračenja).

Elementi Grejovog koda mogu se jednostavno predstaviti u obliku niski karaktera. Iako dopisivanje karaktera na početak niske može biti neefikasna operacija, s obzirom na to da su niske sa kojima radimo prilično kratke (najduža ima 32 karaktera), o tom ne moramo da brinemo.

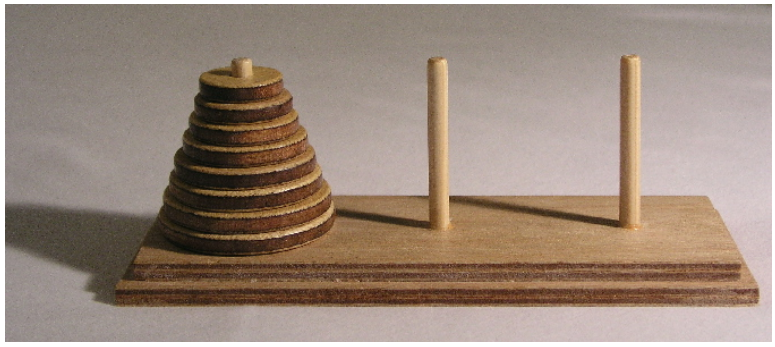
```
string grej(int n, int k)
{
  if (n == 0) return "";
  if (k < (1ul << (n - 1)))
     return "0" + grej(n - 1, k);
  else
     return "1" + grej(n - 1, (1ul << n) - 1 - k);
}
```

Poziv funkcije `grej(5, 17)` vraća nisku `11001`. Zaista, ako obeležimo funkciju skraćemo sa G, važi:

$$\begin{aligned} G(5,17) &= \text{"1"} + G(4,14) = \text{"11"} + G(3,1) = \text{"110"} + G(2,1) \\ &= \text{"1100"} + G(1,1) = \text{"11001"} + G(0,0) = \text{"11001"} \end{aligned}$$

4.6.3 *Hanojske kule*

Problem kula Hanoja[4] glasi ovako: date su tri kule i na prvoj od njih 64 diska opadajućih veličina; zadatak je prebaciti sve diskove sa prve na treću kulu (koristeći i drugu) ali tako da nikada nijedan disk ne stoji iznad nekog manjeg.



Kule Hanoja

Ovaj zadatak jedan je od onih koje je lakše rešiti u opštijem obliku: umesto da smatramo da na prvoj kuli ima 64 diska, smatraćemo da ih ima n.

Iterativno rešenje ovog problema veoma je kompleksno, a rekurzivno prilično jednostavno: ukoliko je $n = 0$, nema diskova koji treba da se prebacuju; inače, prebaci (rekurzivno) $n-1$ diskova sa polaznog na pomoćnu kulu (korišćenjem dolazne kule kao pomoćne), prebaci najveći disk sa polazne na dolaznu kulu i, konačno, prebaci (rekurzivno) $n-1$ diskova sa pomoćne na dolaznu kulu (korišćenjem polazne kule kao pomoćne). U nastavku je implementacija ovog rešenja:

[4] Ovaj problem je u formi autentičnog mita opisao francuski matematičar De Parvil u jednom časopisu 1884.

```
void kule(int n, char polazna, char dolazna, char pomocna)
{
  if (n > 0) {
    kule(n-1, polazna, pomocna, dolazna);
    printf("Prebaci disk sa kule %c na kulu %c\n",
                                  polazna, dolazna);
    kule(n-1, pomocna, dolazna, polazna);
  }
}
```

Poziv navedene funkcije `kule(3,'A','C','B')` daje sledeći izlaz:

```
Prebaci disk sa kule A na kulu C
Prebaci disk sa kule A na kulu B
Prebaci disk sa kule C na kulu B
Prebaci disk sa kule A na kulu C
Prebaci disk sa kule B na kulu A
Prebaci disk sa kule B na kulu C
Prebaci disk sa kule A na kulu C
```

U ovom algoritmu ključna ideja je da se rešenje za n diskova rekurzijom svede na rešenje za $n-1$ diskova, ali nije svejedno kojih $n-1$ diskova. Kada se najveći disk prebaci sa polazne na dolaznu kulu, to ne onemogućava prebacivanje drugih diskova (jer je najveći), te se prirodno može iskoristiti rešenje za $n-1$. S druge strane, ne vodi do rešenja pristup da se koristi rekurzivno rešenje za $n-1$ najvećih diskova.

Izračunajmo broj prebacivanja diskova $T(n)$ koje opisuje navedena funkcija. Važi $T(0) = 0$ i $T(n) = 2T(n-1)+1$ (i $T(1) = 2T(0)+1 = 1$). Dakle, jednačina koja opisuje ovaj niz je nehomogena rekurentna jednačina prvog reda. 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, njena 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 $T(n) = 1 \cdot 2^n + (-1) \cdot 1^n = 2^n - 1$. Potpuno analogno računa se vremenska složenost ove funkcije.

Razmotrimo sada i prostornu složenost $S(n)$ funkcije 'kule'. Jedna instanca funkcije 'kule' zauzima (na programskom steku) konstantan prostor c. Primetimo da se u okviru funkcije 'kule' u jednom trenutku može izvršavati najviše jedan od dva rekurzivna poziva. Zato za prostornu složenost važi:

$$\begin{aligned} S(0) &= c \\ S(n) &= S(n-1) + c \end{aligned}$$

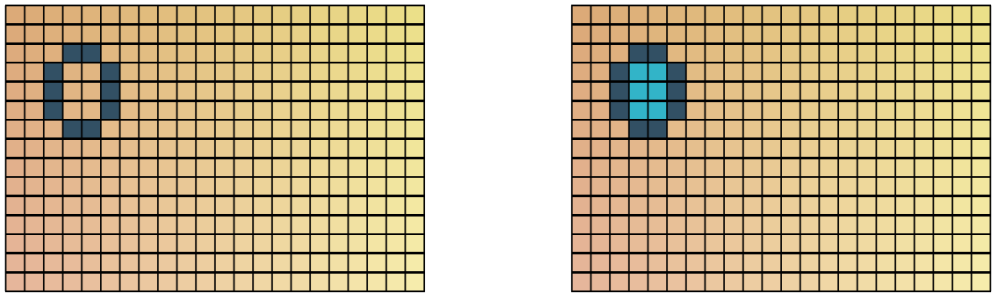
(a ne $S(n) = 2S(n-1) + c$), odakle se dobija da $S(n)$ pripada klasi $O(n)$. Na programskom steku, za ulazni argument n, može se u jednom trenutku naći najviše $n + 1$ stek okvira instanci funkcije 'kule'.

4.6.4 *Popunjavanje konture na slici*

Često postoji potreba da se sistematično obiđu i obrade elementi nekog skupa koji su međusobno povezani. To, na primer, mogu biti elementi neke matrice, istobojna polja na nekoj slici ili gradovi (koji su povezani ako između njih postoji direktan put) i slično. Obilazak svih elemenata do kojih se može stići od nekog početnog elementa često se sprovodi rekurzivnim algoritmima. Ilustrujmo takvu primenu rekurzije ovim i narednim primerom.

Programi za obradu slika obično imaju alat za popunjavanje kontura ("fill" alatka). Potrebno je izabrati piksel određene boje i čitava oblast slične boje koja je ograničena konturom druge boje ili rubom slike biće obojena zadatom bojom (pri čemu smatramo da su pikseli susedni ako imaju jednu stranicu zajedničku). Slika 4.1 prikazuje jednu konturu (levo) i rezultat popunjavanja ako je bila izabrana tačka unutar polazne konture (desno).

Razmotrimo pojednostavljenu varijantu problema: smatrajmo da svi pikseli imaju vrednost 0, a da pikseli koji čine konture imaju vrednost 1 i da tom istom bojom treba obojiti i unutrašnjost izabrane konture. U rekurzivnom rešenju opisanom u nastavku (to je, takozvano, *flood fill* rešenje), kreće se od zadatog piksela, on se boji i onda se ista funkcija poziva za četiri susedna piksela. Za tekući piksel se proverava da li je već obojen ili je van granica slike i ta provera omogućava izlazak iz rekurzije. Slika je predstavljena dvodimenzionim nizom `slika` koji sadrži nule i



Slika 4.1: Popunjavanje konture na slici

jedinice. Jednostavnosti radi pretpostavićemo da je taj niz statički alociran, a da su poznate dimenzije m i n dela koji treba obraditi, kao i koordinate početnog polja v i k.

```
const int MAX_DIM = 50;

void floodFill(int slika[MAX_DIM][MAX_DIM], int m, int n,
               int v, int k)
{
  // da li je polje (v,k) van slike?
  if (!(0 <= v && v < m && 0 <= k && k < n))
    return;

  // da li je polje (v,k) vec obojeno?
  if (slika[v][k])
    return;

  slika[v][k] = 1;
  floodFill(slika, m, n, v+1, k);
  floodFill(slika, m, n, v-1, k);
  floodFill(slika, m, n, v, k-1);
  floodFill(slika, m, n, v, k+1);
}

int main() {
```

```
  int slika[MAX_DIM][MAX_DIM];

  // ucitavamo dimenzije slike
  int m, n;
  cin >> m >> n;

  // ucitavamo polje od kojeg krece popunjavanje
  int x, y;
  cin >> x >> y;

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      char c;
      cin >> c;
      slika[i][j] = (c == '1'); /* '1' oznacava konturu */
    }
  }
  // popunjavanje konture krece od polja (x,y)
  floodFill(slika, m, n, x, y);

  // ispisujemo popunjenu sliku
  cout << endl;
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
      cout << slika[i][j];
    cout << endl;
  }
  cout << endl;
  return 0;
}
```

Program za naredne ulazne podatke:

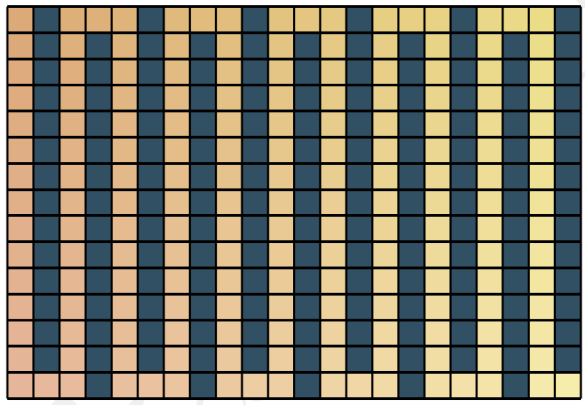
```
5 8
2 2
00100100
01000100
01000100
```

```
00100100
00011111
```

daje sledeći izlaz:

```
00111100
01111100
01111100
00111100
00011111
```

Funkcija `floodFill` ima četiri rekurzivna poziva, te deluje da će lako doći do eksplozije broja polja koja se ispituju. Međutim, ne ulazi se u rekurziju za piksele koji su već obrađeni i obojeni, te je broj piksela koji se obrađuju reda $O(mn)$ i tolika je i vremenska složenost funkcije `floodFill`. Stek okvir za funkciju `floodFill` je konstante veličine, ali je pitanje koliko rekurzivnih poziva može biti aktivno u jednom trenutku, tj. koliko najviše može biti stek okvira na programskom steku. Taj broj je, naravno, ograničen odozgo vrednošću mn, a naredna slika ilustruje jedan tip situacija u kojma je broj stek okvira reda $\Theta(mn)$:



Slika 4.2: Slika na kojoj je rekurzije veoma duboka

Ovaj problem može se rešiti i bez korišćenja rekurzije, ali zato uz korišćenje dodatnih struktura podataka, kao što su stek i red.

4.7 Uzajamna rekurzija

U dosadašnjim primerima, rekurzivne funkcije su pozivale same sebe direktno. Postoji i mogućnost da se funkcije međusobno pozivaju i tako stvaraju *uzajamnu rekurziju*. Poželjno je da prevodilac zna definiciju funkcije pre svakog njenog poziva. Međutim, ako se dve funkcije uzajamno pozivaju, tada to nije moguće (jer u bilo kom redosledu definisanja pre poziva druge funkcije u sklopu prve funkcije definicija te druge funkcije još nije viđena). Pošto je ispravno prevođenje moguće ako se pre poziva zna samo deklaracija (prototip funkcije), rešenje može biti tako što se pre definicija funkcija navedu njihove deklaracije (dovoljno bi bilo navesti i jednu deklaraciju, ali simetrije radi ne smeta da se navedu obe).

Uzajamna rekurzija nije samo sintaksička osobina programskog jezika, već je ponekad i veoma važna tehnika konstrukcije algoritama. Prikažimo to na jednom primeru.

Mnoge konstrukcije u matematici i računarstvu definišu se i obrađuju korišćenjem uzajamne rekurzije. Razmotrimo, na primer, aritmetičke izraze koji sadrže prirodne brojeve i operatore + i *. Svaki izraz je niz sabiraka razdvojenih operatorom + (to uključuje i mogućnost da postoji samo jedan sabirak), svaki sabirak je niz činilaca razdvojenih operatorom * (to uključuje i mogućnost da postoji samo jedan činilac), dok je svaki činilac ili ceo broj ili izraz u zagradama. Dakle, izraz smo definisali preko sabiraka, sabirke preko činilaca, a činioce preko izraza, što predstavlja svojevrsnu uzajamnu rekurziju.

Za obradu izraza u računarstvu često se koristi tehnika poznata pod nazivom *rekurzivni spust*, koja se zasniva na uzajamnoj rekurziji. Funkcije koje čitaju aritmetički izraz i izračunavaju njegovu vrednosti mogu biti definisane na sledeći način (korišćenjem uzajamne rekurzije). Pretpostavićemo da funkcija `procitaj_sledeci_simbol` čita sledeći simbol sa ulaza (to može biti cifra, operator, neka zagrada ili kraj ulaza) i u promenljivoj `sledeci_simbol` beleži o kojoj vrsti simbola se radi, a da prilikom čitanja broja u promenljivu `vrednost_broja` smesti i njegovu brojnu vrednost. Kada se na ulazu pojavi bilo koji simbol koji ne može biti deo izraza, vraća se oznaka da je prepoznat kraj izraza, učitavanje se prekida i ispisuje se vrednost do tada učitanog izraza. Funkcija `main` poziva funkciju `izraz()` i ispisuje vrednost unetog izraza.

```
typedef enum {PLUS = 0, PUTA, BROJ, OTVORENA_ZAGRADA,
              ZATVORENA_ZAGRADA, KRAJ} Simbol;
```

```
Simbol sledeci_simbol;
int vrednost_broja;

void prijavi_gresku();
void procitaj_sledeci_simbol();

// deklaracije funkcija
int izraz();
int sabirak();
int cinilac();

void procitaj_sledeci_simbol()
{
  int c;
  // preskacemo beline na pocetku
  while (isspace(c = getchar()))
    ;

  if (isdigit(c)) {
    sledeci_simbol = BROJ;
    vrednost_broja = c - '0';
    while (isdigit(c = getchar()))
      vrednost_broja = 10 * vrednost_broja + (c - '0');
    ungetc(c, stdin);
  } else {
    switch(c) {
    case '+':
      sledeci_simbol = PLUS;
      break;
    case '*':
      sledeci_simbol = PUTA;
      break;
    case '(':
      sledeci_simbol = OTVORENA_ZAGRADA;
      break;
    case ')':
```

```
      sledeci_simbol = ZATVORENA_ZAGRADA;
      break;
    default:
      sledeci_simbol = KRAJ;
      break;
    }
  }
}

int izraz()
{
  int vrednost = sabirak();
  while (sledeci_simbol == PLUS) {
    procitaj_sledeci_simbol();
    vrednost += sabirak();
  }
  return vrednost;
}

int sabirak()
{
  int vrednost = cinilac();
  while (sledeci_simbol == PUTA) {
    procitaj_sledeci_simbol();
    vrednost *= cinilac();
  }
  return vrednost;
}

int cinilac()
{
  if (sledeci_simbol == BROJ) {
    procitaj_sledeci_simbol();
    return vrednost_broja;
  } else if (sledeci_simbol == OTVORENA_ZAGRADA) {
    procitaj_sledeci_simbol();
```

```
    int vrednost = izraz();
    if (sledeci_simbol != ZATVORENA_ZAGRADA)
      prijavi_gresku();
    procitaj_sledeci_simbol();
    return vrednost;
  } else
    prijavi_gresku();
}

int main()
{
  procitaj_sledeci_simbol();
  cout << izraz() << endl;
  return 0;
}
```

Program za ulaz `(1+2)*(3+45)` daje izlaz `144`, a za ulaz `(1+2.` prijavljuje grešku.

4.8 Dobre i loše strane rekurzije

Dobre strane rekurzije su (obično) čitljiv i kratak kod, jednostavan za razumevanje, analizu, dokazivanje korektnosti i održavanje. Ipak, rekurzivna rešenja imaju i mana.

4.8.1 Cena poziva

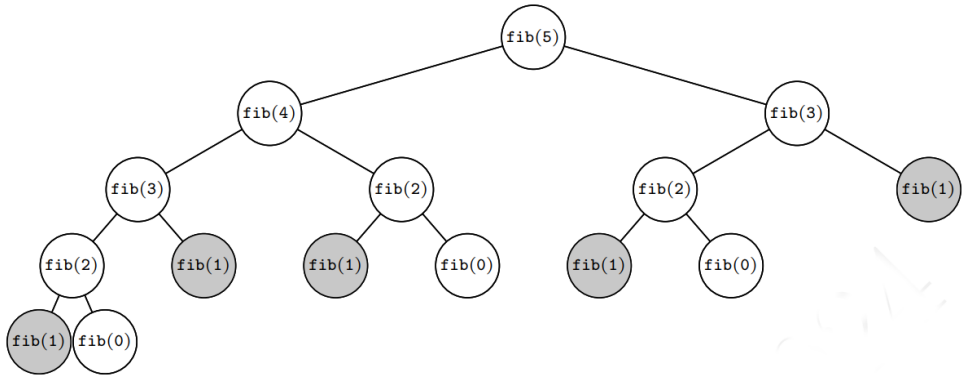
Prilikom svakog rekurzivnog poziva kreira se novi stek okvir i kopiraju se argumenti funkcije. Kada rekurzivnih poziva ima mnogo, to može biti veoma zahtevno u smislu memorije (a donekle i u smislu vremena). Kako je veličina stek segmenta i na savremenim sistemima relativno mala,[5] rekurzivne funkcije mogu dovesti do prekoračenja programskog steka i prekida rada programa. Zato je u nekim situacijama poželjno rekurzivno rešenje zameniti iterativnim (vidi poglavlje 4.9). Jedna

[5] Iako današnji računari imaju nekoliko gigabajta radne memorije, programi na raspolaganju obično imaju svega nekoliko megabajta za programski stek. Predefinisana veličina programskog steka može se promeniti zadavanjem odgovarajuće opcije kompilatoru, u nekoj meri, u zavisnosti od konkretnog sistema.

opšta preporuka je da bi dubina rekurzije trebalo da raste znatno sporije od dimenzije ulaza (jer dimenzija ulaza koja se može obraditi je relativno mala čak i u slučaju kada dubina rekurzije linearno zavisi od dimenzije ulaza).

4.8.2 *Suvišna izračunavanja*

U nekim slučajevima prilikom razlaganja problema na manje potprobleme dolazi do preklapanja potproblema i do višestrukih rekurzivnih poziva za iste potprobleme.



Slika 4.3: Ilustracija ponovljenih izračunavanja prilikom izvršavanja rekurzivne funkcije

Razmotrimo, na primer, izvršavanje navedene funkcije koja izračunava elemente Fibonačijevog niza za $n = 5$. U okviru tog izvršavanja, funkcija `fib` je tri puta pozvana za $n = 0$, pet puta za $n = 1$, i tako dalje. Naravno, na primer, poziv `fib(1)` je svaki put izvršavan iznova i nije korišćena vrednost koju je vratio prethodni takav poziv (slika 4.3). Zbog ovoga, izvršava se mnogo suvišnih izračunavanja i količina takvih izračunavanja u ovom primeru raste. Rešenje ovog problema opisano je u poglavlju **??**.

Treba voditi računa o tome da se ponovljena izračunavanja greškom ne uvedu tamo gde ih je veoma jednostavno izbeći. Razmotrimo narednu rekurzivnu implementaciju funkcije za izračunavanje minimuma nepraznog niza.

```
int minimum(int a[], int n) {
  if (n == 1) return a[0];
  if (a[n-1] < miminum(a, n-1))
     return a[n-1];
  else
     return minimum(a, n-1);
}
```

U slučaju jednočlanog niza minimum je njegov jedini element. Ako niz ima više elemenata, onda se rekurzivno izračunava minimum prefiksa i rezultat je manji od te vrednosti i vrednosti poslednjeg elementa. Međutim, u prethodnoj implementaciji se može desiti da se u istom pozivu dva puta rekurzivno pozove `minimum(a, n-1)`, što za posledicu ima eksponencijalnu složenost ove funkcije. Ovo se lako može rešiti tako što se rezultat poziva zapamti u pomoćnoj promenljivoj.

```
int minimum(int a[], int n) {
  if (n == 1) return a[0];
  int M = miminum(a, n-1);
  if (a[n-1] < M)
     return a[n-1];
  else
     return M;
}
```

Alternativno, možemo upotrebiti i bibliotečku funkciju `min`, koja pronalazi minimum dva broja.

```
int minimum(int a[], int n) {
  if (n == 1) return a[0];
  return min(a[n-1], miminum(a, n-1));
}
```

4.9 Oslobađanje od rekurzije

Svaku rekurzivnu funkciju je moguće implementirati na drugi način tako da ne koristi rekurziju. Ne postoji jednostavan opšti postupak za generisanje takvih al-

ternativnih implementacija. Opšti postupak bi uključivao implementaciju strukture podataka u koju bi eksplicitno bili smeštani podaci koji se inače smeštaju na programski stek. Pošto je programski stek prilično ograničen, ovim se mogu sprečiti neke greške prekoračenja steka, ali takvi programi i dalje koriste veliku količinu memorije.

Sistematičan postupak eliminacije rekurzije postoji za neke specijalne slučajeve. Rekurzija se obično može eliminisati iz onih funkcija u kojima se tokom svakog izvršavanja rekurzivne funkcije izvrši najviše jedan rekurzivni poziv (u slučajevima kada jedan rekurzivni poziv proizvede više novih rekurzivnih poziva, eliminacija rekurzije je obično neizvodiva bez korišćenja pomoćnih struktura podataka koje emuliraju programski stek).

4.9.1 *Repna rekurzija*

Naročito je zanimljiv slučaj *repne rekurzije*, jer se u tom slučaju rekurzija može jednostavno eliminisati na veoma sistematičan način[6]. Rekurzivni poziv je *repno rekurzivni* ukoliko je vrednost rekurzivnog poziva upravo i konačan rezultat funkcije, tj. nakon rekurzivnog poziva ne izvršava se nikakva dodatna naredba (uključujući ni bilo kakva ispisivanja). Na primer, u funkciji

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

prvi rekurzivni poziv je repno-rekurzivan, dok drugi nije (zato što je po izlasku iz rekurzije neophodno još pomnožiti rezultat sa x). Za rekurzivnu funkciju kažemo da je *repno-rekurzivna* ako joj je svaki rekurzivni poziv repni.

[6] Neki kompilatori (pre svega za funkcionalne programske jezike) automatski detektuju repno rekurzivne pozive i eliminišu ih.

Izvršavanje repno-rekurzivne funkcije omogućava određenu optimizaciju. Pošto nakon povratka iz repnog rekurzivnog poziva neće biti više potrebni podaci koji se nalaze u tekućem stek okviru, nema potrebe za rekurzivni poziv alocirati novi stek okvir, već je moguće podatke koji odgovaraju rekurzivnom pozivu sačuvati u tekućem stek okviru. Na taj način se celokupno izvršavanje repno-rekurzivne funkcije realizuje korišćenjem samo jednog stek okvira. Mnogi kompilatori automatski vrše ovu optimizaciju. S druge strane, moguće je da programer samostalno izvrši odgovarajuću optimizaciju, tako što će eliminisati rekurziju i zameniti je petljom (u kojoj će se na mestu rekurzivnog poziva, na kraju tekuće iteracije, ažurirati vrednosti promenljivih koje odgovaraju ulaznim parametrima funkcije).

```
void r(int x)
{
  if (p(x))
    a(x);
  else {
    b(x);
    r(f(x));
  }
}
```

gde su `p`, `a`, `b` i `f` proizvoljne funkcije.

Ključni korak je da se pre rekurzivnog koraka vrednosti parametara funkcije (u ovom slučaju parametra `int x`) postave na vrednosti parametara rekurzivnog poziva, a zatim da se kontrola toka izvršavanja nekako prebaci na početak funkcije. Ovo je najjednostavnije (ali ne previše elegantno) uraditi korišćenjem bezuslovnog skoka.

```
void r(int x)
{
pocetak:
  if (p(x))
    a(x);
  else {
    b(x);
    x = f(x);
```

```
    goto pocetak;
  }
}
```

Daljom analizom moguće je ukloniti bezuslovni skok i dobiti sledeći iterativni kôd.

```
void r(int x)
{
  while (!p(x)) {
    b(x);
    x = f(x);
  }
  a(x);
}
```

Primer 4.9.1. *Demonstrirajmo tehniku uklanjanja repne rekurzije i na primeru Euklidovog algoritma.*

```
unsigned nzd(unsigned a, unsigned b)
{
  if (b == 0)
    return a;
  else
    return nzd(b, a % b);
}
```

Pošto je poziv repno-rekurzivan, potrebno je pripremiti nove vrednosti parametara `a` *i* `b` *i preneti kontrolu izvršavanja na početak funkcije.*

```
unsigned nzd(unsigned a, unsigned b)
{
pocetak:
  if (b == 0)
    return a;
  else {
    unsigned tmp = a % b; a = b; b = tmp;
```

```
    goto pocetak;
  }
}
```

Daljom analizom, jednostavno se uklanja naredba `goto` *i dobija identičan kôd onom koji smo ranije prikazali.*

```
unsigned nzd(unsigned a, unsigned b)
{
  while (b != 0) {
    unsigned tmp = a % b; a = b; b = tmp;
  }
  return a;
}
```

Izračunavanje Grejovih kodova (poglavlje 4.6.2) takođe je zasnovano na repnoj rekurziji, pa se i ta funkcija može jednostavno definisati nerekurzivno:

```
void grej(unsigned n, unsigned k, char kod[])
{
  int i;
  i = 0;
  while (n > 0) {
    if (k < 1u << (n-1))
      kod[i] = '0';
    else {
      kod[i] = '1';
      k = (1ul << n) - 1 - k;
    }
    n--;
    i++;
  }
  kod[i] = '\0';
}
```

S obzirom na svojstva repne rekurzije, ponekad deluje poželjno rekurzivne funkcije formulisati tako da koriste iskjučivo repnu rekurziju. Na primer, pokušajmo

da implementiramo repno-rekurzivnu funkciju za izračunavanje n-tog Fibonačijevog broja. U narednoj implementaciji, u svakom pozivu rekurzivne funkcije `fib_` njoj se, uz broj n, prosleđuju i dve uzastopne vrednosti Fibonačijevog niza F_k (u promenljivoj `fpp`) i F_{k+1} (u promenljivoj `fp`). Funkcija onda na osnovu njih izračunava sledeću vrednost F_{k+2} i zatim narednom rekurzivnom pozivu prosleđuje vrednosti $n-1$, F_{k+1} i F_{k+2}. Pošto se u početnom rekurzivnom pozivu prosleđuju početna vrednost n (nazovimo je n_0) i vrednosti $F_0 = 0$ i $F_1 = 1$, sve vreme važi invarijanta da je $k + n = n_0$. Izlaz iz rekurzije je slučaj kada je $n = 0$, pa se tada tražena vrednost nalazi u promenljivoj $F_k = F_{n_0}$.

```
unsigned fib(unsigned n, unsigned fp, unsigned fpp)
{
  if (n == 0)
    return fpp;
  return fib(n-1, fp+fpp, fp);
}

unsigned fib(unsigned n)
{
  return fib(n, 1, 0);
}
```

Izvršavanje broja F_5 primenom ove funkcije se može opisati sledećim nizom jednakosti:

$$\begin{aligned} \texttt{fib}(5) &= \texttt{fib}(5,1,0) = \texttt{fib}(4,1,1) = \texttt{fib}(3,2,1) \\ &= \texttt{fib}(2,3,2) = \texttt{fib}(1,5,3) = \texttt{fib}(0,8,5) = 5 \end{aligned}$$

Eliminacijom ove repne rekurzije dobijamo veoma elegantnu iterativnu implementaciju.

```
unsigned fbb(unsigned n)
{
  unsigned fp = 1, fpp = 0;
  while (n > 0) {
```

```
    int f = fp + fpp;
    fpp = fp; fp = f;
    n--;
  }
  return fpp;
}
```

Primetimo da se tokom izvršavanja repno-rekurzivnih funkcija parametri tokom rekurzivnih poziva postepeno menjaju od nekih početnih vrednosti (u primeru Fibonačijevih brojeva to su vrednosti $F_0 = 0$ i $F_1 = 1$) pa sve do krajnjih, traženih vrednosti (u primeru Fibonačijevih brojeva to su vrednosti F_n i F_{n+1}). Takva izmena promenljivih suštinski je iterativna (promenljive se na potpuno isti način menjaju i u iterativnoj verziji, sa petljama) i odgovarajuće repno-rekurzivne funkcije često se nazivaju *iterativne funkcije*. Pošto je imperativnim jezicima, kakav je C, takva promena promenljivih karakteristična je za petlje, u njima retko kada pišu repno-rekurzivne funkcije (kada programer osmisli rešenje u kojem se promenljive menjaju na opisani način, on po običaju takvo rešenje odmah formuliše u vidu petlji).

I rekurzija koja nije repno-rekurzivna često se može ukloniti i zameniti jednostavnom iteracijom, ali to nije jednostavno uraditi na sistematičan način, kao u slučaju repne rekuzije.

Primer 4.9.2. *Pokažimo kako se može eliminisati rekurzija iz funkcije za brzo stepenovanje navedene na početku ovog poglavlja (videti primer 4.4.4). Postupak izračunavanja vrednosti* 2^{10} *tom funkcijom može se opisati sledećim nizom jednakosti:* $2^{10} = 4^5 = 4 \cdot 4^4 = 4 \cdot 16^2 = 4 \cdot 256^1 = 4 \cdot 256 \cdot 256^0 = 1024 \cdot 1 = 1024$ *U svakom koraku proizvod* $s \cdot x^k$ *ima konstantnu vrednost, pri čemu je* s *dodatna promenljiva koja u prva dva koraka ima vrednost* 1*. Napišimo i kako se promenljive* s*,* x *i* k *menjaju tokom rekurzivnih poziva.*

s	x	k
1	*2*	*10*
1	*4*	*5*
4	*4*	*4*
4	*16*	*2*
4	*256*	*1*
1024	*256*	*0*

Možemo uočiti dve vrste koraka, u zavisnosti od toga da li je k paran ili neparan broj. Ako je k paran broj, tada se vrednost x kvadrira, a k se deli sa 2. Ako je k neparan broj, tada se vrednost s množi vrednošću x, dok se vrednost k smanjuje za 1. Postupak se završava kada je $k = 0$ i konačan rezultat je tekuća vrednost promenljive s. Dakle, iterativni algoritam koji izvršava brzo stepenovanje može se implementirati na sledeći način:

```
unsigned stepen(unsigned x, unsigned k)
{
  unsigned s = 1;
  while (k > 0) {
    if (k % 2 == 0) {
      x = x * x;
      k = k / 2;
    } else {
      s = s * x;
      k = k - 1;
    }
  }
  return s;
}
```

U slučajevima u kojima funkcija tokom jednog izvršavanja pravi više rekurzivnih poziva

4.9.2 *Oslobađanje od rekurzije korišćenje pogodnih struktura podataka*

Za mnoge probleme rekurzivno rešenje može biti veoma elegantno ali može biti i zahtevno u smislu potrebnog vremena ili prostora za izvršavanje. Čak i ako rekurzivna funkcija zahteva mali stek okvir, ukoliko ima mnogo rekurzivnih poziva može doći do prepunjavanja steka i prekida izvršavanja programa. Broj rekurzivnih poziva koji može dovesti do prekida izvršavanja programa zavisi od mnogo faktora, ali obično je reda nekoliko desetina hiljada pre neko nekoliko miliona. U situacijama kada je moguće da broj rekurzivnih poziva bude veliki poželjno je rekurzivno rešenje zameniti nerekurzivnim. Kao što je pokazano, u nekim situacijama (za neke forme rekurzije kao što je repna rekurzija) dovoljna je relativno

jednostavna transformacija programa. U nekim situacija, međutim, potrebne promene mogu iziskivati potpunu promenu logiku programa i oslanjanje na dodatne pogodne strukture podataka.

Razmotrimo ponovo problem popunjavanja konture (koji je opisan u poglavlju **??**). Ovaj problem može biti rešen i bez korišćenja rekurzije. Međutim, umesto rekurzije biće potrebna neka pogodna dinamička struktura, i u ovom slučaju to će biti stek. Elementi steka će biti polja mape, tj. parovi celih brojeva.

U prikazanom rešenju, kreće se od polaznog polja (polja (x, y)), ono se stavlja na vrh steka i onda, dok god ih ima, obrađuje jedno po jedno polje sa vrha steka. Na vrh steka stavljaju se sva susedna polja tekućeg polja koja nisu prepreke. Koordinate susednih polja određuju se jednostavno, koristeći pomoćni niz 'smer' koji sadrži pomeraje za četiri moguća smera.

```
const int MAX_DIM = 50;
const int smer[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

int main() {
  int slika[MAX_DIM][MAX_DIM];

  // ucitavamo dimenzije slike
  int m, n;
  cin >> m >> n;

  // ucitavamo polje od kojeg krece popunjavanje
  int x, y;
  cin >> x >> y;

  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
      char c;
      cin >> c;
      slika[i][j] = (c == '1'); /* '1' oznacava konturu */
    }
  }
  stack<pair<int,int>> s;
  s.push(make_pair(x,y));
```

```
  while (!s.empty()) {
    pair<int,int> p = s.top();
    s.pop();
    for (int i = 0; i < 4; i++) {
      int x = p.first + smer[i][0];
      int y = p.second + smer[i][1];
      if (0 <= x && x < m && 0 <= y && y < n && !slika[x][y]) {
        s.push(make_pair(x,y));
        slika[x][y] = 1;
      }
    }
  }

  cout << endl;
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++)
      cout << slika[i][j];
    cout << endl;
  }
  cout << endl;
  return 0;
}
```

4.10 Ojačavanje induktivne hipoteze

Prilikom primene induktivno-rekurzivne konstrukcije ponekada se ispostavlja da samo rešenje potproblema manje dimenzije nije dovoljno da bi se rekonstruisalo rešenje polaznog problema, ali da se prilikom rekurzivnog rešavanja potproblema može uraditi i nešto dodatno (na primer, izračunati neke pomoćne informacije) koje nam onda pomažu da rešimo i polazni problem. Dakle, do rešenja je moguće doći zahvaljujući "kreditu" koji smo dobili iz rekurzivnog poziva, ali ne smemo zaboraviti da vratimo taj "dug" tj. da našem pozivaocu pored traženog rešenja problema uradimo i taj dodatni posao (na primer, vratimo te pomoćne informacije) — iako to nije potrebno u polaznom rekurzivnom pozivu, na svim drugim nivoima rekurzije jeste. Takoće, i baza mora biti proširena tako da obuhvati ovaj dodatni posao.

Ova tehnika se naziva *ojačanje induktivne hipoteze*. U slučaju ojačanja induktivne hipoteze, klasičnu shemu

$$A_0 \wedge ((\forall n)(A_n \Rightarrow A_{n+1})) \Rightarrow (\forall n)(A_n)$$

menjamo shemom

$$(A_0 \wedge B_0) \wedge ((\forall n)(A_n \wedge B_n \Rightarrow A_{n+1} \wedge B_{n+1})) \Rightarrow (\forall n)(A_n \wedge B_n).$$

Ilustrujmo ovu tehniku kroz rešenje nekoliko zadataka. Krenimo od problema određivanja maksimalnog zbira segmenta niza, koji smo već uspešno rešavali primenom različitih algoritamskih tehnika. Zadatak ćemo rešiti induktivnom konstrukcijom. U prvom koraku ćemo prkazati rešenje koje koristi pomoćni niz i dva prolaza kroz podatke, dok ćemo u drugom rešenju upotrebiti tehniku ojačanja induktivne hipoteze da bismo se oslobodili potrebe za dodatnom pomoćnom memorijom i zadatak rešili u samo jednom prolazu.

Zadatak: Maksimalni zbir segmenta

Ovaj zadatak je ponovljen u cilju uvežbavanja različitih tehnika rešavanja. Vidi tekst zadatka.

Pokušaj da zadatak uradiš korišćenjem tehnika koje se izlažu u ovom poglavlju.

Rešenje

Maksimalni sufiksi

Svaki segment je sufiks nekog prefiksa niza. Pošto se sufiksi mogu analizirati inkrementalno (svi sufiksi $i+1$-vog prefiksa niza se dobijaju proširivanjem svih sufiksa i-tog prefiksa niza dodatnim elementom), problem analize svih segmenata je poželjno svesti na problem analize sufiksa.

Zadatak može da se svede na to da se za svaki prefiks u nizu odredi maksimalan sufiks, a da se onda među maksimalnim sufiksima za svaki prefiks pronađe onaj sufiks koji je globalno maksimalan.

Jedini sufiks prvih nula elemenata niza je prazan i njegov zbir je po definiciji 0. Svi sufiksi prvih $i+1$ elemenata niza, izuzev praznog sufiksa dobijaju se dodavanjem

elementa na poziciji i na kraj nekog sufiksa prvih i elemenata niza. Među nepraznim sufiksima najveći zbir ima onaj koji je dobijen dodavanjem poslednjeg elementa upravo na sufiks prvih i elemenata niza koji ima maksimalni zbir. Od njega jedino može biti veći zbir praznog sufiksa (i to kada se nakon proširivanja prethodno maksimalnog sufiksa poslednjim elementom dobije negativan zbir). Ako vrednosti maksimalnog zbira sufiksa pamtimo u nizu, tada niz lako popunjavamo na osnovu veza $S_0 = 0$ i $S_{i+1} = \max(S_i + a_i, 0)$, gde je sa S_i obeležena vrednost maksimalnog zbira sufiksa prvih i elemenata niza a.

Na kraju nalazimo maksimum niza S_i.

Primer 4.10.1. *Prikažimo rad algoritma na primeru niza* `-2 3 2 -3 -3 -2 4 5 -8 3`. *U tablici popunjavamo vrednosti* S_i.

i	a_{i+1}	S_i
0		0
1	-2	$0 = \max(0 + (-2), 0)$
2	3	$3 = \max(0 + 3, 0)$
3	2	$5 = \max(3 + 2, 0)$
4	-3	$2 = \max(5 + (-3), 0)$
5	-3	$0 = \max(2 + (-3), 0)$
6	-2	$0 = \max(0 + (-2), 0)$
7	4	$4 = \max(0 + 4, 0)$
8	5	$9 = \max(4 + 5, 0)$
9	-8	$1 = \max(9 + (-8), 0)$
10	3	$4 = \max(1 + 3, 0)$

Maksimalna vrednost u koloni S_i *je* 9.

Pošto koristimo niz maksimalnih zbirova sufiksa, memorijska složenost je $O(n)$. Niz popunjavamo element po element, inkrementalno, u jednom prolazu za šta je dovoljno n koraka, a zatim maksimum nalazimo u novom prolazu tj. u novih n koraka. Ukupna složenost je, dakle, linearna tj. $O(n)$.

```
// maksimalni sufiks prvih i elemenata niza
vector<int> maxSufiks(n+1);
maxSufiks[0] = 0;
```

```
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks[i+1] = max(maxSufiks[i] + x, 0);
}

// maksimalni segment je maksimalni od svih maksimalnih sufiksa
cout << *max_element(begin(maxSufiks), end(maxSufiks)) << endl;
```

Kadanov algoritam

Maksimalne vrednosti zbirova sufiksa ne moramo da pamtimo u nizu, ako istovremeno sa njihovim određivanjem održavamo i njihov maksimum. Ovaj algoritam poznat je pod imenom *Kadanov algoritam*.

Jedan način da se dođe do tog algoritma je sledeći. Pokušavamo da algoritam zasnujemo na induktivnoj konstrukciji.

- Za prazan niz, jedini segment je prazan i njegov je zbir nula (to je ujedno najveći zbir koji se može dobiti).
- Smatramo da umemo da problem rešimo za proizvoljan niz dužine n i na osnovu toga pokušavamo da rešimo zadatak za niz dužine $n+1$ (polazni niz proširen jednim dodatnim elementom).

  Segment najvećeg zbira u proširenom nizu se ili ceo sadrži u polaznom nizu dužine n ili čini sufiks proširenog niza, tj. završava se na poslednjoj poziciji (uključujući i mogućnost da je tu i prazan segment).

  Na osnovu induktivne hipoteze znamo da izračunamo najveći zbir segmenta niza dužine n i potrebno je da još odredimo maksimalni zbir sufksa proširenog niza. Jedan način da se to uradi je da prilikom svakog proširenja niza iznova analiziramo sve segmente koji se završavaju na tekućoj poziciji, ali čak iako to radimo inkrementalno (krenuvši od praznog sufiksa, pa dodajući unazad jedan po jedan element) najviše što možemo dobiti je algoritam kvadratne složenosti (probajte da se uverite da je to zaista tako). Ključni uvid je to da najveći zbir sufiksa koji se završava na tekućoj poziciji možemo inkrementalno izračunati znajući najveći zbir sufiksa niza pre proširenja. Najveći

zbir nekog nepraznog sufiksa koji se završava na tekućoj poziciji je zbir tekućeg elementa niza i najvećeg zbira nekog sufiksa koji se završava na prethodnoj poziciji. Od njega može biti povoljniji samo prazan sufiks (i to samo ako je prethodni zbir negativan).

Dakle, ako sa S_i obeležimo maksimalni zbir nekog sufiksa prvih i elemenata niza, a sa M_i maksimalni zbir nekog segmenta prvih i elemenata niza, važi da je $M_0 = S_0 = 0$, da je $S_{i+1} = \max(S_i + a_i, 0)$ i $M_{i+1} = \max(M_i, S_{i+1})$.

Implementaciju možemo napraviti iterativnim algoritmom kome je invarijanta da u svakom koraku petlje znamo ove dve vrednosti (maksimum segmenta i maksimum sufiksa).

Primer 4.10.2. *Prikažimo rad algoritma na primeru niza* `-2 3 2 -3 -3 -2 4 5 -8 3`. *U tablici popunjavamo vrednosti* S_i *i* M_i.

i	a_{i+1}	S_i	M_i
0		0	
1	-2	$0 = \max(0 + (-2), 0)$	$0 = \max(0, 0)$
2	3	$3 = \max(0 + 3, 0)$	$3 = \max(0, 3)$
3	2	$5 = \max(3 + 2, 0)$	$5 = \max(3, 5)$
4	-3	$2 = \max(5 + (-3), 0)$	$5 = \max(5, 2)$
5	-3	$0 = \max(2 + (-3), 0)$	$5 = \max(5, 0)$
6	-2	$0 = \max(0 + (-2), 0)$	$5 = \max(5, 0)$
7	4	$4 = \max(0 + 4, 0)$	$5 = \max(5, 4)$
8	5	$9 = \max(4 + 5, 0)$	$9 = \max(5, 9)$
9	-8	$1 = \max(9 + (-8), 0)$	$9 = \max(9, 1)$
10	3	$4 = \max(1 + 3, 0)$	$9 = \max(9, 4)$

Pošto elemente učitavamo jedan po jedan i ne pamtimo ih istovremeno, memorijska složenost je $O(1)$. Maksimalni zbir segmenta i sufiksa inkrementalno izračunavamo jednim prolaskom kroz zadate elemente i vremenska složenost je linearna tj. $O(n)$.

Primetimo da smo u prethodnom razmatranju proširili induktivnu hipotezu pretpostavljajući da pored tražene vrednosti tj. maksimuma nekog segmenta prvih n elemenata niza znamo dodatno i vrednost maksimalnog sufiksa prvih n elemenata niza.

```
int maxSufiks = 0, maxSegment = maxSufiks;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  maxSufiks += x;
  if (maxSufiks < 0)
    maxSufiks = 0;
  if (maxSufiks > maxSegment)
    maxSegment = maxSufiks;
}
cout << maxSegment << endl;
```

Zadatak: Maksimalni zbir nesusednih elemenata niza

Napiši program koji određuje najveći zbir podniza datog niza celih brojeva koji ne sadrži dva uzastopna člana niza.

Opis ulaza

Sa standardnog ulaza se unosi broj članova niza n ($1 \leq n \leq 10^5$), a zatim iz narednog reda članovi niza razdvojeni razmacima.

Opis izlaza

Na standardni izlaz ispisati traženi maksimalni zbir.

Primer 1

Ulaz	*Izlaz*	*Objašnjenje*
`6` `7 3 2 4 1 5`	`16`	Maksimalni zbir je zbir segmenta $7 + 4 + 5 = 16$.

Primer 2

Ulaz	*Izlaz*
`12` `3 -2 4 5 -1 3 -4 -5 4 5 2 -1`	`17`

Rešenje

Zadatak možemo jednostavno rešiti induktivno-rekurzivnom konstrukcijom. Niz možemo razložiti na poslednji element i prefiks pre njega. Da bismo odredili mak-

simalni zbir nesusednih elemenata niza, analiziraćemo slučaj kada je poslednji element niza deo tog maksimalnog zbira i kada nije (to su jedine dve mogućnosti i jedna od njih je sigurno tačna). Zato je maksimalni zbir nesusednih elemenata niza maksimum sledeća dva zbira:

- prvog, koji se dobija tako što se poslednji element doda na maksimalni zbir nesusednih elemenata prefiksa, pri čemu u tom zbiru ne uključuje pretposlednji element niza (tj. poslednji element prefiksa) i
- drugog, koji se dobija kao maksimalni zbir nesusednih elemenata prefiksa koji može da uključi i pretposlednji element niza (tj. poslednji element prefiksa).

Dakle, osim što pretpostavljamo da umemo da rešimo problem manje dimenzije tj. da za prefiks umemo da odredimo maksimalni zbir nesusednih elemenata, potrebno je da ojačamo induktivnu hipotezu i da za prefiks umemo da odredimo i maksimalni zbir nesusednih elemenata tog prefiksa ako poslednji element prefiksa nije uključen u taj zbir. Naravno, kao i uvek kada se ojačava induktivna hipoteza, "dug" se mora vratiti pa za prefiks proširen dodatnim elementom pored maksimalnog zbira nesusednih elemenata, moramo da umemo da odredimo i maksimalni zbir nesusednih elemenata kada poslednji element nije uključen. Međutim, to nije teško, jer je to upravo maksimalni zbir nesusednih elemenata prefiksa. Maksimalni zbir proširenog niza (bez obzira na to da li uključuje ili ne uključuje poslednji element) određujemo kao maksimum dva opisana zbira (koja na osnovu induktivne hipoteze lako izračunavamo).

Izlaz iz rekurzije je slučaj praznog niza. Maksimalni zbir njegovih nesusednih elemenata u svakoj varijanti jednak je nuli.

Dakle, obeležimo sa m_k maksimalni zbir nesusednih elemenata prvih k elemenata datog niza. Cilj je da odredimo m_n. Neka je m'_k maksimali zbir nesusednih elemenata prvih k elemenata niza u koji nije uključen element a_{k-1}. Važe sledeće rekurentne relacije: $m_0 = m'_0 = 0$, dok za $k > 0$, važi $m'_k = m_{k-1}$ i $m_k = \max(a_{k-1} + m'_{k-1}, m_{k-1})$.

Složenost ovog algoritma je $O(n)$.

```
int maks_zbir_bez = 0;
int maks_zbir = 0;
for (int i = 0; i < n; i++) {
  int x;
  cin >> x;
  int maks_zbir_sa = maks_zbir_bez + x;
  maks_zbir_bez = maks_zbir;
  maks_zbir = max(maks_zbir_bez, maks_zbir_sa);
}
cout << maks_zbir << endl;
```

Još jedna induktivno-rekurzivna konstrukcija kojom se problem može rešiti podrazumeva da se za svaki prefiks niza odredi maksimalni zbir nesusednih elemenata kada je poslednji element prefiksa uključen i maksimalni zbir nesusednih elemenata kada poslednji element prefiksa nije uključen.

Obeležimo sa m_k^{sa} maksimalni zbir nesusednih elemenata prvih k elemenata niza, koji sadrži i element a_{k-1} i sa m_k^{bez} maksimalni zbir nesusednih elemenata prvih k elemenata niza, koji ne sadrži element a_{k-1}. Baza indukcije može biti jednočlan niz i važi $m_1^{sa} = a_0$, $m_1^{bez} = 0$. Za svaku vrednost $k > 1$, važi $m_k^{sa} = a_{k-1} + m_{k-1}^{bez}$ i $m_k^{bez} = \max(m_{k-1}^{sa}, m_{k-1}^{bez})$. Tražena vrednost jednaka je $\max(m_n^{sa}, m_n^{bez})$.

Složenost ovog algoritma je $O(n)$.

```
// prvi element se ucitava van petlje zbog inicijalizacije
// to je ujedno i baza indukcije, jednoclani prefiks niza
int x;
cin >> x;
int maks_zbir_bez = 0;
int maks_zbir_sa = x;

// u petlji obradjujemo tekuci element
for (int i = 1; i < n; i++) {
  int x;
  cin >> x;
  int maks_zbir = max(maks_zbir_sa, maks_zbir_bez);
```

```
  maks_zbir_sa = maks_zbir_bez + x;
  maks_zbir_bez = maks_zbir;
}

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

Induktivno-rekurzivna konstrukcija može teći i na sledeći način. Pretpostavljamo da umemo da izračunamo maksimalni zbir nesusednih elemenata svakog prefiksa niza. Ako je prefiks prazan, maksimalni zbir nesusednih elemenata je nula, a ako je jednočlan, tada je jednak većem od broja nula i prvog elementa niza. Ako je prefiks bar dvočlan onda je maksimalni zbir nesusednih elemenata tog prefiksa maksimum maksimalnog zbira prefiksa bez poslednjeg elementa i zbira poslednjeg elementa i maksimalnog zbira bez poslednja dva elementa.

Dakle, dobijamo da je $m_0 = 0$, $m_1 = \max(a_0, 0)$ i $m_i = \max(m_{i-1}, a_{i-1} + m_{i-2})$.

Ova rekurzivna konstrukcija se može isprogramirati rekurzivnom funkcijom. Nažalost, to rešenje je prilično neefikasno.

Složenost ovog rešenja zadovoljava jednačinu $T(n) = T(n-1) + T(n-2) + O(1)$, $T(1) = T(0) = 1$, čije je rešenje eksponencijalna funkcija (ista jednačina se javlja i prilikom direktne rekurzivne implementacije izračunavanja elemenata Fibonačijevog niza).

```
int maksZbir(const vector<int>& a, int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return max(a[0], 0);
  return max(maksZbir(a, n-1), a[n-1] + maksZbir(a, n-2));
}
```

Rekurziju možemo ukloniti i dobiti narednu iterativnu implementaciju.

Složenost ovog algoritma je $O(n)$.

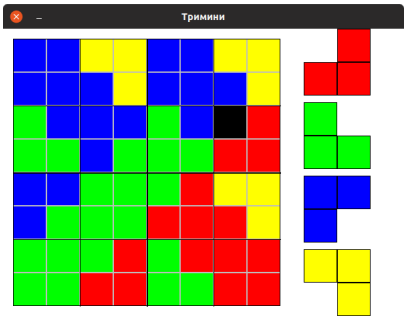
```
int maks_zbir_p = 0;
int x;
cin >> x;
int maks_zbir = max(0, x);
for (int i = 2; i <= n; i++) {
  int x;
  cin >> x;
  int tmp = max(maks_zbir, maks_zbir_p + x);
  maks_zbir_p = maks_zbir;
  maks_zbir = tmp;
}
cout << maks_zbir << endl;
```

Sistematičan način oslobađanja neefikasnosti prouzrokovane rekurzijom ovog tipa dolazi u obliku tehnika dinamičkog programiranja, koje je opisano u zasebnom poglavlju.

5. Tehnika podeli-pa-vladaj

Tehnika *"podeli-pa-vladaj"* (engl. divide-and-conquer) ili *dekompozicija* podrazumeva da se problem rešava tako što se razloži na dva ili više manjih problema, koji se zatim nezavisno rekurzivno rešavaju. Obično su najveći izazovi u primeni ove tehnike osmisliti kako da se problem razloži na potrobleme i kako da se od rešenja potproblema dobije rešenje polaznog problema (oba ova koraka mogu zahtevati određeno vreme koje onda određuje ukupnu efikasnost algoritma).

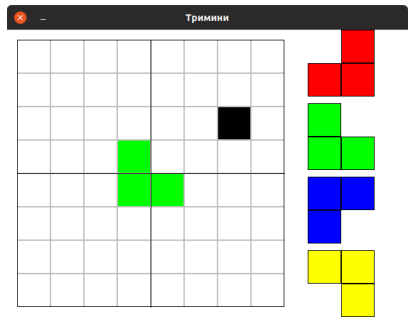
Ilutrujmo ovu tehniku najpre na primeru rešavanja jedne interesantne zagonetke. Neka je data tabla dimenzije 8×8 na kojoj nedostaje jedno polje. Zadatak je popuniti preostala 63 polja *triminima* (oblicima koji se dobiju kada se iz kvadrata dimenzije 2×2 izbaci jedno polje). Na slici 5.1 je prikazano jedno rešenje ove zagonetke (4 moguća položaja trimino oblika su prikazana desno).



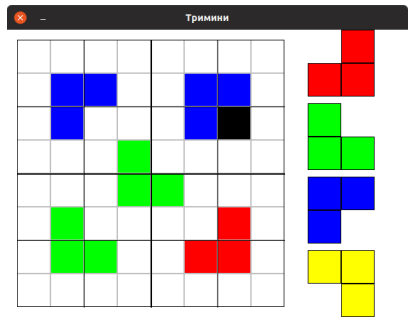
Slika 5.1: Trimini

Razmislimo da li je moguće da, u cilju rešavanja, podelimo ovaj problem na potprobleme istog oblika, ali manje dimenzije. Kvadrat nije moguće podeliti na dva manja kvadrata, ali jeste moguće na četiri manja kvadrata (ako je polazni kvadrat

dimenzije 8×8, tada su 4 manja kvadrata dimenzije 4×4. Postavljanjem jednog trimina u sredinu table možemo postići da u svakoj od te 4 table nedostaje po jedno polje - polje koje nedostaje tom triminu treba da bude u manjem kvadratu koji sadrži nedostajuće polje polaznog zadatka. Time se dobijaju četiri problema manje dimenzije, koji su istog oblika kao polazni - svakom manjem kvadratu nedostaje po jedno polje.



Nakon toga, svaki od 4 potproblema rešavamo rekurzivno. Svaka tabla dimenzije 4×4 se može razložiti na 4 table dimenzije 2×2. Postavljanjem jednog trimina možemo postići da u svakoj od te 4 table nedostaje po jedan trimino.



Na kraju, svaku tablu dimenzije 2×2 kojoj nedostaje jedno polje možemo jednostavno popuniti postavljanjem odgovarajućeg trimina. Time se dobija rešenje prikazano na slici 5.1.

Važni primeri uspešne primene tehnike “podeli-pa-vladaj” su i efikasni algoritmi sortiranja koje ćemo opisati u nastavku. Naravno, sortiranje je uvek poželjno vršiti

primenom bibiliotečke funkcije, koja je zasnovana na ovim ili sličnim algoritmima, pa se ovi algoritmi sortiranja danas retko ručno implementiraju. Prikazaćemo i nekoliko problema koji se rešavaju praktično istim tehnikama kojim se rešava i problem sortiranja, što ukazuje na korist poznavanja ovih opštih tehnika konstrukcije algoritama.

5.1 Sortiranje objedinjavanjem (MergeSort)

Algoritam *merge sort* deli niz na dva dela čije se dužine razlikuju najviše za 1 (ukoliko je dužina niza paran broj, onda su ova dva dela jednakih dužina), rekurzivno sortira svaki od njih i zatim objedinjuje sortirane polovine. Za objedinjavanje je neophodno koristiti dodatni, pomoćni niz. Na kraju se objedinjeni niz kopira u polazni niz. Iz rekurzije se izlati ako je niz jednočlanog (slučaj praznog niza ne može da nastupi).

Ključna operacija u ovom algoritmu je operacija objedinjavanja opisana u poglavlju 3.7.2. Funkciju ćemo prilagoditi tako da objedinjava samo delove dva vektora smeštajući rezultat u deo trećeg vektora.

```
// objedinjava n elemenata sortiranog niza a od pozicije p i
// m elemenata sortiranog niza b od pozicije q,
// smestajuci rezultat u sortirani vektor c od pozicije r

void merge(const vector<int>& a, int n, int p,
           const vector<int>& b, int m, int q,
           vector<int>& c, int r) {
  vector<int> c(m + n);
  int i = p, j = q, k = r;
  while (i < p + n && j < q + 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++];
}
```

Funkcija `mergesort` algoritmom *merge sort* sortira deo niza `a[l, d]`, uz korišće-

nje niza tmp kao pomoćnog.

```
void mergesort(vector<int>& a, int l, int d,
               vector<int>& tmp) {
  if (l < d) {
    int i, j;
    int n = d - l + 1, s = l + n/2;
    int n1 = n/2, n2 = n - n/2;

    mergesort(a, l, s-1, tmp);
    mergesort(a, s, d, tmp);
    merge(a, n1, l, a, n2, s, tmp, 0);

    // copy(tmp.begin(), tmp.begin() + (d-l+1), a.begin() + l);
    for (i = l, j = 0; i <= d; i++, j++)
      a[i] = tmp[j];
  }
}
```

Promenljiva n čuva broj elemenata koji se sortiraju u okviru ovog rekurzivnog poziva, a promenljiva s čuva središnji indeks u nizu između l i d. Rekurzivno se sortira n1 = n/2 elemenata između pozicija l i s-1 i n2 = n - n/2 elemenata između pozicija s i d. Nakon toga, sortirani podnizovi objedinjuju se u pomoćni niz tmp.

Pomoćni niz može se pre početka sortiranja alocirati i koristiti kroz rekurzivne pozive:

```
void mergesort(vector<int>& a) {
  vector<int> tmp(a.size());
  mergesort(a, 0, n-1, tmp);
}
```

Napravimo jednu paralelu između rekurzivne varijante algoritama *selection sort* i algoritma *merge sort*. Osnovna ideja algoritma *selection sort* je da se jedan element postavi na svoje mesto i da se zatim ista metoda rekurzivno primeni na niz koji je za jedan kraći od polaznog. S obzirom na to da je pripremna akcija zahtevala

$O(n)$ operacija, ovaj rekurzivni pristup za vremensku složenost daje jednačinu $T(n) = T(n-1) + O(n)$, te $T(n)$ pripada klasi $O(n^2)$. S druge strane, kod algoritma *merge sort* sortiranje se svodi na sortiranje dva podniza polaznog niza dvostruko manje dimenzije. S obzirom na to da korak objedinjavanja dva sortirana niza zahteva $O(n)$ operacija, dobija se jednačina $T(n) = 2T(n/2) + O(n)$, pa (na osnovu teoreme **??**) $T(n)$ pripada klasi $O(n \log n)$. Dakle, značajno je efikasnije da se problem dimenzije n svodi na dva problema dimenzije $n/2$ nego na jedan problem dimenzije $n-1$. To zapažanje je u osnovni pristupa "podeli i vladaj" (engl.~divide-and-conquer) algoritama, u koje spada i algoritam *merge sort*. Pošto rekurzija nije repna (postoje dva rekurzivna poziva i korak objedinjavanja nakon njih), eliminacija rekurzije nije jednostavna te implementacija algoritma *merge sort* najčešće ostaje rekurzivna. Razmotrimo sada i prostornu složenost ovog algoritma. Primetimo da se u funkciji drugi rekurzivni poziv vrši tek kada je prvi završen. Zato prostornu složenost opisuje jednačina $S(n) = S(n/2) + O(1)$, te $S(n)$ pripada klasi $O(\log n)$ (ovde se misli na *dodatnu složenost*, bez pomoćnog niza). Ukupno, aalgoritam koristi pomoćni niz veličine $O(n)$ i kreira najviše $O(\log n)$ stek okvira, te njegova dodatna prostorna složenost pripada klasi $O(\log n)$, a (ukupna) prostorna složenost pripada klasi $O(n)$. Kako broj stek okvira na programskom steku logaritamski zavisi od broja elemenata niza, ne postoji opasnost da dođe do prekoračenja steka.

U nastavku ćemo prikazati kako se osnovna ideja algoritma *merge sort* može primeniti na rešavanje dva algoritamska problema.

Zadatak: Broj inverzija

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

Napomena: Broj inverzija nam daje neku meru koliko je niz blizu neopadajuće sortiranog stanja (broj inverzija neopadajuće sortiranog niza je 0, dok najviše inverzija ima niz koji je sortiran nerastuće). Broj inverzija u nizu ima primenu u analizi efikasnosti algoritama sortiranja, određivanju složenosti zadataka u veštačkoj inteligenciji, proceni sličnosti rangiranja u statistici, proveri rešivosti logičkih zagonetki poput slagalice 15 i slično.

Opis ulaza

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

Opis izlaza

Na standardni izlaz ispisati samo traženi broj inverzija.

Primer

Ulaz	*Izlaz*
5 3 1 4 2 5	3

Rešenje

Gruba sila

Grubom silom se zadatak rešava tako što se pomoću ugnežđenih petlji ispitaju svi parovi pozicija $0 \leq i < j < n$ i prebroje svi slučajevi kada je $a_i > a_j$ (brojimo elemente filtrirane serije). Složenost ovog algoritma odgovara broju parova, a to je $O(n^2)$.

```
long long broj_inverzija(const vector<int>& a) {
  int n = a.size();
  long long broj = 0;
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      if (a[j] < a[i])
        broj++;
  return broj;
}
```

Podeli pa vladaj - modifikacija algoritma MergeSort

Razmotrimo kako bismo problem rešili dekompozicijom. Prazan niz i jednočlan niz nemaju inverzija. Ako je niz podeljen na dve polovine, ukupan broj inverzija jednak je zbiru broja inverzija među elementima prve polovine, broja inverzija među elementima druge polovine i broja parova elemenata gde prvi element pripada prvoj, drugi element pripada drugoj polovini i prvi je veći od drugog. Prva dva broja možemo odrediti rekurzivno i ostaje samo pitanje kako efikasno odrediti treći broj. Da bismo dobili ukupnu složenost $O(n \log n)$ taj problem je potrebno rešiti u složenosti $O(n)$ tako da ispitivanje svih parova elemenata iz prve i druge

polovine ne dolazi u obzir. Zadatak bi se mogao lakše rešiti ako bi prva i druga polovina bile sortirane (ključni uvid je da sortiranje elemenata tih polovina ne menja treći broj). Tada možemo primeniti tehniku dva pokazivača i veoma slično kao u slučaju objedinjavanja dva sortirana niza odrediti željeni treći broj. Međutim, umesto da sortiramo polovine zasebno, algoritam sortiranja možemo integrisati sa brojanjem inverzija i proširiti invarijantu naše funkcije (ojačati induktivnu hipotezu) tako da funkcija vraća broj inverzija i ujedno i sortira niz. Na osnovu invarijante, rekurzivni pozivi će sortirati levu i desnu polovinu, a da bismo je održali, tokom određivanja trećeg broja vršićemo objedinjavanje sortiranih nizova (isto kao u algoritmu MergeSort).

Primer 5.1.1. *Pokažimo na jednom primeru kako možemo da prebrojimo inverzije tokom objedinjavanja. Neka je dat niz* $1, 3, 5, 4, 7, 6, 2, 8$. *Sve inverzije u njemu su* $(3, 2)$, $(5, 4)$, $(5, 2)$, $(4, 2)$, $(7, 6)$, $(7, 2)$ *i* $(6, 2)$ *i ima ih 7.*

Podelom se dobijaju polovine $1, 3, 5, 4$ *i* $6, 7, 2, 8$. *Rekurzivni pozivi sortiraju polovine niza i u prvoj polovini pronalaze inverziju* $(5, 4)$, *a u drugoj polovini inverzije* $(7, 6)$, $(7, 2)$ *i* $(6, 2)$. *Nedostaju još inverzije gde je prvi element iz prve, a drugi iz druge polovine niza (to su inverzije* $(3, 2)$, $(4, 2)$ *i* $(5, 2)$*). Sortirane polovine su* $1, 3, 4, 5$ *i* $2, 6, 7, 8$. *Započnimo objedinjavanje ovih nizova.*

- *Na početku se porede elementi* 1 *i* 2. *Pošto je element iz leve polovine manji, on se prebacuje u rezultujući niz. Pošto je druga polovina sortirana, znamo da je element* 1 *manji od svih elemenata druge polovine, pa on učestvuje u 0 inverzija.*

- *Nakon toga se porede elementi* 3 *i* 2. *Ovaj put je element desne polovine* 2 *manji od elementa iz leve polovine* 3*, pa zato njega prebacujemo u rezultat. Element* 2 *je manji od elementa* 3*, a pošto je leva polovina sortirana, manji je i od svih elemenata iza njega. Zato znamo da on učestvuje u 3 inverzije.*

- *Nakon toga se porede elementi* 3 *i* 6. *Element* 3 *je manji od elementa* 6*, pa se on prebacuje u rezultat. Pošto se element* 6 *nalazi na poziciji* 1 *u desnoj polovini i pošto je desna polovina sortirana, možemo zaključiti da element* 3 *učestvuje u jednoj inverziji (to je ona sa elementom* 2*).*

- *Nakon toga se porede elementi* 4 *i* 6. *Element* 4 *je manji od elementa* 6*, pa se on prebacuje u rezultat. Pošto se element* 6 *nalazi na poziciji* 1 *u desnoj polovini i pošto je desna polovina sortirana, možemo zaključiti da element* 4 *učestvuje u jednoj inverziji (to je ona sa elementom* 2*).*

- *Nakon toga se porede elementi* 5 *i* 6*. Element* 5 *je manji od elementa* 6*, pa se on prebacuje u rezultat. Pošto se element* 6 *nalazi na poziciji* 1 *u desnoj polovini i pošto je desna polovina sortirana, možemo zaključiti da element* 5 *učestvuje u jednoj inverziji (to je ona sa elementom* 2*).*

- *Pošto su svi elementi iz prve polovine prepisani u rezultujući niz, prepisuju se elementi iz druge polovine. Prepisuje se element* 6 *i pošto je on veći od svih elemenata iz prve polovine, on ne učestvuje ni u jednoj inverziji. Isto važi i za elemente* 7 *i* 8*.*

Dakle, prilikom objedinjavanja za svaki element možemo izračunati broj inverzija u kojima učestvuje (naravno, govorimo samo o inverzijama između dve polovine niza).

- Kada se u rezultujući niz prepisuje element iz leve polovine niza on je strogo veći od svih elemenata desne polovine koji su već prepisani (njihov broj se lako određuje na osnovu vrednosti desnog pokazivača), a manji ili jednak od ostalih elemenata desne polovine, pa je broj inverzija u kojima on učestvuje jednak vrednosti desnog pokazivača (od koje treba oduzeti poziciju početka desne polovine niza, ako njeni elementi nisu smešteni od pozicije 0). Ako desna polovina počinje na poziciji $s+1$, tada se broj inverzija može odrediti kao $p_d - s - 1$.

- Kada se u rezultujući niz prepisuje element iz desne polovine niza, on je strogo manji od tekućeg elementa u levoj polovini i svih elemenata iza njega. Dakle, broj inverzija u kojima učestvuje može se izračunati tako što se od ukupnog broja elemenata leve polovine oduzme pozicija levog pokazivača (naravno, opet je potrebno u obzir uzeti i poziciju na kojoj počinje leva polovina niza). Ako se leva polovina završava na poziciji s onda se broj inverzija može izračunati kao $s + 1 - p_l$.

Ukupan broj inverzija se može izračunati bilo tako što se sabere broj pojedinačnih inverzija za svaki element iz leve polovine (tada brojač uvećavamo u trenutku kada se prebacuje element iz leve polovine) ili tako što se sabere broj pojedinačnih inverzija za svaki element iz desne polovine (tada brojač uvećavamo u trenutku kada se prebacuje element iz desne polovine). Možda je malo jednostavnije sabrati inverzije svih elemenata u desnoj polovini, jer tada u fazi prepisivanja elemenata

preostale polovine (kada se jedna polovina iscrpi) nema potrebe za ažuriranjem broja inverzija. Zaista, ako su preostali samo elementi leve polovine, za svaki element desne polovine je već izračunat i sabran broj inverzija, a ako su preostali samo elementi desne polovine, pošto su iscrpljeni svi elementi leve polovine, ti preostali elementi desne polovine ne učestvuju ni u jednoj inverziji. Ipak, pošto rekurzivna funkcija pored izračunavanja broja inverzija treba da sortira niz, preostale elemente moramo prepisati u rezultat (čak iako broj inverzija znamo i ranije).

```
long long broj_inverzija(vector<int>& a, int l, int d,
                         vector<int>& b) {
  if (l >= d)
    return 0;
  int s = l + (d - l) / 2;
  long long broj = 0;
  broj += broj_inverzija(a, l, s, b);
  broj += broj_inverzija(a, s+1, d, b);
  int pl = l, pd = s+1, pb = 0;
  while (pl <= s && pd <= d) {
    if (a[pl] <= a[pd])
      b[pb++] = a[pl++];
    else {
      broj += s - pl + 1;
      b[pb++] = a[pd++];
    }
  }
  while (pl <= s)
    b[pb++] = a[pl++];
  while (pd <= d)
    b[pb++] = a[pd++];

  copy(begin(b), next(begin(b), d-l+1), next(begin(a), l));

  return broj;
}

long long broj_inverzija(const vector<int>& a) {
```

```
  vector<int> tmp1(a.size()), tmp2(a.size());
  // kopiramo vektor a u vektor tmp1 da mogao da se menja
  copy(begin(a), end(a), begin(tmp1));
  return broj_inverzija(tmp1, 0, a.size()-1, tmp2);
}
```

Zadatak: Najbliži par tačaka

Za dati skup tačaka u ravni odrediti koliko je rastojanje između dve tačke koje su međusobno najbliže.

Opis ulaza

Sa standardnog ulaza se unosi broj tačaka n ($1 \leq n \leq 50000$), a zatim u narednih n redova koordinate tačaka (dva cela broja između -10^9 i 10^9, razdvojena razmakom).

Opis izlaza

Na standardni izlaz ispisati traženo rastojanje, zaokruženo na pet decimala.

Primer

Ulaz	*Izlaz*
5 0 0 0 2 2 0 2 2 1 1	1.41421

Rešenje

Gruba sila

Rešenje grubom silom podrazumeva ispitivanje svih parova tačaka i složenost mu je $O(n^2)$.

```
struct Tacka {
  int x, y;
};
```

```
double rastojanje(const Tacka& t1, const Tacka& t2) {
  double dx = t1.x - t2.x;
  double dy = t1.y - t2.y;
  return sqrt(dx*dx + dy*dy);
}

double najblizeTacke(vector<Tacka>& tacke) {
  int n = tacke.size();
  double d = numeric_limits<double>::max();
  for (int i = 0; i < n; i++)
    for (int j = i+1; j < n; j++)
      d = min(d, rastojanje(tacke[i], tacke[j]));
  return d;
}
```

Tehnika "podeli-pa-vladaj"

Jedan način da se do rešenja dođe efikasnije je da se primeni dekompozicija.

Bazni slučaj predstavlja situacija u kojoj imamo manje od četiri tačke, jer njih ne možemo podeliti u dve polovine u kojima postoji bar po jedan par tačaka. U tom slučaju rešenje nalazimo poređenjem rastojanja svih parova tačaka (pošto je tačaka malo, ovaj korak je složenosti $O(1)$). Ukoliko u skupu nema tačaka ili postoji samo jedna tačka, smatraćemo da je rezultat ∞.

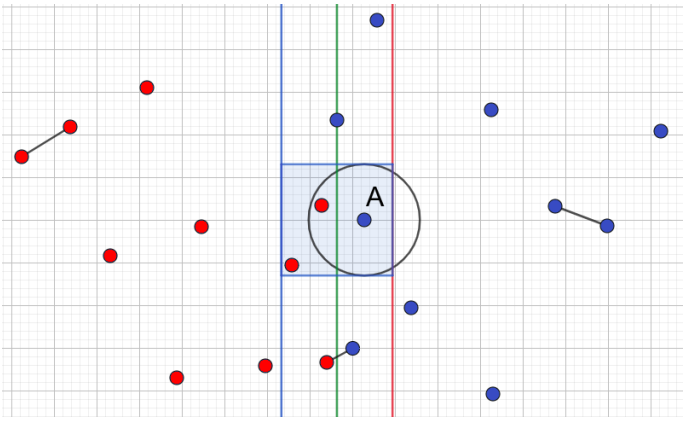
Skup tačaka možemo jednom vertikalnom pravom podeliti na dve otprilike istobrojne polovine. Ako u fazi pretprocesiranja tačke sortiramo po koordinati x, vertikalna prava može odgovarati koordinati središnje tačke. Rekurzivno određujemo najmanje rastojanje u prvoj polovini (te tačke se nalaze levo od vertikalne prave ili eventualno na njoj) i najmanje rastojanje u drugoj polovini (te tačke se nalaze desno od vertikalne prave ili eventualno na njoj). Najbliži par je takav da su (1) obe tačke u levoj polovini, (2) obe tačke u desnoj polovini ili (3) jedna tačka je u levoj, a druga u desnoj polovini. Za prva dva slučaja već znamo rešenja (na osnovu rezultata rekurzivnih poziva) i ostaje da se razmotri samo treći.

Neka je d_l minimalno rastojanje tačaka u levoj polovini, d_r minimalno rastojanje tačaka u desnoj polovini, a d minimum ta dva rastojanja. Ako vertikalna prava ima

x-koordinatu x, tada je moguće primeniti tehniku odsecanja i odbaciti sve tačke koje su levo od $x-d$ i desno od $x+d$, jer je njihovo rastojanje do najbliže tačke iz suprotne polovine sigurno veće od d. Potrebno je ispitati sve preostale tačke, tj. sve tačke iz pojasa $[x-d, x+d]$, proveriti da li među njima postoji neki par tačaka čije je rastojanje strogo manje od d i vrednost d ažurirati na vrednost najmanjeg rastojanja takvog para tačaka. Ako su tačke nasumično raspoređene, većina tačaka će biti van tog pojasa. Međutim, problem je to što u najgorem slučaju u pojasu može biti puno tačaka (moguće je čak i da se svih n tačaka nađe u tom pojasu) i ako ispitujemo sve parove, tada dolazimo do oko $n^2/4$ poređenja (ako je pola tačaka levo, a pola desno od prave podele). Ipak, proveru je moguće organizovati tako da se proveri samo mali broj parova tačaka.

Jednostavnosti radi pretpostavićemo da na isti način razmatramo sve tačke unutar pojasa $[x-d, x+d]$, bez obzira sa koje strane vertikalne prave se nalaze (unapred znamo da je provera tačaka koje su sa iste strane vertikalne prave podele nepotrebna, ali ne može narušiti korektnost, dok god smo sigurni da se porede i svi potrebni parovi tačaka sa različite strane te prave). Svaku tačku A iz pojasa je dovoljno uporediti samo sa onim tačkama koje leže unutar kruga sa centrom u tački A i poluprečnikom d, jer su sve tačke van tog kruga sigurno od tačke A udaljene više od d, što omogućava značajna odsecanja. Međutim, pripadnost krugu nije jednostavno proveriti i zato umesto njega možemo razmatrati kvadrat stranice dužine $2d$, čiji se centar nalazi na pravoj $y = x$, a na čijoj se horizontalnoj srednjoj liniji nalazi tačka A i tačku A ćemo porediti samo sa tačkama unutar tog kvadrata (znamo da su tačke van tog kvadrata sigurno na rastojanju većem od d). Time će odsecanja biti nešto manje nego u slučaju kruga (jer su neke tačke unutar kvadrata na rastojanju većem od d), ali će detektovanje tačaka koje pripadaju tom kvadratu biti veoma jednostavno – to će biti sve one tačke iz pojasa $[x-d, x+d]$, kojima je koordinata y u intervalu $[y_A - d, y_A + d]$.

Dodatno smanjenje broja poređenja možemo dobiti ako primetimo da svaki par obrađujemo dva puta (jednom dok obrađujemo tačke u okolini prve, a jednom dok obrađujemo tačke u okolini druge tačke). Možemo jednostavno zaključiti da je dovoljno svaku tačku A porediti samo sa onim tačkama koje se su iznad nje (ili su eventualno na istoj visini kao ona), tj. ne u celom kvadratu, nego samo u njegovoj gornjoj polovini. Dakle, svaku tačku A je potrebno uporediti samo sa tačkama čije x koordinate leže unutar intervala $[x-d, x+d]$ i čije y koordinate leže unutar intervala $[y_A, y_A + d]$. Prvi uslov možemo obezbediti tako što pre poređenja sve tačke iz pojasa širine d oko vertikalne prave podele izdvojimo u poseban niz (za



Najbliži par tačaka u levom pojasu, desnom pojasu i između pojaseva. Krug i kvadrat koji sadrže tačke koje se porede sa tačkom A.

to nam je potrebno $O(n)$ dodatne memorije i vremena). Drugi uslov efikasnije možemo obezbediti ako sve tačke tog pomoćnog niza sortiramo po koordinati y (za to nam je potrebno vreme $O(n \log n)$ i zatim tačke obrađujemo u neopadajućem redosledu y koordinata. Za svaku tačku A obrađujemo samo tačke koje se nalaze nakon nje u sortiranom nizu i obrađujemo jednu po jednu tačku sve dok ne naiđemo na tačku čija je koordinata y veća ili jednaka od vrednosti $y_A + d$ (ona od tačke A ne može biti na manjem rastojanju od d, a isto važi i za sve tačke u nizu iza nje).

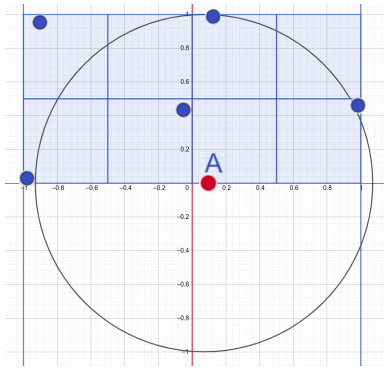
```
// funkcija pronalazi najblizi par tacaka u delu nizu [l, r]
double najblizeTacke(vector<Tacka>& tacke, int l, int r,
                     vector<Tacka>& pojas) {
  // za manje od 4 tacke najblizi par odredjujemo grubom silom
  if (r - l + 1 < 4) {
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++)
        d = min(d, rastojanje(tacke[i], tacke[j]));
    return d;
  }

  // delimo niz tacaka sa pozicija [l, r] u dve polovine
  int s = l + (r - l) / 2;
```

```
  // rekurzivno pronalazimo najblizi par u
  // levoj i desnoj polovini niza
  double d1 = najblizeTacke(tacke, l, s, pojas);
  double d2 = najblizeTacke(tacke, s+1, r, pojas);

  // najmanje rastojanje svih parova tacaka
  double d = min(d1, d2);

  // pronalazimo tacke u pojasu sirine 2d oko sredisnje linije
  double dl = tacke[s].x - d, dr = tacke[s].x + d;
  int k = 0;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas[k++] = tacke[i];

  // sortiramo tacke u pojasu po y koordinati
  sort(begin(pojas), next(begin(pojas), k),
       [](const Tacka& t1, const Tacka& t2) {
         return t1.y < t2.y;
       });

  // analiziramo sve tacke u pojasu
  for (int i = 0; i < k; i++)
    // svaku tacku poredimo samo sa onim tackama koje su u
    // pravougaoniku iznad nje
    for (int j = i+1; j < k && pojas[j].y-pojas[i].y < d; j++)
      d = min(d, rastojanje(pojas[i], pojas[j]));

  // vracamo najkrace rastojanje
  return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
  // sortiramo tacke po x koordinati
  sort(begin(tacke), end(tacke),
```

```
        [](const Tacka& t1, const Tacka& t2) {
          return t1.x < t2.x;
        });
  // pomocni niz (alociramo ga samo jednom, van rekurzije)
  vector<Tacka> pojas(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pojas);
}
```

Odredimo složenost prethodnog algoritma. Algoritam se sastoji od dva rekurzivna poziva za dvostruko manju dimenziju niza tačaka i faze dobijanja krajnjeg rezultata na osnovu rezultata rekurzivnih poziva i dodatne analize tačaka u pojasu $[x - d, x + d]$. Već smo konstatovali da izdvajanje tačaka centralnog pojasa zahteva $O(n)$ memorije i vremena i da sortiranje tih tačaka po koordinati y zahteva dodatnih $O(n \log n)$ koraka. Ostaje još da se proceni složenost ugnežđenih petlji u kojima se porede tačke unutar pojasa. Iako deluje da je složenost kvadratna, elementarnim geometrijskim rezonovanjem dokazaćemo da je složenost tog koraka linearna tj. $O(n)$ i da se u svakom koraku spoljašnje petlje unutrašnja petlja može izvršiti samo veoma mali broj puta (dokazaćemo da je taj broj izvršavanja ograničen odozgo sa 7, mada je u praksi on često i dosta manji od toga i za nasumično generisane tačke ta petlja se najčešće izvršava 0, 1 ili eventualno 2 puta).

Za svaku tačku A možemo konstruisati 8 kvadrata dimenzije $d/2$, kao što je prikazano na slici (kvadrati su upisani u pojas $[x - d, x + d]$, u dva reda od po četiri kvadrata i tačka A leži na donjoj ivici donjih kvadrata).



Najbliži par tačaka

Najveće rastojanje između dve tačke unutar nekog kvadrata se postiže kada oni leže u njegovim naspramnim temenima, a pošto je dužina dijagonale kvadrata stranice

$\frac{d}{2}$ jednaka $\frac{d\sqrt{2}}{2} \approx 0,70711 \cdot d$, rastojanje između svake dve tačke unutar istog kvadrata je strogo manje od d. Pošto svi kvadrati leže bilo potpuno sa leve strane vertikalne prave podele, bilo potpuno sa njene desne strane, unutar svakog od kvadrata može se naći najviše jedna tačka našeg skupa (u suprotnom bi se bilo sa leve, bilo sa desne strane centralne prave podele nalazio par tačaka sa rastojanjem strogo manjim od d, što je kontradiktorno sa definicijom veličine d). To znači da se iznad tačke A može nalaziti najviše 7 tačaka koje pripadaju ostalim kvadratima (sama tačka A već pripada jednom od kvadrata) i da se sve ostale tačke koje su iznad A nalaze i iznad naših kvadrata, što znači da im je rastojanje od A sigurno veće od d (jer im je vertikalno rastojanje veće od d) i njih nije potrebno razmatrati.

Tačke koje su sa iste strane prave podele kao i tačka A možemo prosto preskočiti u telu unutrašnje petlje i tako uštediti na računanju njihovog rastojanja od tačke A, ali to neće promeniti red složenosti algoritma. Druga mogućnost za implementaciju je da ne čuvamo sve tačke iz pojasa u istom skupu, već da ih podelimo u dva pojasa i da zatim obradimo najpre sve tačke iz levog pojasa gledajući rastojanja u odnosu na naredne najviše 4 tačke iz desnog pojasa, a zatim da obradimo sve tačke iz desnog pojasa gledajući rastojanja u odnosu na najviše 4 tačke iz levog pojasa (jer u suprotnom pojasu postoji 4 kvadrata dimezije $d/2$, za koje smo dokazali da ne mogu da sadrže dve tačke istovremeno). Implementacija na taj način je malo komplikovanija, a red složenosti algoritma ostaje isti (i eksperimenti ne ukazuju na značajne dobitke).

Dakle, nakon rekurzivnih poziva, za dobijanje konačnog rezultata je potrebno izvršiti dodatnih $O(n \log n)$ koraka i dekompozicija zadovoljava rekurentnu jednačinu $T(n) = 2T(n/2) + O(n \log n)$. Rešenje ove jednačine, na osnovu master teoreme, je $O(n(\log n)^2)$.

Modifikacija sortiranja objedinjavanjem

Složenost opisanog algoritma može se popraviti ako se sortiranje po koordinati y vrši istovremeno sa pronalaženjem najbližeg para tačaka, tj. ako se ojača induktivna hipoteza i ako se pretpostavi da će rekurzivni poziv vratiti rastojanje između najbliže dve tačke i ujedno sortirati date tačke po koordinati y. U koraku objedinjavanja dva sortirana niza objedinjujemo u jedan. To možemo uraditi uobičajenim algoritmom objedinjavanja, zasnovanom na tehnici dva pokazivača, koji objedinjavanje vrši u linearnoj složenosti. U jeziku C++ taj algoritam je dostupan i pomoću bibliotečke funkcije `merge`. Na taj način dobijamo algoritam koji zadovoljava jednačinu $T(n) = 2T(n/2) + O(n)$ i složenosti je $O(n \log n)$. Naglasimo da ova

optimizacija nije revolucionarna, ali može malo poboljšati efikasnost.

Na nivou implementacije, malo poboljšanje bismo mogli dobiti i tako što bismo izbegli alokacije pomoćnog vektora unutar rekurzivnih poziva i kod izmeniti tako da se u svakom rekurzivnom pozivu koristi isti, unapred alociran pomoćni vektor. Još jedna moguća optimizacija o kojoj bi se moglo razmisliti je smanjivanje broja operacija korenovanja.

```
bool porediX(const Tacka& t1, const Tacka& t2) {
  return t1.x < t2.x;
}

bool porediY(const Tacka& t1, const Tacka& t2) {
  return t1.y < t2.y;
}

// funkcija pronalazi najblizi par tacaka u delu nizu [l, r] i
// dodatno sortira tacke unutar tog dela niza po y koordinati
double najblizeTacke(vector<Tacka>& tacke, int l, int r,
                     vector<Tacka>& pojas) {
  // za manje od 4 tacke, grubom silom odredjujemo najblizi par
  if (r - l + 1 < 4) {
    // odredjujemo najblizi par
    double d = numeric_limits<double>::max();
    for (int i = l; i < r; i++)
      for (int j = i+1; j <= r; j++)
        d = min(d, rastojanje(tacke[i], tacke[j]));
    // sortiramo tacke po y-koordinati
    sort(next(begin(tacke), l), next(begin(tacke), r+1),
         porediY);
    return d;
  }

  // delimo niz tacaka sa pozicija [l, r] u dve polovine
  int s = l + (r - l) / 2;
  int x = tacke[s].x;
  // rekurzivno pronalazimo najblizi par u levoj i desnoj
```

```
  // polovini niza sortirajuci te polovine po y koordinati
  double d1 = najblizeTacke(tacke, l, s, pojas);
  double d2 = najblizeTacke(tacke, s+1, r, pojas);

  // najmanje rastojanje svih parova tacaka
  double d = min(d1, d2);

  // objedinjavamo dva sortirane polovine
  // (koristimo niz pojas kao pomocni)
  merge(next(begin(tacke), l), next(begin(tacke), s+1),
        next(begin(tacke), s+1), next(begin(tacke), r+1),
        begin(pojas), porediY);
  copy(begin(pojas), next(begin(pojas), r-l+1),
       next(begin(tacke), l));

  // pronalazimo tacke u pojasu sirine 2d oko sredisnje linije
  int k = 0;
  double dl = x - d, dr = x + d;
  for (int i = l; i <= r; i++)
    if (dl <= tacke[i].x && tacke[i].x <= dr)
      pojas[k++] = tacke[i];

  // analiziramo sve tacke u pojasu
  for (int i = 0; i < k; i++)
    // svaku tacku poredimo samo sa onim tackama koje su u
    // pravougaoniku iznad nje
    for (int j = i+1; j < k && pojas[j].y-pojas[i].y < d; j++)
      d = min(d, rastojanje(pojas[i], pojas[j]));

  return d;
}

double najblizeTacke(vector<Tacka>& tacke) {
  sort(begin(tacke), end(tacke), porediX);
  vector<Tacka> pojas(tacke.size());
  return najblizeTacke(tacke, 0, tacke.size() - 1, pojas);
}
```

5.2 Brzo sortiranje (QuickSort)

Kao algoritam *merge sort*, i algoritam *quick sort* pripada grupi *podeli-pa-vladaj* algoritama. Slično kao kod algoritma *selection sort*, u svakom koraku težimo da neki element niza dovedemo na svoju poziciju u sortiranom nizu. Umesto da to obavezno bude minimum (ili maksimuma), u algoritam *quick sort* u svakom koraku na svoje mesto dovodi se neki element (obično nazivan *pivot*) koji je poželjno blizu sredine niza. Međutim, da bi nakon toga, problem mogao biti sveden na sortiranje dva manja podniza, potrebno je prilikom dovođenja pivota na svoje mesto grupisati sve elemente manje od njega ili jednake njemu levo od njega, a sve elemente veće od njega desno od njega (ako se niz sortira neopadajuće). To pregrupisavanje elemenata niza, *korak particionisanja* (već opisano u poglavlju **??**) ključni je korak algoritma *quick sort*. Primetimo da je u algoritmu *merge sort* razdvajanje na potprobleme trivijalno, a objedinjavanje rezultata netrivijalno, a u algoritmu *quick sort* je suprotno: razdvajanje na potprobleme je netrivijalno, a objedinjavanje rezultata je trivijalno.

Algoritam *quick sort* može se implementirati na sledeći način.

```
void quicksort(vector<int>& a, int l, int d)
{
  if (l < d) {
    // pocetni polozaj pivota
    int p = izbor_pivota(a, l, d);
    // pivot dovodimo na pocetak niza
    swap(a[l], a[p]);
    // particionisemo niz i odredjujemo novi polozaj pivota
    p = particionisi(a, l, d);
    // rekurzivno sortiramo delove levo i desno od pivota
    quicksort(a, l, p - 1);
    quicksort(a, p + 1, d);
  }
}
```

Poziv `quicksort_(a, l, d)` sortira deo niza `a[l, d]`. Funkcija `quicksort` onda se jednostavno implementira:

```
void quicksort(vector<int>& a)
{
  quicksort(a, 0, a.size()-1);
}
```

Funkcija `izbor_pivota` bira za pivot neki element niza `a[l, d]` i vraća njegov indeks (u nizu `a`). Pozivom funkcije `razmeni` pivot se postavlja na poziciju `l`. Funkcija `particionisanje` vrši particionisanje niza (pretpostavljajući da se pre particionisanja pivot nalazi na poziciji `l`) i vraća poziciju na kojoj se nalazi pivot nakon particionisanja. Funkcija se poziva samo za nizove koji imaju više od jednog elementa, te joj je preduslov da je `l` manje od `d`. Postuslov funkcije `particionisi` je da je (multi)skup elemenata niza `a` nepromenjen nakon njenog poziva, međutim njihov redosled je takav da su svi elementi niza `a[l, p-1]` manji od ili jednaki elementu `a[p]`, dok su svi elementi niza `a[p+1, d]` veći od ili jednaki elementu `a[p]`.

Na osnovu teoreme **??**, kako bi se dobila jednačina $T(n) = 2T(n/2) + O(n)$ i vremenska složenost $O(n \log n)$, funkcija `particionisi` (tj. korak particionisanja) treba da bude izvršena u linearnom vremenu $O(n)$. U nastavku će biti prikazano nekoliko algoritama particionisanja koji ovo zadovoljavaju. Dalje, potrebno je da pozicija pivota nakon particionisanja bude blizu sredini niza (kako bi dužina dva podniza na koje se problem svodi bilo približno jednaka $n/2$). Međutim, određivanje središnjeg člana u nizu brojeva (što predstavlja idealnu strategiju za funkciju `izbor_pivota`) je problem koji nije značajno jednostavniji od samog sortiranja. S obzirom na to da se očekuje da je implementacija funkcije `izbor_pivota` brza (obično $O(1)$), obično se ne garantuje da će za pivot biti izabiran upravo srednji član, već se koriste heuristike koje za pivot biraju elemente koji nisu daleko od središnje pozicije u nizu. Naglasimo da se za svaku strategiju izbora pivota (koja ne koristi slučajno izabrane brojeve) može konstruisati niz takav da u svakom koraku izbor pivota bude najgori mogući — onaj koji deli niz na nizove dužine `0` i `n-1`, što dovodi do jednačine $T(n) = T(n-1) + O(n)$ i kvadratne vremenske složenosti i linearne dodatne prostorne složenosti (u odnosu na dužinu niza). Međutim, većina strategija je takva da se u *prosečnom* slučaju može očekivati da se dužine podnizova ne razlikuju mnogo, te daju prosečnu vremensku složenost $O(n \log n)$ i prosečnu dodatnu prostornu složenost $O(\log n)$ (jer algoritam radi u mestu i na programskom steku se u proseku formira $O(\log n)$ stek okvira).

U praksi, najbolje rezultate kod sortiranja dugačkih nizova daje upravo algoritam

quick sort. Međutim, za sortiranje kraćih nizova naivni algoritmi (na primer, *insertion sort*) mogu da se pokažu pogodnijim. Većina realnih implementacija *quick sort* algoritma koristi hibridni pristup — izlaz iz rekurzije se vrši kod nizova koji imaju nekoliko desetina elemenata i na njih se primenjuje *insertion sort*.

5.2.1 *Implementacije particionisanja*

Korak particionisanja može se implementirati kao što je to opisano u poglavlju 3.7.1. Na primer,

```
int particionisi(vector<int>& a, int l, int d)
{
  int m = l;
  for (int t = l+1; t <= d; t++)
    if (a[t] >= a[l])
      swap(a[++m], a[t]);
  swap(a[m], a[l]);
  return m;
}
```

Još jedan način particionisanja zasnovan je na Dejkstrinom algoritmu „trobojke“ (Dutch National Flag Problem). U ovom slučaju, radi se malo više od onoga što postuslov striktno zahteva — niz se permutuje tako da prvo idu svi elementi striktno manji od pivota, zatim sva pojavljivanja pivota i na kraju svi elementi striktno veći od pivota. Ovaj pristup particionisanju je pogodan za nizove u kojima ima puno ponovljenih elemenata, jer se u jednom koraku veliki broj elemenata dovodi na svoje mesto i rekurzivi pozivi obrađuju kraće podnizove.

```
int particionisi(int a[], int l, int d)
{
  int pn = l-1, pp = d+1, pivot = a[l], t = l;
  while (t < pp) {
    if (a[t] < pivot)
      swap(a[++t], a[++pn]);
    else if (a[t] > pivot)
      swap(a[t], a[--pp]);
    else
```

```
        t++;
    }
    return pn+1;
}
```

5.2.2 *Izbor pivota*

Kao što je već rečeno, iako je poželjno da se pivot izabere tako da podeli niz na dve potpuno jednake polovine, to bi trajalo previše tako da se obično pribegava heurističkim rešenjima. Ukoliko se može pretpostaviti da su elementi niza slučajno raspoređeni (što se uvek može postići ukoliko se pre primene sortiranja niz permutuje na slučajan način), bilo koji element niza se može uzeti za pivot. Na primer,

```
int izbor_pivota(int a[], int l, int d)
{
  return l;
}
```

ili

```
int izbor_pivota(int a[], int l, int d)
{
  return slucajan_broj(l, d);
}
```

Nešto bolje performanse mogu se postići ukoliko se, na primer, za pivot uzima srednji od tri slučajno izabrana elementa niza.

U nastavku ćemo prikazati algoritam *quick select*, koji ilustruje kako se osnovna ideja algoritma brzog sortiranja može primeniti na rešavanje nekoliko srodnih problema primenom *delimičnog sortiranja niza* (engl. partial sort). Delimičnim sortiranjem u vremenu $O(n)$ možemo na prvih k pozicija u nizu dovesti k najmanjih (ili najvećih) elemenata niza (u proizvoljnom redosledu), odrediti neku statistiku tih k elemenata (zbir, proizvod, minimum, maksimum i slično), odrediti k-ti po redu element u nizu (tzv. rangovske statistike), odrediti medijanu niza i slično.

Zadatak: Zbir k najboljih

Ovaj zadatak je ponovljen u cilju uvežbavanja različitih tehnika rešavanja. Vidi tekst zadatka.

Pokušaj da zadatak uradiš korišćenjem tehnika koje se izlažu u ovom poglavlju.

Rešenje

Sortiranje celog niza

Zadatak se može rešiti i tako što će se niz sortirati bibliotečkom funkcijom za sortiranje. Ako se niz sortira nerastuće, onda je nakon sortiranja potrebno sabrati prvih k elemenata niza, a ako se sortira neopadajuće, onda je nakon sortiranja potrebno sabrati poslednjih k elemenata niza (sortiranje neopadajuće je obično podrazumevani način sortiranja i lakše ga je realizovati).

Ako se sortiranje vrši bibliotečkim funkcijama, vremenska složenost ovog rešenja je $O(n \cdot \log(n))$, dok je memorijska složenost jednaka $O(n)$.

Može se primetiti da ovakav algoritam nepotrebno gubi vreme precizno određujući redosled tj. sortirajući elemente koji su manji od prvih k i određujući precizan redosled prvih k elemenata. Naime, da bi se sabralo prvih k elemenata oni ne moraju biti poređani, već je samo potrebno na početak niza (ili na kraj) dovesti prvih k elemenata, a iza njih (ili ispred) postaviti ostale elemente tj. razdvojiti te dve grupe elemenata, pri čemu je redosled elemenata unutar svake od grupa irelevantan.

```
// ucitavamo ulazne podatke
int n, k;
cin >> n >> k;

vector<int> a(n);
for (int i = 0; i < n; i++)
  cin >> a[i];

// sortiramo niz nerastuce
sort(begin(a), end(a), greater<int>());

// sabiramo prvih k elemenata niza
int s = accumulate(begin(a), next(begin(a), k), 0);
```

```
// ispisujemo rezultat
cout << s << endl;
```

QuickSelect

Algoritam *brze selekcije* (QuickSelect) predstavlja modifikaciju algoritma brzog sortiranja. Brza selekcija se koristi da se niz podeli tako da se na prvih k mesta nađe k najvećih (ili najmanjih) elemenata niza, da se na mestima iza njih nalaze elementi manji (ili veći) od njih, pri čemu je redosled elemenata u svakoj od te dve grupe proizvoljan. Pošto je redosled elemenata u grupama proizvoljan, može se dobiti efikasniji algoritam od onoga u kom bi se sortirao ceo niz (jer redosled elemenata u svakoj grupi može biti proizvoljan).

Algoritam brze selekcije se zasniva na koraku particionisanja, identičnom kao u algoritmu brzog sortiranja, koji u linearnoj složenosti elemente niza uređuje tako da se prvo u nizu nađu elementi koji su manji od nekog datog elementa (pivota), da se nakon njih nalazi pivot i da nakon toga slede elementi koji su veći od pivota. Redosled elemenata u svakoj od ovih grupa je potpuno proizvoljan. Ako se pivot javlja više puta, ostala pojavljivanja pivota mogu biti bilo levo, bilo desno od pivota (često se uzima da se levo od pivota nalaze elementi manji ili jednaki od njega, a desno elementi strogo veći od njega ili obratno). Kao kod algoritma brzog sortiranja, za particionisanje se mogu koristiti postupci zasnovani na tehnici dva pokazivača.

Pošto se u zadatku traži zbir k najvećih elemenata u nizu, jednostavnosti radi, pretpostavićemo da elemente niza sortiramo u obratnom redosledu – želimo da se k najvećih elemenata niza nađe na njegovom početku.

Osovni algoritam je rekurzivan i parametar rekurzije su granice l i d i broj k, i zadatak algoritma je da deo niza na pozicijama $[l, d]$ reorganizuje tako da na početku bude k najvećih elemenata tog dela niza.

Izlaz iz rekurzije može biti kada je k veće ili jednako od dužine segmenta $[l, d]$, tj. kada je $k \geq d - l + 1$ i tada svi elementi niza spadaju među prvih k. Ako je u startu $k \leq n$, gde je n broj elemenata niza, tada k nikada neće biti strogo veće od dužine segmenta $[l, d]$ (ali može biti jednako).

Nakon izbora pivota i particionisanja, poznata je pozicija na kojoj se pivot nalazi. Neka je to neka pozicija m (važi da je $l \leq m \leq d$).

- Ako je broj elemenata levo od pivota (vrednost $m-l$) veća ili jednaka k onda je dovoljno naći k najvećih elemenata levog dela niza. Naime svi elementi u levom delu niza su veći ili jedanki od pivota i od elemenata u desnom delu, pa se svih k najvećih elemenata dela niza sa pozicija $[l, d]$ nalaze u levom delu niza, ispred pivota. Dakle, potrebno je izvršiti rekurzivni poziv za interval $[l, m-1]$ i broj k tj. pronaći k najvećih elemenata u delu niza na pozicijama $[l, m-1]$.
- U suprotnom se zaključno sa pivotom nalazi $m-l+1$ od k najvećih elemenata niza i zato je potrebno u desnom delu odrediti još $k-(m-l+1)$ najvećih elemenata iz tog dela, tako da se rekurzivni poziv vrši za interval $[m+1, d]$ i broj $k-m+l-1$.

Primetimo da u funkciji postoji samo jedan rekurzivni poziv i to repni, tako da se on može jednostavno eliminisati.

```
// QuickSelect - odredjujemo najvecih k elemenata niza a tj.
// niz permutujemo tako da se najvecih k elemenata nadju na
// prvih k pozicija (u proizvoljnom redosledu)
void qsortK(vector<int>& a, int l, int d, int k) {
  if (k >= d - l + 1)
    return;

  // niz particionisemo tako da se pivot (element a[l]) dovede
  // na svoje mesto, da ispred njega budu svi elementi koji su
  // veci ili jednaki od njega, a da iza njega budu svi
  // elementi veci od njega
  int m = particionisanje(a, l, d);

  if (k <= m - l)
    // svih k elemenata su levo od pivota -
    // obradjujemo deo ispred pivota
    qsortK(a, l, m - 1, k);
  else
    // mozda su neki kod k najvecih iza pivota -
    // obradjujemo deo iza pivota
    qsortK(a, m+1, d, k - (m - l + 1));
```

```
}

// QuickSelect - pomocna funkcija zbog lepseg interfejsa
void qsortK(vector<int>& a, int k) {
  qsortK(a, 0, a.size() - 1, k);
}
```

Primer 5.2.1. *Prikažimo rad ovog algoritma na primeru pronalaženja 5 najvećih elemenata niza* $5, 9, 6, 3, 10, 13, 1, 7, 8, 14, 2$. *Važi da je* $k = 5$ *i* $n = 11$.

- *Na početku je* $[l, d] = [0, n-1] = [0, 10]$. *Posle particionisanja dobija se niz* $9, 6, 10, 13, 7, 8, 14, 5, 3, 1, 2$, *gde je pivot* 5 *završio na poziciji* $m = 7$. *Pošto je broj elemenata levo od pivota* $m - l = 7$ *veći od* $k = 5$, *rekurzivno pronalazimo 5 najvećih elemenata u delu niza na pozicijama u intervalu* $[l, m-1] = [0, 6]$.

- *Particionišemo deo niza određen sa* $[l, d] = [0, 6]$. *Posle particionisanja dobija se niz* $10, 13, 14, 9, 6, 7, 8, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, *gde je pivot* 9 *završio na poziciji* $m = 3$. *Ovaj put je broj elemenata levo od pivota* $m - l = 3$ *strogo manji od* $k = 5$. *Zato je potrebno rekurzivno ponaći* $k - (m - l + 1) = 1$ *najveći element u delu niza na pozicijama* $[m + 1, d] = [4, 6]$

- *Particionišemo deo niza određen sa* $[l, d] = [4, 6]$. *Posle particionisanja dobija se niz* $\mathbf{10}, \mathbf{13}, \mathbf{14}, \mathbf{9}, 7, 8, 6, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, *gde je pivot* 6 *završio na poziciji* $m = 6$. *Broj elemenata levo od pivota* $m - l = 2$ *i on je veći od* k, *pa rekurzivno tražimo* $k = 1$ *najveći element u delu niza na pozicijama* $[l, m-1] = [4, 5]$.

- *Particionišemo deo niza određen sa* $[l, d] = [4, 5]$. *Posle particionisanja dobija se niz* $\mathbf{10}, \mathbf{13}, \mathbf{14}, \mathbf{9}, 8, 7, \mathbf{6}, \mathbf{5}, \mathbf{3}, \mathbf{1}, \mathbf{2}$, *gde je pivot* 7 *završio na poziciji* $m = 5$. *Broj elemenata levo od pivota* $m - l = 1$ *i on je jednak* k, *pa rekurzivno tražimo* $k = 1$ *najveći element u delu niza na pozicijama* $[l, m-1] = [4, 4]$.

- *Pošto je dužina* $d - l + 1$ *intervala* $[l, d] = [4, 4]$ *jednaka* $k = 1$, *postupak je završen. Konačno stanje niza je* $10, 13, 14, 9, 8, 7, 6, 5, 3, 1, 2$. *Primetimo da niz nije sortiran opadajuće, ali smo sigurni da se 5 najvećih elemenata niza sada nalazi na njegovom početku.*

Primetimo i da dokazivanje zaustavljanja algoritma nije sasvim trivijalno. Ako je k u startu veće od dužine niza, algoritam će se odmah zaustaviti. U suprotnom će sve vreme izvršavanja algoritma važiti invarijanta da je $0 \leq k \leq d - l + 1$ (u trenutku kada se postigne gornja jednakost, algoritam će se zaustaviti), dok će se vrednost $d - l + 1$ smanjivati kroz rekurzivne pozive.

- Ako je $k \leq m - l$, izvršiće se rekurzivni poziv za $l' = l$, $d' = m - 1$ i $k' = k$. Pošto je $m \leq d$, važi i da je $m < d + 1$. Zato je $d' - l' + 1 = m - 1 - l + 1 = m - l < d - l + 1$.

  Važi i da je $k' \leq d' - l' + 1$, jer je $k' = k \leq m - l = d' - l' + 1$. Važi i da je $0 \leq k' = k$.

- Ako je $k > m - l$, izvršiće se rekurzivni poziv za $l' = m + 1$, $d' = d$ i $k' = k - (m - l + 1)$. Tada je $d' - l' + 1 = d - (m + 1) + 1 = d - m$. Pošto je $l \leq m$, važi da je $d' - l' + 1 = d - m \leq d - l < d - l + 1$.

  Važi i da je $k' \leq d' - l' + 1$. Zamenom dobijamo da je taj uslov ekvivalentan $k - (m - l + 1) \leq d - (m + 1) + 1$, tj. $k + l - 1 \leq d$, no to važi, jer je $k \leq d - l + 1$.

  Pošto je $k > m - l$, važi i da je $k \geq m - l + 1$, pa je $k' = k - (m - l + 1) \geq 0$.

Iz ove analize jasno je i da ako je $k \leq n$, tada uslov zaustavljanja algoritma (izlaz iz rekurzije) može biti i kada niz postane prazan tj. kada je $l - d + 1$ postane 0. Takođe, uslov zaustavljanja bi mogao biti i to da je $k = 0$.

Pod pretpostavkom da će pivot deliti niz na delove koji su otprilike jednake veličine, složenost ovog algoritma se opisuje jednačinom $T(n) = T(n/2) + O(n), T(0) = O(1)$, čije je rešenje $T(n) = O(n)$. Može se dokazati da je prosečna složenost algoritma brze selekcije linearna, dok je složenost najgoreg slučaja kvadratna (što se veoma retko dešava, kao i u algoritmu *Quick sort*). Pošto se ceo niz učitava i čuva u memoriji i prostorna složenost je $O(n)$.

Bibliotečka funkcija

U jeziku C++ bibliotečka funkcija `nth_element` vrši podelu niza tako da se na poziciji n nađe element koji tu i pripada u sortiranom redosledu, da se ispred te pozicije nađu elementi koji su svi manji ili jednaki od njega, a da se iza te pozicije

nađu elementi koji su svi veći ili jednaki od njega. Funkciji se prosleđuje iterator na početak dela niza (vektora) koji se obrađuje (obično dobijen pomoću `begin`), iterator na neku poziciju na sredini niza i iterator koji ukazuje neposredno iza kraja niza (obično dobijen pomoću `end`). Ako središnji iterator ukazuje na n-tu poziciju u nizu nakon primene funkcije na toj poziciji će se naći n-ti po veličini element, dok će svi elementi levo od njega biti manji ili jednaki od svih elemenata desno od njega. Recimo i da postoji funkcija `partial_sort` koja je slična prethodnoj ali ujedno elemente ispred date pozicije uređuje (soritra) po veličini, međutim, u ovom slučaju to nam nije potrebno i time bi se samo nepotrebno gubilo vreme.

```
// niz particionisemo tako da je k-ti element na svom mestu i
// da su svi elementi ispred njega manji ili jednaki od svih
// elemenata iza
nth_element(a.begin(), next(a.begin(), k), a.end(),
            greater<int>());

// odredjujemo i ispisujemo zbir prvih k elemenata
cout << accumulate(a.begin(), next(a.begin(), k), 0) << endl;
```

5.3 Karacubin algoritam za množenje polinoma

Prikažimo sada i još jedno rešenje problema određivanja maksimalnog zbira segmenta niza, koje je zasnovano na tehnici “podeli-pa-vladaj”.

Zadatak: Maksimalni zbir segmenta

Rešenje

Razlaganje na potprobleme

Dekompozicija nam sugeriše da je poželjno da niz podelimo na dva podniza jednake dužine čija rešenja možemo da konstruišemo na osnovu induktivne hipoteze (najčešće rekurzivnim pozivima). Bazu i ovaj put čini slučaj praznog niza, koji sadrži samo prazan segment čiji je zbir nula. Fiksirajmo središnji element niza. Sve segmente niza možemo da grupišemo u tri grupe: segmente koji su u potpunosti

levo od središnjeg elementa, segmente koji su u potpunosti desno od središnjeg elementa i segmente koji sadrže središnji element. Najveće zbirove segmenata u prvoj i u drugoj grupi znamo na osnovu induktivne hipoteze. Najveći zbir segmenta u trećoj grupi možemo lako odrediti analizom svih segmenata koji sadrže središnji element: krećemo od jednočlanog segmenta koji sadrži samo središnji element i inkrementalno se širimo nalevo dodajući jedan po jedan element i računajući tekući maksimum, a zatim krećemo od maksimalnog segmenta proširenog nalevo i inkrementalno ga proširujemo jednim po jednim elementom nadesno, tražeći novi maksimum.

```
int maksZbirSegmenta(const vector<int>& a, int l, int d) {
  if (l > d)
    return 0;
  int s = l + (d - l) / 2;
  int maks_zbir_levo = maksZbirSegmenta(a, l, s-1);
  int maks_zbir_desno = maksZbirSegmenta(a, s+1, d);
  int zbir_sredina = a[s];
  int maks_zbir_sredina = zbir_sredina;
  for (int i = s-1; i >= l; i--) {
    zbir_sredina += a[i];
    if (zbir_sredina > maks_zbir_sredina)
      maks_zbir_sredina = zbir_sredina;
  }
  zbir_sredina = maks_zbir_sredina;
  for (int i = s+1; i <= d; i++) {
    zbir_sredina += a[i];
    if (zbir_sredina > maks_zbir_sredina)
      maks_zbir_sredina = zbir_sredina;
  }
  return max({maks_zbir_levo, maks_zbir_desno,
              maks_zbir_sredina});
}

int maksZbirSegmenta(const vector<int>& a) {
  return maksZbirSegmenta(a, 0, a.size() - 1);
}
```

Ako sa n označimo dužinu niza $d-l+1$ i ako vreme izvršavanja obeležimo sa $T(n)$, tada važi da je $T(0) = O(1)$ i da je $T(n) = 2T(n/2) + O(n)$. Naime, vrše se dva rekurzivna poziva za duplo manje nizove, a najveći zbir segmenata koji obuhvataju središnji element izračunavamo u vremenu $O(n)$ (što je prilično očigledno, jer imamo dve petlje koje se ukupno izvršavaju n puta, a čija su tela konstantne složenosti). Na osnovu master teoreme lako se zaključuje da je $T(n) = O(n \log n)$. Dakle, ovaj algoritam je manje efikasan od algoritama složenosti $O(n)$, ali je i dalje prilično upotrebljiv (i sigurno je mnogo bolji od algoritama grube sile koji su kvadratne ili kubne složenosti).

Ojačanje induktivne hipoteze

Na ideji dekompozicije možemo izgraditi i efikasniji algoritam. Ključni uvid je da se najveći zbir segmenta oko srednjeg elementa može dobiti kao zbir najvećeg sufiksa niza levo od tog elementa i najvećeg prefiksa niza desno od tog elementa. Možemo ojačati induktivnu hipotezu i umesto da prefiks i sufiks računamo u petlji, u linearnom vremenu, možemo pretpostaviti da za obe polovine niza prefiks i sufiks dobijamo kao rezultat rekurzivnog poziva. To nam je dovoljno da odredimo maksimalni zbir funkcije, ali moramo „vratiti dug'' i naša funkcija sada pored maksimalnog zbira segmenta mora izračunati i maksimalni zbir prefiksa i maksimalni zbir sufiksa celog niza. Maksimalni zbir prefiksa celog niza je veći broj od maksimalnog zbira prefiksa levog dela i od zbira celog levog dela i maksimalnog zbira prefiksa desnog dela. Slično, maksimalni zbir sufiksa celog niza je veći od maksimalnog zbira sufiksa desnog dela i od zbira maksimalnog zbira sufiksa levog dela i celog desnog dela. Zato je neophodno dodatno ojačati induktivnu hipotezu i tokom rekurzije računati i zbir celog niza.

Pretpostavimo da je P zbir maksimalnog prefiksa niza, da je S zbir maksimalnog sufiksa niza, da je Z zbir celog niza i da je M maksimalni zbir segmenta niza. Označimo indeksom l te statistike u levoj polovini niza i indeksom d te statistke u desnoj polovini niza. Tada važi:

$$\begin{aligned} Z &= Z_l + Z_d \\ P &= \max(P_l, Z_l + P_d) \\ S &= \max(S_d, S_l + Z_d) \\ M &= \max(M_l, M_d, S_l + P_d) \end{aligned}$$

Jednačina koja opisuje ovu rekurziju je $T(n) = 2T(n/2) + O(1)$, pa je složenost ovog rešenja linearna tj. $O(n)$.

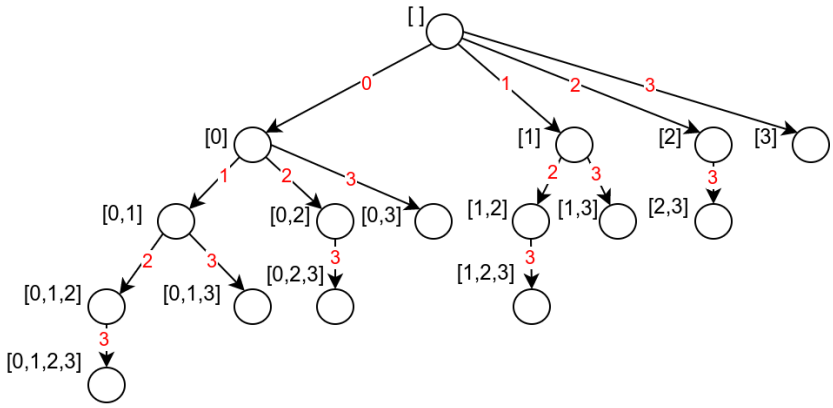
```
void maksZbirSegmenta(const vector<int>& a, int l, int d,
                      int& zbir, int& maks_zbir,
                      int& maks_prefiks, int& maks_sufiks) {
  if (l == d) {
    zbir = maks_zbir = maks_prefiks = maks_sufiks = a[l];
    return;
  }
  int s = l + (d - l) / 2;
  int zbir_levo, maks_zbir_levo,
      maks_sufiks_levo, maks_prefiks_levo;
  maksZbirSegmenta(a, l, s,
                   zbir_levo, maks_zbir_levo,
                   maks_prefiks_levo, maks_sufiks_levo);
  int zbir_desno, maks_zbir_desno,
      maks_sufiks_desno, maks_prefiks_desno;
  maksZbirSegmenta(a, s+1, d,
                   zbir_desno, maks_zbir_desno,
                   maks_prefiks_desno, maks_sufiks_desno);
  zbir = zbir_levo + zbir_desno;
  maks_prefiks = max(maks_prefiks_levo,
                     zbir_levo + maks_prefiks_desno);
  maks_sufiks = max(maks_sufiks_desno,
                    maks_sufiks_levo + zbir_desno);
  maks_zbir = max({maks_zbir_levo, maks_zbir_desno,
                   maks_sufiks_levo + maks_prefiks_desno});
}

int maksZbirSegmenta(const vector<int>& a) {
  int zbir, maks_zbir, maks_prefiks, maks_sufiks;
  maksZbirSegmenta(a, 0, a.size() - 1,
                   zbir, maks_zbir, maks_prefiks, maks_sufiks);
  return maks_zbir;
}
```

6. Generisanje kombinatornih objekata

Problemi se često mogu rešiti iscrpnom pretragom (grubom silom), što podrazumeva da se ispitaju svi mogući kandidati za rešenja. Preduslov za to je da umemo sve te kandidate da nabrojimo. Iako u realnim primenama prostor potencijalnih rešenja može imati različitu strukturu, pokazuje se da je u velikom broju slučaja to prostor određenih klasičnih *kombinatornih objekata*: svih podskupova nekog konačnog skupa, svih varijacija (sa ili bez ponavljanja), svih kombinacija (sa ili bez ponavljanja), svih permutacija, svih particija i slično. U ovom poglavlju ćemo proučiti mehanizme njihovog sistematičnog generisanja. Naglasimo da po pravilu ovakvih objekata ima eksponencijalno mnogo u odnosu na veličinu ulaza, tako da su svi algoritmi praktično neupotrebljivi osim za veoma male dimenzije ulaza.

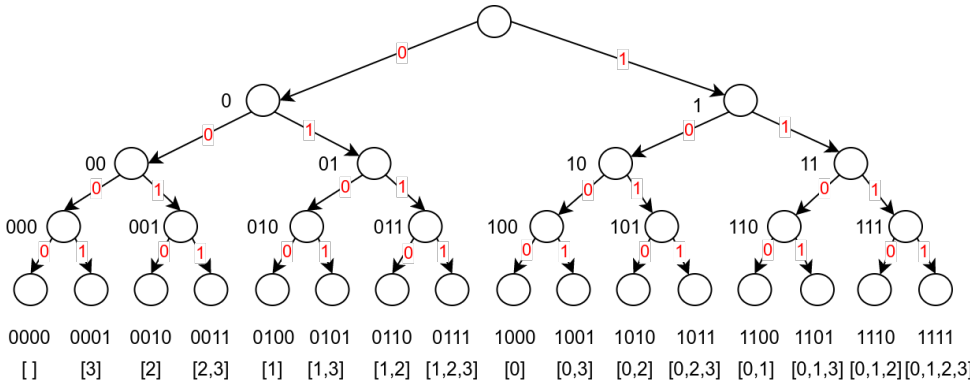
Objekti se obično predstavljaju n-torkama brojeva, pri čemu se isti objekti mogu torkama modelovati na različite načine. Na primer, svaki podskup skupa $\{a_0, \ldots, a_{n-1}\}$ se može predstaviti konačnim nizom indeksa elemenata koji mu pripadaju. Da bi svaki podskup bio jedinstveno predstavljen, potrebno je da taj niz bude kanonizovan (na primer, uređen strogo rastući). Na primer, torka $(0, 2, 3)$ jednoznačno određuje podskup $\{a_0, a_2, a_3\}$. Drugi način da se podskupovi predstave su n-torke logičkih vrednosti ili vrednosti 0-1. Na primer, ako je $n = 6$, i ako pretpostavimo da se bitovi čitaju sleva nadesno, tada torka 101100 označava skup $\{a_0, a_2, a_3\}$.

Svi objekti se obično mogu predstaviti drvetom i to drvo odgovara procesu njihovog generisanja tj. obilaska (ono se ne pravi eksplicitno, u memoriji, ali nam pomaže da razumemo i organizujemo postupak pretrage). Obilazak drveta se najjednostavnije izvodi u dubinu (često rekurzivno). Za prvu navedenu reprezentaciju podskupova drvo je dato na slici **??**. Svaki čvor drveta odgovara jednom podskupu, pri čemu se odgovarajuća torka očitava na granama puta koji vodi od korena do tog čvora.



Svi podskupovi četvoročlanog skupa - svaki čvor drveta odgovara jednom podskupu

Za drugu navedenu reprezentaciju podskupova drvo je dato na slici **??**. Na početku se bira da li će element a_0 biti uključen u podskup, na narednom nivou da li će biti uključen element a_1, zatim element a_2 i tako dalje. Samo listovi drveta u kojima je za svaki element doneta odluka da li pripada ili ne pripada podskupu, odgovaraju podskupovima, pri čemu se odgovarajuća torka logičkih vrednosti očitava na granama puta koji vodi od korena do tog čvora.



Svi podskupovi četvoročlanog skupa - svaki list drveta odgovara jednom podskupu

Primetimo da oba drveta sadrže 2^n čvorova kojima se predstavljaju podskupovi (u prvom slučaju su to svi čvorovi drveta, a u drugom samo listovi).

Prilikom generisanja objekata često je poželjno ređati ih određenim redom. S ob-

zirom na to to da se svi kombinatorni objekti predstavljaju određenim torkama (konačnim nizovima), prirodan poredak među njima je *leksikografski poredak* (koji se koristi za utvrđivanje redosleda reči u rečniku). Podsetimo se, torka $a_0 \ldots a_{m-1}$ leksikografski prethodi torci $b_0 \ldots b_{n-1}$ akko postoji neki indeks i takav da za svako $0 \leq j < i$ važi $a_j = b_j$ i važi ili da je $a_i < b_i$ ili da je $i = m < n$. Na primer važi da je $11 < 112 < 221$ (ovde je $i = 2$, a zatim $i = 0$).

Na primer, ako podskupove skupa $\{1, 2, 3\}$ predstavimo na prvi način, torkama u kojima su elementi uređeni rastuće, leksikografski poredak bi bio $\emptyset$, $\{1\}$, $\{1, 2\}$, $\{1, 2, 3\}$, $\{1, 3\}$, $\{2\}$, $\{2, 3\}$, $\{3\}$. Ako bismo ih predstavljali na drugi način, torkama u kojima se nulama i jedinicama određuje da li je neki element uključen u podskup, leksikografski redosled bi bio: 000 ($\emptyset$), 001 ($\{3\}$), 010($\{2\}$), 011($\{2, 3\}$), 100($\{1\}$), 101($\{1, 3\}$), 110($\{1, 2\}$) i 111($\{1, 2, 3\}$).

U nastavku će biti prikazano kako je moguće nabrojati sve objekte koji imaju neku zadatu kombinatornu strukturu. U većini zadataka moguće je razmatrati dve vrste rešenja. Jedna grupa rešenja je zasnovana na rekurzivnom postupku nabrajanja objekata, dok je druga grupa rešenja zasnovana na pronalaženju narednog kombinatornog objekta u odnosu na neki zadati redosled (najčešće leksikorafski).

Zadatak: Sledeći podskup

Napisati program koji određuje podskup skupa brojeva $\{1, \ldots, n\}$ koji u leksikografskom redosledu sledi neposredno iza datog podskupa. Podskupovi su zadati u obliku strogo rastuće sortiranih nizova.

Opis ulaza

Prva linija sadrži broj n ($1 \leq n \leq 100$), a naredna linija sadrži podskup čiji su elementi zadati sortirano rastuće, razdvojeni po jednim razmakom.

Opis izlaza

Na standardni izlaz u jednoj liniji ispisati elemente traženog podskupa tj. - ako je učitani podskup leksikografski najveći.

Primer 1

Ulaz	*Izlaz*
5 1 2 3 4 5	1 2 3 5

Primer 2

Ulaz	*Izlaz*
5 5	-

Rešenje

Napišimo, na primer, leksikografski uređen spisak svih podskupova skupa brojeva od 1 do 4.

```
-, 1, 12, 123, 1234, 124, 13, 134, 14, 2, 23, 234, 24, 3, 34, 4
```

Možemo primetiti da postoje dva načina da se dođe do narednog podskupa. Analizirajmo ove skupove u istom redosledu, grupisane i na osnovu broja elemenata.

```
-  1  12  123  1234
          124
       13  134
       14
   2   23  234
       24
   3   34
   4
```

Jedan način je *proširivanje* kada se naredni podskup dobija dodavanjem nekog elementa u prethodni. To su koraci u prethodnoj tabeli kod kojih se prelazi iz jedne u narednu kolonu. Da bi dobijeni podskup sledio neposredno iza prethodnog u leksikografskom redosledu, dodati element podskupu mora biti najmanji mogući. Pošto je svaki podskup sortiran, element mora biti za jedan veći od poslednjeg elementa podskupa koji se proširuje (izuzetak je prazan skup, koji se proširuje elementom 1). Jedini slučaj kada proširivanje nije moguće je kada je poslednji element podskupa najveći mogući (u našem primeru to je 4).

Drugi način je *skraćivanje* kada se naredni element dobija uklanjanjem nekih elemenata iz podskupa i izmenom preostalih elemenata. To su koraci u prethodnoj tabeli kod kojih se prelazi sa kraja jedne u narednu vrstu. U ovom slučaju skraćivanje funkcioniše tako što se iz podskupa izbaci završni najveći element, a zatim se najveći od preostalih elemenata uveća za 1 (on ne može biti najveći, jer su elementi unutar svakog podskupa strogo rastući). Ako nakon izbacivanja najvećeg elementa ostane prazan skup, naredna kombinacija ne postoji.

Podskupove možemo predstaviti dinamičkim nizom koji nam omogućva da elemente dodajemo i uklaljamo sa desnog kraja. U jeziku C++ možemo upotrebiti vektor (tj. kolekciju `vector`).

```
// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraca da li on postoji
bool sledeciPodskup(vector<int>& podskup, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (podskup.empty()) {
    podskup.push_back(1);
    // podskup je uspešno pronađen
    return true;
  }

  // proširivanje
  if (podskup.back() < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup.push_back(podskup.back() + 1);
    // podskup je uspešno pronađen
    return true;
  }

  // skraćivanje
  // uklanjamo poslednji najveći element
  podskup.pop_back();
  // ako nema preostalih elemenata ne postoji naredni podskup
  if (podskup.empty())
    return false;
  // najveći od preostalih elemenata uvećavamo za 1
  podskup.back()++;
  // podskup je uspešno pronađen
  return true;
}
```

Podskupove možemo čuvati i u okviru niza koji je unapred alociran tako da može da smesti elemente najvećeg podskupa (onog koji ima tačno n elemenata). U tom slučaju je neophodno da održavamo i promenljivu u kojoj beležimo broj elemenata podskupa. Pošto se ona menja u funkciji koja određuje naredni podskup, potrebno je preneti je po referenci.

```
// na osnovu datog podskupa skupa {1, ..., n} određuje
// leksikografski naredni podskup i vraća da on postoji.
// Tekući podskup je smešten u nizu dužine k
bool sledeciPodskup(int podskup[], int& k, int n) {
  // specijalni slučaj proširivanja praznog skupa
  if (k == 0) {
    podskup[k++] = 1;
    return true;
  }

  // proširivanje
  if (podskup[k-1] < n) {
    // u podskup dodajemo element koji je za 1 veći od
    // trenutno najvećeg elementa
    podskup[k] = podskup[k-1] + 1;
    k++;
    return true;
  }

  // skraćivanje
  // izbacujemo najveći element iz podskupa
  k--;
  // ako nema preostalih elemenata, naredni podskup ne postoji
  if (k == 0)
    return false;
  // najveći od preostalih elemenata uvećavamo za 1
  podskup[k-1]++;
  return true;
}
```

Zadatak: Svi podskupovi

Napisati program koji ispisuje sve podskupove datog skupa.

Opis ulaza

Sa standardnog ulaza se učitava broj n (važi $3 \leq n \leq 10$), a zatim n prirodnih

brojeva, rastuće sortiranih, razdvojenih po jednim razmakom.

Opis izlaza

Na standardni izlaz ispisati sve podskupove učitanog skupa brojeva, svaki u posebnom redu, sa elementima razdvojenim jednim razmakom. Prvo se ređaju podskupovi u kojima prvi element nije uključen, a zatim oni u kojima jeste. U svakoj od te dve grupe, prvo se ispisuju podskupovi u kojima drugi element nije uključen, a zatim oni gde jeste i tako dalje.

Primer

Ulaz	*Izlaz*
3	3
1 2 3	2
	2 3
	1
	1 3
	1 2
	1 2 3

Rešenje

6.1 Rekurzivni postupak generisanja svih varijacija dužine n skupa $\{0, 1\}$

Generisanje svih podskupova odgovara generisanju svih varijacija dužine n od nula i jedinica (svaki element je ili uključen ili isključen). Poredak opisan u postavci zadatka ukazuje na to da podskupovi treba da budu uređeni leksikografski, u odnosu na njihovu reprezentaciju u obliku varijacija.

Opišimo induktivno-rekurzivnu konstrukciju funkcije koja generiše sve podskupove skupa S.

- Ako je skup S prazan, onda je jedini njegov podskup prazan.
- Ako skup S nije prazan, onda se može razložiti na neki element x i skup $S' = S \setminus x$ dobijen kada se taj element izbaci iz polaznog skupa. Pošto je skup S' manji od skupa S, njegovi se podskupovi mogu odrediti rekurzivno. Svi podskupovi polaznog skupa S su onda oni koji su određeni za manji skup S', kao i svi oni koji se od njih dobijaju dodavanjem izdvojenog elementa x.

Prethodnu konstrukciju nije ekonomično programski realizovati, jer se pretpostavlja da rezultat rada funkcije predstavlja skup svih podskupova skupa. Umesto takve funkcije definisaćemo proceduru koja neće istovremeno čuvati i vraćati sve podskupove već samo jedan po jedan nabrojati i obraditi (u našem slučaju samo ispisati).

Do rešenja se može doći tako što se u rekurzivnoj funkciji prosleđuje neki podsup P skupa $\{a_0, \ldots, a_{i-1}\}$ i koji ona na sve moguće načine proširuje elementima skupa $\{a_i, \ldots, a_{n-1}\}$.

- Ako je skup $\{a_i, \ldots, a_{n-1}\}$ prazan (ako je $i = n$) tada je podskup P kompletno formiran i obrađuje se (tj. ispisuje).
- U suprotnom razmatramo element a_i i dve mogućnosti: da taj element bude izostavljen iz podsupa i mogućnost da taj element bude dodat u podskup P. U oba slučaja nastavljamo rekurzivno proširivanje skupa P (prvo neproširenog, a zatim i proširenog elementom a_i) elementima skupa $\{a_{i+1}, \ldots, a_{n-1}\}$.

Inicijalno je $i = 0$, a podskup je prazan (prazan skup je, zaista, jedini podskup praznog supa $\{a_0, \ldots, a_{i-1}\}$ i on se na sve moguće načine proširuje elementima skupa $\{a_i, \ldots, a_{n-1}\} = \{a_0, \ldots, a_{n-1}\}$).

Iako mnogi savremeni jezici pružaju tip za reprezentovanje skupova, implementacija je jednostavnija i efikasnija ako se elementi skupa čuvaju u nizu. Da bismo izbegli potrebu za produžavanjem i skraćivanjem niza i korišćenjem dinamičkih nizova, lista ili vektra, niz možemo alocirati na maksimalnu moguću dužinu (broj elemenata polaznog skupa) i paralelno sa nizom možemo održavati broj elemenata podskupa koji je trenutno smešten u niz (on je skoro uvek strogo manji od dužine niza).

Dakle, definišemo rekurzivnu funkciju koja na svakom narednom nivou rekurzije obrađuje naredni element polaznog skupa (predstavljenog nizom), sve dok se ne iscrpe svi elementi. U prvom slučaju taj element ne dodaje u rezultujući podskup (takođe predstavljen nizom, koji prosleđujemo kao dodatni parametar) i prelazi na naredni nivo rekurzije, a u drugom ga dodaje na kraj trenutnog rezultujućeg podskupa i prelazi na naredni nivo rekurzije. Kada se ceo polazni niz iscrpi (kada je dubina rekurzije jednaka dužini polaznog niza), tada se trenutno akumulirani podskup obrađuje tj. ispisuje.

Rad rekurzivne funkcije odgovara slici **??**.

```
// procedura određuje i obrađuje sve moguće skupove koji se
// dobijaju tako što se na elemente prosleđenog podskupa p
// dužine j, dodaju podskupovi prosleđenog skupa smeštenog u
// nizu a, od pozicije i nadalje
void obradi_sve_podskupove(const vector<int>& a, int i,
                           vector<int>& p, int j) {
  // skup preostalih elemenata u nizu a koji se mogu ubaciti u
  // podskup je prazan
  if (i == a.size())
    // obrađujemo formirani podskup
    obradi(p, j);
  else {
    // element na poziciji i ne uključujemo u podskup
    obradi_sve_podskupove(a, i + 1, p, j);
    // element na poziciji i uključujemo u podskup
    p[j] = a[i];
    obradi_sve_podskupove(a, i + 1, p, j + 1);
  }
}

void obradi_sve_podskupove(const vector<int>& a) {
  // podskup je na početku prazan, i u njega potencijalno
  // dodajemo sve elemente skupa a od pozicije 0 nadalje
  vector<int> p(a.size());
  obradi_sve_podskupove(a, 0, p, 0);
}
```

U implementaciji možemo koristiti i bibliotečke kolekcije za reprezentovanje niza elemenata. Međutim, treba biti veoma obazriv jer je moguće da se tokom rekurzije grade novi nizovi i kopiraju elementi, što znatno utiče na efikasnost. Naredna implementacija je zbog toga veoma loša.

```
// procedura određuje i obrađuje sve moguće skupove koji se
// dobijaju tako što se na elemente prosleđenog podskupa
// dodaju podskupovi prosleđenog skupa
void obradiSvePodskupove(const vector<int>& skup,
```

```
                          const vector<int>& podskup) {
  // skup je prazan
  if (skup.size() == 0)
    // na prosleđeni podskup možemo dodati samo prazan skup
    obradi(podskup);
  else {
    // izdvajamo i uklanjamo proizvoljan element skupa
    int x = skup.back();
    vector<int> smanjenSkup = skup;
    smanjenSkup.pop_back();
    // u podskup dodajemo sve podskupove skupa bez izdvojenog
    // elementa
    vector<int> podskupBez = podskup;
    obradiSvePodskupove(smanjenSkup, podskupBez);
    // u podskup uključujemo izdvojeni element i zatim sve
    // podskupove skupa bez izdvojenog elementa
    vector<int> podskupSa = podskup;
    podskupSa.push_back(x);
    obradiSvePodskupove(smanjenSkup, podskupSa);
  }
}
```

Moguće je napraviti rešenje koje koristi bibliotečke kolekcija podataka, a ujedno ne vrši njihovo kopiranje i dovoljno je efikasno. U implementaciji se koriste dva niza (jedan za skup, drugi za podskup) koji se tokom rada algoritma menjaju. Nizovi će se menjati dodavanjem i ukljanjanjem elemenata sa kraja. Zato je pre početka funkcije neohpodno obrnuti redosled elemenata u nizu (da bi se na kraju prvo našli početni elementi niza).

U implementacijama koje menjaju nizove često je važno osigurati da se na kraju tela rekurzivne funkcije stanje nizova vrati na isto stanje kakvo je bilo na ulasku u funkciju, jer se time garantuje da rekurzivni poziv neće modifikovati nizove koji su mu predati kao parametri (što koristimo kada se oslanjamo na to da će i nakon prvog i nakon drugog rekurzivnog poziva skup i podskup biti isti kakvi su bili i pre tog rekurzivnog poziva).

```
// procedura određuje i obrađuje sve moguće skupove koji se
// dobijaju tako što se na elemente prosleđenog podskupa dodaju
// podskupovi prosleđenog skupa
void obradiSvePodskupove(vector<int>& skup,
                         vector<int>& podskup) {
  // skup je prazan
  if (skup.size() == 0)
    // na prosleđeni podskup možemo dodati samo prazan skup
    obradi(podskup);
  else {
    // izdvajamo i uklanjamo proizvoljan element skupa
    int x = skup.back();
    skup.pop_back();
    // u podskup dodajemo sve podskupove skupa bez izdvojenog
    // elementa
    obradiSvePodskupove(skup, podskup);
    // u podskup uključujemo izdvojeni element i zatim sve
    // podskupove skupa bez izdvojenog elementa
    podskup.push_back(x);
    obradiSvePodskupove(skup, podskup);
    // vraćamo skup i podskup u početno stanje
    podskup.pop_back();
    skup.push_back(x);
  }
}

void obradiSvePodskupove(vector<int>& skup) {
  // pošto elementi iz skupa u podskup prebacuju sa desnog
  // kraja, da bismo dobili traženi redosled podskupa,
  // potrebno je da obrnemo redosled elemenata skupa
  vector<int> skupObratno = skup;
  reverse(begin(skupObratno), end(skupObratno));
  // krećemo od praznog podskupa
  vector<int> podskup;
  // efikasnosti radi rezervišemo potrebnu memoriju za
  // najveći podskup
```

```
  podskup.reserve(skup.size());
  // prazan skup proširujemo svim podskupovima datog skupa
  obradiSvePodskupove(skupObratno, podskup);
}
```

Funkcija za određivanje naredne varijacije u leksikografskom redosledu

Jedno rešenje je da se u posebnom nizu logičkih vrednosti nabrajaju sve varijacije skupa tačno-netačno. Svaka takva varijacija odgovara jednom podskupu, tako što se u podskup uključuju elementi sa onih pozicija na kojima je je vrednost tačno. Varijacije nabrajamo korišćenjem funkcije za određivanje sledeće varijacije.

```
void obradiSvePodskupove(const vector<int>& a) {
  vector<bool> v(a.size(), false);
  do {
    obradi(v, a);
  } while (sledecaVarijacija(v));
}
```

Zadatak: Sledeća varijacija

Napisati program koji određuje narednu varijaciju dužine k skupa $\{1, \ldots, n\}$ u leksikografskom poretku.

Opis ulaza

Prva linija standardnog ulaza sadrži broj k ($1 \leq k \leq 100$), a druga broj n ($1 \leq n \leq 100$). U trećoj liniji se nalazi varijacija opisana brojevima razdvojenim po jednim razmakom.

Opis izlaza

Na standardni izlaz ispisati sledeću varijaciju u leksikografskom poretku, ako ona postoji, ili `-`, ako je učitana varijacija leksikografski maksimalna.

Primer

Ulaz	*Izlaz*
5 4 1 1 2 3 4	1 1 2 4 1

Rešenje

Sledeća varijacija u leksikografskom poretku se može generisati tako što se uveća poslednji broj u varijaciji koji se može uvećati, i što se nakon uvećavanja svi brojevi iza uvećanog broja postave na 1. Pozicija na kojoj se broj uvećava naziva se *prelomna tačka* (engl. turning point). Na primer, ako nabrajamo varijacije skupa $\{1, 2, 3\}$ dužine 5 naredna varijacija za varijaciju 21332 je 21333 (prelomna tačka je pozicija 4, koja je poslednja pozicija u nizu), dok je njoj naredna varijacija 22111 (prelomna tačka je pozicija 1 na kojoj se nalazio element 1). Niz 33333 nema prelomnu tačku, pa samim tim ni leksikografski sledeću varijaciju.

Jedan način implementacije je da prelomnu tačku nađemo linearnom pretragom od kraja niza, ako prelomna tačka postoji da uvećamo element i da od sledeće pozicije do kraja niz popunimo jedinicama. Međutim, te dve faze možemo objediniti. Varijaciju obilazimo od kraja postavljajući na 1 svaki element u varijaciji koji je jednak broju n. Ako se zaustavimo pre nego što smo stigli do kraja niza, znači da smo pronašli element koji se može uvećati i uvećavamo ga. U suprotnom je varijacija imala sve elemente jednake n i bila je maksimalna u leksikografskom redosledu.

```
bool sledecaVarijacija(int n, vector<int>& varijacija) {
  // od kraja varijacije tražimo prvi element koji se moze
  // povecati
  int i;
  int k = varijacija.size();
  for (i = k-1; i >= 0 && varijacija[i] == n; i--)
    varijacija[i] = 1;
  // svi elementi su jednaki n, pa ne postoji naredna
  // varijacija
  if (i < 0) return false;
  // uvecavamo element koji je moguće uvecati
  varijacija[i]++;
  return true;
}
```

Zadatak: Sve varijacije

Napiši program koji određuje sve varijacije sa ponavljanjem dužine k skupa $\{1, \ldots, n\}$.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 5$) i u narednoj liniji broj k ($1 \leq k \leq 5$).

Opis izlaza

Na standardni izlaz ispisati sve varijacije, sortirane leksikografski.

Primer

Ulaz	*Izlaz*
2	1 1 1
3	1 1 2
	1 2 1
	1 2 2
	2 1 1
	2 1 2
	2 2 1
	2 2 2

Rešenje

Rekurzivno generisanje varijacija

Varijacije se mogu nabrojati induktivno rekurzivnom konstrukcijom.

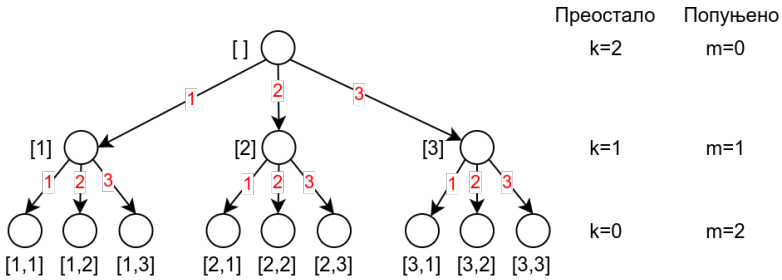
- Jedina varijacija dužine nula je prazna.
- Sve varijacije dužine k se mogu dobiti tako što se na prvo mesto upiše bilo koji od brojeva od 1 do n, a zatim se preostala mesta dopune svim varijacijama dužine $k - 1$.

Recimo i da je moguće na poslednje mesto postavljati jedan po jedan broj od 1 do n, a zatim rekurzivno popunjavati prefiks, no time bi redosled varijacija bio drugačiji od traženog.

Umesto da definišemo funkciju koja vraća kolekciju varijacija, definisaćemo rekurzivnu proceduru koja prima niz kome su popunjeni svi elementi osim poslednjih k,

i koji će na sve moguće načine dopunjavati varijacijama dužine k (koja će se smanjivati kroz rekurzivne pozive). Dakle, nakon popunjenog dela niza postavljamo jednu po jednu vrednost od 1 do n i zatim rekurzivno pozivamo proceduru da popuni ostatak niza (time što smanjujemo dužinu k i time prelazimo na narednu poziciju).

Da bismo izbegli potrebu za dinamičkim proširivanjem nizova unapred ćemo alocirati niz dužine k, a onda ćemo mu u rekurziji popunjavati jednu po jednu poziciju (tekuća pozicija se može odrediti kao razlika između dužine niza i tekuće vrednosti broja k).



Rekurzivni postupak generisanja varijacija dužine 2 od elemenata skupa $\{1, 2, 3\}$

```
// Sve varijacije duzine k elemenata skupa {1, ..., n}
// Data varijacija duzine varijacije.size() - k se
// dopunjuje svim mogucim varijacijama duzine k skupa
// {1, ..., n} i sve tako dobijene varijacije se
// obradjuju funkcijom obradi
void obradiSveVarijacije(int k, int n,
                         vector<int>& varijacija) {
  // k je 0, pa je jedina varijacija duzine nula prazna i
  // njenim dodavanjem na polazni niz on se ne menja
  if (k == 0)
    obradi(varijacija);
  else
    // na tekucu poziciju postavljamo sve moguce vrednosti
    // od 1 do n i dobijeni niz onda rekurzivno prosirujemo
    for (int nn = 1; nn <= n; nn++) {
       varijacija[varijacija.size() - k] = nn;
```

```
        obradiSveVarijacije(k-1, n, varijacija);
    }
}


// sve varijacije duzine k skupa {1, ..., n}
void obradiSveVarijacije(int k, int n) {
  vector<int> varijacija(k);
  obradiSveVarijacije(k, n, varijacija);
}
```

Pronalaženje leksikografski sledeće varijacije

Druga mogućnost je da se krene od leksikografski najmanje varijacije (to je varijacija $\underbrace{11\ldots11}_{k}$) i da se korišćenjem funkcije opisane u zadatku *Sledeća varijacija* određuje naredna varijacija date varijacije u odnosu na leksikografski redosled, sve dok takva postoji.

```
void obradiSveVarijacije(int k, int n) {
   // krećemo od varijacije 11...11
   // ona je leksikografski najmanja
   vector<int> varijacija(k, 1);
   // obradjujemo redom varijacije dok god postoji
   // leksikografski sledeca
   do {
     obradi(varijacija);
   } while(sledecaVarijacija(n, varijacija));
}
```

Zadatak: Sve reči od datih slova

Stringom s dat je skup malih slova engleskog alfabeta (slova su u stringu uređena u rastućem poretku) i prirodan broj k. Napisati program kojim se prikazuju u leksikografskom poretku sve reči dužine k koje se mogu formirati od datog skupa.

Opis ulaza

Na standardnom ulazu u prvoj liniji nalazi se string s dužine najviše 10, u drugoj liniji nalazi se prirodan broj k ($k \leq 6$, $k \leq n$).

Opis izlaza

Na standardnom izlazu prikazati tražene reči u leksikografskom poretku, svaku reč u posebnoj liniji.

Primer

Ulaz	*Izlaz*
amx	aa
2	am
	ax
	ma
	mm
	mx
	xa
	xm
	xx

Rešenje

Zadatak možemo rešiti veoma slično zadatku *Sve varijacije*. Definišemo rekurzivnu funkciju koja prima skup slova, reč koja se malo po malo popunjava i indeks naredne pozicije i u toj reči koju treba popuniti (inicijalno je ta pozicija jednaka nuli). Kada je pozicija jednaka dužini reči, reč je cela formirana i može se ispisati. U suprotnom na poziciju i postavljamo redom jedno po jedno slovo iz datog skupa i rekurzivno pozivamo funkciju da se popune slova od pozicije $i + 1$, nadalje.

```
void sveReci(string& s, const string& slova, int i) {
  if (i == s.length())
    cout << s << endl;
  else
    for (char slovo : slova) {
      s[i] = slovo;
      sveReci(s, slova, i+1);
    }
}
```

```
void sveReci(const string& slova, int k) {
  string s;
  s.resize(k);
  sveReci(s, slova, 0);
}
```

Zadatak: Sledeća kombinacija

Kombinacije dužine k od n elemenata podrazumevaju da se vrši odabir k elemenata skupa $\{1, \ldots, n\}$, slično kao što se, na primer, u igri loto bira 7 od 39 kuglica. Napisati program koji za datu kombinaciju određuje narednu u leksikografskom poretku.

Opis ulaza

Sa standardnog ulaza se unosi broj n ($2 \leq n \leq 100$) a zatim u narednom redu jedna kombinacija dužine $1 \leq k \leq n$. Elementi su zadati sortirani rastuće, odvojeni po jednim razmakom.

Opis izlaza

Na standardni izlaz ispisati kombinaciju koja je naredna u leksikografskom redosledu u odnosu na datu, tj. - ako takva kombinacija ne postoji.

Primer 1		**Primer 2**		**Primer 3**	
Ulaz	*Izlaz*	*Ulaz*	*Izlaz*	*Ulaz*	*Izlaz*
`5`	`1 3 5`	`5`	`1 4 5`	`5`	`-`
`1 3 4`		`1 3 5`		`3 4 5`	

Rešenje

Opišimo postupak kojim od date kombinacije možemo dobiti sledeću kombinaciju u leksikografskom redosledu.

Napišimo, ilustracije radi, sve kombinacije dužine 3 iz skupa od 5 elemenata.

Ponovo tražimo *prelomnu tačku* tj. element koji se može uvećati. Pošto su kombinacije dužine k i organizovane su strogo rastuće, maksimalna vrednost na poslednjoj poziciji je n, na pretposlednjoj $n-1$ itd. Dakle, poslednji element se može uvećati ako nije jednak n, pretposlednji ako nije jednak $n-1$ itd. Prelomna tačka je

pozicija prvog elementa koji je manji od svog maksimuma. Ako pozicije brojimo od 0, maksimum na poziciji $k-1$ je n, na poziciji $k-2$ je $n-1$ itd. tako da je maksimum na poziciji i jednak $n-k+1+i$. Ako prelomna tačka ne postoji (ako su sve vrednosti na svojim maksimumima), naredna kombinacija u leksikografskom redosledu ne postoji. U suprotnom uvećavamo element na prelomnoj poziciji i da bismo nakon toga dobili leksikografski što manju kombinaciju, sve elemente iza njega postavljamo na najmanje moguće vrednosti. Pošto kombinacija mora biti sortirana strogo rastuće, nakon uvećanja prelomne vrednosti sve elemente iza nje postavljamo na vrednost koja je za jedan veća od vrednosti njoj prethodne vrednosti u nizu.

Primer 6.1.1. *Na primer, ako je $n = 6$ i $k = 4$, tada je naredna kombinacija kombinaciji 1256, kombinacija 1345 - prelomna vrednost je 2 i ona se može uvećati na 3, nakon čega slažemo redom elemente za po jedan veće.*

Prikažimo sve prelomne tačke prilikom generisanja kombinacija dužine $n = 3$ iz skupa od $k = 5$ elemenata.

$$12\mathbf{3} \to 12\mathbf{4} \to 1\mathbf{2}5 \to 13\mathbf{4} \to 1\mathbf{3}5 \to \mathbf{1}45 \to 23\mathbf{4} \to 23\mathbf{5} \to \mathbf{2}45 \to 345$$

Zadatak: Sve kombinacije

Kombinacije dužine k od n elemenata podrazumevaju da se vrši odabir k elemenata skupa $\{1, \ldots, n\}$, slično kao što se, na primer, u igri loto bira 7 od 39 kuglica. Napiši program koji za date vrednosti k i n nabraja i ispisuje sve kombinacije, poređane po leksikografskom redosledu.

Opis ulaza

Prva linija standardnog ulaza sadrži broj k ($1 \leq k \leq n$), a naredna broj n ($2 \leq n \leq 20$).

Opis izlaza

Na standardni izlaz ispisati sve kombinacije. Svaka kombinacija treba da bude predstavljena nizom brojeva sortiranim strogo rastuće, a sve kombinacije treba da budu poređane u leksikografskom redosledu.

Primer

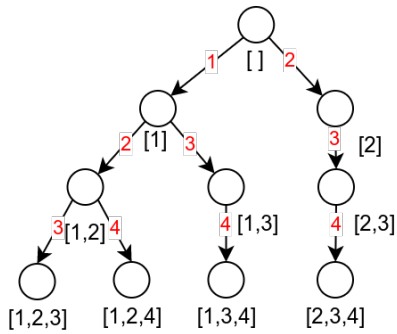
Ulaz	*Izlaz*
3	1 2 3
5	1 2 4
	1 2 5
	1 3 4
	1 3 5
	1 4 5
	2 3 4
	2 3 5
	2 4 5
	3 4 5

Rešenje

Rekurzivni pozivi po pozicijama

Zadatak rekurzivne funkcije biće da dopuni niz dužine k od pozicije i pa do kraja. Kada je $i = k$, niz je popunjen i potrebno je obraditi (u našem slučaju ispisati) dobijenu kombinaciju. U suprotnom biramo element koji ćemo postaviti na poziciju i. Pošto su kombinacije uređene strogo rastuće, on mora biti veći od prethodnog (ako prethodni ne postoji, onda može biti 1) i manji ili jednak n. Zapravo, ovo gornje ograničenje mora da se smanji. Pošto su elementi strogo rastući, a od pozicije i pa do kraja niza treba postaviti $k-i$ elemenata, na poziciji i može biti $n+i-k+1$ i tada će na poziciji $k-1$ biti vrednost n. U petlji stavljamo jedan po jedan od tih elemenata na poziciju i i rekurzivno nastavljamo generisanje od naredne pozicije. Dakle, u opštem slučaju, drvo rekurzivnih poziva neće biti binarno (funkcija može da napravi mnogo više od dva rekurzivna poziva).

```
// niz kombinacije dužine k na pozicijama [0, i) sadrži uređen
// niz elemenata iz skupa [1, n-i+1). Procedura na sve moguće
// načine dopunjava elementima iz skupa [1, n) tako da niz
// bude uređen rastući
void obradiSveKombinacije(vector<int>& kombinacija,
                          int i, int n) {
  // tražena dužina kombinacije
  int k = kombinacija.size();
```



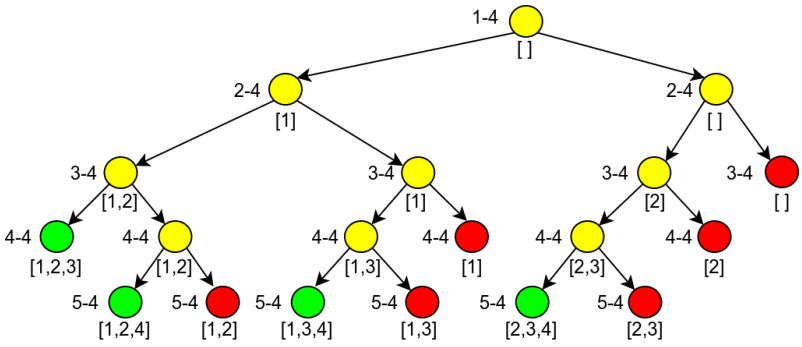
Rekurzivno generisanje kombinacija dužine 3 iz skupa $\{1, 2, 3, 4\}$

```
  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }
  // određujemo raspon elemenata na poziciji i
  int pocetak = i == 0 ? 1 : kombinacija[i-1]+1;
  int kraj = n + i - k + 1;
  // jedan po jedan element upisujemo na poziciju i, pa
  // nastavljamo generisanje rekurzivno
  for (int x = pocetak; x <= kraj; x++) {
    kombinacija[i] = x;
    obradiSveKombinacije(kombinacija, i+1, n);
  }
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, n);
}
```

Rekurzivni pozivi po vrednostima

Postoji način da izbegnemo rekurzivne pozive u petlji. Tokom rekurzije možemo da čuvamo informaciju o tome koji je raspon elemenata kojim se proširuje niz. Znamo da su to elementi skupa $\{1, \ldots, n\}$, međutim, pošto su kombinacije sortirane rastuće skup kandidata je uži. U prethodnom programu smo najmanju vrednost za poziciju i određivali na osnovu vrednosti sa pozicije $i-1$, međutim, alternativno možemo i eksplicitno da održavamo promenljive n_{min} i n_{max} koje određuju skup $\{n_{min}, \ldots, n_{max}\}$ čiji se elementi raspoređuju u kombinaciji na pozicijama iz intervala $[i, k)$. Ako je taj interval prazan, kombinacija je popunjena i može se obraditi. U suprotnom, ako je $n_{min} > n_{max}$, tada ne postoji vrednost koju je moguće staviti na poziciju i, pa možemo izaći iz rekurzije, jer se trenutna kombinacija ne može popuniti do kraja. U suprotnom možemo razmotriti dve mogućnosti. Prvo na poziciju i možemo postaviti element n_{min} i rekurzivno izvršiti popunjavanje niza od pozicije $i+1$, a drugo možemo taj element preskočiti i u rekurzivnom pozivu ponovo zahtevati da se popuni pozicija i. U oba slučaja se skup elemenata sužava na $\{n_{min}+1, \ldots, n_{max}\}$.

Pretragu možemo saseći i malo ranije. Naime, pošto su ponavljanja zabranjena kada je broj elemenata tog skupa (a to je $n - n_{min} + 1$) manji od broja preostalih pozicija koje treba popuniti (a to je $k - i$), već tada možemo saseći pretragu, jer ne postoji mogućnost da se kombinacija uspešno dopuni do kraja.



Rekurzivno generisanje kombinacija - levo od svakog čvora je prikazan raspon preostalih vrednosti, ispod čvora tekuća kombinacija. U zelenim čvorovima su uspešno generisane kombinacije, a u crvenim nastupa odsecanje, jer raspon ne sadrži dovoljno vrednosti da bi se kombinacija generisala do kraja

```
void obradiSveKombinacije(vector<int>& kombinacija, int i,
                          int n_min, int n_max) {
  // tražena dužina kombinacije
  int k = kombinacija.size();

  // ako je popunjen ceo niz samo ispisujemo kombinaciju
  if (i == k) {
    obradi(kombinacija);
    return;
  }

  // ako tekuću kombinaciju nije moguće popuniti do kraja
  // prekidamo ovaj pokušaj
  if (n_max - n_min + 1 < k - i)
    return;

  // vrednost n_min uključujemo na poziciju i, pa rekurzivno
  // proširujemo tako dobijenu kombinaciju
  kombinacija[i] = n_min;
  obradiSveKombinacije(kombinacija, i+1, n_min+1, n_max);
  // vrednost n_min preskačemo i isključujemo iz kombinacije
  obradiSveKombinacije(kombinacija, i, n_min+1, n_max);
}

// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  vector<int> kombinacija(k);
  obradiSveKombinacije(kombinacija, 0, 1, n);
}
```

Leksikografski sledeća kombinacija

Jedan način da se zadatak reši bez rekurzije je da se upotrebi funkcija za određivanje naredne kombinacije u leksikografskom poretku koja je opisana u zadatku *Sledeća kombinacija*.

```
// nabraja i obradjuje sve kombinacije dužine k skupa
// {1, 2, ..., n}
void obradiSveKombinacije(int k, int n) {
  // krecemo od kombinacije 1, 2, ..., k
  vector<int> kombinacija(k);
  for (int i = 0; i < k; i++)
    kombinacija[i] = i + 1;

  // obradjujemo kombinacije dokle god postoji sledeca
  do {
    obradi(kombinacija);
  } while (sledecaKombinacija(n, kombinacija));
}
```

Zadatak: Sledeća permutacija

Sve permutacije brojeva od 1 do n se mogu poređati leksikografski. Na primer za $n = 3$ permutacije u leksikografskom poretku su

```
123
132
213
231
312
321
```

Napisati program kojim se za dati prirodan broj n i datu permutaciju brojeva od 1 do n prikazuje sledeća permutacija u leksikografskom poretku (prva permutacija koja se nalazi posle date permutacije).

Opis ulaza

Prva linija standardnog ulaza sadrži prirodan broj n ($n < 1000$). U svakoj od n narednih linija standardnog ulaza, nalaze se redom elementi permutacije, svaki u posebnoj liniji.

Opis izlaza

Na standardnom izlazu prikazati redom elemente sledeće permutacije u leksiko-

grafskom poretku, svaki element u posebnoj liniji. Ako ne postoji sledeća permutacija (data permutacija je poslednja) prikazati u jednoj liniji poruku `ne postoji`.

Primer 1

Ulaz	*Izlaz*
5	3
3	1
1	5
4	2
5	4
2	

Primer 2

Ulaz	*Izlaz*
3	ne postoji
3	
2	
1	

Rešenje

Algortitam za određivanje naredne permutacije u leksikografskom redosledu

Primer 6.1.2. *Razmotrimo permutaciju* 13542. *Zamenom elementa* 2 *i* 4 *bi se dobila permutacija* 13524 *koja je leksikografski manja od polazne i to nam ne odgovara. Slično bi se desilo i da se zamene elementi* 5 *i* 4. *Činjenica da je niz* 542 *strogo opadajući nam govori da nije mogući ni na koji način razmeniti ta tri elementa da se dobije leksikografski veća permutacija, tj. da je ovo najveća permutacija koja počinje sa* 13. *Dakle, naredna permutacija će biti leksikografski najmanja permutacija koja počinje sa* 14, *a to je* 14235.

Dakle, u prvom koraku algoritma pronalazimo prelomnu tačku, tj. prvu poziciju i zdesna, takvu da je $a_i < a_{i+1}$ (za sve $i+1 \leq k < n-1$ važi da je $a_k > a_{k+1}$). Ovo radimo najjobičnijom linearnom pretragom. Ako takva pozicija ne postoji, naša permutacija je skroz opadajuća i samim tim leksikografski najveća. Nakon toga, pronalazimo prvu poziciju j zdesna takvu da je $a_i < a_j$ (opet linearnom pretragom) i razmenjujemo elemente na pozicijama i i j. Pošto je ovom razmenom rep iza pozicije i i dalje striktno opadajući, da bismo dobili željenu permutaciju (leksikografski najmanju permutaciju koja počinje sa $a_0 \ldots a_{i-1}a_j$), potrebno je obrnuti redosled elemenata repa tj. deo niza od pozicije $i+1$ do kraja niza.

Primer 6.1.3. *Ako označimo pozicije elemenata dobijamo* $1^0 3^1 5^2 4^3 2^4$. *Zato je* $i = 1$ *i* $a_i = 3$, *dok je* $j = 3$ *i* $a_j = 4$. *Nakon razmene dobijamo* $1^0 4^1 5^2 3^3 2^4$. *Da bismo dobili traženu permutaciju* $1^0 4^1 2^2 3^3 5^4$ *obrćemo deo niza od pozicije* $i+1 = 2$ *do kraja niza.*

```
bool sledecaPermutacija(vector<int>& a){
  int n = a.size();

  // linearnom pretragom pronalazimo prvu poziciju i takvu da
  // je a[i] > a[i+1]
  int i = n - 2;
  while (i >= 0 && a[i] > a[i+1])
    i--;
  // ako takve pozicije nema,
  // permutacija a je leksikografski maksimalna
  if (i < 0) return false;
  // linearnom pretragom pronalazimo prvu poziciju j takvu da
  // je a[j] > a[i]
  int j = n - 1;
  while (a[j] < a[i])
    j--;
  // razmenjujemo elemente na pozicijama i i j
  swap(a[i], a[j]);
  // obrcemo deo niza od pozicije i+1 do kraja
  for (j = n - 1, i++; i < j; i++, j--)
    swap(a[i], a[j]);
  return true;
}
```

Bibliotečka funkcija

U jeziku C++ postoji bibliotečka funkcija `next_permutation` koja određuje sledeću permutaciju u leksikografskom redosledu (i vraća informaciju o tome da li ona postoji).

```
// ucitavamo polaznu permutaciju
int n;
cin >> n;
vector<int> a(n);
for(int i = 0; i < n; i++)
```

```
  cin >> a[i];

// odredjujemo sledecu i ispisujemo rezultat
if (next_permutation(begin(a), end(a)))
  obradi(a);
else
  cout << "ne postoji" << endl;
```

Zadatak: Sve permutacije

Napiši program koji generiše i ispisuje sve permutacije skupa $\{1, 2, \ldots, n\}$.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 8$).

Opis izlaza

Na standardni izlaz ispisati tražene permutacije. Svaku permutaciju ispisati u posebnom redu, a elemente razdvojiti po jednim razmakom. Redosled permutacija može biti proizvoljan.

Primer

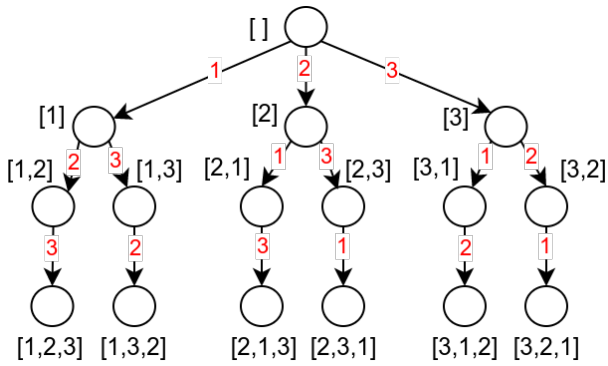
Ulaz	*Izlaz*
3	1 2 3
	1 3 2
	2 1 3
	2 3 1
	3 1 2
	3 2 1

Rešenje

Rekurzivno generisanje permutacija sa eksplicitnom proverom da li je element već upotrebljen

Permutacije se mogu rekurzivno generisati veoma slično postupku generisanja varijacija bez ponavljanja. U rekurzivnoj funkciji obrađujemo jednu po jednu poziciju i na nju stavljamo one elemente skupa $\{1, \ldots, n\}$ koji se ne nalaze na prethodnim

pozicijama. Da bi se izbegla linearna pretraga prethodnih pozicija, moguće je koristiti pomoćni niz logičkih vrednosti u kome se za svaki element označava da li je već iskorišćen ili nije.



Rekurzivno generisanje permutacija niza 123 - na tekuću poziciju se postavlja jedan po jedan element niza 123, koji nije već postavljen na prethodne pozicije

```
// popunjava se permutacija a od pozicije i nadalje elementima
// skupa {1, ..., n} pri cemu se u nizu upotrebljen beleze
// upotrebljeni elementi u delu permutacije pre pozicije i
void permutacije(vector<int>& a, int n,
                 vector<bool>& upotrebljen, int i) {
  // permutacija je cela popunjena, pa je ispisujemo
  if (i == a.size())
    obradi(a);
  else {
    // na poziciju i stavljamo redom svaki neupotrebljen
    // element
    for (int x = 1; x <= n; x++)
      if(!upotrebljen[x]) {
        a[i] = x;
        upotrebljen[x] = true;
        permutacije(a, n, upotrebljen, i+1);
        upotrebljen[x] = false;
      }
  }
```

```
}

// ispisuje sve permutacije skupa {1, ..., n}
void permutacije(int n) {
  vector<int> a(n);
  vector<bool> upotrebljen(n+1, false);
  permutacije(a, n, upotrebljen, 0);
}
```

Rekurzivno generisanje permutacija bez eksplicitne provere da li je element već upotrebljen

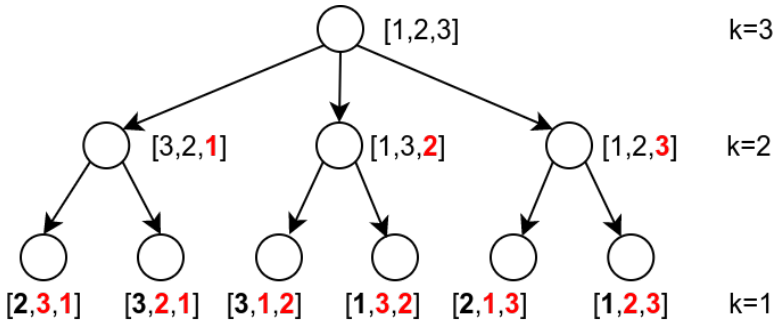
Ako se odreknemo uslova da permutacije budu uređene leksikografski, nije neophodno vršiti proveru da li je tekući element već upotrebljen tj. možemo postupiti na sledeći način.

Na poslednju poziciju u nizu u kojem čuvamo tekuću permutaciju treba da postavljamo jedan po jedan element skupa, a zatim da rekurzivno određujemo sve permutacije preostalih elemenata (alternativno bismo mogli da krenemo i od prve pozicije). Time se na svakom nivou rekurzije razlikuju prefiks permutacije koji sadrži elemente koje tek treba permutovati i sufiks permutacije koji sadrži elemente koji su fiksirani i već se nalaze na svojim pozicijama. Fiksirane elemente i elemente koje treba permutovati možemo čuvati u istom nizu. Neka na pozicijama $[0, k)$ čuvamo elemente koje treba permutovati, a na pozicijama $[k, n)$ čuvamo fiksirane elemente. Rekurzivna funkcija, dakle, prima niz, i broj k i pokušava da fiksiran sufiks elemenata na pozicijama $[k, n)$ na sve moguće načine proširi permutacijama elemenata na pozicijama $[0, k)$.

- Ako je $k = 1$, tada postoji samo jedna permutacija jednočlanog niza na poziciji 0, nju pridružujemo fiksiranim elementima (pošto je ona već na mestu 0 nema potrebe ništa dodatno raditi) i obrađujemo je (tj. ispisujemo).
- Ako je $k > 1$, tada je situacija komplikovanija. Razmotrimo poziciju $k-1$. Jedan po jedan element dela niza sa pozicija $[0, k)$ treba da dovodimo na mesto $k-1$ i da rekurzivno pozivamo permutovanje dela niza na pozicijama $[0, k-1)$. Ideja koja se prirodno javlja je da vršimo razmenu elementa na poziciji $k-1$ redom sa svim elementima iz intervala $[0, k)$ i da nakon svake razmene vršimo rekurzivne pozive.

Primer 6.1.4. *Na primer, ako je niz na početku* 123*, onda menjamo element* 3 *sa elementom* 1*, dobijamo* 321 *i pozivamo rekurzivno generisanje permutacija niza* 32 *sa fiksiranim elementom* 1 *na kraju. Zatim u početnom nizu menjamo element* 3 *sa elementom* 2*, dobijamo* 132 *i pozivamo rekurzivno generisanje permutacija niza* 13 *sa fiksiranim elementom* 2 *na kraju. Zatim u početnom nizu menjamo element* 3 *sa samim sobom, dobijamo* 123 *i pozivamo rekurzivno generisanje permutacija niza* 12 *sa fiksiranim elementom* 3 *na kraju.*

Ovaj postupak je prikazan i na slici.



Rekurzivno generisanje permutacija niza 123 korišćenjem razmena elemenata niza

Međutim, sa tim pristupom može biti problema. Naime, da bismo bili sigurni da će na poslednju poziciju stizati svi elementi niza, razmene moramo da vršimo u odnosu na *početno* stanje niza. Jedan način je da se pre svakog rekurzivnog poziva pravi kopija niza, ali postoji i efikasnije rešenje. Naime, možemo kao invarijantu funkcije nametnuti da je nakon svakog rekurzivnog poziva raspored elemenata u nizu isti kao pre poziva funkcije. Ujedno to treba da bude i invarijanta petlje u kojoj se vrše razmene. Na ulasku u petlju raspored elemenata u nizu biće isti kao na ulasku u funkciju. Vršimo prvu razmenu, rekurzivno pozivamo funkciju i na osnovu invarijante rekurzivne funkcije znamo da će raspored nakon rekurzivnog poziva biti isti kao pre njega. Da bismo održali invarijantu petlje, potrebno je niz vratiti u početno stanje. Međutim, znamo da je niz promenjen samo jednom razmenom, tako da je dovoljno uraditi istu tu razmenu i niz će biti vraćen u početno stanje. Time je invarijanta petlje očuvana i može se preći na sledeću poziciju. Kada se petlja završi, na osnovu invarijante petlje znaćemo da je niz isti kao na ulazu u funkciju. Na osnovu toga znamo i da će invarijanta funkcije biti održana i nije potrebno uraditi ništa dodatno nakon petlje.

```
void obradiSvePermutacije(vector<int>& permutacija, int k) {
  if (k == 1)
    obradi(permutacija);
  else {
    for (int i = 0; i < k; i++) {
      swap(permutacija[i], permutacija[k-1]);
      obradiSvePermutacije(permutacija, k-1);
      swap(permutacija[i], permutacija[k-1]);
    }
  }
}

void obradiSvePermutacije(int n) {
  vector<int> permutacija(n);
  for (int i = 1; i <= n; i++)
    permutacija[i-1] = i;
  obradiSvePermutacije(permutacija, n);
}
```

Bibliotečka funkcija za sledeću permutaciju

U jeziku C++ funkcija `next_permutation` deklarisana u zaglavlju `<algorithm>` određuje narednu permutaciju u odnosu na datu. Funkciji se prosleđuju dva iteratora koji ograničavaju raspon elemenata u kojima se nalazi permutacija.

```
// inicijalizujemo permutaciju na 1, 2, ..., n
vector<int> permutacija(n);
for (int i = 0; i < n; i++)
  permutacija[i] = i+1;

// ispisujemo permutaciju i trazimo narednu,
// sve dok naredna postoji
do {
  obradi(permutacija);
} while (next_permutation(begin(permutacija),
```

```
                                    end(permutacija)));
```

Zadatak: Sledeća particija

Particije broja n predstavljaju razlaganje tog broja na sabirke čija je vrednost između 1 i n. Na primer, broj 10 se može particionisati kao $5 + 2 + 2 + 1$. Svaka particija se može normalizovati tako što se pretpostavi, na primer, da su sabirci sortirani nerastuće. Napisati program koji za datu particiju određuje sledeću particiju u leksikografskom redosledu.

Opis ulaza

Sa standardnog ulaza se unosi normalizovana particija pri čemu su sabirci razdvojeni karakterom + (njihov zbir je manji od 1000).

Opis izlaza

Na standardni izlaz ispisati narednu normalizovanu particiju u leksikografskom redosledu (u jednoj liniji, pri čemu su sabirci razdvojeni karakterom +) ili reč `ne` ako takva particija ne postoji.

Primer

Ulaz	*Izlaz*
`5+2+2+1`	`5+3+1+1`

Rešenje

Da bismo dobili sledeću particiju u leksikografskom redosledu, potrebno je uvećati za jedan neki element koji je što bliži kraju niza, dok prefiks niza treba da ostane nepromenjen. Ako je particija jednočlana, tada je ona leksikografski najveća. Poslednji element niza nije moguće povećati za jedan, jer bi zbog očuvanja zbira elemenata particije neki element pre njega morao biti smanjen. Naredni kandidat za povećanje je pretposlednji element, a on se može uvećati na jedan samo ako se na mestu ispred njega ne nalazi element koji mu je jednak, jer bi se tada povećanjem pretposlednjeg elementa dobila particija koja nije normalizovana (uređena nerastući). U suprotnom razmatramo element pre pretposlednjeg i tako redom, sve dok ne naiđemo na element ispred kojeg ne stoji element koji mu je jednak. Taj element povećavamo za 1. Nakon toga potrebno je popraviti elemente iza tog uvećanog elementa, tako da particija bude leksikografski što manja. To će se desiti

ako se iza uvećanog elementa postave samo jedinice. Da se zbir ne bi promenio, broj postavljenih jedinica treba da bude za jedan manji od zbira svih elemenata iza elementa koji smo uvećali za jedan (taj zbir možemo izraunavati dok obilazimo niz unazad tražeći najdešnji element koji se može uvećati za 1).

Pošto se dužina particije može promeniti prilikom prelaska na sledeću particiju, umesto klasičnog niza particiju možemo predstaviti nekim oblikom niza koji dopušta dodavanje elemenata na kraj.

```
bool sledecaParticija(vector<int>& particija)
{
  int k = particija.size();
  // ako je tekuca particija jednoclana, ne postoji sledeca
  if (k == 1)
    return false;

  // pronalazimo poziciju prvog elementa zdesna koji se moze
  // uvecati i ujedno racunamo zbir elemenata iza njega
  int i;
  int zbir = particija[k-1];
  for (i = k-2; i > 0 && particija[i] == particija[i-1]; i--)
    zbir += particija[i];

  // uklanjamo sve elemente iza pozicije i
  particija.resize(i+1);

  // uvecavamo element koji smemo uvecati
  particija[i]++;

  // dodajemo jedinice do kraja particije
  for (int m = 0; m < zbir - 1; m++)
    particija.push_back(1);
  return true;
}
```

Zadatak: Sve particije

Particije broja n predstavljaju razlaganje tog broja na sabirke čija je vrednost između 1 i n. Na primer, broj 10 se može particionisati kao $5 + 2 + 2 + 1$. Svaka particija se može normalizovati tako što se pretpostavi, na primer, da su sabirci sortirani nerastuće. Napiši program koji ispisuje sve particije datog broja.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 25$).

Opis izlaza

Na standardni izlaz ispisati sve normalizovane particije broja n, sortirane leksikografski rastuće.

Primer

Ulaz	*Izlaz*
5	1 1 1 1 1
	2 1 1 1
	2 2 1
	3 1 1
	3 2
	4 1
	5

Rešenje

Svaka particija ima svoj prvi sabirak. Svakoj particiji broja n kojoj je prvi sabirak s (pri čemu je $1 \leq s \leq n$) jednoznačno odgovara neka particija broja $n - s$, što ukazuje da se problem može rešavati induktivno-rekurzivnom konstrukcijom. Pošto je sabiranje komutativno, da ne bismo suštinski iste particije ponavljali više puta nametnućemo uslov da sabirci u svakoj particiji budu sortirani nerastuće. Dakle, ako je prvi sabirak s, svi sabirci iza njega moraju da budu manji ili jednaki od s. Zato nam nije dovoljno samo da umemo da generišemo sve particije broja $n - s$, već je potrebno da ojačamo induktivnu hipotezu. Pretpostavićemo da se u datom vektoru na pozicijama $[0, i)$ nalaze ranije postavljeni elementi particije i da je zadatak procedure da taj niz dopuni na sve moguće načine particijama broja n u kojima su svi sabirci manji ili jednaki s_{max}. Izlaz iz rekurzije predstavljaće slučaj $n = 0$ u kom je jedina moguća particija broja 0 prazan skup, u kome nema sabiraka. Tada smatramo da je particija uspešno formirana i obrađujemo sadržaj vektora.

Rekurzija po pozicijama

Jedan način da se particije dopune je da se razmotre sve moguće varijante za sabirak na poziciji i. Na osnovu uslova oni moraju biti veći od nule, manji ili jednaki s_{max}, a prirodno je da moraju da budu i manji ili jednaki od n (jer prvi sabirak ne može biti veći od zbira prirodnih brojeva). Ako je m manji od brojeva n i s_{max}, mogući prvi sabirci su svi brojevi $1 \leq s' \leq m$. Kada fiksiramo sabirak s' niz rekurzivno dopunjavamo svim particijama broja $n-s'$ u kojima su svi sabirci manji ili jednaki s', jer je potrebno preostali deo zbira predstaviti kao particiju brojeva koji nisu veći od s'.

U glavnoj funkciji ćemo alocirati niz dužine n (jer najduža particija ima n sabiraka koji su svi jednaki 1) i zahtevaćemo da se taj niz popuni počevši od pozicije 0 particijama broja n u kojima su svi sabirci manji ili jednaki n.

```
void obradiParticije(int n, int smax, vector<int>& particija,
                     int k) {
  if (n <= 0)
    obradi(particija, k);
  else {
    for (int s = 1; s <= min(n, smax); s++) {
      particija[k] = s;
      obradiParticije(n-s, s, particija, k+1);
    }
  }
}

void obradiParticije(int n) {
  vector<int> particija(n);
  obradiParticije(n, n, particija, 0);
}
```

Rekurzija po vrednostima

Umesto da se analiziraju sve moguće vrednosti sabirka na poziciji i, moguće je razmatrati samo dve mogućnosti: prvu da se na poziciji i javlja sabirak s_{max}, a drugu da se na poziciji i javlja neki sabirak strogo manji od s_{max}. Prvi slučaj je moguć

samo ako je $n \geq s_{max}$ i kada se na poziciju i postavi s_{max} niz dopunjujemo od pozicije $i+1$ particijama broja $n - s_{max}$ u kojima su svi sabirci manji ili jednaki s_{max}. Drugi slučaj je uvek moguć i tada particiju dopunjujemo particijama broja n u kojima je najveći sabirak $s_{max} - 1$. U zavisnosti od redosleda ova dva rekurzivna poziva određuje se da li će permutacije biti sortirane leksikografski rastuće ili opadajuće.

```
void obradiParticije(int n, int smax, vector<int>& particija,
                     int k) {
  if (n == 0)
    obradi(particija, k);
  else {
    if (smax == 0) return;
    obradiParticije(n, smax-1, particija, k);
    if (smax <= n) {
      particija[k] = smax;
      obradiParticije(n-smax, smax, particija, k+1);
    }
  }
}

void obradiParticije(int n) {
  vector<int> particija(n);
  obradiParticije(n, n, particija, 0);
}
```

Leksikografski sledeća particija

Još jedan način da se reši zadatak je da se iterativno nabrajaju sledeće particije u leksikografskom redosledu (krenuvši od particije koja ima n jedinica) sve dok se ne dođe do poslednje particije koja sadrži samo broj n. Pronalaženje sledeće particije u leksikografskom redosledu moguće je uraditi funkcijom koja je opisana u zadatku *Sledeća particija*.

```
void obradiParticije(int n) {
  vector<int> particija(n, 1);
  int k = n;
  do {
    obradi(particija, k);
  } while (sledecaParticija(particija, k));
}
```

7. Iscrpna pretraga, pretraga sa povratkom

U nastavku ćemo razmatrati probleme sledećeg tipa:

- Ispitati da li postoji neki kombinatorni objekat (često predstavljen torkom brojeva) koji zadovoljava neke date uslove ili, alternativno, prebrojati ili nabrojati sve takve objekte. Ovakvi problemi se nazivaju *problemi zadovoljavanja ograničenja* (engl. constraint satisfaction problems, CSP). Na primer, potrebno je proveriti da li postoji neki podskup datog skupa brojeva čiji je zbir jednak datoj vrednosti.

- Među svim kombinatornim objektima koji zadovoljavaju neke date uslove naći najbolji tj. naći onaj objekat na kome je vrednost neke date funkcije minimalna ili maksimalna. Ovakvi problemi se nazivaju *problemi optimizacije uz ograničenja* (engl. constraint optimization problems, COP). Ovi problemi se nazivaju i problemi *kombinatorne optimizacije*.

Ovakvi problemi se često rešavaju raznim varijantama algoritama pretrage u kojima se nabrajaju i proveravaju neki kandidati za rešenja.

Algoritmi grube sile podrazumevaju da se u pretrazi iscrpno nabroje svi kandidati za rešenje i da se za svakog od njih proveri da li zadovoljava uslov (ako je potrebno pronaći bilo koji element koji zadovoljava dati uslov) ili da li je optimalan (ako je potrebno pronaći element koji minimalizuje ili maksimalizuje datu ciljnu funkciju). Svi kandidati se ponekada mogu nabrojati jednostavno, ugnežđenim petljama, dok je nekada potrebno koristiti neki složeniji postupak za nabrajanje određene familije kombinatornih objekata (kombinacija, permutacija, varijacija, podskupova, particija itd. ali i drugih, specifičnih familija kombinatornih objekata).

Algoritmi pretrage sa povratkom tj. *bektreking* (engl. backtracking) poboljšavaju tehniku grube sile tako što vrše provere i tokom generisanja kandidata za rešenja i tako što se odbacuju parcijalno popunjeni kandidati, za koje se može unapred utvrditi da se ne mogu proširiti do ispravnog tj. optimalnog rešenja. Dakle, bektreking podrazumeva da se tokom obilaska u dubinu drveta kojim se predstavlja prostor potencijalnih rešenja odsecaju oni delovi drveta za koje se unapred može utvrditi da ne sadrže ni jedno rešenje problema tj. da ne sadrže optimalno rešenje, pri čemu se odsecanje vrši i u čvorovima bliskim korenu koji mogu da sadrže i samo parcijalno popunjene kandidate za rešenja. Dakle, umesto da se čeka da se tokom pretrage stigne do lista (ili eventualno unutrašnjeg čvora koji predstavlja nekog kandidata za rešenje) i da se provera zadovoljenosti uslova ili optimalnosti vrši tek tada, prilikom pretrage sa povratkom provera se vrši u svakom koraku i vrši se provera parcijalno popunjenih rešenja (obično su to neke parcijalno popunjene torke brojeva).

Na primer, ako se proverava da li postoji podskup nekog skupa pozitivnih brojeva, čiji je zbir elemenata jednak datom broju i ako se ustanovi da nekoliko odabranih elementa polaznog skupa imaju zbir veći od tog broja, taj deo prostora pretrage se odmah može odseći i nema potrebe generisati sve podskupove u kojima su ti elementi uključeni (jer unapred znamo da nijedan od njih neće predstavljati zadovoljavajuće rešenje).

Efikasnost algoritama zasnovanog na ovom obliku pretrage uveliko zavisi od kvaliteta kriterijuma na osnovu kojih se vrši odsecanje. Iako obično složenost najgoreg slučaja ostaje eksponencijalna (kakva je po pravilu kod algoritama grube sile tj. iscrpne pretrage), pažljivo odabrani kriterijumi odsecanja mogu odseći jako velike delove pretrage (koji su često takođe eksponencijalne veličine u odnosu na dimenzije ulaznog problema) i time značajno ubrzati proces pretrage.

Naglasimo i da neki autori ne prave eksplicitnu razliku između algoritama grube sile (iscrpne pretrage) i algoritama pretrage sa povratkom i da u algoritme tipa pretrage sa povratkom ubrajaju sve algoritme u kojima se drvo koje sadrži sva potencijalna rešenja obilazi u dubinu.

Formulišimo opštu shemu rekurzivne implentacije pretrage sa povratkom. Pretpostavljamo, jednostavnosti radi, da su parametri procedure pretrage trenutni niz `v` (u kome se smeštaju torke brojeva koji predstavljaju kandidate za rešenja) i dužina trenutno popunjenog dela niza `k`, pri čemu je niz alociran tako da se u njega može smestiti i najduže rešenje. Takođe, pretpostavljamo da na raspolaganju imamo funkciju `odsecanje` koja proverava da li je trenutna torka smeštena u niz (na prvih `k` pozicija) kandidat da bude rešenje ili deo nekog rešenja. Pretpostavljamo i da znamo

da li trenutna torka predstavlja rešenje (to utvrđujemo funkcijom `jesteResenje` — u realnim situacijama se taj uslov često svede ili na to da je uvek tačan, što se dešava kada je svaki čvor drveta potencijalni kandidat za rešenje ili na to da je tačan samo u listovima, što se detektuje tako što se proveri da je `k` dostiglo dužinu niza `v`). Na kraju, pretpostavljamo i da za svaku torku dužine `k` možemo eksplicitno odrediti sve kandidate za vrednost na poziciji `k` (pozivom funkcije `kandidati(v, k)`). Rekurzivnu pretragu tada možemo realizovati narednim (pseudo)kodom.

```
void pretraga(const vector<int>& v, int k) {
   if (odsecanje(v, k))
      return;
   if (jesteResenje(v, k))
      ispisi(v, k);
   for (int x : kandidati(v, k)) {
      v[k] = x;
      pretraga(v, k+1);
   }
}
```

Alternativno, provere umesto na ulazu u funkciju možemo vršiti pre rekurzivnih poziva (čime se malo štedi na broju rekurzivnih poziva, ali se ponekad implementacija može malo zakomplikovati).

```
void pretraga(vector<int>& v, int k) {
   if (jesteResenje(v, k))
       ispisi(v, k);
   for (int x : kandidati(v, k)) {
      v[k] = x;
      if (!odsecanje(v, k+1)) {
         pretraga(v, k+1);
      }
   }
}
```

Rekurzije je moguće osloboditi se uz korišćenje steka.

U svim čvorovima koji predstavljaju kandidate za rešenje (to su obično potpuno

popunjene torki brojeva) potrebno je potpuno precizno ispitati da li je tekući kandidat ispravno tj. optimalno rešenje. To znači da u prethodnom kodu funkcija `jesteResenje` mora potpuno precizno da detektuje da li trentuna torka jeste ili nije ispravno tj. optimalno rešenje (niti sme ispravno rešenje da proglasi neispravnim, jer će se tada ono propustiti, niti sme neispravno rešenje da proglasi ispravnim, jer će se tada ono greškom pojaviti u spisku rešenja). Sa druge strane, u čvorovima u kojima se proveravaju parcijalno popunjena rešenja, provera kriterijuma odsecanja ne mora biti potpuno precizna — sa jedne strane dopušteno je da se ne odseku delovi drveta u kojima nema rešenja (time algoritam ostaje korektan, ali je neefikasniji), međutim, ako se odsecanje izvrši moramo biti apsolutno sigurni da se u odsečenom delu drveta ne nalazi nijedno ispravno rešenje tj. da se ne nalazi se optimalno rešenje. U prethodnom kodu to znači da kada funkcija `odsecanje` vrati vrednost tačno, moramo biti apsolutno sigurni da se trenutna torka smeštena na prvih `k` pozicija u nizu `v` ne može nikako dopuniti do ispravnog tj. optimalnog rešenja (jer bismo u suprotnom propustili neka rešenja). Sa druge strane, ta funkcija može da vrati vrednost netačno praktično bilo kada i time neće biti narušena korektnost (ali se narušava efikasnost). U praksi se ponekad dešava da je veoma komplikovano napraviti funkciju `odsecanje` koja potpuno precizno određuje da li se torka može produžiti do traženog rešenja problema tako da se zadovoljavamo kriterijumima odsecanja koji se mogu relativno jednostavno proveriti, a garantuju korektnost.

Dodatno ubrzavanje algoritma može da se napravi ako se na neki način može definisati funkcija koja ponekad može da pogodi vrednost `x` koju treba upisati na poziciju `k`, bez isprobavanja različitih kandidata. Na primer, ako se prilikom popunjavanja magičnog kvadrata (kvadrata u kom su brojevi raspoređeni tako da sve vrste, sve kolone i obe dijagonale imaju isti, unapred poznat, zbir) u nekoj vrsti popune svi elementi osim jednog, lako možemo da izračunamo koja vrednost mora da bude upisana na tom preostalom mestu. Takve korake zovemo koraci *zaključivanja* (engl. inferrence). Ni u ovom slučaju nije potrebna potpuna preciznost. Samo je bitno obezbediti da kada se zaključivanjem predloži neka konkretna vrednost, da je ostale mogućnosti bezbedno odseći, jer se među njima ne krije nijedno ispravno rešenje. Sa druge strane, ako je zaključivanje previše komplikovano ostvariti u nekom koraku pretrage, ono se može preskočiti (algoritam je korektan i bez ikakvog oblika zaključivanja).

Prethodne funkcije ispisuju sva rešenja problema tj. sve torke koje zadovoljavaju date uslove. Pretragu je moguće prekinuti i nakon pronalaska prvog rešenja (tada funkcija obično vraća podatak o tome da li jeste ili nije pronašla rešenje u delu

drveta koji pretražuje).

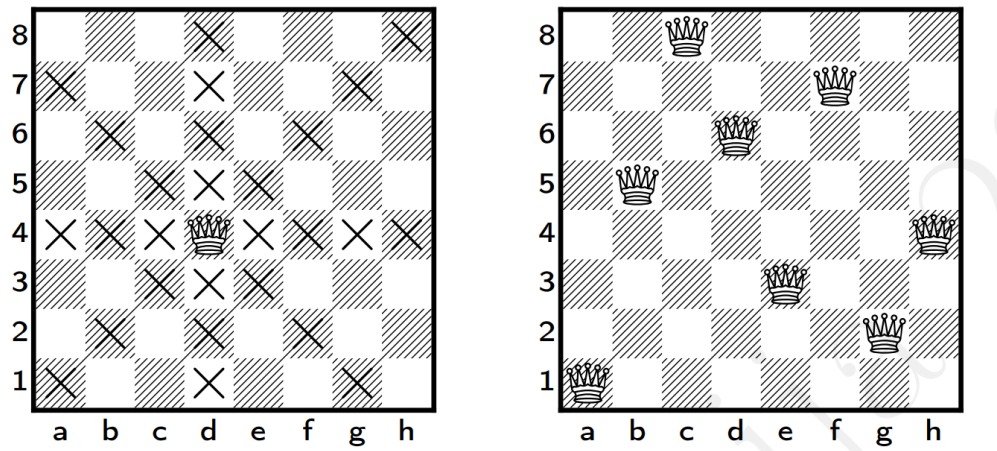
```
bool pretraga(const vector<int>& v, int k) {
   if (odsecanje(v, k))
      return false;
   if (jesteResenje(v, k)) {
      ispisi(v, k);
      return true;
   }
   for (int x : kandidati(v, k)) {
      v[k] = x;
      if (pretraga(v, k+1))
         return true;
   }
   return false;
}
```

Kod rešavanja optimizacionih problema, odsecanje može da nastupi i kada se proceni da u delu drveta koje se trenutno pretražuje ne postoji rešenje koje je bolje od najboljeg trenutno pronađenog rešenja. Dakle, rešenja pronađena u dosadašnjem delu pretrage se koriste da bi se odredile granice na osnovu kojih se vrši odsecanje u drugim delovima pretrage. Ovakav oblik optimizacije naziva se ponekad *grananje sa ograničavanjem* (engl. branch and bound).

7.0.1 *Problem 8 dama*

Ilustrujmo pristup grube sile, a zatim i pretrage sa povratkom na jednom klasičnom problemu, *problemu osam dama*, u kojem se zahteva da se osam dama (kraljica) rasporedi na šahovskoj tabli tako da nijedna ne napada neku drugu (dame se napadaju horizontalno, vertikalno i dijagonalno, kako je prikazano na slici 7.1, levo). Na slici 7.1, desno, prikazano je jedno od 92 rešenja koliko ih problem ima. Problemom se (bez upotrebe računara) bavio još znameniti matematičar Gaus, a uz primenu računara do rešenja se dolazi prilično jednostavno.

Prvo pitanje koje se postavlja je kako reprezentovati problem. Jedna mogućnost je da se položaj dama predstavi matricom dimenzija 8×8 koja sadrži istinitosne vrednosti (recimo 0 na onim poljima na kojima nije dama i 1 na onim poljima na kojima jeste dama). Međutim, može se zaključiti da se u svakoj vrsti table mora



Slika 7.1

naći tačno jedna dama, pa se svako rešenje može jednostavno predstaviti nizom brojeva (od 1 do 8, ili od 0 do 7) koji predstavljaju redne brojeve kolona u kojima se nalaze dame (po jedan broj za svaku vrstu, redom). Na primer, raspored na slici 7.1 može se predstaviti nizom $0, 6, 4, 7, 1, 3, 5, 2$. Dakle, svaki potencijani raspored dama predstavlja jednu varijaciju dužine osam niza brojeva od 0 do 7.

Provera da li se dame na poljima (k_1, v_1) i (k_2, v_2) napadaju može se izvršiti primenom aritmetičkih operacija. Naime, dve dame su u istoj koloni ako i samo ako je $k_1 = k_2$, a u istoj su vrsti ako i samo ako je $v_1 = v_2$. Dve dame se napadaju dijagonalno ako i samo ako je jednakokrak trougao čija su temena polja dve dame i polje koje se dobije projektovanjem polja jedne dame na vrstu u kojoj se nalazi druga dama. Dužine kateta tog jednakokrakog trougla određene su apsolutnim vrednostima razlike između rednih brojeva vrsta i između rednih brojeva kolona polja na kojih se dame nalaze. Dakle, provera da li se dve dame napadaju može se izvršiti pozivanjem naredne tri funkcije.

```
int ista_vrsta(int v1, int k1, int v2, int k2)
{
  return v1 == v2;
}
```

```
int ista_kolona(int v1, int k1, int v2, int k2)
{
  return k1 == k2;
}

int ista_dijagonala(int v1, int k1, int v2, int k2)
{
   return abs(v2 - v1) == abs(k2 - k1);
}
```

Provera da li niz kolona u kojima se dame nalaze predstavlja rešenje može se izvršiti na sledeći način (vrednost 8 zamenjena je simboličkim imenom `DIM`, pa se program može lako modifikovati da radi i za druge dimenzije table):

```
const int DIM = 8;
// ...

int jeste_resenje(int kolone_dama[])
{
  int v1, v2;
  for (v1 = 0; v1 < DIM; v1++)
    for (v2 = v1+1; v2 < DIM; v2++)
      if (ista_vrsta(v1, kolone_dama[v1],
                     v2, kolone_dama[v2]) ||
          ista_kolona(v1, kolone_dama[v1],
                      v2, kolone_dama[v2]) ||
          ista_dijagonala(v1, kolone_dama[v1],
                          v2, kolone_dama[v2]))
        return 0;
  return 1;
}
```

Kako je izabranom reprezentacijom osigurano da nema dve dame u istoj vrsti, nema potrebe pozivati funkciju `ista_vrsta`. Pozivi preostale dve funkcije mogu se zameniti njihovim (kratkim) telima, pa se k^od može malo uprostiti.

```
int jeste_resenje(int kolone_dama[])
{
  int v1, v2;
  for (v1 = 0; v1 < DIM; v1++)
    for (v2 = v1+1; v2 < DIM; v2++)
      if (kolone_dama[v1] == kolone_dama[v2] ||
          abs(v1-v2) == abs(kolone_dama[v1]-kolone_dama[v2]))
        return 0;
  return 1;
}
```

Ostaje pitanje kako nabrojati sve moguće nizove koji reprezentuju položaj dama. Jedno, prilično naivno rešenje je da se za to upotrebi 8 ugnežđenih petlji (za svaku vrstu po jedna). Umesto toga može se upotrebiti sledeće rekurzivno rešenje.

```
void ispisi(int kolone_dama[])
{
  static int rbr; // redni broj resenja
  cout << rbr++ << ": ";
  for (int j = 0; j < DIM; j++)
    cout << kolone_dama[j];
  cout << endl;
}

void dame(int kolone_dama[], int broj_postavljenih_dama)
{
  if (broj_postavljenih_dama == DIM) {
    if (jeste_resenje(kolone_dama))
      ispisi(kolone_dama);
  }
  else {
    int j;
    for (j = 0; j < DIM; j++) {
      kolone_dama[broj_postavljenih_dama] = j;
      dame(kolone_dama, broj_postavljenih_dama + 1);
    }
```

```
  }
}
```

Primetimo da navedena funkcija generiše sve varijacije sa ponavljanjem dužine `DIM` nad `DIM` elemenata, slično kao što je opisano u delu **??**. Tih varijacija ima $8^8 = 16777216$. Iako navedeni k^od ispravno radi, može se značajno unaprediti. Pošto se ni u jednoj koloni ne mogu nalaziti dve dame, potrebno je da su svi brojevi različiti. Dakle, svaki potencijani raspored dama predstavlja jednu moguću permutaciju niza brojeva od 0 do 7 (s druge strane, ne zadovoljava svaka permutacija uslov da se nikoje dve dame ne napadaju). Imajući to u vidu, malko unapređen pristup grubom silom koristio bi generisanje svih permutacija (njih $8! = 40320$) i za svaku od njih proveru da li zadovoljaju uslov o nenapadanju dama, tj. da li predstavlja ispravno rešenje. I generisanje permutacija može se implementirati rekurzivno (kao što je opisano u delu **??**). U ovom pristupu, provera da li se u nekoj koloni nalaze dve dame prebacuje se iz faze konačne provere u fazu generisanja potencijalnih kandidata za rešenje (u vidu uslova `!popunjena_kolona[j]`). To je primer *ranog odsecanja* tokom pretrage koje može dosta doprineti njenoj efikasnosti. Algoritmi koji odsecaju grane stabla pretrage za koje se rano može utvrditi da se u njima ne nalazi rešenje nazivaju se *bektreking* (engl. backtracking) algoritmi (što se ponekad prevodi kao *pretraga sa povratkom* ili *pretraga sa odsecanjem*).

```
int jeste_resenje(int kolone_dama[])
{
  int v1, v2;
  for (v1 = 0; v1 < DIM; v1++)
    for (v2 = v1+1; v2 < DIM; v2++)
      if (abs(v1-v2) == abs(kolone_dama[v1]-kolone_dama[v2]))
        return 0;
  return 1;
}

void dame(int kolone_dama[], int popunjena_kolona[],
          int broj_postavljenih_dama)
{
  if (broj_postavljenih_dama == DIM) {
    if (jeste_resenje(kolone_dama))
      ispisi(kolone_dama);
```

```
  }
  else {
    int j;
    for (j = 0; j < DIM; j++) {
      if (!popunjena_kolona[j]) {
        kolone_dama[broj_postavljenih_dama] = j;
        popunjena_kolona[j] = 1;
        dame(kolone_dama, popunjena_kolona,
             broj_postavljenih_dama + 1);
        popunjena_kolona[j] = 0;
      }
    }
  }
}
```

Uz navedeni k^od, pozivom naredne funkcije `main`:

```
int main()
{
  int kolone_dama[DIM], popunjena_kolona[DIM];
  for (int j = 0; j < DIM; j++)
    popunjena_kolona[j] = 0;
  dame(kolone_dama, popunjena_kolona, 0);
}
```

dobija se 92 rešenja problema:

```
0: 04752613
1: 05726314
2: 06357142
...
91: 73025164
```

U unapređivanju navedenog algoritma, može se otići još jedan korak dalje. Naime, često se do efikasnije pretrage dolazi ako se odsecanje vrši što ranije. Primetimo da se u prethodnom rešenju provera dijagonala vrši tek na kraju, kada su sve dame postavljene. Mnogo je, međutim, bolje odsecanje vršiti čim se na tablu postavi nova dama koja se napada sa nekom od postojećih dama. Tako se osigurava da

će u svakom koraku pretrage dame na tabli zadovoljavati dati uslov nenapadanja i stoga više nije potrebno vršiti nikakve provere na samom kraju, onda kada su sve dame postavljene.

```
int dama_se_moze_postaviti(int kolone_dama[],
                           int broj_postavljenih_dama,
                           int v, int k)
{
  int v1;
  for (v1 = 0; v1 < broj_postavljenih_dama; v1++)
    if (ista_dijagonala(v1, kolone_dama[v1], v, k))
      return 0;
  return 1;
}

void dame(int kolone_dama[], int popunjena_kolona[],
          int broj_postavljenih_dama)
{
  if (broj_postavljenih_dama == DIM)
    ispisi(kolone_dama);
  else {
    int j;
    for (j = 0; j < DIM; j++) {
      if (!popunjena_kolona[j] &&
          dama_se_moze_postaviti(kolone_dama,
                                 broj_postavljenih_dama,
                                 broj_postavljenih_dama, j)) {
          kolone_dama[broj_postavljenih_dama] = j;
          popunjena_kolona[j] = 1;
          dame(kolone_dama, popunjena_kolona,
               broj_postavljenih_dama + 1);
          popunjena_kolona[j] = 0;
      }
    }
  }
}
```

Zadatak: Sudoku

Napisati program koji popunjava Sudoku zagonetku čiji je cilj da se u matricu dimenzije 9 puta 9 rasporede brojevi od 1 do 9, tako da u svakoj vrsti, u svakoj koloni i u svakom od 9 kvadrata dimenzije 3 puta 3 svi brojevi različiti.

Opis ulaza

Sa standardnog ulaza se učitava matrica dimenzije 9 puta 9 u kojoj su već upisani neki brojevi, a na poljima koja su prazna upisana je nula.

Opis izlaza

Na standardni izlaz ispisati rešenje zagonetke (test-primeri će biti takvi da je rešenje sigurno jedinstveno).

Primer

Ulaz	*Izlaz*
749030680	749132685
006508000	326548179
000760324	518769324
800057060	892357461
407000508	437621598
050980002	651984732
184076000	184276953
000403800	275493816
063010247	963815247

Rešenje

Zadatak rešavamo bektrekingom tj. rekurzivno implementiranom pretragom u dubinu. Rekruzivna funkcija uz matricu koja se popunjava dobija i redni broj polja koje treba popuniti (ona vraća informaciju o tome da li je matricu bilo moguće potpuno popuniti). Popunjavanje kreće od gornjeg levog ugla i teče vrstu po vrstu, sve dok ne popunimo celu matricu. Koordinate polja se veoma jednostavno mogu odrediti na osnovu njegovog rednog broja. Ako je trenutno polje već popunjeno (jer su u startu neka polja već popunjena), tada odmah prelazimo na naredno polje. U suprotnom proveravamo sve moguće vrednosti za to polje. Nakon upisa vrednosti proveravamo da li je time napravljen neki konflikt tj. da li se desilo da je u istoj vrsti, u istoj koloni ili u istom kvadratu već postojao broj koji je upisan. Ako jeste, pretragu prekidamo, a ako nije, nastavljamo je dalje, popunjavanjem nared-

nog polja. Čim neki od rekurzivnih poziva uspe da popuni celu matricu, pretraga se prekida i naredni rekurzivni pozivi se ne vrše.

```
const int n = 3;

bool konflikt(const vector<vector<int>>& A, int i, int j) {
  // da li se A[i][j] nalazi već u koloni j
  for (int k = 0; k < n * n; k++)
    if (k != i && A[i][j] == A[k][j])
      return true;

  // da li se A[i][j] nalazi već u vrsti i
  for (int k = 0; k < n * n; k++)
    if (k != j && A[i][j] == A[i][k])
      return true;

  // da li se A[i][j] već nalazi u kvadratu koji sadrži
  // polje (i, j)
  int x = i / n, y = j / n;
  for (int k = x * n; k < (x + 1) * n; k++)
    for (int l = y * n; l < (y + 1) * n; l++)
      if ((k != i || l != j) && A[i][j] == A[k][l])
        return true;

  // ne postoji konflikt
  return false;
}

bool sudoku(vector<vector<int>>& A, int rbr) {
  int i = rbr / (n*n), j = rbr % (n*n);
  // ako je polje (i, j) već popunjeno
  if (A[i][j] != 0) {
    // ako je u pitanju poslednje polje, uspešno smo popunili
    // ceo sudoku
    if (rbr == n * n * n * n - 1)
      return true;
```

```
    // rekurzivno nastavljamo sa popunjavanjem
    return sudoku(A, rbr + 1);
  } else {
    // razmatramo sve moguće vrednosti koje možemo da upišemo
    // na polje (i, j)
    for (int k = 1; k <= n*n; k++) {
      // upisujeAo vrednost k
      A[i][j] = k;
      // ako time napravljen neki konflikt, nastavljamo
      // popunjavanje (pošto je polje popunjeno, na sledeće
      // polje će se automatski preći u rekurzivnom pozivu)
      // ako se sudoku uspešno popuni, prekidaAo pretragu
      if (!konflikt(A, i, j))
        if (sudoku(A, rbr))
          return true;
    }
    // poništavamo vrednost upisanu na polje (i, j), jer se
    // popunjena polja smatraju fiksiranim
    // (datim u postavci problema)
    A[i][j] = 0;
    // konstatujemo da ne postoji rešenje
    return false;
  }
}
```

Zadatak: Bojenje grafa sa tri boje

U jednoj zemlji postoji nekoliko planinskih vrhova na kojima će se postaviti predajnici najsavremenije mobilne mreže. Oni mogu da rade na jednoj od tri različite radio-frekvencije. Svaki predajnik može da prenosi specijalni signal drugim predajnicima koji su blizu njega, pri čemu dva predajnika koji su blizu jedan drugome ne smeju da koriste istu frekvenciju. Napisati program koji određuje da li je moguće dodeliti frekvencije svim predajnicima tako da nema sudaranja.

Opis ulaza

Sa standardnog ulaza se učitava broj predajnika n ($1 \leq n \leq 100$), a nakon toga

broj parova bliskih predajnika m ($n-1 \leq m \leq \frac{n(n-1)}{2}$). Nakon toga, u narednih m redova se učitavaju parovi bliskih predajnika (svi predajnici su obeleženi brojevima od 0 do $n-1$). Sistem je građen tako da se signal sigurno može preneti od bilo kog do bilo kog drugog predajnika.

Opis izlaza

Na standardni izlaz ispisati oznake frekvencija (`1`, `2` i `3`) koje su redom dodeljene predajnicima ili `-` ako frekvencije nije moguće dodeliti tako da se bliski predajnici ne sudaraju. Ako je frekvencije moguće dodeliti na više načina, ispisati onaj koji je najmanji u leksikografskom redosledu.

Primer

Ulaz	*Izlaz*

`5`
`5`
`0 1`
`0 4`
`1 2`
`1 4`
`2 3`
`2 4`
`3 4`
	`1 2 1 2 3`

Rešenje

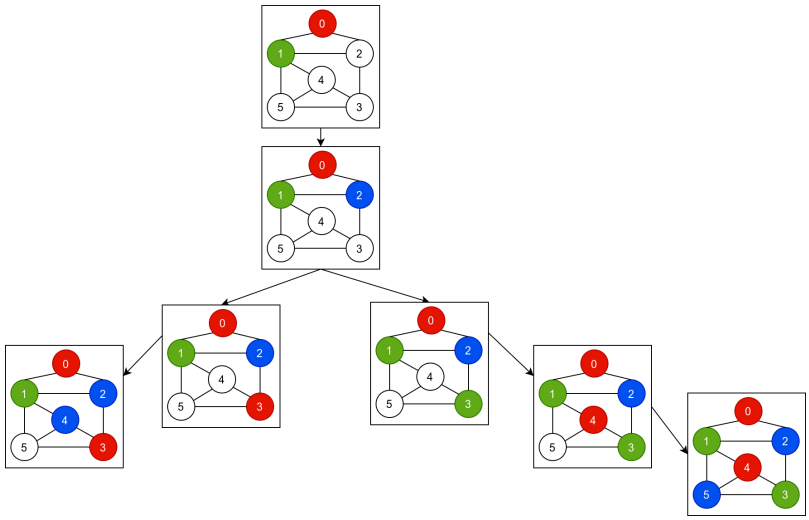
Ovaj problem je u literaturi poznat kao problem 3-bojenja grafova (svaki predajnik predstavlja čvor grafa, a svaka frekvencija jednu od 3 dopuštene boje).

Zadatak se jednostavno (ali ne i efikasno) rešava bektreking pretragom.

Primetimo da je problem simetričan i da ako se prvom čvoru može dodeliti neka boja, oznake boja se mogu promeniti tako da se tom prvom čvoru dodeli bilo koja druga boja. Stoga možemo pretpostaviti da je prvi čvor obojen bojom 1 (jer se traži najmanji raspored boja po leksikografskom poretku). Ni jedan od njegovih suseda (a oni sigurno postoje, jer je graf povezan) ne može biti obojen bojom 1. Pošto se traži najmanji raspored boja leksikografskom poretku, njegov sused koji ima najmanji redni broj treba da bude obojen bojom 2. Za ostale čvorove pokušavamo bojenje raznim bojama.

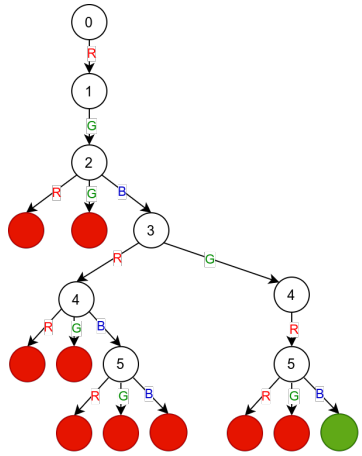
Primer 7.0.1. *Na slikama je prikazan primer postupka bojenja grafa sa tri boje. Pre početka pretrage čvor 0 se boji u crveno, a čvor 1 (najmanji sused čvora 0) u zeleno.*

Tada čvor 2 mora da se oboji u plavo. Ako se čvor 3 oboji prvom raspoloživom bojom (crvenom), tada čvor 4 mora da se oboji u plavo i čvor 5 ne može da se oboji (jer je susedan i sa crvenim i sa zelenim i sa plavim čvorom). Tada se pretraga vraća unatrag i čvor 3 boji u narednu raspoloživu boju – zelenu. Čvor 4 se boji u prvu raspoloživu boju – crvenu, nakon čega čvor 5 mora da se oboji u plavu. U tom trenutku je ceo graf uspešno obojen sa tri boje.



Primer bojenja grafa sa 3 boje

```
// funkcija boji dati cvor, pri cemu su zadati
// susedi svih cvorova i boje do sada obojenih cvorova
bool oboj(const vector<vector<int>>& susedi, int cvor,
          vector<int>& boje) {
  // ako su svi cvorovi obojeni, bojenje sa 3 boje je uspesno
  int brojCvorova = susedi.size();
  if (cvor >= brojCvorova)
    return true;

  // ako je cvor vec obojen, preskacemo ga
  if (boje[cvor] != 0)
    return oboj(susedi, cvor + 1, boje);
```



Drvo pretrage pridruženom bojenju grafa sa 3 boje

```
// pokusavamo da cvoru dodelimo svaku od 3 raspolozive boje
for (int boja = 1; boja <= 3; boja++) {
  // linearnom pretragom proveravamo da li je moguce obojiti
  // cvor u tekucu boju
  bool mozeBoja = true;
  // proveravamo sve susede
  for (int sused : susedi[cvor])
    // ako je neki od njih vec obojen u tekucu boju
    if (boje[sused] == boja) {
      // bojenje nije moguce
      mozeBoja = false;
      break;
    }
  if (mozeBoja) {
    // bojimo tekuci cvor
    boje[cvor] = boja;
    // pokusavamo rekurzivno bojenje narednog cvora i ako
    // uspemo, tada je bojenje moguce
    if (oboj(susedi, cvor+1, boje))
      return true;
    // ponistavamo boju tekuceg cvora
    boje[cvor] = 0;
```

```
    }
  }

  // probali smo sve tri boje i ni jedno bojenje nije moguce
  return false;
}

bool oboj(const vector<vector<int>>& susedi, vector<int>& boje) {
  // broj cvorova grafa
  // prva cvor 0 i njegov prvi sused se boje u boje 1 i 2
  boje[0] = 1; boje[*min_element(begin(susedi[0]),
                                 end(susedi[0]))] = 2;
  // krecemo bojenje od cvora 0
  return oboj(susedi, 0, boje);
}
```

7.1 Odsecanje u pretrazi sa povratkom

Pronalaženje dobrih kriterijuma za odsecanje pretrage u najvećoj meri utiče na ukupnu efikasnost pretrage sa povratkom. Ilustrujmo ovo kroz jedan primer.

Zadatak: Broj podnizova datog zbira

Napisati program koji određuje koliko podnizova (ne obavezno uzastopnih elemenata) datog niza pozitivnih brojeva ima zbir jednak datom broju.

Opis ulaza

Sa standardnog ulaza se učitava broj $1 \leq n \leq 30$, a zatim u narednom redu n pozitivnih realnih brojeva (zaokruženih na dve decimale), razdvojenih razmacima.

Opis izlaza

Na standardni izlaz ispisati traženi broj podnizova (dva realna broja se mogu smatrati jednakima ako se razlikuju za manje od 10^{-5}).

Primer

Ulaz	*Izlaz*	*Objašnjenje*

`4`
`3.2 5.7 9.4 6.9`
`12.6`
	`2`	Važi da je $3,2+9,4=5,7+6,9=12,6$.

Rešenje

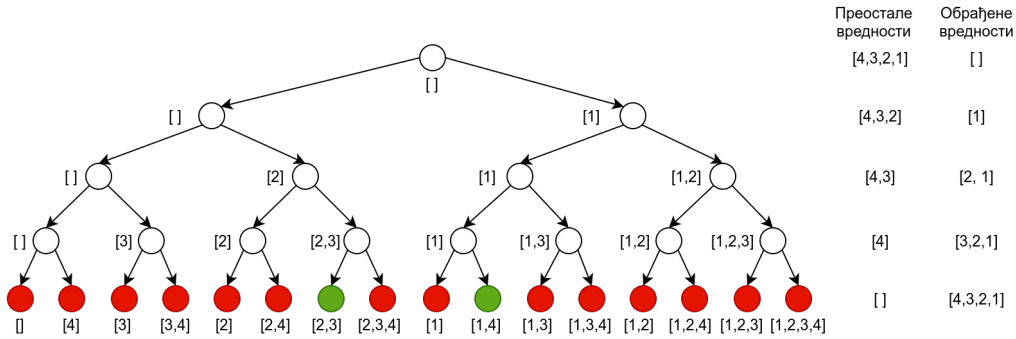
Gruba sila

Jedna mogućnost je da se zadatak reši grubom silom, tj. da se nabroje svi podnizovi i da se za svaki od njih proveri da li mu je zbir jednak traženom. Nabrajanje podnizova se može postići bilo pomoću funkcije koja određuje naredni podniz u leksikografskom redosledu, bilo pomoću rekurzivnog najbrajanja svih mogućnosti. Jedna mogućnost rekurzivne implementacije je zasnovana na tome da svaki neprazni niz razložimo na njegov prefiks bez poslednjeg elementa i na poslednji element i da nezavisno razmatramo mogućnosti kada poslednji element jeste i kada nije uključen u podniz. Čuvaćemo trenutno odabrani podniz obrađenog dela niza na pozicijama $[n, n_0)$, gde je n_0 dužina celog niza, a n parametar koji se smanjuje tokom rekurzije. Zadatak rekurzivne funkcije je da na sve moguće načine dopuni taj podniz elementima sa pozicija $[0, n)$ iz polaznog niza i da vrati broj tako napravljenih podnizova koji imaju dati zbir. U startu je $n = n_0$, a podniz je prazan (to je zaista jedini mogući podniz dela niza na pozicijama $[n_0, n_0)$).

- Ako je $n = 0$, ceo niz je obrađen i podniz ne može više da se dopunjava (jer je skup elemenata na pozicijama $[0, n) = [0, 0)$ prazan). Izračunava se njegov zbir i ako je on jednak ciljnom zbiru, tada funkcija vraća 1 (pronađen je jedan traženi niz koji proširuje trenutni podniz elementima praznog skupa i ima zbir jednak ciljanom), a u suprotnom vraća 0 (ne postoji ni jedan niz koji proširuje trenutni podniz elementima praznog skupa i ima zbir jednak ciljanom).
- Ako je n pozitivno, razmatramo posebno mogućnosti da poslednji element prefiksa $[0, n)$ tj. da element niz_{n-1} bude ili da ne bude uključen u podniz. U oba slučaja rekurzivno pozivamo funkciju za skraćeni prefiks (tj. za vrednost $n - 1$).

Da bismo izbegli realokacije, podniz možemo unapred alocirati na dužinu n_0, ali tada parametar funkcije treba da bude i broj elemenata podniza p.

Drvo rekurzivnih poziva koje nastaje prilikom traženja svih podnizova niza $[4, 3, 2, 1]$, čiji je zbir 5 je prikazano na narednoj slici (jednostavnosti ilustracije radi, pretpostavili smo da su brojevi u nizu i ciljni zbir celi).



Gruba sila – ispitivanje svih podnizova

```
const double EPS = 0.00001;

// funkcija izracunava broj nacina da se dati podniz duzine p
// prosiri elementima niza sa pozicija [0, n) tako da se
// dobije podniz ciji je zbir jednak datom ciljnom zbiru
int brojPodnizovaDatogZbira(const vector<double>& niz, int n,
                            double ciljniZbir,
                            vector<double>& podniz, int p) {
  // u nizu nema preostalih elemenata, pa je trenutni podniz
  // jedini kandidat
  if (n == 0) {
    // racunamo zbir elemenata trenutnog podniza
    double zbirPodniza = 0.0;
    for (int i = 0; i < p; i++)
      zbirPodniza += podniz[i];
    // proveravamo da li je jednak ciljnom zbiru
    if (abs(zbirPodniza - ciljniZbir) < EPS)
      return 1;
    else
      return 0;
  } else {
```

```
    // broj podnizova bez ukljucenog poslednjeg elementa niza
    int broj = 0;
    broj += brojPodnizovaDatogZbira(niz, n-1, ciljniZbir,
                                    podniz, p);
    // broj podnizova sa ukljucenim poslednjim elementom niza
    podniz[p] = niz[n-1];
    broj += brojPodnizovaDatogZbira(niz, n-1, ciljniZbir,
                                    podniz, p+1);
    return broj;
  }
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak
// ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz,
                            double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  vector<double> podniz(n);
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, n, ciljniZbir,
                                  podniz, 0);
}
```

Druga mogućnost implementacije pretrage grubom silom je da kao parametar rekurzivne funkcije prosleđujemo trenutni ciljni zbir tj. razliku između traženog zbira i zbira elemenata trenutno uključenih u podniz obrađenog dela niza. Sam taj podniz nije neophodno održavati. Drugim rečima, zadatak rekurzivne funkcije je da vrati koliko podnizova trenutno neobrađenog dela niza ima zbir jednak datom ciljnom zbiru, pri čemu se taj ciljni zbir sada smanjuje kroz rekurzivne pozive (kada god se uključi neki novi element).

Ilustracije radi, recimo da rekurzivna konstrukcija može biti takva da neprazne nizove razlaže na njihov prvi element i sufiks niza iza tog prvog elementa. To znači da će trenutno obrađeni deo niza biti na pozicijama $[0, k)$, dok će neobrađeni deo niza biti na pozicijama $[k, n)$ gde je k trenutni parametar rekurzije, a n dužina celog niza. Rekurzija počinje kada je $k = 0$, a završava se kada je $k = n$.

Ako je ciljni zbir jednak nuli, to znači da je zbir trenutnog podniza elemenata na

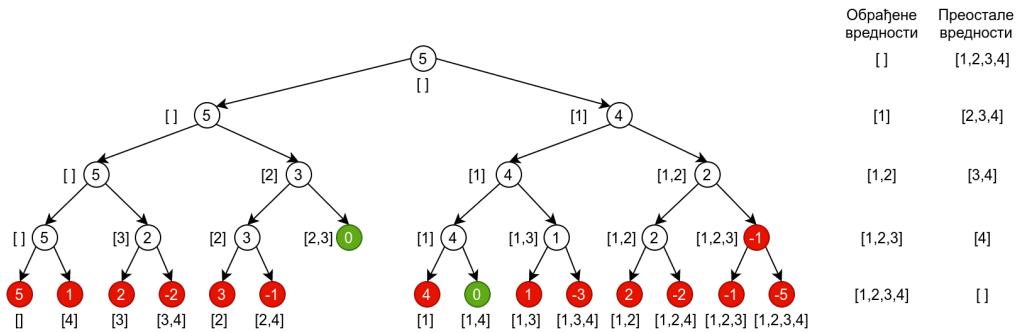
pozicijama $[0, k)$ jednak polaznom traženom zbiru i da smo našli jedan zadovoljavajući podniz. Dodatno, pošto su svi elementi niza pozitivni, podniz se ne može nikako proširiti dodatnim elementima tako da zbir i dalje ostane jednak ciljnom, tako da nije potrebno dalje nastavljati pretragu. Drugim rečima, jedino prazan niz elemenata sa pozicija $[k, n)$ može imati zbir 0, pa funkcija može da vrati rezultat 1 (ona vraća broj nizova).

Ako je ciljni zbir različit od nule, nastavljamo kao i ranije.

Ako je $k = n$, tada u polaznom nizu nema neobrađenih elemenata tj. preostali niz čije podskupove razmatramo je prazan i on ne može sadržati podskup čiji će ciljni zbir biti pozitivan.

Ako je $k < n$, tada razmatramo mogućnost da se element niz_k uključi i mogućnost da se ne uključi u podniz polaznog niza. U prvom slučaju umanjujemo ciljni zbir za vrednost tog elementa (to znači da tražimo broj podnizova elemenata sa pozicija $[k + 1, n)$ koji daju taj umanjeni zbir), a u drugom ciljni zbir ostaje nepromenjen.

Na slici je prikazano drvo rekurzivnih poziva kada se određuje broj podnizova niza $[1, 2, 3, 4]$ čiji je zbir jednak 5.



Gruba sila – preostali ciljni zbir

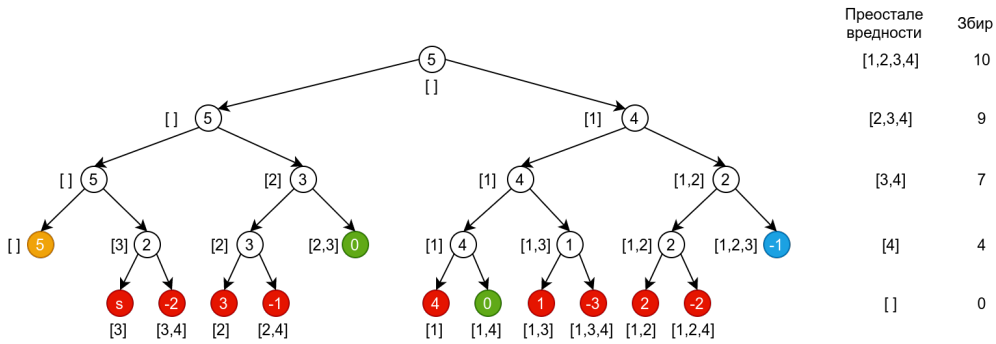
```
// funkcija odredjuje broj podnizova niza odredjenog
// elementima na pozicijama [k, n) takvih da je zbir elemenata
// podniza jednak ciljnom zbiru
int brojPodnizovaDatogZbira(const vector<double>& niz,
                            double ciljniZbir, int k) {
  // jedino prazan niz ima zbir nula
```

```
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je
  // pozitivan
  if (k == niz.size())
    return 0;

  // posebno brojimo podnizove koji ukljucuju niz[k] i one
  // koji ne ukljucuju niz[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir-niz[k], k+1) +
         brojPodnizovaDatogZbira(niz, ciljniZbir, k+1);
}

int brojPodnizovaDatogZbira(const vector<double>& niz,
                            double ciljniZbir) {
  // brojimo podnizove niza odredjenog elementima na
  // pozicijama [0, n)
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, 0);
}
```

Odsecanja

Efikasnija rešenja od rešenja grubom silom se mogu dobiti primenom različitih odsecanja. Jedno važno odsecanje se može sprovesti na osnovu poznavanja intervala u kome mogu ležati zbirovi svih podnizova preostalih elemenata niza. Pošto su svi elementi pozitivni, najmanja moguća vrednost zbira podniza je nula (u slučaju praznog niza), dok je najveća moguća vrednost zbira podniza jednaka zbiru svih elemenata niza. Dakle, ako je ciljni zbir strogo manji od nule ili strogo veći od zbira svih elemenata preostalog dela niza, tada ne postoji ni jedan podniz čiji je zbir jednak ciljnom i moguće je izvršiti odsecanje pretrage. Umesto da zbir svih elemenata preostalog dela niza (dela niza na pozicijama $[k, n)$) računamo iznova u svakom rekurzivnom pozivu, možemo primetiti da se u svakom narednom rekurzivnom pozivu niz samo može smanjiti za jedan element, pa se zbir može računati inkrementalno, umanjivanjem tokom rekurzije zbira polaznog niza za elemente uklonjene iz niza.

Na slici je prikazano drvo rekurzivnih poziva sa ovim odsecanjima kada se u nizu $[1, 2, 3, 4]$ traže podnizovi čiji je zbir jednak 5. Primetimo da je jedno odsecanje izvršeno kada je ciljni zbir bio jednak 5 i kada je u neobrađenom delu niza ostala samo vrednost 4, jer je zbir svih preostalih vrednosti bio manji od ciljnog zbira, a da je drugo odsecanje nastalo jer je nakon uključivanja vrednosti $[1, 2, 3]$ zbir već pretekao vrednost 5 (dobijen je čvor čija je nova ciljna vrednost -1).



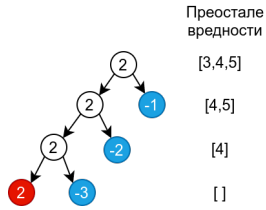
Pretraga uz osnovno odsecanje

```
// racuna se broj podnizova elemenata niza na pozicijama [k,
// n) koji imaju dati zbir, pri cemu se zna da je zbir tih
// elemenata jednak zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz,
                            double ciljniZbir,
                            double zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan
  // element
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je
  // pozitivan
  if (k == niz.size())
    return 0;

  // posto su svi brojevi pozitivni, nije moguce dobiti
  // negativan ciljni zbir
```

```
  if (ciljniZbir + EPS < 0)
    return 0;

  // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog
  // zbira, pa nema podnizova koji bi dali ciljni zbir
  if (zbirPreostalih + EPS < ciljniZbir)
    return 0;

        // broj podnizova u kojima ucestvuje element a[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                        zbirPreostalih - niz[k], k+1) +
        // broj podnizova u kojima ne ucestvuje element a[k]
        brojPodnizovaDatogZbira(niz, ciljniZbir,
                        zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak
// ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz,
                            double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  // izracunavamo zbir elemenata niza
  double zbirNiza = 0;
  for (int i = 0; i < n; i++)
    zbirNiza += niz[i];
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}
```

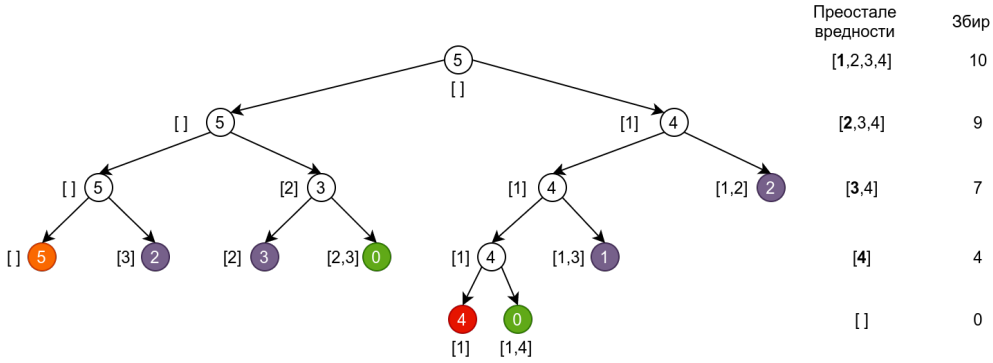
Još jedno moguće odsecanje se može izvršiti kada se ustanovi da je najmanji od preostalih brojeva u nizu veći od ciljnog zbira. Ako je taj ciljni zbir pozitivan, tada nije moguće dostići ga (jer prazan podniz ima zbir nula, a bilo koji neprazan podniz ima zbir veći ili jednak od tog minimalnog elementa). Minimalni element preostalog dela niza je jednostavno odrediti ako se niz sortira (što možemo uraditi pre početka pretrage). Naglasimo da je ova odsecanje samo mala optimizacija odsecanja čvorova koji imaju negativan ciljni zbir. Na primer, ako bismo imali ciljni

zbir 2 i preostale elemente 3, 4, 5 i 6, pomoću ove optimizacije bismo odmah mogli prekinuti pretragu, dok bi se bez nje dobilo drvo pretrate prikazano na narednoj slici. Dakle, drveta koja se odsecaju ovom optimizacijom, a ne bi bila odsečena bez nje, su samo linearne (a ne eksponencijalne) veličine u odnosu na broj preostalih elemenata niza.



Doprinos odsecanja na osnovu vrednosti najmanjeg elementa

Na slici je prikazano drvo rekurzivnih poziva sa ovim odsecanjima kada se u nizu $[1, 2, 3, 4]$ traže podnizovi čiji je zbir jednak 5. Kada su odabrani elementi $[1, 2]$, tada je ciljni zbir 2, pa pošto je najmanji preostali element 3, može da se izvrši odsecanje. Slično se događa i kada su odabrani elementi $[3]$ (ciljni zbir je 2), $[2]$ (ciljni zbir je 3) i $[1, 3]$ (ciljni zbir je 1), i kada je najmanji (zapravo jedini) preostali element jednak 4. Odsecanje nastupa i kada nije izabran nijedan element (ciljni zbir je 5), a jedini preostali element je 4 (tada je ciljni zbir veći od zbira svih preostalih elemenata).



Pretraga sa odsecanjem

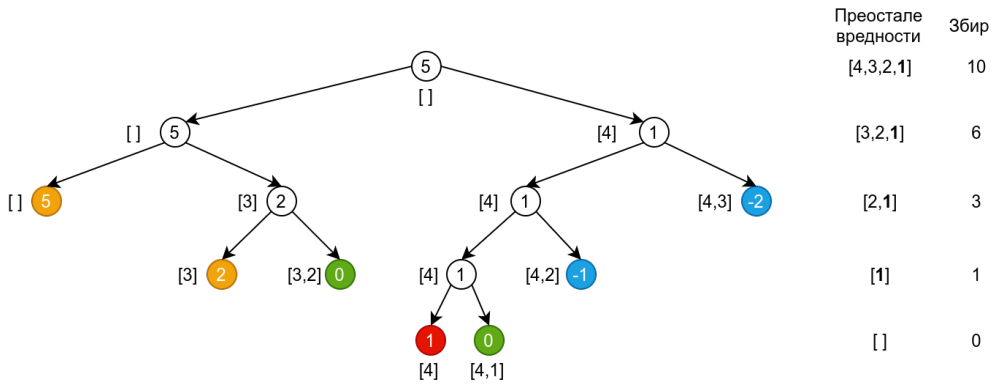
```
// racuna se broj podnizova elemenata niza na pozicijama [k,
// n) koji imaju dati zbir, pri cemu se zna da je zbir tih
// elemenata jednak zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz,
                            double ciljniZbir,
                            double zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan
  // element
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je
  // pozitivan
  if (k == niz.size())
    return 0;

  // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog
  // zbira, pa nema podnizova koji bi dali ciljni zbir
  if (zbirPreostalih + EPS < ciljniZbir)
    return 0;

  // vec uzimanje najmanjeg elementa prevazilazi ciljni zbir,
  // pa nema podnizova koji bi dali ciljni zbir
  if (niz[k] > ciljniZbir + EPS)
    return 0;

        // broj podnizova u kojima ucestvuje element a[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                            zbirPreostalih - niz[k], k+1) +
        // broj podnizova u kojima ne ucestvuje element a[k]
        brojPodnizovaDatogZbira(niz, ciljniZbir,
                            zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak
// ciljnom
```

```
int brojPodnizovaDatogZbira(vector<double>& niz,
                            double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  // sortiramo elemente niza neopadajuce
  sort(begin(niz), end(niz));
  // izracunavamo zbir elemenata niza
  double zbirNiza = 0;
  for (int i = 0; i < n; i++)
    zbirNiza += niz[i];
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}
```

Kada se vrše odsecanja, redosled obilaska vrednosti u drvetu pretrage može značajno da utiče na veličinu drveta. Na primer, umesto da se prvo razmatra uključivanje najmanjeg elementa niza u podniz, može se prvo razmatrati uključivanje najvećeg elementa uz podniz (uz ista odsecanja koja su ranije opisana). Na početku je niz potrebno sortirati nerastuće (umesto neopadajuće), a najmanji element neobrađenog dela niza se uvek nalazi na kraju samog niza.

Na slici je prikazano drvo rekurzivnih poziva sa ovim odsecanjima kada se u nizu $[1, 2, 3, 4]$ traže podnizovi čiji je zbir jednak 5. Primetimo da se u ovom primeru većina odsecanja vrši već na osnovu toga da li preostali ciljni zbir pripada intervalu od 0 do zbira svih preostalih elemenata, tako da ova strategija grananja otvara mnogo više mogućnosti za ta osnovna odsecanja (u ovom malom primeru se čak nijednom nije desilo da je nastupilo odsecanje na osnovu vrednosti minimalnog elementa u preostalom delu niza).

```
// racuna se broj podnizova elemenata niza na pozicijama [k,
// n) koji imaju dati zbir, pri cemu se zna da je zbir tih
// elemenata jednak zbirPreostalih
int brojPodnizovaDatogZbira(const vector<double>& niz,
                            double ciljniZbir,
                            double zbirPreostalih, int k) {
  // ciljni zbir 0 se dobija samo ako se ne uzme ni jedan
  // element
```



Pretraga sa odsecanjem

```
  if (abs(ciljniZbir) < EPS)
    return 1;

  // jedini podniz praznog niza je prazan, a ciljni zbir je
  // pozitivan
  if (k == niz.size())
    return 0;

  // cak ni uzimanje svih elemenata ne moze dovesti do ciljnog
  // zbira, pa nema podnizova koji bi dali ciljni zbir
  if (zbirPreostalih + EPS < ciljniZbir)
    return 0;

  // vec uzimanje najmanjeg elementa prevazilazi ciljni zbir,
  // pa nema podnizova koji bi dali ciljni zbir
  if (niz[niz.size()-1] > ciljniZbir + EPS)
    return 0;

        // broj podnizova u kojima ucestvuje element a[k]
  return brojPodnizovaDatogZbira(niz, ciljniZbir - niz[k],
                                 zbirPreostalih - niz[k], k+1) +
        // broj podnizova u kojima ne ucestvuje element a[k]
        brojPodnizovaDatogZbira(niz, ciljniZbir,
```

```
                                zbirPreostalih - niz[k], k+1);
}

// funkcija racuna koliko podnizova datog niza ima zbir jednak
// ciljnom
int brojPodnizovaDatogZbira(vector<double>& niz,
                            double ciljniZbir) {
  // broj elemenata niza
  int n = niz.size();
  // sortiramo elemente niza nerastuce
  sort(begin(niz), end(niz), greater<int>());
  // izracunavamo zbir elemenata niza
  double zbirNiza = 0;
  for (int i = 0; i < n; i++)
    zbirNiza += niz[i];
  // rekurzivnom funkcijom racunamo trazeni broj podnizova
  return  brojPodnizovaDatogZbira(niz, ciljniZbir, zbirNiza, 0);
}
```

7.2 Grananje sa ograničavanjem

Tehnika *grananja sa ograničavanjem* (engl. branch and bound) je metoda za rešavanje optimizacionih problema koja sistematski istražuje prostor mogućih rešenja grananjem (engl. branch), odnosno deljenjem problema na manje potprobleme. Istovremeno, koristi se i ograničenje (engl. bound) – procena najboljeg mogućeg rešenja u datom potprostoru – kako bi se eliminisali potprostori koji ne mogu dati bolje rešenje od već pronađenih. Na taj način se izbegava nepotrebno pretraživanje i ubrzava pronalaženje optimalnog rešenja. Dakle, već pronađene vrednosti rešenja u jednom delu prostora pretrage se koriste kao ograničenja u narednim delovima prostora pretrage. Ilustrujmo ovu tehniku na problemu pronalaženja najmanjeg broja boja potrebnih da se oboji graf tako da su svi susedni čvorovi obojeni različitim bojama.

Zadatak: K bojenje

Potrebno je rasporediti promenljive u registre procesora. Pri tom, je poznato da neke promenljive ne smeju da budu smeštene u isti registar (jer se koriste istovremeno). Napisati program koji određuje najmanji broj registara potreban da se smeste sve promenljive. Na primer, u kodu

```
x = a + b;
y = x * b;
return y;
```

Promenljive `a` i `b` kao i promenljive `x` i `b` ne smeju biti smeštene u isti registar. Moguće je upotrebiti samo dva registra. Prvo se u registar 1 smešta promenljiva `a`, a u registar 2 promenljiva `b`. Nakon toga se vrednost promenljive `x` smešta u registar 1 (jer vrednost promenljive `a` nije nadalje potrebna). Nakon toga se vrednost promenljive `y` može smestiti bilo u registar 1, bilo u registar 2, jer nadalje nisu potrebni ni vrednost promenljive `x` ni vrednost promenljive `b`.

Opis ulaza

Sa standardnog ulaza se unosi broj promenljivih n ($1 \leq n \leq 50$). Nakon toga se do kraja ulaza unose parovi promenljivih koje ne smeju da se smeste u iste registre (promenljive se broje od 0 do $n - 1$).

Opis izlaza

Na standardni izlaz ispisati najmanji broj registara potrebnih da se sve promenljive smeste.

Primer

Ulaz	*Izlaz*
6 0 1 0 2 1 2 1 4 1 5 2 3 3 4 3 5 4 5	3

Rešenje

Promenljive možemo predstaviti čvorovima grafa tako da grane postoje između promenljivih koje ne smeju biti obojene istom bojom, dok registre možemo predstaviti bojama čvorova grafa. Time se polazni problem svodi na određivanje najmanjeg broja boja potrebnog da se oboje čvorovi grafa tako da nikoja dva susedna čvora nisu obojena istom bojom. Jednostavnosti radi pretpostavićemo da je graf sačinjen od promenljivih i od ograničenja između njih povezan.

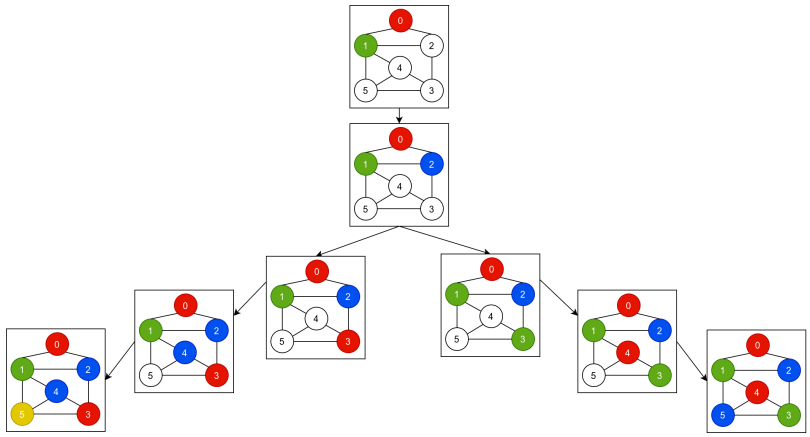
Zadatak je moguće rešiti korišćenjem funkcije koja proverava da li se bojenje može izvršiti pomoću k boja i primenom binarne pretrage za određivanje prelomne tačke, tj. najmanje vrednosti k takve da se graf može obojiti sa k boja. Provera da li se graf može obojiti sa k boja može se izvršiti bektreking pretragom.

Zadatak se može rešiti i samo jednom bektreking pretragom. Za razliku od provere da li se graf može obojiti sa k boja, koja se može zaustaviti čim se naiđe na prvo uspešno bojenje, u ovom algoritmu (za određivanje najmanjeg dovoljnog broja boja) potrebno je obići celo stablo pretrage. Kada se pronađe neko uspešno bojenje sa određenim brojem boja (recimo k), prilikom povratka u pretrazi vrši se ograničavanje boja čvorova na vrednosti od 1 do $k-1$ (jer nam je cilj da pokušamo da pronađemo bojenje sa manjim brojem boja od k). Početno ogrančenje broja boja je n, jer smo sigurni da se graf sa n čvorova može obojiti pomoću n boja.

Jednostavnosti radi trenutnu vrednost optimuma čuvamo u globalnoj promenljivoj.

Primer 7.2.1. *Na slici je prikazano bojenje grafa sa najmanjim brojem boja. Pretpostavimo da su boje obeležene brojevima od 1 do n. Bez gubitka na opštosti se može pretpostaviti da se čvor 0 može obojiti u boju broj 1 (crvenu), a njegov prvi sused, čvor 1, u boju broj 2 (zelenu). Za čvor 2 su nam inicijalno na raspolaganju boje od 1 do 6 (jer graf ima 6 čvorova). Bojenje u boje 1 i 2 nije moguće, pa se čvor 2 boji u boju broj 3 (plavu). Za čvor 3 i dalje su moguće sve boje od 1 do 6, pa se on boji u prvu raspoloživu boju tj. boju 1 (crvenu). Za čvor 4 su raspoložive boje od 1 do 6, pa sa i on boji u prvu raspoloživu boju tj. boju 3 (plavu). Na kraju, i za čvor 5 su raspoložive boje od 1 do 6 i on se boji u prvu raspoloživu boju tj. boju 4. U tom trenutku smo pronašli bojenje sa 4 boje i pošto u nastavku pretrage tražimo samo bolja rešenja, raspoložive će biti samo prve tri boje (1, 2 i 3). Vraćamo se na čvor 4 i njegovu boju ne možemo da promenimo (jer već ima najveću raspoloživu boju). Zato se vraćamo na čvor 3. Njegovu boju možemo da promenimo u boju 2 (zelenu). Tada čvor 4 bojimo u prvu raspoloživu boju 1 (crvenu), a čvor 5 u boju*

3 (plavu), jer boje 1 i 2 nisu moguće, a na raspolaganju su nam boje od 1 do 3. U tom trenutku je pronađeno uspešno bojenje sa 3 boje i razmatramo samo bojenja sa 2 boje (kandidati za boje su sada samo 1 i 2). Pošto su na putu do čvora 4 već upotrebljene tri boje, vraćamo se unazad, bez razmatranja daljih mogućnosti promene boje konkretnog čvora sve do čvora 2. Pošto čvor 2 ne može da promeni boju (jer je već obojen u boju 3, a na raspolaganju su nam sada samo boje 1 i 2), vraćamo se na čvorove 1 i 0 koji imaju fiksirane boje i pretraga se završava.



Primer bojenja grafa sa minimalnim brojem boja

```
// poznato je da se graf moze obojiti sa ovim brojem boja
// jednostavnosti radi koristimo globalnu promenljivu (mada to
// nije neohpodno)
int brojBoja;

// funkcija pokusava da prosiri bojenje graf dato nizom boje,
// u kome je trenutno upotrebljen broj boja dat promenljivom
// upotrebljenoBoja, tako sto boji dati cvor, koristeci
// raspolozive boje, koje su odredjene globalnom promenljivom
// brojBoja
void oboj(const vector<vector<int>>& susedi,
          int cvor, vector<int>& boje, int upotrebljenoBoja) {
  // ako su svi cvorovi obojeni, bojenje je uspesno
  int brojCvorova = susedi.size();
```

```
  if (cvor >= brojCvorova) {
    brojBoja = upotrebljenoBoja;
    return;
  }

  // ako je cvor vec obojen, preskacemo ga
  if (boje[cvor] != 0) {
    oboj(susedi, cvor + 1, boje, upotrebljenoBoja);
    return;
  }

  // pokusavamo da cvoru dodelimo svaku od raspolozivih boja
  for (int boja = 1; boja < brojBoja; boja++) {
    // linearnom pretragom proveravamo da li je moguce obojiti
    // cvor u tekucu boju
    bool mozeBoja = true;
    // proveravamo sve susede
    for (int sused : susedi[cvor])
      // ako je neki od njih vec obojen u tekucu boju
      if (boje[sused] == boja) {
        // bojenje nije moguce
        mozeBoja = false;
        break;
      }
    if (mozeBoja) {
      // bojimo tekuci cvor
      boje[cvor] = boja;
      // pokusavamo rekurzivno bojenje narednog cvora i ako
      // uspemo tada je bojenje moguce
      oboj(susedi, cvor+1, boje, max(upotrebljenoBoja, boja));
      boje[cvor] = 0;

      // ako smo vec upotrebili previse boja, mozemo odmah
      // preseci pretragu
      if (upotrebljenoBoja >= brojBoja)
        return;
```

```
    }
  }
}

// najmanji broj boja kojima se mogu obojiti cvorovi grafa,
// tako da nikoja dva cvora nisu susedna
int minBrojBoja(const vector<vector<int>>& susedi) {
  // broj cvorova grafa
  int n = susedi.size();
  // boja svakog cvora
  vector<int> boje(n, 0);
  // graf se sigurno moze obojiti sa n boja
  brojBoja = n;
  // prva cvor 0 i njegov prvi sused se boje u boje 0 i 1
  boje[0] = 1; boje[susedi[0][0]] = 2;
  // krecemo bojenje od cvora 0
  oboj(susedi, 0, boje, 2);
  return brojBoja;
}
```

8. Dinamičko programiranje

U poglavlju 4.8 videli smo da je jedna od najopasnijih odlika rekurzivno implementiranih funkcija neefikasnost koja nastaje usled rekurzivnih poziva ponovljenih za iste vrednosti argumenata (engl. overlapping recursive calls). Tipičan primer je izračunavanje elemenata Fibonačijevog niza, gde naivna rekurzivna implementacija dovodi do eksponencijalne složenosti. U ovom poglavlju prikazaćemo tehniku *dinamičkog programiranja*, koja predstavlja sistematičan način da se reši problem ponovljenih rekurzivnih poziva.

8.1 Fibonačijev niz

Naredna naivna implementacija funkcije koja izračunava n-ti element Fibonačijevog niza zanemaruje činjenicu da je prilikom izračunavanja vrednosti `fib(n-1)` već izračunala vrednost `fib(n-2)`, pa tu vrednost izračunava iznova, što dovodi do ogromne neefikasnosti.

```
int fib(int n)
{
  if (n == 0 || n == 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

Kako bi se izbegla suvišna izračunavanja moguće je koristiti tehniku *memoizacije*, koja podrazumeva da se u posebnoj strukturi podataka čuvaju svi rezultati već zavr-

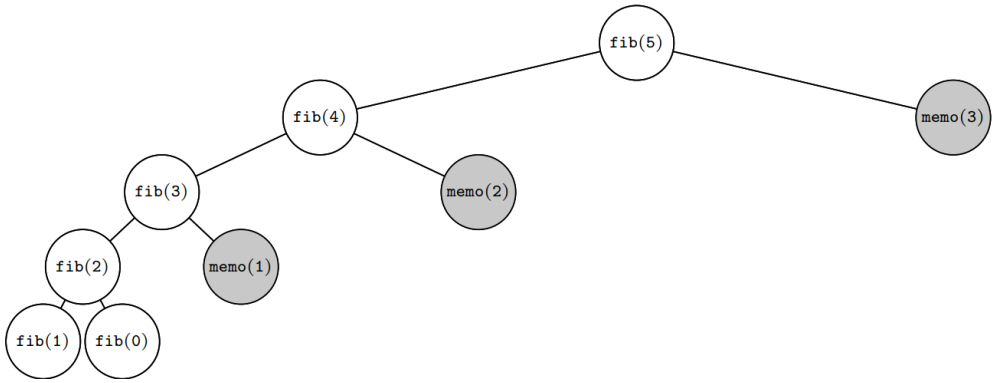
šenih rekurzivnih poziva. Pri svakom ulasku u funkciju konsultuje se ova struktura i, ako je rezultat već poznat, vraća se prethodno izračunat rezultat.

```
vector<int> memo;

int fib_(int n)
{
  if (memo[n] != 0) return memo[n];
  if (n == 0 || n == 1)
    return memo[n] = n;
  else
     return memo[n] = fib_(n-1) + fib_(n-2);
}

int fib(int n) {
  memo.resize(n+1, 0);
  return fib(n);
}
```

U ovoj varijanti funkcije, poziv za svaku vrednost parametra n izvršava se tačno dva puta (osim za vrednosti 0, n i $n-1$, za koje se poziv izvršava samo jednom), pa je složenost izračunavanja linearna tj. $O(n)$. Na primer, tokom izračunavanja vrednosti `fib(n-1)` biće izračunata i memoizovana vrednost `fib(n-2)`, pa će poziv `fib(n-2)` samo pročitati i vratiti ranije izračunatu vrednost. Stablo rekurzivnih poziva memoizovane funkcije za $n = 5$ prikazano je na slici 8.1.

Globalni niz možemo izbeći uvođenjem dodatnog parametra. Pošto vremenska složenost izračunavanja vrednosti u slučaju izlaza iz rekurzije pripada klasi $O(1)$, složenost se ne menja ni ako se preskoči memoizacija za $n = 0$ i $n = 1$.

```
int fib(int n, vector<int>& memo)
{
  if (n == 0 || n == 1)
    return n;
  if (memo[n])
    return memo[n];
  else
```



Slika 8.1: Ilustracija izračunavanja n-og elementa Fibonačijevog niza primenom memoizacije – pozivi u kojima se samo čita ranije izračunata vrednost označeni su sivom bojom

```
    return memo[n] = fib(n-1, memo) + fib(n-2, memo);
}

int fib(int n)
{
  vector<int> memo(n+1, 0);
  return fib(n, memo);
}
```

Drugi pristup rešavanja problema suvišnih izračunavanja naziva se *dinamičko programiranje*[1], koje podrazumeva da se rekurzija eliminiše i zameni induktivnom konstrukcijom. Prilikom rekurzivnog rešavanja vrši se poziv funkcije koja treba da izračuna rešenje polaznog problema, a zatim se taj problem svodi na jednostavnije potprobleme koji se zatim rešavaju novim pozivima iste funkcije. Potproblemi se rešavaju samo kada je to potrebno, tj. samo kada funkcija izvrši rekurzivni poziv. Kod dinamičkog programiranja, obično se unapred vrši rešavanje svih potproblema manje dimenzije, bez provere da li će njihovo rešavanje zaista biti neophodno (u

[1] U nekim tekstovima memoizacija i dinamičko programiranje razmatraju se kao dva oblika jedne iste tehnike (jer se u oba slučaja uvodi pomoćni niz u kojem se pamte rezultati rekurzivnih poziva). Tada se memoizacija naziva dinamičko programiranje naniže, a (klasično) dinamičko programiranje naziva dinamičko programiranje naviše.

praksi se često pokazuje da jeste). Na osnovu rešenih potproblema manje dimenzije, kreira se rešenje problema veće dimenzije, sve dok se ne kreira i rešenje glavnog problema. Implementacije rešenja dinamičkim programiranjem ne uključuju rekurzivne funkcije (mada je veza između rešenja problema veće i potproblema manje dimenzije i dalje suštinski rekurentna). Na primer, gore navedena funkcija `fib` može se zameniti iterativnom funkcijom koja od početnih elemenata niza postepeno kreira dalje elemente niza. Iako je potrebno izračunati vrednost funkcije `fib` samo za parametar n, izračunavaju se vrednosti funkcije `fib` za sve parametre manje od ili jednake n.

```
int fib(int n)
{
  if (n == 0 || n == 1)
    return n;
  vector<int> f(n+1, 0);
  f[0] = 0; f[1] = 1;
  for (int i = 2; i <= n; i++)
    f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

Primetimo da za izračunavanje n-tog elementa niza nije neophodno pamtiti sve elemente niza do indeksa n već samo dva prethodna, pa se funkcija može implementirati i jednostavnije:

```
int fib(int n)
{
  unsigned i, fpp, fp, f;
  if (n == 0 || n == 1)
    return n;
  fpp = 0;
  fp = 1;
  for (i = 2; i <= n; i++) {
    f = fpp + fp;
    fpp = fp;
    fp = f;
```

```
    }
    return f;
}
```

U narednim primerima ćemo videti da se isti niz koraka koji je sproveden u ovom primeru (memoizacija ili dinamičko programiranje naviše, praćeno memorijskom optimizacijom) može sprovesti i na drugim problemima kod kojih dolazi do ponovljenih rekurzivnih poziva.

8.2 Brojanje kombinatornih objekata

Čest zadatak je da se prebroje kombinatorni objekti tj. da se odredi koliko postoji objekata (nizova, lista, drveta i slično) koji zadovoljavaju neko zadato svojstvo. U ovakvim problemima često dolazi do ponavljanja rekurzivnih poziva i oni se zato često rešavaju tehnikom dinamičkog programiranja.

Zadatak: Broj kombinacija

Napisati program koji određuje broj kombinacija bez ponavljanja dužine k iz skupa od k elemenata (tj. broj različitih kombinacija u igri loto ako se iz bubnja koji sadrži n loptica izvlači njih k).

Opis ulaza

Sa standardnog ulaza se zadaje broj k ($1 \leq k \leq n$) i broj n ($1 \leq n \leq 40$).

Opis izlaza

Na standardni izlaz ispisati traženi broj kombinacija.

Primer 1

Ulaz	*Izlaz*	*Objašnjenje*
`3` `5`	`10`	To su kombinacije $(1,2,3)$, $(1,2,4)$, $(1,2,5)$ $(1,3,4)$, $(1,3,5)$, $(1,4,5)$, $(2,3,4)$, $(2,3,5)$, $(2,4,5)$ i $(3,4,5)$.

Primer 2

Ulaz	*Izlaz*
`7` `39`	`15380937`

Rešenje

Iako postoje razni načini da se do rešenja ovog zadatka dođe, prikazaćemo tehniku zasnovanu na tome da program koji nabraja sve kombinatorne objekte malo po malo transformišemo do efikasnog programa koji ih broji. Ova tehnika nije specifična za kombinacije bez ponavljanja i može se primeniti na brojanje bilo koje vrste kombinatornih objekata koje nabrajamo rekurzivnom funkcijom.

Broj kombinacija jednak je binomnom koeficijentu

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}.$$

Međutim, izračunavanje na osnovu direktne primene ove formule bi veoma brzo dovelo do prekoračenja (usled veoma brzog rasta faktorijelske funkcije). Malo bolja situacija je da se izračuna

$$\binom{n}{k} = \frac{n(n-1)\ldots(n-k+1)}{k!},$$

no ni to u potpunosti ne uklanja problem prekoračenja, jer imenilac može biti preveliki.

Rekurzivna funkcija koja izračunava broj kombinacija

Možemo krenuti od ranije opisane procedure za generisanje svih kombinacija u kojoj se održava interval $[n_{min}, n_{max}]$ iz kojeg se mogu uzeti vrednosti kojima se proširuje započeta kombinacija i u kojoj se kroz dva rekurzivna poziva razmatra mogućnost da se vrednost n_{min} uvrsti u kombinaciju i mogućnost da se ona ne uvrsti u kombinaciju.

Umesto procedure koja ispisuje kombinacije, definišemo fukciju koja vraća broj kombinacija. Pošto nam je bitan samo broj kombinacija, a ne i same kombinacije, možemo u potpunosti izbaciti niz koji se popunjava i umesto njega prosleđivati samo njegovu dužinu k.

```
long long brojKombinacija(int i, int k,
                          int n_min, int n_max) {
  // ako je popunjen ceo niz postoji jedna kombinacija
```

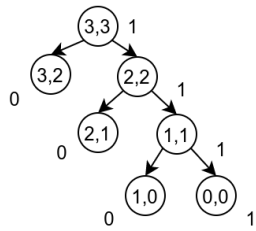
```
  if (i == k) return 1;
  // ako niz nije moguće popuniti do kraja,
  // tada nema kombinacija
  if (k - i > n_max - n_min + 1)
    return 0;
  // broj kombinacija je jednak zbiru
  // broja kombinacija koje sadrže vrednost n_min i
  // broja kombinacija koje ne sadrže vrednost n_min
  return brojKombinacija(i+1, k, n_min+1, n_max) +
         brojKombinacija(i, k, n_min+1, n_max);
}

long long brojKombinacija(int K, int N) {
  return brojKombinacija(0, K, 1, N);
}
```

Možemo primetiti da nam konkretne vrednosti k i i nisu bitne, već je bitan samo broj elemenata u intervalu $[i, k)$ tj. razlika $k - i$. Slično, nisu nam bitne ni konkretne vrednosti n_{max} i n_{min} već samo broj elemenata u segmentu $[n_{min}, n_{max}]$ tj. vrednost $n_{max} - n_{max} + 1$. Ako te dve veličine zamenimo sa k tj. n dobijamo narednu definiciju.

```
long long brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako niz nije moguće popuniti do kraja, tada nema kombinacija
  if (k > n) return 0;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}
```

Ako funkciju pozovemo za vrednosti $k \leq n$, slučaj $k > n$ može nastupiti jedino iz drugog rekurzivnog poziva za $k = n$ (jer odnos između k i n u prvom rekurzivnom pozivu ostaje nepromenjen, a u drugom se menja samo za 1). Međutim, kako je ilustrovano na narednoj slici, u slučaju poziva funkcije za $k = n$ dobiće se uvek povratna vrednost 1 (jedan rekurzivni poziv će uvek vraćati nulu, a drugi će prouzrokovati smanjivanje oba argumenta sve dok se ne dođe do $k = n = 0$, kada

će se 1 vratiti na osnovu prvog izlaza iz rekurzije), što je sasvim u skladu sa tim da tada postoji samo jedna kombinacija.



Izračunavanje koeficijenata za $k = n$

Na osnovu ovoga iz rekurzije možemo izaći za $k = n$ vrativši vrednost 1, čime onda eliminišemo potrebu za proverom da li je $k > n$ (naravno, pod pretpostavkom da ćemo funkciju pozivati samo za $k \leq n$).

Primećujemo da smo ovom transformacijom dobili dobro poznate osobine binomnih koeficijenata.

$$\binom{n}{0} = 1, \qquad \binom{n}{n} = 1, \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

One čine osnovu Paskalovog trougla u kom se nalaze binomni koeficijenti (u narednoj tabeli koeficijent $\binom{n}{k}$ je napisan u vrsti n i koloni k, da bi se prikazao uobičajeni oblik Paskalovog trougla, koji se popunjava vrstu po vrstu).

```
1                        (0,0)
1  1                     (1,0) (1,1)
1  2  1                  (2,0) (2,1) (2,2)
1  3  3  1               (3,0) (3,1) (3,2) (3,3)
1  4  6  4  1            (4,0) (4,1) (4,2) (4,3) (4,4)
1  5 10 10  5  1         (5,0) (5,1) (5,2) (5,3) (5,4) (5,5)
1  6 15 20 15  6  1      (6,0) (6,1) (6,2) (6,3) (6,4) (6,5) (6,6)
```

Prva veza govori da su elementi prve kolone uvek jednaki 1, druga da su na kraju svake vrste elementi takođe jednaki 1, a treća da je svaki element u trouglu jednak zbiru elementa neposredno iznad njega i elementa neposredno ispred tog.

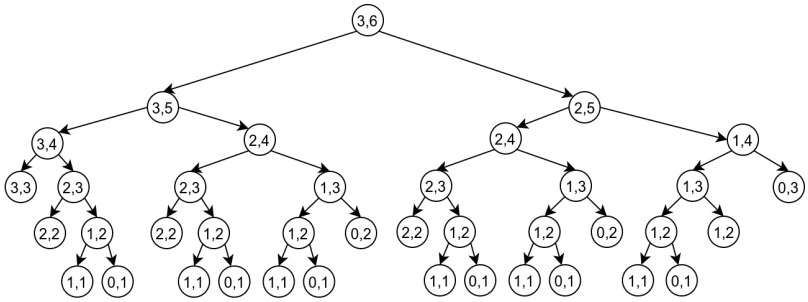
Naravno, do ovih formula i do rekurzivne definicije smo mogli doći i direktno, razmatranjem definicija kombinacija, na primer, u modelu gde se k kuglica bez

vraćanja izvlače iz bubnja u kom se nalazi n različitih kuglica. Postoji jedinstven način da se iz bubnja ne izvuče ni jedna kuglica. Takođe, postoji jedinstven način da se iz bubnja izvuče svih n kuglica. U suprotnom (ako je $0 < k < n$), tada sve načine razdvajamo na one u kojima jeste i na one u kojima nije izvučena prva kuglica (kuglica sa najmanjim brojem). Ako ona jeste izvučena, preostalo je da se izvuče još $k - 1$ kuglica iz bubnja u kom se nalazi $n - 1$ kuglica, a ako nije, tada je preostalo da se izvuče još k kuglica iz bubnja u kom se nalazi još $n - 1$ kuglica (pošto smo pretpostavili da u drugoj grupi načina kuglica sa najmanjim brojem neće biti među izvučenim kuglicama, možemo je odmah izbaciti iz bubnja i skloniti negde sa strane).

```
long long brojKombinacija(int k, int n) {
  // ako je popunjen ceo niz postoji jedna kombinacija
  if (k == 0) return 1;
  // ako treba popuniti još tačno n elemenata, tada postoji
  // tačno jedna kombinacija
  if (k == n) return 1;
  // broj kombinacija je jednak zbiru kombinacija u dva slučaja
  return brojKombinacija(k-1, n-1) + brojKombinacija(k, n-1);
}
```

Vreme koje se utroši u svakom rekurzivnom pozivu (ne računajući rekurzivne pozive koji se iz njega iz pozivaju) je očigledno $O(1)$. Jednačina kojom se opisuje vreme rada funkcije je $T(k, n) = T(k - 1, n - 1) + T(k, n - 1) + O(1)$, $T(0, n) = T(n, n) = O(1)$, i vreme izvršavanja je $T(k, n) = O(\binom{n}{k})$. Za fiksirano k, ovo je složenost opisana polinomom promenljive n, koji može biti veoma visokog stepena (k), dok za fiksirano n ovaj broj raste eksponencijalno sa porastom k. U svakom slučaju, jasno je da je složenost izuzetno visoka i da je ovaj program praktično neupotrebljiv za veće vrednosti n i k.

Razlog ovoj neefikasnosti su ponovljeni rekurzivni pozivi, što se može videti na slici. Svakom listu drveta odgovara tačno jedna kombinacija, pa pošto ukupno kombinacija ima $\binom{n}{k}$, ukupan broj rekurzivnih poziva je ograničen sa $2\binom{n}{k}$ (jer u binarnom drvetu ne može biti više nego duplo više čvorova nego listova). U svakom pozivu se vrši $O(1)$ operacija, pa je složenost $O(\binom{n}{k})$.



Memoizacija

Iako korektna, gornja funkcija je neefikasna i može se popraviti tehnikom dinamičkog programiranja. Najjednostavnije prilagođavanje je da se upotrebi memoizacija. Pošto funkcija ima dva parametra, za memoizaciju ćemo upotrebiti matricu. Ako se $\binom{n}{k}$ pamti u matrici na poziciji (n, k), matricu možemo alocirati na $n + 1$ vrsta, gde poslednja vrsta ima $n + 1$ elemenata, a svaka prethodna jedan element manje (u matricu će se popunjavati elementi Paskalovog trougla).

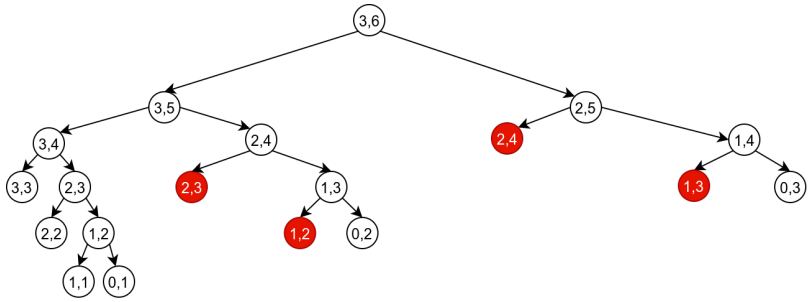
Pošto nas neće zanimati vrednosti veće od polaznog k i pošto se i k i n smanjuju tokom rekurzije, pri čemu je $k \leq n$, možemo i odseći deo trougla desno od pozicije k.

Pošto su brojevi kombinacija uvek veći od nule, vrednosti 0 u matrici će nam označavati da poziv za te parametre još nije izvršen.

I memorijska i vremenska složenost memoizovane verzije je $O(nk)$. Naime, u drvetu rekurzivnih poziva se svaki čvor može javiti najviše dva puta. Pošto svaki čvor sadrži par brojeva takvih da je prvi od 0 do k, a drugi od 0 do n, ukupan broj čvorova je odozgo ograničen sa $2nk$ (a može biti i manji, jer se neki parovi ne mogu javiti).

Ako je poznato gornje ograničenje na vrednosti n i k tada umesto matrice koja se dinamički alocira možemo upotrebiti statički alociranu matricu, čime se izbegava nepotrebno trošenje vremena na dinamičku alokaciju, po cenu da program i za manje vrednosti n i k zauzima veliku količinu memorije. U tom slučaju nije moguće alocirati samo trougaoni deo matrice, već ceo, pravougaoni blok memorije.

```
long long brojKombinacija(int k, int n,
                          vector<vector<long long>>& memo) {
  // ako smo već računali broj kombinacija, ne računamo ga
```



Drvo rekurzivnih poziva uz memoizaciju

```
  // ponovo
  if (memo[n][k] != 0) return memo[n][k];

  // broj kombinacija na početku i na kraju svake vrste jednak
  // je 1
  if (k == 0 || k == n) return memo[n][k] = 1;
  // broj kombinacija u sredini jednak je zbiru broja
  // kombinacija iznad i iznad levo od tekućeg elementa
  return memo[n][k] = brojKombinacija(k-1, n-1, memo) +
                      brojKombinacija(k, n-1, memo);
}

long long brojKombinacija(int K, int N) {
  // alociramo prostor za rezultate rekurzivnih poziva koji se
  // mogu desiti i popunjavamo matricu nulama
  vector<vector<long long>> memo(N+1);
  for (int n = 0; n <= N; n++)
    memo[n].resize(min(K+1, n+1), 0);
  // pozivamo funkciju koja će izračunati traženi broj
  return brojKombinacija(K, N, memo);
}
```

Dinamičko programiranje naviše

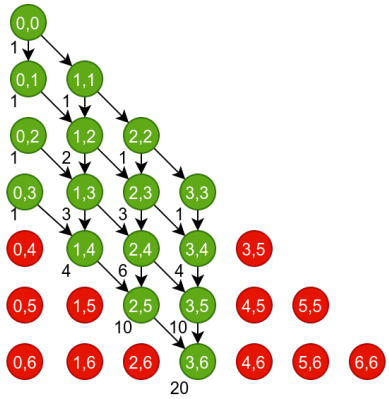
Umesto memoizacije možemo upotrebiti i dinamičko programiranje naviše, oslobditi se rekurzije i popuniti trougao vrstu po vrstu naniže. Popunjavanje celog

trougla je prilično jednostavno.

```
long long brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje celog trougla
  vector<vector<long long>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(n+1);
  // obrađujemo vrstu po vrstu
  for (int n = 0; n <= N; n++) {
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata
    // iznad njih
    for (int k = 1; k < n; k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // na kraju svake vrste nalazi se 1
    dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

I u ovom slučaju možemo odseći nepotrebne desne kolone u trouglu. Moguće je odseći i deo ispod odgovarajuće dijagonale – takva odsecanja se obično ne radi u dinamičkom programiranju naviše jer ne menjaju asimptotsku složenost, dok ih rekurzivna implementacija zasnovana na memoizaciji prirodno izbegava. Na narednoj slici su obeleženi elementi Paskalovog trougla koji su potrebni za izračunavanje broja $\binom{6}{3} = 20$.

```
long long brojKombinacija(int K, int N) {
  // alociramo prostor za smeštanje relevantnog dela trougla
  vector<vector<long long>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(min(K+1, n+1));
  // trougao popunjavamo kolonu po kolonu
  for (int n = 0; n <= N; n++) {
```



Elementi Paskalovog trougla potrebni za izračunavanje broja $\binom{6}{3}$

```
    // na početku svake vrste nalazi se 1
    dp[n][0] = 1;
    // unutrašnje elemente izračunavamo kao zbir elemenata
    // iznad njih
    for (int k = 1; k <= min(n-1, K); k++)
      dp[n][k] = dp[n-1][k-1] + dp[n-1][k];
    // ako je potrebno da znamo krajnji element kolone,
    // postavljamo ga na vrednost 1
    if (n <= K)
       dp[n][n] = 1;
  }
  // vraćamo traženi rezultat
  return dp[N][K];
}
```

Pažljivijom analizom prethodnog koda vidimo da, kako je to obično slučaj u dinamičkom programiranju, ne moramo istovremeno čuvati sve elemente matrice, jer svaka vrsta zavisi samo od prethodne i dovoljno je umesto matrice čuvati samo njene dve vrste (prethodnu i tekuću). Zapravo, dovoljno je čuvati samo jedan vektor vrstu ako je pažljivo popunjavamo i ako tokom njenog ažuriranja u jednom njenom delu čuvamo tekuću, a u drugom narednu vrstu. Pošto element (n, k) zavisi od elementa $(n-1, k-1)$ i od elementa $(n-1, k)$ znači da svaki element zavisi od elemenata koji su levo od njega, ali ne od elemenata koji su desno od njega. Zato ćemo vektor popunjavati zdesna nalevo. Pretpostavićemo da tokom ažurira-

nja važi invarijanta da se na pozicijama strogo većim od k nalaze elementi vrste n, a da se na pozicijama manjim ili jednakim od k nalaze elementi vrste $n-1$. Ažuriranje započinje time što na kraj vrste dopišemo vrednost 1 (osim u slučaju kada vršimo sasecanje desnog dela trougla) i nastavlja se tako što se element na poziciji k uveća za vrednost na poziciji $k-1$. Zaista, pre ažuriranja se na poziciji k nalazi vredost trougla sa pozicije $(n-1,k)$, dok se na poziciji $k-1$ nalazi vrednost trougla sa pozicije $(n-1,k-1)$. Njihov zbir je vrednost trougla na poziciji (n,k), pa se on upisuje na poziciju k i nakon toga se k smanjuje za 1, čime se invarijanta održava. Ažuriranje se vrši do pozicije $k=1$, jer se na poziciji $k=0$ u svim vrstama nalazi vrednost 1.

Memorijska složenost ovog rešenja je $O(k)$, dok je vremenska $O(n \cdot k)$. Primetimo kako smo od veoma neefikasnog rešenja eksponencijalne složenosti tehnikom dinamičkog programiranja dobili veoma efikasno i uz to prilično jednostavno rešenje.

```
l// -*- hide -*-
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

long long brojKombinacija(int K, int N) {
  // tekuća vrsta
  vector<long long> dp(K+1);
  // na početku svake vrste nalazi se 1
  dp[0] = 1;
  // trougao popunjavamo vrstu po vrstu
  for (int n = 1; n <= N; n++) {
    // vrstu ažruriramo zdesna nalevo
    // na kraju svake vrste nalazi se 1
    if (n <= K) dp[n] = 1;
    // ažuriramo unutrašnje elemente
    for (int k = min(n-1, K); k > 0; k--)
        dp[k] += dp[k-1];
  }
```

```
    // vraćamo traženi rezultat
    return dp[K];
}
```

Drugačija rekurzivna definicija

Napomenimo i da smo mogli krenuti od algoritma nabrajanja svih kombinacija u kom se u petlji razmatraju svi kandidati za element na tekućoj poziciji. Time bi se dobio algoritam koji bi element $\binom{n}{k}$ računao po sledećoj formuli:

$$\binom{n}{k} = \sum_{n'=k}^{n} \binom{n'}{k-1}.$$

Zadatak: Broj particija

Particija pozitivnog prirodnog broja n je rastavljanje broja n na zbir nekoliko pozitivnih prirodnih brojeva pri čemu je redosled sabiraka nebitan (stoga možemo pretpostaviti da je taj redosled ili uvek nerastući ili uvek neopadajući). Na primer, ako je redosled nerastući, particije broja 4 su $1+1+1+1$, $2+1+1$, $2+2$, $3+1$, 4. Napisati program koji određuje broj particija za dati prirodan broj n.

Opis ulaza

Prva i jedina linija standardnog ulaza sadrži prirodan broj n ($n \leq 100$).

Opis izlaza

Na standardnom izlazu prikazati u prvoj liniji broj particija prirodnog broja n.

Primer 1

Ulaz	*Izlaz*	*Objašnjenje*

`6`	`11`	Ako su particije sa neopadajuće sortiranim sabircima, to su particije:
		`1+1+1+1+1+1`
		`1+1+1+1+2`
		`1+1+1+3`
		`1+1+2+2`
		`1+1+4`
		`1+2+3`
		`1+5`
		`2+2+2`
		`2+4`
		`3+3`
		`6`

Primer 2

Ulaz	*Izlaz*
`100`	`190569292`

Rešenje

Rekurzivne procedure koje nabrajaju sve particije se mogu prilagoditi tako da izračunaju broj particija.

Svaka particija ima svoj prvi sabirak. Svakoj particiji broja n kojoj je prvi sabirak s (pri čemu je $1 \leq s \leq n$) jednoznačno odgovara neka particija broja $n - s$. Nametnućemo uslov da su sabirci u svakoj particiji sortirani nerastuće. Zato, ako je prvi sabirak s, svi sabirci iza njega moraju da budu manji ili jednaki od s. Zato nam nije dovoljno samo da umemo da prebrojimo sve particije broja $n - s$, već je potrebno da ojačamo induktivnu hipotezu. Označimo sa $p_{n,s_{max}}$ broj particija broja n u kojima su svi sabirci manji ili jednaki od s_{max}.

- Bazu indukcije čini slučaj $n = 0$, jer broj nula ima samo jednu particiju koja ne sadrži sabirke. Dakle, važi da je $p_{0,s_{max}} = 1$. Ako je n veće od nula i $s_{max} = 0$, tada ne postoji ni jedna particija, jer od sabiraka koji su svi jednaki nuli (jer svi moraju da budu manji ili jednaki s_{max}) ne možemo nikako napraviti neki pozitivan broj. Dakle, za $n > 0$ važi da je $p_{n,0} = 0$.

- Induktivni korak možemo ostvariti na više načina. Najjednostavniji je sledeći. Prilikom izračunavanja $p_{n,s_{max}}$ možemo razmatrati dva slučaja: da se u zbiru ne javlja sabirak s_{max} ili da se u zbiru javlja sabirak s_{max}. Ako se u zbiru ne javlja sabirak s_{max}, tada je najveći sabirak $s_{max} - 1$ i broj takvih particija je $p_{n,s_{max}-1}$. Drugi slučaj je moguć samo kada je $n \geq s_{max}$ i broj takvih particija je $p_{n-s_{max},s_{max}}$. Na osnovu ovoga, dobijamo narednu rekurzivnu funkciju.

```
// particije broja n u kojima je najveci sabirak jednak smax
int brojParticija(int n, int smax) {
  if (n == 0) return 1;
  if (smax == 0) return 0;
  int broj = brojParticija(n, smax - 1);
  if (n >= smax)
    broj += brojParticija(n - smax, smax);
  return broj;
}

int brojParticija(int n) {
  // particije broja n u kojima je najveći sabirak jednak n
  return brojParticija(n, n);
}
```

Rekurzivnu funkciju za izračunavanje broja particija možemo dobiti i tako što na tekuće mesto u particiji postavljamo sve moguće kandidate za proširivanje tekuće particije (to su svi brojevi s od 1 do manjeg od brojeva s_{max} i n) i nastavljamo generisanje particija broja $n - s$ kod kojih je najveći sabirak jednak s.

```
// particije broja n u kojima je najveći sabirak jednak smax
int brojParticija(int n, int smax) {
   if (n == 0) return 1;
   if (smax == 0) return 0;
   int broj = 0;
   for (int s = min(smax, n); s >= 1; s--)
     broj += brojParticija(n - s, s);
   return broj;
```

```
}

int brojParticija(int n) {
  // particije broja n u kojima je najveći sabirak jednak n
  return brojParticija(n, n);
}
```

Sva rešenja zasnovana na prostoj rekurziji su neefikasna, jer dolazi do ponavljanja identičnih rekurzivnih poziva i mogu se popraviti dinamičkim programiranjem. Prikažimo to na primeru brojanja nerastuće uređenih permutacija u kojima vršimo dva rekurzivna poziva.

Krenimo sa memoizacijom. Uvodimo matricu dimenzije $(n+1) \times (n+1)$ koju popunjavamo vrednostima -1, čime označavamo da rezultat poziva funkcije još nije poznat. Pre nego što krenemo sa izračunavanjem proveravamo da li je u matrici vrednost različita od -1 i ako jeste, vraćamo tu upamćenu vrednost. Pre svake povratne vrednosti funkcije rezultat pamtimo u matrici.

```
int brojParticija(int n, int smax, vector<vector<int>>& memo) {
  if (memo[n][smax] != -1)
    return memo[n][smax];
  if (n == 0) return memo[n][smax] = 1;
  if (smax == 0) return memo[n][smax] = 0;
  int broj = brojParticija(n, smax-1, memo);
  if (n >= smax)
    broj += brojParticija(n-smax, smax, memo);
  return memo[n][smax] = broj;
}

int brojParticija(int n) {
  vector<vector<int>> memo(n + 1);
  for (int i = 0; i <= n; i++)
    memo[i].resize(n+1, -1);
  return brojParticija(n, n, memo);
}
```

Umesto memoizacije možemo upotrebiti i dinamičko programiranje naviše. Prikažimo tabelu vrednosti funkcije za $n = 7$.

```
n\smax   0  1  2  3  4  5  6  7
0        1  1  1  1  1  1  1  1
1        0  1  1  1  1  1  1  1
2        0  1  2  2  2  2  2  2
3        0  1  2  3  3  3  3  3
4        0  1  3  4  5  5  5  5
5        0  1  3  5  6  7  7  7
6        0  1  4  7  9 10 11 11
7        0  1  4  8 11 13 14 15
```

Na osnovu baze indukcije znamo da će svi elementi prve vrste biti jednaki 1, a da će u prvoj koloni svi elementi osim početnog biti jednaki 0. Jedan od načina da se matrica popunjava je postepeno uvećavajući vrednost n, tj. popunjavajući vrstu po vrstu.

Element $p_{n,s}$ zavisi od elemenata $p_{n,s-1}$ i (ako je $n \geq s$) $p_{n-s,s}$ i ako se vrste popunjavaju od gore naniže i sleva nadesno, prilikom njegovog izračunavanja oba elementa od kojih zavisi su već izračunata, što daje korektan algoritam.

```
int brojParticija(int N) {
  // alociramo matricu
  vector<vector<int>> dp(N+1);
  for (int n = 0; n <= N; n++)
    dp[n].resize(N+1);
  // popunjavamo prvu vrstu
  for (int smax = 0; smax <= N; smax++)
    dp[0][smax] = 1;
  // popunjavamo preostale elemente prve kolone
  for (int n = 1; n <= N; n++)
    dp[n][0] = 0;
  // popunjavamo jednu po jednu vrstu
  for (int n = 1; n <= N; n++)
    for (int smax = 1; smax <= N; smax++) {
      dp[n][smax] = dp[n][smax-1];
      if (n >= smax)
        dp[n][smax] += dp[n-smax][smax];
    }
  return dp[N][N];
```

```
}
```

I vremenska i memorijska složenost prethodnog algoritma je $O(n^2)$. Iako se matrica popunjava vrstu po vrstu, to nije moguće popraviti (jer elementi zavise od elemenata koji se javljaju ne samo u prethodnoj, već i u ranijim vrstama, tako da je potrebno da istovremeno čuvamo sve prethodne vrste). Međutim, ako matricu popunjavamo kolonu po kolonu odozgo naniže, možemo dobiti memorijsku složenost $O(n)$ – vremenska složenost ostaje $O(n^2)$. Naime, svaki element zavisi od elementa u istoj vrsti u prethodnoj koloni i elementa u istoj koloni u nekoj od prethodnih vrsta, tako da ako kolone popunjavamo odozgo naniže, možemo čuvati samo dve uzastopne kolone. Zapravo, možemo čuvati i samo jednu kolonu, ako njeno popunjavanje organizujemo tako da se tokom ažuriranja svi elementi pre tekuće vrste odnose na vrednosti tekuće kolone, a od tekuće vrste do kraja odnose na vrednosti prethodne kolone. Primetimo da se u delu gde je $n < s_{max}$, vrednosti između dve susedne kolone ne menjaju. Time dobijamo narednu optimizovanu implementaciju.

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

using namespace std;

int brojParticija(int N) {
  vector<int> dp(N+1, 0);
  dp[0] = 1;
  for (int smax = 1; smax <= N; smax++)
    for (int n = smax; n <= N; n++)
      dp[n] += dp[n-smax];
  return dp[N];
}

int main() {
  int n;
  cin >> n;
  cout << brojParticija(n) << endl;
  return 0;
```

```
}
```

8.2.1 *Katalanovi brojevi*

8.3 *Optimizacija dinamičkim programiranjem*

Dinamičko programiranje se često primenjuje u rešavanju optimizacionih zadataka (zadataka u kojima se traži da se odredi najmanja ili najveća vrednost koja zadovoljava neki uslov). Da bi optimizacioni zadatak mogao da se rešava dinamičkim programiranjem, potrebno je formulisati njegovo rekurzivno rešenje, što znači da problem mora da zadovoljava takozvano svojstvo *optimalne podstrukture*. To znači da se optimalno rešenje polaznog problema može odrediti na osnovu optimalnih rešenja potproblema istog oblika, ali manje dimenzije.

Na primer, najkraći put izmeću tačke A i tačke B koji prolazi preko tačke C se dobija tako što se iskombinuju najkraći put između tačke A i tačke C i nakraći put između tačke C i tačke B, što znači da problem određivanja najkraćih puteva ima svojstvo optimalne podstrukture i može se rešavati dinamičkim programiranjem (na toj tehnici je zasnovan Dajkstrin algoritam, koji je jedan od najčuvenijih algoritama za rešvanje tog problema).

Međutim, najjeftiniji avionski let od aerodroma A do aerodroma B preko aerodroma C ne mora da bude kombinacija najjeftinijih letova od A do C i od C do B (zbog posebnih popusta koje avio-kompanije nude za povezane letove), tako da se problem određivanja najjeftinijeg leta nema svojstvo optimalne podstrukture i ne može se rešavati dinamičkim programiranjem.

Zadatak: Edit rastojanje

Edit-rastojanje između dve niske se definiše u terminima operacija umetanja, brisanja i izmena slova prve reči kojima se može dobiti druga reč. Svaka od ove tri operacije ima svoju cenu. Definisati program koji izračunava najmanju cenu operacija kojima se od prve niske može dobiti druga. Na primer, ako je cena svake operacije jedinična, tada se niska `zdravo` može pretvoriti u `bravo!` najefikasnije operacijom izmene slova `z` u `b`, brisanja slova `d` i umetanja karaktera `!`.

Opis ulaza

Sa standardnog ulaza se učitavaju dve niske dužine najviše 100 karaktera, a zatim

cene operacije umetanja, brisanja i izmene (prirodni brojevi od 1 do 10, svaki u posebnom redu).

Opis izlaza

Na standardni izlaz ispisati traženu vrednost edit-rastojanja.

Primer 1		**Primer 2**	
Ulaz	*Izlaz*	*Ulaz*	*Izlaz*
`zdravo`	`3`	`kitten`	`7`
`bravo!`		`sitting`	
`1`		`1`	
`1`		`2`	
`1`		`3`	

Rešenje

Rekurzivno rešenje

Izvedimo prvo induktivno-rekurzivnu konstrukciju.

- Ako je prva niska prazna, najefikasniji način da se od nje dobije druga niska je da se umetne jedan po jedan karakter druge niske, tako da je minimalna cena jednaka proizvodu cene operacije umetanja i broja karaktera druge niske.
- Ako je druga niska prazna, najefikasniji način da se od prve niske dobije prazna je da se jedan po jedan njen karakter izbriše, tako da je minimalna cena jednaka proizvodu cene operacije brisanja i broja karaktera prve niske.
- Induktivna hipoteza će biti da umemo da rešimo problem za bilo koja dva prefiksa prve i druge niske. Ako su poslednja slova prve i druge niske jednaka, onda je potrebno pretvoriti prefiks bez poslednjeg slova prve niske u prefiks bez poslednjeg slova druge niske. Ako nisu, onda imamo tri mogućnosti. Jedna je da izmenimo jedan od ta dva karaktera u onaj drugi i onda da, kao u prethodnom slučaju, prevedemo prefikse bez poslednjih karaktera jedan u drugi. Druga mogućnost je da obrišemo poslednji karakter prve niske i probamo da pretvorimo tako njen dobijeni prefiks u drugu nisku. Treća mogućnost je da prvu nisku transformišemo u prefiks druge niske bez poslednjeg karaktera i da zatim dodamo poslednji karakter druge niske.

Na osnovu ovoga lako možemo definisati rekurzivnu funkciju koja izračunava edit-rastojanje. Da nam se niske ne bi menjale tokom rekurzije (što može biti sporo),

efikasnije je da niske prosleđujemo u neizmenjenom obliku i da samo prosleđujemo brojeve karaktera njihovih prefiksa koji se trenutno razmatraju.

U direktnoj rekurzivnoj implementaciji dolazi do velikog broja ponovljenih rekurzivnih poziva, što dovodi do veoma loše (eksponencijalne) složenosti.

```
int editRastojanje(const string& s1, const string& s2, int n1, int n2) {
  if (n1 == 0)
    return n2 * cenaUmetanja;
  if (n2 == 0)
    return n1 * cenaBrisanja;
  if (s1[n1-1] == s2[n2-1])
    return editRastojanje(s1, s2, n1-1, n2-1);
  int r1 = editRastojanje(s1, s2, n1-1, n2) + cenaUmetanja;
  int r2 = editRastojanje(s1, s2, n1, n2-1) + cenaBrisanja;
  int r3 = editRastojanje(s1, s2, n1-1, n2-1) + cenaIzmene;
  return min({r1, r2, r3});
}

int editRastojanje(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  return editRastojanje(s1, s2, n1, n2);
}
```

Dinamičko programiranje naviše

Rešenje direktnom rekurzijom je, naravno, izrazito neefikasno zbog preklapajućih rekurzivnih poziva. Algoritam dinamičkog programiranja naviše za ovaj problem poznat je pod imenom Vagner-Fišerov algoritam. Rezultate za prefikse dužine i i j pamtićemo u matrici na polju (i, j). Dakle, ako su dužine niski n_1 i n_2, potrebna nam je matrica dimenzije $(n_1+1) \times (n_2+1)$, a konačan rezultat će se nalaziti na mestu (n_1, n_2). Ako matricu popunjavamo vrstu po vrstu, sleva nadesno, prilikom izračunavanja elementa na poziciji (i, j), biće izračunati svi elementi matrice od kojeg on zavisi (a to su $(i-1, j-1)$, $(i-1, j)$ i $(i, j-1)$).

Primer 8.3.1. *Pod pretpostavkom da su cene jedinične, za niske* `zdravo` *i* `bravo!` *dobija se sledeća matrica.*

```
     b r a v o !
   0 1 2 3 4 5 6
   -------------
 0|0 1 2 3 4 5 6
z1|1 1 2 3 4 5 6
d2|2 2 2 3 4 5 6
r3|3 3 2 3 4 5 6
a4|4 4 3 2 3 4 5
v5|5 5 4 3 2 3 4
o6|6 6 5 4 3 2 3
```

Pošto se tokom rada algoritma popunjava matrica dimenzije $(n_1+1) \times (n_2+1)$, a svako polje matrice se popunjava u vremenu $O(1)$, ukupna vremenska i memorijska složenost algoritma je $O(n_1 \cdot n_2)$.

```
int editRastojanje(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2+1);

  dp[0][0] = 0;
  for (int i = 0; i <= n1; i++)
    dp[i][0] = i * cenaBrisanja;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      if (s1[i-1] == s2[j-1])
        dp[i][j] = dp[i-1][j-1];
      else {
        int r1 = dp[i-1][j] + cenaUmetanja;
        int r2 = dp[i][j-1] + cenaBrisanja;
        int r3 = dp[i-1][j-1] + cenaIzmene;
        dp[i][j] = min({r1, r2, r3});
      }
    }
```

```
  return dp[n1][n2];
}
```

Memorijska optimizacija

Na osnovu postavke zadatka, nije potrebno odrediti same izmene, već samo rastojanje (na primer, ako se vrši provera da li su dve niske bliske prilikom pretrage u kojoj se dopušta da je korisnik napravio i nekoliko slovnih grešaka). Pošto elementi tekućeg reda zavise samo od prethodnog, možemo izvršiti memorijsku optimizaciju i istovremeno čuvati samo jedan red. Tokom ažuriranja elementa na poziciji j njegov deo na pozicijama strogo manjim od j će čuvati elemente tekućeg reda i, deo od pozicije j nadalje će čuvati elemente prethodnog reda $i-1$. Promenljiva `prethodni` će čuvati vrednost sa polja $(i-1, j-1)$, a promenljiva `tekuci` će čuvati vrednost sa polja $(i-1, j)$.

Vremenska složenost nakon ove optimizacije je $O(n_1 \cdot n_2)$, dok se memorijska složenost može spustiti na $O(\min(n_1, n_2))$ (tako što se odabere da li će se popunjavati vrsta po vrsta ili kolona po kolona).

```
int editRastojanje(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1);
  dp[0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[j] = j * cenaUmetanja;
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    dp[0] = i * cenaBrisanja;
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni;
      else {
        int r1 = tekuci + cenaUmetanja;
        int r2 = dp[j-1] + cenaBrisanja;
        int r3 = prethodni + cenaIzmene;
```

```
        dp[j] = min({r1, r2, r3});
      }
      prethodni = tekuci;
    }
  }
  return dp[n2];
}
```

Rekonstrukcija optimalnog puta

Na osnovu popunjene matrice lako je rekonstruisati i sam niz koraka koji prvu nisku transformiše u drugu. Krećemo od donjeg desnog ugla matrice i krećemo se unazad. U svakom koraku proveravamo kako smo došli na tekuću poziciju i u skladu sa tim korak ubacujemo u niz (vektor). Na kraju, kada stignemo do gornjeg levog ugla, niz koraka ispisujemo unazad.

Primer 8.3.2. *U tekućem primeru na polje (6, 6) smo stigli sa polja (6, 5) što znači da je poslednji korak umetanje karaktera* `!`*. Na polje (6, 5) smo stigli sa polja (5, 4) pri čemu nije vršen nikakva izmena. Slično, na polje (5, 4) smo stigli sa (4, 3), na polje (4, 3) smo stigli sa (3, 2), a na polje (3, 2) smo stigli sa (2, 1). Na polje (2, 1) smo mogli stići sa (1, 1) pri čemu je obrisan karaker* `d`*, a na polje (1, 1) smo stigli sa (0, 0) tako što je karakter* `z` *promenjen u* `b`*. Dakle, jedan niz mogućih koraka je*

```
zdravo
          izmena z u b
bdravo
          brisanje d
bravo
          umetanje !
bravo!
```

Primetimo da smo na polje (2, 1) mogli stići i sa polja (1, 0) operacijom izmene `d` *u* `b`*. Na polje (1, 0) smo stigli sa (0, 0) operacijom brisanja slova* `z`*. Na taj način dobijamo sledeći niz koraka.*

```
zdravo
          brisanje z
dravo
          izmena d u b
```

```
bravo
          umetanje !
bravo!
```

Implementacija funkcije kojom se vrši rekonstrukcija (jednog) rešenja može biti sledeća.

Zadatak: Najduži zajednički podniz dve niske

Napiši program koji izračunava dužinu najvećeg zajedničkog podniza dve niske. Podniz čine karakteri niske koji ne moraju biti uzastopni, ali se javljaju u istom redosledu kao u originalnoj niski. Na primer za niske `abacbc` i `babbca` najduža zajednička podniska je `babc`.

Opis ulaza

Dve linije standardnog ulaza sadrže dve niske koje se sastoje od malih slova engleske azbuke i dugačke su najviše 1000 karaktera.

Opis izlaza

Na standardni izlaz ispisati samo traženu dužinu.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`xmjyauz` `mzjawxu`	`4`	Podniz obe niske dužine 4 je `mjau`

Rešenje

Rekurzivno rešenje

Ako je bilo koja od dve niske prazna, tada je jedini njen podniz prazan, pa je dužina najdužeg zajedničkog podniza jednaka nuli.

Ako su obe niske neprazne, tada možemo uporediti njihova poslednja slova.

- Ako su ona jednaka, ona mogu biti uključena u najduži zajednički podniz dve niske i problem se rekurzivno svodi na pronalaženje najdužeg zajedničkog podniza prefiksa tih niski dobijenih isključivanjem poslednjih slova. Naglasimo da nije greška eksplicitno analizirati i slučajeve kada neko od ta dva

poslednja slova nije uključeno u najduži zajednički podniz (time smo sigurniji da ćemo dobiti korektan algoritam), ali se može dokazati da za tim nema potrebe.

- U suprotnom, nije moguće da oba poslednja slova budu uključena u zajednički podniz. Zato razmatramo najduži zajednički podniz prve niske i prefiksa druge niske bez njenog poslednjeg slova i zajednički podniz druge niske i prefiksa prve niske bez njenog poslednjeg slova. Duži od dva podniza biće najduži zajednički podniz te dve niske. Naglasimo i da u ovom slučaju nije neophodno eksplicitno analizirati najduži zajednički podniz ta dva prefiksa (jer on ne može biti duži od najdužih zajedničkih podnizova dobijenih kada se neki od tih prefiksa proširi dodatnim slovom).

Pošto rekurzija teče po prefiksima niski, jedini promenljivi parametri tokom rekurzije mogu biti dužine tih prefiksa. Ako sa $f(n_1, n_2)$ označimo dužinu najdužeg zajedničkog podniza prefiksa niske s dužine n_1 i prefiksa niske t dužine n_2, tada važi sledeća rekurentna veza:

$$
\begin{aligned}
f(0, n_2) &= 0 \\
f(n_1, 0) &= 0 \\
f(n_1, n_2) &= f(n_1 - 1, n_2 - 1) + 1, \quad \text{za } s_{n_1 - 1} = t_{n_2 - 1} \\
f(n_1, n_2) &= \max\left(f(n_1, n_2 - 1), f(n_1 - 1, n_2)\right), \quad \text{za } s_{n_1 - 1} \neq t_{n_2 - 1}
\end{aligned}
$$

```
// najduzi zajednici podniz (longest common substring, LCS)
// prefiksa duzine n1 niske s1 i
// prefiksa  duzine n2 niske s2
int LCS(const string& s1, int n1,
        const string& s2, int n2) {
  if (n1 == 0 || n2 == 0)
    return 0;
  int rez = max(LCS(s1, n1, s2, n2-1),
                LCS(s1, n1-1, s2, n2));
  if (s1[n1-1] == s2[n2-1])
    rez = max(rez, LCS(s1, n1-1, s2, n2-1) + 1);
  return rez;
```

```
}

int LCS(const string& s1, const string& s2) {
  return LCS(s1, s1.size(), s2, s2.size());
}
```

Memoizacija

U direktnom rekurzivnom rešenju ima mnogo preklapajućih rekurzivnih poziva. Stoga je efikasnost moguće popraviti tehnikom dinamičkog programiranja. Jedan mogući pristup je da upotrebimo memoizaciju. Vrednost dužine najdužeg podniza za svaki par dužina prefiksa možemo pamtiti u matrici.

```
// najduzi zajednici podniz (longest common substring, LCS)
// prefiksa duzine n1 niske s1 i
// prefiksa  duzine n2 niske s2
int LCS(const string& s1, int n1,
        const string& s2, int n2,
        vector<vector<int>>& memo) {
  if (memo[n1][n2] != -1)
    return memo[n1][n2];

  if (n1 == 0 || n2 == 0)
    return memo[n1][n2] = 0;
  int rez = max(LCS(s1, n1, s2, n2-1, memo),
                LCS(s1, n1-1, s2, n2, memo));
  if (s1[n1-1] == s2[n2-1])
    rez = max(rez, LCS(s1, n1-1, s2, n2-1, memo) + 1);
  return memo[n1][n2] = rez;
}

int LCS(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> memo(n1+1);
  for (int i = 0; i <= n1; i++)
    memo[i].resize(n2 + 1, -1);
```

```
  return LCS(s1, n1, s2, n2, memo);
}
```

Dinamičko programiranje naviše

Problem prekalpajućih rekurzivnih poziva se može rešiti ako se upotrebi dinamičko programiranje naviše. Dužine najdužih podnizova prefiksa možemo čuvati u matrici. Element matrice na poziciji (n_1, n_2) zavisi samo od elemenata na pozicijama $(n_1 - 1, n_2)$, $(n_1, n_2 - 1)$ i $(n_1 - 1, n_2 - 1)$, tako da matricu požemo da popunjavamo bilo vrstu po vrstu, bilo kolonu po kolonu.

Primer 8.3.3. *Prikažimo matricu za primer dve niske* `xmjyauz` *i* `mzjawxu`.

```
     mzjawxu
    01234567
   ---------
  0|00000000
x 1|00000011
m 2|01111111
j 3|01122222
y 4|01122222
a 5|01123333
u 6|01123334
z 7|01223334
```

```
// najduzi zajednici podniz (longest common substring, LCS)
int LCS(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<vector<int>> dp(n1+1);
  for (int i = 0; i <= n1; i++)
    dp[i].resize(n2 + 1);

  for (int i = 0; i <= n1; i++)
    dp[i][0] = 0;
  for (int j = 0; j <= n2; j++)
    dp[0][j] = 0;
```

```
  for (int i = 1; i <= n1; i++)
    for (int j = 1; j <= n2; j++) {
      dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
      if (s1[i-1] == s2[j-1])
        dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
    }

  return dp[n1][n2];
}
```

Memorijska optimizacija

Možemo primetiti da se prilikom popunjavanja matrice vrstu po vrstu sadržaj svake naredne vrste popunjava samo na osnovu prethodne vrste. Stoga nije potrebno istovremeno pamtiti celu matricu, već je dovoljno pamtiti samo jednu, tekuću vrstu. Ažuriranje vrste moramo vršiti s leva nadesno, jer svaki element u tekućoj vrsti zavisi od elementa koji mu prethodi u toj vrsti. Primetimo da nam je u nekom trenutku potrebno da znamo prethodni element tekuće vrste, a ponekad prethodni element prethodne vrste, tako da prilikom ažuriranja vrste moramo da u pomoćnoj promenljivoj pamtimo staru vrednost prethodnog elementa vrsta (jer se ažuriranjem prethodnog elementa njegova stara vrednost gubi, a ona nam može zatrebati u slučaju da su odgovarajući karakteri u niskama jednaki).

```
// najduzi zajednici podniz (longest common substring, LCS)
int LCS(const string& s1, const string& s2) {
  int n1 = s1.size(), n2 = s2.size();
  vector<int> dp(n2 + 1, 0);
  for (int i = 1; i <= n1; i++) {
    int prethodni = dp[0];
    for (int j = 1; j <= n2; j++) {
      int tekuci = dp[j];
      if (s1[i-1] == s2[j-1])
        dp[j] = prethodni + 1;
      else
        dp[j] = max(dp[j-1], dp[j]);
```

```
        prethodni = tekuci;
      }
    }
    return dp[n2];
  }
```

Zadatak: Najduži rastući podniz

Napiši program koji određuje najduži strogo rastući podniz (ne obavezno uzastopnih elemenata) unutar datog niza.

Opis ulaza

Sa standardnog ulaza se učitava broj elemenata niza n ($1 \leq n \leq 50000$), a zatim elementi niza (celi brojevi, svaki u posebnom redu).

Opis izlaza

Na standardni izlaz ispisati dužinu najdužeg rastućeg podniza.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`10` `3 2 6 9 5 4 3 7 2 8`	`4`	Jedan rastući podniz dužine 4 je `2 6 7 8`.

Rešenje

Prikazaćemo dva rešenja zasnovana na dinamičkom programiranju, koja su različite efikasnosti.

Najduži rastući podniz koji se završava na datoj poziciji u nizu

U prvoj grupi rešenja razmatraćemo poziciju po poziciju u nizu. Za svaku poziciju odredićemo najduži rastući podniz čiji je to poslednji element. Najduži rastući podniz se onda dobiti kao najduži od svih tih podnizova (jer on se sigurno završava na nekoj poziciji u nizu). Prilikom određivanja dužine najdužeg rastućeg podniza koji se završava na poziciji $i \geq 0$, pretpostavićemo da za svaku prethodnu poziciju (ako ih ima) umemo da odredimo dužinu najdužeg rastućeg podniza koji se na njoj završava. Niz koji se završava na poziciji i sigurno sadrži element a_i, a može

produžiti sve one nizove koji se završavaju na nekoj poziciji j, takvoj da je $0 \leq j < i$ i $a_j < a_i$. Da bi niz koji se završava na poziciji i bio što duži, njegov prefiks koji se završava na poziciji j mora biti što duži (a dužinu najdužeg rastućeg niza za svaku poziciju j možemo odrediti rekurzivno). Zato od svih nizova koji se završavaju na pozicijama j, takvim da je $a_j < a_i$ određujemo najduži i produžavamo ga elementom a_i (ako takvih elemenata nema, tada je najduži niz koji se završava na poziciji a_i jednočlan).

U direktnoj rekurzivnoj implementaciji dolazi od preklapanja rekurzivnih poziva, što dovodi do veoma neefikasnog rešenja, eksponencijalne složenosti.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
// koji se zavrsava elementom na poziciji i
int najduziRastuciPodniz(const vector<int>& a, int i) {
  int maksI = 1;
  for (int j = 0; j < i; j++)
    if (a[j] < a[i]) {
      int maksJ = najduziRastuciPodniz(a, j);
      if (maksJ + 1 > maksI)
        maksI = maksJ + 1;
    }
  return maksI;
}

// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}
```

Memoizacija

Neefikasnost koja nastaje usled ponavljanja identičnih rekurzivnih poziva rešavamo dinamičkim programiranjem. Za memoizaciju dovoljno je da pamtimo niz dužina najdužih rastućih nizova koji se završavaju na svakoj poziciji u nizu. Pošto su sve te dužine veće ili jednake od 1 (svaki element sam za sebe čini rastući niz), niz u koji memorišemo rešenja možemo inicijalizovati nulama (što znači da je tražena dužina još nepoznata).

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
// koji se zavrsava elementom na poziciji i
int najduziRastuciPodniz(const vector<int>& a, int i,
                         vector<int>& memo) {
  if (memo[i] != 0)
    return memo[i];
  int maksI = 1;
  for (int j = 0; j < i; j++) {
    if (a[j] < a[i]) {
      int maksJ = najduziRastuciPodniz(a, j, memo);
      if (maksJ + 1 > maksI)
        maksI = maksJ + 1;
    }
  }
  return memo[i] = maksI;
}

// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  vector<int> memo(a.size(), 0);
  int maks = 0;
  for (int i = 0; i < a.size(); i++) {
    int maksI = najduziRastuciPodniz(a, i, memo);
    if (maksI > maks)
      maks = maksI;
  }
  return maks;
}
```

Dinamičko programiranje naviše

Jednostavno možemo formulisati i dinamičko programiranje naviše, tako što niz popunjavamo sleva nadesno. Nakon popunjavanja niza određujemo njegov maksimum. Naglasimo da svaki naredni element niza potencijalno zavisi od velikog broja prethodnih tako da nije moguće redukovati memorijsku složenost time što bi se pamtili samo neki elementi niza (moramo uvek znati dužine svih prethodnih elemenata).

Primer 8.3.4. *Prikažimo na primeru kako će se taj niz popunjavati.*

```
i  0 1 2 3 4 5 6 7 8 9
ai 3 2 6 9 5 4 3 7 8 2
dp 1 1 2 3 2 2 2 3 4 1
```

Na primer, kada izračunavamo element na poziciji 5 analiziramo nizove koji se završavaju elementima manjim od vrednosti 4 koja se nalazi na poziciji 5. To su vrednosti 3 na poziciji 0 i 2 na poziciji 1. U oba slučaja maksimalna dužina podniza koji se završava na toj poziciji je 1, pa se bilo koji od tih nizova produžava elementom 4 i dobija se rastući niz dužine 2.

Rešenje koje se dobija na ovaj način je memorijske složenosti $O(n)$ i vremenske složenosti $O(n^2)$.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduzi_rastuci_podniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  for (int i = 0; i < n; i++) {
    dp[i] = 1;
    for (int j = 0; j < i; j++)
      if (a[j] < a[i] && dp[j] + 1 > dp[i])
        dp[i] = dp[j] + 1;
  }

  int max = dp[0];
  for (int i = 0; i < n; i++)
    if (dp[i] > max)
      max = dp[i];
```

```
    return max;
}
```

Najmanji element kojim se završava rastući podniz date dužine

Izmenom induktivno-rekurzivne konstrukcije možemo dobiti i mnogo efikasnije rešenje. Ključna ideja je da pretpostavimo da uz dužinu d_{max} najdužeg rastućeg podniza do sada obrađenog dela niza možemo za svaku dužinu podniza $1 \leq d \leq d_{max}$ da odredimo najmanji element kojim se završava neki rastući podniz dužine d.

Primer 8.3.5. *Neka je, na primer, dat niz* $a = [3, 2, 6, 9, 5, 4, 3, 7, 2, 8]$. *Tokom prolaska kroz elemente niza* a *održavaćemo niz* DP, *u kome je na svakoj poziciji* $d - 1$ $(d \geq 1)$ *najmanji element niza* a, *kojim se završava neki rastući podniz od* a, *dužine* d.

Niz DP *menja se na sledeći način:*

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

Razmislimo sada o postupku transformisanja niza DP.

Ako se dosadašnji ukupno najduži podniz završavao elementom koji je manji od tekućeg, na kraj niza DP *dopisujemo tekući element jer smo našli podniz koji je duži za jedan od prethodno najdužeg. Tekući element je jedini, pa time i najmanji kojim se završava rastući podniz nove dužine. To se u primeru dešava prilikom obrade elementa* `3`*, elementa* `6`*, elementa* `9` *i elementa* `8`.

Razmotrimo i situaciju u kojoj obrađujemo element `5`. *Do tada smo videli elemente* `3`*,* `2`*,* `6` *i* `9`. *Element* `2` *na prvoj poziciji u tabeli označava da je najmanji element kojim*

se može završiti jednočlani rastući niz jednak 2. Element 6 na drugoj poziciji u tabeli označava da je najmanji element kojim se može završiti dvočlani rastući niz jednak 6 (u pitanju je niz 2 6 ili niz 3 6). Element 9 na trećoj poziciji u tabeli označava da je najmanji element kojim se može završiti tročlani rastući niz jednak 9 (u pitanju je niz 2 6 9 ili 3 6 9). Pošto je 5 manji od 9 nijedan od ovih tročlanih nizova nije moguće proširiti elementom 5, pa četvoročlanih rastućih nizova nema. Postavlja se pitanje da li se možda tročlani nizovi mogu završiti elementom 5, no ni to nije moguće. Naime, pošto je u tabeli dvočlanim nizovima pridružena vrednost 6, to znači da se svi dvočlani rastući nizovi završavaju bar sa 6, pa nije moguće 9 zameniti sa 5. Sa druge strane, pošto je 5 veće od 2, završni element dvočlanih nizova 6 je moguće zameniti sa 5 i time dobiti manju završnu vrednost dvočlanih nizova (to su u ovom slučaju nizovi 3 5 i 2 5). Dakle, u tabeli vrednost 6 treba zameniti vrednošću 5. Vrednost 2 levo od 6 nema smisla zameniti sa 5, jer bi se time završna vrednost jednočlanih nizova uvećala, a mi u tabeli pamtimo najmanje završne vrednosti.

Primetimo da niz DP u svakom trenutku mora da bude strogo rastući. Zaista, ako je a_i najmanji završetak strogo rastućeg niza dužine d, onda se pretposlednjim elementom a_j tog rastućeg niza (koji je manji od a_i) završava jedan rastući niz dužine $d-1$, a $DP_{d-2} \leq a_j < a_i = DP_{d-1}$.

Na osnovu analize ovog primera možemo da zaključimo da je prilikom analize svakog tekućeg elementa potrebno pronaći prvu poziciju d u tabeli na kojoj se nalazi element koji je veći ili jednak od tekućeg i poziciju d umesto toga upisati tekući element. Ako su svi elementi manji od tekućeg (ako je $d = d_{max}$), onda se tekući element dodaje na kraj niza (i u tom slučaju zapravo radimo isto - upisujemo element na poziciju d). Ostali elementi u tabeli ostaju nepromenjeni. Zaista na svim pozicijama u tabeli levo od pozicije d upisani su elementi strogo manji od tekućeg i njihovom zamenom sa tekućim se ne bi smanjila vrednost završnog elemenata tih nizova. Za elemente desno od pozicije d, iako su veći od tekućeg, ažuriranje nije moguće. U svim nizovima dužine $d' > d$ neki prefiks se morao završavati elementom na poziciji d ili elementom većim od njega, a pošto je on bio veći ili jednak od tekućeg, zamenog poslednjeg elementa tekućim ne bismo dobili više rastući niz.

Ključni dobitak nastaje kada se primeti da, pošto su elementi u tabeli sortirani, poziciju prvog elementa koji je veći ili jednak od tekućeg možemo ostvariti binarnom pretragom. Otuda sledi efikasna implementacija (u nizu `dp` vrednost najmanjeg završnog elementa za nizove dužine d pamtimo na poziciji $d-1$).

Vremenska složenost takve implementacije je $O(n \log(n))$, dok je memorijska složenost $O(n)$.

Binarna pretraga može biti izvršena bibliotečkom funkcijom.

```
// pronalazi duzinu najduzeg strogo rastuceg podniza niza a
int najduziRastuciPodniz(const vector<int>& a) {
  int n = a.size();
  vector<int> dp(n);
  int max = 0;
  for (int i = 0; i < n; i++) {
    auto it = lower_bound(dp.begin(), next(dp.begin(), max),
                          a[i]);
    *it = a[i];
    int d = distance(dp.begin(), it);
    if (d + 1 > max)
      max = d + 1;
  }
  return max;
}
```

Svođenje na problem najdužeg zajedničkog podniza

Postoji veoma jednostavno svođenje ovog problema na problem pronalaženja najdužeg zajedničkog podniza dva niza. Naime, dužina najdužeg rastućeg podniza datog niza jednaka je dužini najdužeg zajedničkog podniza tog niza i niza koji se dobija neopadajućim sortiranjem i uklanjanjem duplikata tog niza.

```
int najduziRastuciPodniz(const vector<int>& a) {
  // sortirana kopija niza a
  vector<int> b = a;
  sort(begin(b), end(b));
  // uklanjamo duplikate iz vektora b
  b.erase(unique(begin(b), end(b)), end(b));
  return najduziZajednickiPodniz(a, b);
}
```

Zadatak: Ranac 0-1

Programer se seli iz jedne kompanije u drugu i želi da ponese predmete iz svoje stare kancelarije, međutim, na raspolaganju samo ima jedan veliki ranac u koji možda ne mogu da stanu svi predmeti. Ako je poznata masa i vrednost svakog od predmeta i ako je poznata nosivost ranca, napiši program koji određuje maksimalnu vrednost skupa predmeta koje programer može da prenese u rancu.

Opis ulaza

Sa standardnog ulaza se unosi nosivost ranca (ceo broj između 1 i 150), zatim broj predmeta n (ceo broj između 1 i 30), zatim u narednih n redova mase i cene predmeta (mase su celi brojevi između 1 i 10, a cene realni brojevi između $1,0$ i $100,0$, zaokruženi na dve decimale).

Opis izlaza

Na standardni izlaz ispisati najveću vrednost predmeta koji se mogu preneti u rancu, zaokruženu na dve decimale.

Primer 1

Ulaz	*Izlaz*	*Objašnjenje*

`5`
`3`
`1 6.00`
`2 10.00`
`3 12.00`
	`22.00`	Najveća vrednost se dobije ako se prenesu predmeti mase 2 i mase 3 (vrednost je tada $10,0 + 12,0 = 22,0$).

Primer 2

Ulaz	*Izlaz*	*Objašnjenje*

`15`
`5`
`12 4.00`
`2 2.00`
`1 2.00`
`1 1.00`
`4 10.00`
	`15.00`	Najveća vrednost se dobije ako se prenesu predmeti masa 2, 1, 1 i 4 kilograma (vrednost je tada $2,0 + 2,0 + 1,0 + 10,0 = 15,0$).

Rešenje

Gruba sila

Iako neke varijante problema ranca dopuštaju određena gramziva rešenja, u ovoj varijanti u kojoj se predmeti ne mogu deliti već se uzimaju u celini (tzv. 0-1 varijanta problema ranca), potrebno je napraviti algoritam zasnovan na nekom obliku (optimizovane) iscrpne pretrage. Nije teško konstruisati primere koji pokazuju da gramzivi algoritam koji bi u ranac prvo stavljao najvredniji predmet ili koji bi prvo stavljao predmet koji je najvredniji po jedinici mase ne dovode uvek do optimalnog rešenja.

Rešenje grubom silom podrazumeva da se ispitaju svi mogući podskupovi predmeta i da se pronađe onaj koji može da stane u ranac a ima najveću vrednost.

Nabrajanje podskupova može da se izvrši rekurzivnom funkcijom, čiji je parametar dužina niza predmeta n.

- Ako je niz predmeta prazan (ako je $n = 0$), tada je maksimalna vrednost koja se može spakovati jednaka nuli.
- Ako niz predmeta nije prazan (ako je $n > 0$), tada analiziramo mogućnost da njegov poslednji element izostavimo ili da ga spakujemo u ranac. U prvom slučaju rekurzivnu funkciju pozivamo za istu vrednost kapaciteta ranca i dužinu niza $n - 1$. Drugi slučaj je moguć samo ako je masa poslednjeg predmeta manja od kapaciteta ranca. U tom slučaju na rezultat rekurzivnog poziva gde je nosivost ranca umanjena za masu tog predmeta i gde je prosleđena dužina niza $n - 1$ dodajemo cenu poslednjeg predmeta. Najveću vrednost dobijamo tako što izračunamo maksimum dva analizirana slučaja (kada poslednji predmet nije i kada jeste stavljen u ranac).

Ako sa $f(n, W)$ obeležimo maksimalnu vrednost koja se može dobiti tako što se neki od prvih n predmeta spakuju u ranac nosivosti W, važe sledeće veze:

$$
\begin{aligned}
f(0, W) &= 0 \\
f(n, W) &= \max\left(f(n-1, W), f(n-1, W - w_{n-1})\right) + v_{n-1}, \quad \text{za } w_{n-1} \leq W \\
f(n, W) &= f(n-1, W), \quad \text{za } w_{n-1} > W
\end{aligned}
$$

Primetimo da se rekurzivnim pozivima traži optimum za skup bez poslednjeg predmeta, što je u redu, jer je za globalni optimum neophodno i da su predmeti iz tog podskupa odabrani optimalno (zadovoljen je uslov optimalne podstrukture).

U naivnoj rekurzivnoj implementaciji dolazi do preklapanja identičnih rekurzivnih poziva, pa je ta implementacija veoma neefikasna (složenost najgoreg slučaja joj je eksponencijalna).

```
double maxCena(const vector<int>& mase,
               const vector<double>& cene,
               int nosivost, int n) {
  vector<double> dp(nosivost + 1);
  dp[0] = 0.0;
  for (int N = 1; N <= n; N++)
    for (int M = nosivost; M >= 0; M--)
      if (mase[N-1] <= M)
        dp[M] = max(dp[M], dp[M - mase[N-1]] + cene[N-1]);
  return dp[nosivost];
}
```

Rešenje problema ponovljenih rekurzivnih poziva, naravno, dolazi u obliku dinamičkog programiranja (bilo memoizacije, bilo dinamičkog programiranja naviše). Pošto imamo dva promenljiva parametra (broj predmeta n i nosivost ranca W), alociramo takvu matricu da svakoj vrsti odgovara jedan prefiks niza predmeta, a svakoj koloni jedna nosivost. Matricu možemo popunjavati vrstu po vrstu.

Primer 8.3.6. *Prikažimo rad algoritma na primeru kada je nosivost ranca 15 kilograma i kada imamo 5 predmeta čije su mase i cene redom* $(12, 4.00)$, $(2, 2.00)$, $(1, 2.00)$, $(1, 1.00)$ *i* $(4, 10.00)$.

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

Vrednost 15 u donjem desnom uglu ukazuje na to da ako imamo ranac nosivosti 15 kilograma i svih 5 predmeta na raspolaganju, tada je najveća vrednost koju možemo poneti 15. Dalje, na primer, vrednosti u vrsti 2 ukazuju na maksimalne vrednosti koje možemo poneti ako uzimamo samo neke od prva dva predmeta (to su onaj mase 12 i vrednosti 4,0 i onaj mase 2 i vrednosti 2,0) Ako je nosivost manja od 2, nijedan od ova dva predmeta ne može biti spakovan. Ako je nosivost između 2 i 11 tada može biti spakovan samo predmet vrednosti 2,0. Ako je nosivost 12 ili 13, tada nam se više isplati da spakujemo predmet mase 12 i vrednost 4,0. Kada nosivost pređe 14, tada može zapakovati oba predmeta, pa je ukupna vrednost 6,0.

Ovim smo dobili algoritam čija je i vremenska i memorijska složenost $O(n \cdot W)$ gde je n broj predmeta, a W nosivost ranca.

Primetimo da je veoma važno bila pretpostavka da su nosivost ranca i mase predmeta celobrojne. Takođe, obratimo pažnju na to da iako deluje da smo ovaj problem rešili u polinomskojo vremenskoj složenosti, to zapravo nije slučaj. Naime, složenost u ovom slučaju ne zavisi i nije izražena samo u terminima veličine ulaza, već u terminima vrednosti na ulazu (vrednosti nosivosti ranca). Za ovakve algoritme se kaže da su pseudo-polinomski. Veličina ulaza vezanog za broj W odgovara broju cifara broja W (npr. broju binarnih cifara upotrebljenih u zapisu), a vreme izvršavanja algoritma eksponencijalno raste u odnosu na taj broj. Za veće vrednosti W dobili bismo veoma neefikasan algoritam (i što se tiče utrošene memorije i što se tiče vremena izvršavanja).

```
double maxCena(const vector<int>& mase,
               const vector<double>& cene,
               int nosivost, int n) {
  vector<vector<double>> dp(nosivost + 1);
  for (int M = 0; M <= nosivost; M++) {
    dp[M].resize(n+1);
    dp[M][0] = 0.0;
  }

  for (int N = 1; N <= n; N++) {
    for (int M = 0; M <= nosivost; M++) {
      dp[M][N] = dp[M][N-1];
      if (mase[N-1] <= M)
```

```
        dp[M][N] = max(dp[M][N-1],
                       dp[M-mase[N-1]][N-1] + cene[N-1]);
    }
  }
  return dp[nosivost][n];
}
```

Možemo primetiti da elementi svake vrste zavise samo od prethodne, tako da ne moramo čuvati celu matricu, već samo tekuću vrstu. Ažuriranje vrste tada vršimo s njenog desnog kraja.

Ovom optimizacijom se memorijska složenost smanjuje na $O(W)$, a vremenska složenost ostaje $O(n \cdot W)$. Primetimo da vremenska, ali i memorijska složenost ostaje eksponencijalna u odnosu na veličinu ulaza (broj bitova potrebnih za zapis vrednosti W).

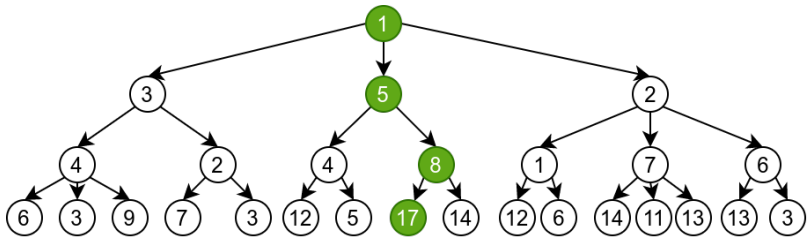
```
double maxCena(const vector<int>& mase,
               const vector<double>& cene,
               int nosivost, int n) {
  if (n == 0)
     return 0.0;
  double cenaBez = maxCena(mase, cene, nosivost, n-1);
  if (mase[n-1] > nosivost)
     return cenaBez;
  double cenaSa = maxCena(mase, cene, nosivost-mase[n-1], n-1) +
                  cene[n-1];
  return max(cenaBez, cenaSa);
}
```

8.3.1 *Isecanje šavova*

9. Gramzivi algoritmi

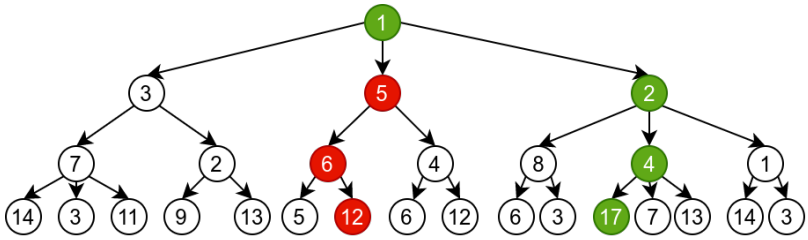
Algoritmi zasnovani na pretrazi do rešenja dolaze kroz niz koraka gde u svakom koraku analiziraju nekoliko mogućnosti. Algoritmi kod kojih se umesto analize različitih mogućnosti u svakom koraku uzima neko *lokalno optimalno* rešenje nazivaju se **pohlepni** ili **gramzivi algoritmi** (engl. greedy algorithms). Takve algoritme obično ima smisla primenjivati samo kod problema kod kojih postoji garancija da će takvi izbori na kraju dovesti do *globalno optimalnog* rešenja.

Na slici su prikazana dva drveta pretrage takva da u slučaju prvog drveta pohlepni algoritam koji u svakom koraku bira naslednika sa najvećom mogućom vrednošću dovodi do optimalnog rešenja (maksimalne moguće vrednosti), dok u slučaju drugog drveta pohlepni algoritam dovodi do neoptimalnog rešenja (vrednosti koja nije optimalna).



Drvo pretrage kod kojeg gramzivi algoritam dovodi do optimalnog rešenja tj. rešenja maksimalne vrednosti

Pohlepni algoritmi nam, dakle, pružaju jasnu strategiju (biraj što bolju od svih raspoloživih mogućnosti) kako da u svakom koraku izaberemo jednu od više ponuđenih mogućnosti tako da na kraju dođemo do željenog optimalnog rešenja. U nastavku ćemo pojam gramzivih algoritama malo proširiti i razmatraćemo algoritme koji u svakom koraku biraju samo jednu mogućnost na osnovu neke precizno



Drvo pretrage kod kojeg gramzivi algoritam ne dovodi do optimalnog rešenja tj. rešenja maksimalne vrednosti

definisane strategije, tako da ti izbori garantuju da će se na kraju doći do željenog (ispravnog tj. optimalnog) rešenja problema.

Pohlepni algoritmi ne vrše ispitivanje različitih slučajeva niti iscrpnu pretragu i stoga su po pravilu veoma efikasni (u svakom koraku je izvršeno maksimalno moguće odsecanje). Takođe, obično se veoma jednostavno implementiraju. Sa druge strane, kao i kod svih drugih algoritama u kojima se koristi odsecanje, potrebno je unapred dokazati da se pohlepnim algoritmom dobija korektno tj. optimalno rešenje, što u nekim slučajevima može biti veoma izazovno. Samo nalaženje ispravnog pohlepnog algoritma (tj. strategije) može predstavljati ozbiljan izazov i često nije trivijalno odrediti da li za neki problem postoji ili ne postoji pohlepno rešenje.

Kod nekih problema pohlepni algoritmi ne dovode uvek do optimalnog rešenja, ali se može dokazati da će rešenja koja se dobijaju biti kvalitetna i neće se puno razlikovati od optimalnih, što može opravdati upotrebu pohlepnih algoritama (jer oni mogu biti mnogo efikasniji od iscrpnih algoritama koji garantuju optimalnost). U tom slučaju pohlepni algoritmi su *heuristike* (tehnike koje ne garantuju da će uvek dovesti do optimalnog rešenja, ali koji dovode do dovoljno dobrih rešenja).

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

Da bi se dokazala korektnost pohlepnog algoritma, obično je potrebno dokazati nekoliko stvari. Iako ćemo nekada gramzivim algoritmima rešavati probleme u

kojima se zahteva da se ispita da se proveri da li postoji neko rešenje koje zadovoljava date uslove i da se pronađe bilo koje takvo rešenje, najčešće ćemo razmatrati probleme u kojima se zahteva da se u grupi rešenja koja zadovoljavaju neke date uslove (ispravnih rešenja) pronađe ono optimalno (u slučaju kada postoji više takvih optimalnih rešenja obično je dovoljno da se pronađe bilo koje).

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

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

    - Jedna tehnika da se optimalnost dokaže je to da se pokaže da se optimalno rešenje malo po malo, primenom transformacije pojedinačnih koraka može pretvoriti u rešenje dobijeno na osnovu naše strategije. Obično je dovoljno dokazati da se prvi korak optimalnog rešenja može zameniti prvim korakom koji gramziva strategija sugeriše, tako da se korektnost i kvalitet rešenja time ne narušavaju i korektnost dalje sledi na osnovu induktivnog argumenta. Ovu tehniku nazivaćemo **tehnika razmene** (engl. exchange).

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

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

Zadatak: Žaba na kamenju

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

Opis ulaza

Sa standardnog ulaza se unosi broj n ($1 \leq n \leq 50000$), a zatim u narednom redu n pozitivnih celih brojeva broj (u pitanju je rastuće sortiran niz brojeva koji predstavlja koordinate kamenja). U poslednjem redu se nalazi pozitivan ceo broj r.

Opis izlaza

Na standardni izlaz ispisati najmanji broj skokova potreban da žaba stigne do poslednjeg kamena ili -1 ako to nije moguće.

Primer

Ulaz	*Izlaz*

`5`
`0 3 8 14 16`
`10`
	`2`

Rešenje

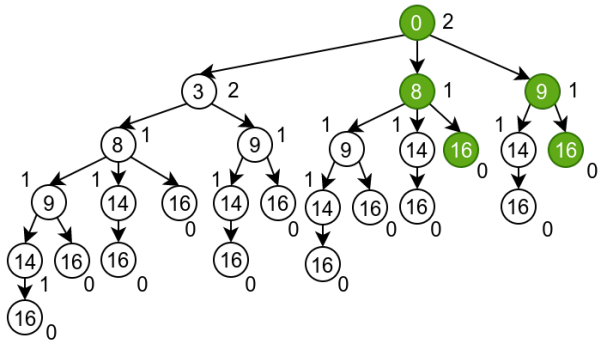
Gruba sila - dinamičko programiranje

Jedan način da se zadatak reši je grubom silom, tj. isprobavanjem svih mogućnosti. Iako se na ovaj način dobija algoritam čija je korektnost prilično očigledna (jer su sve mogućnosti eksplicitno ispitane), u ovom zadatku takvo rešenje je nepotrebno neefikasno, jer, kako ćemo videti, postoji veoma jednostavna gramziva strategija koja uvek dovodi do optimalnog rešenja (najmanjeg broja skokova).

Osnovna ideja algoritma grube sile je da se pokuša skok sa početnog kamena na sve kamenove na koje se može doskočiti i da se zatim rekurzivno izračuna minimalni broj skokova sa kamena na koji smo doskočili do krajnjeg kamena. Ako eliminišemo kamenove na koje smo doskočili sa kojih ne postoji put do krajnjeg kamena i od preostalih kamenova pronađemo minimalni broj skokova do krajenjeg kamena,

minimalni broj skokova od početnog kamena do krajnjeg je broj koji je za jedan veći od tog broja.

Primer 9.0.1. *Na slici je prikazano drvo iscrpne pretrage za niz kamenova* $0, 3, 8, 9, 14, 16$ *i dužinu skoka* $r = 10$*. Sa kamena* 0*, žaba može da skoči na kamenove* 3*,* 8 *i* 9*, sa kamena* 3 *na kamenove* 8 *i* 9*, sa kamena* 8 *na kamenove* 9*,* 14 *i* 16*, sa kamena* 9 *na kamenove* 14 *i* 16 *i sa kamena* 14 *na kamen* 16*. Uz svaki čvor je obeležen i najmanji broj skokova potrebnih da žaba dođe do završnog kamena* 16*. Označene su i dve najkraće putanje (*$0 - 8 - 16$ *i* $0 - 9 - 16$*).*



Drvo iscrpne pretrage

Pošto jednostavna rekurzivna implementacija dovodi do ponovljenih rekurzivnih poziva (može se očekivati da ćemo tokom pretrage često dolaziti na neki kamen i računati minimalni broj skokova od tog kamena do kraja), program treba optimizovati dinamičkim programiranje.

Jedno rešenje je da se upotrebi memoizacija. Ako bi se sa svakog kamena moglo doskočiti do svakog drugog, ovim bismo dobili rešenje složenosti $O(n^2)$ (uz korišćenje pomoćnog niza dužine n).

Umesto memoizacije možemo upotrebiti i dinamičko programiranje naviše. Niz u kome čuvamo minimalni broj skokova do krajnjeg kamena treba popunjavati od krajnjeg kamena nalevo, u opadajućem redosledu koordinata. Složenost ovog rešenja je takođe $O(n^2)$ (uz korišćenje pomoćnog niza dužine n).

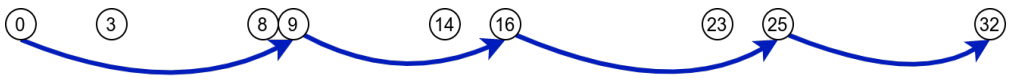
Optimalna strategija - gramzivi algoritam

Iako u svakom koraku žaba može imati izbor između nekoliko kamenova na koje može da skoči, intuitivno nam je sasvim jasno da ona ništa ne gubi time što skoči

što dalje. Stoga je jasno da žaba u svakom koraku treba da skoči na što dalji kamen koji je na rastojanju najviše r. Ako nijedan kamen koji je na rastojanju najviše r ne postoji, tada žaba ne može stići do kraja. Nakon što žaba napravi prvi skok, postupak se ponavlja na isti način, sve dok ne stigne do kraja ili dok se ne dođe do situacije u kojoj žaba ne može da skoči dalje.

U svakom koraku tražimo najdalji kamen na koji žaba može da skoči i to tako što tražimo prvi kamen na koji žaba ne može da doskoči (to možemo uraditi linearnom, ali i binarnom pretragom). Kada nađemo kamen na koji žaba ne može da doskoči žaba skače na kamen ispred njega (ako žaba može da skoči na svaki kamen, tada skače na poslednji, a ako ne može da skoči ni na jedan kamen, tada konstatujemo da do poslednjeg kamena ne može da doskoči).

Primer 9.0.2. *Razmotrimo primer kada je niz kamenja* $0, 3, 8, 9, 14, 16, 23, 25, 32$ *i kada je dužina skoka* $r = 10$. *Sa početnog kamena* 0 *žaba skače na kamen* 9 *(to je najdalji kamen na koji može da doskoči sa kamena* 0*). Nakon toga skače na kamen* 16 *(to je najdalji kamen na koji može da doskoči sa kamena* 9*), nakon toga na kamen* 25 *(to je najdalji kamen na koji može da doskoči sa kamena* 16*) i na kraju na kamen* 32 *(to je najdalji kamen na koji može da doskoči sa kamena* 25*).*



Rešenje dobijeno gramzivom strategijom

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

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

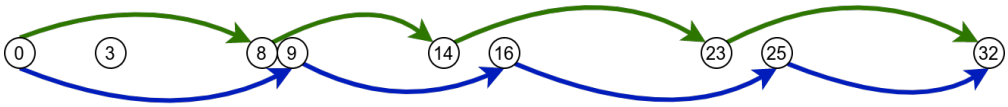
Dodatno je potrebno da dokažemo da je i u slučaju kada funkcija vraća -1, taj odgovor korektan tj. da tada zaista nije moguće da žaba dođe do kraja. To se dešava kada postoje dva kamena sa koordinatama x_i i x_{i+1} takvi da je $x_{i+1} > x_i + r$. Ako bi postojalo neko ispravno rešenje, žaba bi morala da skoči sa nekog kamena

pre i (ili sa kamena i) na neki kamen nakon x_{i+1} (ili na kamen x_{i+1}), no to je nemoguće jer je zbog sortiranosti niza rastojanje svakog takvog para kamenova strogo veće od r.

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

Dokazaćemo da se kamen na kom se žaba nalazi nakon i koraka primene strategije nikada ne nalazi strogo iza kamena na kom se žaba nalazi nakon i koraka u optimalnom rešenju (žaba skače po strategiji je u svakom koraku ili ispred ili na istom kamenu u odnosu na sve žabe koje dostižu cilj u najmanjem broju koraka). Ovo je tipičan dokaz tehnikom *pohlepno rešenje je uvek ispred*.

Primer 9.0.3. *Razmotrimo primer kada je niz kamenja* $0, 3, 8, 9, 14, 16, 23, 25, 32$ *i kada je dužina skoka* $r = 10$. *Rešenje na osnovu strategije je* $0, 9, 16, 25, 32$. *Jedno drugačije optimalno rešenje je* $0, 8, 14, 23, 32$. *Zaista, ni u jednom koraku rešenje na osnovu strategije ne zaostaje za optimalnim rešenjem (a u mnogim koracima prednjači). Važi* $0 = 0$, $9 \geq 8$, $16 \geq 14$, $25 \geq 23$ *i* $32 = 32$.



Rešenje dobijeno gramzivom strategijom prikazano dole, prednjači u odnosu na bilo koje drugo optimalno rešenje prikazano gore

Formalno, neka je optimalna vrednost broja skokova k i i neka je $x_0^*, x_1^*, \ldots, x_{k-1}^*$ sortiran niz x-koordinata kamenja na koje žaba staje u jednom takvom optimalnom rešenju. Neka je $x_0, x_1, \ldots, x_{m-1}$ rešenje dobijeno na osnovu naše strategije. Pošto nijedno rešenje ne može biti bolje od optimalnog, važi da je $k \leq m$. Indukcijom dokazujemo da za svako $i < k$ važi $x_i \geq x_i^*$.

- Bazu čini slučaj $i = 0$ i tada je $x_0 = x_0^*$, jer se žaba u oba slučaja nalazi na početnom kamenu.
- Pod pretpostavkom da važi $x_i \geq x_i^*$ dokazujemo da važi $x_{i+1} \geq x_{i+1}^*$. Ako je $x_i \geq x_{i+1}^*$ (tj. ako je u i-tom koraku gramzivom strategijom žaba već na kamenu na koji dolazi u koraku $i+1$ u optimalnom rešenju ili negde ispred tog kamena), pošto žaba skače samo napred, važi da je $x_{i+1} > x_i \geq x_{i+1}^*$. U suprotnom je $x_i < x_{i+1}^*$ (nakon i koraka gramzive strategije žaba se nalazi

iza kamena na kom se nalazi nakon $i+1$ koraka u otpimalnom rešenju). Pošto je optimalno rešenje korektno, znamo da je $x^*_{i+1} \leq x^*_i + r$. Pošto na osnovu induktivne hipoteze znamo da važi $x_i \geq x^*_i$ važi i $x^*_{i+1} \leq x_i + r$. Dakle, žaba sa kamena x_i može sigurno da doskoči na kamen x^*_{i+1} (on se nalazi ispred kamena x_i, na rastojanju manjem od r), a možda može i dalje. Pošto gramziva strategija uzima uvek najdalji skok, važi da je $x_{i+1} \geq x^*_{i+1}$.

Na osnovu dokazanog važi i da je $x_{k-1} \geq x^*_{k-1}$, međutim, pošto je x^*_{k-1} koordinata poslednjeg kamena, to mora biti i x_{k-1} (pa je $m = k$). Dakle, i optimalna strategija stiže do poslednjeg kamena u k koraka, pa optimalno rešenje nije bolje od pohlepnog.

```
int brojSkokova(const vector<int>& kamenje, int r) {
  // strategija: u svakom koraku skoci sto dalje

  int n = kamenje.size();
  int broj = 0;   // broj skokova
  int kamen = 0;  // kamen na kom se zaba trenutno nalazi
  while (kamen < n - 1) {
    // odredjujemo najdalji kamen na koji mozemo stici od
    // tekuceg
    int noviKamen = kamen;
    // dok god mozes da skocis dalje, skoci dalje
    while (noviKamen + 1 < n &&
           kamenje[noviKamen + 1] - kamenje[kamen] <= r)
      noviKamen++;
    // zaba ne moze da skoci ni na jedan kamen
    if (noviKamen == kamen)
      return -1;
    // zaba skace na najdalji moguci kamen i uvecavamo broj
    // skokova
    kamen = noviKamen;
    broj++;
  }
  // vracamo broj skokova dobijen ovom strategijom
```

```
    return broj;
}
```

U implementaciji se koriste dva pokazivača (`kamen` i `novi_kamen`), koji se kroz niz kamenova kreću samo unapred, tako da je ukupan broj koraka algoritma sigurno ograničen dvostrukom dužinom niza i složenost ovog rešenja je $O(n)$.

Zadatak: Šahovske ekipe

Šahovska ekipa A je pozvala na pripreme šahovsku ekipu B. Svaka ekipa ima isti broj igrača i za svakog igrača je poznat rejting. Ekipa domaćina ima mogućnost da odabere parove koji će igrati u prvom kolu (par čini po jedan igrač iz svake ekipe). Ako svaki igrač domaćina pobeđuje gosta koji ima manji ili jednak rejting, a gubi od gosta koji ima strogo veći rejting, napiši program koji određuje koji je najveći broj pobeda koje ekipa domaćina (ekipa A) može da ostvari u prvom kolu.

Opis ulaza

Sa standardnog ulaza se unosi broj n ($1 \leq n \leq 50000$), a zatim u naredom redu rejtinzi igrača ekipe domaćina (prirodni brojevi) razdvojeni razmacima, a u naredom redu rejtinzi igrača ekipe gostiju (prirodni brojevi) razdvojeni razmacima.

Opis izlaza

Na standardni izlaz ispisati samo jedan broj koji predstavlja najveći mogući broj pobeda domaćina.

Primer

Ulaz	*Izlaz*	*Objašnjenje*

`4`
`2120 1985 2205 1842`
`2045 2100 1990 1980`
	`3`	Domaćin može da ostvari najviše tri pobede. Na primer, igrač 2205 može da dobije igrača 2100, igrač 2120 može da dobije igrača 2045, a igrač 1985 može da dobije igrača 1980.

Rešenje

Zadatak možemo rešiti pomoću nekoliko različitih gramzivih algoritama. Cilj je da odredimo najveći broj parova domaćih i gostujućih igrača u kojima domaći igrač nema manji rejting od svog para. Zato ćemo prilikom raspoređivanja obraćati pažnju samo na one partije u kojima domaćini pobeđuju, dok ćemo ostale partije

zanemariti (potpuno nam je nebitno kako su tačno raspoređeni igrači u partijama u kojima domaćini gube).

Najbolji domaćin protiv najboljeg gosta kojeg može da pobedi

Jedna mogućnost je da se parovi formiraju tako što se upari k najboljih domaćih igrača sa k najlošijih gostujućih igrača (dok ostale igrače uparimo na proizvoljan način). Ta strategija bi bila korektna, ali njena implementacija nije trivijalna, jer nije jasno koliko maksimalno može da bude k tako da u svih k parova domaćini dobijaju.

Slična strategija, koja se jednostavno implementira je sledeća. Ako domaćin može da ostvari bar jednu pobedu, nju može da donese najbolji domaći igrač. Naime, ako bi on izgubio svoj meč, uvek bismo nekog od igrača koji je dobio meč mogli zameniti njime (jer je on bolji od svih igrača domaćina) i dobiti isti broj pobeda. Postavlja se pitanje sa kojim gostujućem igračem on treba da igra. Cilj nam je da nakon formiranja tog para preostanu što lošiji gostujući igrači, da bi slabiji igrači tima domaćina imali šanse da ostvare pobede. Jasno je da u skup parova u kojima domaćini dobijaju ne možemo da uključimo goste koji su bolji od tog najboljeg domaćeg igrača (jer njih niko od igrača domaćina ne može da pobedi). Dobra strategija je da od preostalih gostujućih igrača izaberemo najboljeg. Nakon uparivanja najboljeg domaćeg igrača i gosta sa kojim će on da igra i eliminisanja svih gostiju boljih od njega, problem je sveden na problem istog oblika, ali manje dimenzije (smanjen je broj preostalih domaćih i broj gostujućih igrača koje pokušavamo da uparimo tako da domaćini pobeđuju). Izlaz iz ovog rekurzivnog postupka predstavlja slučaj kada su svi domaći igrači uspešno upareni ili kada među preostalim gostujućim igračima nema lošijih od najboljeg među preostalim domaćinima.

Dokažimo i formalno korektnost ovakve gramzive strategije.

Rešenje koje prethodni algoritam daje je korektno uparivanje i zadovoljava uslove zadatka jer se svaki domaćin uparuje sa gostom koja nije bolji od njega (to se eksplicitno proverava) i nije ni uparen ni sa jednim drugim domaćinom (jer se nakon uparivanja i domaćin i gost eliminišu iz daljeg razmatranja). Dakle, sigurni smo da zaista postoji korektno uparivanje u kome domaćin ostvaruju broj prijavljen broj pobeda.

Pokažimo i da naša strategija pravi optimalni broj pobeda za domaću ekipu. Dokaz će ići tehnikom razmene, tj. time što ćemo se pokazati da se optimalno uparivanje može transformisati u ono dobijeno gramzivom strategijom, održavajući ukupan

broj parova u kojima domaćin pobeđuje (za igrače domaćina koji pobeđuju reći ćemo da su *dobitno upareni*). Posmatrajmo neko optimalno uparivanje. Neka je d_i najbolji domaćin koji učestvuje u njemu i neka je g_i gost sa kojim je on uparen.

Ako on nije ukupno najbolji domaćin d_s, tada najbolji domaćin sigurno nije dobitno uparen. Možemo domaćina d_i izbaciti iz dobitnog uparivanja i njemu pridruženog gosta g_i pridružiti ukupno najboljem domaćinu d_s (to je moguće jer je $d_s \geq d_i \geq g_i$). Takvo uparivanje je i dalje optimalno (jer se broj dobitnih parova za domaćina nije promenio).

Neka je g_s gost koja bi bio odabran strategijom (najbolji gost koji nije bolji od d_s, tj. najbolji gost za koga važi $d_s \geq g_s$).

- Ako on nije deo trenutnog dobitnog uparivanja, onda gosta g_i koji trenutno igra sa domaćinom d_s možemo izbaciti i zameniti gostom g_s (to je moguće jer je $d_s \geq g_s$).
- Ako jeste raspoređen tako da gubi od nekog domaćina d_j, onda možemo napraviti razmenu tako da d_s igra sa g_s, a d_j sa g_i. Dokažimo da je ovo i dalje korektno uparivanje. Važi da je $d_s \geq g_s$ i $d_s \geq g_i$. Pošto je g_s najbolji gost koga d_s može da pobedi, važi da je $g_s \geq g_i$. Zato je $d_j \geq g_s \geq g_i$. Sa ove dve eventualne razmene dobijamo i dalje optimalan raspored koji je u skladu sa našom strategijom što se tiče prvog para.

Nastavljajući razmene po istom principu (tj. na osnovu induktivnog argumenta), uparivanje možemo transformisati u ono formirano našom strategijom, zadržavajući sve vreme optimalnost.

Prilikom implementacije, skupove domaćih i gostujućih igrača možemo čuvati u nizovima uređenim u nerastućem redosledu rejtinga i algoritam možemo realizovati tehnikom dva pokazivača (prodiskutovaćemo kasnije i ostale moguće varijante). Niz domaćina obilazimo redom, element po element, a niz gostiju razdvajamo na one koje su eliminisani (one koji su do sada upareni i one koje nisu upareni, ali su bolji od tekućeg domaćeg igrača) i preostale. Održavamo mesto početka niza gostiju koji još nisu obrađeni i prilikom traženja para za tekućeg domaćeg igrača niz gostiju obilazimo od te pozicije. Svakog gosta ili eliminišemo, jer je bolji od tekućeg domaćeg igrača ili ga uparujemo sa tekućim domaćim igračem i onda ih obojicu eliminišemo. Naglasimo da se u implementaciji ne moramo vraćati na eliminisane goste, jer ako je neki gost bolji od tekućeg domaćina (najboljeg među preostalim), biće bolji i od svih narednih (preostalih) domaćina.

Pošto se oba pokazivača kreću samo u jednom smeru i složenost faze uparivanja je linearna. Ukupnim algoritmom, dakle, dominira složenost sortiranja, pa je ukupna složenost $O(n \log n)$.

Moguće je formulisati i gramzivu strategiju koja će biti dualna ovoj upravo opisanoj. U toj gramzivoj strategiji obrađujemo goste u neopadajućem redosledu rejtinga (od lošijih ka boljima) i svakom gostu dodeljujemo najlošijeg domaćina koji može da ga pobedi. Analogno prethodnoj, jednostavno se dokazuje da je i ta gramziva strategija takođe korektna.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // Ako domacini mogu da ostvare nekih k pobeda, onda tih k
  // pobeda moze da ostvari njihovih k najjacih igraca. Zato
  // domacine obradjujemo u nerastucem redosledu rejtinga i za
  // svakog redom odredjujemo gosta kojeg moze da pobedi. Za
  // svakog domacina odredjujemo najboljeg gosta kojeg moze da
  // pobedi jer time losijim igracima domace ekipe ostavljamo
  // prostor da pobede nekoga.

  // broj igraca
  int n = domaci.size();
  // zelimo da i domacine i goste obilazimo u nerastucem
  // redosledu rejtinga (od boljih ka losijim)
  sort(begin(domaci), end(domaci), greater<int>());
  sort(begin(gosti), end(gosti), greater<int>());
  // broj pobeda domacih igraca
  int brojPobeda = 0;
  // tekuci indeks domaceg i gostujuceg igraca
  int d = 0, g = 0;
  while (true) {
    // trazimo najboljeg gosta kojeg moze da pobedi domacin na
    // poziciji d goste koji su jaci od trenutno najaceg
    // domacina ne moze da pobedi niko od preostalih domacina,
    // pa ih eliminisemo iz daljeg razmatranja
    while (g < n && domaci[d] < gosti[g])
      g++;
    // ako najaci nerasporedjeni domacin ne moze da pobedi
```

```
    // nijednog gosta, ne mozemo povecati broj pobeda
    if (g >= n) break;
    // u suprotnom smo nasli gosta g kojeg uparujemo sa
    // domacinom d
    brojPobeda++;
    // obojicu eliminisemo iz daljeg uparivanja
    g++, d++;
  }
  return brojPobeda;
}
```

Neefikasna implementacija

Skrenimo pažnju i na važnost efikasne implementacije. Razmotrimo rešenje zasnovano na dualnoj strategiji u kojoj uparujemo loše goste. Kao što smo videli, u efikasnoj implementaciji bi prilikom prelaska na svakog novog gosta domaćina trebalo tražiti samo među onima koji u ranijim koracima nisu eliminisani (bilo tako što su upareni ili tako što je ustanovljeno da ne mogu da pobede nekog od slabijih gostiju). Ako pretragu domaćina svaki put počinjemo iz početka (vodeći računa o tome da ranije uparene domaćine ne uparujemo ponovo, tako što u posebnom nizu registrujemo one domaćine koje smo već uparili) dobićemo neefikasan algoritam.

Pošto se za svakog od n gostiju iznova pretražuje niz od n domaćina, složenost najgoreg slučaja ove implementacije je $O(n^2)$. Naglasimo da nije problem u gramzivoj strategiji, već u njenoj lošoj implementaciji.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // broj igraca
  int n = domaci.size();
  // Ako je moguce pobediti nekih k gostiju, onda tih k
  // gostiju mogu biti k najlosijih gostiju. Zato goste
  // obradjujemo u rastucem redosledu rejtinga (od losijih ka
  // boljima) i za svakog redom odredjujemo domacina koji moze
  // da pobedi tog gosta, a nije ranije uparen.
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  // broj pobeda domacih igraca
```

```
  int brojPobeda = 0;
  // da li je domacin trenutno uparen
  vector<bool> zauzet(n, false);
  // obilazimo sve goste
  for (int g = 0; g < n; g++)
    // trazimo najslabijeg neuparenog domacina koji moze
    // pobediti gosta g
    for (int d = 0; d < n; d++)
      if (!zauzet[d] && domaci[d] >= gosti[g]) {
        brojPobeda++;
        zauzet[d] = true;
        break;
      }

  return brojPobeda;
}
```

Raspoređivanje svakog domaćina sa najlošijim ili najboljim gostom

Još jedna varijanta naše pobedničke strategije (tj. njene dualne varijante) obilazi sve domaćine i goste u neopadajućem redosledu rejtinga i ako domaćin može da pobedi najslabijeg trenutno neraspoređenog gosta, onda ga uparujemo sa njim, a u suprotnom ga uparujemo sa najjačim gostom (jer on ne može pobediti nikoga od preostalih gostiju, a mora biti uparen sa nekim, pa je najbolje upariti ga sa najjačim gostom za kog je sada izvesno da niko ne može da ga pobedi). Na taj način ne dobijamo samo broj pobeda, već i efektivno uparivanje svih igrača u kom se postiže taj maksimalni broj pobeda.

Tu strategiju možemo implementirati tako što čuvamo skup preostalih domaćina i gostiju, pronalazimo najmanji tj. najveći element u skupu i uklanjamo ih. Ako se skup implementira preko niza (vektora, liste), dobijamo veoma neefikasan algoritam.

Ovaj algoritam je složenosti $O(n^2)$ (jer se i pronalaženje minimuma i maksimuma i uklanjanje elementa sa date pozicije vrši u linearnoj složenosti).

Program postaje mnogo efikasniji ako upotrebimo bibliotečke kolekcije koje nam pružaju efikasniju implementaciju skupa (koja dozvoljava efikasno traženje i uklanjanje minimalnog i maksimalnog elementa). U jeziku C++ možemo upotrebiti

multiskupove (jer možda postoji više igrača sa istim rejtingom) koje na raspolaganju imamo kroz kolekciju `multiset`. Pošto je multiskup uređen iterator `begin()` ukazuje na najmanji element, a iterator `prev(end())` na najveći element. Uklanjanje elementa možemo izvršiti pomoću metoda `erase`.

Pod pretpostavkom da se operacije sa multiskupom vrše u logaritamskoj složenosti u odnosu broj elemenata u multiskupu, ovaj algoritam će biti složenosti $O(n \log n)$.

```
int maksBrojPobeda(multiset<int>& domaci, multiset<int>& gosti) {
  int brojPobeda = 0;

  // sve dok ne rasporedimo sve domace igrace
  while (domaci.size() > 0) {
    // rasporedjujemo najboljeg domacina
    int najmanjidom = *domaci.begin();
    domaci.erase(domaci.begin());
    // sa najlosijim gostom ako moze da ga pobedi ili sa
    // najboljim gostom ako ne moze
    int najmanjigost = *gosti.begin();
    if (najmanjidom >= najmanjigost) {
      brojPobeda++;
      gosti.erase(gosti.begin());
    } else {
      gosti.erase(prev(gosti.end()));
    }
  }

  return brojPobeda;
}
```

Ipak najefikasnije rešenje dobijamo ako multiskupove predstavimo sortiranim nizom, a brisanje ne vršimo efektivno, već samo čuvamo pokazivač na trenutno neobrađenog domaćina, dok u skupu gostiju neobrađene goste čuvamo između dva pokazivača.

```
int maksBrojPobeda(vector<int>& domaci, vector<int>& gosti) {
  // broj igraca
```

```
  int n = domaci.size();
  // broj pobeda domacina
  int brojPobeda = 0;
  // sortiramo oba tima u neopadajucem redosledu rejtinga (od
  // najslabijih do najboljih)
  sort(begin(domaci), end(domaci));
  sort(begin(gosti), end(gosti));
  // pozicija najlosijeg i najboljeeg nerasporedjenog gosta
  // svi gosti iz intervala [0, gLos) i (gDobar, n) su vec
  // upareni
  int gLos = 0, gDobar = n-1;
  // rasporedjujemo sve domace igrace
  for (int d = 0; d < n; d++) {
    // rasporedjujemo najboljeg domacina sa najlosijim gostom
    // ako moze da ga pobedi ili sa najboljim gostom ako ne
    // moze
    if (domaci[d] >= gosti[gLos]) {
      // uparujemo domacina d i gosta gLos
      brojPobeda++;
      gLos++;
    } else {
      // uparujemo domacina d i gosta gDobar
      gDobar--;
    }
  }
  return brojPobeda;
}
```

Zadatak: Raspored aktivnosti

U jednom kabinetu se subotom održava obuka programiranja. Svaki nastavnik je napisao termin u kom želi da drži nastavu (poznat je sat i minut početka i sat i minut završetka časa). Odredi kako je moguće napraviti raspored časova tako da što više nastavnika bude uključeno.

Opis ulaza

Sa standardnog ulaza se učitava prvo broj n (ukupan broj nastavnika, $1 \leq n \leq$

50000), a zatim u n narednih redova po četiri broja razdvojena razmacima koji predstavljaju sat i minut početka tj. završetka časa (pretpostaviti da je završetak uvek iza početka).

Opis izlaza

Na standardni izlaz ispisati najveći broj nastavnika koji mogu da održe svoje časove.

Primer

Ulaz	*Izlaz*	*Objašnjenje*

`7`
`8 15 9 20`
`10 45 11 30`
`11 20 12 45`
`9 30 12 40`
`10 20 11 20`
`12 00 13 00`
`11 30 13 30`
	`3`	Mogu se održati, na primer, časovi od 8:15 do 9:20, zatim čas od 10:20 do 11:20 i na kraju od 11:30 do 13:30.

Rešenje

Svaki čas možemo predstaviti parom brojeva koji predstavljaju broj minuta od prethodne ponoći do početka i do kraja časa (već prilikom učitavanja sate i minute možemo prevesti samo u minute).

Iscrpna pretraga

Naivno rešenje se zasniva na ispitivanju svih mogućih podskupova skupa časova koji su takvi da se svi časovi mogu održati (nikoja dva časa iz tog skupa se ne seku). Presek dva intervala postoji ako i samo ako je kasniji početak časa posle ranijeg kraja časa. Generisanje svih podskupova možemo vršiti rekurzivno.

S obzirom na veliki broj podskupova koje treba ispitati ovo rešenje je veoma neefikasno (složenost mu je eksponencijalna $O(2^n)$, gde je n ukupan broj časova).

Gramzivi pristup

Efikasno rešenje problema se može dobiti gramzivim pristupom. Postoji nekoliko gramzivih strategija koje je logično razmotriti, međutim, neke od njih neće garantovati optimalnost pronađenog rešenja.

Jedan pristup može biti onaj u kome prvo raspoređujemo čas koji prvi počinje. Na slici je prikazan kontra-primer, koji pokazuje da se već sa tri časa na taj način može dobiti raspored koji nije optimalan.

Jedan pristup može biti onaj u kome težimo da rasporedimo časove koji kratko traju, sa idejom da na taj način ostavljamo više prostora da se u slobodnim terminima održe drugi časovi. Na slici je prikazan kontra-primer koji pokazuje da se već sa tri časa na taj način može dobiti raspored koji nije optimalan.

Primer na kome strategija koja raspoređuje čas koji prvi počinje i primer na kome strategija koja raspoređuje čas koji je najkraći daje neoptimalno rešenje

Jedna gramziva strategija koja daje optimalno rešenje je sledeća. Od svih neraspoređenih časova biramo onaj koji se najranije završava i koji se može održati (ne seče se sa do sada održanim časovima, tj. počinje nakon završetka prethodnog časa). Intuitivno, takvim izborom ostavljamo što veću mogućnost za raspoređivanje naknadnih časova. Ovim dobijamo jednu rekurzivnu konstrukciju.

- Ako je skup časova prazan, nema časova koji se mogu rasporediti.
- U suprotnom biramo čas koji se najranije završava, odbacujemo časove koji se sa njim seku i rekurzivnom pravimo raspored za preostale časove.

Dokažimo da je ova rekurzivna formulacija korektna, tako što ćemo da dokažemo da uvek postoji ispravan, optimalan raspored (raspored sa najviše časova) u kom učestvuje čas c_0 koji se prvi završava. Pretpostavimo da je O neki ispravan, optimalan raspored. Ako u njemu učestvuje čas c_0, onda je O je taj traženi raspored. Ako ne učestvuje, onda pretpostavimo da je o_0 čas u O koji se prvi završava. Svi drugi časovi u O počinju nakon završetka časa o_0. Zaista, pošto je O ispravan, nijedan čas u o_i se ne seče sa o_0, ako bi neki počinjao pre o_0, tada bi morao i da se završi pre o_0, što je nemoguće, jer se od svih časova u O čas o_0 prvi završava. Čas c_0 se ne završava kasnije nego čas o_0, jer je on čas koji se prvi završava od svih časova. Dakle, čas c_0 se ne seče ni sa jednim časom u O (osim eventualno sa o_0). Kada zamenimo c_0 i o_0, dobijamo ispravan, optimalan raspored (broj časova se nije promenio) koji sadrži c_0.

Dakle, jasno je da će nas izbor časa c_0 voditi do nekog optimalnog rešenja. Takođe je jasno da u tom rešenju ne može da učestvuje nijedan čas koji se seče sa c_0. Ostatak časova biramo rekurzivno, pa korektnost algoritma sledi na osnovu induktivnog argumenta (pretpostavljamo da rekurzivni poziv korektno pronalazi optimalni raspored u preostalom skupu časova).

Tehnika koja je upotrebljena u prethodnom dokazu naziva se tehnika zamene tj. razmene, jer se od nekog proizvoljnog optimalnog rasporeda zamenom dobio raspored koji naša gramziva strategija bira.

Algoritam se može formulisati i iterativno. Časove možemo sortirati neopadajuće na osnovu vremena njihovog završetka i obilaziti ih u tom redosledu. Prvi čas sigurno biramo da bude održan. Redom prolazimo kroz naredne časove i ako tekući čas počinje nakon poslednjeg odabranog časa, biramo ga, a u suprotnom ga preskačemo.

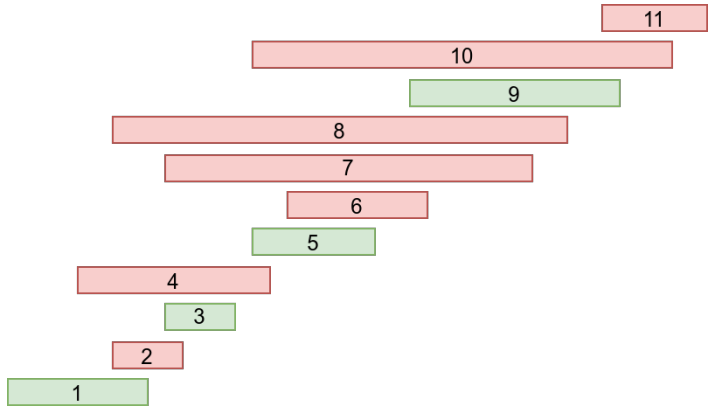
Primer 9.0.4. *Na slici su prikazani časovi sortirani po vremenu završetka. Gramzivom strategijom se prvo održava čas 1, zatim se čas 2 preskače (jer se preklapa sa 1), pa se zatim čas 3 održava (jer se ne preklapa sa 1, jer je desno od njega), pa se čas 4 preskače (jer se preklapa sa 3), pa se čas 5 održava (jer se ne preklapa sa 3, pa samim tim ni sa 1, jer je desno od njih), pa se časovi 6, 7 i 8 preskaču (jer se preklapaju sa časom 5), pa se čas 9 održava (jer se ne preklapa sa 5, pa samim tim ni sa 3 ni sa 1, jer je desno od njih) i na kraju se preskaču časovi 10 i 11 (jer se preklapaju sa časom 9). Maksimalan broj časova koji se mogu održati bez preklapanja je 4, međutim, časovi 1, 3, 5 i 9 nisu jedino rešenje. Moguće je, na primer, održati časove 1, 3, 6 i 11.*

Ispišimo i dokaz korektnosti iterativne varijante algoritma.

Formalno, pretpostavimo da je $O = [o_1, o_2, \dots, o_k]$, niz časova koji predstavlja neko ispravno optimalno rešenje i dokažimo da on sadrži isti broj časova kao i raspored koji bi odabrala naša strategija. Pretpostavimo da su časovi o_1 do o_k sortirani neopadajuće po redosledu njihovog završetka. Pošto se svi ti časovi mogu održati, između njih nema preklapanja i svaki naredni počinje nakon završetka prethodnog.

Neka je $S = [s_1, s_2, \dots, s_{k'}]$ niz časova koji bi bio odabran našom gramzivom strategijom. Pošto je O optimalan, S ne može da sadrži više časova od njega važi da je $k \leq k'$.

Dokažimo prvo da postoji optimalan raspored O takav da za svako $1 \leq i \leq k'$ važi da je $o_i = s_i$. Taj raspored možemo dobiti postupnim izmenama početnog



Rezultat primene gramzive strategije

rasporeda O. Pretpostavimo da postoji neko $1 \leq i \leq k'$ tako da je $o_i \neq s_i$ (u suprotnom je polazni raspored taj traženi). Neka je i prvi takav indeks tj. neka za svako $1 \leq j < i$ važi da je $o_j = s_j$. Pokažimo da se zamenom časa o_i časom s_i u nizu O dobija takođe raspored koji je ispravan (on je svakako optimalan jer se broj časova ne menja). Pokažimo prvo da se s_i ne završava kasnije nego o_i.

- Zaista, ako je $i = 0$, tada naša strategija bira s_0 koji se prvi završava on ne može da se završava kasnije nego o_0.
- Ako je $i > 0$, tada znamo da se o_i mora da počinje posle $o_{i-1} = s_{i-1}$, međutim, naša strategija za s_i bira onaj čas koji počinje nakon s_{i-1} koji se prvi završava, pa se s_i ni u ovom slučaju ne može završavati kasnije nego o_i.

Ako postoje časovi u O pre časa o_i, oni ostaju nepromenjeni i čas s_i se ne preklapa sa njima (jer ga je strategija bira tako da počinje nakon završetka časa $s_{i-1} = o_{i-1}$). Pošto se s_i ne završava kasnije nego o_i on se sigurno ne preklapa ni sa jednim časom iz O koji ide posle o_i (jer svi oni počinju i završavaju se nakon kraja časa o_i). Dakle, s_i se ne preklapa ni sa jednim časom iz O (osim eventualno sa o_i, koji je u sklopu razmene uklonjen) i raspored dobijen zamenom je ispravan.

Nastavkom ovog procesa zamena stići ćemo do željenog optimalnog rasporeda O takvog da za svako $1 \leq i \leq k'$ važi $o_i = s_i$.

Dokažimo sada da nije moguće da važi da je $k' < k$. Ako bi važilo da je $k' < k$, tada bi važilo da čas $o_{k'+1}$ pripada O a ne pripada S. Pošto je O ispravan, to bi

bio čas koji bi počinjao posle završetka časa $o_{k'} = s_{k'}$. Međutim, nije moguće da takav čas postoji, jer bi on počinjao i završavao se posle časa s'_k, što znači da bi naša gramziva strategija morala da ga odabere, što je u kontradikciji sa tim da se on ne nalazi u S. Dakle, važi da je $k' \geq k$.

Pošto je $k' \geq k$ i $k' \leq k$, važi da je $k' = k$, pa naša gramziva strategija bira isti broj časova koji je u nekom (pa i svakom) optimalnom rasporedu. Dakle, strategijom se dobija jedan ispravan, optimalan raspored.

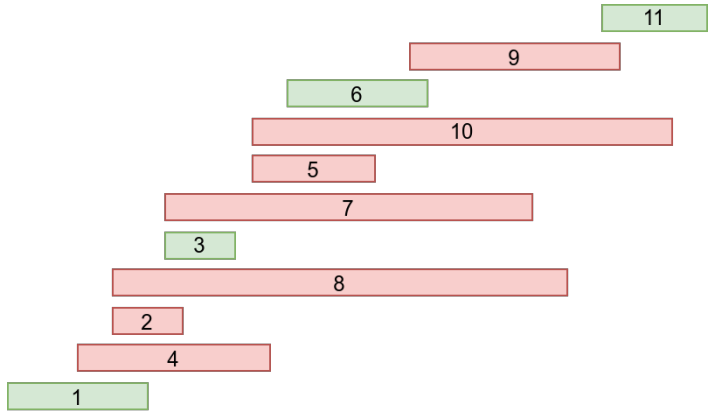
Primer 9.0.5. *Ilustrujmo tehniku razmene na kojoj leži prethodni dokaz, tako što ćemo objasniti kako da se od rasporeda* $1, 3, 6, 11$ *koji je optimalan, ali nije u skladu sa našom strategijom dobije raspored* $1, 3, 5, 9$ *koji jeste u skladu sa našom strategijom.*

- *Prvi čas gde se rasporedi razlikuju je čas* 5 *tj.* 6*. Zamenom časa* 6*, časom* 5 *dobija se raspored* $1, 3, 5, 11$*, koji je takođe ispravan. Zaista, čas* 6 *se ne seče ni sa časovima* 1 *i* 3*, ni sa časom* 11*, jer je raspored* $1, 3, 6, 11$ *ispravan. Čas* 5 *se ne seče sa časovima* 1 *i* 3*, jer ga u suprotnom gramziva strategija ne bi odabrala. Međutim, čas* 5 *se ne može završavati posle časa* 6*, jer se čas* 6 *ne seče sa časovima* 1 *i* 3*, a od svih časova koji se ne seku sa časovima* 1 *i* 3*, strategija bira onaj koji se najranije završava. Dakle,* 5 *se ne završava kasnije od* 6*, pa pošto se* 6 *ne seče sa* 11 *i pošto je* 11 *potpuno desno u odnosu na* 6*, ni* 5 *se ne seče sa* 11*. Dakle, raspored* $1, 3, 5, 11$ *je ispravan.*
- *U narednom koraku se upoređuju rasporedi* $1, 3, 5, 11$ *i* $1, 3, 5, 9$ *(koji naša strategija preporučuje). Prvi čas gde se rasporedi razlikuju je čas* 9 *tj.* 11*. Zamenom časa* 11*, časom* 9 *dobija se raspored* $1, 3, 5, 9$*, koji strategija preporučuje.*

Dakle, proizvoljni optimalni raspored smo razmenama transformisali u raspored koji preporučuje naša strategija, što znači da je broj časova koje naša strategija rasporedi za održavanje optimalan.

Recimo da postoji i dualno rešenje u kojem se bira onaj čas koji poslednji počinje i časovi se obilaze unazad, po nerastućem redosledu njihovog početka.

Primer 9.0.6. *Na slici su prikazani časovi sortirani po redosledu početka. Prvo se održava čas 11, koji poslednji počinje, zatim se preskače čas 9 koji se sa njim preklapa, zatim se održava čas 6, nakon čega se preksaču časovi 10, 5 i 7 koji se sa*



Rezultat primene dualne gramzive strategije

njim preklapaju, zatim se održava čas 3, preskaču se časovi 8, 2 i 4 koji se sa njim preklapaju i na kraju se održava čas 1.

```
// casove predstavljamo uredjenim parovima (pocetak, kraj), u
// minutima
typedef pair<int, int> cas;

inline int pocetakCasa(const cas& c) {
  return c.first;
}

inline int krajCasa(const cas& c) {
  return c.second;
}

cas napraviCas(int pocSat, int pocMin,
               int krajSat, int krajMin) {
  return make_pair(pocSat*60 + pocMin, krajSat*60 + krajMin);
}

// ucitava se niz casova
vector<cas> ucitajCasove() {
```

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

// maksimalni broj casova koji se mogu odrzati tako da nema
// preklapanja medju rasporedjenim casovima
int maksBrojCasova(vector<cas>& casovi) {
  // broj casova
  int n = casovi.size();

  // sortiramo casove na osnovu vremena zavrsetka
  sort(begin(casovi), end(casovi),
       [](const cas& a, const cas& b) {
         return krajCasa(a) < krajCasa(b);
       });

  // broj odrzanih casova
  int brojOdrzanihCasova;
  // kraj poslednjeg odrzanog casa
  int kraj;

  // rasporedjujemo prvi cas
  brojOdrzanihCasova = 1;
  kraj = krajCasa(casovi[0]);

  // analiziramo ostale casove u redosledu zavrsetka
  for (int i = 1; i < n; i++)
    // ako se tekuci cas ne preklapa sa poslednjim
    // rasporedjenim
```

```
    if (pocetakCasa(casovi[i]) >= kraj) {
      // on se odrzava
      brojOdrzanihCasova++;
      kraj = krajCasa(casovi[i]);
    }

  return brojOdrzanihCasova;
}
```

Zadatak: Raspored sa najmanjim brojem učionica

Za svaki od n časova poznato je vreme početka i završetka. Kada se u nekoj učionici završi jedan čas, istog trenutka u njoj može da počne neki drugi čas. Napiši program koji određuje minimalan broj učionica potreban da se svi časovi održe.

Opis ulaza

Sa standardnog ulaza se učitava broj n ($1 \leq n \leq 10^5$), a zatim u n narednih redova podaci o n časova (sat i minut početka i sat i minut završetka, pri čemu je vreme početka strogo manje od vremena završetka).

Opis izlaza

Na standardni izlaz ispisati traženi broj potrebnih učionica.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
`6`	`2`	Jedan mogući raspored navedenih 6 časova u dve učionice je sledeći:
`8 0 8 45`		`Učionica 1` `Učionica 2`
`10 0 10 45`		
`9 0 9 45`		`8:00-8:45 Čas 1` `8:30-9:15 Čas 4`
`8 30 9 15`		`9:00-9:45 Čas 3` `10:30-11:15 Čas 6`
`10 45 11 30`		`10:00-10:45 Čas 2`
`10 30 11 15`		`10:45-11:30 Čas 5`

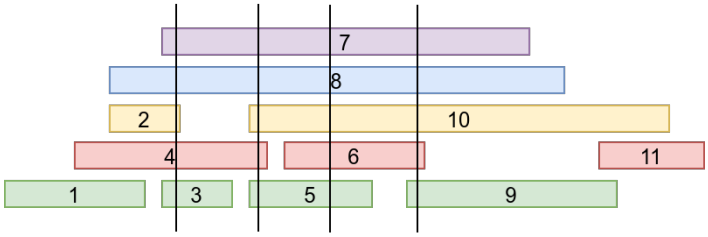
Rešenje

Pošto se zahteva da se svi časovi održe, obilazićemo ih u određenom redosledu i svaki čas ćemo pridruživati nekoj od slobodnih učionica. U trenutku u kom poči-

nje neki čas i u kome nema više slobodnih učionica koje su ranije upotrebljavane, otvaraćemo novu učionicu u koju ćemo raspoređivati taj čas.

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

Primer 9.0.7. *Na slici je prikazan jedan mogući raspored časova. Učionice se nalaze jedna iznad druge (svi časovi u istoj učionici su prikazani na istoj visini). Vertikalnim linijama su označeni trenuci u kojima su sve učionice popunjene tj. trenuci u kojima postoji 5 časova koji se preklapaju. Zbog toga je jasno da nije moguće napraviti raspored sa manje od 5 učionica.*



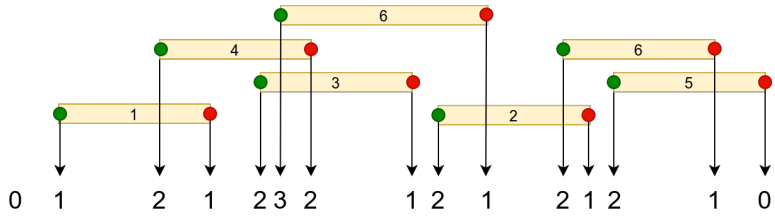
Raspored časova

Jedan mogući redosled obrade časova je rastući (neopadajući) redosled njihovih početaka.

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

Primetimo da je u ovom dokazu određena granica kvaliteta rešenja (raspored ne može biti u manje učionica nego što je broj časova u najvećoj grupi časova koji se održavaju u nekom trenutku) i zatim je pokazano da pohlepno rešenje dostiže tu granicu, pa je zbog toga optimalno.

Ako nas zanima samo broj potrebnih učionica, a i konkretan raspored, tada možemo napraviti veoma jednostavnu implementaciju koja održava broj trenutno otvorenih učionica i obilazi sve značajne vremenske trenutke (početke i završetke časova) redom. Na kraju časa smanjuje se broj zauzetih učionica, a na početku časa povećava se broj učionica. Traženi rezultat se dobija kao maksimalna vrednost brojača zauzetih učionica.



Obilazak svih karakterističnih tačaka uz ažuriranje brojača

Potrebno je još obratiti pažnju na specijalan slučaj kada se neki časovi završavaju u istom trenutku u kom drugi časovi počinju. Elegantno rešenje je da u datom vremenskom trenutku prvo obradimo sve časove koji se završavaju u tom trenutku i oslobodimo učionice, a zatim da obradimo časove koji počinju u tom trenutku (tako nove časove raspoređujemo u upravo oslobođene učionice, što je u skladu sa tim da je vreme trajanja svakog časa poluotvoreni interval $[s, f)$).

Za n časova postoji $2n$ karakterističnih trenutaka. Njihovo sortiranje se vrši u vremenu $O(n \log n)$. Obilazak sortiranog niza trenutaka i ažuriranje brojača se zatim vrši u vremenu $O(n)$. Složenošću, dakle, dominira sortiranje i ukupna složenost je $O(n \log n)$.

```
// karakteristicni trenuci
struct Vreme {
  // vreme izrazeno u minutima
  int minut;
  // da li je u pitanju pocetak ili kraj casa
  bool pocetak;
```

```
};
// funkcija rasporedjuje casove u ucionice, vraca minimalni
// potrebni broj ucionica i svakom casu dodeljuje broj
// ucionice
int minUcionica(vector<Vreme>& vremena) {
  sort(begin(vremena), end(vremena),
       [](const Vreme& v1, const Vreme& v2) {
         return v1.minut < v2.minut ||
           (v1.minut == v2.minut && !v1.pocetak && v2.pocetak);
       });
  // trenutan broj zauzetih ucionica
  int brojUcionica = 0;
  // maksimalni broj zauzetih ucionica u nekom trenutku
  int maksBroj = 0;
  for (const Vreme& v : vremena)
    if (v.pocetak)
      maksBroj = max(++brojUcionica, maksBroj);
    else
      brojUcionica--;
  return maksBroj;
}
```

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

vreme završetka časa postavljeno na vreme završetka tekućeg časa (broj učionice je za jedan veći od dotadašnjeg broja učionica u redu).

Broj otvorenih učionica je uvek jednak broju elemenata reda. Naime, element u red dodajemo samo kada otvaramo novu učionicu jer su sve do tada otvorene učionice u redu zauzete, dok u suprotnom samo menjamo element koji je bio na vrhu reda novim (časovi koji su se završili a nisu zamenjeni novim časovima u istoj učionici ostaju u redu).

Ukupna složenost algoritma je $O(n \log n)$ – i u fazi sortiranja i u fazi raspoređivanja (jer se čitanje i izbacivanje minimuma, kao i ubacivanje novog elementa u red sa prioritetom vrši u vremenu $O(\log n)$). Kada bismo umesto reda sa prioritetom koristili običan niz i u njemu stalno tražili minimum, složenost najgoreg slučaja bi porasla na $O(n^2)$.

```
struct Cas {
  // redni broj casa (njegov jedinstveni identifikator)
  int broj;
  // minut pocetka i kraja casa
  int pocetak, kraj;
  // redni broj ucionice u kojoj se cas odrzava
  int ucionica;
};
struct Ucionica {
  // broj ucionice (njen jedinstveni identifikator)
  int broj;
  // minut od kog je ucionica slobodna
  int slobodnaOd;
};
// funkcija rasporedjuje casove u ucionice, vraca minimalni
// potrebni broj ucionica i svakom casu dodeljuje broj
// ucionice
int minUcionica(vector<Cas>& casovi) {
  // sortiramo časove na osnovu vremena njihovog početka
  sort(begin(casovi), end(casovi),
       [](const Cas& c1, const Cas& c2) {
         return c1.pocetak < c2.pocetak;
       });
```

```
  // red sa prioritetom u kom su trenutno zauzete ucionce
  // slozene po vremenu zavrsetka casa
  struct PorediUcionice {
    bool operator()(const Ucionica& u1, const Ucionica& u2) {
      return u1.slobodnaOd > u2.slobodnaOd;
    }
  };
  priority_queue<Ucionica, vector<Ucionica>, PorediUcionice>
    redUcionica;
  for (Cas& c : casovi) {
    int brojUcionice;
    if (redUcionica.empty() ||
        redUcionica.top().slobodnaOd > c.pocetak)
      brojUcionice = redUcionica.size() + 1;
    else {
      brojUcionice = redUcionica.top().broj;
      redUcionica.pop();
    }
    c.ucionica = brojUcionice;
    redUcionica.push(napraviUcionicu(c.kraj, brojUcionice));
  }

  // sortiramo časove na osnovu rednog broja
  sort(begin(casovi), end(casovi),
       [](const Cas& c1, const Cas& c2) {
         return c1.broj < c2.broj;
       });

  return redUcionica.size();
}
```

Zadatak: Isplata sa posebnim novčićima

U Srbiji se koriste apoeni od 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000 i 5000 dinara. Napisati program koji formira dati iznos dinara od što manjeg broja apoena

(novčanica i novčića) i ispisuje upotrebljen broj apoena.

Opis ulaza

Sa standardnog ulaza se učitava jedan ceo broj između 0 i 100 000.

Opis izlaza

Na standardni izlaz napisati najmanji broj apoena potrebnih da se isplati učitani iznos novca.

Primer

Ulaz	*Izlaz*	*Objašnjenje*
243	5	$243 = 200 + 20 + 20 + 2 + 1$.

Rešenje

Jedan direktan način da se problem reši je da se ispitaju sva moguća razlaganja datog iznosa na zbirove sastavljene od ovih brojeva i da se pronađe onaj zbir koji ima najmanji broj novčića (jednostavnosti radi, zanemarimo razliku između novčića i novčanica). Rešenje ovog tipa bi se moglo zasnovati na dinamičkom programiranju kombinovanom sa odsecanjem tokom pretrage.

Dokaz korektnosti nije težak i ovaj pristup je neophodno primeniti kada su novčići proizvoljni. Međutim, u situaciji u kojoj su dati veoma specifični apoeni, ovaj pristup je nepotrebno komplikovan i neefikasan. Naime specifičnosti apoena omogućavaju da se zadatak reši mnogo efikasnije.

```
// najmanji broj novčića potreban da se naplati iznos S kada
// su nam na raspolaganju n vrednosti novčića datih u nizu v
int minBrojNovcica(const vector<int>& v, int n, int S) {
  vector<int> dp(S+1);
  // iznos 0 se naplaćuje sa 0 novčića
  dp[0] = 0;
  // računamo minimalni broj novčića za sve ostale iznose
  for (int s = 1; s <= S; s++) {
    // minimalni broj novčića da se naplati iznos s
    // (pretpostavljamo da iznos nije moguće naplatiti)
    dp[s] = INF;
    // razmatramo sve mogućnosti za poslednji novčić
    for (int i = 0; i < n; i++)
```

```
      // proveravamo da li je iznos s moguće naplatiti
      // novčićem i
      if (v[i] <= s)
        // ažuriramo minimum ako je to potrebno
        dp[s] = min(dp[s], dp[s-v[i]] + 1);
  }
  // vraćamo rezultat za iznos S
  return dp[S];
}
```

I bez formalnog matematičkog objašnjenja, svaki prodavac u prodavnici i na pijaci zna da se optimalno rešenje dobija tako što se u svakom trenutku vraća najveći apoen koji je manji ili jednak od trenutnog iznosa i nakon toga se isti princip primenjuje na preostali iznos sve dok se ne vrati ceo kusur (u pitanju je, dakle, gramziva induktivno-rekurzivna konstrukcija). Ovo rešenje je veoma efikasno, lako se implementira, međutim, dokaz njegove korektnosti nije nimalo očigledan.

Naime, postojanje novčića od 4 dinara bi pokvarilo situaciju. 8 dinara bi se moglo dobiti od dva novčića od 4 dinara, dok bi gramziva strategija upotrebila tri novčića (od 5, 2 i 1 dinar). Dakle, dokaz korektnosti mora da uključi analizu konkretnih apoena koji su u opticaju i male promene ovih apoena mogu da utiču na to da opisani pristup daje ili ne daje uvek optimalno rešenje.

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

- Optimalno rešenje ne može da sadrži više od jednog novčića od 1 dinar. Kada bi postojala makar dva novčića od 1 dinar, oni bi mogli biti zamenjeni jednim novčićem od 2 dinara, čime bi se broj upotrebljenih novčića smanjio, što je u kontradikciji sa pretpostavkom da je polazno rešenje optimalno. Potpuno analogno se dokazuje da optimalno rešenje ne može sadržati ni više od jednog novčića od 10 dinara.

- Dalje, optimalno rešenje ne može sadržati više od dva novčića od 2 dinara. Naime, ako bi sadržalo bar tri novčića od 2 dinara, oni bi mogli biti zamenjeni jednim novčićem od 1 i jednim novčićem od 5 dinara, čime bi se dobilo manje rešenje, što je u kontradikciji sa pretpostavkom da je polazno

rešenje optimalno. Potpuno analogno, optimalno rešenje ne može da sadrži ni više od dva novčića od 20 dinara.

- Na kraju, u optimalnom rešenju ne može biti više od jednog novčića od 5 dinara. Naime, ako bi postojala bar dva, ona bi mogla biti zamenjena jednim novčićem od 10 dinara čime bi se dobilo manje rešenje, što je kontradikcija.

- U rešenju nije moguće ni da istovremeno postoje dva novčića od 2 dinara i novčić od 1 dinara, jer bi se svi oni mogli zameniti sa jednim novčićem od 5 dinara, što je opet kontradikcija. Analogno važi i za novčiće od 10 i 20 dinara.

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

- Novčići od 1 dinar mogu da naprave najviše iznos od 1 dinara (jer se smeju pojaviti samo jednom).

- Novčići od 1 i 2 dinara mogu da naprave najviše iznos od 4 dinara (jer može biti najviše dva novčića od 2 dinara i u tom slučaju se ne sme koristiti i novčić od 1 dinar).

- Novčići od 1, 2 i 5 dinara mogu da naprave najviše iznos od 9 dinara (jer ne može biti više od jednog novčića od 5 dinara, dva novčića od 2 i jednog novčića od 1 dinara, a ako ima dva novčića od 2 dinara, ne sme se javiti i novčić od 1 dinara).

- Slično, novčići od 1, 2, 5 i 10 dinara mogu da naprave najviše iznos od 19 dinara (jer se 10 dinara može javiti samo jednom, a od 1, 2 i 5 se može napraviti najviše 9).

- Novčići od 1, 2, 5, 10 i 20 mogu da naprave najviše iznos od 49 dinara (jer 1, 2 i 5 mogu da naprave najviše 9, kako smo objasnili, a pošto se 10 dinara javlja najviše jednom, a 20 dinara najviše dva puta, ali ne sva tri takva zajedno, od 10 i 20 se može napraviti najviše 40).

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

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

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

```
int minBrojApoena(int iznos) {
  int brojApoena = 0;
  vector<int> apoeni
    {5000, 2000, 1000,
     500, 200, 100,
     50, 20, 10,
     5, 2, 1};
  while (iznos > 0) {
```

```
    for (int apoen : apoeni)
      if (iznos >= apoen) {
        iznos -= apoen;
        brojApoena++;
        break;
      }
  }
  return brojApoena;
}
```