GLAVA

1 Uvod

Realizacija velikih računarskih sistema je vrlo složen zadatak iz mnogih razloga. Jedan od njih je da veliki programski projekti zahtevaju koordinisani trud timova stručnjaka različitog profila. Za hvatanje u koštac sa ovim aspektom složenosti velikih računarskih sistema pomažu nam razni metodi softverskog inženjerstva. Međutim, to je samo jedan deo problema. Drugi deo je da veliki programi moraju rešavati algoritamske probleme za koje očigledna rešenja nisu uvek dovoljno efikasna. To je mesto gde metodi dizajna i analize algoritama dolaze do izražaja.

Pojednostavljeno govoreći, računari su nam stvarno nepohodni kada je potrebno da obradimo velike količine podataka. Zato kada izvršimo neki program koji obrađuje veliku količinu ulaznih podataka, moramo biti sigurni da će on koristiti prihvatljivu veličinu memorijskog prostora i da će se završiti u nekom prihvatljivom vremenskom intervalu. Ovo je skoro uvek nezavisno od primenjene softverske tehnologije (na primer, da li je ona proceduralna ili objektno-orijentisana), već uglavnom zavisi od redosleda programskih operacija koje računar treba da izvrši. Prema tome, druga strana problema rada velikih sistema je efikasan način na koji takvi sistemi treba da obavljaju svoj zadatak.

Mada algoritmi koje izučavamo u ovoj knjizi obično predstavljaju vrlo mali deo programskog koda velikih računarskih sistema, taj skoro zanemarljivi deo može biti od suštinske važnosti za uspeh celog projekta. Naivni pristup u rešavanju nekog algoritamskog problema pomoću pojednostavljenog algoritma često dovodi do loših performansi u slučaju velikih ulaznih podataka. Što je

još gore, nikakvo naknadno doterivanje loše dizajniranog algoritma neće rezultirati značajnim poboljšanjem. Prema tome, dizajn algoritama koji imaju dobre performanse kada na ulazu imaju velike količine podataka je jedan od najfundamentalnijih zadataka dobrog programiranja.

1.1 Algoritmi

Da bismo unekoliko bacili svetlo na to kako treba dizajnirati efikasne algoritme, ograničićemo se na pomenute male komponente velikih sistema i razmotriti (apstraktne) algoritamske probleme. ***Apstraktni algoritamski problem*** je problem opisan neformalno u opštim crtama kojima se obično nagoveštava algoritamski problem. ***Algoritamski problem*** je jasna specifikacija apstraktnog problema čije se rešenje traži na računaru. Pošto se u ovoj knjizi bavimo samo algoritamskim problemima, radi jednostavnosti ćemo često izostaviti pridev algoritamski i zvati ih prosto problemima.

Na primer, apstraktni *problem sortiranja* može se navesti na sledeći način: ako je dat niz brojeva u proizvoljnom redosledu, treba ih poređati u rastućem redosledu. Odgovarajući algoritamski problem sortiranja zahteva precizniju specifikaciju željenog ulazno-izlaznog odnosa: ulaz za problem sortiranja je niz brojeva u proizvoljnom redosledu, a izlaz je (moguće isti) niz istih brojeva koji su preuređeni tako da se pojavljuju od najmanjeg do najvećeg. Još jedan primer je takozvani *problem vraćanja kusura*: ako su dati apoeni novčića i neki novčani iznos, treba naći minimalni broj novčića kojima se dati iznos može potpuno usitniti. Ako je za problem sortiranja neformalan opis možda bio dovoljno jasan, za problem vraćanja kusura je očigledno potrebna preciznija specifikacija: ako je dat niz od $k \geq 1$ pozitivnih celih brojeva $c_1, \ldots, c_k$ („apoeni novčića") i pozitivan ceo broj a („novčani iznos"), treba naći niz od k nenegativnih celih brojeva $b_1, \ldots, b_k$ („pojedinačni brojevi novčića") tako da bude $a = \sum_{i=1}^{k} b_i \cdot c_i$ i zbir $\sum_{i=1}^{k} b_i$ minimalan.

Prema tome, opšti algoritamski problem se navodi skupom ***ulaznih vrednosti*** i skupom ***izlaznih vrednosti***, pri čemu oba skupa zadovoljavaju određene uslove kojima se definiše sâm problem. ***Instanca problema*** se sastoji od valjanog ulaza za problem, a ***rešenje instance problema*** se sastoji od odgovarajućeg izlaza za ulaznu instancu. Standardni format za specifikovanje problema se sastoji od dva dela koji definišu njegove ulazne i izlazne parametre. Oba dela se izražavaju pomoću pogodnih matematičkih objekata, kao što su brojevi, skupovi, funkcije i tako dalje, pri čemu ulazni deo predstavlja generičku instancu problema i izlazni deo predstavlja generičko rešenje za ovu generičku instancu.

Na primer, problem vraćanja kusura možemo formalno specifikovati u obliku u kojem se traži da bude zadovoljen ovaj ulazno-izlazni odnos:

Ulaz: Pozitivni ceo broj k, niz od k pozitivnih celih brojeva $[c_1, \ldots, c_k]$ i pozitivni ceo broj a.

Izlaz: Niz od k nenegativnih celih brojeva $[b_1, \ldots, b_k]$ tako da su zadovoljena dva uslova: $a = \sum_{i=1}^{k} b_i \cdot c_i$ i zbir $\sum_{i=1}^{k} b_i$ je minimalan.

Jedna instanca problema vraćanja kusura je specifikacija konkretnih vrednosti za ulazne parametre ovog problema, odnosno broj k, niz od k elemenata i broj a. Na primer, $k = 4$, niz $[1, 5, 10, 25]$ od četiri elementa i $a = 17$ predstavljaju jednu instancu problema vraćanja kusura. Rešenje ove instance problema je niz $[2, 1, 1, 0]$ od četiri elementa.

Algoritam je tačno definisana računarska procedura koja pretpostavlja neke vrednosti kao ***ulaz*** i proizvodi neke vrednosti kao ***izlaz***. Slično kuhinjskom receptu, algoritam je precizna i nedvosmislena specifikacija niza koraka koji se mogu mehanički izvršiti. Algoritam za neki problem obezbeđuje instrukcije kojima se korak-po-korak transformiše ulaz problema u njegov izlaz tako da bude zadovoljen željeni ulazno-izlazni odnos. ***Ispravan (korektan)*** algoritam završava rad sa ispravnim izlazom za svaku ulaznu instancu. Tada kažemo da algoritam ***rešava*** problem. Drugim rečima, algoritam za neki problem je jasno naveden niz instrukcija koje računar treba da izvrši za rešavanje problema.

Dizajn algoritama obuhvata postupak kojim se dolazi do tih nedvosmislenih instrukcija koje čine algoritam. Za ovaj proces konstruisanja algoritama ne postoji čarobna formula koja se može primeniti u opštem slučaju, već se od dizajnera zahteva i doza kreativnosti. Ipak, videćemo da postoje standardni metodi koje možemo koristiti da bismo algoritamski rešili mnoge tipove problema. Tu spadaju algoritamske paradigme kao što su to algoritmi tipa podeli-pa-reši, „pohlepni" algoritmi, algoritmi dinamičkog programiranja, randomizirani algoritmi i tako dalje.

Analiza algoritama pokušava da izmeri upotrebu računarskih resursa nekog algoritma. Najskuplji resursi koje obično želimo da minimizujemo su vreme izvršavanja i memorijska zauzetost nekog algoritma, i prema tome govorimo o njegovoj vremenskoj i memorijskoj složenosti. Ovde treba obratiti pažnju da mera količine resursa koju neki algoritam zahteva ipak zavisi od modela računara na kojem se algoritam izvršava. U ovoj knjizi radimo isključivo u granicama modela sekvencijalnih mašina — modeli paralelnih računara i paralelni algoritmi zaslužuju posebnu knjigu.

1.2 Računarski model

Uopšteno govoreći, računarski modeli su pojednostavljeni opisi stvarnih računara kojima se zanemaruje njihova puna složenost, osim nekoliko svojstava za koje se smatra da najbolje odslikavaju fenomen koji se izučava. Pošto je merenje računarskih resursa (vreme i prostor) koje algoritmi koriste jedan od predmeta našeg izučavanja, to znači da najpre treba precizno formulisati model računarske mašine na kojoj se algoritmi izvršavaju i definisati primitivne operacije koje se koriste prilikom izvršavanja. Međutim, nećemo ići u toliko detalja za opis formalnog računarskog modela, već ćemo u nastavku samo neformalno obrazložiti kako određujemo složenost algoritama. To je dovoljno za našu svrhu, jer će nas interesovati asimptotske performanse algoritama, odnosno njihova efikasnost za velike količine ulaznih podataka. Pored toga, tačni računarski modeli su zaista potrebni za dokazivanje donjih granica složenosti algoritama. Pošto ćemo o ovome nešto više reći samo u poslednjem poglavlju, nema potrebe da time komplikujemo izlaganje u ostatku knjige.

Ipak, radi veće jasnoće, sada ćemo kratko opisati glavne odlike računarskog modela sa kojim ćemo raditi. U većem delu knjige pretpostavljamo da naš glavni računarski model predstavlja idealizovan „realan" računar sa jednim procesorom. Ne postavljamo nikakvu gornju granicu na veličinu memorije takvog računara, ali pretpostavljamo da programi koji se na njemu izvršavaju ne mogu da modifikuju sami sebe. Tačan repertoar njegovih instrukcija nije posebno bitan, pod uslovom da je skup tih instrukcija sličan onom koji se može naći kod pravih računara. Pretpostavljamo da postoje aritmetičke operacije (za sabiranje, oduzimanje, množenje, deljenje, deljenje sa ostatkom, celobrojni deo broja), logičke operacije (i, ili, negacija), ulazno-izlazne operacije (čitanje, pisanje), operacije za premeštanje podataka (kopiranje iz memorije u procesor i obratno), operacije za pomeranje bitova podataka levo ili desno za mali broj pozicija, kao i upravljačke operacije (za uslovno i bezuslovno grananje, poziv i povratak iz potprocedure). Ove operacije rade nad svojim operandima koji su smešteni u memorijskim rečima čije se adrese određuju korišćenjem direktnih ili indirektnih šema adresiranja. U našem modelu ne postoji memorijska hijerarhija, na primer keš ili virtualna memorija, kao kod pravih računara. Naš model je dakle generički jednoprocesorski računar sa proizvoljnim pristupom memoriji (RAM) (engl. *random-access machine*) i trebao bi biti poznat svima onima koji su radili na mašinskom jeziku. To drugim rečima znači da kao postulat uzimamo da će naši algoritmi biti realizovani u obliku (mašinskih) programa koji se izvršavaju na RAM mašini.

Osnovni podaci kojima se manipuliše u našem RAM modelu su brojevi celobrojnog tipa i oni sa pokretnim zarezom. Dodatno pretpostavljamo da postoji granica veličine memorijske reči tako da brojevi koji se čuvaju u njima ne mogu rasti neograničeno. Na primer, ako je veličina memorijske reči jednaka w, tada u našem modelu možemo predstaviti cele brojeve koji su manji ili jednaki 2^{w-1} po apsolutnoj vrednosti. Brojevi sa pokretnim zarezom su predstavljeni u standradnom IEEE formatu.

U sekvencijalnom RAM modelu se instrukcije izvršavaju jedna za drugom bez paralelizma. Zato pre nego što možemo analizirati složenost nekog algoritma, moramo znati i vreme koje je potrebno za izvršavanje svake instrukcije, kao i to koliki memorijski prostor zauzima svaki operand. U tu svrhu ćemo koristiti ***jedinični model složenosti***, u kojem se svaka instrukcija izvršava za jednu jedinicu vremena i svaki operand zauzima jednu jedinicu memorijskog prostora.

Ovaj model je odgovarajući ako se može pretpostaviti da se svaki broj koji se pojavljuje u programu može smestiti u jednu memorijsku reč. Ako program izračunava vrlo velike brojeve, koji se ne mogu realistično uzeti kao operandi mašinskih instrukcija, a jedinični model složenosti se koristi sa nepažnjom ili zlom namerom, poznato je da možemo dobiti lažnu sliku o efikasnosti programa.[1] Ali ako se jedinični model složenosti primenjuje u svom duhu (na primer, rad sa velikim brojevima se deli u više jednostavnijih koraka i ne koriste se nerealistično moćne instrukcije), tada je analiza složenosti u ovom modelu umnogome jednostavnija i precizno odražava eksperimentalne rezultate.

Ukratko, u dizajnu i analizi algoritama vodimo se time kako rade pravi računari. To jest, našem idealizovanom računaru ne želimo da damo nerealistično moćne mogućnosti. Sa druge strane, voleli bismo da nam on omogućuje komforan rad bar sa nizovima podataka sa proizvoljnim pristupom, kao i sa jediničnim operacijama čiji su operandi srednje veliki brojevi.

1.3 Zapis algoritama

Sledeće pitanje koje se postavlja je notacija koju koristimo da bismo jasno naveli niz naredbi koje čine neki algoritam. Naš jedini uslov za to je da ta specifikacija mora da obezbedi precizan opis procedure koju računar treba da sledi. Da bismo precizno izrazili korake od kojih se sastoji neki algoritam,

[1] Zato neki autori koriste tzv. *logaritamski model složenosti*, u kojem se svaka instrukcija izvršava za vreme koje je proporcionalno logaritamskoj dužini operanada instrukcije, a svaki operand zauzima memorijski prostor koji je proporcionalan logaritmu njegove magnitude.

možemo koristiti, po redosledu rastuće preciznosti, prirodan jezik, pseudo jezik i pravi programski jezik. Nažalost, lakoća izražavanja ide u obrnutom redosledu.

Za opis i objašnjenje algoritama želeli bismo da koristimo zapis koji je prirodniji i lakši za razumevanje nego što je to odgovarajući program za RAM mašinu, ili čak i računarski program na programskom jeziku visokog nivoa. Ali u složenijim slučajevima, da bismo nedvosmisleno naveli algoritam, ipak je potrebno da koristimo programske konstrukcije.

Obratite pažnju na to da pojam algoritma nije isto što i konkretniji pojam računarskog programa. Računarski program se izvršava na pravom računaru, pa se zato on piše na nekom programskom jeziku poštujući njegova striktna pravila. Znamo da su tu važni svako slovo, svaka zapeta i svaki simbol, tako da nemamo pravo na grešku prilikom pisanja ako hoćemo da napisani program izvršimo na računaru. Za algoritme nam ne treba takav apsolutni formalizam — na kraju krajeva, algoritmi se ne izvršavaju na pravim i „glupim" računarima. Algoritmi se mogu zamisliti da se izvršavaju na jednoj vrsti idealizovanog računara sa neograničenom memorijom. Oni su u stvari matamatički objekti koji se mogu analizirati da bi se prepoznali suštinski elementi naizgled teških problema. Otuda, vrlo je bitno da možemo apstrahovati nevažne detalje programskog jezika. Još jedna razlika između algoritama i pravih programa je da nam kod algoritama nisu bitna pitanja softverskog inženjerstva. Na primer, da bi se suština algoritama prenela što jasnije i jezgrovitije, obrada grešaka kod algoritama se obično zanemaruje.

Algoritmi su obično mali po veličini ili se sastoje od kratkih delova naredbi prožetih tekstom koji objašnjava glavne ideje ili naglašava ključne tačke. Ovo je uglavnom prvi korak za rešavanje problema na računaru pošto sitni, nevažni detalji programskih jezika mogu samo da smetaju za dobijanje prave slike problema. Ovo ne znači da algoritam sme da sadrži bilo kakve dvosmislenosti u vezi sa vrstom naredbi koje treba izvršiti ili sa njihovim tačnim redosledom izvršavanja. To samo znači da možemo biti nemarni da li neki znak dolazi ovde ili onde, ako to neće zbuniti onoga koji pokušava da razume algoritam. Pored toga, neki elementi koji su formalno neophodni kod računarskih programa, na primer, tipovi podataka, obično se podrazumevaju iz konteksta bez navođenja.

Iz svih ovih razloga se algoritmi najčešće izražavaju neformalno u stilu pseudo jezika. U stvari, često ćemo ideje algoritma navoditi najpre na običnom jeziku, a zatim prelaziti na pseudo jezik da bismo pojasnili neke ključne detalje algoritma. Pseudo jezik koji koristimo u ovoj knjizi je sličan programskim jezicima C, Pascal i Java, tako da čitalac koji je programirao na ovim je-

zicima neće imati nikakvih problema da razume napisane algoritme. U tom pseudo jeziku se koriste uobičajeni koncepti programskih jezika kao što su promenljive, izrazi, uslovi, naredbe i procedure. Ovde nećemo pokušavati da damo precizne definicije sintakse i semantike našeg pseudo jezika, pošto to nije potrebno za našu svrhu i svakako prevazilazi namenu ove knjige. Umesto toga, u nastavku ćemo samo navesti kratak spisak konvencija koje se koriste u našem pseudo jeziku.

1. Osnovni tipovi podataka su celi brojevi, brojevi sa pokretnim zarezom, znakovi i pokazivači. Dodatne strukture podataka kao što su stringovi, liste, redovi za čekanje i grafovi se uvode po potrebi.

2. Promenljive i (složeni) tipovi podataka se ne deklarišu formalno, već se njihovo značenje uzima na osnovu njihovog imena ili iz konteksta.

3. Komentari se navode između simbola ⟦ i ⟧.

4. Naredbe petlje **for**, **while** i **until**, kao i uslovna naredba **if-then-else**, imaju uobičajenu interpretaciju. Promenljiva brojača u **for** petlji zadržava svoju vrednost nakon završetka petlje. Deo **else** u uslovnoj naredbi nije obavezan.

5. Svaka naredba se završava tačkom-zarez (;). Jedan red algoritma može sadržati više kratkih naredbi.

6. Operator dodeljivanja se označava simbolom za jednakost (=). Višestruko dodeljivanje u obliku $x = y = e$ dodeljuje obema promenljivim x i y vrednost izraza e. Ovo je ekvivalentno dvema naredbama dodeljivanja $x = e$; i $y = e$;, tim redom.

7. Logički operatori se označavaju matematičkim simbolima $\wedge$, $\vee$ i $\neg$. Logički izrazi u kojima učestvuju ovi operatori se uvek navode sa svim zagradama. Ti izrazi se skraćeno izračunavaju sleva nadesno. To jest, da bi se izračunala vrednost izraza $p \wedge q$, najpre se izračunava p. Ako je vrednost za p jednaka netačno (FALSE), vrednost za q se ne izračunava pošto vrednost izraza $p \wedge q$ mora biti netačno. Slično, za izraz $p \vee q$ se izračunava q samo ako je izračunata vrednost za p jednaka netačno, jer u svim ostalim slučajevima vrednost izraza $p \vee q$ mora biti tačno (TRUE). Logički izrazi koji se izračunavaju na ovaj način omogućavaju nam da ispravno koristimo uslove kao što je $(i \leq n) \wedge (A(i) = 0)$ čak i u slučaju kada indeks i premašuje veličinu n niza A.

8. Blokovska struktura složenih naredbi se naznačava jedino njihovim uvlačenjem. Upotreba uvlačenja u kratkim algoritmima umesto uobičajenih simbola za ograničavanje bloka naredbi kao što su { i } ili **begin** i **end** zapravo doprinosi boljoj čitljivosti.

9. Sve promenljive su lokalne za proceduru u kojoj se koriste. Isto ime za dve različite promenljive se nikad ne koristi. Globalne promenljive su uvek eksplicitno naglašene i koriste se samo kada su potrebne. Podrazumeva se da one postoje u odgovarajućem spoljašnjem okruženju.

10. Rad sa elementima niza se zapisuje navodeći ime niza i indeks željenog elementa u zagradama. Na primer, $A(i)$ označava i-ti element niza A. Tri tačke (...) se koriste da bi se naznačio opseg elemenata niza. Tako $A(i \ldots j)$ označava podniz niza A čiji su elementi $A(i), \ldots, A(j)$.

11. Procedure se koriste na dva načina. Jedan način je kao funkcija koja daje jedan rezultat. U ovom slučaju poslednja izvršena naredba u proceduri mora biti naredba **return** iza koje sledi neki izraz. Naredba **return** dovodi do izračunavanja tog izraza i do kraja izvršavanja procedure. Rezultat procedure je vrednost izračunatog izraza. Drugi način korišćenja procedure je kao potprogram. U ovom slučaju naredba **return** nije obavezna, a izvršavanje poslednje naredbe ujedno predstavlja kraj izvršavanja same procedure.

    Procedura može komunicirati sa drugim procedurama pomoću globalnih promenljivih i navođenjem parametara procedure. Ulazni i ulazno/izlazni parametri se pišu u zaglavlju procedure, a izlazni parametri se pišu iza naredbe **return**. Argumenti se proceduri prenose na transparentan način kojim se ispravno tretiraju formalni ulazni, ulazno/izlazni i izlazni parametri procedure. Argumenti složenih tipova podataka se prenose samo po referenci, odnosno pozvana procedura dobija pokazivač na stvarni argument.

12. Koriste se opisne naredbe običnog jezika svuda gde takve naredbe čine algoritam razumljivijim od ekvivalentnog niza programskih naredbi. Na primer, naredba

    for svaki čvor v grafa G **do**

    je mnogo izražajnija od njene detaljnije ali irelevantne programske realizacije.

1.4 Dizajn algoritama

Osnovni alati za konstruisanje dobrih algoritama su algoritamske paradigme. One predstavljaju opšte pristupe u konstruisanju *ispravnih* i *efikasnih* algoritamskih rešenja za probleme. Takvi pristupi su značajni zato što

- obezbeđuju šablone koji se mogu primeniti na široku klasu različitih problema;
- mogu se lako pretvoriti u standardne kontrolne strukture i strukture podataka koje postoje u većini programskih jezika visokog nivoa;
- vode do algoritama čija se vremenska i memorijska složenost može precizno analizirati.

U nastavku knjige ćemo se upoznati sa više algoritamskih paradigmi među kojima su podeli-pa-reši, „pohlepni" metod, dinamičko programiranje i randomizacija. Mada se na konkretan problem ponekad može primeniti više tehnika, često je algoritam konstruisan određenim pristupom bolji od rešenja dobijenih alternativnim pristupima. Zato izbor algoritamske paradigme predstavlja važan aspekt dizajna algoritma.

Kada se konstruišu dobri algoritmi mora se obratiti posebna pažnja na dva pitanja: ispravnost i efikasnost. Pošto su algoritmi za nas matematičke apstrakcije, ispravnost algoritama moramo pokazati pomoću matematičkih metoda i rasuđivanja. To je naročito važno za složene algoritme kod kojih njihova ispravnost nije očigledna, već zahteva pažljiv matematički dokaz. Za jednostavne algoritme je obično dovoljno kratko objašnjenje njihovih osnovnih osobina.

Efikasnost algoritma je mera količine računarskih resursa, obično su to vreme i memorija, koje algoritam koristi tokom izvršavanja. Intuitivno, ta količina računarskih resursa umnogome zavisi od veličine i strukture ulaznih podataka za algoritam. Na primer, ako primenimo analogiju algoritama i kuhinjskih recepata, jasno je da izvršavanje recepta za pravljenje pica zahteva više vremena i posuđa kada se to radi za 10 osoba nego za dve osobe.

Prema tome, da bismo odredili efikasnost algoritma, treba da izmerimo računarske resurse koje algoritam koristi tokom izvršavanja kao funkciju veličine ulaza. Sa druge strane, ova funkcija je obično vrlo komplikovana, pa da bismo pojednostavili njenu analizu možemo uzeti maksimalno moguće vreme izvršavanja (ili memorijsku zauzetost) za sve ulaze algoritma iste veličine. U knjizi se obično bavimo ovim procenama u najgorem slučaju, mada

ćemo na odgovarajućim mestima pominjati i druge vidove mere performansi algoritama.

Drugi aspekti dobrih algoritama su takođe važni. Tu naročito spadaju jednostavnost i jasnoća. Jednostavni algoritmi su poželjni zato što ih je lakše realizovati kao radne programe, jer dobijeni programi tada najverovatnije nemaju skrivenih grešaka koje se manifestuju tek u proizvodnom okruženju. Naravno, algoritam je bolji ukoliko je lako razumljiv onome ko čita njegov opis na pseudo jeziku. Nažalost, u praksi ovaj zahtev za efikasnim i jednostavnim algoritmima je često kontradiktoran, pa moramo pronaći balans između njih kada pišemo algoritme.

Ispravnost algoritama

Bez obzira na to kako dolazimo do nekog algoritma, prvi i najvažniji uslov koji dobijeni algoritam mora da zadovolji je da on ispravno rešava dati problem. Na putu dokazivanja ispravnosti algoritma postoje mnoge prepreke, kako praktične tako i teorijske. Što se tiče praktičnih teškoća, one su u vezi sa činjenicom da se većina programa piše tako da zadovoljavaju neke neformalne specifikacije, koje su i same nepotpune ili protivrečne. Na teorijskoj strani, uprkos brojnim pokušajima da se formalno definiše značenje programa, nije se iskristalisao opšteprihvaćeni formalizam kojim se pokazuje ispravnost netrivijalnih algoritama.

Ipak, korisno je navesti i dokazati odlike algoritama. Odlike o kojima je reč su najčešće kratka objašnjenja koje odslikavaju način na koji konkretan algoritam radi. Onaj ko piše algoritam bi bar trebalo da ima u vidu te odlike, iako je možda nepraktično da se one dokazuju u svim detaljima. Te odlike treba da služe i kao određen vodič u kreativnom procesu konstruisanja nekog algoritma.

Posebno koristan alat za dokazivanje odlika algoritama je matematička indukcija. Induktivni dokazi su od suštinskog značaja kada se pokazuje da neki algoritam radi ono što treba. Ovo se odnosi i na iterativne i na rekurzivne algoritme.

Dokazivanje ispravnosti iterativnih algoritama

Ključni korak u dokazivanju ispravnosti iterativnih algoritama je pokazati da petlje zadovoljavaju određene uslove. Neophodan uslov za svaku petlju je da njen izlazni uslov bude zadovoljen u jednom trenutku kako bi se izvršavanje petlje sigurno završilo.

Druga vrsta uslova je ona čija se tačnost ne menja izvršavanjem petlje. Takav uslov se naziva ***invarijanta petlje*** i predstavlja formalan iskaz koji je tačan pre svake iteracije petlje. To znači da će invarijanta petlje biti tačna nakon izvršavanja poslednje iteracije petlje (ili pre prve neizvršene iteracije), odnosno ona će biti tačna nakon završetka rada cele petlje. Prepoznati pravu invarijantu petlje je važna veština, jer znajući invarijantu petlje i izlazni uslov petlje obično možemo zaključiti nešto korisno o ispravnosti samog algoritma.

Da bismo pokazali da se invarijanta petlje ne menja izvršavanjem petlje, moramo pokazati dve činjenice o invarijanti petlje:

1. Ona je tačna pre prve iteracije petlje.
2. Ako je ona tačna pre neke iteracije petlje, ona ostaje tačna i nakon izvršavanja te iteracije petlje (to jest, pre izvršavanja sledeće iteracije).

Obratite ovde pažnju na sličnost sa matematičkom indukcijom, gde dokazujemo bazni slučaj i induktivni korak. Pokazivanje da je invarijanta tačna pre prve iteracije je slično dokazivanju da važi bazni slučaj indukcije, dok pokazivanje da svaka iteracija ne menja tačnost invarijante petlje je slično dokazivanju induktivnog koraka.

Kao primer, razmotrimo problem izračunavanja najvećeg zajedničkog delioca dva pozitivna cela broja x i y. Najveći zajednički delilac za x i y označavamo sa $\gcd(x, y)$. ***Najveći zajednički delilac*** (ili kraće ***nzd***) para pozitivnih celih brojeva (x, y) se označava sa $\gcd(x, y)$ i definiše kao najveći ceo broj koji deli bez ostatka oba broja x i y. To jest, $d = \gcd(x, y)$ deli i x i y bez ostatka, pa $x = md$ i $y = nd$ za neke $m, n \in \mathbb{N}$,[2] i svaki drugi takav zajednički delilac brojeva x i y je manji od d.

Da li je ovo dobra definicija? Drugim rečima, da li svaki par pozitivnih celih brojeva ima najveći zajednički delilac? Jasno je da svaki par pozitivnih celih brojeva ima broj 1 kao zajedničkog delioca, a najveći broj koji istovremeno može da deli x i y bez ostataka je onaj broj koji je minimum od x i y. Dakle, $\gcd(x, y)$ uvek postoji i leži negde u zatvorenom intervalu između 1 i $\min\{x, y\}$.

Iterativni algoritam traži $\gcd(x, y)$ tako što počinje sa proverom minimuma brojeva x i y i zatim redom sve manjih celih brojeve da li su zajednički delioci za x i y, sve dok se ne otkrije prvi zajednički delilac. Pošto se provera obavlja redom od najveće do najmanje mogućnosti, prvi nađeni zajednički delilac biće najveći.

[2] U ovoj knjizi sa $\mathbb{N}$ označavamo skup $\{0, 1, 2, \dots\}$ nenegativnih celih brojeva.

```
GCD1(x, y)
  d = min{x, y};
  while (x ≠ 0 (mod d)) ∨ (y ≠ 0 (mod d)) do
    d = d − 1;
  return d;
```

Radi ilustracije, pokažimo na formalniji način da algoritam GCD1(x, y) ispravno izračunava $\gcd(x, y)$. Na početku, pokažimo da se **while** petlja završava, odnosno da nije beskonačna. U tom cilju, uočite najpre da se u izlaznom uslovu proverava da li d deli x i y bez ostatka. Ako je to nije slučaj, telo petlje se ponavlja; u suprotnom slučaju, petlja se završava. Da bismo videli zašto će izlazni uslov naposletku postati netačan, primetimo da je početno $d = \min\{x, y\}$, a da se d u svakoj iteraciji smanjuje za 1. Ovim uzastopnim smanjivanjem će d u najgorem slučaju postati jednako 1, izlazni uslov će tada biti netačan pošto $x = 0 \pmod 1$ *i* $y = 0 \pmod 1$, pa će se petlja završiti.

Sada još moramo pokazati tačnost izračunavanja $\gcd(x, y)$ u algoritmu GCD1. Odgovarajuća invarijanta **while** petlje je uslov $d \geq \gcd(x, y)$. Jasno je da je ovaj uslov zadovoljen pre prve iteracije te petlje, jer d početno dobija vrednost $\min\{x, y\}$ i uvek je $\min\{x, y\} \geq \gcd(x, y)$.

Zatim treba pokazati da ako ovaj uslov važi pre neke iteracije, on će važiti i posle izvršavanja te iteracije. Zato pretpostavimo da je vrednost promenljive d veća ili jednaka vrednosti $\gcd(x, y)$ pre neke iteracije, i pretpostavimo da je ta iteracija **while** petlje izvršena. To znači da je izlazni uslov petlje bio tačan i da se vrednost promenljive d smanjila za 1. Pošto je izlazni uslov bio tačan, stara vrednost promenljive *d nije* bila delilac oba broja x i y, pa zato stara vrednost promenljive d nije jednaka $\gcd(x, y)$ nego je striktno veća od $\gcd(x, y)$. Pošto je nova vrednost promenljive d za jedan manja od njene stare vrednosti, možemo zaključiti da je nova vrednost promenljive d opet veća ili jednaka vrednosti $\gcd(x, y)$ pre sledeće iteracije. Ovim je dokazano da $d \geq \gcd(x, y)$ predstavlja invarijantni uslov **while** petlje u algoritmu GCD1.

Na kraju, ispitajmo koji uslov važi kada se **while** petlja završi. Iz izlaznog uslova vidimo da se ta petlja završava kada je vrednost promenljive d delilac oba broja x i y. Pošto je $\gcd(x, y)$ najveći delilac oba broja x i y, sledi da nakon završetka **while** petlje važi $d \leq \gcd(x, y)$. Ali pošto nakon završetka **while** petlje važi i invarijanta petlje $d \geq \gcd(x, y)$, sledi da mora biti $d = \gcd(x, y)$. Pošto se ova vrednost promeljive d vraća kao rezultat algoritma GCD1, to pokazuje da ovaj algoritam ispravno izračunava $\gcd(x, y)$.

Dokazivanje ispravnosti rekurzivnih algoritama

Ako je dat neki problem, za rekurzivno definisanje rešenja tog problema implicitno koristimo indukciju. Nakon toga je obično jednostavno pretočiti to rekurzivno rešenje u rekurzivni algoritam koji ga implementira.

Rekurzivnom definicijom se neki koncept definiše pomoću samog tog koncepta. Naravno, takva definicija ne sme biti besmislena ili paradoksalna, nego ona mora definisati klasu složenijih objekata pomoću sličnih prostijih objekata. Preciznije, u svakoj rekurzivnoj definiciji prepoznajemo bazni slučaj, kojim se definišu jedan ili više najprostijih objekata, kao i induktivni korak, kojim se složeniji objekti konstruišu pomoću sličnih prostijih objekata određene klase.

Kao primer, razmotrimo ponovo problem izračunavanja najvećeg zajedničkog delioca dva pozitivna cela broja x i y. Pretpostavimo da smo nekako došli do ekvivalentne rekurzivne definicije za $\gcd(x,y)$ koja kaže da ako je $x \geq y$ tada je

$$\gcd(x,y) = \begin{cases} y, & x = 0 \pmod{y} \\ \gcd\big(y, x \pmod{y}\big), & x \neq 0 \pmod{y}. \end{cases} \tag{1.1}$$

Odmah u nastavku ćemo dokazati da je ova rekurzivna definicija ekvivalentna nerekurzivnoj definiciji koju smo dali na početku prethodnog odeljka. Pre toga želimo da demonstriramo lakoću kojom se ova rekurzivna definicija može pretvoriti u rekurzivni algoritam čija je ispravnost očigledna. Naime, sledeći rekurzivni algoritam GCD2(x,y) predstavlja praktično samo drugačiji zapis rekurzivne definicije (1.1). U tom algoritmu pretpostavljamo da unapred važi $x \geq y$, što se lako može obezbediti razmenom vrednosti promenljivih x i y pre poziva procedure GCD2.

```
GCD2(x, y)
  ⟦ Pretpostavljamo x ≥ y ⟧
  z = x (mod y);
  if z = 0 then
    return y;
  else
    return GCD2(y, z);
```

Teži deo je pokazati ekvivalentnost nerekurzivne i rekurzivne definicije najvećeg zajedničkog delioca. To jest, za svaka dva pozitivna cela broja x i y tako da $x \geq y$, rekurzivno izračunavanje prema jednačini (1.1) daje $\gcd(x,y)$.

Da bismo ovo pokazali, koristimo indukciju po broju k primene rekurzivne jednačine (1.1) u takvom izračunavanju.

Ako je $k = 1$, to znači da $x = 0 \pmod{y}$, to jest, y deli x bez ostatka, pa zato $\gcd(x, y) = y$. Pošto je y i rezultat rekurzivnog izračunavanja nzd-a u jednom koraku, bazni slučaj indukcije je tačan. Kao indukcijsku pretpostavku uzmimo da je tvrđenje tačno za neko $k \geq 1$. To jest, za svaka dva pozitivna cela broja u i v tako da je $u \geq v$, ako rekurzivno izračunavanje nzd-a za par (u, v) ima k koraka, tada dobijamo $\gcd(u, v)$ kao rezultat. Dalje, uzmimo dva pozitivna cela broja x i y tako da je $x \geq y$, i pretpostavimo da rekurzivno izračunavanje nzd-a za par (x, y) ima $k+1$ korak. Sekvencu primene jednačine (1.1) tačno $k+1$ put možemo predstaviti kao sledeće rekurzivno izračunavanje od $k+1$ koraka:

$$\text{nzd za } (x, y) \to \text{nzd za } \big(y, x \pmod{y}\big) \to \cdots \to d_2\,,$$

gde strelica ($\to$) označava jednu primenu jednačine (1.1) na par pozitivnih celih brojeva. Neka je $d = \gcd(x, y)$ i $d_1 = \gcd\big(y, x \pmod{y}\big)$. Pošto je $0 < x \pmod{y} < y$ i rekurzivno izračunavanje nzd-a za par $\big(y, x \pmod{y}\big)$ ima k koraka, po indukcijskoj pretpostavci dobijamo da $d_2 = d_1$. Dakle, dovoljno je pokazati da $d_1 = d$, pošto rekurzivno izračunavanje nzd-a za par (x, y) od $k+1$ koraka daje d_2 a $d_2 = d_1$. Da bismo pokazali $d = d_1$, pokazaćemo da su skupovi zajedničkih delioca celobrojnih parova (x, y) i $\big(y, x \pmod{y}\big)$ jednaki, jer su tada i njihovi nzd-ovi jednaki. U tom cilju, ako a deli x i y bez ostatka, tada a deli bez ostatka i y i $x \pmod{y} = x - iy$, $(i \in \mathbb{N})$. Obrnuto, ako b deli bez ostatka i y i $x \pmod{y} = x - iy$, $(i \in \mathbb{N})$, tada b deli bez ostatka kako $x = x \pmod{y} + iy$, $(i \in \mathbb{N})$, tako i y. Ovim je završen indukcijski dokaz ekvivalentnosti nerekurzivne i rekurzivne definicije nzd-a.

1.5 Analiza algoritama

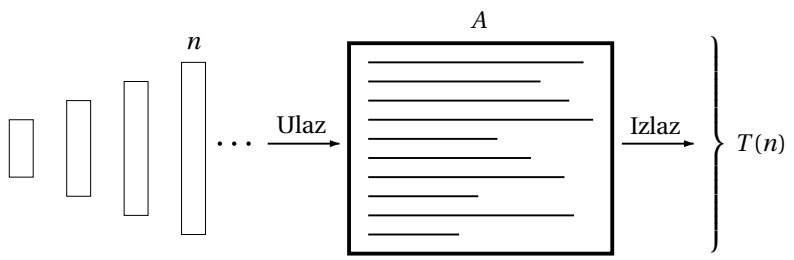
Pošto smo napisali algoritam za dati problem i pokazali da je algoritam ispravan, sledeći važan korak je određivanje količine računarskih resursa koje taj algoritam zahteva za izvršavanje. Ovaj zadatak procene efikasnosti nekog algoritma se naziva analiza algoritma. To ne treba shvatiti kao nezavisan i odvojen proces od dizajna algoritma, već oni međusobno utiču jedan na drugi i podjednako doprinose našem krajnjem cilju dobijanja ispravnih, efikasnih i elegantnih algoritama.

Dva najvrednija resursa koje svaki algoritam treba da štedi su vreme i memorija. Neki algoritam kome trebaju godine da bi završio svoj rad ili koji ko-

risti nekoliko gigabajta radne memorije nije baš mnogo koristan iako je možda potpuno ispravan. Drugi mogući kriterijumi za efikasnost algoritama su, na primer, količina mrežnog saobraćaja u toku njihovog rada ili količina podataka koji se dvosmerno prenose na diskove. Međutim, u ovoj knjizi skoro ekskluzivno ćemo proučavati vremensku složenost algoritama — memorijski zahtevi biće pomenuti samo ako nisu u „normalnim" granicama. Drugim rečima, identifikovaćemo efikasnost algoritma sa njegovim vremenom izvršavanja.

Vreme koje je svakom algoritmu potrebno za izvršavanje skoro uvek zavisi od količine ulaznih podataka koje on mora da obradi. Tako, prirodno je očekivati da sortiranje 10.000 brojeva zahteva više vremena nego sortiranje samo 10 brojeva. Vreme izvršavanja nekog algoritma je dakle funkcija veličine ulaznih podataka. Algoritmi za različite probleme mogu imati različite vidove veličine ulaza. Na primer, za problem sortiranja niza brojeva, veličina ulaza je broj članova niza koje treba sortirati; za množenje dva velika cela broja x i y, veličina ulaza je broj cifara broja x plus broj cifara broja y; za grafovske probleme, veličina ulaza je broj čvorova i/ili broj grana grafa. Srećom, veličina ulaza za konkretan problem je obično očigledna iz same prirode problema.

Prema tome, da bismo analizirali neki algoritam, najpre grupišemo ulazne podatke prema njihovoj veličini koju predstavljamo nenegativnim celim brojem n. Zatim želimo da vreme izvršavanja algoritma za ulaz veličine n predstavimo pomoću neke funkcije $T(n)$. Drugim rečima, za svaki nenegativan ceo broj n, vrednost $T(n)$ treba da daje broj vremenskih jedinica koliko traje izvršavanje algoritma za svaki ulaz veličine n. Na slici 1.1 je ilustrovan ovaj pristup.



SLIKA 1.1: Vreme izvršavanja $T(n)$ algoritma A kao funkcija veličine ulaznih podataka n.

Međutim, za dati ulaz veličine n, tačno vreme izvršavanja algoritma za taj konkretni ulaz zavisi od mnogo faktora kao što su brzina procesora raču-

nara ili struktura samih ulaznih podataka, a ne samo od veličine ulaza. Zato, da bismo pojednostavili analizu i da bi definicija funkcije $T(n)$ imala smisla, pretpostavljamo da se sve *osnovne operacije* izvršavaju za jednu vremensku jedinicu i posmatramo najgori slučaj ulaznih podataka. Tada da bismo dobili funkciju vremena izvršavanja $T(n)$ nekog algoritma treba da prosto prebrojimo osnovne operacije koje će se u algoritmu izvršiti *u najgorem slučaju*. Preciznije, funkciju $T(n)$ ***vremena izvršavanja u najgorem slučaju*** nekog algoritma definišemo da bude maksimalno vreme izvršavanja tog algoritma za sve ulaze veličine n. Ova funkcija se naziva i ***vremenska složenost u najgorem slučaju***. Poštujući tradiciju da se najgori slučaj podrazumeva, ako nije drugačije rečeno skoro uvek ćemo za termine vreme izvršavanja u najgorem slučaju i vremenska složenost u najgorem slučaju koristiti kraće nazive ***vreme izvršavanja*** i ***vremenska složenost***.

Osnovna operacija je računarska operacija čije se vreme izvršavanja može ograničiti nekom konstantom koja zavisi samo od konkretne realizacije (računara, programskog jezika, prevodioca itd.). Kasnije ćemo videti da nas u stvari interesuje samo asimptotsko ponašanje vremena izvršavanja, to jest, njegova brzina rasta u obliku neke konstante pomnožene nekom prostom funkcijom veličine ulaza.

Prema tome, za analizu algoritma nam je važan samo ukupan broj osnovnih operacija koje se izvršavaju, a ne njihovo tačno ukupno vreme izvršavanja. Ekvivalentno ovo možemo reći tako da se svaka osnovna operacija izvršava za jedinično vreme. Tipične osnovne operacije su:

- dodela vrednosti promenljivoj;
- poređenje dve promenljive;
- aritmetičke operacije;
- logičke operacije;
- ulazno/izlazne operacije.

Jedno upozorenje: neka matematička operacija, koja izgleda prosto kao što je to na primer sabiranje, ponekad se ne može smatrati osnovnom operacijom ako je veličina njenih operanada velika. U tom slučaju, vreme potrebno za njeno izvršavanje se povećava sa veličinom njenih operanada, pa se ne može ograničiti nekom konstantom. Imajući ovo na umu, ipak ćemo prethodne operacije smatrati osnovnim sem ako to nije realistično za dati problem.

Najgori slučaj izvršavanja algoritma je onaj u kojem se izvršava *najveći broj* osnovnih operacija. Kao jednostavan primer, razmotrimo ova dva algoritamska fragmenta A1 i A2:

A1

$n = 5$;
repeat
  read(m);
  $n = n - 1$;
until $(m = 0) \vee (n = 0)$;

A2

read(n);
repeat
  read(m);
  $n = n - 1$;
until $(m = 0) \vee (n = 0)$;

U fragmentu A1 imamo 5 iteracija petlje u najgorem slučaju, dok u fragmentu A2 imamo n iteracija iste petlje u najgorem slučaju.

Određivanje broja osnovnih operacija u najgorem slučaju kako bi se dobilo vreme izvršavanja nekog algoritma može se umnogome pojednostaviti ako znamo vremenske relacije tipičnih programskih konstrukcija. Naime, vreme izvršavanja složenijih konstrukcija možemo izvesti na osnovu vremena izvršavanja njihovih sastavnih delova. U tabeli 1.1 su sumirana vremena izvršavanja osnovnih algoritamskih konstrukcija, gde T_B označava vreme izvršavanja (u najgorem slučaju) bloka naredbi B. Tom tabelom nije obuhvaćena rekurzija, pošto će o njoj biti više reči u poglavlju 5.

Konstrukcija	**Vreme izvršavanja**
Sekvenca naredbi S: P; Q;	$T_S = T_P + T_Q$
Uslovna naredba S: **if** C **then** P; **else** Q;	$T_S = \max\{T_P, T_Q\}$
Petlja S: (1) **while** C **do** P; (2) **repeat** P; **until** C; (3) **for** $i = j$ **to** k **do** P;	$T_S = T_P \cdot n$, gde je n broj iteracija u najgorem slučaju

TABELA 1.1: Vremenska složenost osnovnih algoritamskih konstrukcija.

Najvažnije pravilo u ovoj tabeli je da je vreme izvršavanja petlje najviše jednako vremenu izvršavanja naredbi u telu petlje (računajući i izlazni uslov)

pomnoženo sa brojem iteracija petlje. (Uočite da radi jednostavnosti u tabeli 1.1 nismo uzeli u obzir vreme potrebno za proveru uslova u uslovnoj naredbi i izlaznog uslova u naredbama petlje.) Ovo se odnosi i na ugnježđene petlje: vreme izvršavanja naredbi unutar grupe prosto ugnježđenih petlji jednako je vremenu izvršavanja naredbi u telu najunutrašnjije petlje pomnoženo sa proizvodom broja iteracija svih petlji. Na primer, za ovaj algoritamski fragment:

```
for i = 1 to n do
  for j = 1 to n do
    if i < j then
      swap(A(i, j), A(j, i));   ⟦ osnovna operacija ⟧
```

vreme izvršavanja je $T(n) = n \cdot n \cdot 2 = 2n^2$.

Pored vremena izvršavanja u najgorem slučaju, mogući su i drugi pristupi za merenje efikasnosti (determinističkih) algoritama. Jedan od njih je probabilistička analiza srednjeg vremena izvršavanja nekog algoritma. ***Prosečno vreme izvršavanja*** je funkcija $T_a(n)$ koja daje srednju vrednost vremena izvršavanja algoritma za slučajne ulaze veličine n. Prosečno vreme izvršavanja je realističnija mera performansi algoritma u praksi, ali je često to mnogo teže odrediti nego vreme izvršavanja u najgorem slučaju. Razlog za to je da moramo znati raspodelu verovatnoće slučajnih ulaznih podataka, što je teško zamislivo u praksi. Ali da bismo ipak mogli lakše izvesti matematička izračunavanja, obično pretpostavljamo da su svi ulazi veličine n jednako verovatni, što u nekom konkretnom slučaju možda i nije pravo stanje stvari.

Sasvim drugačiji pristup je merenje efikasnosti algoritma u odnosu na efikasnost drugih algoritama koji rešavaju isti problem. Ovaj metod se sastoji u tome da se formira mala kolekcija tipičnih ulaznih podataka koji služe kao tzv. ***benčmark*** podaci. To znači da ove podatke smatramo reprezentativnim za sve ulazne podatke, odnosno podrazumevamo da će algoritam koji ima dobre performanse za benčmark podatke imati isto tako dobre performanse za *sve* ulazne podatke.

U ovoj knjizi ćemo se uglavnom baviti vremenom izvršavanja u najgorem slučaju, a benčmark analizu nećemo uopšte više razmatrati. Ovo stanovište ćemo drastično promeniti u poglavlju 11, u kojem izučavamo algoritme koji su po prirodi „slučajni”. Ti randomizirani algoritmi u toku izvršavanja donose slučajne odluke, pa njihovo vreme izvršavanja zavisi od slučajnih ishoda tih odluka. Za randomizirane algoritme ćemo određivati njihovo ***očekivano vreme izvršavanja***, pri čemu se matematičko očekivanje izračunava u odnosu na slučajne odluke tokom izvršavanja.

Na osnovu nađenog vremena izvršavanja nekog algoritma možemo pokušati da odgovorimo i na tri dodatna pitanja:

1. Kako možemo reći da li je neki algoritam dobar?
2. Kako znamo da li je neki algoritam optimalan?
3. Zašto nam je uopšte potreban efikasan algoritam — zar ne možemo prosto kupiti brži računar?

Da bismo odgovorili na prvo pitanje, lako možemo uporediti vreme izvršavanja posmatranog algoritma sa vremenima izvršavanja postojećih algoritama koji rešavaju isti problem. Ako ta usporedba ide u prilog posmatranom algoritmu, možemo kvantitativno zaključiti da je on dobar algoritam za neki problem.

Na drugo pitanje je teško odgovoriti u opštem slučaju. To je zato što iako za bilo koja dva algoritma možemo reći koji je bolji ako uporedimo njihova vremena izvršavanja, ipak ne možemo reći da li je onaj bolji algoritam zaista i najbolji mogući. Drugim rečima, moramo nekako znati donju granicu vremena izvršavanja svih algoritama za konkretan problem. To znači da za dati problem ne samo poznati algoritmi, nego i oni koji još nisu otkriveni, moraju se izvršavati za vreme koje je veće od neke granice. Određivanje ovakvih dobrih granica nije lak zadatak i time se nećemo baviti u ovoj knjizi.

Treće pitanje obično postavljaju neuki ljudi koji misle da se novcem može sve kupiti. Pored toga što su intelektualno izazovniji, efikasni algoritmi će se skoro uvek brže izvršavati na sporijem računaru nego loši algoritmi na super računaru za dovoljno velike ulazne podatke, u šta ćemo se i sami uveriti. U stvari, instance problema ne moraju da postanu toliko velike pre nego što brži algoritmi postanu superiorniji, čak i kada se izvršavaju na običnom personalnom računaru.

Zadaci

1. Razmotrimo *problem pretrage*:

Ulaz: Broj n, niz od n brojeva $A = [a_1, a_2, \ldots, a_n]$ i broj x.

Izlaz: Prvi indeks i takav da je $x = A(i)$ ili $i = 0$ ako se x ne nalazi u A.

a) Navedite nekoliko instanci problema pretrage i odgovarajuća rešenja.

b) Napišite iterativni algoritam koji traži broj x u nizu ispitivanjem niza od njegovog početka. Koristeći odgovarajuću invarijantu petlje, pokažite ispravnost vašeg algoritma.

2. Razmotrimo problem izračunavanja mesečnih rata prilikom otplate kredita. Neka su L iznos kredita, I godišnja kamatna stopa (pri čemu $I = 0.08$ označava kamatnu stopu od 8%), i P iznos mesečne rate za kredit. Koliki je iznos neotplaćenog kredita L_n posle n meseci?

Svakog meseca, iznos kredita se povećava usled mesečne kamate i smanjuje usled otplate mesečne rate. Prema tome, ako je $M = 1 + I/12$, tada je $L_0 = L$ početni iznos kredita, $L_1 = ML_0 - P$ je iznos kredita posle prvog meseca, $L_2 = ML_1 - P$ je iznos kredita posle drugog meseca, i tako dalje.

a) Napišite iterativni algoritam za izračunavanje L_n, ako su date vrednosti za L, I, P i n. Koristeći odgovarajuću invarijantu petlje, pokažite ispravnost vašeg algoritma. (*Savet:* Iznos kredita koji preostaje nakon i-te iteracija petlje jeste L_i.)

b) Izvedite rekurzivnu formulu za L_n i napišite rekurzivni algoritam za izračunavanje L_n.

c) Dokažite indukcijom da za svako $n \geq 0$,

$$L_n = L \cdot M^n - P \cdot \left(\frac{M^n - 1}{M - 1} \right).$$

d) Izvedite formulu za aktuelni iznos mesečne rate P ukoliko je iznos kredita L, godišnja kamata je I i broj rata je n. (*Savet:* Pretpostavite da se iznos kredita smanjio na 0 nakon n otplaćenih rata.)

3. Tačno odredite vreme izvršavanja (u najgorem slučaju) sledećih algoritamskih fragmenata:

a)
```
if x = 0 then
    for i = 1 to n do
        A(i) = i;
```

b)
```
i = 1;
repeat
    A(i) = B(i) + C(i);
    i = i + 1;
until i = n;
```

c)
```
if x = 0 then
  for i = 1 to n do
    for j = 1 to n do
      A(i, j) = 0;
else
  for i = 1 to n do
    A(i, i) = 1;
```

d)
```
for i = 1 to n do
  for j = 1 to n do
    for k = 1 to j do
      if i = j then A(k, k) = 0;
```