1. Ispravnost programa

1.1 Ispravnost programa

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

1.2 Specifikacija programa

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

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

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

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

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

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

Dva osnovna pristupa verifikaciji su:

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

O svakom od njih će biti više reči u nastavku.

1.2.1 Najčešći izvori grešaka

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

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

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

- **Greške prekoračenja.** Čest razlog grešaka u programima je korišćenje neodgovarajućih tipova podataka, tj. tipova koji uzrokuju da se usled prekoračenja ili potkoračenja dobiju netačni rezultati izračunavanja. Stoga je uvek poželjno imati u vidu moguće raspone vrednosti ulaznih podataka i na osnovu njih proveriti da li su rasponi svih međurezultata i završnih rezultata takvi da mogu da se predstave odabranim tipovima promenljivih. U sklopu provere graničnih vrednosti, poželjno je uvek uključiti i namjanje i najveće moguće vrednosti ulaznih veličina.
- **Provera graničnih vrednosti.** Veliki broj grešaka javlja se na granicama opsega petlje, graničnim indeksima niza, graničnim vrednostima argumenata aritmetičkih operacija i slično. Zbog toga je testiranje na graničnim vrednostima izuzetno važno i program koji prolazi takve ulazne veličine često je ispravan i za druge. Na primer, ako se testira program koji učitava jedan red teksta u neki niz, trebalo bi proveriti da li program ispravno radi kada je na ulazu prazan red (tj. red koji sadrži samo karakter `'\n'`). Drugom granicom mogu se smatrati ulazni redovi koji su veoma dugi (duži od broja elemenata niza u koji se učitavaju podaci) ili pre čijeg kraja se nalazi kraj toka podataka. Stoga bi trebalo proveriti i da li se program ispravno ponaša pri učitavanju redova koji su dugi koliko i niz u koji se učitavaju podaci ili kraći za jedan karakter, duži za jedan karakter i slično.

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

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

1.3 Dinamičko verifikovanje programa – testiranje i debagovanje

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

1.3.1 Testiranje

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

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

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

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

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

Postoje različite metode testiranja, a neke od njih su:

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

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

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

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

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

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

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

1.3.2 Automatsko testiranje

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

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

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

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

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

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

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

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

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

```
  0.5
[  FAILED  ] StepenTest.NegativanEksponent

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

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

1.3.3 Debagovanje

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

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

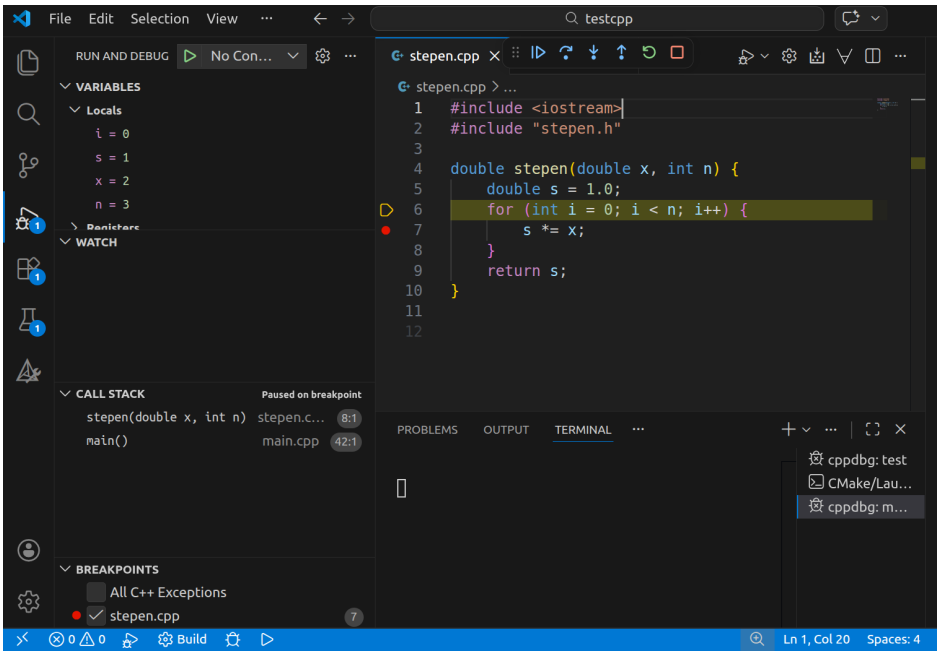
Osnovni pojmovi u radu sa debagerom su:

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

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

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

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



Slika 1.1: Ilustracija rada debagera u okruženju VS Code

1.4 Statičko ispitivanje ispravnosti programa

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

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

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

1.4.1 Neformalno ispitivanje ispravnosti algoritama

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

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

1.4.2 *Induktivno-rekurzivna konstrukcija*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1.4.3 *Dokazivanje ispravnosti rekurzivnih funkcija*

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

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

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

Rekurzivna implementacija, kao što smo videli je sledeća.

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

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

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

Dokaz. Teoremu možemo dokazati indukcijom.

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

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

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

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

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

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

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

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

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

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

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

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

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

1.4.4 *Dokazivanje ispravnosti iterativnih algoritama i invarijante petlji*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dokaz. Dokažimo da je uslov invarijanta.

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

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

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

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

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

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

□

Pokažimo da dokazana invarijanta obezbeđuje korektnost.

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

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

Iterativno brzo stepenovanje

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

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

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

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

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

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

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

Dokaz. Razmotrimo inicijalizaciju i održanje invarijante.

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

– Ako je n paran broj, tada je $n' = n/2$, $x' = x^2$ i $s' = s$. Tada je $s' \cdot {x'}^{n'} = s \cdot (x^2)^{n/2} = s \cdot x^n = x_0^{n_0}$. Pošto je $n > 0$, važi da je $n' \geq 0$. □

Iz ove invarijante sledi korektnost.

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

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

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

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

Još nekoliko primera ovako preciznih dokaza korektnosti dato je u dodatku 1.A.2. U nastavku ovog poglavlja videćemo još nekoliko primera primene tehnike invarijante petlje. Mora se priznati da kada se tehnika koristi potpuno formalno, da bi se dokazala korektnost već napisanog programskog koda, to ne deluje naročito inspirišuće (pogotovo, ako su programi jednostavni i ako je jednostavno intuitivno razumeti razloge njihove korektnosti). Retko kada se u praktičnom programiranju korektnost zaista dokazuje potpuno formalno (osim u slučaju softvera koji može da ugrozi veliki broj života, poput, na primer, softvera koji upravlja metro-sistemom u

Parizu, koji jeste u potpunosti formalno verifikovan). Međutim, argumente i invarijante na kojima korektnost počiva programer često "provrti po glavi". Videćemo i da se tehnika invarijanti može upotrebiti i pre nego što je program napisan u cilju izvođenja programskog koda iz specifikacije. Jasne invarijante često jednoznačno ukazuju na to kako programski kôd treba da izgleda i na taj način pomažu u procesu programiranja.

Zadatak: Trobojka

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

Opis ulaza

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

Opis izlaza

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

Primer

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

Rešenje

Dva prolaza kroz niz

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

Jedan prolaz kroz niz

Zadatak možemo rešiti pomoću samo jednog prolaza kroz niz i to "u mestu" tj. bez korišćenja pomoćnog niza. Algoritam u nastavku poznat je pod nazivom "Holandska zastava trobojka" (engl. Dutch national flag) 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.*

Nakon inicijalizacije promenljivih stanje je sledeće.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
}
```

Zadatak: Najmanji broj koji nije zbir elemenata skupa

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

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

Opis ulaza

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

Opis izlaza

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

Primer

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

Rešenje

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

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

Pretpostavimo da se tegovima $m_0, \ldots, m_i$ mogla izmeriti svaka masa iz intervala $[0, Z_i]$. Tada se dodavanjem tega m_{i+1} može izmeriti svaka masa iz intervala $[0 + m_{i+1}, Z_i + m_{i+1}] = [m_{i+1}, Z_{i+1}]$ (dodavanjem tega m_{i+1} na prethodno odabrane tegove). Ako je novi učitani element $m_{i+1} > Z_i + 1$, onda se $Z_i + 1$ ne može

izmeriti (jer je niz sortiran, pa su mase svih preostalih tegova veće ili jednake m_{i+1}), i to je tražena vrednost. U suprotnom, unija intervala $[0, Z_i] \cup [m_{i+1}, Z_{i+1}]$ pokriva interval $[0, Z_{i+1}]$, pa se sve mase iz tog intervala mogu izmeriti i invarijanta ostaje na snazi.

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

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

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

Dokažimo korektnost ovog algoritma.

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

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

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

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

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

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

Dokaz. Slučaj kada se petlja završi prekidom, jer je $m_i > Z + 1$ je već razmotren. Kada se petlja završi, važi da je $i = n$. Na osnovu invarijante Z je zbir svih elemenata niza, i svaki broj iz $[0, Z]$ jeste zbir nekog podskupa prvih $i = n$ elemenata

niza, tj. celog niza. Zato je $Z + 1$ najmanji element koji nije moguće dobiti (jer se uključivanjem svih elemenata dobija najviše Z) i prikazano rešenje je ispravno. □

1.4.5 *Ispitivanje zaustavljanja programa*

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

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

Razmotrimo nekoliko primera.

Stepenovanje

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

Brzo stepenovanje

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

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

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

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

Kolacova hipoteza

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

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

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

1.5 Formalno ispitivanje korektnosti programa

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

Semantika određenog programskog koda može se zapisati trojkom oblika

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

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

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

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

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

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

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

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

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

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

2. Efikasnost programa i složenost algoritama

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

- vremenska složenost;
- prostorna (memorijska) složenost.

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

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

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

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

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

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

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

Ocena potrebnih resursa se obično vrši:

- u terminima konkretnog vremena/prostora utrošenog za neke konkretne ulazne podatke;

- u terminima asimptotskog ponašanja vremena/prostora kada veličina ulaza raste.

2.1 Efikasnost programa za konkretne ulazne podatke

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

2.1.1 Merenje utrošenog vremena

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

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

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

void f(void) {  ... }

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

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

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

2.1.2 *Profajliranje*

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

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

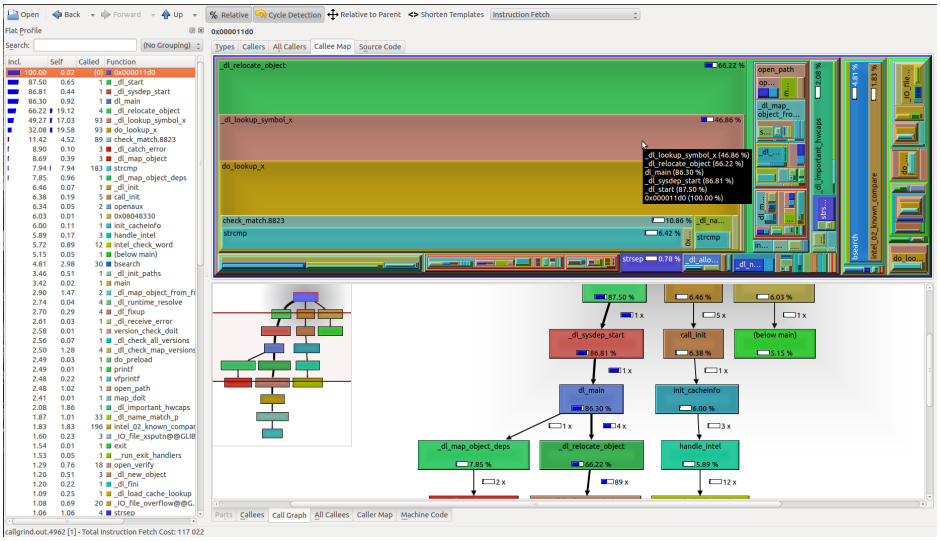
```
valgring --tool=callgrind mojprogram argumenti
```

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

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

Slika @fig:profiler 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.



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

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

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

2.1.3 *Procenjivanje potrebnog vremena*

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

koje su primenjene u kreiranju izvršivog programa. Ipak, procene su jako važne, jer programer i pre pisanja koda treba da ima neki grubi osećaj koliko resursa će program trošiti i da li će odabrani algoritam biti dovoljno efikasan.

2.2 Asimptotsko ponašanje i red složenosti algoritma

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

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

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

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

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

- U nekim situacijama moguće je izvršiti **analizu prosečnog slučaja** i izračunati prosečno vreme izvršavanja algoritma, ali da bi se to uradilo, potrebno je poznavati prostor dopuštenih ulaznih vrednosti i verovatnoću da se svaka dopuštena ulazna vrednost pojavi na ulazu programa. U slučajevima kada je bitna garancija efikasnosti svakog pojedinačnog izvršavanja programa, procena prosečnog slučaja može biti varljiva, previše optimistična, i može da se desi da u nekim situacijama program ne može da se izvrši sa raspoloživim resursima. Na primer, analiza prosečnog slučaja bi za pomenuti program prijavila da se u proseku izvršava ispod jedne sekunde, međutim, za neke ulaze on se može izvršavati i preko deset sekundi.
- Analiza najboljeg slučaja je previše optimistična i najčešće nema smisla. Njeno poznavanje može biti korisno kao poznavanje donje granice izvršavanja.

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

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

2.2.1 *Zavisnost između vremena izvršavanja i veličine ulaza*

Neka je funkcija $T(n)$ predstavlja meru vremena potrebnog za izvršavanje algoritma za ulaz veličine n (u najgorem slučaju, ako se ne naglasi drugačije). Pod pretpostavkom da se svaka instrukcija izvršava približno isto vreme, ova funkcija može

se dobiti i na osnovu funkcije kojom se izražava broj instrukcija koje algoritam izvršava za ulaz veličine n.

2.2.1.1 *Elementarne funkcije*

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

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

Tabela 2.1: Vreme izračunavanja. {#tbl:slozenost}

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

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

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

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

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

$n/T(n)$	2^n	$n!$
10	1 μs	3,63 ms
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.

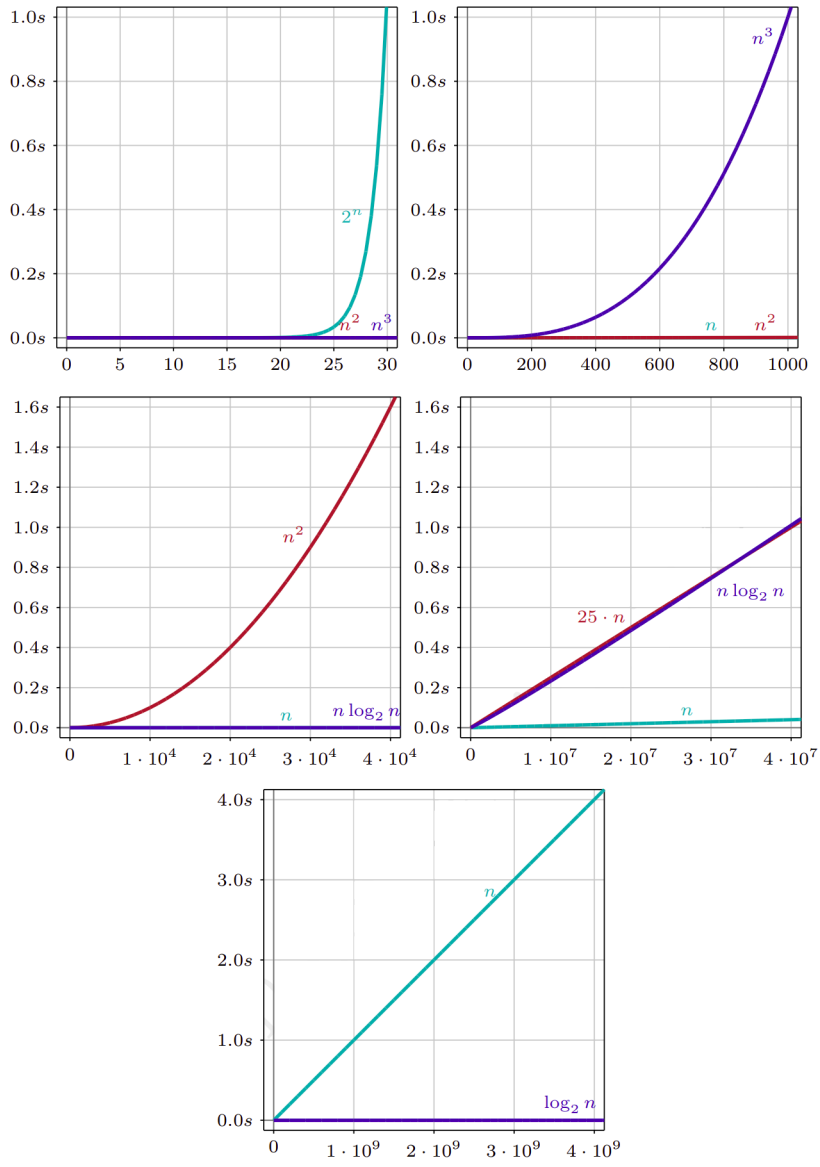
t	n	$n \log n$	n^2	n^3	2^n	$n!$
$1ms$	10^6	$63,000$	$1,000$	100	20	9
$10ms$	$10 \cdot 10^6$	$530,000$	$3,200$	215	23	10
$100ms$	$100 \cdot 10^6$	$4,5 \cdot 10^6$	$10,000$	465	27	11
$1s$	10^9	$40 \cdot 10^6$	$32,000$	$1,000$	30	12
$1min$	$60 \cdot 10^9$	$1,9 \cdot 10^9$	$245,000$	$3,900$	36	14

2.2.1.2 *Grafičko predstavljanje elementarnih funkcija*

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

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

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



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

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

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

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

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

2.2.1.3 *Kombinacija elementarnih funkcija*

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

Primer 2.2.2. *Na primer, neka je* $T(n) = 2n^2 + 10n + 8$*? Koliko je vreme potrebno za* $n = 10^5$*? Broj potrebnih koraka je* $2 \cdot (10^5)^2 + 10 \cdot 10^5 + 8$*, a potrebno vreme je* $2 \cdot 10^{10} \cdot 10^{-9}\text{s} + 10 \cdot 10^5 \cdot 10^{-9}\text{s} + 8 \cdot 10^{-9}\text{s} = 20\text{s} + 1\text{ms} + 8\text{ns} \approx 20\text{s}$*. Dakle, faktor* $2n^2$ *daje vreme* 20s*, faktor* $10n$ *daje vreme* 1ms*, a faktor* 8 *daje vreme* 8ns*. Očigledno je da je udeo vremena koje dolazi od članova* $10n$ *i* 8 *zanemariv u ukupnom*

zbiru. Nema, dakle, za velike vrednosti n nikakve značajne razlike između funkcija $T(n) = 2n^2$, $T(n) = 2n^2 + 10n + 8$, pa čak ni $T(n) = 2n^2 + 1000n + 5000$.

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

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

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

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

2.2.2 *Asimptotske oznake O, Ω i Θ*

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

- Oznaka $f(n) = O(g(n))$ se koristi da bi se iskazalo da funkcija $f(n)$ ne raste brže od funkcije $g(n)$, tj. da je $g(n)$ asimptotski *gornje ograničenje* brzine rasta funkcije $f(n)$.

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

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

Razmotrimo kako možemo definisati pojam $f(n) = O(g(n))$. Želimo da iskažemo da će počevši od neke veličine ulaza vrednosti funkcije f biti manje od vrednosti funkcije g. Pošto želimo da zanemarimo vrednost vodećeg koeficijenta ispred dominantnog člana, dopustićemo da se funkcija g skalira proizvoljnom konstantnom c tj. da se dopusti da vrednosti funkcije f budu manje od vrednosti funkcije $c \cdot g$. Takođe, pošto nas odnos funkcija f i $c \cdot g$ zanima samo za velike vrednosti n, zahtevaćemo da važi $f(n) \leq c \cdot g(n)$ za dovoljno velike vrednosti n, tj. za svako n veće od neke proizvoljne vrednosti n_0. Dakle, pojam $f(n) = O(g(n))$ se formalno može definisati na sledeći način.

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

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

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

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

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

Primer 2.2.4. *Može se dokazati da važi:*

- $7n^2 = O(n^2)$
  Tvrđenje važi jer za $c = 7$ (ali i za veće vrednosti c), važi $7n^2 \leq c \cdot n^2$ za sve vrednosti n.

- $7n^2 + 8n + 9 = O(n^2)$
  Za $c = 16$ *važi* $7n^2 + 8n + 9 \leq c \cdot n^2$ *za sve vrednosti* n *veće od* 2 *(jer za takve vrednosti* n *važi* $7n^2 \leq 7n^2$ *i* $8n \leq 8n^2$ *i* $9 \leq n^2$*).*
- $7n^2 + 8n + 9 = O(n^3)$
  Za $c = 1$*, važi* $7n^2 + 8n + 9 \leq c \cdot n^3$ *za sve vrednosti* n *veće od* 8*.*
- $2^n + n^2 = O(2^n)$
  Za $c = 2$ *važi* $2^n + n^2 \leq c \cdot 2^n$ *za sve vrednosti* n *veće od* 3 *(jer za takve vrednosti* n *važi* $2^n \leq 2^n$ *i* $n^2 \leq 2^n$*; da važi* $n^2 \leq 2^n$ *za* $n > 3$ *može se dokazati, na primer, matematičkom indukcijom).*
- $5 \cdot 3^n + 7 \cdot 2^n = O(3^n)$
  Za $c = 12$ *važi* $5 \cdot 3^n + 7 \cdot 2^n \leq c \cdot 3^n$ *za sve vrednosti* n *veće od* 0 *(jer za takve vrednosti* n *važi* $5 \cdot 3^n \leq 5 \cdot 3^n$ *i* $7 \cdot 2^n \leq 7 \cdot 3^n$*; da važi* $2^n \leq 3^n$ *za* $n > 0$ *može se dokazati, na primer, matematičkom indukcijom).*

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

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

Relacije Ω i Θ se definišu slično relaciji O.

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

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

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

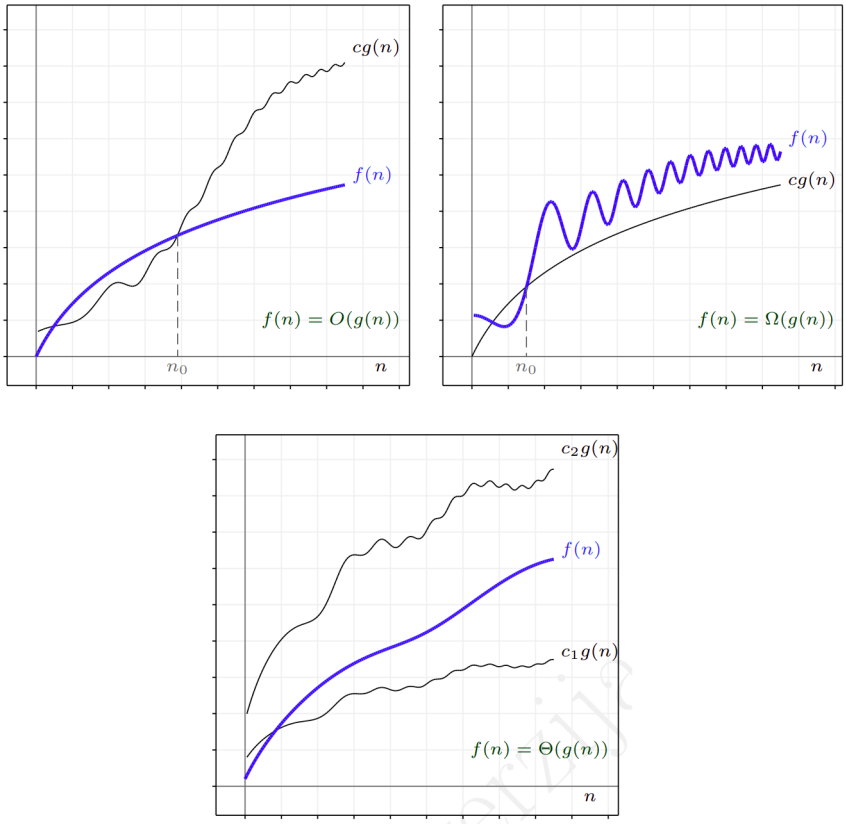
Definicija 2.2.4. *Ako postoje pozitivne realne konstante* c_1 *i* c_2 *i prirodan broj* n_0 *takvi da za funkcije* f *i* g *nad prirodnim brojevima važi*

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

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

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

Pojmovi *veliko o* (O), *veliko omega* (Ω) i *veliko teta* (Θ) ilustrovani su na slici @fig:asimptotska.



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

Primer 2.2.5. *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. *Relacije* O*,* Ω *i* Θ *imaju sledeća svojstva.*

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

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

Složenost algoritama u terminima Ω razmatra se ređe nego složenost u terminima Θ i O, ali ponekad takođe može da bude veoma važna. Složenost u terminima O daje gornju granicu za neku funkciju, a složenost u terminima Ω daje donju granicu. To se može razumeti i ovako: složenost u terminima O govori nam koliko je neki algoritam dobar ("asimptotski nije lošiji nego…"), a složenost u terminima Ω govori nam koliko je neki algoritam loš ("asimptotski nije bolji nego…"). Na primer, može se pokazati da je prosečno vreme izvršavanja nekog algoritma za sortiranje reda $\Omega(n^2)$ i to govori da je taj algoritam loš (jer postoje algoritmi čije vreme izvršavanje pripada klasi $O(n \log n)$.

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

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

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

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

2.2.3 *Određivanje složenosti*

Određivanje (vremenske i prostorne) složenosti algoritama zasniva se na određivanju funkcije $T(n)$ tj. određivanje tačnog ili približnog broja instrukcija koje se izvršavaju i memorijskih jedinica koje se koriste. Tačno određivanje tih vrednosti najčešće je veoma teško ili nemoguće, te se obično koriste razna pojednostavljivanja. Na primer, često se smatra da sve pojedinačne instrukcije (osim poziva funkcija)

troše jednako vremena. Ono što je važno je da takva pojednostavljivanja ne utiču na klasu složenosti kojoj algoritam pripada (jer, kao što je rečeno, konstantni aditivni i multiplikativni faktori ne utiču na red algoritma).

- Ukoliko se deo programa sastoji od nekoliko instrukcija bez grananja i petlji, onda se procenjuje da je njegovo vreme izvršavanja uvek isto, konstantno, te da pripada klasi $O(1)$.

- Ukoliko program ima dva dela, vremenske složenosti[3] $O(f)$ i $O(g)$, koji se izvršavaju jedan za drugim, ukupna složenost je[4] $O(f+g)$.

  To važi i u slučaju kada se u tim delovima javljaju rekurzivni pozivi (tada, na primer, vremenska složenost izražena u terminima vrednosti $f(n)$ može da zavisi od vremenske složenosti izražene u terminima vrednosti $f(n-1)$), ali tada efektivno izračunavanje složenosti zahteva korišćenje dodatnih tehnika (videti poglavlje 2.2.3.3).

- Ukoliko deo programa sadrži jedno grananje i ukoliko vreme izvršavanja jedne grane pripada klasi $O(f)$ a druge grane pripada klasi $O(g)$, onda je ukupno vreme izvršavanja tog dela programa ograničeno vremenom koje zahteva složenija grana, pa pripada klasi $O(\max(f,g))$, a ona je jednaka klasi $O(f+g)$.

- Ukoliko deo programa sadrži petlju koja se izvršava n puta, a vreme izvršavanja tela petlje je konstantno, onda ukupno vreme izvršavanja tog dela programa pripada klasi $O(n)$.

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

  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.

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

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

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

Analiza prostorne složenosti se vrši slično.

- Ukoliko deo programa ne sadrži pozive funkcija, dinamičku alokaciju, niti deklaracije objekata čija veličina zavisi od nekih parametera, onda je prostorna složenost tog dela programa konstanta tj. pripada klasi $O(1)$.

- Ukoliko neki deo programa vrši dinamičku alokaciju nekih n objekata veličine $O(1)$ – onda to doprinosi njegovoj prostornoj složenosti $O(n)$.

- Ukoliko program ima dva dela, prostorne složenosti $O(f)$ i $O(g)$, koji se izvršavaju jedan za drugim, ukupna složenost je $O(f + g)$.[5]

- Ukoliko se u programu javljaju pozivi funkcija, svaki poziv zauzima neki fiksni prostor na programskom steku (veličina tog prostora može biti ograničena jednom konstantom zajedničkom za sve funkcije) kao i dodatni prostor koji zauzimaju lokalne promenljive te funkcije (koje zauzimaju prostor na pozivnom steku ili su delom, kao na primer, elementi vektora, smešteni u hipu).

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

  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.

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

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

2.2.3.1 *Iterativni algoritmi*

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

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

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

Za n jednako 5*, funkcija daje izlaz:*

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

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

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

Grana `if` *izvršava se konstantno vreme* c'*, pa je ukupna složenost funkcije* `mnozenje` $O\left(c' + \frac{n(n+1)c}{2}\right) = O(n(n+1)) = O(n^2)$.
Funkcija `mnozenje` *nema poziva drugih funkcija i ne koristi dinamičku alokaciju, niti objekte čija veličina zavisi od ulaznih parametara, te je njena prostorna složenost konstantna, tj. pripada redu* $O(1)$.

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

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

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

Složenost prethodne petlje je $O(n)$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
for (int i = 0; i < n; i++)
   for (int j = 0; j < n; j++)
      for (int k = 0; k < n; 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*,* $\ldots$*,* $n-1$*. Pošto tročlanih kombinacija ima* $\binom{n}{3} = \frac{n(n-1)(n-2)}{3 \cdot 2 \cdot 1}$*, složenost je kubna, a konstantni faktor je* $\frac{1}{6}$*.*

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

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

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

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

Složenost prethodnog koda je $O(n \log n)$*. Složenost unutrašnje petlje, za svako konkretno* `i` *je* $O(\log i)$*, pa je ukupna složenost otprilike jednaka* $\log 1 + \log 2 + \ldots + \log n$*, a za ovo se može pokazati da je* $O(n \log n)$ *(jasno je da je izraz manji ili jednak* $n \log n$ *jer je svaki sabirak manji ili jednak* $\log n$*, međutim, može se pokazati i da je zbir veći ili jednak od* $\frac{n}{2} \log \frac{n}{2}$ *(što je takođe* $\Theta(n \log n)$*), zanemarivanjem prvih* $\frac{n}{2}$ *sabiraka, nakon čega ostaju samo sabirci koji su veći ili jednaki* $\log \frac{n}{2}$*).*

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

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

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

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

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

Sličan kôd se, na primer, može sresti u algoritmu particionisanja niza tako da se elementi razdvoje na one koji ne zadovljavaju svojstvo `P` i na one koji zadovoljavaju svojstvo `P`. I prethodni algoritam je složenosti $O(n)$ iako i on sadrži ugnežđene petlje. To ponovo možemo utvrditi amortizovanom analizom (jer ne znamo pojedinačni broj izvršavanja unutrašnjih petlji, ali možemo lako proceniti ukupan broj koraka koji se u njima napravi). Naime, promenljiva `l` se samo uvećava krenuvši od početka, a `d` se

samo smanjuje krenuvši od kraja niza, dok se ne susretnu, što će se desiti u najviše n koraka.

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

2.2.3.2 *Skrivena složenost*

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

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

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

2.2.3.3 *Rekurzivne funkcije*

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

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

```
unsigned faktorijel(unsigned n)
{
  if (n == 0)
    return 1;
  else
```

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

Neka $T(n)$ označava broj instrukcija koje zahteva poziv funkcije `faktorijel` *za ulaznu vrednost n. Za $n = 0$, važi $T(n) = a$, gde je a neka konstanta. Za $n > 0$, važi $T(n) = b + T(n-1)$, gde je b neka konstanta (u tom slučaju izvršava se jedno poređenje, jedno oduzimanje, broj instrukcija koje zahteva funkcija* `faktorijel` *za argument $n-1$, jedno množenje i jedna naredba* `return`*). Dakle, $T(n) = b + T(n-1) = b + b + T(n-2) = ... = \underbrace{b + b + ... + b}_{n} + T(0) = bn + a$*

Dakle, navedena funkcija ima linearnu vremensku složenost. Skrenimo pažnju i na to da vrednost faktorijela jako brzo raste i da zato veoma brzo dolazi do prekoračenja.

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

$$S(0) = c$$

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

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

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

2.3 Popravljanje efikasnosti programa

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

- Ne treba terati računar da vrši dugotrajna izračunavanja koja se jednostavno mogu izvršiti i “peške”, primenom matematičih uvida.
- Ne treba terati računar da više puta izračunava jedno te isto – rezultate izračunavanja moguće je upamtiti u memoriji, da se ne bi računali više puta.

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

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

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

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

2.3.1 Kompilatorske optimizacije

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

- **Otežana analiza:** Transformacija izvornog koda u asembler onemogućava efikasno korišćenje debagera i profajlera.
- **Varijacije u brzini:** Iako su ubrzanja često značajna, postoji rizik da neke tehnike u specifičnim slučajevima zapravo uspore program.
- **Vreme kompilacije:** Proces je znatno sporiji, pa se preporučuje tek u završnim fazama razvoja uz obavezno ponovno testiranje.

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

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

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

2.3.2 *Optimizacija samo bitnih delova programa*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Iz ovih primera možemo izvući neke ključne zaključke.

1. **Prioriteti:** Optimizacija delova programa koji troše mali procenat vremena je praktično zanemariva.
2. **Fokus na "uska grla":** Optimizacija je najefikasnija kada se primeni na delove koda koji zauzimaju najveći deo vremena izvršavanja.

3. **Teorijski limit:** Čak i ako deo p ubrzate do beskonačnosti ($s \to \infty$), ukupno ubrzanje nikada ne može preći vrednost $1/(1-p)$. To znači da deo koji se ne može optimizovati predstavlja fiksnu barijeru za performanse.

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

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

2.3.3 *Izbegavanje ponovljenih izračunavanja*

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

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

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

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

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

[7] Samuel Loyd, 1841-1911

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

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

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

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

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

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

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

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

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

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

2.3.4 *Slabljenje operacija*

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

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

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

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

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

2.3.5 *Pretprocesiranje i upotreba unapred izračunatih vrednosti*

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

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

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

2.3.6 *Odabir programskog jezika*

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

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

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

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

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

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

2.3.7 *Paralelizacija*

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

2.3.8 *Popravljanje prostorne složenosti*

Za razliku od nekadašnjih računara, na savremenim računarima memorija obično nije kritični resurs. Optimizacije se obično usredsređuju na štednju vremena ili energije, a ne prostora. Ipak, postoje situacije u kojima je potrebno štedeti memoriju, na primer, onda kada program barata ogromnim količinama podataka, kada je s^am 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.A Dodatak: matematičke osnove izračunavanja složenosti

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

2.A.1 Asimptotsko ponašanje elementarnih funkcija

Podsetimo se matematičkih svojstava elementarnih funkcija.

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