2

Analiza vremenske složenosti algoritama

Pri rješavanju nekog problema često može postojati više algoritama koji predstavljaju rješenje za taj problem. Analizom vremenske složenosti nekog algoritma određujemo trajanje algoritamskog procesa, pri čemu nas ne zanima stvarno vrijeme izvođenja (milisekunde, sekunde, minute, itd.) algoritma na nekom konkretnom računaru, nego trajanje algoritamskog procesa izražavamo aproksimacijom broja elementarnih operacija koje će taj algoritam izvesti. Štoviše, kada analiziramo efikasnost nekog algoritma, najčešće nas zanima u koju klasu stupnja rasta vremenske složenosti taj algoritam spada, a ne tačan broj operacija koje će taj algoritam izvršiti.

2.1 Uvod

Ako pretpostavimo da su za neke algoritme dozvoljene iste klase ulaznih objekata, te ako su rezultati takvih algoritama jednaki za jednake ulazne objekte, onda takve algoritme nazivamo *ekvivalentnim algoritmima*. Pojam ekvivalentnih algoritama ćemo ilustrirati na jednom vrlo jednostavnom problemu. Recimo da je potrebno dizajnirati algoritam za zbrajanje prvih n brojeva. Prema tome, ulaz u algoritam je cijeli broj n, a izlaz iz algoritma je suma s:

$$s = 1 + 2 + \ldots + n, \; n \geq 1 \qquad (2.1)$$

Algoritam prikazan u tabeli 2.1, u obliku funkcije SUMA, koristi pristup postupne izgradnje rezultata akumulacijom sume u varijabli označenoj sa s.

Tabela 2.1

```
    SUMA (n)
1   s = 0
2   for i = 1 to n do
3       s = s + i
4   end_for
5   return s
```

Drugi mogući pristup se temelji na primjeni formule za sumu aritmetičkog niza, prema kojoj imamo:

$$s = \frac{n(n+1)}{2} \tag{2.2}$$

Algoritam koji koristi izraz 2.2 je prikazan u tabeli 2.2, u obliku funkcije SUMA-M.

Tabela 2.2

```
    SUMA-M (n)
1   s = n*(n+1)/2
2   return s
```

Algoritmi prikazani u tabelama 2.1 i 2.2 su ekvivalentni, jer za isti skup ulaznih objekata daju iste rezultate. Međutim, ključna razlika između ovih algoritama je **trajanje algoritamskog procesa**. U prvom algoritmu **for** petlja se izvršava *n* puta, pa ukupan broj operacija koje se izvršavaju unutar petlje ovisi o veličini broja *n*. U drugom algoritmu je potrebno obaviti jedno zbrajanje, jedno množenje, jedno dijeljenje, te jednu operaciju dodjele vrijednosti, bez obzira na veličinu broja *n*. Prema tome, trajanje algoritamskog procesa prvog algoritma je *proporcionalno* sa *n*, dok je trajanje drugog algoritamskog procesa *konstantno* bez obzira na veličinu broja *n*.

Ovaj jednostavan primjer ilustrira činjenicu da **često postoji više algoritama kojima se mogu riješiti dati problemi**. Za ilustraciju ove činjenice mogli bismo uzeti npr. i problem sortiranja, za koji postoji na desetine različitih algoritama. Postojanje različitih algoritama za iste probleme nameće i pitanje kriterija za prikladan izbor algoritma u datim okolnostima. Iako često jedan od bitnih kriterija može biti jednostavnost implementacije, ipak u velikom broju slučajeva ključni kriterij je efikasnost algoritma, koja se obično mjeri potrošnjom dva važna resursa: *vrijeme* i *memorijski prostor*. Uzimajući u obzir da je tehnološki napredak u proizvodnji različitih tipova memorije omogućio da memorija predstavlja sve manje ograničenje, vrijeme algoritamskog procesa zapravo postaje ključni kriterij pri dizajniranju i izboru algoritama.

Analizom vremenske složenosti nekog algoritma određujemo trajanje algoritamskog procesa. To trajanje ne izražavamo u konkretnim vremenskim jedinicama (milisekunde, sekunde, minute, itd.), nego **trajanje algoritamskog procesa izražavamo aproksimacijom broja elementarnih operacija, koje će taj algoritam izvesti**. Međutim, pošto je broj operacija u vezi sa stvarnim vremenom izvođenja algoritma, onda se ponekada koristi i termin vrijeme u kontekstu izražavanja vremenske složenosti. Prema tome, pri analizi vremenske složenosti algoritama, ne zanima nas stvarno vrijeme izvođenja algoritma na nekom konkretnom računaru, nego nas zapravo zanima relativna efikasnost algoritma kojim rješavamo neki dati problem.

Stvarno trajanje algoritamskog procesa, izraženo nekom vremenskom jedinicom, ovisi od sljedećih faktora:

- tehnoloških parametara koji ovise od konkretne arhitekture i organizacije računara na kojem se algoritam izvršava;
- kvalitete generiranog mašinskog koda od strane prevoditelja;
- ulaznih podataka;
- inherentne vremenske složenosti algoritma.

Prva dva kriterija su očito ovisna o tehnološkoj platformi koja omogućuje izvršavanje algoritama, te na sličan način imaju utjecaj na različite algoritme. Nameće se pitanje da li je vrijeme ključni kriterij i uz današnju tehnološku platformu koja uključuje:

- računarske arhitekture sa vrlo velikim frekvencijama takta, te uz različita unapređenja koja znatno poboljšavaju performanse mjerene različitim parametrima;
- intuitivna grafička korisnička sučelja;
- različite tipove računarskih mreža;
- objektno orijentirane sisteme.

Na gornje pitanje odgovor je potvrdan, prije svega zbog sljedećeg razloga:

- Iako će se većina implementacija aplikacija temeljiti na brzoj hardverskoj platformi, grafičkom korisničkom sučelju, umrežavanju, itd., ipak bitni sastavni elementi većine aplikacija, osim onih najjednostavnijih, su različiti algoritamski sadržaji.

Nadalje, ako aplikacije čak i ne uključuju algoritamske sadržaje, one se zapravo temelje na algoritmima zbog sljedećih razloga:

- Dizajn i implementacija bilo kojeg grafičkog sučelja se u biti temelje na algoritmima.
- Rutiranje u računarskim mrežama se temelji na algoritmima.
- Aplikacije su pisane u nekom od programskih jezika, a to znači da će se koristiti kompajler, interpreter ili možda asembler, koji se u velikoj mjeri temelje na algoritmima.

Prema tome, možemo zaključiti da su **algoritmi zapravo jezgra većine tehnologija** korištenih u današnjim računarima. Nadalje, konstantnim napredovanjem tih tehnologija se zapravo rješavaju problemi sa sve većim veličinama ulaza, a efikasnost postaje izuzetno važna upravo za velike veličine ulaza. Iako je sa današnjom tehnologijom moguće obaviti veliki spektar zadataka bez poznavanja algoritama i struktura podataka, ipak je solidno znanje algoritama i struktura podataka izuzetno bitno, da bi se na profesionalan način kreirala rješenja za različite probleme, koji se javljaju kod dizajniranja i implementacije odgovarajućih programskih sistema.

Prije analize vremenske složenosti algoritama, potrebno je usvojiti model implementacijske tehnologije. Obično se kao model pretpostavlja tzv. **generički jednoprocesorski sistem** sa RAM modelom izračunavanja. Ovaj model podrazumijeva da se instrukcije izvršavaju jedna za drugom sekvencijalno, bez mogućnosti izvršavanja bilo kojih konkurentnih operacija. RAM model uključuje instrukcije koje se mogu naći i kod stvarnih računara: aritmetičke (zbrajanje, oduzimanje, množenje, dijeljenje, ostatak, itd.), pomjeranje podataka (load, store, copy, itd.) i kontrolu toka programa (uvjetna i bezuvjetna grananja, pozivi podrutina, itd.). Isto tako, usvojeni RAM model ne uključuje tzv. memorijsku hijerarhiju, a to znači da analizom nisu obuhvaćeni model *cash* memorije ili model *virtualne* memorije, iako su navedene memorije prisutne u stvarnim računarima. Uključivanje memorijske hijerarhije bi rezultiralo puno kompleksnijim modelom. Međutim, ako uzmemo u obzir da je cilj analize da se dobije uvid u najvažnija svojstva algoritama, onda ipak možemo reći da analiza koja se temelji na RAM modelu daje jako dobra

predviđanja o ponašanju algoritama. Strogo formalno govoreći, RAM modelom bi bilo potrebno precizno definirati skup instrukcija i vrijeme njihovog izvršavanja. Međutim, takav pristup bi bio preopširan i neprikladan, jer ne bi doprinosio da se dobije uvid u ključna svojstva nekog algoritma. Zato se vrijeme izvršavanja procjenjuje na način da se ne uzimaju u obzir stvarna vremena izvršavanja pojedinih naredbi iz algoritma, nego se vrijeme izvršavanja svodi na jedinično vrijeme izvršavanja apstraktne operacije. Ovakav pristup zapravo omogućuje analizu algoritama, koja je nezavisna od konkretne računarske platforme na kojoj će se izvoditi dotični algoritam.

Analiza vremenske složenosti, čak i jednostavnih algoritama i uz ograničenja postavljena usvajanjem RAM modela, može biti prilično kompleksna, jer zahtijeva korištenje matematičkih alata kao što su: kombinatorika, račun vjerojatnosti, različite algebarske transformacije, itd. Osim toga, analiza algoritama često zahtijeva i sposobnost da se identificiraju najznačajniji članovi u formulama. Pošto ponašanje nekog algoritma može biti različito za sve moguće ulazne podatke, potrebno je osmisliti jedan jednostavan način za sažet opis tog ponašanja, uz korištenje lako razumljivih formula.

Određivanje efikasnosti nekog algoritma se svodi na izračun broja elementarnih operacija pri izvođenju algoritamskog procesa, u zavisnosti od veličine ulaza u algoritam. U analizi ćemo koristitu oznaku n za **veličinu ulaza**. Značenje pojma veličina ulaza zavisi od problema koji se razmatra. Ako je npr. riječ o sortiranju elemenata nekog niza, onda za veličinu ulaza uzimamo broj elemenata niza, koji je potrebno sortirati. Kao rezultat postupka analize, najčešće se dobija izraz koji se sastoji od jednog ili više članova koji zavise od n. Na primjer, kao rezultat analize možemo dobiti izraz, koji ima sljedeći oblik:

$$T(n) = an^2 + bn + c, \tag{2.3}$$

gdje:

- a, b i c označavaju neke konstante;
- $T(n)$ označava broj elementarnih operacija, koje algoritam treba izvesti za veličinu ulaza n.

Pošto nas kao rezultat analize zapravo zanima samo procjena performansi algoritma za velike ulaze n, onda lako možemo identificirati da je u izrazu 2.3, za velike vrijednosti n, dominantan prvi član, pa se kaže da je složenost reda n^2, pri čemu se zanemaruje konstantni faktor a, koji stoji uz najviši stepen, kao i ostala dva člana nižeg stepena, čiji doprinos je zanemarljiv kada n postaje vrlo veliko. Obično se za izražavanje vremenske složenosti algoritama koristi tzv. *big-O* notacija, prema kojoj se složenost reda n^2 zapisuje kao $O(n^2)$. Formalna definicija za *big-O* notaciju je data i opisana u odjeljku 2.3.1.

2.1.1 Ponašanje algoritama za različite ulaze

Ulazi u algoritam imaju vrlo važnu ulogu pri analizi algoritama, zato što upravo ulazi definiraju koji će biti put izvršavanja algoritamskog procesa. Pretpostavimo da je potrebno pronaći maksimalnu vrijednost u nekom nizu K. Algoritam je prikazan u tabeli 2.3, u obliku funkcije MAX-ELEMENT. Radi ilustracije različitog ponašanja algoritama za različite ulazne podatke, fokusirajmo se samo na operacije dodjele, kojima se varijabli *max* dodjeljuje vrijednost nekog elementa iz niza K (linije 1 i 4). Ako pretpostavimo da je niz sortiran u opadajućem poretku, onda će se naredba dodjele kojom se ažurira privremena vrijednost *max* izvršiti samo jednom prije izvršavanja **for** petlje (linija 1). S druge strane,

ako je niz sortiran u rastućem poretku, onda će se izvršiti jedna operacija dodjele prije **for** petlje i n-1 operacija dodjele unutar navedene petlje (linija 4). Dakle, ako pri razmatranju vremenske složenosti algoritma uzmemo samo jedan mogući primjer ulaza, tada nećemo dobiti potpunu sliku o ponašanju tog algoritma. Ako pretpostavimo da je najveći element na prvoj poziciji, onda možemo reći da će za takve ulaze naredba dodjele biti izvršena samo jednom. Nadalje, ako pretpostavimo da se najveća vrijednost nalazi na drugoj poziciji, tada će se spomenute operacije dodjele izvršiti tačno dva puta. Međutim, operacije dodjele će se izvršiti dva puta i za određene ulaze kod kojih je najveći element smješten na pozicijama s indeksima 3..9. Na primjer, za ulazni niz K[0..9]={20, 3, 8, 12, 17, 11, 9, 23, 19, 15}, broj dodjela, kojima se varijabli *max* dodjeljuje naka vrijednost iz niza *K*, je također 2, iako se najveći element 23 nalazi na poziciji sa indeksom 7. Analizom funkcije MAX-ELEMENT možemo zaključiti da će broj operacija dodjele biti između 1 i n. Iz ovog primjera možemo vidjeti jedno vrlo važno svojstvo algoritama, prema kojem, općenito govoreći, ponašanje algoritma, koje izražavamo trajanjem algoritamskog procesa, ovisi o ulaznim podacima koje taj algoritam obrađuje. Dakako, treba istaknuti da su česte i situacije kod kojih je trajanje algoritamskog procesa konstantno, bez obzira na različite ulazne podatke.

Tabela 2.3

```
    MAX-ELEMENT (K)
1   max = K[0]
2   for i = 1 to n-1 do
3       if (K[i] > max) then
4           max = K[i]
5       end_if
6   end_for
7   return max
```

Napomenimo da smo prethodno razmatrali samo operacije dodjele, kojima se mijenja vrijednost varijable *max*, dok smo zanemarili ostale operacije dodjele, kao što su npr. operacije dodjele pri promjeni vrijednosti kontrolne varijable i u petlji **for** (linija 2). Isto tako, u ovom razmatranju smo zanemarili operacije usporedbe, kao npr. operacije usporedbe u liniji 3. Nadalje, treba imati na umu da se pri zapisu nekog algoritma u pseudokodu, a ponekada i u nekom stvarnom programskom jeziku, zbog prirode skrivanja implementacijskih detalja, ne mogu uočiti neke skrivene operacije. Na primjer, u pseudokodu za **for** petlju je definirana promjena vrijednosti kontrolne varijable i, u rasponu od 1 do n-1 (linija 2). Iako se u zapisu u liniji 2 eksplicitno ne vidi i operacija usporedbe, jasno je da će u stvarnoj implementaciji postojati i ta operacija, kojom se treba utvrditi da li je kontrolna varijabla i izašla izvan definiranog raspona petlje, odnosno da li je $i > n$-1. Nadalje, istaknimo da broj operacija koje smo izostavili zapravo linearno raste sa n pri izvođenju algoritamskog procesa, te da razmještaj podataka u ulaznom nizu K nema utjecaja na njihov broj, pa je to još jedno opravdanje zašto te operacije nisu bile u fokusu prethodnog razmatranja.

2.2 Najbolji, najgori i prosječni slučaj

Pri analizi vremenske složenosti algoritama, potrebno je odlučiti koje operacije su bitne sa stanovišta efikasnosti algoritama. Postoje različiti tipovi operacija, kao npr. operacije usporedbi, operacije premještanja elemenata sa jedne lokacije na drugu, aritmetičke operacije, operacije dodjele vrijednosti, itd. Koje operacije će se tretirati kao relevantne pri nekoj analizi vremenske složenosti, ovisi o konkretnoj situaciji, tj. o konkretnoj implementaciji algoritma. Na primjer, u algoritmima pretraživanja, važan korak je usporedba dviju vrijednosti da se odredi da li je to vrijednost koja se traži. Ili, na primjer, u algoritmima sortiranja, operacijama usporedbi dviju vrijednosti se utvrđuje da li se dotične vrijednosti nalaze u ispravnom poretku. Operacije usporedbi uključuju: „jednako“, „nije jednako“, „manje od“, „veće od“, „manje ili jednako od“, „veće ili jednako od“. Aritmetičke operacije možemo svrstati u dvije grupe: *aditivne* i *multiplikativne*. Aditivne operacije uključuju: zbrajanje, oduzimanje, inkrementiranje i dekrementiranje. Multiplikativne operacije uključuju: množenje, dijeljenje i modulo. Ponekada je prikladno različite tipove operacija razmatrati odvojeno, zbog toga što operacije jednog tipa mogu zahtijevati više vremena za njihovo izvođenje, od operacija drugog tipa. Suprotno tome, u određenim situacijama je opravdano sve tipove operacija razmatrati zajedno.

Specijalan slučaj je kada imamo cjelobrojno množenje ili dijeljenje sa potencijom broja 2, jer se tada takve operacije mogu svesti na operaciju pomaka, pa se takve operacije zapravo izvršavaju vrlo brzo. Ipak, općenito govoreći, takve operacije se relativno rijetko susreću u algoritmima kao operacije koje bitno utiču na ukupnu efikasnost nekog algoritma, pa se obično pri analizi vremenske složenosti zanemaruju.

Prethodno smo vidjeli da trajanje algoritamskog procesa može varirati ne samo u ovisnosti od veličine ulaza, nego i o konkretnim vrijednostima ulaznih podataka. Zbog toga se pri analizi algoritama mogu razmatrati **najbolji, najgori i prosječni** slučaj.

2.2.1 Najbolji slučaj

Najbolji slučaj za neki algoritam predstavlja ulaz za koji je trajanje algoritamskog procesa najkraće. Takav ulaz je kombinacija vrijednosti za koju se algoritam izvrši u najkraćem vremenu. Na primjer, najbolji slučaj kod sekvencijalnog algoritma pretraživanja je situacija kada se vrijednost koja se traži nalazi na prvoj poziciji u nizu. U tom slučaju se izvršava samo jedna usporedba, pa je za izvođenje algoritma potrebno neko konstantno vrijeme, bez obzira na dužinu niza koji se pretražuje. Pošto analiza najboljeg slučaja ne omogućuje dovoljno dobar uvid u ponašanje algoritma, obično je analiza takvog slučaja od najmanjeg interesa, pa se ona najrjeđe i radi.

2.2.2 Najgori slučaj

Najgori slučaj kod analize vremenske složenosti nekog algoritma je gornja granica trajanja algoritamskog procesa za bilo koje vrijednosti ulaznih podataka, što znači da se ovom analizom dobija *garantirana* efikasnost algoritma. Analiza najgoreg slučaja zahtijeva da identificiramo ulazne vrijednosti za koje je trajanje algoritamskog procesa najduže. Na primjer, najgori slučaj u algoritmima pretraživanja se događa kada vrijednost koju pokušavamo pronaći nije niti prisutna u nizu. Napomenimo da su pokušaji pronalaženja nepostojećih vrijednosti česta pojava u mnogim stvarnim situacijama pretraživanja

podataka. Ako se ograničimo samo na uspješna pretraživanja, koja završavaju pronalaženjem tražene vrijednosti, onda se najgori slučaj pri sekvencijalnom pretraživanju pojavljuje u situaciji kada se tražena vrijednost nalazi na zadnjoj poziciji u nizu. Pri izvršavanju nekih algoritama, najgori slučaj se pojavljuje relativno često.

2.2.3 Klase ulaza i prosječni slučaj

Da bismo dobili potpunu sliku o ponašanju nekog algoritma, potrebno je razmotriti sve mogućnosti ulaza. Međutim, s porastom broja elemenata n, vrlo brzo raste broj mogućih ulaza. Na primjer, pri sekvencijalnom pronalaženju traženog elementa u nekom nizu K, uz pretpostavku da taj niz ima 10 elemenata, postoji 10! (3 628 800) različitih načina na koje elementi u nizu mogu biti razmješteni. Da bismo smanjili broj mogućnosti koje treba razmotriti, često je prikladno podijeliti sve moguće ulaze u različite **klase ulaza**, koje se temelje na tome kako se algoritam ponaša za svaki ulaz. Tada je, da bismo dobili potpuniju sliku o ponašanju algoritma, potrebno razmotriti sve klase ulaznih skupova. Ako pretpostavimo da je traženi element na prvoj poziciji, onda možemo reći da postoji 362 880 različitih ulaza. Za sve te ulaze će biti potrebna samo jedna operacija usporedbe da bi se pronašao traženi element, pa sve dotične ulaze možemo tretirati kao jednu klasu ulaza. Isto tako, postoji 362 880 različitih ulaza kod kojih je najveći element smješten na drugoj poziciji, itd. Razmotrimo sada algoritam sekvencijalnog pretraživanja prikazan u tabeli 2.4, u obliku funkcije TRAZI-SEKV, pri čemu ćemo se fokusirati samo na operacije usporedbi. Možemo napraviti dvije analize slučaja za taj algoritam. U prvoj analizi možemo pretpostaviti da je pretraživanje uvijek uspješno, odnosno da se traženi ključ nalazi u nizu. U drugoj analizi možemo pretpostaviti da se ponekada traženi ključ ne nalazi u nizu.

Tabela 2.4

```
    TRAZI-SEKV (K, k)
1   i = 0
2   while (i ≤ n-1) do
3      if (K[i] == k) then
4         return i
5      else
6         i = i+1
7      end_if
8   end_while
9   return -1
```

Ako se traženi ključ nalazi u nizu, postoji n pozicija na kojima se on može nalaziti. Kao što smo prethodno već rekli, ako je traženi ključ na prvom mjestu, onda je potrebna jedna operacija usporedbe (linija 3). Nadalje, ako je traženi ključ na drugom mjestu, onda su potrebne dvije operacije usporedbi. Općenito, ako se traženi ključ nalazi na nekoj i-toj poziciji, biti će potrebno i operacija usporedbi (i=1, 2, ... n). Prema tome, sve ulaze u algoritam možemo grupirati u n klasa ulaza. U prvu klasu spadaju svi ulazi kod kojih je traženi ključ na prvoj poziciji, u drugu klasu spadaju svi ulazi kod kojih je traženi ključ na drugoj poziciji, itd. Prema tome, ako pretpostavimo da je vjerojatnost pojavljivanja svake klase ulaza jednaka, onda je prosječan broj operacija poređenja:

$$T(n) = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1+2+\ldots+n}{n} = \frac{1}{n}\cdot\frac{n(n+1)}{2} = \frac{n+1}{2} \tag{2.4}$$

Pretpostavimo sada do postoji i mogućnost da se traženi ključ ne nalazi u nizu. Tada postoji n+1 klasa ulaza (n klasa ulaza kada je ključ prisutan u nizu i jedna klasa ulaza kada nije prisutan). U slučaju da traženi ključ nije u nizu, potrebno je n operacija poređenja. Ako pretpostavimo da su sve klase ulaza jednako vjerojatne, onda je prosječan broj operacija poređenja sljedeći:

$$T(n) = \frac{1}{n+1}\left(\sum_{i=1}^{n} i + n\right) = \frac{1}{n+1}\cdot\frac{n(n+1)}{2} + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1} \tag{2.5}$$

Odnosno, za velike vrijednosti n možemo pisati:

$$T(n) \approx \frac{n}{2} + 1 \tag{2.6}$$

Prema tome, uključivanjem mogućnosti da se traženi ključ ne nalazi u nizu, prosječan broj operacija se povećava za približno 1/2, u odnosu na slučaj kada isključimo mogućnost da se traženi ključ ne nalazi u nizu. Ako uzmemo u obzir da nas broj operacija zanima prije svega za velike vrijednosti n, onda je povećanje za 1/2 praktički beznačajno.

Na temelju prethodnog primjera ćemo poopćiti način određivanja **prosječnog slučaja**. Ako sa $T(n)$ označimo funkciju kojom izražavamo broj operacija u ovisnosti o veličini ulaza n, onda se, uz određene pretpostavke, vremenska složenost za prosječni slučaj dobija na sljedeći način:

1. Utvrditi broj različitih grupa m, odnosno broj različitih klasa ulaza, u koje se mogu svrstati svi mogući ulazi za neki algoritam.
2. Za svaku identificiranu moguću grupu ulaza u_i, treba odrediti broj operacija $t(u_i)$, koje se izvršavaju u algoritmu.
3. Zbrojiti operacije za sve grupe ulaza i zbroj podijeliti sa brojem grupa:

$$T(n) = \frac{1}{m}\sum_{i=1}^{m} t(u_i), \tag{2.7}$$

Svi ulazi u nekoj grupi bi trebali imati isti broj operacija. Ako to nije slučaj, onda je grupu potrebno podijeliti na dvije podgrupe. Međutim, bitno je istaknuti da prethodno opisani postupak pretpostavlja da su sve grupe ulaza jednako vjerojatne, što dakako često nije slučaj. Ako je distribucija vjerojatnosti pojavljivanja ulaza poznata, onda prosječno očekivani broj operacija $T(n)$, pri izvođenju nekog algoritma, možemo izračunati kao:

$$T(n) = \sum_{i=1}^{m} p(u_i)t(u_i), \tag{2.8}$$

gdje je:

- n veličina ulaza;
- m broj različitih klasa ulaza;

- $t(u_i)$ broj operacija za klasu ulaza u_i.
- $p(u_i)$ vjerojatnost pojavljivanja klase ulaza u_i, pri čemu je:

$$p(u_i) \geq 0, \sum_{i=1}^{m} p(u_i) = 1.$$

Prema tome, pri analizi vremenske složenosti prosječnog slučaja, nije potrebno razmatrati sve moguće ulaze, nego je potrebno identificirati različite klase ulaza i razmotriti ponašanje algoritma za identificirane klase. Još jednom istaknimo da svi ulazi, koji pripadaju jednoj identificiranoj klasi, bi trebali imati isti broj operacija.

Analizom prosječnog slučaja se dobijaju očekivana vremena trajanja algoritamskog procesa. Najčešće je ova analiza i najzahtjevnija, prije svega zbog toga što često nije tako jednostavno ustanoviti šta zapravo sačinjava prosječan slučaj, jer sve ulazne vrijednosti podataka ne moraju imati iste vjerojatnosti. Pošto je potrebno na neki način izraziti prosječno ponašanje algoritma za sve moguće ulazne vrijednosti podataka, često se koristi analiza sa slučajnim vrijednostima. S druge strane, u nekim analizama je opravdano pretpostaviti da sve ulazne vrijednosti podataka imaju istu vjerojatnost.

Na kraju ćemo napraviti još jednu analizu, ali uz pretpostavku da vjerojatnosti nisu jednake za sve klase ulaza i uz pretpostavku da je traženi ključ prisutan u nizu (pretraživanje je uspješno). Prepostavimo, na primjer, da je distribucija vjerojatnosti za pojedine klase ulaza sljedeća:

- vjerojatnost da je traženi ključ smješten na prvoj poziciji je 1/2, $p(u_1)=1/2$
- vjerojatnost da je traženi ključ smješten na drugoj poziciji je 1/4, $p(u_2)=1/4$
- vjerojatnost da se najveći broj nalazi na nekoj od preostalih pozicija je jednaka i iznosi:

$$p(u_i) = \frac{1 - \frac{1}{2} - \frac{1}{4}}{n - 2} = \frac{1}{4(n - 2)}, \; i = 3, 4, \ldots, n. \tag{2.9}$$

Prema tome, na temelju izraza 2.8, prosječan broj operacija poređenja je:

$$T(n) = \sum_{i=1}^{n} p(u_i) \cdot t(u_i) = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \sum_{i=3}^{n} \frac{1}{4(n-2)} \cdot i \tag{2.10}$$

Odnosno, za prosječan broj operacija $T(n)$ imamo:

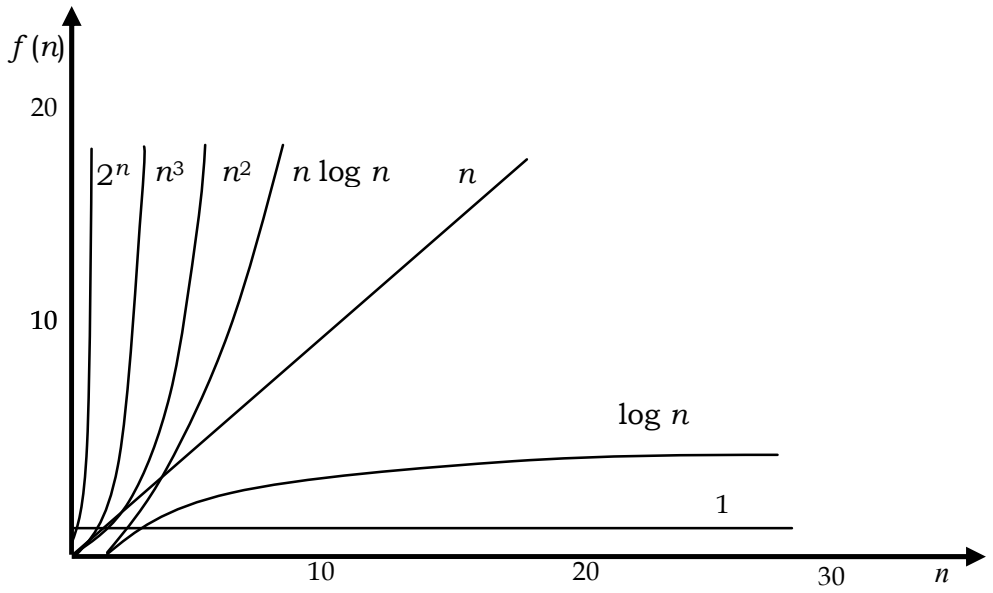
$$T(n) = 1 + \frac{\frac{n(n+1)}{2} - 3}{4(n-2)} = 1 + \frac{n(n+1) - 6}{8(n-2)} = 1 + \frac{(n-2)(n+3)}{8(n-2)} = 1 + \frac{n+3}{8} \tag{2.11}$$

Dakle, za velike vrijednosti n, prosječan broj operacija poređenja je približno $n/8$. Ako uzmemo u obzir da je kod uniformne distribucije vjerojatnosti broj operacija poređenja približno $n/2$ za velike vrijednosti n, onda zaključujemo da je prosječan broj operacija poređenja u ovom slučaju 4 puta manji u odnosu na slučaj sa uniformnom distribucijom vjerojatnosti.

2.3 Asimptotska analiza i klasifikacija stupnjeva rasta

Pri analizi vremenske složenosti algoritama ne interesira nas tačan broj operacija koje će neki algoritam izvršiti, već nas zanima *stupanj* tog rasta sa porastom veličine ulaza. Nadalje, ne zanima nas šta se događa sa ponašanjem algoritma za male ulaze, nego nas zapravo zanima šta se događa sa efikasnošću algoritma za velike ulaze. Dakle, kada analiziramo efikasnost nekog algoritma zapravo nas zanima u koju klasu stupnja rasta taj algoritam spada, a ne tačan broj operacija koje će taj algoritam izvršiti.

Bitno je napomenuti da su razlike velike kod različitih klasa algoritama upravo za velike veličine ulaza, odnosno za velike vrijednosti n. Osim toga, ako smo analizom utvrdili da je vremenska složenost kombinacija dvije ili više klasa složenosti, onda zbog velikog stupnja rasta pojedinih klasa, te klase počinju dominirati nad ostalim funkcijama koje imaju sporiji rast. To znači da se određivanjem dominantnog člana ostali članovi mogu zanemariti. Na primjer, ako smo analizom utvrdili da je broj operacija poređenja nekog algoritma $T(n)=n^3-5n$, onda nakon neke vrijednosti n_0, član n^3 počinje dominirati tako da dotični algoritam zapravo spada u klasu algoritama koji imaju stupanj n^3. Neke uobičajene klase algoritama su prikazane na slici 2.1



Slika 2.1. Neke uobičajene klase stupnjeva rasta

2.3.1 Notacija *big-O*

Jedna od najčešće korištenih notacija za klasifikaciju algoritama s obzirom na stupanj rasta vremenske složenosti je tzv. *big-O* notacija. Za funkciju $f(n)$ kažemo da je $O(g(n))$, ako postoje pozitivne konstante c i n_0 tako da je ispunjeno:

$$f(n) \le cg(n), \ za\ svako \quad n \ge n_0. \tag{2.12}$$

Ako, na primjer, pretpostavimo da funkcija $f(n)$ ima sljedeći oblik:

$$f(n) = 3n^2 + 5n + 1, \tag{2.13}$$

onda uz pretpostavku da smo uzeli $g(n)=n^2$, dobijamo sljedeću nejednačinu:

$$3n^2 + 5n + 1 \le cn^2. \tag{2.14}$$

Odnosno, ekvivalentna nejednačina je:

$$3 + \frac{5}{n} + \frac{1}{n^2} \le c. \tag{2.15}$$

Pošto gornja nejednačina ima dvije nepoznate, postoje različiti parovi za c i n_0, koji zadovoljavaju tu nejednačinu. U tabeli 2.3 je prikazano nekoliko kombinacija.

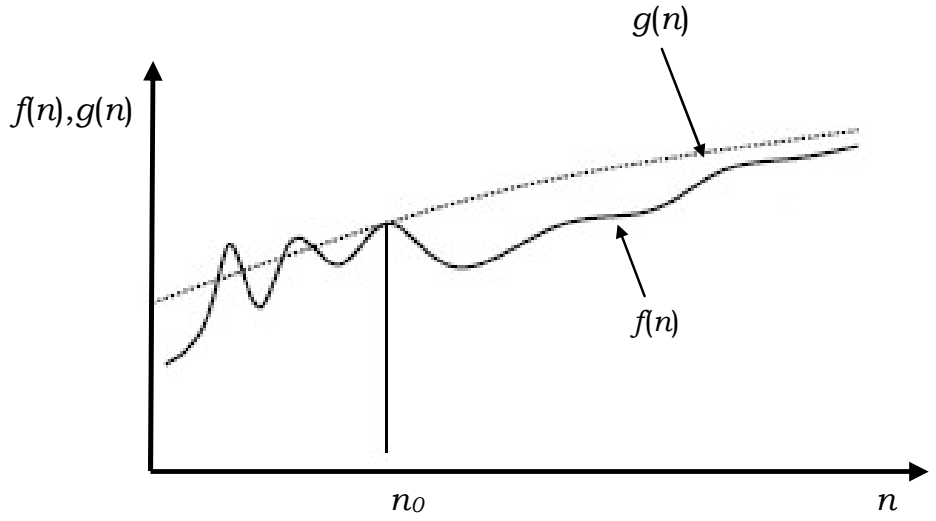
Tabela 2.3

n_0	1	2	3	4	....	→	∞
c	≥9	≥23/4	≥43/9	≥69/16	....	→	3

Za fiksiranu funkciju g, može se identificirati beskonačno parova za c i n_0. Prema tome, nejednačinom 2.12 se zapravo definira da je vrijednost funkcije $g(n)$, pomnožena konstantom c, veća od $f(n)$ za sve vrijednosti $n \ge n_0$, a to znači za gotovo sve vrijednosti od n. Iz tabele 2.3 vidimo da sa približavanjem vrijednosti n_0 beskonačnosti, vrijednost c teži broju 3. Općenito, možemo pisati:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c.$$

Možemo zaključiti da se *big*-O notacijom zapravo definira **gornja granica složenosti**, jer je $f(n)$ asimptotski ograničeno sa $g(n)$ i sigurno nema veći red veličine rasta sa porastom veličine ulaza n. Na slici 2.2 je prikazana grafička interpretacija značenja funkcija $f(n)$ i $g(n)$, u definiciji *big-O* notacije.



Slika 2.2. Grafički prikaz značenja funkcija $f(n)$ i $g(n)$ u definiciji *big-O* notacije

Napomenimo da može postojati beskonačno puno funkcija $g(n)$ za datu funkciju $f(n)$. Na primjer, za funkciju $f(n)$ definiranu izrazom 2.13 pored izbora za $g(n) = n^2$, u skladu s definicijom *big-O* notacije mogle bi se izabrati i sljedeće funkcije $g(n) = n^3$, $g(n) = n^4$, $g(n) = n^5$, itd. Da bi se izbjegle različite nejasnoće, za $g(n)$ se izabire funkcija s najmanjim rastom.

Istaknimo neka važna svojstva koje ima notacija *big-O*:

1. Ako je $f(n)$ reda $O(h(n))$ i ako je $g(n)$ reda $O(h(n))$, onda je i $f(n)+g(n)$ također reda $O(h(n))$.

Dokaz: Pema izrazu 2.12, ako je $f(n)$ reda $O(h(n))$, onda postoje pozitivni brojevi c_1 i n_1 takvi da je ispunjeno $f(n) \le c_1 h(n)$, za $n \ge n_1$. Isto tako, prema izrazu 2.12, ako je $g(n)$ reda $O(h(n))$, to znači da postoje brojevi c_2 i n_2 takvi da je ispunjeno $g(n) \le c_2 h(n)$, za $n \ge n_2$. Iz prethodne dvije nejednakosti zbrajanjem lijevih i desnih strana dobijamo $f(n) + g(n) \le (c_1 + c_2) h(n)$, za $n \ge max(n_1, n_2)$. Ako uzmemo $c = c_1 + c_2$ i $n_0 = max(n_1, n_2)$, dobijamo $f(n) + g(n) \le ch(n)$, za $n \ge n_0$, što znači da je $f(n) + g(n)$ reda $O(h(n))$.

2. Funkcija an^k je reda $O(n^k)$

Dokaz: Iz izraza 2.12 proizlazi da je potrebno naći c i n_0, tako da vrijedi $an^k \le cn^k$ za svaki $n \ge n_0$. Dovoljno je uzeti $c \ge a$, pa je traženi uvjet ispunjen.

3. Funkcija n^k je reda $O(n^{k+j})$ za sve pozitivne vrijednosti j.

Dokaz: Ako, na primjer, uzmemo $c = n_0 = 1$, traženi uvjet $n^k \le n^{k+j}$ je ispunjen za sve vrijednosti $n \ge n_0$ i sve pozitivne vrjednosti j.

Iz svojstava 1, 2 i 3 proizlazi da vremenska složenost polinomskog oblika:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0,$$

izražena *big-O* notacijom, ima sljedeći oblik:

$$f(n) = O(n^k)$$

Naravno, i u slučaju polinomskog oblika funkcije vremenske složenosti $f(n)$ vrijedi da je $f(n)$ istovremeno i reda $O(n^{k+j})$, za sve pozitivne vrijednosti j.

4. Ako je $f(n)$ reda $O(g(n))$, a $g(n)$ reda $O(h(n))$, onda je i $f(n)$ isto tako reda $O(h(n))$. Ovo svojstvo se naziva svojstvo *tranzitivnosti*.

Dokaz: Prema izrazu 2.12, $f(n)$ je reda $O(g(n))$, ako postoje pozitivni brojevi c_1 i n_1, takvi da je $f(n) \le c_1 g(n)$ za sve $n \ge n_1$. Isto tako, prema izrazu 2.12, $g(n)$ je reda $O(h(n))$, ako postoje pozitivni brojevi c_2 i n_2, takvi da je $g(n) \le c_2 h(n)$ za sve $n \ge n_2$. Prema tome, dobijamo da je $c_1 g(n) \le c_1 c_2 h(n)$ za sve $n \ge n_2$. Nadalje, dobijamo $f(n) \le c_1 g(n) \le c_1 c_2 h(n)$ za sve $n \ge n_0$, gdje je n_0 veći od brojeva n_1 i n_2, $n_0 = max(n_1, n_2)$. Ako uzmemo $c = c_1 c_2$, dobijamo $f(n) \le ch(n)$, za sve $n \ge n_0$, odnosno dobijamo da je $f(n)$ reda $O(h(n))$.

5. Ako je $f(n) = cg(n)$, onda je $f(n)$ reda $O(g(n))$.

Jedna od najvažnijih funkcija pri izražavanju efikasnosti algoritama je **logaritamska funkcija**.

6. Funkcija $f(n)=\log_a n$ je reda $O(\log_b n)$, za bilo koje pozitivne brojeve a i b, $b \neq 1$. Ovo svojstvo zapravo pokazuje da sve logaritamske funkcije imaju isti stepen rasta. Prema tome, baza logaritma je nebitna u kontekstu *big*-O notacije, što znači da uvijek možemo koristiti jednu bazu logaritma.

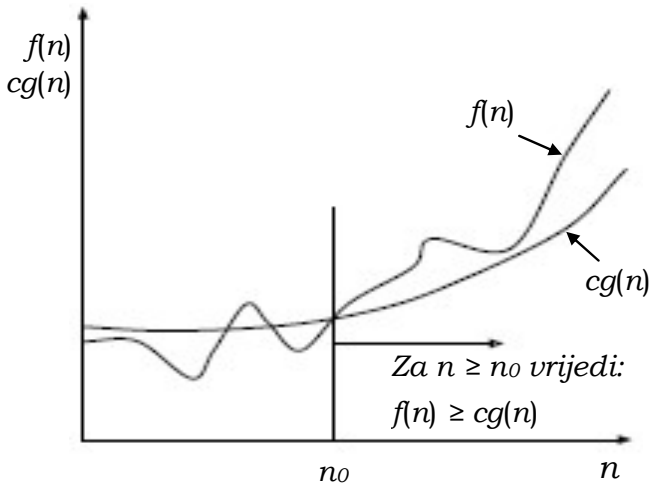
2.3.2 Notacija *big*-Ω

Izražavanjem vremenske složenosti pomoću *big*-O notacije dobijamo uvid o ponašanju algoritma prije svega za velike vrijednosti n. Na primjer, za $n = 100$ zapravo je efikasniji algoritam sa vremenskom složenošću $f(n)=n^3$, nego algoritam sa vremenskom složenošću $f(n)=200n^2$, iako je prvi algoritam reda $O(n^3)$, a drugi je reda $O(n^2)$.

Zbog navedenih razloga se ponekada koristi i **donja granica složenosti** algoritama, koja se izražava *big*-Ω notacijom. Za funkciju $f(n)$ kažemo da je $\Omega(g(n))$, ako postoje pozitivni brojevi c i n_0 takvi da je:

$$f(n) \geq cg(n), \text{ za svako } \quad n \geq n_0. \tag{2.16}$$

Na slici 2.3 je grafički ilustrirano značenje funkcija $f(n)$ i $g(n)$ u definiciji notacije *big*-Ω.



Slika 2.3. Grafički prikaz značenja funkcija $f(n)$ i $g(n)$ u definiciji *big*- Ω notacije

2.3.3 Notacija *big*-Θ

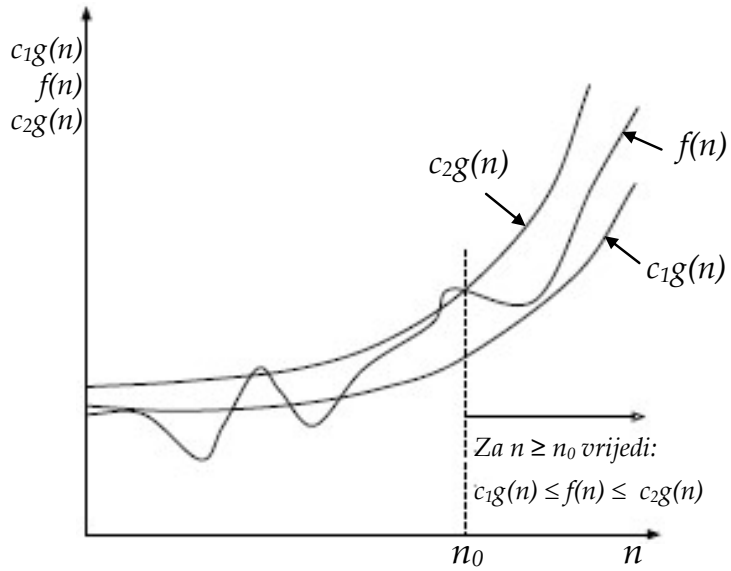
Za predstavljanje asimptotske granice performansi nekog algoritma i sa gornje i sa donje strane, koristi se notacija *big*-Θ. Za funkciju $f(n)$ kažemo da je $\Theta(g(n))$, ako postoje pozitivni brojevi c_1, c_2 i n_0 takvi da je:

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \text{ za svako } \quad n \geq n_0. \tag{2.17}$$

Iz 2.17 vidimo da je $f(n) = \Theta(g(n))$, ako je $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$, što u biti predstavlja asimptotsku granicu performansi i sa gornje i sa donje strane.

Prethodno smo vidjeli da funkcija $f(n)$ definirana izrazom 2.3 ima složenost izraženu *big*-O notacijom $f(n)=O(g(n))$, gdje $g(n)$ mogu biti funkcije n^2, n^3, n^4, itd. Međutim, jedina od navedenih funkcija za $g(n)$, kojom možemo izraziti vremensku složenost funkcije u *big*-Ω notaciji $f(n) = \Omega(g(n))$, je funkcija $g(n) = n^2$. To znači da možemo izraziti složenost

funkcije $f(n)$ i u *big-*Θ notaciji, $f(n) = \Theta(n^2)$. Značenje notacije *big-*Θ je grafički ilustrirano na slici 2.4.



Slika 2.4. Grafički prikaz značenja funkcija $f(n)$ i $g(n)$ u definiciji *big-*Θ notacije

Istaknimo da se u literaturi pojavljuju različiti zapisi kojima se prikazuje da je neka funkcija f reda $O(g)$, $\Omega(g)$ ili $\theta(g)$. Kao što smo već prethodno vidjeli, u ovoj knjizi je korišten zapis sljedećeg oblika:

$$f = O(g).$$

Međutim, ako imamo na umu da oznake $O(g)$, $\Omega(g)$ i $\theta(g)$ zapravo predstavljaju skupove različitih funkcija koje pripadaju klasi sa istim stupnjem rasta, onda je prihvatljiv i sljedeći zapis kojim prikazujemo da neka funkcija f pripada skupu $O(g)$:

$$f \in O(g).$$

2.3.4 Neke tipične klase stupnjeva rasta vremenske složenosti

U tabeli 2.4 su navedene najčešće klase stupnjeva rasta vremenske složenosti algoritama.

Tabela 2.4

Notacija	Naziv algoritama	Kratak opis
$O(1)$	**Konstantni algoritmi**	Kod ovih algoritama trajanje algoritamskog procesa je konstantno i ne ovisi o ulaznim podacima. Primjeri algoritma ove klase su: umetanje novog čvora na početak povezane liste, zamjena dva elementa u nizu, itd.
$O(\log n)$	**Logaritamski algoritmi**	Kod ovih algoritama problem se obično rješava redukcijom problema polovljenjem, sve dok se ne dođe do problema jedinične dužine. Primjer logaritamskog algoritma je binarno pretraživanje sortiranog niza.

(*nastavak tabele* 2.4)

$O(n)$	**Linearni algoritmi**	Ovi algoritmi su karakteristični za probleme kod kojih se rješenje dobija ponavljanjem naredbi u petljama, gdje je broj tih ponavljanja proporcionalan veličini ulaza. Primjer algoritma koji ima linearnu vremensku složenost je sekvencijalno pretraživanje nesortiranog niza.
$O(n \log n)$	**Linearno logaritamski algoritmi**	Ovi algoritmi su karakteristični za probleme koji se rješavaju polovljenjem, pri čemu se ipak obrađuju svi ulazni podaci. Neki efkasniji algoritmi sortiranja imaju ovu vremensku složenost.
$O(n^2)$	**Kvadratni algoritmi**	Ovi algoritmi najčešće imaju dvije ugnježđene petlje, pri čemu je broj izvršavanja svake od petlji proporcionalan sa n. Jednostavniji algoritmi sortiranja imaju ovu vremensku složenost.
$O(n^3)$	**Kubni algoritmi**	Ovi algoritmi najčešće imaju tri ugnježđene petlje, pri čemu je broj izvršavanja svake od petlji proporcionalan sa n.
$O(n^k)$	**Stepeni algoritmi**	Ovi algoritmi tipično imaju k ugnježđenih petlji, pri čemu je broj izvršavanja svake od petlji proporcionalan sa n. Kod ovih algoritama dolazi do vrlo brzog porasta trajanja algoritamskog procesa za velike veličine ulaznih podataka n.
$O(k^n),\ k>1$	**Eksponencijalni algoritmi**	Ovi algoritmi su karakteristični za probleme koji se rješavaju iscrpnim pretraživanjem svih mogućnosti. Takve algoritme nazivamo algoritmi „grube sile“. Trajanje algoritamskog procesa kod ovih algoritama raste izuzetno brzo sa porastom veličine ulaznih podataka n, pa često imaju samo teorijsku važnost za probleme velikih dimenzija.

Za svaku od navedenih klasa u tabeli 2.5 je naveden broj operacija i potrebno vrijeme za izvršavanje tih operacija, uz pretpostavku da računar na kojem se izvodi algoritam može obaviti 10^7 operacija u sekundi, odnosno deset operaciju u mikrosekundi (10 o/μsec).

Tabela 2.5

Klasa / veličina ulaza n	$O(1)$	$O(\log n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
10	1	3.32	33.2	10^2	10^3	1024
	0.1 μsec	0.3 μsec	3.3 μsec	10 μsec	100 μsec	1 msec
10^2	1	6.64	664	10^4	10^6	10^{30}
	0.1 μsec	0.7 μsec	66.4 μsec	1 msec	0.1 sec	$3.17 \cdot 10^{15}$ god.
10^3	1	9.97	$9.97 \cdot 10^3$	10^6	10^9	10^{301}
	0.1 μsec	1 μsec	1 msec	0.1 sec	1.67 min	$3.17 \cdot 10^{286}$ god.
10^4	1	13.3	$133 \cdot 10^3$	10^8	10^{12}	10^{3010}
	0.1 μsec	1.3 μsec	13.3 msec	0.17 min	1.16 dana	
10^5	1	16.6	$1.66 \cdot 10^6$	10^{10}	10^{15}	10^{30103}
	0.1 μsec	1.7 μsec	0.16 sec	16.7 min	3.17 god.	
10^6	1	19.93	$19.93 \cdot 10^6$	10^{12}	10^{18}	10^{301030}
	0.1 μsec	2 μsec	2 sec	1.16 dana	3170.9 god.	

Iz tabele 2.5 možemo vidjeti da npr. algoritmi složenosti $O(n^3)$ za veličinu ulaza 10^5 praktično imaju samo teoretski značaj, jer je potrebno više od 3 godine da bi se algoritam te složenosti izveo. Ili, na primjer, za veličinu ulaza 10^6, potrebno je više od 3170 godina. Još kompleksnija situacija je sa eksponencijalnim algoritmima, koji imaju složenost npr. $O(2^n)$. Iz tabele 2.5 možemo vidjeti da je već za veličinu ulaza 10^3 potrebno izvršiti 10^{301} operacija, dok bi za izvođenje tolikog broja operacija bilo potrebno $3.17 \cdot 10^{286}$ godina. To je nezamislivo mnogo vremena. Dakako, još kompleksnija situacija je za veličine ulaza 10^5 ili 10^6. Da bi dobili dojam o veličini nekih vremenskih intervala prikazanih u tabeli 2.5, spomenimo npr. da je starost planete Zemlje oko $4{,}5 \cdot 10^9$ godina. Ili npr. procjenjuje se da je ukupan broj atoma u svemiru između 10^{78} i 10^{81}.

2.4 Pitanja i zadaci za vježbu

1. Poredajte sljedeće izraze po stupnju rasta, od manjeg prema većem:

   $30n$ $\log_2 n$ $n!$ 4^n 5 $\log 5n$ $n^{3/5}$ $3n^2$

2. (*a*) Pretpostavimo da neki algoritam ima vremensku složenost $T(n) = 5\times 2^n$. Nadalje, pretpostavimo da je za veličinu ulaza n potrebno vrijeme za izvođenje tog algoritma jednako t. Sada pretpostavimo da koristimo računar, koji je 128 puta brži. Koja veličina ulaza bi se mogla obraditi na tom računaru u t sekundi?

   (*b*) Pretpostavimo da neki drugi algoritam ima vremensku složenost $T(n) = n^2$, te da izvođenje tog algoritma na nekom računaru za veličinu ulaza n uzima t sekundi. Pretpostavimo sada da je neki drugi računar 128 puta brži. Koja veličina ulaza se može obraditi na novom računaru u t sekundi.

   (*c*) Treći algoritam ima vremensku složenost $T(n)= 4n$. Izvođenje tog algoritma na nekom računaru traje t sekundi, za veličinu ulaza n. Kolika veličina ulaza se može obraditi u t sekundi na računaru koji je 128 puta brži.

3. Koristeći definicije za *big-O* i *big-Ω,* nađite gornju i donju granicu za sljedeće izraze:

   (*a*) $c_1 n$

   (*b*) $c_2 n^3 + c_3$

   (*c*) $c_4 n \log n + c_5 n$

4. Za sljedeće fragmente C++ programskog koda odredite u *big*-O notaciji vremensku složenost u ovisnosti od veličine n. Pretpostavite da su sve varijable tipa **int**.

   (*a*)
   ```
   a=b+c+2;
   d=2*(a+b);
   ```

   (*b*)
   ```
   s=0;
   for(int i=0; i<n; i++)
     s++;
   ```

   (*c*)
   ```
   s=0;
   for (int i=0; i<4; i++)
     for (int j=0; j<n; j++)
       s++;
   ```

   (*d*)
   ```
   s=0;
   for (int i=0; i<n*n; i++)
     s++;
   ```

(*e*)
```
for (int i=0; i<n-1; i++)
  for (j=i+1; j<n; j++) {
    priv= A[i][j];
    A[i][j]=A[j][i];
    A[j][i]=priv
  }
```

(*f*)
```
s=0;
for (int i=1; i<=n; i*=2)
  s++;
```

(*g*)
```
s=0;
for (int i=0; i<=n; i++)
  for (int j=1; j<=n; j*=2)
    s++;
```