UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČNIH NAUKA

Miloš Sirar IN 3/2020
Luka Stajić IN 21/2020

Klase složenosti algoritama i asimptotske notacije

- seminarski rad -

Profesor: dr Silvia Gilezan
Asistent: Simona Kašterović

Novi Sad, jun 2022.

Sadržaj

1 Uvod

Osnovne osobina algoritama su: diskretnost, determinisanost, efektivnost, rezultativnost, generičnost i po mogućnosti optimalnost. Pored ovih svojstava, pre svega je potrebno da je algoritam *ispravan*. No, iako on ispunjava sve ove kriterijume i osobine, vrlo važno pitanje je koje resurse algoritam zahteva. Često to znači procenu potrebne memorije, procesorske moći ili nekog drugog kapaciteta hardvera, ali je to najčešće **vreme**.

Pre analize vremena izvršavanja potrebno je napraviti model tehnologije na kom se on implementira, kao i hardver na kom se on izvršava (*RAM model*). Analiza kroz RAM model je teška, zato što se u obzir moraju uzeti mnogi aspekti kao što su veličina tipova podataka, ograničenje veličine mašinske reči, proračunavanje tačnog vremena izvršavanja svakog koraka spram vremena mašine... Takođe, potrebni matematički alati mogu uključivati kombinatoriku, teoriju verovatnoće ili algebru da bi identifikovali najznačajnije pojmove u formuli. Ovo je sve izuzetno komplikovano i sporo, jer nam fokus prebacuje na „dosadne" detalje koji nam u ovom trenutku nisu toliko bitni. Rešenje ove teške analize je u tome da uvodimo **simplifikaciju**.

2 Vreme izvršavanja i složenost algoritama

Kada gledamo vreme izvršavanja algoritma, nama nije potrebno tačno vreme već nešto na osnovu čega možemo da uporedimo kompleksnost jednog algoritma u odnosu na drugi. Uvodimo pretpostavku da postoje *jedinične (primitivne) instrukcije (koraci)* algoritma čije je izvršavanje u konstantom vremenu. *Kompleksne instrukcije* se sastoje od jediničnih, tako da se do vremena kompleksnih instrukcija dolazi sabiranjem vremena izvršavanja jediničnih instrukcija.

Vreme izvršavanja algoritma zavisi od **veličine ulaza**. Ono može biti procenjeno ili izmereno za neke konkretne veličine i neko konkretno izvršavanje. Veličina ulazne vrednosti može biti broj ulaznih elemenata koje treba obraditi, broj bitova potrebnih za zapisivanje ulaza koji treba obraditi, nekad može biti 2 broja... Uvek je potrebno eksplicitno navesti u odnosu na koju veličinu se razmatra složenost. Ali, vreme izvršavanja algoritma može biti opisano i opširnije, to jest vreme izvršavanja je potrebno parametrizovati, tako da bi se videla njegova zavisnost od veličine ulaza.

2.1 Vrste analize vremena

1. **Worst-case** analiza

   $T(n)$ = najduže vreme potrebno za izvršavanje algoritma nad ulazom veličine n. Ovo je najgori moguči slučaj, odnosno naš algoritam nikad neče raditi duže od ovog vremena. Ta procena može da bude varljiva, odnosno moguće je da će algoritam dostići samo jednom ovu vrednost a ostatak izvršavanja manju. Worst-case analiza je dobar opšti način za poređenje efikasnosti algoritma.

2. **Average-case** analiza

   $T(n)$ = očekivano prosečno vreme potrebno za izvršavanje algoritma nad ulazom prosečne veličine n. Da bi se vršila ovakva vrsta analize, potrebno je precizno poznavati prostor dopuštenih ulaznih vrednosti i verovatnoću da se svaka dopuštena ulazna vrednost pojavi na ulazu algoritma. U stvarnosti se algoritam ponaša bolje ili gore, ali je potrebno proceniti neko prosečno ponašanje u realnih uslovima.

3. **Best-case** analiza

   $T(n)$ = najbrže moguće vreme izvršavanja algoritma koje radi samo na nekim određenim ulazima. Ovo je zapravo ponašanje sistema u optimalnim okolnostima, odnosno u njegovom najboljem mogućem scenariju izvršavanja.

Od ova tri slučaja, gotovo uvek se gleda i koristi **worst-case** scenario. Average-case je dosta specijalizovan, a pogotovo best-case. Odnosno, ova dva slučaja teoretski gledano nisu skroz tačna, jer ne gledaju sve moguće ishode, već one koji im odgovaraju i zato su u većini slučajeva irelevantni pri dokazivanju efikasnosti algoritma, odnosno pokazivanju njegove brzine izvršavanja.

2.2 Klase složenosti algoritama

Složenost algoritma danas u mnogome zavisi od njegove implementacije, kao i od toga na kakvom računaru se taj algoritam izvršava. Savremeni procesori podržavaju protočnu obradu i paralelno izvršavanje instrukcija, što takođe čini stvarno ponašanje programa drugačijim od klasičnog, sekvencijalnog modela koji se najčešće podrazumeva prilikom analize algoritma. Takođe, u našem algoritmu ne vidimo da li se podaci nalaze u keš memoriji ili je potrebno pristupiti RAM-u, što isto značajno utiče na rad algoritma, a kasnije i programa. Dakle, stvarno vreme izvršavanja programa zavisi od karakteristika konkretnog računara na kom se program izvršava, ali i od karakteristika programskog prevodioca i operativnog sistema na kom se taj program izvršava.

Stvarno vreme izvršavanja zavisi i od konstanti sakrivenih u asimptotskim oznakama, međutim, asimptotsko ponašanje obično prilično dobro određuje njegov red veličine. Ako pretpostavimo da se svaka instrukcija na računaru izvršava za jednu nano sekundu ($10^{-9}s$), a da broj instrukcija zavisi od veličine ulaza n na osnovu funkcije $f(n)$, tada je vreme potrebno da se algoritam izvrši dat u sledećim tabelama.

Algoritmi čija je složenost odozgo ograničena polinomijalnim fukcijama smatraju se *efikasnim*.

$n/f(n)$	1	$logn$	n	$nlogn$	n^2	n^3
10	$0s$	0.003 µs	0.01 µs	0.033 µs	0.1 µs	1 µs
100	$0s$	0.007 µs	0.1 µs	0.644 µs	10 µs	1 ms
1000	$0s$	0.010 µs	1 µs	9.966 µs	1 ms	1 s
10000	$0s$	0.013 µs	10 µs	130 µs	0.1 s	16.7 min
100000	$0s$	0.017 µs	100 µs	1.67 ms	10 s	11.57 dan
1000000	$0s$	0.020 µs	1 ms	19.93 ms	16.7 min	31.7 god
10000000	$0s$	0.023 µs	10 ms	0.23 s	1.16 dan	3×10^5 god
100000000	$0s$	0.027 µs	0.1 s	2.66 s	115.7 dan	
1000000000	$0s$	0.030 µs	1 s	29.9 s	31.7 god	

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

$n/f(n)$	2^n	$n!$
10	1 µs	3.63 ms
20	1 ms	77.1 god
30	1 s	8.4×10^{15} god
40	18.3 min	
50	13 dana	
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.

Gornja granica složenosti se obično izražava korišćenjem $O - notacije$ (o čemu će biti više reči u narednom poglavlju. Navedimo karakteristike osnovnih klasa složenosti.

- $O(1)$ - *konstantna složenost*

  U zavisnosti od prirode algoritma, složenost algoritma zavisi od vrednosti argumenata ili od broja argumenata. Ako se svi iskazi algoritma izvršavaju samo po jednom ili najviše nekoliko puta (nezavisno od veličine i broja ulaznih podataka), kaže se da je vreme izvršavanja tog algoritma konstantno. To je najefikasnija vrsta algoritma.

- $O(logn)$ - *logaritamska složenost*

  Vreme izvršavanja programa lagano raste. Jedan od onih algoritama koji imaju ovakvu vremensku složenost je algoritam binarnog pretraživanja. Ako je osnova logaritma 10, za $n = 1000$, $log_{10}n$ je 3, a za $n = 1000000$, $\log_{10} n$ je samo dva puta veći, odnosno 6. Ovde vidimo kako za promenu ulaza sa 1000 na 1000000 imamo porast samo za 2 puta. Bez obzira da li je osnova 2, 10 ili neki drugi broj ona je nebitna, jer se svi oni ponašaju isto do na konstantu.

- $O(n)$ - *linearna složenost*

  Kada se mala količina obrade izvrši nad svakim ulaznim podatkom, vreme izvršavanja algoritma je linearno. To je optimalno vreme izvršavanja za algoritam koji mora da obradi n ulaznih podataka ili da proizvede n izlaznih podataka (na primer trazenje minimuma ili maksimuma). Kada je n milion, vreme izvršavaja je neka konstanta puta milion. Kada se n udvostruči, udvostruči se i vreme izvršavanja.

- $O(nlogn)$ - *linearno-logaritmaska složenost*

  Vreme proporcionalno sa $nlogn$ utroše na svoje izračunavanje algoritmi koji problem rešavaju tako što ga razbiju u manje potprobleme, reše te potprobleme nezavisno jedan od drugog i zatim kombinuju rešenja. Dva od onih algoritama koji imaju ovakvu vremensku složenost su Quick Sort i Merge Sort. Kada je n milion, $nlogn$ je oko dvadeset miliona. Kada se n udvostruči, vreme izvršavanja se poveća za više nego dvostruko, ali ne mnogo više!

- $O(n^2)$ - *kvadratna složenost*

  Kvadratno vreme izvršavanja nastaje za algoritme koji obrađuju sve parove ulaznih podataka, odnosno prolaze kroz ugnježdenu petlju. Algoritam sa ovakvim vremenom izvršavanja praktično je primenjivati samo za relativno male probleme, to jest tamo gde će biti malo n. Ovakvu vremensku složenost imaju na primer osnovni algoritmi za sortiranje. Kada je n hiljadu, vreme izvršavanja je milion. Kada se n udvostruči, vreme izvršavanja se uveća četiri puta.

- $O(n^3)$ - *kubna složenost*

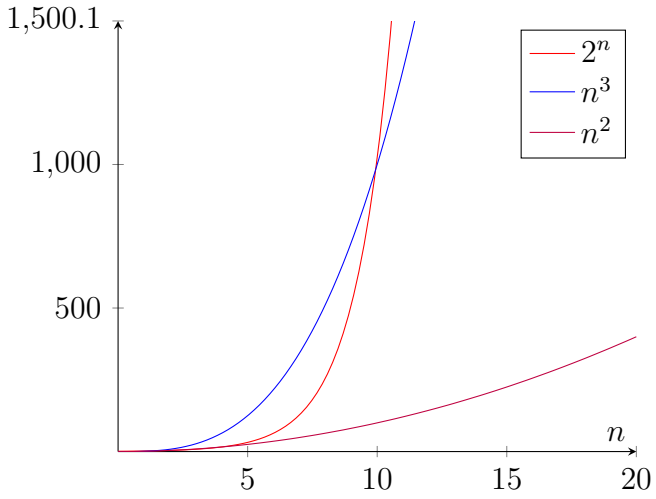
  Kubno vreme izvršavanja nastaje za algoritme koji obrađuju sve trojke ulaznih podataka, odnosno prolaze kroz trostruku ugnježdenu petlju. Algoritam sa ovakvim vremenom izvršavanja praktično je primenjivati samo za jako male probleme, odnosno za malo n. Kada je n sto, vreme izvršavanja je milion. Kada se n udvostruči, vreme izvršavanja se uveća osam puta.

- $O(2^n)$ - *eksponencijalna složenost*

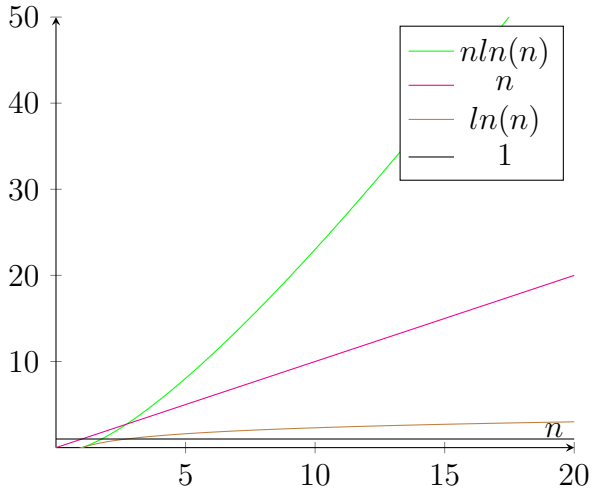
  Algoritmi sa eksponencijalnim vremenom izvršavanja nastaju prirodno pri primeni rešenja „brute force", odnosno rešenja koja obično prvo padnu na pamet prilikom rešavanja. Ipak, za neke probleme nisu nađeni algoritmi manje vremenske složenosti, i sumnja se da će ikada biti pronađeni. Ovakvi algoritmi su vrlo retko praktični za upotrebu. Kada je $n = 20$, vreme izvršavanja je milion. Kada se n udvostruči, vreme izvršavanja se kvadrira.

- $O(n!)$ - *složenost reda faktorijela*

  Ovakvi algoritmi imaju najveću moguću složenost, i praktično nisu uopšte u upotrebi.



Grafik 1. - velike vremenske složenosti



Grafik 2. - manje vremenske složenosti

2.3 Analiza insertion sortiranja

Vreme potrebno za proceduru **INSERTION-SORT** zavisi od unosa, neće isto trajati sortiranje 100000 brojeva i 3 broja. Takođe, moguće je da INSERTION-SORT traje različito vreme iako imaju isti broj elemenata, ukoliko je jedan već delimično sortiran. Vreme potrebno algoritmu da se izvrši raste sa veličinom ulaza, tako da je normalno opisati vreme rada algoritma kao funkciju veličine njegovog ulaza. Da bismo to uradili, potrebno je da preciznije definišemo pojmove *vreme rada* i *veličina ulaza*.

- *Veličina ulaza* zavisi od problema koji se proučava. Za mnoge probleme, kao sto je sortiranje ili izračunavanje Furijeovih transformacija, najprirodnija mera je broj stavki u ulazu, na primer veličina niza n za sortiranje. Sa druge strane, u nekim drugim problemima kao što su množenje dva cela broja, najbolja mera veličine ulaza je ukupan broj bitova potrebnih za predstavljanje ulaza u običan binaran zapis.

- *Vreme rada* algoritma na određenom ulazu je broj izvršenih jediničnih operacija, odnosno koraka. Potrebno je konstantno vreme da se izvrši svaki red našeg pseudokoda. Jednoj liniji može biti potrebno različito vreme od druge linije, ali pretpostavičemo da je za svako izvršenje i-te linije potrebno vreme c_i, gde je c_i konstanta.

Počinjemo predstavljanjem procedure INSERTION-SORT sa „**cost**" troškom vremena svakog iskaza i koliko se puta svaki iskaz izvršava. While petlja se u 5. redu za svako j izvršava određen broj puta, što se može iskazati t_j vremenom za svaku vrednost $j = 2, 3, ..., n$, gde je $n = A.length$. Kada se for ili while petlja završavaju na uobičajeni način oni se izvrše jedan put više nego telo petlje. Komentari nisu izvršni iskazi, pa stoga oni ne oduzimaju vreme.

	Insertion-Sort(A)	cost	times
1	**for** $j = 2$ to $A.lenght$	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// Insert $A[j]$ into the sorted sequence $A[1..j-1]$	0	$n-1$
4	$i = j - 1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] >$ key	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	$A[i+1] = key$	c_8	$n-1$

Vreme rada algoritma je zbir vremena rada za svaku izvršenu naredbu. Naredba koja preduzima c_i koraka da se izvrši i izvršava se n puta doprineće $c_i n$ ukupnom vremenu rada. Da bismo izračunali $T(n)$, moramo da saberemo proizvod *cost* i *times* kolona. Kada to uradimo dobijamo:

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$

U najboljem slučaju, kada je niz već sortiran, while se izvršava samo jednom za svako j, pa je vreme sortiranja:

$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = n(c_1 + c_2 + c_4 + c_5 + c_8) - (c_2 + c_4 + c_5 + c_8)$.

Ovo vreme rada možemo izraziti kao $an + b$ za konstante a i b koje zavise od *cost-a* c_i. To je dakle linearna funkcija od n.

U najgorem slučaju, kada je niz sortiran u obrnutom redosledu, svaki element niza $A[j]$ mora da se poredi sa svakim elementom podniza $A[1, ..., j-1]$ tako da je $t_j = j$.

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

i

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

A to dovodi do worst-case vremena izvršavanja INSERTION-SORT:

$T(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2} + c_7(\frac{n(n-1)}{2}) + c_8(n-1) = n^2(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}) + n(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8) - (c_2 + c_4 + c_5 + c_8)$.

Ova formula se može svesti na: $an^2 + bn + c$ za konstante a i b koje opet zavise od *cost-a* c_i. To je dakle kvadratna funkcija od n.

2.4 Asimptotsko vreme izvršavanja

Kao što je malopre pokazano, vreme izvršavanja kompleksnijih koraka algoritma se može dobiti sabiranjem vremena izvršavanja jediničnih koraka algoritma.

Sledeće pojednostavljenje se ogleda u uvođenju *stepena rasta (order of growth)*. To znači da se posmatra $T(n)$ za $n \to \infty$.

U inženjerskom smislu:

- Odbaciti sve elemente nižeg reda (na primer: $bn + c$ u $an^2 + bn + c$).
- Ignorisati vodeću konstantu (na primer: a u an^2).

Tako da se dobija procena vremena od n^2, a to približno vreme izvršavanja se naziva *asimptotsko vreme izvršavanja*.

3 Asimptotske notacije

Dobar uvid u složenost algoritma zasniva se na pravoj meri pojednostavljenja njihove analize.

Analiza algoritma pojednostavljuje se do krajnjih granica kako bi se pravilno odredilo asimptotsko vreme izvrsavanja algoritma. Ne interesuje nas apsolutno tačno vreme izršavanja, već samo to koliko brzo raste vreme izvršavanja algoritma sa povećanjem veličine ulaza algoritma.

- Uobičajeno, algoritam koji je asimptotski efikasniji je ujedno i bolji izbor za većinu slučajeva, osim kada je ulaz mali.
- Za dovoljno velike ulaze, brzinom izvršavanja algoritma preovlađuje njen dominantni term.
- Algoritam je brži ukoliko je njegova asimptotska funkcija rasta manja, to jest ako je stepen rasta sporiji.

Za složenije slučajeve je potrebno imati preciznu definiciju toga šta se podrazumeva kada se tvrdi da je neka funkcija manja od druge. *Asimptotska notacija* omogućava da se na precizan način uvede relativni poredak za funkcije. One se koriste da označe na koje vreme se odnose: worst-case, average-case i best-case.

Postoji nekoliko asimptotskih notacija:

- $O - notacija$ (veliko o)
- $\Theta - notacija$ (veliko teta)
- $\Omega - notacija$ (veliko omega)

- $o - notacija$ (malo o)
- $\omega - notacija$ (malo omega)

U praksi se najviše koristi $O - notacija$ jer ju je najlakše odrediti i dokazati.

3.1 O-notacija

Koristi se za precizno definisanje svojstva da je neka funkcija manja od druge. To je gornja granica složenosti. Ako za dve funkcije f i g kažemo da je f manja od g, onda mislimo da je $f(n) \leq g(n)$, za sve vrednosti n, od nekog n_0. Nas zanima rast funkcije, to jest kako se funkcija ponaša kada raste n, tako da u početku funkcija f ne mora biti manja od g. $O - notacija$ upravo predstavlja zapis kojim se predstavlja da je funkcija f manja od funkcije g, odnosno da funkcija g dominira nad funkcijom f.

Definicija 3.1. Za dve nenegativne funkcije $f, g\colon N \to R^+$ kažemo da je $f(n) = O(g(n))$ ako postoje pozitivne konstante c i n_0 takve da je $f(n) \leq c \cdot g(n)$, za svako $n \geq n_0$.

Zapis $f(n) = O(g(n))$ formalno znači da postoji neka tačka n_0 takva da za sve vrednosti n od te tačke, funkcija $f(n)$ je ograničena vrednošću proizvoda neke pozitivne konstante c i funkcije $g(n)$. $O - notacijom$ označavamo da funkcija $g(n)$ predstavlja *asimptotsku gornju granicu* za funkciju $f(n)$.

Primer 1. Neka funkcija $f(n)$ predstavlja vreme izvršavanja nekog algoritma $T(n)$. Neka je $g(n) = n^2$. Ako se dokaže da je $T(n) = O(n^2)$, onda je sigurno dokazano da je vreme izvršavanja algoritma ograničeno kvadratnom funkcijom. Tako dobijamo gornju granicu vremena izvršavanja, odnosno worst-case scenario. Uzmimo da je $T(n) = n^2 - 2n + 2$. Za $n_0 = 1$ i $c = 2$ dobijamo da je $T(n) < c \cdot g(n)$, odnosno $n^2 - 2n + 2 \leq 2n^2$. Po definiciji zaključujemo da je $T(n) = O(n^2)$.

Osobine $O - notacije$:

- Konstantni faktori nisu važni

  $T(n) = 0(c \cdot T(n))$ za svako c.

  Primer 2. $T(n) = O(n)$ je isto što i $O(134n)$ ili $O(0.1n)$.

- Najdominantniji termi su najvažniji

  Ako je $T(n) = O(f(n) + g(n))$ i $g(n) = O(f(n))$, onda je $T(n) = O(f(n))$

  Primer 3. Ukoliko imamo $T(n) = n^3 + n^2 + 1$, onda je $T(N) = O(n^3)$, jer je $n^2 = O(n^3)$ i $1 = O(n^3)$.

- Pravilo sabiranja

  $O(f(n) + g(n)) = O(f(n)) + O(g(n))$.

  Primer 4. Ako je $T(n) = O(n^2) + O(n)$, onda je $T(n) = O(n^2 + n)$, odnosno $T(n) = O(n^2)$, na osnovu pravila o najdominantnijem termu.

- Pravilo množenja

  $O(f(n) \cdot g(n)) = O(f(n)) \cdot O(g(n))$

  PRIMER 5. Ako je $T(n) = n^2 n$, onda je $T(n) = O(n^2) \cdot O(n) = O(n^2)$

- Pravilo tranzitivnosti

  Ako je $T(n) = O(f(n))$ i $f(n) = O(g(n))$ onda je i $T(n) = O(g(n))$

  PRIMER 6. Ako je $T(n) = O(n)$, kako je očevidno $n = O(n^2)$, sledi da je i $T(n) = O(n^2)$.

Ukoliko koristimo ove osobine, možemo mnogo brže i lakše doći do dominantne funkcije vremena izvršavanja.

Ali, ipak postoji jedan mali problem kod $O - notacije$, a to je da može da se zloupotrebi i gleda na drugačiji način. To ćemo pokaziti pomoću sledećeg primera.

PRIMER 7. Neka je vreme izvršavanja nekog algoritma $T(n) = n^2 + 2n - 1$, odonosno $T(n) = O(n^2)$. Pravilom tranzitivnosti možemo reći $T(n) = O(n^{100})$. Da li je to netačno? Nije netačno, ali naš cilj je da pronađemo najbolju dominirajuću funkciju za vreme izvršavanja. Nema smisla da uzimamo $O(n^{100})$ kao ocenu za $T(n) = n^2 + 2n - 1$. Po pravilu, uvek uzimamo najprostiju $O - notaciju$. To znači da se iskazuje sa jednim termom i konstantom $c = 1$.

NAPOMENA 1. Ukoliko je $f_1(n) = O(g(n))$ i $f_2(n) = O(g(n))$, to ne implicira da je $f_1(n) = f_2(n)$!

3.2 Ω-notacija

Veoma je slična $O - notaciji$, ali sa suprotne strane; koristi se za precizno definisanje svojstva da je neka funkcija veća od neke druge. Ako za dve funkcije f i g kažemo da je f veća od g, onda se misli na to da je $f(n) \leq g(n)$, za sve ili neke vrednosti argumenta n .Ni ovde zahtev nije toliko strog, to jest ne mora biti za sve vrednosti n. $\Omega - notacija$, $f(n) = \Omega(g(n))$ predstavlja zapis kojim se predstavlja svojstvo da je funkcija f, veća od funkcije g.

Koristi se da označi da se neki algoritam mora izvršiti bar za neko određeno vreme, to jest best-case scenario. Vreme izvršavanja algoritma nikada neće imati bolji rast od Ω-zapisane funkcije.

Definicija 3.2. Za dve nenegativne funkcije $f, g\colon N \to R^+$ kažemo da je $f(n) = \Omega(g(n))$ ako postoje pozitivne konstante c i n_0 takve da je $f(n) \geq c \cdot g(n)$, za svako $n \geq n_0$.

Zapis $f(n) = \Omega(g(n))$ formalno znači da postoji neka tačka n_0 takva da za sve vrednosti n od te tačke, funkcija $f(n)$ ograničava vrednost proizvoda neke pozitivne konstante c i funkcije $g(n)$. $\Omega - notacijom$ označavamo da funkcija $g(n)$ predstavlja *asimptotsku donju granicu* za funkciju $f(n)$.

Osobine $\Omega - notacije$ su iste kao i kod $O - notaciju$.

NAPOMENA 2. $f_1(n) = \Omega(g(n))$ i $f_2(n) = \Omega(g(n))$ ne implicira $f_1(n) = f_2(n)$!

3.3 Θ-notacija

$\Theta - notacija$ predstavlja *asimptotski najbolju ocenu* za vreme izvršavanja, to jest za funkciju $f(n)$. Kada se ona koristi u analizi algoritama, ne daje se samo gornja granica izvršavanja, već i njegova donja granica, pa to znači da je vreme izvršavanja najbolje moguće ocenjeno.

Standardno se mnogo manje koristi od $O - notacije$, jer je daleko teže odrediti i donju i gornju granicu.

Definicija 3.3. Za dve nenegativne funkcije $f, g\colon N \to R^+$ kažemo da je $f(n) = \Theta(g(n))$ ako postoje pozitivne konstante c_1, c_2 i n_0 takve da za svako $n \geq n_0$ važi $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$.

Drugim rečima, $\Theta - notacija$ označava da funkcija $g(n)$ predstavlja *strogu asimptotsku granicu* (i gornju i donju) za funkciju $f(n)$.

Na šta se svodi određivanje $\Theta - notacije$?

Primer 8. Da bi dokazali da je $\frac{1}{2}n^2 - 3n = \Theta(n^2)$, treba pronaći konstante c_1, c_2 i n_0 takve da je za svako $n \geq n_0$:

$$c_1 \cdot n^2 \leq \tfrac{1}{2}n^2 - 3n \leq c_2 \cdot n^2$$

Deljenjem sa n^2 dobijamo: $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$

Desna strana nejednakosti postaje tačna za $n \geq 1$ i $c_2 = \frac{1}{2}$

Leva strana nejednakosti postaje tačna za $n \geq 7$ i $c_1 = \frac{1}{4}$

Za $n_0 = 7$, $c_1 = \frac{1}{4}$ i $c_2 = \frac{1}{2}$, pokazali smo da je:

$$\tfrac{1}{2}n^2 - 3n = \Theta(n^2)$$

Za $n_0 = 7$, $c_1 = \frac{1}{4}$ i $c_2 = \frac{1}{2}$, pokazali smo da je:

$$\tfrac{1}{2}n^2 - 3n = \Theta(n^2)$$

Naravno, postoje i druge vrednosti za koje jednakost važi, ali važno je da neke vrednosti postoje (makar jedna kombinacija). Ove vrednosti umnogome zavise i od vrste konstanti i terma koji čine $T(n)$. Druga funkcija koju određuje $\Theta(n)$ bi zahtevala drugačije konstante.

Napomena 3. $f_1(n) = \Theta(g(n))$ i $f_2(n) = \Theta(g(n))$ ne implicira $f_1(n) = f_2(n)$!

Teorema 3.4. Za dve nenegativne funkcije $f, g\colon N \to R^+$ kažemo da je $f(n) = \Theta(g(n))$ ako i samo ako je $f(n) = O(g(n))$ i $f(n) = \Theta(g(n))$.

3.4 o-notacija

$O - notacija$ ne mora biti čvrsta asimptotska gornja granica. Veza $4n^2 = O(n^2)$ je asimptotski čvrsta gornja granica (neposredno iznad), dok veza $4n = O(n^2)$ nije asimptotski čvrsta gornja granica.

Za asimptotske gornje granice funkcija koje po prirodi nisu čvrste koristi se $o - notacija$.

Definicija 3.5. Za dve nenegativne funkcije $f, g\colon N \to R^+$ kažemo da je $f(n) = o(g(n))$ ako za svaku pozitivnu konstantu c postoji konstanta n_0 takva da je $f(n) \leq c \cdot g(n)$, za svako $n \geq n_0$.

Zapis $f(n) = o(g(n))$ označava da je brzina rasta funkcije $f(n)$ reda veličine manja od brzine rasta funkcije $g(n)$.

Primer 9. $4n^2 \neq o(n^2)$, ali je zato $2n = o(n^2)$.

$O - notacija$ i $o - notacija$ su slične, ali dok za $f(n) = O(g(n))$ veza $f(n) \leq c \cdot g(n)$ važi samo za neke konstante $c > 0$, za $f(n) = o(g(n))$ to važi za svako $c > 0$.

3.5 ω-notacija

$\omega - notacija$ se odnosi prema $\Omega - notaciji$, isto što i $o - notacija$ za $O - notaciju$.

Za asimptotske donje granice funkcija koje po prirodi nisu čvrste koristi se $\omega - notacija$.

Definicija 3.6. Za dve nenegativne funkcije $f, g\colon N \to R^+$ kažemo da je $f(n) = \omega(g(n))$ za svaku pozitivnu konstantu c postoji n_0 takva da je $f(n) \geq c \cdot g(n)$, za svako $n \geq n_0$.

Zapis $f(n) = \omega(g(n))$ označava da je brzina rasta funkcije $f(n)$ reda veličine veća od brzine rasta funkcije $g(n)$.

Primer 10. $\frac{n^2}{2} = \omega(n)$, ali je zato $\frac{n^2}{2} \neq \omega(n^2)$.

$\Omega - notacija$ i $\omega - notacija$ su slične, ali dok za $f(n) = \Omega(g(n))$ veza $f(n) \leq c \cdot g(n)$ važi samo za neke konstante $c > 0$, za $f(n) = \omega(g(n))$ to važi za svako $c > 0$.

3.6 Poređenje asimptotskih funkcija

Ukoliko pretpostavimo da su funkcije asimptotski pozitivne, onda za njih važe sledeće osobine :

- Tranzitivnost

  $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \to f(n) = \Theta(h(n))$

  $f(n) = O(g(n)) \wedge g(n) = O(h(n)) \to f(n) = O(h(n))$

  $f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \to f(n) = \Omega(h(n))$

  $f(n) = o(g(n)) \wedge g(n) = o(h(n)) \to f(n) = o(h(n))$

  $f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \to f(n) = \omega(h(n))$

- Refleksivnost

  $f(n) = \Theta(f(n))$

  $f(n) = O(f(n))$

  $f(n) = \Omega(f(n))$

- Simetričnost

  $f(n) = \Theta(g(n))$ ako i samo ako $g(n) = \Theta(f(n))$

- Transponovana simetričnost

  $f(n) = O(g(n))$ ako i samo ako je $g(n) = \Omega(f(n))$

  $f(n) = o(g(n))$ ako i samo ako je $g(n) = \omega(f(n))$

Pošto ove osobine važe za asimptotske notacije, možemo povući analogiju između asimptotičkog poređenja dve funkcije f i g, i poređenja dva realna broja a i b:

- $f(n) = O(g(n))$ je kao $a \leq b$
- $f(n) = \Omega(g(n))$ je kao $a \geq b$
- $f(n) = \Theta(g(n))$ je kao $a = b$
- $f(n) = o(g(n))$ je kao $a < b$
- $f(n) = \omega(g(n))$ je kao $a > b$

Kažemo da je $f(n)$ asimptotski manja od $g(n)$ ako je $f(n) = o(g(n))$, dok se za $f(n)$ kaže da je asimptotski veća od $g(n)$ ako je $f(n) = \omega(g(n))$.

Napomena 4. Jedino pravilo koje se ne prenosi sa realnih brojeva na asimptotsku notaciju. Ne važi pravilo da za svaka dva realna broja mora biti tačno samo jedno od ova tri tvrđenja:

- $a < b$
- $a = b$
- $a > b$

zato što se *ne mogu sve funkcije asimptotski uporediti*!

4 Zaključak

U današnje vreme imamo eksponencijalni rast podataka. Svakim danom sve više i više podataka dospeva na računare, a samim tim i u naše programe koji implementiraju algoritme. Rast broja podataka je već uveliko prestigao razvijanje računarskih tehnologija, što je otvorilo novu dimenziju problema. Zbog toga je od izuzetne važnosti da uvidimo koliko je zapravo bitan problem složenosti algoritama. Nije svejedno da li će naš algoritam imati vremensku složenost $O(n)$ ili $O(n^2)$. Treba da težimo da algoritmi budu što je moguće jednostavniji, a samim tim će njihovo izvršavanje biti brže. Takođe je od velikog značaja da poznajemo asimptotske notacije, jer će nam one pomoći da pravilno odredimo rast složenosti algoritama, ukoliko ih pravilno koristimo. Pored razvijanja algoritama, a kasnije i programa, treba naći prostora i posvetiti se ovim problemima, zato što su od ključne važnosti da vaš algoritam bude bolji i efikasniji od ostalih.

Literatura

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms, Third Edition*, The MIT Press, 2009.

[2] Dinu Dragan *Slajdovi i beleške sa predavanja iz predmeta Teorija algoritama*, Fakultet Tehničkih nauka, Novi Sad, 2021.

[3] Milena Vujošević Janičić, Jelena Graovac *Programiranje 2, složenost algoritama, prezentacija*, Matematički fakultet, Beograd, 2020.

[4] Filip Marić *Efikasnost i složenost algoritama, pdf*, Matematički fakultet, Beograd