Složenost algoritma

Na početku složenost algoritma definisaćemo neformalno kao **maksimalni broj operacija potrebnih za izvršavanje algoritma**. Ovde smo prvo uveli pretpostavku da su sve (osnovne) operacije iste složenosti, s obzirom da nam je potreban samo njihov broj. Sa druge strane, broj operacija će svakako zavisiti od samog ulaza. Iz tog razloga, kada se ispituje složenost nekog algoritma treba razmatrati "najgori mogući slučaj". U daljem delu ćemo videti da nas zanima samo asimptotsko ponašanje vremenske složenosti.

Osnovne operacije predstavljaju skup operacija čije se vreme izvršavanja može ograničiti nekom konstantom koja zavisi samo od konkretne realizacije (računara, programskog jezika, prevodioca…). Drugim rečima, pretpostavljamo da se svaka osnovna operacija izvršava za jedinično vreme. Naravno nisu sve operacije takve: primera radi stepenovanje ne možemo smatrati osnovnom operacijom[1]. Tipične osnovne operacije su:

- dodela vrednosti promenjivoj
- poređenje dve promenjive
- aritmetičke i logičke operacije
- ulazno / izlazne operacije

Kako je i sama gornja definicija složenosti još uvek komplikovana, moramo ignorisati još neke faktore. Iz tog razloga uvodimo novo pravilo **zanemarivanja konstanti**. Složenost će zavisiti od ulaznih veličina (ograničenja brojnih vrednosti, veličina matrica i nizova...), dok ćemo konstante koje su "mnogo" manje od ulaznih ograničenja zanemariti. Naredni primeri će vam približiti ovu priču.

Posmatrajmo *Algoritam 1* na kojem je prikazan deo funkcije za računanje srednje vrednosti niza. Broj operacija koji će da se izvrši u toku ovog algoritma je $2n + 2$ (n puta ćemo povećati vrednost promenjive i; n puta ćemo povećati sumu za vrednost $a[i]$; jedna operacija za inicijalizaciju $suma$ na 0; jedna za računanje srednje vrednosti). Kako konstante zanemarujemo dobijamo da je složenost ovog algoritma: n operacija. Ovu činjenicu ćemo zapisivati kao $O(n)$ – čitamo "o od n". U ovom slučaju kažemo da je algoritam **linearne složenosti**.

```
=============================================================
01     suma = 0;
02     for i = 1 to n
03        suma = suma + a [i];
04     avg = suma / n;+
=============================================================
```

Algoritam 1. Nalaženje srednje vrenosti niza

Konstante zanemarujemo iz razloga što su one obično dosta manje od uzlanih veličina. Primera radi za gore opisani algoritam, da li ćemo izvršiti $n + 1$ operaciju ili n neće uticati za veće veličine niza

[1] Algoritam brzog stepenovanja će biti izložen u problemu *Vrednost polinoma*.

(naravno za manje veličine složenost je svakako mala). Slično se i množilac može zanemariti ukoliko je konstanta. Složenost algortima tražimo kao funkciju od parametara veličine ulaza.

```
=============================================================
01     for i = 1 to n - 1
02        for j = i + 1 to n
03           if a [i] < a [j] then
04              pom = a [i];
05              a [i] = a [j];
06              a [j] = tmp;
=============================================================
```

Algoritam 2.

Složenost *Algoritma 2*[2] je kvadratna tj. $O(n^2)$. Tačan broj operacija je $3\frac{n(n-1)}{2}$, ukoliko bi morali svaka dva elementa da zamenimo (3 operacije su potrebne za zamenu; i i j biramo na $\frac{n(n-1)}{2}$ kojima upoređujemo svaka dva elementa niza). Zanemarujući konstante dobijamo da je broj operacija zapravo $n^2 - n$. Kako je n mnogo manje od n^2 za veće vrednosti, možemo zanemariti linearni faktor (između $1.001.000$ ili $1.000.000$ operacije nema značajnih razlika u vremenu).

Nije uvek jednostavno utvrditi složenost algoritma. Nažalost za ovaj problem ne postoji univerzalni metod koji možemo primeniti. Međutim, vežbom svakako možemo steći dobru intuiciju o njoj. Poznavanje složenosti nekih standardnijih algoritama i programskih konstrukcija (pogledati *Tabelu 1*) nam u tome mogu dosta pomoći.

Rb	Programska konstrukcija	Vremenska složenost konstrukcije u zavisnosti od njenih delova

01	Sekvenca naredbi S:
	`P;`
	`Q;`
		$O(S) = O(P) + O(Q)$

02	Uslovna naredba S:
	`if (uslov) then`
	`  P;`
	`else`
	`  Q;`
		$O(S) = \max\{O(P), \mathrm{O(Q)}\}$

03	For petlja S:
	`for i = 1 to n do`
	`  P;`
		$O(S) = nO(P)$

04	While / Repeat petlja S:
	`while (uslov) do`
	`  P;`
		$O(S) = mO(P)$
		gde je m broj interacija petlje u najgorem mogućem slučaju

Tabela 1: Složenosti osnovnih programskih konstrukcija.

Pored računanja vremenske složenosti kao broj operacija u najgorem slučaju, postoje i drugi pristupi za merenje vremenske efikasnosti algoritma. Jedan od njih je probabilistički metod srednjeg vremena. **Prosečno vreme izvršavanja** tj. vremensku složenost na ovaj način definišemo kao očekivani broj potrebnih operacija da bi se algoritam izvršio. Ovo je dosta realističniji pristup u praksi, ali je svakako i mnogo teže proceniti ga. Razlog za to je što bi njegova procena zahtevala poznavanje raspodele

[2] Algoritam 2 predstavlja algoritam za sortiranje nizova.

verovatnoće ulaznih podataka. Sa druge strane, korpusi test primera kod takmičarskih problema obuhvataju i specijalne slučajeve i najgore ulaze. Na ovaj način se testira ponašanje algoritma u svim mogućim scenarijima. Naravno, za randomizirane algoritme, koji u toku izvršavanja donose slučajne odluke, procena vremenske složenosti se mora zasnivati na **očekivanom vremenu izvršavanja** koji smo spomenuli. O ovome će biti još priče u delu o **Quicksort-u**.