Algoritmi i složenost

Algoritam je detaljan opis nekog procesa. U računarstvu se to odnosi na niz koraka (instrukcija) koji za određene ulazne podatke daje nekakav željeni izlaz. Algoritam bi intuitivno trebao imati iduća svojstva:

1) za bilo koji ulaz, izlaz će uvijek biti predvidljiv i jednak pri svakom izvođenju algoritma
2) algoritam ima definiran izlaz za bilo koji dozvoljeni ulaz
3) razumno vrijeme izvođenja – usko vezano uz složenost i biti će kasnije opisano

U stvarnosti postoje cijele klase algoritama koji nemaju sva ta svojstva. Formalnije, algoritam se može definirati kroz iduća svojstva:

1) Konačnost – algoritam će završiti u ograničenom broju koraka
2) Definitivnost – svaki korak je jasno određen i nedvosmislen
3) Ulaz – algoritam ima nekakve podatke kao ulaz
4) Izlaz – algoritam će za svaki ulaz generirati izlaz
5) Efikasnost – svaki korak će biti izveden u ograničenom vremenu

Ova svojstva su uglavnom potrebna za teoretsku obradu algoritama, ali navedena svojstva vrijede za osnovne algoritme koje ćemo vidjeti na ovom predmetu i sve rečeno će se odnositi na njih, osim ako to nije posebno naznačeno.

Složenost je apstraktni način za opis potrebnih resursa za izvođenje nekakvog procesa. Preciznije, složenost će opisivati rast potrebnih resursa sa rastom veličine ulaza u proces. U programiranju nas zanima računska složenost gdje je proces nekakav algoritam ili gotov program, a cijena je obično trajanje izvođenja programa ili potrebna memorija za izvođenje programa. Kako stvarno trajanje izvođenja nekog programa ovisi o puno faktora, poput brzine procesora i implementacijskih detalja, složenost nam omogućuje da odbacimo sve detalje koji su neovisni o kvaliteti samog algoritma kojeg implementiramo. U ovom poglavlju ćemo se referirati uglavnom na vremensku složenost koja je u direktnoj vezi sa brzinom izvođenja programa jer je to često najzanimljivija kvaliteta nekog programa ili algoritma. Povremeno će se spomenuti i memorijska složenost koja se odnosi na potreban memorijski kapacitet za izvođenje programa i to će biti posebno naznačeno u tekstu. Uz te dvije, za programe gdje je to relevantno, složenost može opisivati i druge resurse poput broja pisanja po disku, broja zahtjeva prema serveru ili broja upita prema bazi podataka.

Ideja složenosti u računarstvu

Složenost (vremenska) je mjera koja u najgrubljim crtama opisuje broj linija kôda koje računalo mora izvesti prilikom izvođenja nekog programa. Naravno da će različiti programski jezici imati različit broj linija kôda za isti program, ali složenost će nam omogućiti generalniji opis neovisan i o samom jeziku. Za sada možemo razmatrati samo C programski jezik. Na primjer, za iduću funkciju možemo pokušati odrediti koliko će linija kôda biti izvedeno:

```
void ispis(int *niz, int n) {
      int i = 0;
      while (i < n) {
            printf("%d", niz[i]);
            i++;
      }
}
```

Ako pretpostavimo da je *n=10* i prebrojim linije, dobiti ćemo otprilike 12 linija ovisno o tome da li u „linije“ spadaju i vitičaste zagrade. Očigledno, broj izvedenih linija će ovisiti i o količini podataka koji obrađujemo. U primjeru to može biti precizno opisano duljinom niza *n* jer je sve drugo relativno konstantno i ne ovisi o ulaznim podacima. Naravno da u stvarnosti to nije potpuno istinito jer npr. funkcija *printf* može interno imati različit broj linija za ispis jednoznamenkastog broja u odnosu na ispis brojeva sa više znamenaka. Za sada ćemo svejedno raditi takve „praktične“ pretpostavke, ali kasnije ćemo ih matematički opravdati ili redefinirati. Ono što je u svakom slučaju točno je da će broj izvedenih linija ovisiti o duljini niza i zbog toga ćemo složenost uvijek izražavati kao funkciju količine podataka ili kao funkciju nekog drugog parametra kojeg smatramo ključnim za izračun složenosti. U ovom primjeru bi broj izvedenih linija (odnosno vremensku složenost) zato mogli prikazati kao funkciju *T(n)*:

$$T(n) = 1 + 3*n$$

1 linija će se uvijek izvesti (deklaracija varijable), a 3 linije (usporedba, *printf* i uvećavanje brojača) će se izvesti ovisno o broju elemenata niza *n*. Ta funkcija opisuje vremensku složenost prethodne funkcije odnosno algoritma za ispis niza dužine *n*.

Da bi podjednako jednostavno mogli odrediti složenost drugih funkcija ili cijelih programa, za neformalni izračun složenosti radimo iduće pretpostavke:

1) Konstantno vrijeme izvođenja osnovnih operacija (usporedba, aritmetičke operacije, …)
2) Konstantno vrijeme ispisa i poziva funkcije
3) Konstantno vrijeme pristupa bilo kojoj memorijskoj lokaciji (*niz[i]*)

Takve pretpostavke nam omogućuju da broj izvedenih linija prevedemo u vrijeme izvođenja programa koje će biti funkcija jednaka *T(n)* izrazu, pomnoženim sa nekom konstantom koja će ovisiti o detaljima računala i implementacije. Za većinu sličnih algoritama, ovisnost vremena izvođenja o veličini niza ne možemo eliminirati nikakvom konstantom i zbog toga će vrijeme izvođenja i dalje biti funkcija veličine ulaza.

Procjena kvalitete algoritma izračunom *T(n)* funkcije je u stvarnosti previše detaljan i često će nam za usporedbu algoritama biti dovoljan jednostavniji izraz u obliku notacije „velikog O“. Taj jednostavniji izraz će biti utemeljen na *T(n)* ili na kombiniranju drugih „O“ izraza. „O“ notacija će uvijek biti zapisana u obliku *O(f(n))*, gdje je f(n) funkcija koja opisuje složenost algoritma (najčešće pojednostavljena *T(n)* funkcija, npr. n^2). Neformalno opet možemo gledati prelazak iz *T(n)* na *O(f(n))* kao da iz izraza koji opisuje vrijeme izvođenja algoritma uzmemo član koji najbrže raste. Kasnije ćemo to opažanje opravdati matematički, ali za sada je dovoljno primijetiti da će za n koji raste u beskonačno, najbrže rastući član učiniti sve ostale članove zbroja zanemarivim. Takva analiza, gdje *n* teži u beskonačno, se naziva asimptotska analiza složenosti algoritma. U konačnici, *O(f(n))* će, isto kao i *T(n)*, opisivati rast potrebnih resursa (vremena) sa rastom veličine ulaza *n*. Za prethodni primjer, gdje vrijeme izvođenja linearno raste sa veličinom ulaza, složenost bi izrazili kao *O(n)*. Još jedan primjer O notacije je konvencija da se algoritmi koji imaju konstantno vrijeme izvođenja (neovisno o veličini ulaza) označavaju sa *O(1)*. Kasnije ćemo spomenuti i druge tipične klase složenosti i njihove oznake.

Vremenska i memorijska složenost

Uz vremensku složenost koju smo povezali sa brojem izvedenih linija ponekad nas zanima i memorijska složenost koja odražava (najveću) količinu memorije potrebnu za izvođenje algoritma. U tu memoriju ne ubrajamo memoriju potrebnu za predstavljanje ulaza jer nas zanima samo dodatna potrebna memorija. Kod većine algoritama, pa tako i za funkciju iz prethodnog primjera, memorijska složenost *M(n)* je konstantna. To znači da će nam za izvođenje prethodne funkcije trebati određena količina memorije (za varijablu *i*) koja neće ovisiti o veličini ulaza. To se može lako provjeriti u primjeru. *M(n)* je memorijske ekvivalent *T(n)* funkciji i često će se i ona izraziti kroz O notaciju. Tako konstantnu memorijsku složenost u ovom slučaju možemo označiti sa *O(1)*.

Primjeri izračuna složenosti nekih programa

Ovdje je dano nekoliko osnovnih primjera za izračun vremenske složenosti analizom kôda. Za svaki primjer ćemo prvo neformalno odrediti *T(n)* funkciju pod istim „praktičnim“ pretpostavkama kao u prethodnom primjeru. Nakon toga ćemo napisati pojednostavljenu funkciju složenost u O notaciji.

```
void ispis(int *niz, int n) {
	int i = 0;
	while (i < n) {
		int j = 0;
		while (j < n) {
			printf("%d %d", niz[i], niz[j]);
			j++;
		}
		i++;
	}
}
```

U ovom primjeru imao dvije ugniježdene petlje. Možemo krenuti od vanjske petlje i, slično kao u prethodnom primjeru, opisati složenost kao $T(n) = c1 + n * P(n)$, gdje je $P(n)$ predstavlja složenost unutrašnje petlje. Kako će broj iteracija unutrašnje petlje rasti direktno sa rastom n parametra, $P(n)$ možemo izraziti kao $P(n) = c3 + c4*n$, slično kao i za vanjsku petlju samo sa novim konstantama i n umjesto $P(n)$. Ako uvrstimo $P(n)$ u $T(n)$, dobijemo $T(n) = c1 + n * (c2 + c3*n)$. Kada pomnožimo n sa izrazom u zagradama, dobijemo konačnu kvadratnu funkciju $T(n) = c1 + c2 * n + c3 * n^2$. Iz takve funkcije možemo odbaciti sve članove osim najbrže rastućeg i sve konstante i izraziti je u O notaciji kao $O(n^2)$. Očigledno, do istog rezultata bi došli i krenuvši od unutrašnje petlje. Kasnije ćemo vidjeti pravila za O notaciju pomoću kojih ćemo moći kombinirati petlje (i dijelove algoritma) na formalniji način.

```
void ispis(int *niz, int n) {
	int i = 0;
	while (i < n) {
		int j = 0;
		while (j < 10) {
			printf("%d %d", niz[i], niz[j]);
			j++;
		}
		i++;
	}
}
```

U ovom primjeru se unutarnja petlja vrti 10 puta, neovisno o veličini niza. Zbog toga će unutrašnja petlja imati konstantno vrijeme izvođenja, a izvođenje vanjske petlja će i dalje rasti linearno sa veličinom ulaza. Kao i u prethodnom primjeru, izraz za vanjsku petlju je $T(n) = c1 + n * P(n)$, ali

sada je *P(n) = c2* odnosno vrijeme je konstanto i ne ovisi o dužini ulaza. Na kraju dobijemo linearnu funkciju *T(n) = c1 + n * c2* i *O(n)* kao i u prvom primjeru što je i logično jer je unutrašnja petlja efektivno samo jedna velika naredba.

```
void ispis(int *niz, int n) {
	int i = 1;
	while (i < n) {
		printf("%d", niz[i]);
		i = i * 2;
	}
}
```

Ovaj primjer je malo nezgodniji jer varijabla *i* više ne raste linearno nego geometrijski, krenuvši od 1. Kako uvjet ograničava taj rast sa n, potrebno je odrediti u koliko iteracija ćemo doći do n što će nam odrediti složenost. Na primjer, ako je *n=8*, u samo tri iteracije će varijabla i postati jednaka *n* i petlja će završiti. Ako je *n=25*, petlja će završiti nakon 5 iteracija, kada *i* dosegne 32. Generalno, ako izrazimo broj *n* kao 2^x, *x* će označavati koliko iteracija će petlja proći prije nego što završi jer će tada varijabla i preći parametar *n*. Koristeći funkciju logaritma, možemo izraziti *x* preko *n*: $x = log_2(n)$. Sada složenost možemo izraziti kao funkciju $T(n) = c1 + c2 * log_2(n)$. U O notaciji ćemo zadržati samo logaritam jer raste najbrže i ignorirati bazu logaritma jer su svi logaritmi linearno ovisni: $O(log(n))$.

Druge primjere ćete vidjeti na predavanjima, ali osnova cijele analize je uvijek izvođenje algoritma „u glavi“, odnosno jasno razumijevanje sintakse i semantike programa.

Matematička analiza složenosti algoritma

U stvarnosti, pretpostavke o konstantnom izvođenju osnovnih naredbi nije daleko od stvarne analize. Jedino što moramo naći je konstantu koja će nadvisiti bilo koje stvarno vrijeme izvođenja. Zbog toga će svi izračuni biti pretvoreni u nejednakosti koje će ovisiti o pretpostavljenim konstantama. Te konstante nećemo nikada izraziti brojčano jer nam je za analizu dovoljno da postoje. Tako bi analiza funkcije vremena izvođenja *T(n)* za prethodni primjer bila:

$$T(n) \leq c_1 + c_2 * n$$

Jasno je da, ako smo ispravno pretpostavili dijelove algoritma koji ne ovise o veličini ulaza *n*, potrebne konstante možemo uvijek pronaći (u ovom slučaju c_1). Druge dijelovi programa (koji ovise o *n*) nećemo moći nadvisiti nikakvom konstantom nego moramo pretpostaviti nekakvu funkciju od *n* koju ćemo opet moći pomnožiti sa nekom konstantom (u ovom slučaju c_2). Ako vrijeme izvođenja programa raste s kvadratom veličine ulaza (ili više) onda će nam trebati ispravne funkcije od n koje ćemo onda moći pomnožiti sa odgovarajućom konstantom. *T(n)* je vrlo precizna mjera za brzinu izvođenja algoritma, ali često je previše detaljna i teška za izračun jer ovisi o detaljima implementacije algoritma. Zbog toga se za algoritme često preferira jednostavnija O notacija odnosno asimptotska analiza složenosti algoritma.

Asimptotska analiza algoritma

Asimptotska analiza algoritma traži funkciju ovisnu o *n* koja će ograničiti rast vremena izvođenja *T(n)* za *n* koji teži u beskonačnost. Da bi dodatno pojednostavnili mjeru za kvalitetu algoritama, trebamo dodati minimalnu veličinu ulaza za koju će analiza vrijediti – *n0*. Tada će cijela analiza vrijediti za sve $n > n0$, a to će nam omogućiti da pojednostavnimo opis algoritma i klasificiramo ih u tipične klase složenosti. Za prethodni primjer gdje smo krenuli sa:

$$T(n) \leq c1 + c2n$$

sada možemo naći *c3*, takav da je:

$$T(n) \leq c3n$$

pod uvjetom da je n veći od nekog *n0*. To možemo lako dokazati:

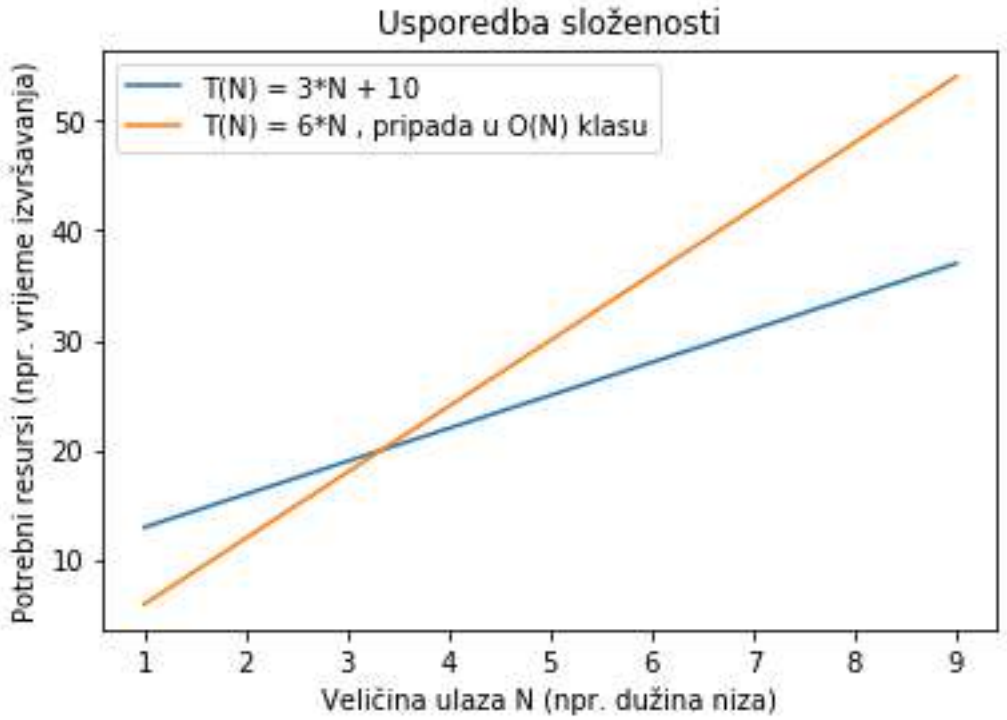
$c1 + c2n \leq c3n$ (podijelimo sve sa n, koji je pozitivan)

$c1/n + c2 \leq c3$ (za c3 možemo uzeti c1/n0 + c2)

$c1/n + c2 \leq c1/n0 + c2 = c3$, što očigledno vrijedi za $n > n0$ jer je *n* u nazivniku

Sličan dokaz se može napraviti sa bilo kojom funkcijom, a vidimo da efektivno zadržimo samo član koji najbrže raste. Zbog svega toga možemo reći da algoritam pripada u *O(n)* klasu složenosti koja sadrži sve algoritme koje možemo nadvisiti/ograničiti od gore sa nekom funkcijom $f(n) = c*n$.

Cijelu analizu možemo promatrati i grafički:



Na slici se vidi da će jednostavniji izraz *c*n* nadvisiti složeniji izraz *c1*n + c2* za dovoljno veliki *n* (veći od 3 u ovom slučaju) i dovoljno veliku konstantu *c*.

Treba primijetiti da znak jednakosti u svim ovim analizama nije baš uobičajen i ne znači potpunu jednakost lijeve i desne strane. Ispravno bi bilo koristiti znak „pripada skupu“ (ϵ) kada *T(n)* spada u klasu *O(f(n))*:

$$T(n) \in O(f(n))$$

ili bi se moglo pisati $T(n) \leq O(f(n))$, ali često se samo napiše $T(n) = O(f(n))$.

Pravila za O() notaciju

Asimptotička analiza nam omogućava da analiziramo algoritme i dijelove algoritma na prilično visokoj razini. U stvarnosti nam je dovoljno da imamo „garancije“ za trajanje pojedinih dijelova algoritma da bi mogli izraziti ukupnu složenost algoritma. Te „garancije“ su u biti složenosti izražene u O notaciji i ukupnu složenost ćemo dobiti u obliku O izraza. Ovo je vrlo korisno kada, na primjer, koristimo biblioteku čije detalje implementacije ne znamo, ali imamo složenosti pojedinih funkcija u biblioteci. Drugi primjer bi bio da koristimo složenu strukturu podataka (vidjeti ćemo ih kasnije) i želimo opisati složenost pojedinih operacija nad njom. U oba slučaja će se koristiti O notacija.

Najbolje da krenemo sa jednostavnim primjerom koji će prikazati pravilo za zbrajanje O izraza:

```
void ispis(int *niz, int n) {
	int i = 0;
	while (i < n) {
		printf("%d", niz[i]);
		i++;
	}
	while (i > 0) {
		printf("%d", niz[i - 1]);
		i--;
	}
}
```

Iz koda se može vidjeti da efektivno dva puta ispisujemo cijeli niz (po redu i unazad). Kada bi htjeli izraziti ukupnu vremensku složenost, u analogiji s prvim primjerom, mogli bi je opisati izrazom *T(n) = 1 + 3 * n + 3 * n*. Mogli bi analizirati i pojedine petlje i utvrditi da je ukupna vremenska složenost *T(n) = O(1) + O(n) + O(n)*, prema dosada prikazanim pravilima za O notaciju. Ovdje je možda malo čudno da *T(n)* izražavamo pomoću O notacije, ali to je često jedina opcija ako nemam uvid u detalje algoritma i jedino imamo njihovu složenost u O notaciji. Izraz se može dalje pojednostavniti prema pravilu zbrajanja O izraza:

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

Primjenjujući ovo pravilo dva puta možemo napisati da je:

$$T(n) = O(1) + O(n) + O(n) = O(1 + n + n) = O(1 + 2*n)$$

i „odbacujući“ konstante vidimo da *T(n)* pripada u *O(n)* klasu složenosti.

To pravilo možemo i dokazati asimptotskom analizom:

$$O(f(n)) + O(g(n)) = c1*f(n) + c2*g(n) \leq c3*(f(n) + g(n)) = O(f(n) + g(n))$$

Pod uvjetom da zaista možemo naći konstantu c3 za dovoljno veliki n.

Na sličan način možemo dokazati i iduća pravila:

Pravilo:

$$c * O(f(n)) = O(c * f(n)) = O(f(n))$$

Dokaz:

$$c * O(f(n)) \leq c * c1 * f(n) = O(f(n))$$

Pravilo:

$$O(f(n)) * O(g(n)) = O(f(n) * g(n))$$

Dokaz

$$O(f(n)) * O(g(n)) \leq c1 * f(n) * c2 * g(n) = c1 * c2 * f(n) * g(n) = O(f(n) * g(n))$$

Na analogan način se po potrebi mogu odrediti i druga pravila.

Tipične klase složenosti

Kada uspoređujemo algoritme praktično je definirati kategorije složenosti koje će nam ugrubo odmah reći koliko je algoritam kvalitetan. Takva kategorizacija će nam omogućiti da uspostavimo nekakvu hijerarhiju među algoritmima, ali treba uvijek imati na umu da ovakva kategorizacija skriva određene razlike među algoritmima unutar iste klase. Te klase su uglavnom dogovorene i odgovaraju jednostavnim funkcijama veličine ulaza. Takve klase ćete obično naći u osnovnom opisu bilo kojeg algoritma (npr. na Wikipediji). Tipične klase, po rastućoj vremenskoj složenosti su:

$O(1)$ – konstantno vrijeme izvođenja

$O(\log N)$ – logaritamski rast vremena izvođenja

$O(N)$ – linearan rast

$O(N \log N)$ – log-linearan rast ?

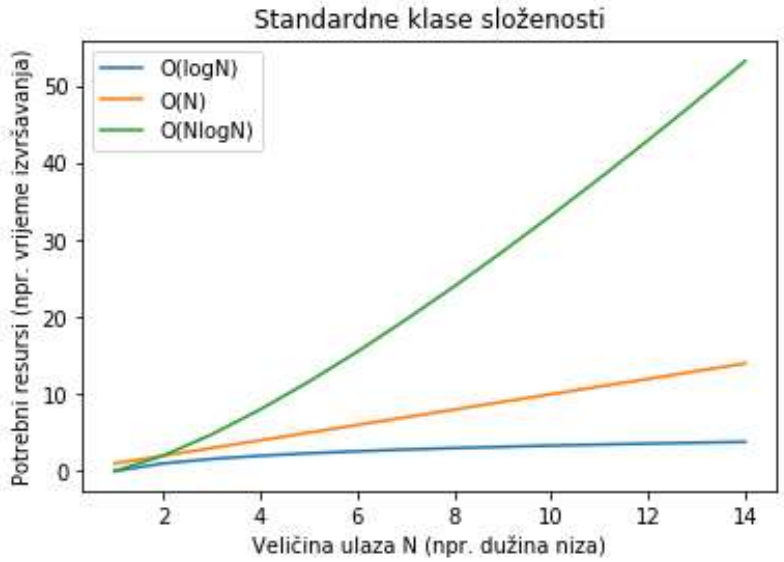
$O(N^2)$ – kvadratni rast

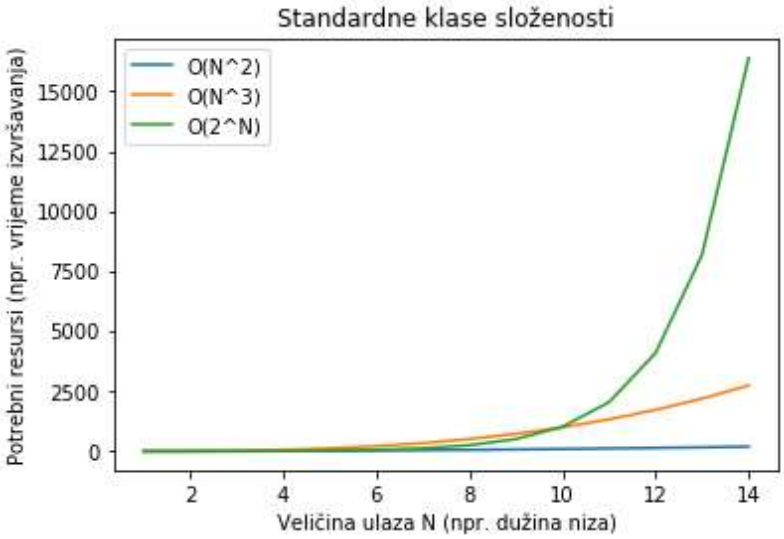
$O(N^c)$ – polinomski rast, $c > 1$

$O(c^N)$ – geometrijski rast, $c > 0$

$O(N!)$ – faktorijelni rast ?

Iz grafičkog prikaza tipičnih složenosti se može vidjeti brzina rasta potrebnih resursa (poput vremena izvođenja) za algoritme koji im pripadaju.





Generalno govoreći, svi algoritmi koji spadaju u klasu $O(N^c)$ ili niže se smatraju „razumnim“ odnosno efikasnim. Svi složeniji algoritmi se u principu smatraju nemogućim za izračun pod uvjetom da je N dovoljno velik. To, naravno, ne odgovara stvarnosti u potpunosti i vjerojatno nećemo moći izračunati u razumnom vremenu algoritam čija je složenost $O(N^{100})$ osim za relativno male veličine ulaza N.

S obzirom na formalnu definiciju asimptotske složenosti, možemo reći i da svaki algoritam koji pripada nekoj klasi složenosti (npr. *O(N)*) pripada i svim višim klasama (npr. $O(N^3)$). Iako je to istina, za ispravnu analizu, zanima nas najmanja moguća klasa odnosno najniža gornja granica u asimptotskoj analizi.

Srednja, najbolja i najgora složenost

Do sada smo pričali samo o prosječnoj složenosti algoritma koja nas najčešće i najviše zanima. Ta srednja složenost opisuje tipično vrijeme izvođenja programa. Ponekad nas zanima i najgori slučaj, odnosno procijeniti gornju vremensku granicu za izvođenje algoritma. Takav slučaj se može konstruirati posebno nepovoljnim ulazom u algoritam. Puno rjeđe nas zanima i najbolji slučaj gdje imamo posebno povoljan ulaz. Sva tri slučaja se mogu opisati na idućem primjeru:

```
int trazi_nulu(int *niz, int n) {
      for (int i = 0; i < n; i++) {
            if (niz[i] == 0)
                  return 1;
      }
      return 0;
}
```

Iz kôda se može vidjeti da funkcija traži prvu nulu u nizu. Koliko će iteracija petlja morati proći ovisi isključivo o sadržaju niza. Kako ne znamo točan sadržaj niza, za tipičan slučaj možemo pretpostaviti da će pretraga morati proći pola niza. Ta pretpostavka je vjerojatno pogrešna jer je nula samo jedan broj od nekoliko milijardi mogućih cjelobrojnih vrijednosti koje integer tip može sadržati. S druge strane, tipični programi će imati samo mali raspon vrijednosti blizu nule. Najispravniji put bi bio procijeniti vjerojatnosti analizirajući potencijalne ulaze, ali to je najčešće nemoguće. Svejedno, petlja će u svakom slučaju imati *c*n* iteracija gdje je c konstanta u rasponu *<0, 1]* odnosno *T(n)* možemo napisati kao *c1 + c*n*. Kako je *c* konstanta, *T(n)* spada u klasu složenosti *O(n)* što odgovara prethodnim analizama. U praksi ćemo često jednostavno pretpostaviti da će funkcija u prosjeku proći pola niza što će nas dovesti do iste klase složenosti *O(n)*. To je srednja složenost i najčešće nas ona najviše zanima.

Možemo pokušati odrediti složenost odnosno klasu složenosti u najgorem slučaju. S obzirom na trajanje izvođenja funkcije, najnepovoljniji ulaz je niz koji ne sadrži nulu. U tom slučaju će broj iteracija biti jednak dužini niza. Očigledno, klasa složenosti u najgorem slučaju je *O(n)*. U ovom slučaju klasa složenosti je jednaka srednjoj složenosti i to nije neuobičajeno. Svejedno, za neke algoritme će najgora složenosti biti za klasu gora nego kod srednjeg slučaja.

I na kraju, najbolji ulaz za funkciju je niz koji odmah na prvom indeksu ima nulu. Funkcija će odraditi samo jednu nepunu iteraciju i završiti. U tom slučaju je vrijeme izvođenja efektivno konstantno i klasa složenosti je *O(1)*.

Zašto bi nas zanimala najgora složenost koja se po definiciji mora događati relativno rijetko? Zamislite da izrađujete program koji će biti na serveru i posluživati zahtjeve različitih klijenata (kao kod tipičnog pristupa preglednikom nekom web sjedištu). Program na serveru prima ulaz dužine n od klijenta, obrađuje ga nekim algoritmom u srednjem vremenu $O(n)$ i vraća odgovor. Za neke zahtjeve/ulaze algoritam se odvije u $O(n^2)$ vremenu odnosno to najgora klasa složenosti za algoritam. Ovakav algoritam je izložen napadima koji bi mogli slati veliki broj nepovoljnih i velikih ulaza i time „zagušiti“ server. Još gore je ako se ove složenosti ne odnose na vrijeme izvođenja nego na memorijsku složenost jer memoriju puno lakše ispunimo. Ako taj problem detektiramo unaprijed analizom algoritma i izračunom najgore složenosti, možda ćemo se odlučiti za neki drugi algoritam ili poduzeti dodatne mjere za zaštitu od takvih napada.

Zašto bi nas zanimala najbolja složenost? Teško da možemo računati na sreću sa vrlo povoljnim ulazima. Svejedno, u rijetkim slučajevima želimo garanciju da je neki izračun vremenski složen bez obzira na ulaz. To se događa kod kriptografskih algoritama za koje želimo da budu zahtjevni za izračun jer nam to garantira da ih nitko neće moći brzo izračunati. Kao primjer, možemo uzeti spremanje lozinki korisnika na serveru nekog web sjedišta. Sve lozinke, svih korisnika se spremaju u posebnu tablicu prilikom registracije korisnika. Problem je što toj tablici netko ipak ima pristup i povremeno se dogodi da ta tablica procuri u javnost (to se događalo i većim imenima poput Facebooka). Da bi se tome doskočilo, u tablicu se lozinke ne spremaju kao tekst nego se sprema rezultat provlačenja lozinke kroz kriptografski algoritam. Taj algoritam nam za neki string (lozinku) vraća nekakav string koji je nemoguće vratiti nazad u originalni string. Prilikom prijave korisnika, upisana lozinka se provlači kroz isti algoritam i rezultat se uspoređuje sa stringom iz tablice. Ako su lozinke iste, rezultati algoritma će biti identični i korisnik će se uspješno prijaviti. Zamislimo da netko uspije kopirati takvu tablicu sa kriptiranim lozinkama i želi otkriti originalne lozinke. To je moguće jedino provlačeći string po string kroz kriptografski algoritam i provjeravajući da li se rezultat nalazi negdje u tablici (tzv. *Rainbow attack*). Ono što želimo je da kriptografski algoritam bude garantirano težak tako da je svako provlačenje lozinke kroz algoritam vremenski zahtjevno (npr. *50ms* na standardnom procesoru). Želimo da to bude teško za sve moguće lozinke bez iznimke, a to nam garantira izračun najbolje složenosti. U stvarnosti je problem još otežan dodatnim tehnikama poput dodavanja „soli i papra“ uz samu lozinku tako da se rezultat algoritma razlikuje od servera do servera i od korisnika do korisnika. Naravno, sve to nas i dalje ne štiti od korisnikovih propusta ili prejednostavnih lozinki.

Koncept složenosti ćemo redovito koristiti za ocjenu algoritama i struktura podataka, ali po potrebi i proširivati dalje u tekstu.

Zadaci

1) Za iduće funkcije odrediti vremensku složenost T(n) i vremensku složenost u O notaciji.

```
int zbroj(int *niz, int n) {
   int z = 0;
   for (int i = 1; i < n; i = i * 2)
```

```
            z = z + niz[i];
    return z;
}

void ispis(int *niz, int n) {
    for (int i = 1; i < n; i++)
            for (int j = n - 1; j > i; j--)
                printf("%d %d\n", niz[i], niz[j]));
}


void izvrti() {
    int i = 0;
    while (i < N) {
            if (i < 10)
                i++;
            else
                i *= 2;
    }
}

void izvrti() {
    int i = N;
    while (i > 0) {
            if (i % 2 == 1)
                i--;
            i = i / 2;
    }
}

void izvrti() {
    int i = 1;
    while (i < N) {
            int j = 0;
            while (j < N)
                j++;
            i = i * 2;
    }
}
```

2) Dokazati iduće pravilo za O() notaciju:
   O(f(n)) * [O(g(n)) + O(h(n))] = O(f(n)g(n) + f(n)h(n)).