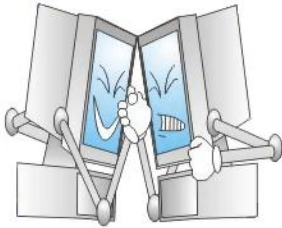


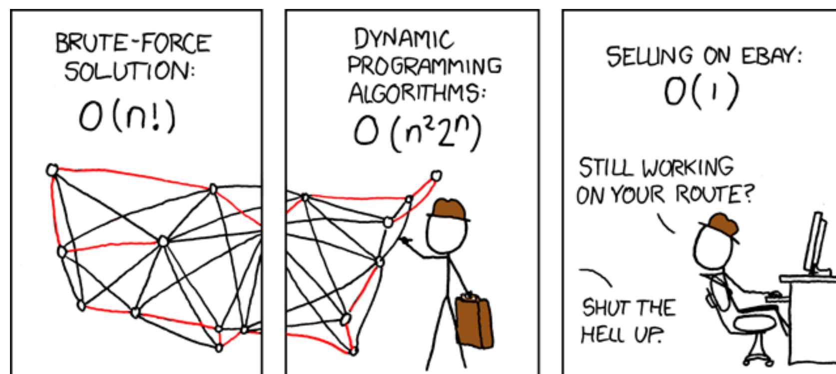
Analiza algoritma

Autor: Andreja Ilić
e-mail: andrejko.ilic@gmail.com

Analiza algoritma[1] predstavlja postupak kojim se predviđa ponašanje i vrši procena potrebnih resursa algoritma. Tačno ponašanje algoritma je nemoguće predvideti, zato se za njegovu analizu razmatraju samo glavne karakteristike, a zanemaruju neki manji faktori sa kojima ćemo se kasnije upoznati. Osnovni metod koji se pri tome koristi je aproksimacija. Na ovaj način se, iz skupa mogućih algoritama za rešavanje nekog konkretnog problema, može izdvojiti najefikasniji (ili klasa efikasnih algoritama). U ovom tekstu ćemo razmatrati vremensku složenost algoritma – memorijski zahtevi će biti pomenuti samo ako nisu u nekim "normalnim" granicama.

Većina učenika, neretko i studenata, nije upoznata sa ovim pojmom i njegovom važnošću. Često u toku takmičenja dobijamo pitanja vezana za ograničenja ulaznih podataka, jer većina takmičara ne razume zašto su ona bitna. Ideja ovog dokumenta je da takmičarima približi ovaj pojam i njegovu značajnost.

U mnogim knjigama ovoj priči nije posvećeno dovoljno pažnje. Neretko se ovaj pojam uvodi jako formalno što predstavlja problem čitaocu da razume njegovu suštinu. Akcenat u ovom dokumentu će biti na razmatranju i analizi konkretnih problema. Razni primeri koje ćemo analizirati će vam pomoći da uočite razlike između algoritama koji rešavaju isti problem.



Slika 1. Strip o složenosti problema putujućeg trgovca
(preuzeto sa **xkcd.com** – sajt posvećen stripovima o ljubavi, sarkazmu, matematici i jeziku)

Ukoliko imate komentara ili dodatnih sugestija povodom teksta, nemojte oklevati da se obratite bilo autoru bilo takmičarskoj komisiji. Takođe ukoliko naiđete na neku grešku, kojih svakako ima, bili bi zahvalni kada bi ste obavestili autora o istoj.

[1] Pod algoritmom podrazumevamo konačan niz nedvosmisleno definisanih naredbi potrebnih za rešavanje konkretnog problema. Formalnu definiciju algoritma ovde nećemo iznositi.

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[2]. 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 (naravno

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

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*[3] 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

[3] Algoritam 2 predstavlja algoritam za sortiranje nizova.

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 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**.

Broj podnizova parne sume

Počećemo sa jednim naizgled jednostavnim primerom. Videćemo da se neka očigledna rešenja mogu dosta ubrzati.

Problem Dat je niz a prirodnih brojeva dužine n. Koliko ima podnizova, sastavljenih od uzastopnih elemenata, čija je suma paran broj?

Drugim rečima, treba naći broj uređenih parova (i, j) , gde je $1 \leq i \leq \mathrm{j} \leq \mathrm{n}$, za koje je

$$a_i + a_{i+1} + \cdots + a_j \text{ paran broj}$$

Primera radi, niz $a = (1,2,3,4)$ ima četiri uzastopna podniza parne sume - $(1,2,3)$, $(1,2,3,4)$, (2) i (4).

Nakon čitanja problema, većina odmah nastupi sa očiglednom idejom: za svaku vrednost promenjivih i i j, sumiraću elemente sa indeksima između njih i proveriti da li je to paran broj. Posmatrajmo kod za ovu ideju prikazan u *Algoritmu 3*.

```
===============================================================
01 brojParnih = 0;
02 for i = 1 to n
03    for j = i to n
04       suma = 0;
05       for k = i to j do
06          suma = suma + a [k];
07       if (suma mod 2 == 0) then
08          brojParnih = brojParnih + 1;
===============================================================
```

Algoritam 3. Prvo rešenje problema parnih podnizova

Složenost *Algoritma 3* je $O(n^3)$, međutim pokušajmo ovu implementaciju da ubrzamo. Linije koda 04 – 06 sumiraju vrednosti elemenata od i do j. Kada fiksiramo levu granicu tog segmenta po kome sumiramo, promenjivu i, posmatrajmo kako se menja suma kada se promenjiva j povećava za 1. Primećujemo da se suma poveća za vrednost $a\,[j]$. Drugim rečima, ne moramo uvek sumirati sve elemente počev od indeksa i. Implementacija ovde ideje je prikazana u *Algoritmu 4*. Složenost novog algoritma je $O(n^2)$.

```
=================================================================
01   brojParnih = 0;
02   for i = 1 to n
03      suma = 0;
04      for j = i to n
05         suma = suma + a [j];
06         if (suma mod 2 == 0) then
07            brojParnih = brojParnih + 1;
=================================================================
```

Algoritam 4

Iz gornjeg primera vidimo da istu ideju možemo implementirati na različite načine. Samom razmatranju implementacije treba posvetiti dosta vremena, nekada čak i više od ideje – **dizajna algoritma**. Naravno najbolji recept je razmatrati ih parelelno. Sa druge strane, od ideje ne treba odustati ukoliko niste dovoljno razmotrili implementaciju.

Međutim, ovaj problem možemo rešiti i u linearnom vremenu. Definišimo niz $suma$ kao

$$suma[k] = a[1] + \cdots + a[k]$$

Sumu elemenata $a[i] + \cdots + a[j]$ možemo preko niza $suma$ izračunati kao $suma[j] - suma[i-1]$, gde uzimamo da je $suma[0] = 0$.[4] Ovaj podniz će imati parnu sumu jedino u slučaju da su vrednosti $suma[j]$ i $suma[i-1]$ iste parnosti. Označimo sa m broj parnih elemenata niza $suma$, računajući i element sa indeksom 0. Ukoliko posmatramo dva parna elementa, podniz koji oni definišu (za koji su oni leva i desna granica) je parne sume. Analogno imamo i sa neparnim elementima. Kako je razlika dva broja parna ako i samo ako su članovi iste parnosti, ovo su jedini mogući slučajevi parnih podnizova. Krajnji rezultat dobijamo kao

$$\binom{m}{2} + \binom{n+1-m}{2}$$

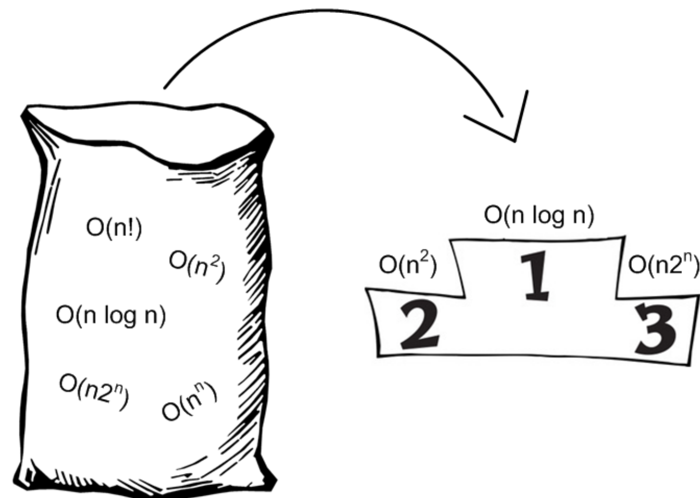
Složenost algoritma je linearna. Niz $suma$ možemo da inicijalizujemo jednim prolaskom kroz niz vezom:

$$suma[0] = 0$$

$$suma[k] = suma[k-1] + a[k], \quad k \geq 1$$

Nakon toga jednostavnim prebrojavanjem parnih odnosno neparnih elemenata niza, rešenje dobijamo gornjoj formulom.

[4] Ovakva konstukcija je jako česta i može se uopštiti za matrice.



Slika 2. Iz „vreće” mogućih algoritama treba izabrati onaj najefikasniji

Intervali

Problem Dat je niz a, dužine n, celobrojnih intervala na realnoj pravoj. Koordinate intervala su iz segmenta $[-M, M]$. Naći celobrojnu tačku sa realne prave koja pripada najvećem broju intervala. Tačka pripada intervalu i kada se nalazi na njegovom rubu.

Ovo je jedan poznatiji primer. Ovde ćemo izneti tri rešenja i upoređivati njihove složenosti.

- *Algoritam A*: Tražena tačka je celobrojna i pripada intervalu $[-M, M]$, jer za sve ostale tačke sigurno znamo da ne pripadaju ni jednom intervalu. Zato možemo proći kroz svaku tačku iz ovog segmenta i ispitati koliko je intervala sadrže. Algoritam možemo ubrzati ukoliko na početku izračunamo najmanju levu koordinatu intervala i najveću desnu koordinatu i ispitivanje svedemo samo na tačke iz ovog segmenta.

  Čak i kod ubrzane verzije segment koje treba ispitati može biti upravo $[-M, M]$ (najgori mogući slučaj). Za svaku tačku iz ovog segmenta obilazimo ceo niz intervala dužine n. Dobijamo da je ukupna složenost ovog algoritma jednaka $O(nM)$. Kako M može biti veliki broj, vidimo da ovaj pristup nije baš efikasan.

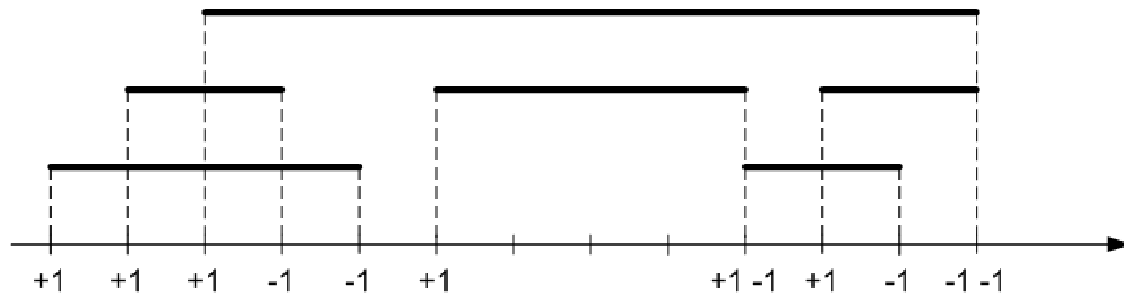
- *Algoritam B:* Označimo sa A traženu tačku (ona ne mora biti jedinstvena). Pokazaćemo da postoji tačka B koja pripada istom broju interval (dakle i ona može biti rešenje) a koja je kraj nekog od datih intervala, konkretno levog. Pretpostavimo suprotno, da tačka A nije levi kraj intervala. Posmatrajmo tačku $A - 1$. Ova tačka pripada istom broju intervala, jer smo pretpostavili da A nije levi kraj (u suprotnom bi ona ispala iz nekog intervala a kako su oni celobrojni to je nemoguće). Ukoliko je $A - 1$ leva granica nekog intervala, nju uzimamo za tačku B. U suprotnom slično rezonovanje primenjujemo na tačku $A - 2$. Kako su intervali iz konačnog segmenta u nekom trenutku moramo naići na kraj nekog od početnih segmenata koji sadrže A.

Iz gornjeg razmatranja vidimo da ne moramo da obilazimo sve tačke iz segmenta u prvom algoritmu. Pretragu možemo vršiti samo po krajevima intervala. Krajeva ima ukupno $2n$, čime dobijamo da je novi algoritam složenosti $O(n^2)$.

- *Algoritam C:* U prethodnom algoritmu smo videli da se isptivanje može svesti na granice ulaznih intervala. Međutim, pri ispitivanju pripadnosti neke tačke ispitujemo sve intervale. Da li ovo možemo ubrzati?

Razmatrajmo primer skupa intervala sa slike:

$$[1, 5], [2, 4], [3, 13], [6, 10], [10,12], [11,13]$$



Slika 3. Primer intervala

Skenirajmo po koordinatama granica redom sa leva na desno i zapisujmo broj intervala kojima pripadaju:

$$1, 2, 3, 3, 2, 2, 3, 3, 3, 2$$

Primećujemo da kada naiđemo na "otvaranje" nekog intervala, njegovu levu granicu, broj pripadnosti intervalima se povećava za 1. Slično kada naiđemo na zatvaranje smanjuje se za 1. Specijalan slučaj je kada se u nekoj koordinati i otvaraju i zatvaraju neki intervali, a kako smo rekli da tačka sa granica pripada intervalima, onda prvo povećavamo broj preseka a tek kada napustimo tačku smanjujemo.

Kreirajmo dva nova niza $cord$ i $value$ dužine $2n$ i inicijalizujmo ih na sledeći način:

$$cord\ [2k-1] = l_k \quad \text{i} \quad value[2k-1] = +1$$
$$cord\ [2k] = d_k \quad \text{i} \quad value[2k] = -1$$

Dakle u niz $cord$ smo poređali vrednosti granica intervala, a paralelno u drugom nizu postavljali $+1$ za levu granicu intervala i -1 za desnu. Sortirajmo ove nizove po vrednostima niza $cord$, dok

paralelno menjamo i elemente u nizu vrednosti. Ukoliko su koordinate jednake manja je ona kojom se segment otvara (dakle prednost imaju $value = 1$). Nakon sortiranja imamo sledeće stanje:

$$cord = (1, 2, 2, 3, 4, 5, 6, 10, 10, 11, 12, 13, 13)$$
$$value = (1, 1, 1, -1, -1, 1, 1, -1, 1, -1, 1, -1, -1, -1)$$

Skeniranje po koordinatama i računanje pripadnosti se sada svodi na jednostavno sabiranje elemenata niza $value$. Pravimo parcijalne sume niza $value$.

$$valueSuma = (1, 2, 3, 2, 1, 2, 3, 2, 3, 2, 3, 2, 1, 0)$$

Kako tražimo maksimalni broj intervala koji sadrži neku tačku, rezultat će biti maksimum parcijalnih suma. Složenost ovog algoritma je $O(n \log n + n) = O(n \log n)$.

Kao što vidimo za isti problem smo imali tri moguća rešenja. Sva tri algoritma su korektna što se rešenja tiče. Međutim njihove složenosti se drastično razlikuju (bar za red veličine). Gledano sa takmičarske strane, broj poena koji bi osvojili po gornjim algoritmima bi aproksimativno bio 10, 40 i 100, redom.

Vrednost polinoma

Izračunavanje vrednosti polinoma u datoj tački x_0 se sastoji u računanju vrednosti $p(x_0)$. Problem nije komplikovan, tako da ćemo kroz ovaj primer proći samo u grubim crtama.

Problem Dat je polinom p stepena n preko niza koeficienata $(p_i)_{i=0}^{n}$. Za konkretnu tačku a izračunati vrednost

$$p(x_0) = p_0 + p_1 a + p_2 a^2 + \cdots + p_n a^n$$

Najjednostavniji pristup je simuliranje izračunavanja svakog sabirka pojedinačno. Naivna implementacija ove ideje dovodi do složenosti $O(n^2)$ (pretpostavljamo da se koeficinete polinoma nalaze u nizu p).

```
=================================================================
01     value = 0;
02     for i = 0 to n do
03        power = 1;
04        for j = 1 to i do
05           power = power * a;
06
07        value = value + power * p [i];
=================================================================
```

Algoritam 5

Redovi 03 – 05 u datom pseudo kodu algoritma stepenuju dati broj a na stepen i. Računanje stepena a^k može se implementirati i u logaritamskom vremenu stepena. Ideja je zasnovana na razlaganjima

$$a^{2k} = a^k a^k \qquad a^{2k+1} = a a^k a^k$$

Na ovaj način možemo problem svesti na problem sa dvostuko manjim eksponentom. Funkcija stepenovanja se najlakše implementira rekurzivno. Malo detaljnije objašenjenje brzog stepenovanja možete naći u [2].

```
=================================================================
01      function Stepen (Integer a, Integer n)
02         if n = 0 then
03            return 1;
04         if n = 1 then
05            return a;
06
07         tmp = Stepen (a, n DIV 2);
08         if (n MOD 2 = 0) then
09            return tmp * tmp;
10         else
11            return tmp * tmp * a;
=================================================================
```

Algoritam 6. Funckija brzog stepenovanja

Problem se može rešiti i u linearnom vremenu. Naime, stepen ne moramo uvek računati od početka već ga možemo nadograđivati. U svakom koraku, nakon promene vrednosti stepen ćemo množiti sa a.

```
=================================================================
 01     value = 0;
 02     power = 1;
 03     for i = 0 to n do
 04           value = value + power * p [i];
 05           power = power * a;
=================================================================
```

Algoritam 7

Na kraju napomenimo da se problem može implemenitrati i preko **Hornerove šeme**. Složenost će takođe biti linearna, međutim ovim pristupom imamo minimalni broj operacija sabiranja i množenja.

Elementi sa datom sumom

Problem Dat je niz a prirodnih brojeva dužine n. Za dati prirodni broj s ispitati da li postoje dva elemente niza a, čija je suma jednaka datom broju s.

Navedeni primer se često sreće kao podproblem nekog većeg problema i na jako lep način ilustruje ideju o složenosti algoritma. Prva ideja, koja je svakako korektna ali ne i efikasna, jeste da se za svaka dva elementa niza proveri da li u sumi daju broj s. Ovaj pristup ima kvadratnu složenost.

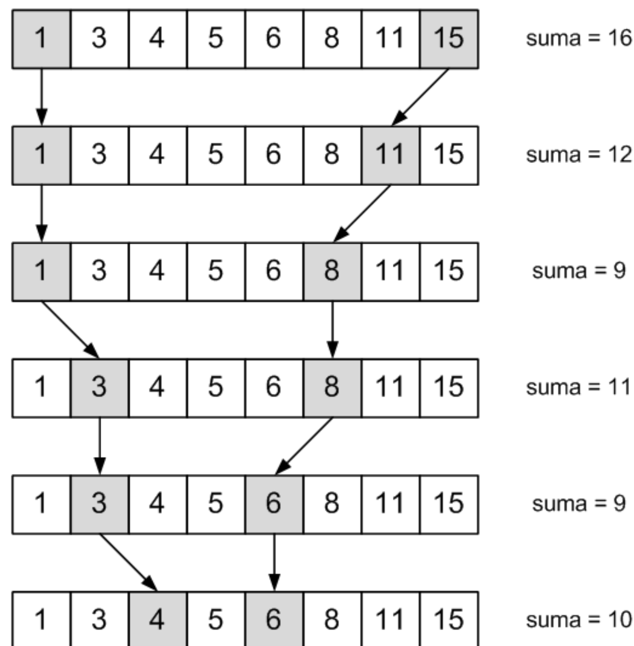
Mana ovog pristupa je što ispitujemo parove elemenata bez ikakvog poretka. Zbog toga nemamo nikakvih pretpostavki o sumi novog para na osnovu prethodnih. Da bi mogli imati neku intuiciju o ponašanju sume elemenata sortirajmo niz a u neopadajući poretku. Definišimo sa $s(i,j)$ sumu elemenata $a[i]$ i $a[j]$, pri čemu je $i <$ j. Kako je niz sortiran, $a[k] \leq a[k+1]$, imamo da važe nejednakosti

$$s(i,j) \leq s(i,j+1) \text{ i } s(i,j) \leq s(i+1,j)$$

Postavimo na početku vrednost indeksa i, levu granicu, na 1, a indeks j, desnu granicu, na n. Kada desnu granicu j smanjujemo, i sama suma $s(i,j)$ se smanjuje. Moguća su tri slučaja:

1. Naišli smo na element sa indeksom j tako da je $s(1,j) > s$. U ovom slučaju smanjujemo vrednost indeksa j.
2. Naišli smo na element sa indeksom j tako da je $s(1,j) = s$. Tada prekidamo izvršavanje programa i kao rešenje vraćamo upravo ovaj par elemenata.
3. Naišli smo na element sa indeksom j tako da je $s(1,j) < s$. Sada nema potreba smanjivati indeks j i ispitivati sumu za istu vrednost indeksa $i = 1$, upravo zbog gornjih nejednakosti.

Kako smo smanjivanjem desne granice j prekinuli u trenutku kada je prvi put dobijeno $s(1,j) < s$, imamo da je $s(1,j+1) > s$. Zato pretragu ne moramo ponovo počinjati od $j = n$, već samo nastaviti gde smo zadnji put imali da je suma bila veća: $j+1$. Drugim rečima, promenjivu j smanjujemo sve dok ne dobijemo rešenje ili ne dobijemo vrednost koja je manja od s, kada povećavamo i. Ako dobijemo da je $j < i$, prekidamo i prijavljujemo da rešenje ne postoji. Pogledati *Sliku 4* koja ilustruje ponašanje pokazivača na konkretnom primeru.



Slika 4. Primer pretrage para u nizu $a = (1,3,4,5,6,8,11,15)$ za $s = 10$.

Rastojanje između indeksa i i j je na početku jednako $n - 1$. U svakom koraku se njihovo rastojanje smanjuje za 1. Kako smo gore napomenuli da ukoliko se "preskoče" rešenje ne postoji, ukupan broj koraka ne može biti veći od početnog rastojanja. Kako na početku treba sortirati niz[5] dobijamo da je složenost $O(n \log n + n) = O(n \log n)$. Pseudo kod ovog algoritma je dat u *Algoritmu 5*.

```
=================================================================
01     Sort(a);
02     i = 1;
03     j = n;
04     while (i <= j)
05        if (a [i] + a [j] = s) then
06           break;
07        if (a [i] + a [j] > s) then
08          j = j + 1;
09        else
10          j = j + 1;
11          i = i + 1;
12
13     if (i <= j) then
14        return true;
15     return false;
=================================================================
```

Algoritam 8

Još jedan mogući pristup nakon sortiranja niza je **binarna pretraga:** za svaki element $a\,[i]$, ispitujemo da li u nizu postoji njegova dopuna do sume s tj. element čija je vrednost $s - a[i]$. Složenost ovog algoritma je takođe $O(n \log n)$, međutim prvi pristup je za nijasnu efikasniji.

Pri analizi nekog problema možemo naići na dva rešenja iste složenosti. U razmatranje njihove međusobne efikasnosti možemo uključiti konstante ili druge sabirke koje smo zanemarili, ali to neće mnogo uticati na performanse. U prethodnom primeru smo videli da samo imali algoritme čije su složenosti bile $O(n \log n + n)$ i $O(n \log n + n \log n)$. Efikasniji algoritam je svakako prvi.

Čitaocu preporučujemo da nakon ovog primera sam reši *Zadatak 010*.

Quicksort

Quicksort[6] je jako zanimljiv algoritam za sortiranje niza. Verovatno ste svi do sada čuli da je to najbrži algoritam za sortiranje niza. U prvom delu ovog dokumenta smo razmatrali složenost jednoj od najjednostavnijih algoritama za sortiranje. Složenost quicksort-a, u najgorem slučaju, je $O(n^2)$ gde je n broj elemenata niza. Iako je spor u najgorem slučaju, on predstavlja najbolji izbor za ovaj problem jer je

[5] Algoritme sortiranje možete proučiti u bilo koji knjizi koja se bavi dizajnom i analizom algoritama.
[6] Ovde nećemo objašnjavati kako sam algoritam radi. Opis algoritma možete naći skoro u bilo kojoj knjizi koja se bavi programiranjem.

neverovatno brz u srednjem tj. očekivanom vremenu sortiranja: očekivano vreme sortiranje je $O(n \log n)$, dok je konstanta sakrivena iz ove složenosti jako mala.

Sa druge strane, pored ovog algoritma postoji i **sortiranje razdvajanje** koje se zanisniva na metodi **podeli pa vladaj**. Ovaj algoritam je u stranoj literaturi poznat kao **merge sort**. Složenost je $O(n \log n)$ u najgorem slučaju. Međutim, u očekivanom vremenu quicksort je dosta brži. Sličan slučaj imamo i sa **radix sort-om**.

Kako bi bili sigurni da će quicksort raditi u očekivanom vremenu, pre sortiranja niza treba napraviti slučajnu permutaciju a zatim nju sortirati. Na taj način možete izbeći one najgore slučajeve i vreme vratiti na očekivano.

Kako možemo kreirati slučajnu permutaciju? Označimo dati niz sa a, njegovu dužinu sa n. Naintuitivniji pristup je dodeliti svakom elementu niza slučajan broj $b\,[i]$, a zatim sortirati niz a ali po vrednostima niza b. Problem kod ovog metoda je kako obezbediti da su dodeljene vrednosti jedinstvene. Kako se i sami slučajni brojevi generišu pseudo-slučajnim algoritmom ovde se možemo zadovoljiti i jako malom verovatnoćom pojavljivanje istih vrednosti. Može se pokazati da ukoliko vrednosti biramo na slučajan način iz segmenta $[1, n^3]$, verovatnoća da su svi elementi različiti jednaka je $1 - \frac{1}{n}$. Složenost ovog pristupa je jednak složenosti sortiranje.

Najpraktičniji način generisanja slučajne permutacijije prikazan je u *Algoritmu 6*. Složenost algoritma je linearna. Dokaz korektnosti algoritma nećemo izložiti ovde. Za one koji žele više informacija, mogu pogledati [1].

```
=========================================================
01   for i = 1 to n do
02      j = random (i, n);
03      zameni (a [i], a [j]);
=========================================================
```

Algoritam 9

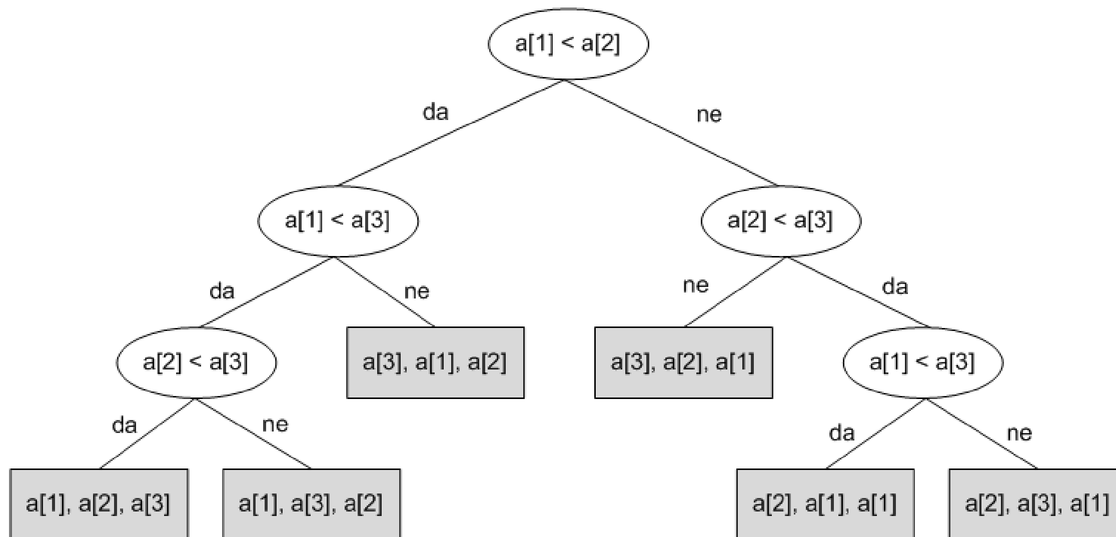
Zanimljiva napomena je da ukoliko se indeks j inicijalizuje kao $random(1, i)$ ne dobija se slučajna permutacija. Dokaz ove činjenice ostavljamo čitaocu.

Priču o quicksort-u završićemo dokazom njegove optimalnosti. Do sada samo uvek računali složenost algoritama i pokazivali kako je neki pristup bolji od drugog. Međutim, nekada je potrebno postaviti pitanje: da li postoji brži algoritam? Kako bi odgovorili na ovo pitanje potrebno je naći donju granicu složenosti algoritma za konkretan problem. Ovo je nemoguće odrediti za generalni slučaj, jer se dokaz odnosi na sve moguće algoritme. Neohpodno je napre specificirati model koji odgovara porizvoljnom algoritmu, a zatim dokazati da je vremenska složenost proizvoljnog algoritma obuhvaćenog tim modelom veća ili jednaka od donje granice.

U daljem delu odeljka pokazaćemo da je optimalna složenost algoritama za sortiranje niza upravo $O(n \log n)$. Model koji ćemo koristiti pri dokazivanju se zove **stablo odlučivanja**. Stablo odlučivanja

modelira izračunavanje koje se najvećim delom zasniva na upoređivanju. Definiše se kao potpuno binarno stablo sa dve vrste čvorova: unutrašnjim čvorovima i listovima. Svakom unutrašnjem čvoru odgovara jedno "pitanje" sa dva moguća odgovora, pri čemu je svaki od njih pridružen grani koja izlazi iz tog čvora. Svakom listu je pridružen jedan od mogućih izlaza algoritma. Izračunavanje počinje od korena stabla. U svakom čvoru se postavlja pitanje u vezi sa ulazom i u zavisnosti od odgovora nastavlja levim ili desnim potomkom. Kada se dostigne list, izlaz pridužen njemu je izlaz izračunavanja. Kako bi ova priča bila jasnija pogledati *Sliku 4* na kojoj je prikazano sortiranje niza dužine 3 pomoću stabla odlučivanja.

Vremenska složenost algoritma definisanog preko stabla odlučivanja jednaka je njegovoj visini, odnosno maksimalnom broju pitanja (operacija) koje je potrebno postaviti kako bi se rešenje jednoznačno odredilo.



Slika 5. Primer stabla odlučivanja za sortiranje niza a dužine $n = 3$

Teorema Proizvoljno stablo odlučivanja za sortiranje n elemenata niza ima visinu $n \log n$.

Dokaz: Ulaz u algoritam je niz a dužine n. Izlaz je niz u sortiranom poretku, drugim rečima permutacija ulaza. Kako ulaz može biti u porivoljnom poretku, svaka permutacija brojeva $(1, 2, \dots, n)$ treba da se pojavi kao mogući izlaz stabla odlučivanja. Prema tome, za svaku permutaciju treba da postoji bar jedan list. Permutacija ima $n!$, pa pošto je stablo binarno, njegova visina je najmanje $\log_2(n!)$. Koristeći Stirlingovu formulu

$$n! \approx \sqrt{2\,\pi\,n}\left(\frac{n}{e}\right)^n$$

dobijamo da je $\log_2(n!) \approx n \log n$. □

Ovaj dokaz donje granice pokazuje da svaki algoritam za sortiranje zasnovan na upoređivanju ima složenost veću ili jednaku od $n \log n$. Sortiranje je moguće izvršiti i korišćenjem nekih drugih osobina. Primera radi ukoliko bi ograničenje brojeve bilo malo (malo u smislu da možemo definisati niz te veličine**), sortiranje razvrstanjem (eng. counting sort)** bi vratilo rezultat u linearnom vremenu. Ovo ne protivureči dokazu prethodne teoreme, jer se ovde koristi činjenica da vrednosti brojeva mogu da se efikasno koriste kao adrese. Takođe, ovaj algoritam nije generalni i ne može se primeniti na proizvoljan niz.

Rezime

Rezime priče ćemo izneti kroz česta pitanja koje je autor dobijao u toku priprema.

1. *Zašto nam je uopšte potreban efikasan algoritam? Može li kupovina bržeg računara rešiti ovaj problem?*

Naravno novac ne može sve kupiti. Pored toga što su intelektualno izazovniji, efikasniji algoritmi će se sigurno izvršavati brže i na sporijem računaru od loših algoritama na super računarima, za dovoljne veličine ulaza. Boljim računarom možete ubrzati sve algoritme, ali procenat tog ubrzanja ne može pomoći u slučaju eksponencionalnih algoritama. U prilog ovoga ide i **Murov zakon** koji predviđa razvijanje računara – snaga računara se udvostručuje približno na svakih 18 – 24 meseca.

2. *Kada možemo reći da je neki algoritam dobar, a kada da je optimalan?*

Za algoritam možemo reći da je dobar ukoliko se pri upoređivanju sa drugim postojećim algoritmima za ovaj problem pokaže kao efikasniji. Na takmičenjima je tako lako zaključiti. Naime, ograničenja su tu da bi pomogla u odgovoru na ovo pitanje. Upoređivanjem složenosti sa efikasnošću računara na kojem se testira lako možete zaključiti da li je vaš algoritam efikasan ili ne.

Drugi deo pitanje je iskomentarisan u prethodnom delu teksta. Na ovo pitanje je nekada nemoguće odgovoriti. Kod primera quicksort-a videli smo da odogovr zahteva poznavanje donje granice vremena izvršavanja svih algoritama za konkretan problem. To znači da za dati problem ne samo poznati algoritmi, već i oni koji još nisu otkriveni, moraju se izvršavati za vreme koje je veće ili jednako od neke granice. Napomenimo i to da se kod takmčarskih zadataka može naći i algoritam efikasniji od datih granica.

3. *Šta je potrebno od matematičkog aparata za analizu složenosti algoritma?*

Težina analize kao pratioca procesa konstrukcije algoritma ne bi trebalo da bude izgovor za odsutajanje od nje. Bitno je dobiti bar nekakvu prredstavu o efikasnosti algoritma. Za njeno računanje obično je dovoljno poznavanje osnovnih suma, zasnovanih na aritmetičkoj i geometrijskoj progresiji, kao i osnovnih principa kombinatorike. U neki slučajevima, naročito kada se koristi rekurzija, nailazi se na diferencne jednačine. Obično su to neki elementarni slučajevi.

(preuzeto sa **xkcd.com** – sajt posvećen stripovima o ljubavi, sarkazmu, matematici i jeziku)

Razni zadaci

Zadatak 001 [Državno takmičenje, Srbija, 2008] Profesor matematike je postavio Dragančetu sledeći zadatak. Na osnovu datog niza brojeva a_1, a_2, ..., a_n, Dragande treba da za svaku trojku indeksa (i, j, k), gde je $1 \le i < j < k \le n$, napiše na tabli najveći od brojeva a_i, a_j i a_k. Zatim treba da izračuna ostatak koji daje zbir svih brojeva koji su napisani na tabli pri deljenju sa 10007. Profesor je obećao Dragančetu peticu za kraj školske godine, ako dobije tačno rešenje pre kraja časa. Pomozite Dragančetu da što brže dobije tačan rezultat. Dužina niza nije veća od 30.000, dok su elementi niza iz segmenta $[-10^5, 10^5]$.

Ulaz:
```
4
3 -1 2 2
```

Izlaz:
11

Zadatak 010 [Državno takmičenje, Srbija, 2003] Dat je skup od $n \le 5000$ duži različitih dužina. Napisati program koji određuje koliko se nepodudardnih trouglova može obrazovati od tih duži. Za obrazovanje trougla se jedna duž može koristiti samo jednom.

Ulaz:
```
4
2 3 6 4
```

Izlaz:
2

Zadatak 011 [TopCoder, SRM 408] Mali Perica je dobio $N \le 1000$ svećica za rođendan. Odlučio je da na sledeći način proslavi: prvog dana će upaliti jednu sveću da gori sat vremena; drugog dana će upaliti dve sveće u periodu od sat vremena; i tako dalje; k-tog dana će upaliti k sveća. Svakog dana upaljena sveća se smanji tačno za 1 cm. Kada sveća dostigne visinu 0, ne mo·ze se više upotrebljavati. Odrediti maksimalni broj dana tokom kojih Perica može da slavi rodjandan.

Ulaz:
```
2 2 2
```

Izlaz:
3

Zadatak 100 [Županjsko takmičenje, Hrvatska, 2007] Japanska buba uletela je u tunel pun prepreka: stalagmita (stoje na pod) i stalaktita (vise sa plafona). Tunel je dužine n (gdje je n paran broj) i visine h. Prva prepreka je stalagmit, a zatim se naizmenično smenjuju stalaktiti i stalagmiti. Japanskoj bubi se ne da izbegavati prepreke, nego odabere jednu od h visina i zaleti se s jednog kraja tunela na drugi, te svojim kung-fu veštinama poruši sve prepreke na putu.

Objašnjenje datog primera

Zadane su dimenzije tunela i dužine svih prepreka. Napišite program koji određuje koliko najmanje prepreka buba mora porušiti da bi prošla na drugu stranu, te na koliko različitih visina se postiže ta najmanja vrijednost.

Ulaz:	Izlaz:
6 7 1 5 4 4 5 3	2

Zadatak 101 Naći broj brojeva iz segmenta $[A;B]$ čiji je zbir cifara paran.

Ulaz:	Izlaz:
5 15	5

Zadatak 110 **[Državno takmičenje, Srbija, 2007]** Dragančetu je dosadilo da živi u gradu pa je rešio da sagradi sebi vikendicu, u kojoj će u miru i tišini moći da sprovodi svoje programerske ideje. Kako je Draganče veoma vredan, rešio je da sam napravi nacrte njegove kuće iz snova. No, na početku je trebalo da odredi mesto na kome će se graditi. Kako Draganče nije baš bogat (matematika i programiranje nije bilo isplativo kako je on mislio), odredio je maksimalni budžet koji može da potroši za kupovinu placeva. Naravno, želi da vikendica bude što lepša, pa je odredio i donju granicu budžeta. Sad je u nedoumici gde da je sagradi. Pomozite Dragančetu da odredi broj mogućih mesta za gradnju vikendice.

Mesta na kojima Draganče može da gradi vikendicu data su u obliku pravougane matrice dimenzije $n \times m$ $(1 \leq n,m \leq 150)$, gde polja predstavljaju placeve. Za svaki plac je data njegova cena. Cene ne prelaze 10^4. Vikendica je u obliku pravougaonika. Na pravougaoniku se može sagraditi vikendica ukoliko suma njenih parcela upada u granice budžeta. Granice budžeta su iz segmenta $[1,10^9]$.

Primer

Ulaz:	Izlaz:
3 3 2 3 1 0 0 0 1 0 0 0 1	7

Literatura

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to algorithms*, The MIT Press, 2001.

[2] Miodrag Živković, *Algoritmi*, Matemtički fakultet, Beograd, 2000.

[3] Donald E. Knuth, *The Art of Computer Programming*, Addison Wesley, Volume 1, 1980.

[4] Dejan Živković, *Osnove dizajna i analize algoritama*, Računarski fakultet, Beograd, 2007.