# 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

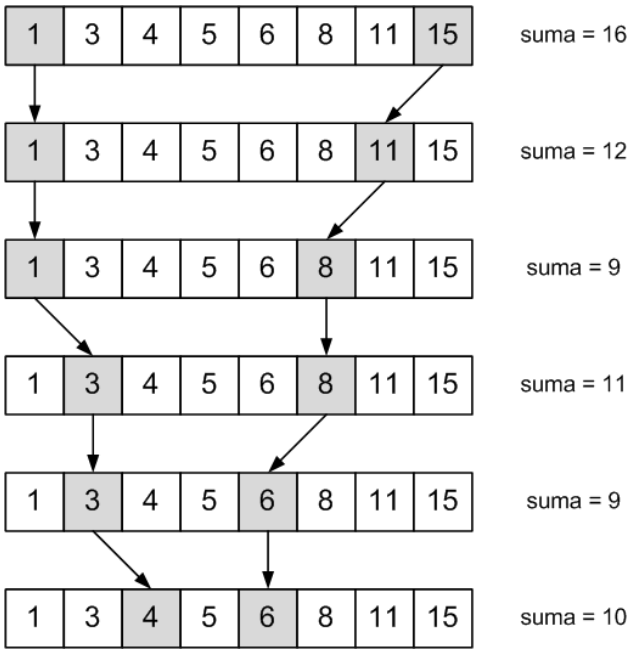
$$s(i,j) \leq s(i,j+1) \text{ i } s(i,j) \leq s(i+1,j)$$

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. Našli smo na element sa indeksom $j$ tako da je $s(1,j) > s$. U ovom slučaju smanjujemo vrednost indeksa $j$.
2. Našli 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. Našli 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[1] 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

[1] Algoritme sortiranje možete proučiti u bilo koji knjizi koja se bavi dizajnom i analizom algoritama.

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.