Kako brže izračunati a^n

Goran Delač

Algoritmi precizno opisuju način rješavanja nekog problema na rastavljajući ga na konačan broj koraka. Svaki korak opisan je nekom instrukcijom. Od velike nam je važnosti znati koliko je algoritam kojeg primjenjujemo, učinkovit. Želimo da on bude što brži i da pritom zauzme što manje memorije računala. Definiramo dva aspekta složenosti: **vremenski** i **prostorni** (*memorijski*).

Vremenska složenost definira se obično kao **procjena broja operacija** potrebnih da se **algoritam izvrši** u ovisnosti o broju **ulaznih podataka**. Kako se u računalu svi procesi odvijaju diskretno (u ovisnosti radnog takta procesora), povećanje broja operacija povećava i potrebno vrijeme izvršavanja. Želimo li da se naš algoritam izvrši u nekom *realnom* vremenu, moramo voditi računa o tome kako broj ulaznih podataka utječe na broj operacija koje će biti potrebne za izvršenje algoritma.

Složenost se iskazuje asimptotskim ponašanjem niza (a_n) koji predstavlja gornju među za broj operacija potrebnih da se algoritam izvrši. Drugim rječima, a_n predstavlja najveći mogući broj operacija potrebnih da se algoritam izvrši u ovisnosti o nekom n. Primjerice, kad bismo htjeli izračunati zbroj

$$1 + 2 + \ldots + n$$

na računalu bismo to napravili na sljedeći način:

$S := 0$;
za $i := 1$ *do* n **radi**
  $S := S + i$;

Računalo za ovaj algoritam mora izvršiti točno n operacija za svaki $n \in \mathbb{N}$. Stoga je u ovom slučaju $a_n = n$.

Kažemo da niz (a_n) asimptotski dominira nizom (b_n) ako postoje $n_0 \in \mathbb{N}$, $M > 0$ takvi da:

$$|b_n| \leq M|a_n|, \qquad \forall n \geq n_0.$$

Neka je $O(a_n)$ skup svih nizova b_n koji su asimptotski dominirani nizom (a_n). To označavamo $b_n = O(a_n)$. Ovakav način obilježavanja naziva se ***O-notacija***. Ukoliko istodobno vrijedi $b_n = O(a_n)$ i $a_n = O(b_n)$ onda za nizove a_n i b_n kažemo da asimptotski rastu istom brzinom. Postoje M_1, M_2 i $n_0 \in \mathbb{N}$ takvi da za svaki $n \geq n_0$ vrijedi:

$$M_1|a_n| \leq |b_n| \leq M_2|a_n|. \tag{3}$$

Ovaj slučaj se označava $\Theta(n)$. Recimo, algoritam koji zbraja

$$1^3 + 2^3 + \ldots + n^3$$

provodimo kao:

$S := 0$;
za $i := 1$ *do* n **radi**
  $S := S + i \cdot i \cdot i$;

Budući da pri svakom koraku provodimo 3 operacije, broj operacija jednak je $b_n = 3n$. Nizovi $a_n = n$ i $b_n = 3n$ rastu istom brzinom jer za $M_1 = 1$ i $M_2 = 3$ vrijedi (3).

Neka je broj ulaznih podataka u algoritam n. Označimo sa $c(n)$ složenost algoritma (procjenu broja instrukcija u ovisnosti o n).

Vrlo nam je bitno znati kako se složenost asimptotski ponaša.

Složenost algoritma može biti:

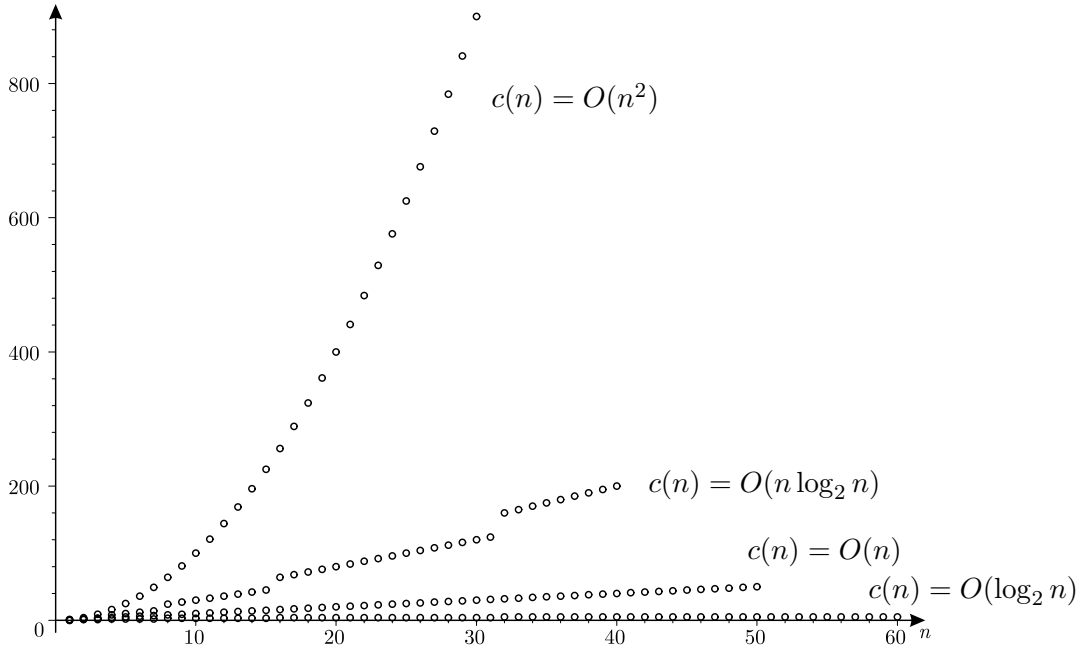
- Logaritamskog rasta, $c(n) = \Theta(\log n)$
- Polinomijalnog rasta, $c(n) = \Theta(n^k)$, gdje je k neki prirodan broj
  Poseban slučaj za $k = 1$ dobivamo složenost linearnog rasta, za $k = 2$ kvadratnog rasta itd.

- Eksponencijalnog rasta, $c(n) = \Theta(a^n)$, $a > 1$
- Faktorijelnog rasta, $c(n) = \Theta(n!)$
- Nekog drugog rasta

U računarstvu se algoritam smatra prihvatljivim ako mu je složenost ***najviše polinomijalnog rasta*** i to što **manjeg stupnja**. (Prethodno navedeni algoritmi linearne su složenosti.) Algoritmi s većim asimptotskim rastom složenosti nisu izvedivi u realnom vremenu za veći broj ulaznih podataka n. Za ilustraciju kako složenost ovisi o broju ulaznih podataka n može pomoći sljedeća tablica:

n	$\lfloor \log_2 n \rfloor$	n	$\lfloor n \cdot \log_2 n \rfloor$	n^2	n^3	2^n	$n!$
5	2	5	11	25	125	32	120
20	4	20	86	400	8000	1048576	$2.43 \cdot 10^{18}$
40	5	40	212	1600	64000	$1.09 \cdot 10^{12}$	$8.15 \cdot 10^{47}$
60	5	60	354	3600	216000	$1.15 \cdot 10^{18}$	$8.32 \cdot 10^{81}$

Vidljivo je da se algoritmi sa eksponencijalnim i faktorijelnim brzinama rasta složenosti ne mogu izračunati u realnom vremenu za veći broj ulaznih podataka (za svaku operaciju računalu je potrebno izvjesno vrijeme).



Slika 1. *Usporedba trajanja algoritama ovisno o njihovoj složenosti.*

Potenciranje

U ovome dijelu pozabavit ćemo se potenciranjem. Ideja ovog algoritma vrlo je jednostavna. Imamo zadan neki realni broj $a > 0$ i $n \in \mathbb{N}$. Želimo izračunati a^n preko osnovnih operacija zbrajanja i množenja. Možemo realizirati petlju koja računa članove niza $y_k := y_{k-1} \cdot a$, $k = 1, \ldots, n$ gdje je $y_1 = a$ i naravno $y_n = a^n$. Ovo obično radimo ovako:

$y := a$;
za $i := 1$ *do* $n - 1$ **radi**
  $y := y \cdot a$;

Nakon što se petlja izvrši, $y = a^n$. Da bismo došli do rezultata, moramo petlju u svakom slučaju izvršiti $n - 1$ puta, pa je složenost ovog algoritma:

$$c(n) = O(n)$$

linearna. *Možemo li isti problem riješiti na efikasniji način?* Pretpostavimo da je $n = 2^k$, $k \in \mathbb{N}$ i rastavimo a^n na sljedeći način:

$$a^n = (a^2)^{2^{k-1}} = ((a^2)^2)^{2^{k-2}} = \ldots = (\ldots (a^2)^2 \ldots)^2.$$

Konstruirajmo petlju koja će izvršavati k uzastopnih kvadriranja broja a:

$y := a$;
za $i := 1$ *do* k **radi**
  $y := y \cdot y$;

Budući da je $n = 2^k$, složenost ovog algoritma je $c(n) = k = \log_2 n$.
Dobili smo logaritamsku složenost, bitno poboljšanje u odnosu na prethodni slučaj. Ukoliko je $n \neq 2^k$ možemo eksponent rastaviti po potencijama broja 2 (tj. pretvoriti u binarni zapis).

Neka je $n = 19$. 19 po potencijama broja 2 (binarno) rastavimo

$$19 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 10011_{(2)}.$$

Rastavimo

$$a^{19} = a^{2^4+2^1+2^0} = a^{2^4} \cdot a^{2^1} \cdot a^{2^0} = a^{16} \cdot a^2 \cdot a$$

Za a^{16} potrebna su nam 4 uzastopna kvadriranja (jer je $16 = 2^4$). Njima su obuhvaćeni i a, a^2 pa su nam potrebna još samo dva množenja da dođemo do rezultata. Do konačnog rezultata trebalo nam je ukupno 6 koraka što je znatno bolje od 18 koraka koliko bi nam trebalo da koristimo algoritam običnog potenciranja. Ovaj primjer lijepo ilustrira kako najjednostavniji algoritam često nije najučinkovitiji način da se dođe do rješenja.

Algoritam za brzo potenciranje možemo realizirati na sljedeći način:

1. Učitaj a, n. Postavi $y := 1$.
2. Podijeli n s 2, spremi kvocijent q i ostatak r.
3. Ako je $r = 1$ spremi $y := y \cdot a$.
4. Ako je $q = 0$ prekini program.
5. Postavi $n := q$, $a := a \cdot a$
6. Skok na korak 2.

Pogledajmo kako se izvršava algoritam na primjeru a^{19} ($n = 19$).

KORAK	n	q	r	a	y
1.	19	9	1	a	a
2.	9	4	1	a^2	a^3
3.	4	2	0	a^4	a^3
4.	2	1	0	a^8	a^3
5.	1	0	1	a^{16}	a^{19}

Ovaj algoritam naziva se još i **brzo potenciranje**. Ovo u pseudokodu pišemo:

ucitaj(a, n);
$y := 1$;
dok $q \neq 0$ **radi**
  $q := n$ div 2; $r := n$ mod 2;
  ako $r = 1$
    $y := y \cdot a$;
  $n := q$;
  $a := a \cdot a$;

Složenost algoritma brzog potenciranja je:

$$c(n) = O(\log_2 n).$$

Ako želimo potencirati broj a na eksponent 10^6, prvom bi algoritmu trebalo 999999 ponavljanja, dok bi ih algoritam brzog potenciranja trebao svega 19.