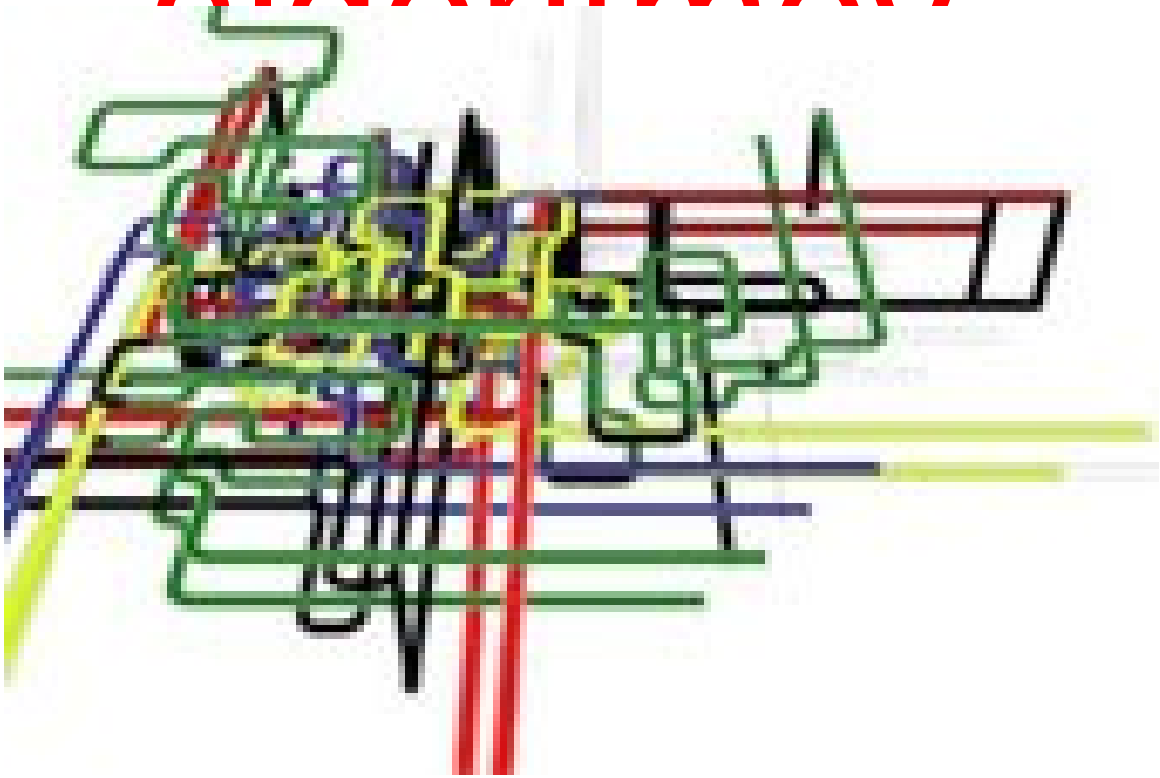
Kompleksnost programov in algoritmov



Kompleksnost programov

Kompleksnost programov ali algoritmov se večinoma ukvarja s tem, koliko časa potrebuje nek program za svoje izvajanje, koliko virov pri tem uporablja, kako razumljiva je njegova koda.

Tudi najbolj preproste probleme lahko rešujemo z različnimi metodami. Lahko jih tudi kodiramo na različne načine.

Nekatere probleme lahko zakodiramo z več vrsticami, vendar morda isto dosežemo z le nekaj stavki v zanki. Velikost programa torej nima direktne povezave z njegovo kompleksnostjo.

Kompleksnost pogosto navezujemo na urejanje in elementov v seznamih in njihovo iskanje. Sezname elementov lahko preiskujemo zaporedno, lahko pa uporabimo metodo “binarnega razreza”

Primer algoritma: Binarno iskanje

- Cilj
  - Iskanje neke vrednosti v zbirki vrednosti
- Ideja
  - Deli in vladaj

Binarno iskanje(2)

11	23	35	47	53	60	72	82	91	99

- Zahteve
  - Zbirka mora biti v obliki polja (“array”)
    - Za skok na poljuben element polja uporabljamo indekse
  - Zbirka naj bo urejena (sortirana)
- Učinkovitost
  - Zelo hitro iskanje
  - Ne potrebujemo dodatnega prostora

Ideja binarnega iskanja

0	1	2	3	~~4~~	~~5~~	~~6~~	~~7~~
11	23	35	47	~~53~~	~~60~~	~~72~~	~~82~~

35 < 47

Iskalno območje: 0 – 7

Iskano število 35

Ideja binarnega iskanja

0	1	2	3	4	5	6	7
11	23	35	47	53	60	72	82

23 < 35

Iskalno območje: 0 – 3
Iskano število: 35

Ideja binarnega iskanja

0	1	2	3	4	5	6	7
11	23	35	47	53	60	72	82

35== 35

Iskalno območje: 2 - 3
Iskano število 35

Kompleksnost programov

Primitivne operacije

- Osnovna računanja, ki jih izvaja algoritem
- Odkrijemo jih v psevdo kodi
- Precej neodvisne od programskega jezika
- Točna definicija niti ni pomembna
- Predvidevamo, da potrebujejo konstanten čas izvajanja

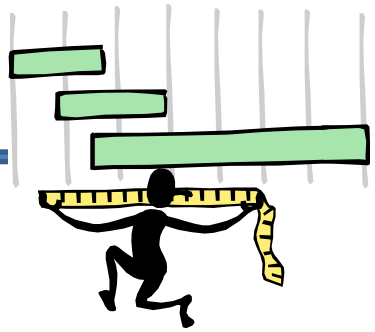
- Primeri:
  - Ocenjevanje izraza
  - Prirejanje vrednosti neki spremenljivki
  - Indeksiranje polja
  - Klic metode
  - Povratek iz metode

Štetje primitivnih operacij

S proučevanjem psevdo kode lahko določimo maksimalno število primitivnih operacij, potrebnih za izvedbo algoritma in v funkciji velikosti vhoda

Algoritem: arrayMax (A,n)	**Št operacij**
currentMax ← A[0]	2
for (i=1;i<n; i++)	2***n***
(i = 1 enkrat , i<n n krat , i++ (n-1) krat	
if(A[i] currentMax then	2(***n*** – 1)
currentMax ← A[i]	2(***n*** – 1)
return currentMax	1
	Skupaj 6***n*** –1

Ocena časa izvajanja



- Algoritem ***arrayMax*** v najslabšem primeru izvede $\mathbf{6n - 1}$ primitivnih operacij.

Definirajmo:

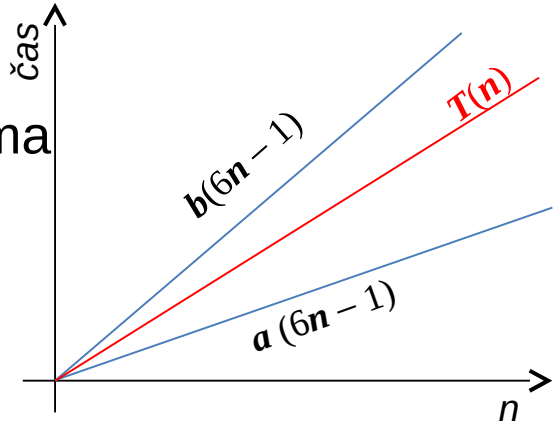
$\boldsymbol{a}$ = čas, potreben za najhitrejšo primitivno operacijo

$\boldsymbol{b}$ = čas, potreben za najpočasnejšo primitivno operacijo

- Naj bo $\boldsymbol{T}(\boldsymbol{n})$ najslabši čas za ***arrayMax.*** Tedaj velja
-

$$\boldsymbol{a}\,(6\boldsymbol{n} - 1) \leq \boldsymbol{T}(\boldsymbol{n}) \leq \boldsymbol{b}(6\boldsymbol{n} - 1)$$

- Torej je $\boldsymbol{T}(\boldsymbol{n})$ omejen z dvema linearnima funkcijama



Učinkovitost

- Če imamo več algoritmov za reševanje danega problema, kateri je najhitrejši?
- Če imamo nek algoritem, ali je sploh uporaben oziroma dovolj učinkovit v praksi?
- Koliko časa potrebuje nek algoritem?
- Koliko pomnilniškega prostora potrebuje nek algoritem?
- V splošnem sta tako časovna kot prostorska zahtevnost algoritma odvisna od velikosti njegovega vhoda.

DEMO

Učinkovitost: merjenje časa

- Merimo čas v sekundah?
  - + uporabno v praksi
  - – odvisno od jezika, prevajalnika in procesorja.

- Štetje korakov algoritma?
  - + neodvisno od prevajalnika in procesorja
  - – odvisno od razdrobljenosti korakov.

- Štetje značilnih operacij? (n.pr. aritmetičnih oper. v matematičnih algoritmih, primerjanj v iskalnih algoritmih)
  - + odvisno od samega algoritma
  - + merimo notranjo učinkovitost algoritma.

Primer: algoritmi potenciranja*(1)*

Preprost algoritem potenciranja:

Za izračun b^n:

1. Nastavimo p na 1.
2. Za $i = 1, \ldots, n$, ponavljamo:
   2.1. p $\leftarrow$ p krat b.
3. Končamo z odgovorom p.

Analiza algoritma potenciranja

$$p = b^n$$

1. Nastavimo *p* na 1.
2. Za *i* = 1, …, *n*, ponavljamo:
   2.1. p ← *p* krat *b*.
3. Končamo z odgovorom

Analiza (štetje množenj):

Korak 2.1 izvaja množenje.
Ta korak ponovimo *n* krat.

Število množenj = *n*

Implementacija v Javi

```
static int power (int b, int n) {
// Return b^n (where n is non-negative).
    int p = 1;
    for (int i = 1; i <= n; i++)
        p *= b;
    return p;
}
```

Primer: Bister algoritem potenciranja

- Zamisel : $b^{1000} = b^{500} \times b^{500}$. Če poznamo b^{500}, lahko izračunamo b^{1000} z le enim dodatnim množenjem!
- **Bister algoritem potenciranja**:

Za izračun b^n:

1. Nastavimo p na 1, nastavimo q na b, nastavimo m na n.
2. Dokler $m > 0$, ponavljamo:
   2.1. Če je m lih, množimo $p \leftarrow p$ krat q.
   2.2. Razpolovimo m (pozabimo na ostanek).
   2.3. Množimo $q \leftarrow q$ krat q .
3. Končamo z odgovorom p.

DEMO

Analiza bistrega algoritma potenciranja

1. Nastavimo p na 1, nastavimo q na b, nastavimo m na n.
2. Dokler $m > 0$, ponavljamo:
   - 2.1. Če je m lih, množimo $p \leftarrow p$ krat q.
   - 2.2. Razpolovimo m (pozabimo na ostanek).
   - 2.3. Množimo $q \leftarrow q$ krat q .
3. Končamo z odgovorom p.

- Analiza (štetje množenj):

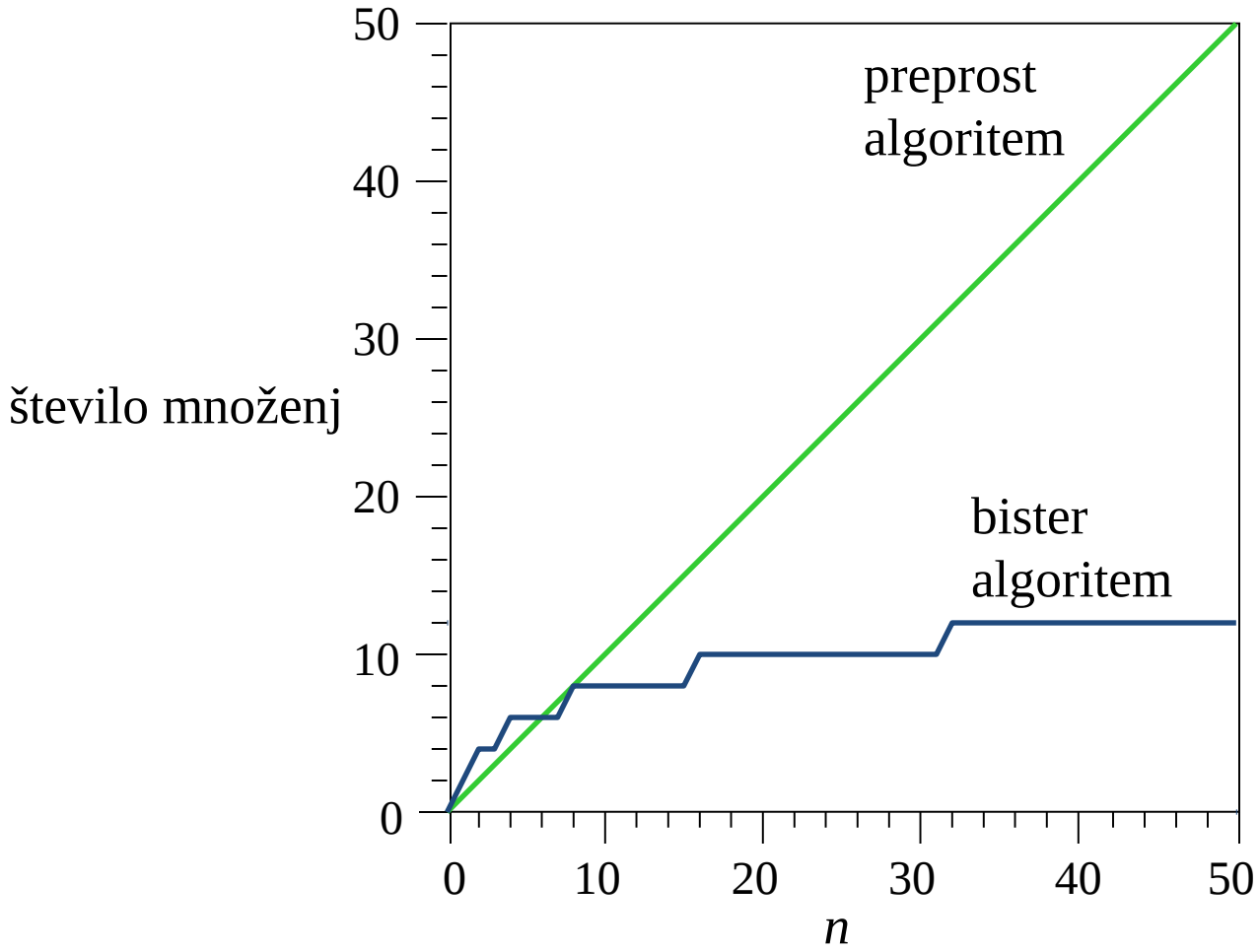
Koraki 2.1–2.3 izvedejo skupaj največ 2 množenji.
Ponavljamo jih, dokler moramo razpolavljati vrednost m (ob zanemarjanju ostanka), da ta doseže vrednost 0, torej, "floor($\log_2 n$) + 1" krat.

Maksimalno število računanj = $2(\text{floor}(\log_2 n) + 1) = 2\,\text{floor}(\log_2 n) + 2$

Implementacija v Javi

```
static int power (int b, int n) {
// Return b^n (where n is non-negative).
   int p = 1, q = b, m = n;
   while (m > 0) {
      if (m%2 != 0)  p *= q;
      m /= 2;  q *= q;
   }
   return p;
}
```

Primerjava algoritmov potenciranja



Performančna analiza

- Določanje časovnih in pomnilniških zahtev algoritr
- Ocenjevanju časa pravimo analiza časovne kompleksnosti
- Ocenjevanju pomnilniških zahtev pravimo analiza prostorske kompleksnosti.
- Ker je pomnilnik cenen in ga imamo kar nekaj, redko izvajamo analizo prostorske kompleksnosti
- Čas je “drag” , zato analizo pogosto omejimo na časovno kompleksnost.

Kompleksnost

- Za veliko algoritmov težko ugotovimo točno število operacij.
- Analizo si poenostavimo:
  - identificiramo izraz, ki najhitreje raste
  - zanemarimo izraze s počasnejšo rastjo
  - v najhitreje rastočem izrazu zanemarimo konstantni faktor.
- Tako dobljena formula predstavlja **časovno kompleksnost algoritma**. Osredotočena je na **rast** časovne zahtevnosti algoritma.
- Podobno velja za prostorsko zahtevnost.

Časovna kompleksnost algoritma potenciranja

- Analiza preprostega algoritma potenciranja (štetje množenj):

  Število množenj = n

  Čas, potreben za izvedbo, je sorazmeren z n.

  Časovna kompleksnost je **reda *n***. To zapišemo kot $\boldsymbol{O(n)}$.

Čas.kompleksnost bistrega algoritma pot.

- Bister algoritem potenciranja (štetje množenj):

Maks. število množenj =

$$2 \text{ floor}(\log_2 n) + 2$$

Zanemarimo izraz s počasnejšo rastjo, +2.

Poenostavimo na $2 \text{ floor}(\log_2 n)$

nato na $\text{floor}(\log_2 n)$

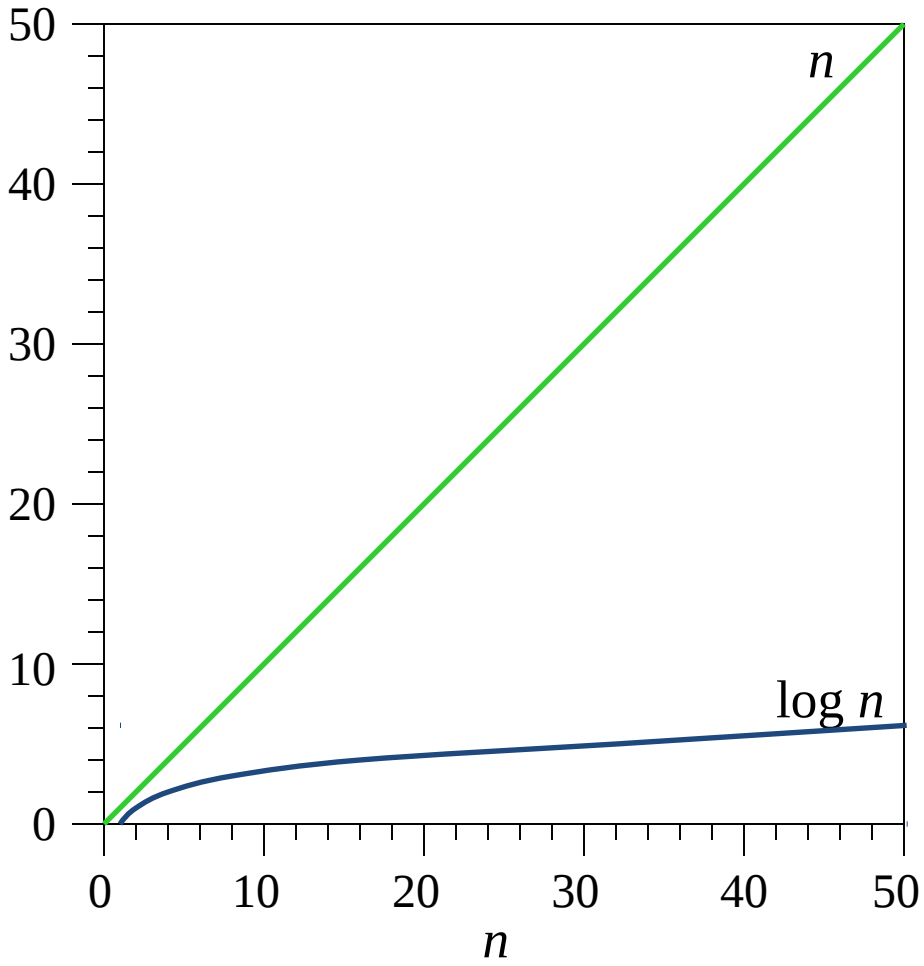
Zanemarimo konstantni faktor, 2.

nato na $\log_2 n$

Časovna kompleksnost je **reda log *n***.
kar zapišemo kot ***O*(log *n*)**.

Zanemarimo floor(), ki v povprečju odšteje 0.5, kar je konstanta.

Primerjava čas. kompleksnosti algoritmov potenc.



Notacija O

- Vidimo, da je algoritem $O(\log n)$ boljši od algoritma $O(n)$ pri velikih vrednostih n. $O(\log n)$ predstavlja počasnejšo rast kot $O(n)$.
- Kompleksnost $O(X)$ pomeni “of **order** X”, torej rast, sorazmerno z X.

  Pri tem smo zanemarili izraze s počasnejšo rastjo in konstantne faktorje.

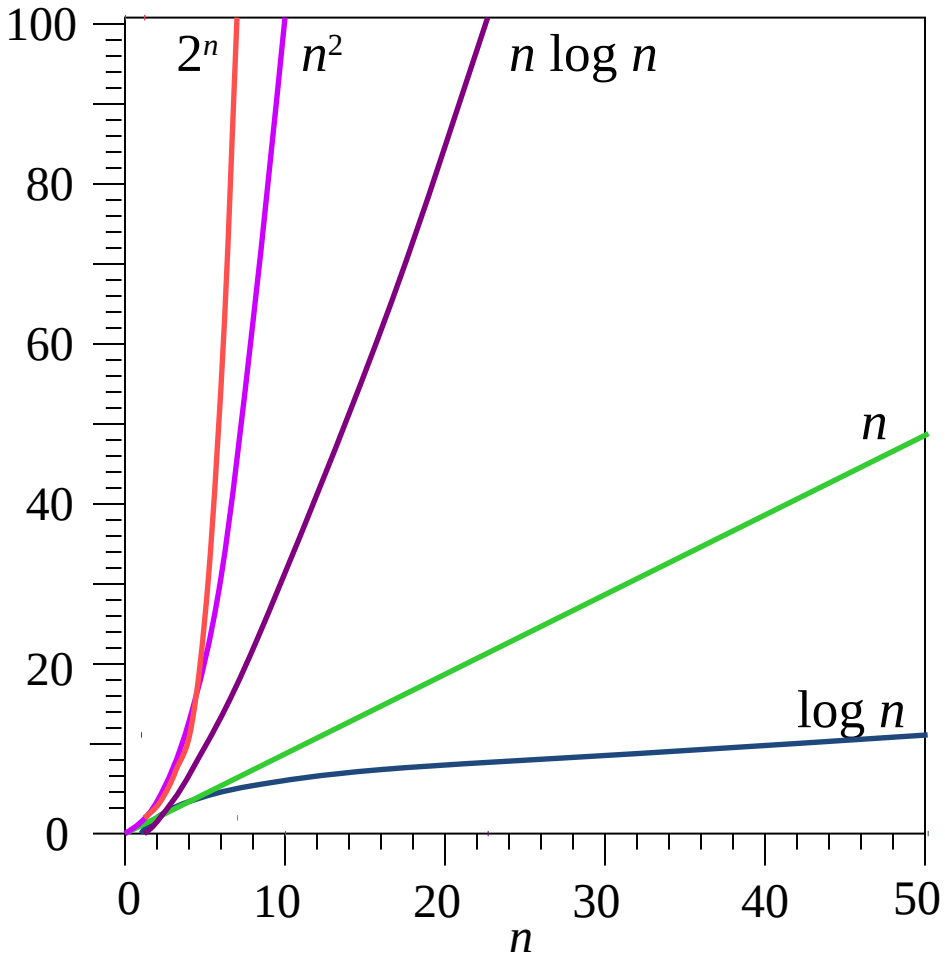
Notacija O primeri kompleksnosti

$O(1)$	**konstanten čas**	(izvedljivo)
$O(\log n)$	**logaritmični čas**	(izvedljivo)
$O(n)$	**linearen** čas	(izvedljivo)
$O(n \log n)$		**log linear** čas (izvedljivo)
$O(n^2)$	**kvadratični** čas	(včasih izvedljivo)
$O(n^3)$	**kubični** čas	(včasih izvedljivo)
$O(2^n)$	**eksponenčni** čas	(redko izvedljivo)

Primerjava časov rasti

1	1	1	1	1
log n	3.3	4.3	4.9	5.3
n	10	20	30	40
n log n	33	86	147	213
n^2	100	400	900	1,600
n^3	1,000	8,000	27,000	64,000
2^n	1,024	1.0 milijon	1.1 miliarda	1.1 trilijon

Grafična ponazoritev časov rasti



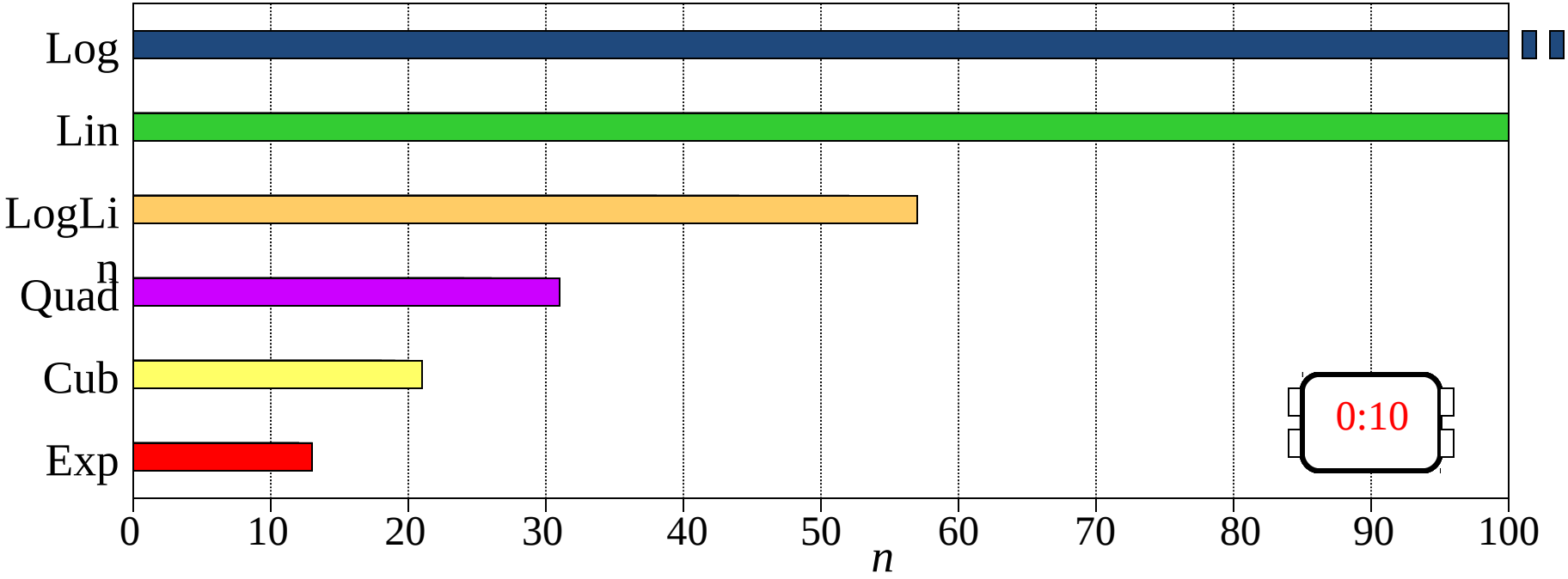
Primerjava v sekundah

- Imejmo problem, v katerem moramo obdelati *n* podatkov.
- Za reševanje problema imejmo na voljo več algoritmov. Predpostavimo, da na danem procesorju ti algoritmi potrebujejo za svojo izvedbo naslednje čase:

AlgoritemLog:	$0.3 \log_2 n$	sec
Algoritem Lin:	$0.1\, n$	sec
Algoritem LogLin:	$0.03\, n \log_2 n$	sec
Algoritem Quad:	$0.01\, n^2$	sec
Algoritem Cub:	$0.001\, n^3$	sec
Algoritem Exp:	$0.0001\, 2^n$	sec

Primerjava v sekundah *(2)*

- Primerjamo, koliko podatkov (*n*) lahko obdela posamezen algoritem v 1, 2, …, 10 sekundah:



Računska kompleksnost

- Primerja rast dveh funkcij
- Neodvisnost od množenja s konstanto in od efektov nižjega reda
- Metrika
  - Notacija “Veliki O” $O()$
  - Notacija “Veliki Omega” $\Omega()$
  - Notacija “Veliki Theta” $\Theta()$

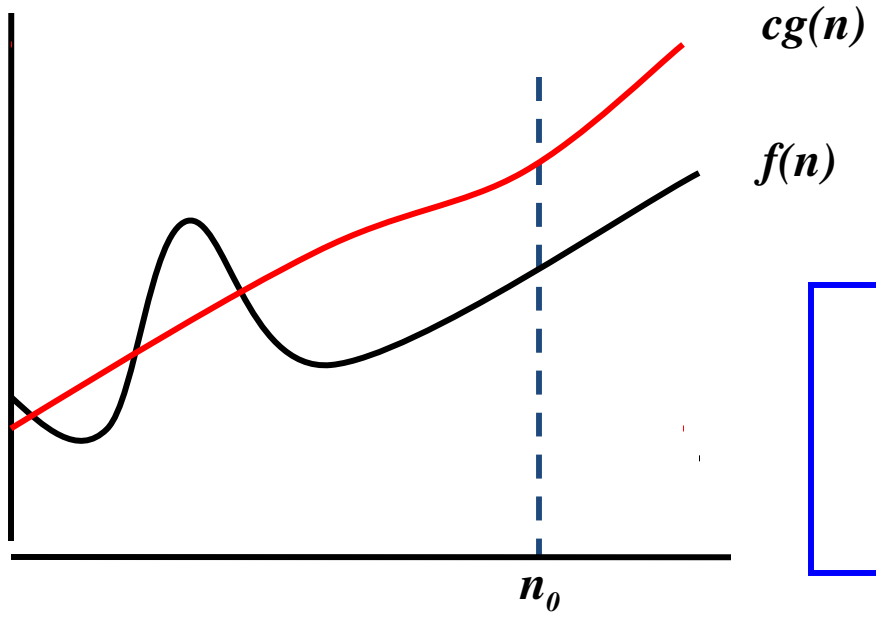
Notacija "veliki O" (big O)

In mathematics, computer science, and related fields, **big O notation** describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. Big O notation allows its users to simplify functions in order to concentrate on their growth rates: different functions with the same growth rate may be represented using the same O notation.

Big O notation is also called **Big Oh notation**, **Landau notation**, **Bachmann–Landau notation**, and **asymptotic notation**. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function; associated with big O notation are several related notations, using the symbols *o*, Ω, ω, and Θ, to describe other kinds of bounds on asymptotic growth rates.

Notacija "veliki O"

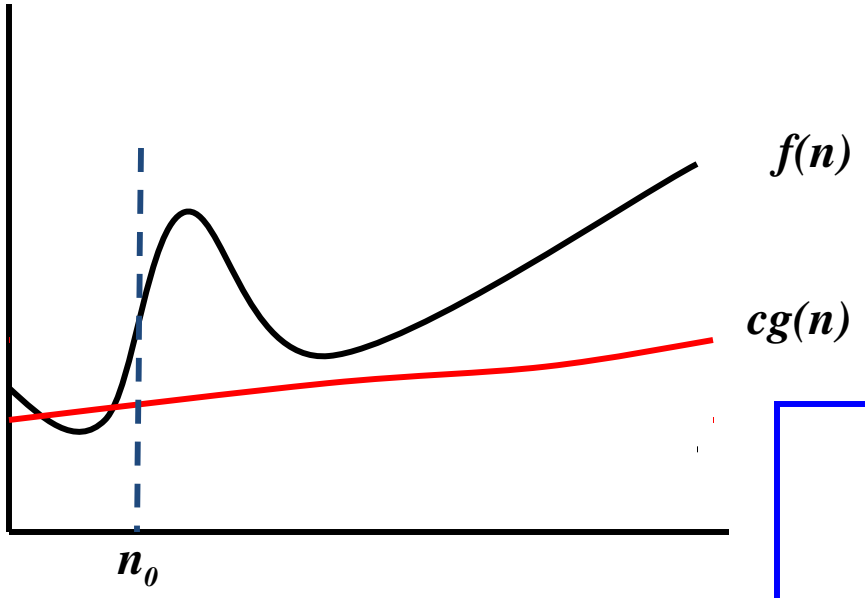
- Naj bo n nenegativno celo število, ki predstavlja velikost vhoda v nek algoritem
- Naj bosta f(n) in g(n) dve pozitivni funkciji, ki predstavljata število osnovnih operacij (instrukcij), ki jih algoritem potrebuje za svojo izvedbo



$f(n)$ je sčasoma navzgor omejena z $g(n)$

Notacija veliki “Omega”

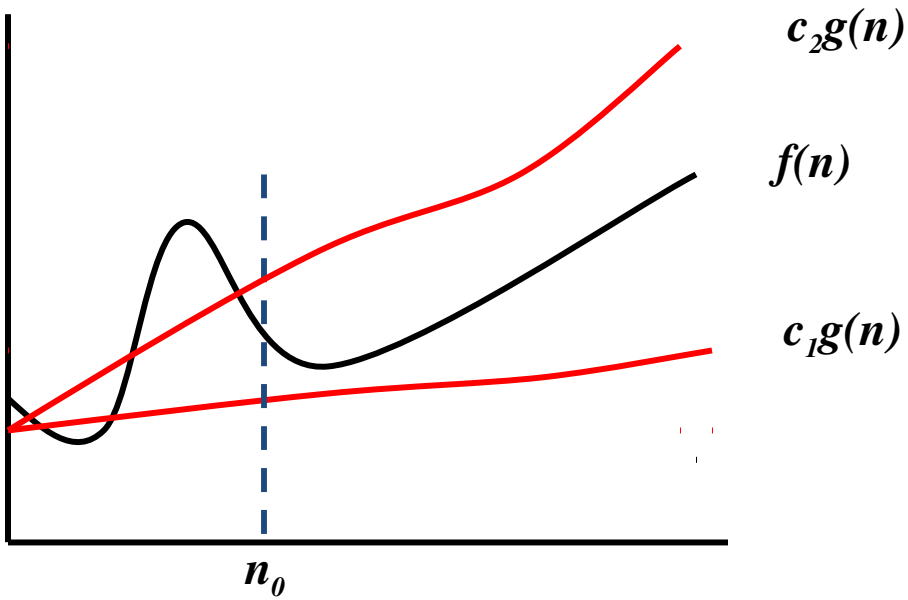
- $f(n) = \Omega(g(n))$
  - iff $\exists\, c, n_0 > 0$ **s.t.** $\forall\, n \geq n0\,,\ 0 \leq cg(n) \leq f(n)$



$f(n)$ je sčasoma navzdol omejena z $g(n)$

Notacija veliki “Theta”

- $f(n) = \Theta(g(n))$
  - iff $\exists\, c_1, c_2, n_0 > 0$ **s.t.** $0 \le c_1 g(n) \le f(n) \le c_2 g(n), \quad \forall\, n >= n_0$



$f(n)$ ima dolgoročno enako rast kot $g(n)$

Analogija z realnimi števili

- $f(n) = O(g(n))$ $\quad (a \leq b)$
- $f(n) = \Omega(g(n))$ $\quad (a \geq b)$
- $f(n) = \Theta(g(n))$ $\quad (a = b)$
- …Ta analogija ni povsem točna, vendar je tako razmišljanje o kmpleksnosti funkcije prikladno
- Svarilo: “Skrite konstante” v teh notacijah imajo pri realnih številih praktično posledico.

Primeri

$$3n^2 + 17$$

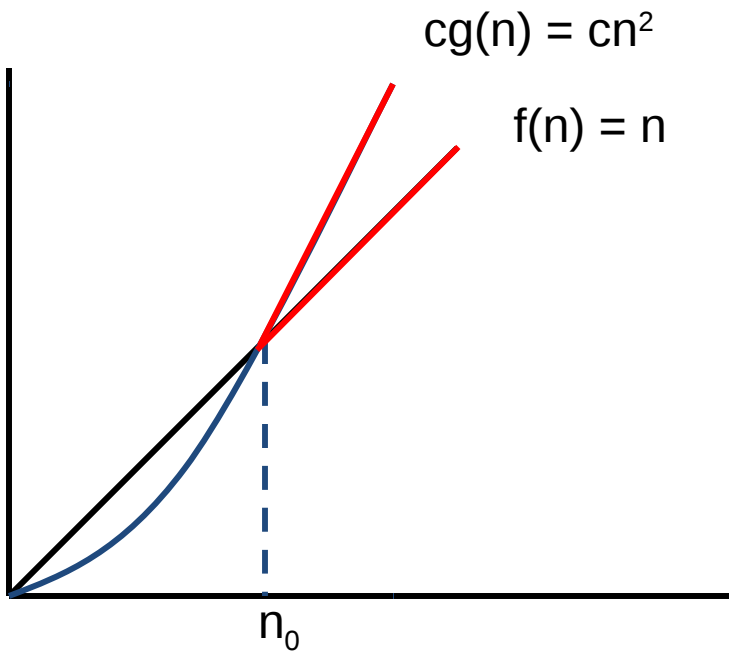
- $\Omega(1)$, $\Omega(n)$, $\Omega(n^2)$ ➔ spodnje meje
- $O(n^2)$, $O(n^3)$, ... ➔ zgornje meje
- $\Theta(n^2)$ ➔ točna meja

Notacija Veliki O (2)

- $f(n)=O(g(n))$ če obstaja pozitivna konstanta C in nenegativno celo število n_0 tako, da velja

$$f(n) \leq Cg(n) \quad \text{za vse } n \geq n_0.$$

- Pravimo, da je g(n) zgornja meja za f(n).



Primeri

- $F(n) = O(1)$
  - $F(n) = 1$
  - $F(n) = 2$
  - $F(n) = c$ (konstanta)
- $F(n) = O(\log(n))$
  - $F(n) = 1$
  - $F(n) = 2\log(n)$
  - $F(n) = 3\log_2(4n^5) + 1$
  - $F(n) = c_1\log_{c2}(c_3n^{c4}) + O(\log(n)) + O(1)$

Primeri (2)

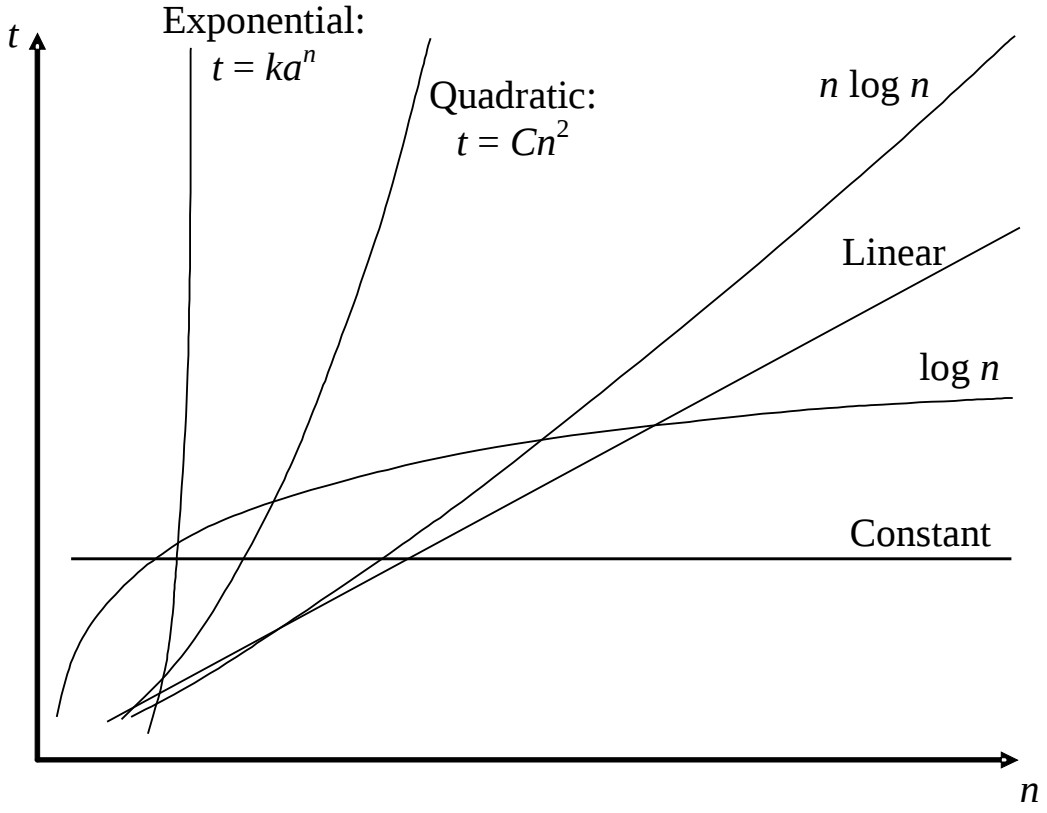
- $F(n) = O(n)$
  - $F(n) = 2\log(n)$
  - $F(n) = n$
  - $F(n) = 3n + 1$
  - $F(n) = c_1 n + O(n) + O(\log(n))$
- $F(n) = O(n\log(n))$
  - $F(n) = 3n + 2$
  - $F(n) = n\log(n)$
  - $F(n) = 3n\log_4(5n^7) + 2n$
  - $F(n) = c_1 n\log_{c2}(c_3 n^{c4}) + O(n\log(n)) + O(n)$

Primeri (3)

- $F(n) = O(n^2)$
  - $F(n) = 3n\log(n) + 2n$
  - $F(n) = n^2$
  - $F(n) = 3n^2 + 2n + 1$
  - $F(n) = c_1n^2 + O(n^2) + O(n\log(n))$

Povzetek o notaciji Veliki-O

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(a^n)$$



Analiza kompleksnosti

- Ocenimo n = velikost vhoda
- Izoliramo vsako atomarno aktivnost (operacijo), ki jo moramo upoštevati
- Najdimo $f(n)$ = število atomarnih aktivnosti, izvedenih pri vhodu velikosti n
- Kompleksnost algoritma = kompleksnost $f(n)$

Časovna kompleksnost zanke

```
for (j = 0; j < n; ++j) {
    // 3 atomarne operacije
}
```

- Kompleksnost = $\Theta(3n) = \Theta(n)$

Zanke s stavkom break

```
for (j = 0; j < n; ++j) {
   // 3 atomarne operacije
   if (pogoj) break;
}
```

- Zgornja meja= $O(4n) = O(n)$
- Spodnja meja= $\Omega(4) = \Omega(1)$
- Kompleksnost= $O(n)$

Zaporedje zank

```
for (j = 0; j < n; ++j) {
    // 3 atomarne operacije
}
for (j = 0; j < n; ++j) {
    // 5 atomarne operacije
}
```

- Kompleksnost = $\Theta(3n + 5n) = \Theta(n)$

Vgnezdene zanke

```
for (j = 0; j < n; ++j) {
    // 2 atomarni operaciji
    for (k = 0; k < n; ++k) {
        // 3 atomarne operacije
    }
}
```

Kompleksnost = $\Theta((2 + 3n)n) = \Theta(n^2)$

Zaporedni stavki

```
for (i = 0; i < n; ++i) {
   // 1 operacija
   if(condition) break;
}

for (j = 0; j < n; ++j) {
   // operacija
   if(condition) break;
   for (k = 0; k < n; ++k)
   {
       // 3 operacije
   }
   if(condition) break;
}
```

Komleksnost =

$$O(2n) + O((2+3n)n)$$

$$= O(n) + O(n^2)$$

$$= O(n^2)$$

If-then-else

```
if(condition)
   i = 0;
else
   for(j = 0; j < n; j++)
       a[j] = j;
```

Kompleksnost =

$$= O(1) + max\ (\ O(1), O(N))$$

$$= O(1) + O(N)$$

$$= O(N)$$

Zaporedno iskanje

Imamo neurejen vektor a[ ], iščemo, če v njem nastopa X

```
for (i = 0; i < n; i++) {
        if (a[i] == X) return true;
}
return false;
```

Velikost vhoda: n = **a.size()**
Kompleksnost = $O(n)$

Binarno iskanje

- Imamo urejen vektor a[].V njem iščemo lokacijo elementa X

```
unsigned int binary_search(vector<int> a, int X)
{
        unsigned int low = 0, high = a.size()-1;

        while (low <= high) {
                int mid = (low + high) / 2;
                if (a[mid] < X)
                        low = mid + 1;
                else if( a[mid] > X )
                        high = mid - 1;
                else
                        return mid;
        }
        return NOT_FOUND;
}
```

- Velikost vhoda: n = **`a.size()`**
- Kompleksnost: $O($ k **iteracij** x $($ 1 **primerjava** + 1 **prirejanje**$)$ **v zanki**$)$ $= O(log(n))$

Štetje iteracij pri binarnem iskanju

```
unsigned int binary_search(vector<int> a, int X) {
   unsigned int low = 0, high = a.size()-1;
   while (low <= high) {
      int mid = (low + high)/2;
      if (a[mid] < X)
         low = mid + 1;
       else if( a[mid] > X )
          high = mid - 1;
       else
          return mid;
    }
    return NOT_FOUND;
}
```

Št.iteracij	prostor iskanja
1	n
2	n/2
3	n/4
k	$n/2^{(k-1)}$

Štetje iteracij pri binarnem iskanju (2)

```
unsigned int binary_search(vector<int> a, int X) {
  unsigned int low = 0, high = a.size()-1;
  while (low <= high) {
    int mid = (low + high)/2;
    if (a[mid] < X)
      low = mid + 1;
          else if( a[mid] > X )
            high = mid - 1;
          else
            return mid;
  }
  return NOT_FOUND;
}
```

- $n/2^{(k-1)} = 1$
- $n = 2^{(k-1)}$
- $\log_2(n) = k - 1$
- $\log_2(n) + 1 = k$
- Funkcija kompleksnosti $f(n) = \log(n)$ iteracij x 1 primerjanje/zanko = $\Theta(\log(n))$

Rekurzija

```
long factorial( int n )
{
   if( n <= 1 )
       return 1;
   else
       return n * factorial( n - 1 );
}
```

To je v resnici navadna zanka, zakrinkana v rekurzijo
kompleksnost = *O(n)*

```
long fib( int n )
{
   if ( n <= 1)
       return 1;
   else
       return fib( n – 1 ) + fib( n – 2 );
}
```

Fibonaccijevo zaporedje:
kompleksnost= $\mathbf{O}(\ (3/2)^N\ )$
torej eksponencialno !!

kompleksnost iskanja maksimuma?

```
double M=x[0];
for i=1 to n-1 do
    if (x[i] > M)
            M=x[i];
    endif
endfor
return M;
```

- $T(n) = a+(n-1)(b+a) = O(n)$
- Pri tem je “a” čas enega prirejanja in “b” je čas ene primerjave
- Tako “a” kot “b” sta konstanti, odvisni od aparaturne opreme
- Opazimo, da nam veliki O prihrani
  - Relativno nepomembne aritmetične podrobnosti
  - Odvisnost od aparaturne opreme

Euklidov algoritem

- Poišči največji skupni delitelj med m in n
  - ob predpostavki $m \geq n$
- Kompleksnost = $O(log(N))$

```
long gcd( long m, long n )
{
    while( n != 0 )
    {
        long rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

Potenciranje

- Izračunaj x^n
- Primeri:
  - $x^{11} = x^5 * x^5 * x$
  - $x^5 = x^2 * x^2 * x$
  - $x^2 = x * x$
- Kompleksnost= $O(\log N)$
- Zakaj tega nismo računali z rekurzijo na naslednji način?
  - **`pow(x,n/2)*pow(x,n/2)*x`**

```
long pow( long x, int n )
{
    if( n == 0 )
        return 1;
    if( n == 1 )
        return x;
    if( isEven( n ) )
        return pow( x * x, n / 2 );
    else
        return pow( x * x, n / 2 ) * x;
}
```

Notacija "Veliki O" v praksi

- Pri računanju kompleksnosti,
  - f(n) je dejanska funkcija
  - g(n) je poenostavljena verzija funkcije
- Ker pogosto gledamo časovno kompleksnost f(n), uporabljamo namesto f(n) zapis T(n)

Načini poenostavljanja

- Če je T(n) vsota konstantnega števila izrazov, izpustimo vse izraze razen najbolj dominantnega (največjega)
- Izpustimo vse množilnike tega izraza
- Kar ostane, je poenostavljena g(n).
- Primeri:

$a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n + a_0 = O(n^m)$.

$n^2 - n + \log n = O(n^2)$

Značilnosti notacije veliki O

- Notacija "veliki O" je mehanizem poenostavitve ocene časa oziroma pomnilniškega prostora.
- Izgubili smo na natančnosti, pridobili na enostavnosti ocene
- Obdržali smo dovolj informacije o občutku za hitrost (ceno) algoritma in za primerjavo konkurenčnih algoritmov.

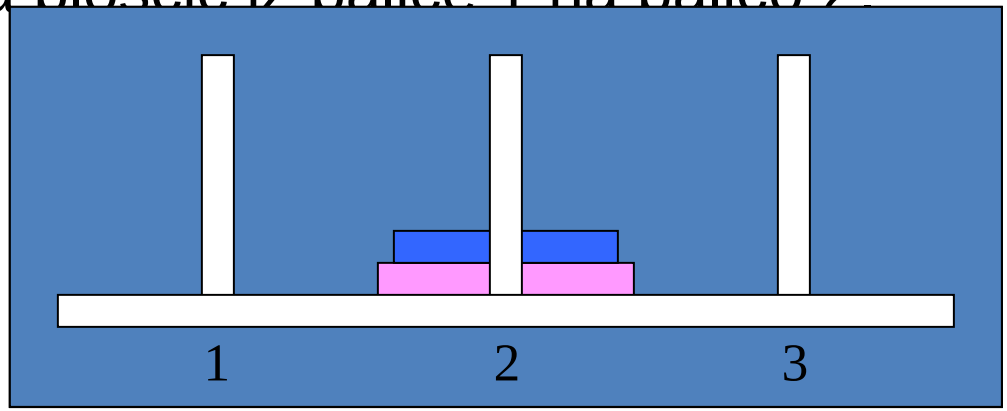
Primeri formul

- $1+2+3+\ldots+n= n(n+1)/2 = O(n^2)$.
- $1^2+2^2+3^2+\ldots+n^2= n(n+1)(2n+1)/6 = O(n^3)$
- $1+x+x^2+x^3+\ldots+x^n=(x^{n+1} – 1)/(x-1) = O(x^n)$.

Primer: Hanojski stolpiči*(1)*

- Na podstavku imamo tri navpične paličice.
- Na palici 1 imamo stolpič iz več ploščic različne velikosti. Največja ploščica je na dnu, najmanjša na vrhu stolpiča.
- Naenkrat lahko premaknemo eno ploščico iz katerekoli palice na katerokoli palico. Nikdar ne smemo postaviti večje ploščice na manjšo.
- Problem: premik stolpiča ploščic iz palice 1 na palico 2.

Animacija (z dvema ploščicama):



Primer: Hanojski stolpiči(2)

Algoritem Hanojskih stolpičev:

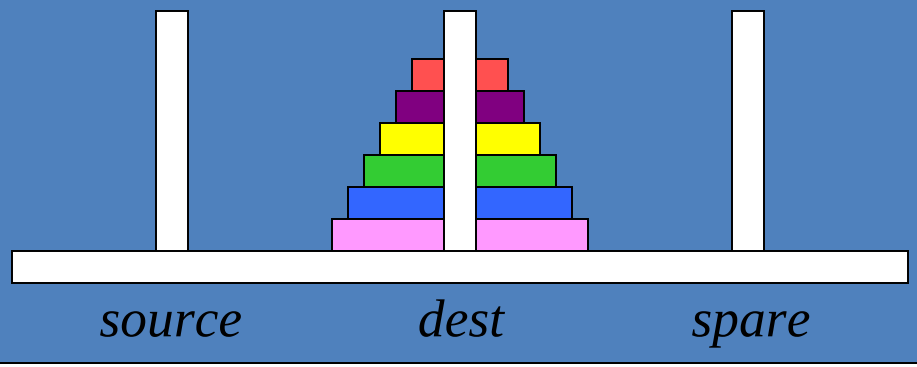
To move a tower of *n* disks from pole *source* to pole *dest*:

1. If $n = 1$:
1.1. Move a single disk from *source* to *dest*.
2. If $n > 1$:
2.1. Let *spare* be the remaining pole, other than *source* and *dest*.
2.2. Move a tower of (n–1) disks from *source* to *spare*.
2.3. Move a single disk from *source* to *dest*.
2.4. Move a tower of (n–1) disks from *spare* to *dest*.
3. Terminate.

Primer: Hanojski stolpiči(3)

Animacija (s 6 ploščicami):

1. If $n = 1$:
   1.1. Move a single disk from *source* to *dest*.
2. If $n > 1$:
   2.1. Let *spare* be the remaining pole, other than *source* and *dest*.
   2.2. Move a tower of (n–1) disks from *source* to *spare*.
   2.3. Move a single disk from *source* to *dest*.
   2.4. Move a tower of (n–1) disks from *spare* to *dest*.
3. Terminate.



Primer: Hanojski stolpiči*(4)*

Analiza (štetje premikov):

Naj bo *moves*(*n)* število premikov, potrebnih za premik stolpiča z *n ploščicami*. Tedaj:

$$moves(n) = 1 \quad \text{if } n = 1$$

$$moves(n) = 1 + 2\, moves(n-1) \quad \text{if } n > 1$$

Rešitev:

$$moves(n) = 2^n - 1$$

Časovna kompleksnost je $O(2^n)$.

WEB

DEMO