# Algoritmy a programování

## Složitost algoritmů

**Vojtěch Vonásek**

Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

# Rychlost programů a algoritmů

- Určení náročnosti výpočtů
  - pro výběr vhodné implementace, knihovny, algoritmu
  - pro výběr vhodného HW
  - v optimalizaci programů
- Časová a paměťová náročnost (složitost)
- Náročnost **programu** (implementace) vs. náročnost **algoritmu**
- Analýza programu (implementace)
  - ovlivněna jazykem, HW, zátěží OS, ...
  - teoretická analýza + empirické měření
  - typicky měřeno jako “čas výpočtu”
- Analýza algoritmu
  - teoretická analýza “big-O” notace
  - nezávisí na použitém jazyce
  - typicky analýza počtu “zajímavých” operací

# Rychlost programu (implementace)

- Rychlost programu je dále ovlivněna programovacím jazykem, způsobem kompilace . . .
- A samozřejmě typem algoritmu, který program realizuje
- Reálný čas: doba vykonání programu/jeho části
  - Rozdíl mezi časem spuštění a ukončení testovaného programu
  - Závisí na rychlosti PC, operačním systému, velikosti RAM, cache. . .
  - Závisí na aktuálním zatížení PC, případně i na vytížení periferií
- CPU čas: čas strávený na CPU
  - Nižší než reálný čas
  - Závislost na PC, operačním systému, . . .
  - Nezávisí na aktuálním vytížení
  - Složitější měření

# Empirické měření rychlosti programu

- Ukážeme si špatné a dobré měření
- Máme dvě verze (liší se použitím knihovny numpy)
- program1.py

```python
import insertionSort as IS
import random, time

a = [ random.random() for _ in range(1000) ]
IS.insertionSort(a)
```

- program2.py

```python
import insertionSort as IS
import random, time
import numpy
a = [ random.random() for _ in range(1000) ]
IS.insertionSort(a)
```

# Empirické měření: nevhodně

- Běh programu změříme na příkazové řádce jako **reálný čas** běhu

```
> time python3 program1.py
```

```
real    0m0,076s
user    0m0,067s
sys     0m0,009s
```

- Pro další rozhodování budeme používat čas 'real'
- Stejný způsob měření můžeme dosáhnout i v Pythonu

```python
import time
a = [-i for i in range(10000) ]
t1 = time.time()
b = a.sort()  #merena operace
t2 = time.time()
print(t2-t1)  #real-time
```

# Empirické měření: nevhodně

- Výsledky měření:
  - `> time python3 program1.py`: real je 1.23s
  - `> time python3 program2.py`: real je 1.21s
- Můžeme provést závěr, že `program2.py` je rychlejší, než `program1.py` ?
- Uveďte argumenty pro a proti!

# Empirické měření: nevhodně

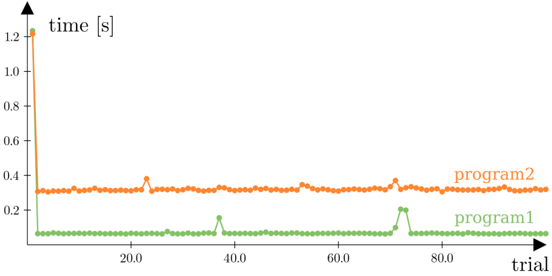


- Změřením obou programů (na stejném PC)
  - `> time python3 program1.py`: real je 1.23s
  - `> time python3 program2.py`: real je 1.21s
- Můžeme provést závěr, že `program2.py` je rychlejší, než `program1.py` ?

## Proč je tento způsob řešení nevhodný

- Jednobodové měření — je zatíženo chybou (aktuální zatížení procesoru, disků . . . )
- Nutnost opakování měření a statistického porovnání

- Více měření (čím více, tím lépe), zde použijeme 100 měření
- `program1.py`: $\bar{t} = 0.0808$, $\sigma = 0.11$
- `program2.py`: $\bar{t} = 0.3283$, $\sigma = 0.089$
- V průměru je `program1.py` $\sim 4\times$ rychlejší než `program2.py`



# Empirické měření: správně

- Důvodem pomalého běhu `program2.py` je importování knihovny numpy
- Knihovna numpy se ale nepoužívá
- `program1.py`

```python
import insertionSort as IS
import random, time

a = [ random.random() for _ in range(1000) ]
IS.insertionSort(a)
```

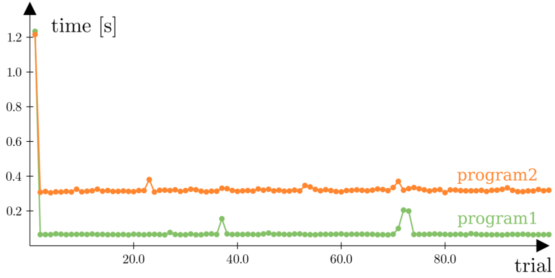
- `program2.py`

```python
import insertionSort as IS
import random, time
import numpy
a = [ random.random() for _ in range(1000) ]
IS.insertionSort(a)
```

- Dobrá praxe: importujeme pouze knihovny, které používáme

# Empirické měření: závěr

- Známe (průměrné) chování programu na vstupu velikosti $n = 1000$
- Průměr ze 100 měření
- Měření je validní pro jedno PC, nelze ho zobecnit
- Neznáme chování pro jinak velké vstupy a pro jiné PC/OS
- Empirické měření zkoumá vlastnosti programu, nikoliv algoritmů



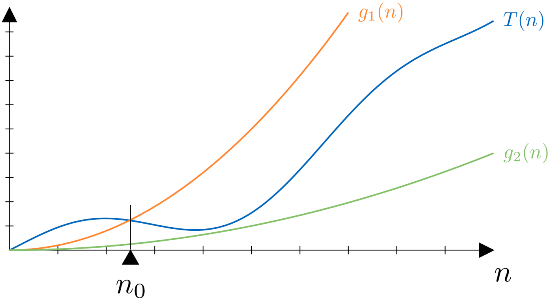
# Časová složitost (Time complexity)



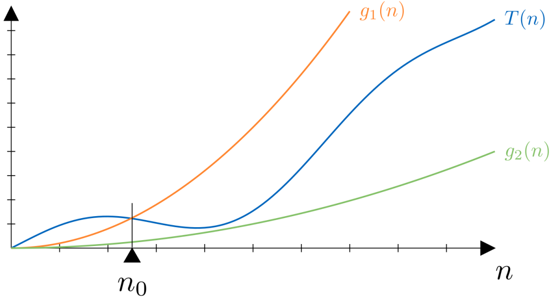
- Způsob analýzy algoritmů
- Najdeme funkci $T(n)$, která popisuje časovou (nebo paměťovou) náročnost algoritmu
- Velikost vstupu je $n$
  - např. velikost vstupních polí pro seřazení
  - celkový počet bitů na vstupu
  - počet vrcholů/hran v grafech
  - velikost matic
  - atd. podle konkrétního algoritmu
- Funkce $T(n)$ nejčastěji popisuje
  - kolik "vybraných operací" se provede (časová náročnost)
  - kolik paměti (v RAM/HDD) bude třeba k vyřešení úlohy

- Chování algoritmu $T(n)$ nás zajímá pouze pro velká $n \to \infty$
- Multiplikativní a aditivní konstanty zanedbáme
- Pomaleji rostoucí části $T(n)$ zanedbáme

- Nechť $f(n) : \mathbb{N} \to \mathbb{R}_0^+$
  $f(n)$ je $\mathcal{O}(g(n))$ pokud existují konstanty $n_0 > 0$ *a* $c > 0$ takové, že $f(n) \leq cg(n)$ pro všechna $n \geq n_0$




- $g_2(n)$ není horní odhad $T(n)$
- $g_1(n)$ je horní odhad $T(n)$ pro $n \geq n_0$. $\Rightarrow$ $T(n)$ je $\mathcal{O}(g_1(n))$

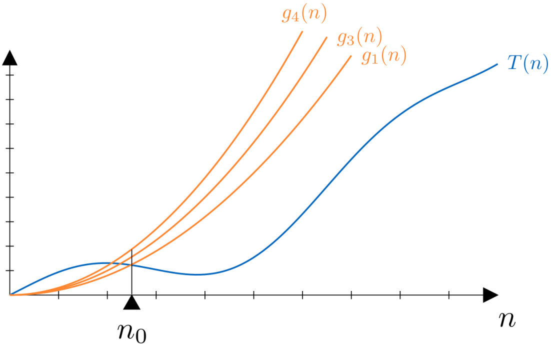



## Příklad

- $T(n) = 2n^2 + 3 + 4n$
- $g_1(n) = n^2$ je horním odhadem $T(n)$ pro $n > n_0$
- Říkáme, že $T(n) = 2n^2 + 3 + 4n$ je $\mathcal{O}(n^2)$

# $\mathcal{O}()$: big-O notace

- $T(n) = 2n^2 + 3 + 4n$
- $g_1(n) = n^2$ je horním odhadem $T(n)$
- Říkáme, že $T(n) = 2n^2 + 3 + 4n$ je $\mathcal{O}(n^2)$




- Odhadem $T(n)$ jsou i další funkce, např. $\mathcal{O}(n^3)$ nebo $\mathcal{O}(n^{100})$
- **Uvádíme ten nejlepší známý odhad**

$$f(n) \overset{n\to\infty}{\sim} g(n) \text{ pokud } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

- Místo $f$ je $\mathcal{O}(n^3)$ píšeme $f \sim O(n^3)$

| $f(n)$ | $\sim g(n)$ | $\mathcal{O}(n)$ |
|---|---|---|
| $4n^2 + n$ | $4n^2$ | $\mathcal{O}(n^2)$ |
| $4n^2 + n + 123456$ | $4n^2$ | $\mathcal{O}(n^2)$ |
| $4n^2 + 10^6 n$ | $4n^2$ | $\mathcal{O}(n^2)$ |
| $4n^2 + 0.01n^3$ | $0.01n^3$ | $\mathcal{O}(n^3)$ |
| $n(n-1)(n-2)$ | $n^3$ | $\mathcal{O}(n^3)$ |
| $4n^3 + 100n^2 + 1000n + 5000$ | $4n^3$ | $\mathcal{O}(n^3)$ |

- Pro polynomy uvažujeme jen člen nejvyššího řádu

$4n^3 + 100n^2 + 1000n + 5000$ je $\mathcal{O}(n^3)$

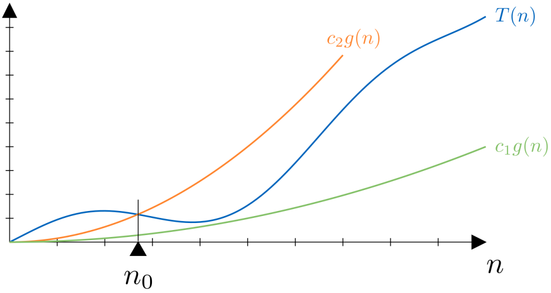
- Pokud $n \geq 100$, pak

$\Rightarrow$ $n^3 \geq 100n^2$,

$\Rightarrow$ $n^3 \geq 1000n$,

$\Rightarrow$ $n^3 \geq 5000$

- Nechť $f(n) : \mathbb{N} \to \mathbb{R}_0^+$
  $f(n)$ je $\Theta(g(n))$ pokud existují konstanty $c_1 > 0, c_2 > 0$ a $n_o > 0$ takové, že $c_1 g(n) \leq f(n) \leq c_2 g(n)$ pro všechna $n \geq n_0$




- $4n^3 + 100n^2$ je $\mathcal{O}(n^3)$ ale i $\mathcal{O}(n^4)$, $\mathcal{O}(n^5)$, ...
- $4n^3 + 100n^2$ **je pouze** $\Theta(n^3)$, nikoliv $\Theta(n^4)$, $\Theta(n^5)$ ...

# Asymptotická notace $\Theta()$

- $\mathcal{O}()$ notace

  $f(n)$ je $\mathcal{O}(g(n))$ pokud existují konstanty $n_0 > 0$ $a$ $c > 0$ takové, že $f(n) \leq cg(n)$ pro všechna $n \geq n_0$

- $\Theta()$ notace

  $f(n)$ je $\Theta(g(n))$ pokud existují konstanty $c_1 > 0, c_2 > 0$ a $n_o > 0$ takové, že $c_1 g(n) \leq f(n) \leq c_2 g(n)$ pro všechna $n \geq n_0$

- $\Theta()$ přesnější, ale je těžší ji najít
- V praxi se používá převážně $\mathcal{O}()$

# Druhy odhadů

- Průměrná složitost (Average complexity)
  - závisí na datech (co jsou “typická” data?)
  - složitá teoretická analýza
  - lze odhadnout experimentálně (vyžaduje znalost ‘typických’ dat)
- Nejhorší složitost (worst-case complexity)
  - lze odhadnout teoreticky
  - experimentálně nelze
- Složitost algoritmů může být závislá na více parametrech

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

## Hledání prvků v poli

```
def findItem(x,query): #x is list
    for item in x:
        if item == query:
            return True
    return False

a = [0,1,0,2]
print( findItem(a, 0 ) )
print( findItem(a, "0") )
```

- Složitost $\mathcal{O}(n)$ ($n$ je velikost vstupního pole)

# Příklad určení složitosti

## Hledání prvku půlením intervalu

```
def binarySearch(x, query):
    L = 0
    R = len(x)-1
    while L <= R:
        M = (L+R) // 2
        if x[M] == query:
            return M
        if x[M] > query:
            R = M-1
        else:
            L = M+1
    return -1
```

- V každé iteraci zmenšujeme velikost prohledávaného intervalu na polovinu
- Složitost $\mathcal{O}(\log n)$

### BubbleSort

```
def bubbleSort(x): #x is list
    for r in range(len(x)-1,0,-1): #r is used
        change = False
        for j in range(r):  #j = 0..r-1
            if x[j] > x[j+1]:
                x[j],x[j+1] = x[j+1], x[j]
                change = True
        if not change:
            break
a = [ 10,-10,0,1,-3,4,4]
print(a)
bubbleSort(a)
print(a)
```

- Vnější smyčka proběhne *n*-krát, vnitřní smyčka proběhe nejvýše *n*-krát
- Složitost $\mathcal{O}(n^2)$

# Příklad určení složitosti

## SelectionSort

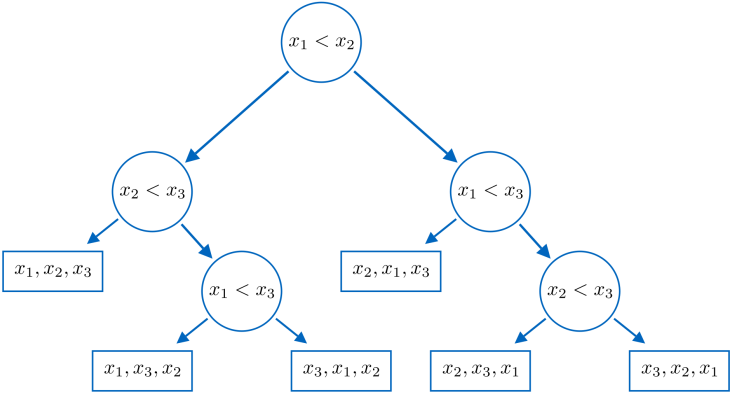
```
def selectionSort(x): #x is list
    for r in range(len(x)-1):
        minidx = r
        for j in range(r+1,len(x)):
            if x[j] < x[minidx]:
                minidx = j
        x[minidx], x[r] = x[r], x[minidx]

a = [7,42,-3,0,5,1,1]
selectionSort(a)
print(a)
```

- Složitost $\mathcal{O}(n^2)$

# Složitost řazení

- Řadící algoritmy založené na porovnání (operátor $<$)
- Kolik těchto porovnání je třeba, aby se seřadilo pole $n$ položek?
- Graf porovnání pro tři prvky $x_1, x_2, x_3$



# Složitost řazení

- Počet listů je $2^h$, kde $h$ je hloubka rozhodovacího stromu
- Strom musí být schopen pro jakoukoliv vstupní permutaci udělat rozhodnutí $\rightarrow$ obsahovat list

$$
\begin{aligned}
n! &\leq 2^h \\
h &\geq \log_2 n!
\end{aligned}
$$

- Stirlingův vzorec

$$
\log(n!) = n \log n - n + \mathcal{O}(\log n)
$$

- Nejvyšší člen je zde $n \log n$
- **Porovnávací řadící algoritmus potřebuje nejméně $n \log n$ porovnání k seřazení pole $n$ položek**
- Řazení (porovnávací) má složitost $\mathcal{O}(n \log n)$

# Příklad určení složitosti

```python
N = 100
i = N
b = 1
while i > 0:
    b*=i
    print(b,i)
    i //= 2
```

- Půlíme interval $N$, $N/2$, $N/4$ atd..
- Složitost $\mathcal{O}(\log n)$

# Příklad určení složitosti



```python
N = 100
i = N
b = 1
while i > N//2:
    b*=i
    print(b,i)
    i //= 2
```

- Složitost ?

# Příklad určení složitosti

```python
def f(n):
    i = n
    a = 0
    while i > 0:
        j = 0
        for k in range(n):
            j += k
            a += 1
            if j >= n:
                break
        i = i // 2
    return a

for n in range(10000):
    print(n, f(n))
```

- Složitost ?

# Příklad určení složitosti

```
n = 100
a = 1
for i in range(n)):
    for j in range(i,n)):
        a += 1
```

- Složitost ?

# Příklad určení složitosti

```
n = 100
for a in range(1,n):
    j = 1
    while j < n:
        j += a
```

- Složitost ?

# Násobení matic

- Máme $n \times n$ matice **A**, **B** a počítáme $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$
- Například pro matice $2 \times 2$

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} b_{11} + a_{12} b_{21} & a_{11} b_{12} + a_{12} b_{22} \\ a_{21} b_{11} + a_{22} b_{21} & a_{21} b_{12} + a_{22} b_{22} \end{bmatrix}$$

- kde $a_{ij}, b_{ij}, cij \in \mathbb{R}$
- Standardní násobení matic vyžaduje 8 násobení a 4 sčítání
- Násobení velkých čtvercových matic — Strassenův algoritmus

# Násobení matic

- Máme $n \times n$ matice $\mathbf{A}$, $\mathbf{B}$ a počítáme $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$
- Naivní algoritmus pro matice $n \times n$

```
#a,b,c are square matrices n x n
#c is assumed to be n x n matrix of zeros
for i in range(n):
    for j in range(n):
        for k in range(n):
            c[i][j] += a[i][k] * b[k][j]
#result is in c
```

- Složitost $\mathcal{O}(n^3)$

# Blokové násobení čtvercových matic

- Nechť máme matici, jejíž počet prvků je mocnina 2: $\mathbf{A}, \mathbf{B}, \mathbf{C}$ jsou matice $2^k \times 2^k = n \times n$, kde $n = 2^k$
- Tyto matice lze reprezentovat po blocích
- $\mathbf{A}_{ij}$ jsou čtvercové matice

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ \vdots & & & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & & a_{nn} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}$$

- Standardní násobení velkých matic (rekurzivně po blocích)
- 8 násobení čtvercových matic a 4 sčítání čtvercových matic

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix}$$

# Strassenův algoritmus

- Nechť máme matici, jejíž počet prvků je mocnina 2: $\mathbf{A}, \mathbf{B}, \mathbf{C}$ jsou matice $2^k \times 2^k = n \times n$, kde $n = 2^k$

$$
\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22});\\
\mathbf{M}_2 &= (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11};\\
\mathbf{M}_3 &= \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22});\\
\mathbf{M}_4 &= \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11});\\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22};\\
\mathbf{M}_6 &= (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12});\\
\mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22}),
\end{aligned}
$$

$$
\mathbf{C} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{bmatrix}
$$

$$
\begin{aligned}
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7\\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5\\
\mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4\\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}
$$

- Strassenův algoritmus potřebuje 7 násobení
- Matice se rekurzivně násobí (blokově) až dojde na násobení matic $2 \times 2$
- Potřebuje ale více operací sčítání/odčítání

# Příklad určení složitosti



## Detekce kolizí mezi dvěma polygony

- Polygon je sekvence vrcholů
- Vstup: $P = ((x_1, y_1), \ldots (x_n, y_n))$ a $Q = ((x_1, y_1), \ldots (x_m, y_m))$
- Detekce kolizí (naivně): pro každou úsečku z $P$ spočítat, jestli má průnik s nějakou úsečkou z $Q$
- Nejsložitější operace je zde **výpočet průniku**

```python
def isCollision(p,q): """ p,q are [[x1,y1], ... [x_n,y_n] ] """
    for i in range(len(p) - 1): #(n-1)-times
        for j in range(len(q) - 1): #(m-1)-times
            r = isCrossing(p[i], p[i+1], q[j], q[j+1])
            if r:
                return True
    return False
```

- Vnější smyčka proběhne (nejvýše) $n - 1$ krát, vnitří smyčka proběhne (nejvýše) $m - 1$ krát.
- Počet volání `isCrossing()` je $(n - 1)(m - 1)$
- Složitost algoritmu je $\mathcal{O}(mn)$
- Kdy nastane případ ‘nejvýše’ ?

# Srovnání složitostí



| | | |
|---|---|---|
| konstatní | $\mathcal{O}(1)$ | nejrychlejší |
| logaritmická | $\mathcal{O}(\log n)$ | velmi rychlá (např. binární půlení) |
| lineární | $\mathcal{O}(n)$ | rychlá i pro velká data |
| | $\mathcal{O}(n \log n)$ | srovnatelná s $\mathcal{O}(n)$ |
| kvadratická | $\mathcal{O}(n^2)$ | pomalejší než lineární, rychle roste s $n$ |
| kubická | $\mathcal{O}(n^3)$ | pomalá |
| polynomiální | $\mathcal{O}(n^k)$ | pomalá |
| exponenciální | $\mathcal{O}(b^n)$ | velmi pomalá |
| | $\mathcal{O}(n!)$ | velmi pomalá |

- Pro velká data se snažíme používat algoritmy s nejmenší složitostí
- Pro malá data lze (v oprávněných případech) použít algoritmus s mírně vyšší složitostí