

# Efektivita algoritmu (složitost algoritmu)

## - časová

počet vykonaných operací (*kterých?*)

→ rychlost výpočtu programu

## - prostorová (paměťová)

velikost datových struktur využívaných algoritmem

→ paměť potřebná na uložení dat při výpočtu programu

Kritéria pro hodnocení kvality algoritmu a programu  
(příp. praktické použitelnosti programu).

Obě kritéria mohou mířit proti sobě (nelze najednou optimalizovat paměť i čas), musíme zvolit, čemu dáme přednost (dnes obvykle čas).

**„výměna času za paměť“**

*Jaké operace započítat:*

- elementární, vyžadující konstantní čas
- aritmetické, logické, přiřazení
- typické pro řešený problém (převažující)

*Kterou paměť započítat:*

- do prostorové složitosti nepočítáme paměť potřebnou na uložení vstupních dat, pokud z ní data pouze čteme (neměníme její obsah)

Efektivita algoritmu = funkce vyjadřující počet vykonaných operací (resp. velikost pracovní paměti) v závislosti na velikosti vstupních dat.

Jako velikost vstupních dat obvykle stačí uvažovat např. počet zpracovaných čísel, nikoliv konkrétní hodnoty (jedná-li se o hodnoty „standardní“ velikosti).

Obecně by se musela uvažovat délka zápisu vstupních dat v bitech.

Funkce časové a prostorové složitosti jsou většinou **rostoucí** (nad většími daty bývá výpočet delší).

Přesné vyjádření funkce časové složitosti (např.  $3.N^2 + 2.N - 4$ ) je jednak obtížné, jednak většinou zbytečné. Podstatná je řádová rychlost růstu této funkce pro rostoucí  $N$ .

Zanedbáme pomaleji rostoucí členy a konstanty

→ **asymptotická časová složitost** - zapisujeme např.  $O(N^2)$ .

**Symbol „velké O“**

$f, g: \mathbf{N} \rightarrow \mathbf{R}$

$f \in O(g) \Leftrightarrow \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) \leq c.g(n)$

tzn. funkce  $f$  se dá **shora** odhadnout funkcí  $g$

(až na multiplikativní konstantu a pro dostatečně velká  $n$ )

Opačný odhad **zdola**:  $f \in \Omega(g) \quad 0 \leq c.g(n) \leq f(n)$

**Přesný** (těsný) odhad:  $f \in \Theta(g) \Leftrightarrow f \in O(g) \ \& \ f \in \Omega(g)$

### *Poznámka:*

Funkce  $3.N^2 + 2.N - 4$  patří nejen do třídy  $O(N^2)$ , ale podle definice také třeba do třídy  $O(N^5)$ . Odhad složitosti  $O(N^5)$  je zde sice správný, ale zbytečně hrubý a nepřesný, vždy se snažíme o co nejlepší (nejnižší) horní odhad složitosti.

Proto je pro nás cennější (ale také obtížnější) odvodit těsný odhad asymptotické časové složitosti algoritmu  $\Theta(N^2)$  než jenom horní odhad  $O(N^2)$ .

# Asymptotická časová složitost

Typické třídy asymptotické časové složitosti algoritmů:

$O(1)$ ,  $O(\log N)$ ,  $O(N)$ ,  $O(N \cdot \log N)$ ,  $O(N^2)$ ,  $O(N^3)$ , ...,  $O(2^N)$ ,  $O(N!)$

Hledáme algoritmus s co nejmenší asymptotickou časovou složitostí, tedy co nejrychlejší pro velká  $N$ . Pro malá  $N$  může být třeba i horší než nějaký jiný algoritmus, ale pro malá  $N$  to nevadí, doba výpočtu je vždy dostatečně krátká.

- polynomiálně omezený čas – obvykle zvládnutelné i pro velká  $N$
- exponenciální čas – pro větší  $N$  časově nezvládnutelné,  
použitelné jen v případě, že vstupní data budou vždy malá

# Exponenciální časová složitost

*Příklad:* Ke zpracování vstupních dat velikosti  $N$  algoritmus vykoná  $2^N$  operací. Uvažujme rychlost počítače  $10^9$  operací za sekundu (řádově GHz – dnešní PC).

$N$	dobu výpočtu
20	1 ms
30	1 s
40	17 min
50	11 dní
60	31 let
70	$3 \cdot 10^4$ let
80	$3 \cdot 10^7$ let
90	$3 \cdot 10^{10}$ let
100	$3 \cdot 10^{13}$ let

Pro vyšší hodnoty  $N$  (cca 50) je algoritmus prakticky nepoužitelný.  
***Nepomůže nám ani rychlejší počítač!***

# Složitost algoritmu v různých případech

Pro každá vstupní data velikosti  $N$  nemusí trvat výpočet stejně dlouho, rozdíl může být jen v konstantě, nebo i v asymptotické složitosti.

## Časová složitost algoritmu v nejhorším případě

= maximální počet operací vykonaných algoritmem pro nějaká data velikosti  $N$

→ nejčastěji používaná pro hodnocení algoritmů

## Časová složitost algoritmu v nejlepším případě

= minimální počet operací vykonaných algoritmem pro nějaká data velikosti  $N$

→ prakticky se nepoužívá

## Časová složitost algoritmu v průměrném případě

= průměrný (očekávaný) počet operací vykonaných algoritmem pro data velikosti  $N$  (průměr pro všechna možná vstupní data velikosti  $N$ )

→ dobře charakterizuje kvalitu algoritmu, ale je obtížné odvodit ji



# Složitost problému

- složitost nejlepšího algoritmu (z hlediska časové složitosti), kterým lze řešit daný problém

Nelze odvodit ze složitosti nějakého konkrétního algoritmu ani třeba všech běžně známých algoritmů, charakterizuje problém obecně (jaký nejlepší algoritmus řešící problém může v principu existovat)

→ odvození bývá často obtížné, pro řadu problémů neznáme

Dokázat, že problém má složitost  $\Theta(N^2)$  pro nejhorší případ, znamená ukázat, že

- existuje algoritmus, který vyřeší problém s asymptotickou časovou složitostí  $O(N^2)$  v nejhorším případě

*to bývá snadné – předvedeme jeden zvolený konkrétní algoritmus*

- každý algoritmus, který řeší uvedený problém, pracuje v čase  $\Omega(N^2)$  v nejhorším případě

*to bývá obtížné – nemůžeme ukázat na jednom konkrétním algoritmu*

*Příklady:*

1. Nalézt maximum z daných  $N$  čísel  $\rightarrow$  časová složitost problému  $\Theta(N)$ 
  - existuje algoritmus s časovou složitostí  $O(N)$  ... triviální
  - nemůže existovat algoritmus s lepší časovou složitostí
    - ... maximem může být kterékoliv z  $N$  čísel, takže na každé se musí algoritmus podívat
  
2. Seřadit daných  $N$  čísel podle velikosti (problém třídění)
  - $\rightarrow$  časová složitost problému  $\Theta(N \cdot \log N)$
  - existuje algoritmus s časovou složitostí  $O(N \cdot \log N)$  ... např. heapsort
  - nemůže existovat algoritmus s lepší časovou složitostí
    - ... oboje si ukážeme později

# Rozklad čísla na cifry

*Úloha:*

Spočítejte ciferný součet zadaného kladného celého čísla.

*Postup řešení:*

zbytek čísla po dělení 10 → poslední cifra

celočíslné vydělení čísla 10 → odstranění poslední cifry

načti vstupní hodnotu do proměnné X

$Y = 0$

dokud X není rovno 0

$Y = Y + X \bmod 10$

$X = X / 10$  (celočíslné dělení)

vypiš jako výsledek hodnotu proměnné Y

*Totéž jako funkce v Pythonu:*

```
def cifsoucet(x):  
    """ciferný součet kladného celého čísla"""  
    y = 0  
    while x != 0:  
        y += x % 10  
        x //= 10  
    return y
```

# Test prvočíslnosti

*Úloha:* Určete, zda je dané číslo  $N$  prvočíslem.

*Postupy řešení:*

1. zkusit všechny dělitele od 2 do  $N-1$   
→ časová složitost  **$O(N)$**  – cca  $N$  testů
2. stačí zkoušet všechny dělitele od 2 do  $N/2$  (větší dělitel neexistuje)  
→ časová složitost opět  **$O(N)$**  – cca  $N/2$  testů, tedy trochu lepší
3. stačí zkoušet všechny dělitele od 2 do  $\sqrt{N}$   
(má-li  $N$  vlastního dělitele, musí mít i dělitele z tohoto intervalu)  
→ časová složitost  **$O(\sqrt{N})$**  – cca  $\sqrt{N}$  testů, asymptoticky lepší

4. stačí zkoušet dělitele od 2 do  $\sqrt{N}$ , do nalezení prvního  
→ asymptotická časová složitost opět  $O(\sqrt{N})$ ,  
většinou se ale vykoná o dost méně testů
- v nejhorším případě se vykoná plných  $\sqrt{N}$  testů  
→ časová složitost v nejhorším případě  $O(\sqrt{N})$
  - v nejlepším případě stačí pouze jeden test
5. stačí zkusit číslo 2 (jediné sudé prvočíslo)  
a pak jenom liché dělitele lichých čísel  
→ další úspora práce

*Poznámka:* Popsaný algoritmus má ve skutečnosti exponenciální časovou složitost vzhledem k délce vstupu, pokud za délku vstupu považujeme  $\lceil \log_2 N \rceil + 1$ , což je délka bitového zápisu čísla  $N$ .

```
from math import sqrt

def prvocislo(n):
    """
        test prvočíselnosti, předpoklad  $n > 1$ 
    """
    # stačí prověřit dělitele do odmocniny z n
    for d in range(2, int(sqrt(n))+1):
        if n % d == 0:           # pokud  $d \mid n$ 
            return False       # n není prvočíslo
    return True
```



```

def prvocislo(n):
    """
        test prvočíselnosti, předpoklad n > 1
    """
    if n % 2 == 0:
        return n == 2          # jediné sudé prvočíslo: 2
    d = 3
    while d * d <= n:          # místo volání sqrt(n)
        if n % d == 0:
            return False
        d += 2
    return True

```

# Eratosthenovo síto

*řecký matematik Eratosthenes z Kyrény, cca 200 př.n.l.*

*Úloha:* Určete všechna prvočísla od 2 do  $N$ .

*Princip řešení:*

- v řadě čísel od 2 do  $N$  postupně vyškrťujeme všechny násobky jednotlivých prvočísel
- co nakonec nebude vyškrtnuto, je prvočíslo

*Programová realizace:*

- síto v programu reprezentujeme polem prvků typu boolean,
- index pole určuje číslo od 2 do  $N$ ,
- hodnota prvku síta říká, zda je prvočíslem.

```

n = int(input("Horní mez prvočísel: "))
sito = [False, False] + [True] * (n-1)

i = 2
while i <= n:
    if sito[i]:
        # číslo "i" je prvočíslo
        j = 2 * i
        # první násobek čísla "i"
        while j <= n:
            sito[j] = False
            # násobek čísla "i" není prvočíslo
            j = j + i
            # další násobek
        i = i + 1
        # další zkoumané číslo

for i in range(n+1):
    if sito[i]:
        print(i)

```

```

def eratosth(n):
    """Eratosthenovo síto"""

    sito = [False, False] + [True] * (n-1)
    prvocisla = []

    i = 2
    while i <= n:
        if sito[i]:
            prvocisla.append(i)
            j = i * i # stačí začít s násobky od kvadrátu "i"
            while j <= n:
                sito[j] = False
                j = j + i
            i = i + 1

    return prvocisla

```

*Časová složitost:*

$$O(N/2 + N/3 + N/5 + \dots) = O(N \cdot \log(\log N))$$

# Dlouhá čísla

Chceme počítat například s kladnými celými čísly s desítkami nebo stovkami cifer (podobně pro čísla se znaménkem, čísla desetinná apod.).

Python to umí sám, většina programovacích jazyků ale nikoliv.

## **Reprezentace:**

- číslo uložíme po cifrách – seznam cifer (pole cifer)
- pro snadnější práci použijeme seznam čísel, nikoliv znaků
- nutno zvolit pořadí cifer odpředu nebo odzadu  
(možné je oboje, ale potom důsledně dodržovat v celém programu!)

## **Operace:**

- po cifrách – jako sčítání, odčítání, násobení či dělení víceciferných čísel na základní škole
- počítání modulo 10, přenosy do vyšších řádů

### ***Modifikace:***

číslo uložíme po skupinách cifer do prvků typu int,  
→ úspora paměti, rychlejší výpočet  
(počítání modulo 100, nebo 1000, atd.)

### ***Desetinné číslo:***

stačí doplnit evidenci polohy desetinné čárky

- speciální hodnota uložená v jednom prvku seznamu (méně šikovné)
- proměnná s indexem nulového řádu
- dvě pole (celá a desetinná část čísla)

***Příklad:***

Součet dvou kladných celých čísel s mnoha ciframi

*Vstup:*  $a, b$  – seznamy cifer sčítaných čísel

$a[0]$  = cifra v řádu jednotek

*Výstup:*  $c$  – výsledný seznam cifer součtu čísel  $a + b$

Nejprve sčítání s přenosem v rozsahu cifer kratšího čísla, dále musíme podobně (s přenosem) ošetřit „přečnávající“ část delšího čísla, nakonec ještě přidat případný poslední nenulový přenos.



```

if len(a) < len(b):
    a, b = b, a                    #číslo "a" je delší

prenos = 0
c = []
for i in range(len(b)):
    x = a[i] + b[i] + prenos
    c.append(x%10)
    prenos = x // 10

for i in range(len(b), len(a)):
    x = a[i] + prenos
    c.append(x%10)
    prenos = x // 10

if prenos > 0:
    c.append(prenos)

```

# Vyhodnocení polynomu v bodě

$$a(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} \dots + a_1 x + a_0$$

$n$  – stupeň polynomu

$a_0, \dots, a_n$  – koeficienty (reálné konstanty)

$x$  – proměnná, za niž dosazujeme různé hodnoty

## ***Přímý výpočet podle uvedeného předpisu***

počet násobení:  $n + (n-1) + (n-2) + \dots + 1 = n \cdot (n+1)/2$

počet sčítání:  $n$

časová složitost algoritmu:  $O(n^2)$

## ***Hornerovo schéma (Horner 1819, ale známo již dříve)***

$$a(x) = (\dots((a_n x + a_{n-1}) \cdot x + a_{n-2}) \cdot x + \dots + a_1) \cdot x + a_0$$

počet násobení:  $n$

počet sčítání:  $n$

časová složitost algoritmu:  $O(n)$

```

def horner(a, x):
    """
    výpočet hodnoty polynomu Hornerovým schématem
    a: seznam s koeficienty polynomu  $a[i] * x^i$ 
    x: bod z definičního oboru
    vrátí: hodnotu polynomu v bodě x
    """
    h = 0
    for i in range(len(a)-1, -1, -1):
        h = h * x + a[i]
    return h

```

*Příklad použití algoritmu:*

vstup čísla po znacích

- co přibližně dělá procedura „read“ v Pascalu, funkce „scanf“ v C

konverze číselného stringu na integer

- co dělá funkce `int` v příkazu `a = int(input())`

# Operace s polynomy

$$a(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} \dots + a_1 x + a_0$$

$$b(x) = b_m x^m + b_{m-1} x^{m-1} + b_{m-2} x^{m-2} \dots + b_1 x + b_0$$

- součet, součin, ...

```
a = [2, -5, 0, 4, 6]      #  $6x^4 + 4x^3 - 5x + 2$   
b = [11, 0, -2]          #  $-2x^2 + 11$ 
```

```
def soucet(a, b):  
    c = []  
    if len(a) < len(b):  
        a, b = b, a  
    for i in range(len(b)):  
        c.append(a[i] + b[i])  
    for i in range(len(b), len(a)):  
        c.append(a[i])  
    while len(c) > 1 and c[-1] == 0:  
        del c[-1]  
    return c
```

```
print(soucet(a, b))
```

```
a = [2, -5, 0, 4, 6]      #  $6x^4 + 4x^3 - 5x + 2$   
b = [11, 0, -2]          #  $-2x^2 + 11$ 
```

```
def soucin(a, b):  
    c = [0] * (len(a) + len(b) - 1)  
    for i in range(len(a)):  
        for j in range(len(b)):  
            c[i+j] += a[i] * b[j]  
    return c
```

```
print(soucin(a, b))
```