

Časová složitost / Time complexity

Jan Kybic

<http://cmp.felk.cvut.cz/~kybic>
kybic@fel.cvut.cz

2016–2018



Složitost algoritmů

Algorithm complexity

- ▶ **Časová** a paměťová složitost
- ▶ Trvání výpočtu v závislosti na vstupních datech
 - ▶ v nejhorším případě, v průměru. . .
- ▶ Který algoritmus je lepší?
- ▶ Jak velká data jsme schopni zpracovat?
- ▶ Empirická vs. teoretická analýza

Empirická časová složitost

Teoretická časová složitost

Empirická časová složitost

- ▶ Změříme dobu běhu pro různá vstupní data
- ▶ Vyhodnocujeme algoritmus + implementace + hardware
- ▶ Kvantitativní výsledky
- ▶ Neposkytuje záruky, obtížná predikce

Příklad: Prvočísla

- ▶ Najdi všechna prvočísla menší než m
- ▶ Metody:
 1. Zkus všechny dělitele do $n - 1$
 2. Zkus všechny dělitele do \sqrt{n}
 3. Eratosthenovo síto
 4. Eratosthenovo síto, vylepšené (\sqrt{n} , jen lichá)

Soubor `porovnaní_prvocislo.py`

Prvočísla — měření času (1)

Implementation : prvocisla_jednoduse

n= 100 CPU time= 0.001s

n= 300 CPU time= 0.004s

n= 1000 CPU time= 0.035s

n= 3000 CPU time= 0.274s

n= 10000 CPU time= 2.716s

n= 30000 CPU time= 21.899s

n=100000 CPU time=218.257s

Prvočísla — měření času (2)

Implementation : prvocisla_odmocnina

n= 100 CPU time= 0.000s

n= 300 CPU time= 0.001s

n= 1000 CPU time= 0.003s

n= 3000 CPU time= 0.012s

n= 10000 CPU time= 0.063s

n= 30000 CPU time= 0.278s

n=100000 CPU time= 1.439s

Prvočísla — měření času (3)

Implementation : prvocisla_eratosthenes

n= 100 CPU time= 0.000s

n= 300 CPU time= 0.000s

n= 1000 CPU time= 0.001s

n= 3000 CPU time= 0.003s

n= 10000 CPU time= 0.009s

n= 30000 CPU time= 0.027s

n=100000 CPU time= 0.095s

Prvočísla — měření času (4)

Implementation : prvocisla_eratosthenes2

n= 100 CPU time= 0.000s

n= 300 CPU time= 0.000s

n= 1000 CPU time= 0.000s

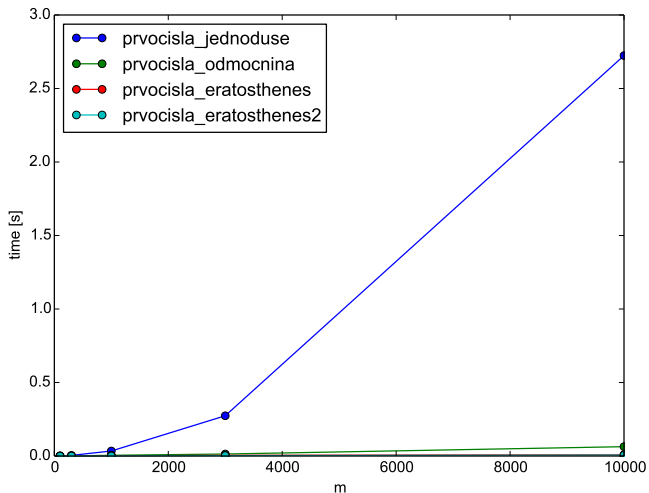
n= 3000 CPU time= 0.002s

n= 10000 CPU time= 0.005s

n= 30000 CPU time= 0.017s

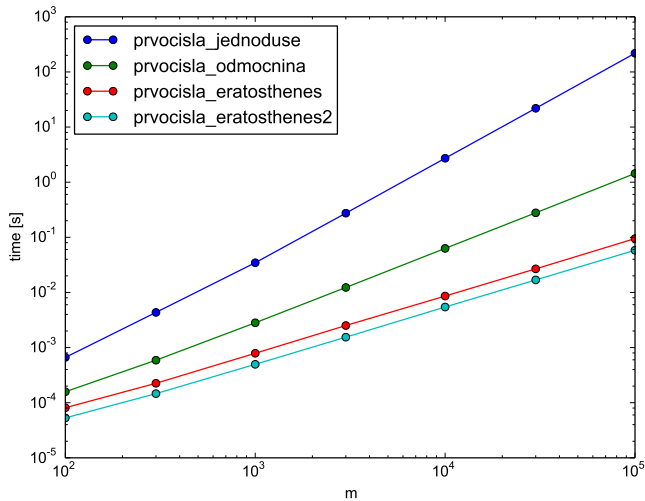
n=100000 CPU time= 0.058s

Prvočísla — graf



Kvalitativní (řádové) rozdíly.

Prvočísla — graf (2)



Empirická časová složitost

Teoretická časová složitost

Teoretická analýza časová složitosti

- ▶ Studujeme funkci $T(n)$
- ▶ Velikost problému n
 - ▶ vstupní parametr
 - ▶ délka vstupních dat
 - ▶ parametrů může být více
- ▶ Čas běhu programu T
 - ▶ Ve fyzických jednotkách
 - ▶ V počtu 'typických operací' — přiřazení, porovnání, sčítání, . . . (*výpočetní model*)

Asymptotická časová složitost

Přesný vzorec pro $T(n)$ není nutný...

- ▶ Zajímají nás pouze kvalitativní rozdíly
- ▶ Multiplikativní konstanty zanedbáme
 - ▶ vliv počítače, programátora, programovacího jazyka...
 - ▶ "vždycky si můžeme koupit rychlejší počítač"
- ▶ Zajímá nás chování pro velká n
 - ▶ Rychlost růstu $T(n)$ pro $n \rightarrow \infty$
 - ▶ Pomaleji rostoucí části $T(n)$ zanedbáme

Asymptotická notace

Řád růstu funkce / big- O notation

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

Pro polynomiální $f(n)$ — člen nejvyššího řádu, bez konstanty.

Příklady:

$$f(n) = 456211n + 235166 \quad \text{je } O(n)$$

$$f(n) = n(n + 2)/2 \quad \text{je } O(n^2)$$

$$f(n) = 442(n + 12) \log n \quad \text{je } O(n \log n)$$

$$f(n) = 4n^3 + 100n^2 + 1000n + 5000 \quad \text{je } O(n^3)$$

Asymptotická notace

Řád růstu funkce / big- O notation

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

Ověření: $f(n) = 4n^3 + 100n^2 + 1000n + 5000$ je $O(n^3)$

Možné ověření:

$$n \geq 100 \quad \Rightarrow \quad n^3 \geq 100n^2, \quad n^3 \geq 1000n, \quad n^3 \geq 5000$$

a proto


$$n \geq 100 \quad \Rightarrow \quad 4n^3 + 100n^2 + 1000n + 5000 \leq 7n^3$$

$$n_0 = 100, \quad c = 7$$

Asymptotická notace

Řád růstu funkce / big- O notation

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

 $O(\cdot)$ notace je *horní odhad*, ale uvádíme ten nejlepší známý.

Tedy $f(n) = 4n^3 + 100n^2$ je nejenom $O(n^3)$, ale zároveň i $O(n^4)$ a $O(n^{10})$.

Asymptotická notace (2)

Big- Θ notation

$f(n) : \mathbb{N} \rightarrow \mathbb{R}_0^+$ je $\Theta(g(n))$ pokud existují konstanty $c_1 > 0$, $c_2 > 0$ a $n_0 > 0$ takové, že $c_1g(n) \leq f(n) \leq c_2g(n)$ pro všechna $n \geq n_0$.

Tedy konstantní násobek g je zároveň horní a dolní odhad.

$4n^3 + 100n^2$ je $O(n^3)$, $O(n^4)$, $O(n^{10}) \dots$

✗

$4n^3 + 100n^2$ je pouze $\Theta(n^3)$, ale nikoliv i $\Theta(n^4)$ či $\Theta(n^{10})$.

Asymptotická notace (2)

Big- Θ notation


$f(n) : \mathbb{N} \rightarrow \mathbb{R}_0^+$ je $\Theta(g(n))$ pokud existují konstanty $c_1 > 0$, $c_2 > 0$ a $n_0 > 0$ takové, že $c_1g(n) \leq f(n) \leq c_2g(n)$ pro všechna $n \geq n_0$.

Tedy konstantní násobek g je zároveň horní a dolní odhad.

$4n^3 + 100n^2$ je $O(n^3)$, $O(n^4)$, $O(n^{10}) \dots$

✗

$4n^3 + 100n^2$ je pouze $\Theta(n^3)$, ale nikoliv i $\Theta(n^4)$ či $\Theta(n^{10})$.

 Θ notace je přesnější, ale v praxi se často setkáte s $O(\cdot)$ ve významu $\Theta(\cdot)$. Odhad $O(\cdot)$ je snazší nalézt.

Asymptotická notace (3)

Limita pro velká n

$$f(n) \stackrel{n \rightarrow \infty}{\sim} g(n) \text{ pokud } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 1.$$

- ▶ Zkráceně $f(n) \sim g(n)$ (je asymptoticky rovno)
- ▶ Pro polynomy, pouze člen nejvyššího řádu, včetně konstanty.

$$f(n) = 4n^3 + 100n^2 + 1000n + 5000 \quad \sim 4n^3$$

$$f(n) = 456211n + 235166 \quad \sim 456211n$$

$$f(n) = n(n+2)/2 \quad \sim n^2/2$$

$$f(n) = 442(n+12) \log n \quad \sim 442n \log n$$

- ▶ Místo f je $O(n^3)$ píšeme $f(n) \sim O(n^3)$

Časová složitost typicky závisí na datech (nejen na velikosti n)

- ▶ **Průměrná složitost** (*Average complexity*)
 - ▶ složitá teoretická analýza
 - ▶ lze odhadnout z experimentů na typických datech
- ▶ **Nejhorsí složitost** (*Worst-case complexity*)
 - ▶ jen experimentálně nelze
 - ▶ teoretická analýza → různě přesné horní odhady

Druhy odhadů

Časová složitost typicky závisí na datech (nejen na velikosti n)

- ▶ **Průměrná složitost** (*Average complexity*)
 - ▶ složitá teoretická analýza
 - ▶ lze odhadnout z experimentů na typických datech
- ▶ **Nejhorsí složitost** (*Worst-case complexity*)
 - ▶ jen experimentálně nelze
 - ▶ teoretická analýza → různě přesné horní odhady

Složitost může záležet na více parametrech vstupních dat (např. počet vstupních čísel a jejich maximální hodnota)

Složitost hledání prvočísel

```
for n in range(2,m): # cyklus 2..m-1
    p=2 # začátek testu
    while p<n:
        if n % p == 0:
            break
        p+=1
    if p==n: # n je prvočíslo
        primes+= [p]
```

Složitost hledání prvočísel

```
for n in range(2,m): # cyklus 2..m-1
    p=2 # začátek testu
    while p<n:
        if n % p == 0:
            break
        p+=1
    if p==n: # n je prvočíslo
        primes+= [p]
```

Analýza:

- ▶ Vnější `for` cyklus proběhne $m - 1$ -krát
- ▶ Každý vnitřní cyklus `while` proběhne max. $n - 2$ -krát, kde $n < m$
- ▶ Vnitřní cyklus tedy proběhne max. m^2 -krát
- ▶ Složitost tohoto algoritmu je tedy $O(m^2)$

Složitost hledání prvočísel

```
for n in range(2,m): # cyklus 2..m-1
    p=2 # začátek testu
    while p<n:
        if n % p == 0:
            break
        p+=1
    if p==n: # n je prvočíslo
        primes+= [p]
```

Poznámky:

- ▶ Operace mimo vnitřní cyklus zanedbáme.
- ▶ Toto je horní odhad. Ve skutečnosti je složitost nižší, díky příkazu `break`.

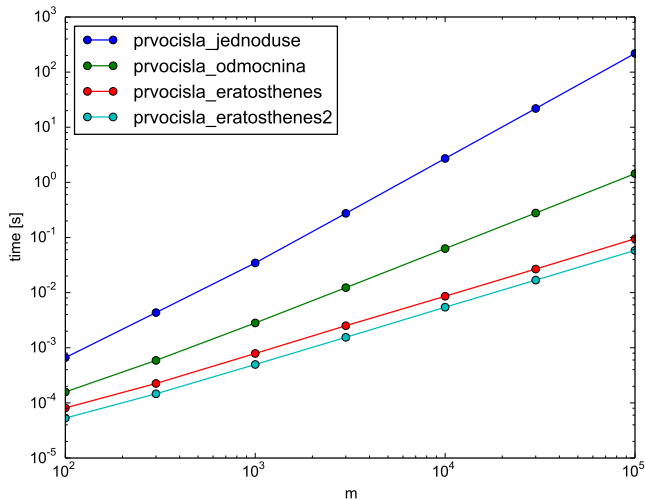
Složitost hledání prvočísel (2)

```
for n in range(2,m): # cyklus 2..m-1
    p=2 # začátek testu
    while p*p<n:
        if n % p == 0:
            break
        p+=1
    if p*p > n: # n je prvočíslo
        primes+= [n]
```

Analýza:

- ▶ Vnější `for` cyklus proběhne $m - 1$ -krát
- ▶ Vnitřní `while` cyklus proběhne max. $(\sqrt{n} - 1)$ -krát, kde $n < m$
- ▶ Vnitřní cyklus tedy proběhne max. $m^{1.5}$ -krát
- ▶ Složitost tohoto algoritmu je tedy $O(m^{1.5})$ (nebo lepší)

Prvočísla — graf



Asymptotická složitost Eratostenova síta je $O(n \log(\log n))$

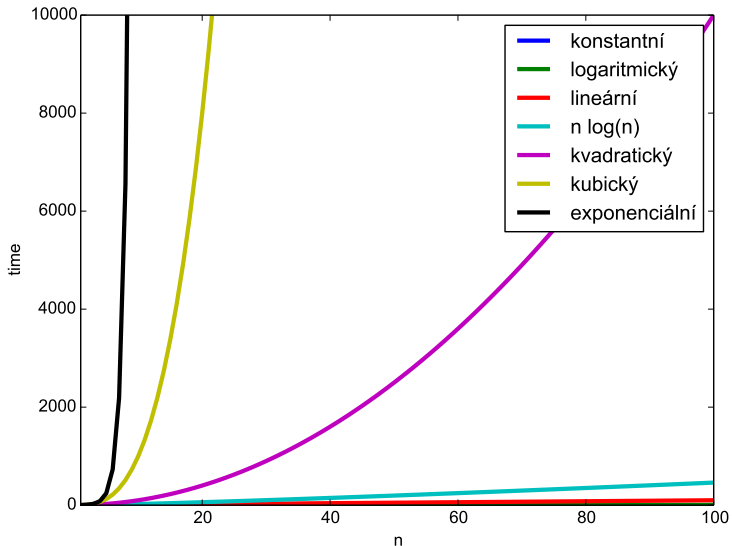
Srovnání složitostí

složitost		poznámka
konstantní	$O(1)$	nejrychlejší
logaritmická	$O(\log n)$	skoro jako $O(1)$, např. vyhledávání
lineární	$O(n)$	rychlé, použitelné pro velká data
	$O(n \log n)$	skoro jako $O(n)$, např. třídění
kvadratické	$O(n^2)$	trochu pomalejší, vhodné do $n \approx 10^6$
kubické	$O(n^3)$	pomalejší, vhodné do $n \lesssim 10^4$
polynomiální	$O(n^k)$	pomalé
exponenciální	$O(b^n)$	velmi pomalé, do $n \approx 50$
	$O(n!), O(n^n)$	nejpomalejší, do $n \approx 15$



Jak se změní čas, zvětší-li se $n \rightarrow 2n$?

Srovnání složitostí (2)




Složitost základních operací v Pythonu

- ▶ **V konstantním čase:** `len()`, `a[i]` (indexace), `a+=[v]` (přidání na konec), `a.pop()` (smazání posledního prvku)
- ▶ **V lineárním čase:** `a+b` (spojení), `a[i:j]` (řez pole), `a.insert()` (vkládání doprostřed pole), `max()`, `sum()`...
- ▶ **V čase $n \log n$:** `a.sort()`

Složitost základních operací v Pythonu

- ▶ **V konstantním čase:** `len()`, `a[i]` (indexace), `a+=[v]` (přidání na konec), `a.pop()` (smazání posledního prvku)
- ▶ **V lineárním čase:** `a+b` (spojení), `a[i:j]` (řez pole), `a.insert()` (vkládání doprostřed pole), `max()`, `sum()`...
- ▶ **V čase $n \log n$:** `a.sort()`

 Složitost přidání prvku na konec pole je konstantní jen v průměru (*amortized complexity*), nikoliv pro každou operaci.