5. Vyhledávání, řazení, složitost

BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Vyhledávání

    Motivační příklad

    Vyhledávání v Pythonu

    Vyhledávací algoritmy

- Část 2 – Řazení / Třídění

    Třídění v Pythonu

    Třídící algoritmy

- Část 3 – Složitost

    Empirická časová složitost

    Teoretická časová složitost

Část I

Vyhledávání

I. Vyhledávání

Motivační příklad

Vyhledávání v Pythonu

Vyhledávací algoritmy

Vyhledání čísla v řadě

Problém

- Myslím si přirozené číslo X mezi 1 a 1000.
- Možné otázky:
    - Je X menší než N?
    - Kolik otázek potřebujete na odhalení čísla?
    - Mezi kolika čísly jste schopni odhalit skryté číslo na K otázek?

Řešení

- Metoda půlení intervalu
- K otázek: rozlišíme mezi 2^K čísly
- N čísel: potřebujeme $log_2 N$ otázek

> Vyhledávání v (připravených) datech je častý problém: web, slovník, ...

Konkrétní problém

Problém

- Mějme posloupnost (pole) $a_0, \ldots, a_{N-1}$
- Mějme hodnotu q
- Úkol: zjistit, zda existuje $a_i = q$

Varianty

- Výstup: ano/ne nebo pozice
- Hledání opakujeme mnohokrát pro stejné a
  - předzpracování
- Posloupnost a je setříděná.
- Stochastické hledání

Vyhledávání a logaritmus

Naivní metoda – průchod seznamu

- lineární vyhledávání, složitost $O(n)$
- pomalé (viz např. databáze s milióny záznamů)
- jen velmi krátké seznamy

Rozumná metoda – půlení intervalu

- logaritmický počet kroků (vzhledem k délce seznamu)
- složitost $O(log(n))$

I. Vyhledávání

Operátor `in` 1/2

- Obsahuje pole daný prvek?

```
>>> a=[17,20,26,31,44,77,65,55,93]
>>> 20 in a
True
>>> 30 in a
False
>>> 30 not in a
True
>>> 20 not in a
False
```

Operátor `in` 2/2

- `in` / `not in` funguje i pro řetězce, n-tice a podobné typy

```
>>> "omo" in "Olomouc"  # podřetězec
True
>>> "" in "Olomouc"     # prázdný řetězec
True
>>> "olo" in "Olomouc"  # rozlišuje velikost písmen
False
>>> 4 in (3,4)
True
>>> 3. in (3,4)         # na typu nezáleží
True
```

Další funkce

- Funkce `index` vrací pozici prvního výskytu prvku v seznamu

```
>>> a=[3,4,5,6,3,4,8,6,5]
>>> a.index(4)
1
```

- Funkce `count` vrací počet výskytů prvku v seznamu

```
>>> a.count(3)
2
```

- Jak najít více výskytů v seznamu? Funkce `enumerate`

```
>>> [i for i, n in enumerate(a) if n == 3]
[0, 4]
```

I. Vyhledávání

Lineární vyhledávání

- Procházíme postupně a než narazíme na q

```
>>> def sequential_search(a,q):
...     """ Returns True if a contains q """
...     for x in a:
...         if x==q:
...             return True
...     return False
...
```

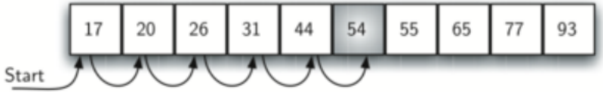


Image from Miller & Ranum: Problem solving with algothms and data structures

Lineární vyhledávání

- Procházíme postupně `a` než narazíme na `q`

```
>>> def sequential_search(a,q):
...     """ Returns True if a contains q """
...     for x in a:
...         if x==q:
...             return True
...     return False
...
```

- **Počet porovnání:**

	nejméně	nejvíce	průměrně
$q \notin a$	N	N	N
$q \in a$	1	N	N/2

  - Složitost $O(N)$, kde `N=len(a)`.
  - Rychleji to nejde, protože na každý prvek a_i se musíme podívat.

Binární vyhledávání

Vyhledávání v setříděné posloupnosti

- Mějme posloupnost (pole) $a_1 \leq a_2 \leq a_3 \leq \cdots \leq a_{N-1} \leq a_N$
- Mějme hodnotu q
- Úkol: zjistit, zda existuje $a_i = q$
- Je vyhledávání v setříděném poli rychlejší?

Binární vyhledávání

Hlavní myšlenky

- Postupně zmenšujeme interval indexů, kde by mohlo ležet q
- V každém kroku porovnáme q s prostředním prvkem intervalu a podle toho zvolíme jeden z podintervalů.
- Skončíme, pokud je prvek nalezen, nebo pokud je podinterval prázdný.

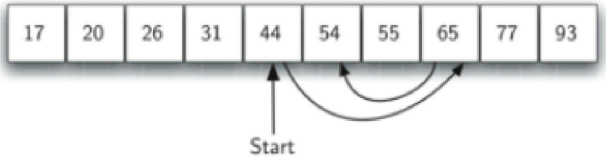


Image from Miller & Ranum: Problem solving with algothms and data structures

Binární vyhledávání – implementace

```
>>> def binary_search(a,q):
...     l=0                 # first index of the subinterval
...     h=len(a)-1          # last index of the subinterval
...     while l<=h:
...         m=(l+h)//2 # middle point
...         if a[m]==q:
...             return True
...         if a[m]>q:
...             h=m-1
...         else:
...             l=m+1
...     return False
...
```

Binární vyhledávání – časová složitost

Počet porovnání

- Počet prvků v intervalu je $d = h - l + 1$ a v první iteraci $d = N$
- V každé iteraci se interval d zmenší nejméně na polovinu
- Po t iteracích je $d \leq N2^{-t}$
- Dokud algoritmus běží, musí platit $d \geq 1$,

$$1 \leq d \leq N2^{-t}$$
$$2^t \leq N \quad \Rightarrow \quad t \leq \log_2 N$$

- Počet porovnání $t \sim \log_2 N$
- Složitost $O(\log n)$
- Strategie *rozděl a panuj (Divide and conquer)*

Část II

Řazení / Třídění

Úvod

Terminologická poznámka

- anglicky "sorting algorithms"
- česky používáno: řadicí algoritmy nebo třídicí algoritmy
- řadicí vesměs považováno za "správnější"

Proč se tím zabývat

- procvičení práce se seznamy
- ilustrace algoritmického myšlení, technik návrhu algoritmů
- typický příklad drobné změny algoritmu s velkým dopadem na rychlost programu

Doporučené zdroje

- `http://www.sorting-algorithms.com/`
  - animace
  - kódy
  - vizualizace
- `http://sorting.at/`
  - elegantní animace
- více podobných: Google → sorting algorithms
- a na zpestření:
  - xkcd Ineffective Sorts: `https://xkcd.com/1185/`
  - Bubble-sort with Hungarian folk dance: `http://www.youtube.com/watch?v=lyZQPjUT5B4`

Třídění

- Mějme posloupnost (pole) $A = [a_0, \dots, a_{N-1}]$ a relaci '$\leq$'
- Najděte takovou permutaci $B = [b_0, \dots, b_{N-1}]$ pole A, aby $b_0 \leq b_1 \leq \dots \leq b_{N-1}$.

Poznámky:

- Formy výstupu:
  - Třídění na místě *(in place)*
  - Vytvoření nového pole B, pole A zůstává nezměněno.
  - Najdeme indexy $j_1, j_2, \dots, j_N$, tak aby $b_i = a_{j_i}$ (`a[j[i]]`)
- Stabilní třídění — zachovává pořadí ekvivalentních prvků.

II. Řazení / Třídění

Třídění v Pythonu

Třídící algoritmy

Funkce `sorted`

- Funkce `sorted` vrací nové setříděné pole

```
>>> a=[80,43,20,15,90,67,51]
>>> sorted(a)
[15, 20, 43, 51, 67, 80, 90]
```

- Metoda `sort` setřídí pole na místě (úspornější)

```
>>> a.sort()
>>> a
[15, 20, 43, 51, 67, 80, 90]
```

- Třídění sestupně

```
>>> sorted(a, reverse=True)
[90, 80, 67, 51, 43, 20, 15]
```

Třídění řetězců

- Lze třídit veškeré porovnatelné typy, například řetězce

```
>>> names=["Barbora", "Adam", "David", "Cyril"]
>>> sorted(names)
['Adam', 'Barbora', 'Cyril', 'David']
```

- Třídění není podle českých pravidel

```
>>> sorted(["pán","paže"])
['paže', 'pán']
```

- Další možnosti

```
>>> s = ['Ostrava', 'automobil', 'Znojmo']
>>> sorted(s)
['Ostrava', 'Znojmo', 'automobil']
>>> sorted(s, key=str.lower)
['automobil', 'Ostrava', 'Znojmo']
>>> sorted(s, key=len)
['Znojmo', 'Ostrava', 'automobil']
```

Třídění *n*-tic

- *n*-tice jsou tříděny postupně podle složek

```
>>> a=[(50,2),(50,1),(40,100),(40,20)]
>>> sorted(a)
[(40, 20), (40, 100), (50, 1), (50, 2)]

>>> studenti=[("Bara",18),("Adam",20), ("David",15),("Cyril",25)]
>>> sorted(studenti)
[('Adam', 20), ('Bara', 18), ('Cyril', 25), ('David', 15)]
```

Co když chci ale třídit podle jiné položky?

Uživatelská třídící `lambda` funkce

- Funkce v parametru `key` transformuje prvky pro třídění.
- Třídění podle druhé složky dvojice

```
>>> a=[(50,2),(50,1),(40,100),(40,20)]
>>> sorted(a,key=lambda x: x[1])
[(50, 1), (50, 2), (40, 20), (40, 100)]

>>> studenti=[("Bara",18),("Adam",20), ("David",15),("Cyril",25)]
>>> sorted(studenti, key=lambda x: x[1])
[('David', 15), ('Bara', 18), ('Adam', 20), ('Cyril', 25)]
```

Uživatelská třídící `lambda` funkce 2/2

- Třídění bez ohledu na velikost písmen

```
>>> s=["Python","Quido","abeceda","zahrada"]
>>> sorted(s)
['Python', 'Quido', 'abeceda', 'zahrada']

>>> sorted(s,key=lambda x: x.lower())
['abeceda', 'Python', 'Quido', 'zahrada']
```

- Třídění podle počtu výskytů znaku ve slově

```
>>> s = ["prase", "Kos", "ovoce", "Pes", "koza", "ovce", "kokos"]
>>> sorted(s,key=lambda x: x.count('o'))
['prase', 'Pes', 'Kos', 'koza', 'ovce', 'ovoce', 'kokos']
```

II. Řazení / Třídění

Třídění v Pythonu

Třídící algoritmy

P5.1 Unikátní prvky, nejčastější prvek

- Máme seznam prvků, např. výsledky dotazníku

Oblíbený programovací jazyk :-)

```
>>> lang = ["Python", "Java", "C", "Python", "PHP", "Python",
... "Java", "JavaScript", "C", "Pascal"]
```

- Chceme:
  - seznam unikátních hodnot
  - nejčastější prvek

Řešení

- přímočaré: opakované procházení seznamu
- efektivní: seřadit a pak jednou projít
- elegantní: využití pokročilých datových struktur / konstrukcí

P5.1 Unikátní prvky – 1. řešení

```
>>> def unique(alist):
...     alist = sorted(alist)
...     # rozdilne chovani od alist.sort() !!
...     result = []
...     for i in range(len(alist)):
...         if i == 0 or alist[i-1] != alist[i]:
...             result.append(alist[i])
...     return result
...
>>> unique(lang)
['C', 'Java', 'JavaScript', 'PHP', 'Pascal', 'Python']
```

Unikátní prvky – 2. řešení


```
>>> def unique(alist):
...     return list(set(alist))
...
>>> unique(lang)
['Python', 'C', 'JavaScript', 'PHP', 'Java', 'Pascal']
```

P5.1 Nejčastější prvek – 1. řešení


```
>>> def most_common(alist):
...     alist = sorted(alist)
...     max_value, max_count = None, 0
...     current_value, current_count = None, 0
...     for value in alist:
...         if value == current_value: current_count += 1
...         else: current_value = value; current_count = 1
...
...         if current_count > max_count:
...             max_value = current_value
...             max_count = current_count
...     return max_value
...
>>> most_common(lang)
'Python'
```

P5.1 Nejčastější prvek – 2. řešení 5/5

```
>>> def most_common(alist):
...     return max(alist, key=alist.count)
...
>>> most_common(lang)
'Python'
```

II. Řazení / Třídění

Třídění v Pythonu

Třídící algoritmy

Třídící algoritmy

- Třídění probubláváním *(bubble sort)*
- Třídění zatřiďováním *(insertion sort)*
- Třídění výběrem *(selection sort)*
- Shell sort
- Třídění spojováním *(merge sort)*
- *Quick sort*
- `sort`, `sorted`

Řazení probubláváním – bubble sort

- Vyměňuje sousední prvky ve špatném pořadí.
- Jeden průchod.

54	26	93	17	77	31	44	55	20	Exchange
26	54	93	17	77	31	44	55	20	No Exchange
26	54	93	17	77	31	44	55	20	Exchange
26	54	17	93	77	31	44	55	20	Exchange
26	54	17	77	93	31	44	55	20	Exchange
26	54	17	77	31	93	44	55	20	Exchange
26	54	17	77	31	44	93	55	20	Exchange
26	54	17	77	31	44	55	93	20	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

Řazení probubláváním – implementace

```
>>> def bubble_sort(a):
...     """ sorts array a in-place in ascending order"""
...     for i in range(len(a)-1,0,-1):
...         # i=n-1..1.  a[i+1:] is already sorted
...         for j in range(i):
...             if a[j] > a[j+1]:
...                 a[j], a[j+1] = a[j+1], a[j] # exchange
...
>>> a = [54, 26, 93, 17, 77, 31, 44, 55, 20]
>>> bubble_sort(a)
>>> a
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Složitost

- Vnější smyčka proběhne $N - 1 \sim N$-krát
- Vnitřní smyčka proběhne vždy i-krát, kde $i < N$, tedy max. N-krát

Řazení probubláváním – vylepšení

- Pokud neproběhla žádná výměna, je pole setříděné.

```
def bubble_sort(a):
  """ sorts array a in-place in ascending order"""
  for i in range(len(a)-1,0,-1):
    # i=n-1..1.  a[i+1:]  is already sorted
    exchanged=False # exchanges in this iteration?
    for j in range(i):
      if a[j]>a[j+1]:
        a[j],a[j+1]=a[j+1],a[j] # exchange
        exchanged=True
    if not exchanged: break
```

Třídění probubláváním – kontrola

- Testování na vzorku dat

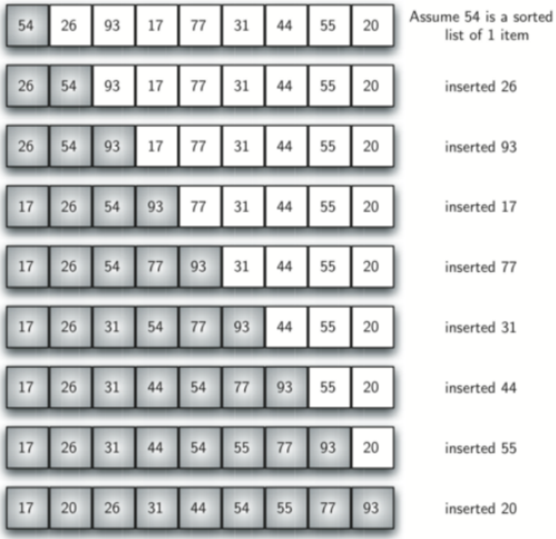
```
from sorting_experiments import *
a=[31, 60, 23, 91, 62, 65, 59, 92, 42, 74]
bubble_sort(a)
print(a)
```

- Správný test je důkladnější:

```
def test_sort(f=bubble_sort):
  for j in range(100):
    n=100
    a=[random.randrange(100000) for i in range(n)]
    f(a)
    for i in range(n-1):
      assert(a[i]<=a[i+1])
  print(f.__name__," sort test passed")
```

Třídění zatřiďováním – insertion sort

- prvek a_i zatřídíme do již setříděných $a_0, \ldots, a_{i-1}$



Třídění zatřiďováním – implementace

```python
def insertion_sort(a):
  """ sorts array a in-place in ascending order"""
  for i in range(1,len(a)):    # a[0:i] is sorted
    val=a[i]
    j=i
    while j>0 and a[j-1]>val:
      a[j]=a[j-1]
      j-=1
    a[j]=val
```

Složitost

- Vnější smyčka proběhne $N-1 \sim N$-krát.
- Vnitřní smyčka proběhne max. $i < N$ krát.
- Počet porovnání je tedy max. $N^2 \quad \rightarrow \quad$ složitost $O(N^2)$.
- Nepoužívá výměny, ale posun (rychlejší).
- Přirozeně detekuje setříděné pole.

Třídění zatřiďováním – kontrola

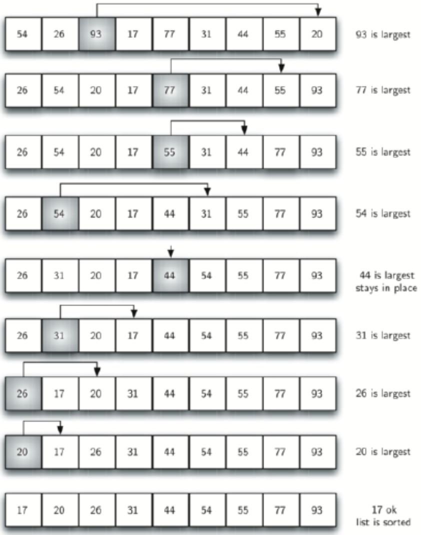
```
from sorting_experiments import *

a=[43, 22, 42, 7, 58, 85, 48, 82, 80, 1]
insertion_sort(a)
print(a)
```

```
[1, 7, 22, 42, 43, 48, 58, 80, 82, 85]
```

Třídění výběrem – selection sort

- Vybere maximum mezi $a_0, \ldots, a_{N-1}$, to umístí do a_{N-1}.
- Vybere maximum mezi $a_0, \ldots, a_{N-2}$, to umístí do $a_{N-2} \ldots$



Třídění výběrem – implementace

```python
def selection_sort(a):
  """ sorts array a in-place in ascending order"""
  for i in range(len(a)-1,0,-1):
    # find out what should go to a[i]
    max_pos=0 # index of the maximum
    for j in range(1,i+1):
      if a[j]>a[max_pos]:
        max_pos=j
    a[i],a[max_pos]=a[max_pos],a[i]
```

Složitost

- Vnější smyčka proběhne $N-1 \sim N$-krát
- Vnitřní smyčka proběhne vždy i-krát, kde $i < N$, tedy max. N-krát
- Počet porovnání je tedy max. $N \quad \rightarrow \quad$ složitost $O(N^2)$
- Pouze jedna výměna v každé vnější smyčce

Třídění výběrem – kontrola

```
from sorting_experiments import *

a=[60, 46, 31, 69, 45, 11, 43, 14, 61, 36]
selection_sort(a)
print(a)
```

```
[11, 14, 31, 36, 43, 45, 46, 60, 61, 69]
```

Část III

Složitost algoritmů

Složitost algoritmů

- **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

III. Složitost algoritmů

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

III. Složitost algoritmů

Empirická časová složitost

Teoretická časová složitost

P5.2 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)

lec04/porovnani_prvocislo.py

```
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
```

P5.2 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)

lec04/porovnani_prvocislo.py

```
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
```

P5.2 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)

lec04/porovnani_prvocislo.py

```
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
```

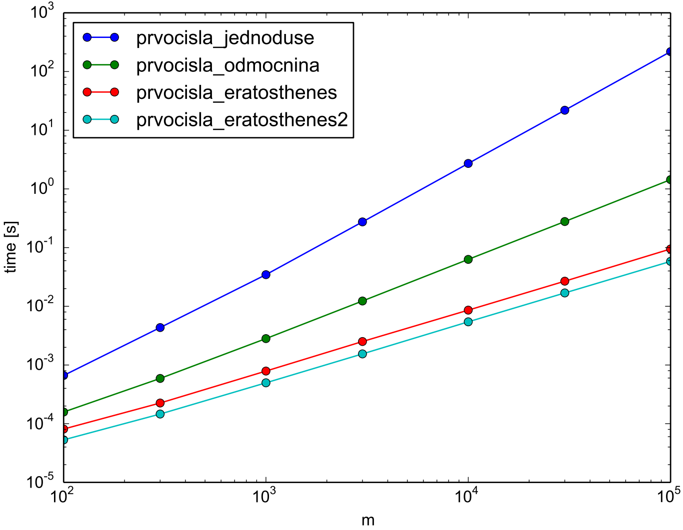
P5.2 Prvočísla

- 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)

lec04/porovnani_prvocislo.py

```
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
```

P5.2 Prvočísla – graf



III. Složitost algoritmů

Empirická časová složitost

Teoretická časová složitost

Teoretická analýza časová složitosti

- Studujme 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 $n \to \infty$
  - Pomaleji rostoucí části $T(n)$ zanedbáme

Řád růstu funkce – big-O notation

$f(n) : \mathbb{N} \to \mathbb{R}_0^+$ je $O\big(g(n)\big)$ 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:

$f(n) = 456211n + 235166$	$O\big(n\big)$
$f(n) = n(n+2)/2$	$O\big(n^2\big)$
$f(n) = 442(n+12)\log n$	$O\big(n \log n\big)$
$f(n) = 4n^3 + 100n^2 + 1000n + 5000$	$O\big(n^3\big)$

$O\big(\cdot\big)$ 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 $O(n^{10})$.

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
- **Nejhorší složitost** *(Worst-case complexity)*
  - jen experimentálně nelze
  - teoretická analýza $\rightarrow$ 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]
```

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`.

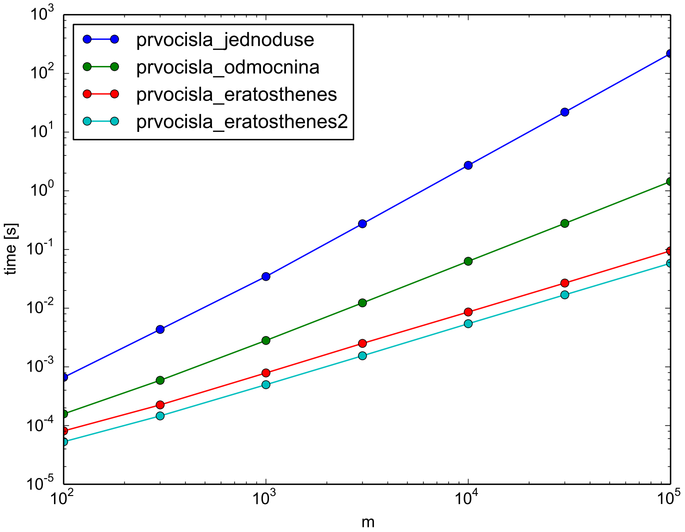
Složitost hledání prvočísel

```python
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
- 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)*

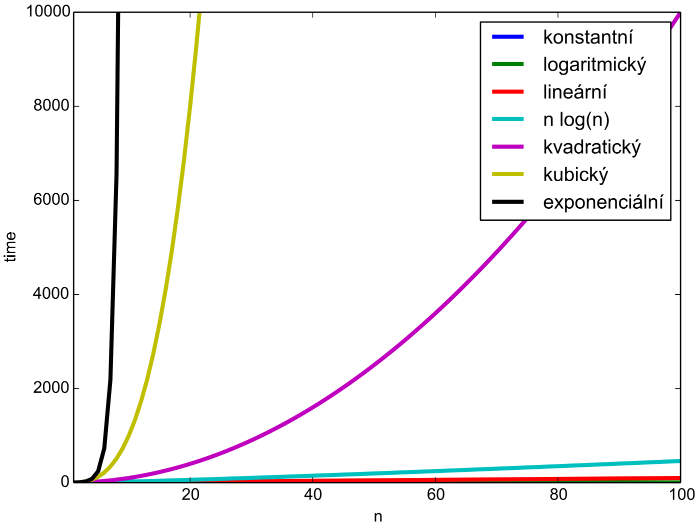
Prvočísla – graf



Srovnání složitostí 1/2

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 $n \approx 15$

Srovnání složitostí



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, nikoliv pro každou operaci.