

4. Pole. Vyhledávání, řazení, složitost

BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Pole
 - Příklady – 1D pole
 - Dvojměrné pole – matice
 - Příklad – 2D pole
- Část 2 – Složitost
 - Empirická časová složitost
 - Teoretická časová složitost
- Část 3 – Vyhledávání
 - Motivační příklad
 - Vyhledávání v Pythonu
 - Vyhledávací algoritmy
- Část 4 – Řazení / Třídění
 - Třídění v Pythonu
 - Třídící algoritmy

Část I

Pole

I. Pole

Příklady – 1D pole

Dvojměrné pole – matice

Příklad – 2D pole

Formulace problému

Kolikrát musíme náhodně s opakováním vybrat z množiny N prvků, než vybereme každý prvek alespoň jednou?

Matematická analýza

$$\text{počet výběrů} = T(N) = \lceil N H_N \rceil \approx N(\log N + \gamma) + 0.5$$

$$T(50) = 225$$

H_N – harmonické číslo n -tého řádu

$\gamma \approx 0,577$ – Euler–Mascheroni konstanta

```
1 import random
2 import sys
4 n=50 # pocet ruznych prvku
6 collectedCount=0 # pocet jiz vybranych ruznych prvku
7 isCollected=[False]*n # jiz jsme videli tento prvek
8 count=0 # kolik prvku jsme jiz vybrali
10 while collectedCount < n:
11     r=random.randrange(n) # nahodny prvek 0..n-1
12     count+=1
13     if not isCollected[r]: # novy prvek
14         collectedCount+=1
15         isCollected[r]=True
17 print("Pro n=%d bylo potreba %d vyberu." % (n,count))
```

- Dělitelná jen 1 a sebou samým

Předmět zájmu matematiků od pradávna, cca od 1970s i velmi důležité aplikace (kryptografie)

- Problémy s prvočísly
 - výpis (počet) prvočísel v zadaném intervalu
 - test prvočíselnosti
 - rozklad na prvočísla
- Přímočarý výpis prvočísel v intervalu

```
1 | for num in range(lower, upper + 1):
2 |     if num > 1:
3 |         for i in range(2, num):
4 |             if (num % i) == 0:
5 |                 break
6 |             else:
7 |                 print(num)
```

- Možná vylepšení naivního algoritmu
 - dělíme pouze dvojkou a lichými čísly
 - dělíme pouze do \sqrt{n}
 - ...
- Eratostenovo síto
 - Opakovaně provádíme
 - označ další neškrtnuté číslo v seznamu jako prvočíslo
 - všechny násobky tohoto čísla vyškrtni

Prvočísla – Eratosthenovo síto

```
1 def reset_multiples(is_candidate, i):
2     k = 0
3     while k < len(is_candidate):
4         is_candidate[k] = 0
5         k += i
6
7
8 def eratosthenes(n):
9     is_candidate = [1 for _ in range(n)]
10    for i in range(2, n):
11        if is_candidate[i]:
12            print(i, end=" ")
13            reset_multiples(is_candidate, i)
```

Histogram

- Histogram (PDF, Probability Density Function) je statistický ukazatel, popisující četnost sledované diskétní veličiny v zadaném intervalu
- Nechť pole `data` obsahuje celá čísla v intervalu 0 – 9

```
1 | data = [0, 1, 3, 6, 8, 1, 6, 3, 4, 3, 4, 1, 7, 5, 4, 9, 0, 0, 4, 2]
```

- Pro určení histogramu pak potřebujeme pole o stejné velikosti, jako je rozsah sledované veličiny. Pro každý výskyt hodnoty veličiny je pak inkrementována příslušná položka pole

```
1 | h = [0]*10;  
2 | for i in range(len(data)):  
3 |     h[data[i]] += 1  
4 | print(h)
```

I. Pole

Příklady – 1D pole

Dvojměrné pole – matice

Příklad – 2D pole

```
1 def print_2d_matrix(a):
2     for i in range(len(a)):
3         print(a[i])
5 a=[[0.]*4]*2
6 print_2d_matrix(a)
```

```
[0.0, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
```

```
1 a[0][1]=1.0
2 print_2d_matrix(a)
```

```
[0.0, 1.0, 0.0, 0.0]
[0.0, 1.0, 0.0, 0.0] # -> nefunguje jak chceme, kvůli aliasingu.
```

```
1 def create_2d_matrix (n, m, v):
2     "Creates a n-by-m matrix with initial value 'v'"
3     a=[]
4     for i in range(n):
5         a += [[v]*m]
6     return a
7
8 a=create_2d_matrix(2, 4, 0.0)
9 a[0][1] = 1.0
10 print_2d_matrix(a)
```

```
[0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
```

Toto je specialita Pythonu

```
1 def create_2d_matrix (n, m, v):
2     "Creates a n-by-m matrix with initial value 'v'"
3     return [[v]*m for i in range(n)]
5 a=create_2d_matrix(2,4,0.0)
6 a[0][1]=1.0
7 print_2d_matrix(a)
```

```
[0.0, 1.0, 0.0, 0.0]
[0.0, 0.0, 0.0, 0.0]
```

I. Pole

Příklady – 1D pole

Dvojměrné pole – matice

Příklad – 2D pole

Binomický koeficient (kombinační číslo)

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad \text{for } n \geq k \geq 0$$

- počet k prvkových podmnožin z n

$$\binom{3}{1} = 3, \quad \binom{4}{2} = 6$$

- koeficient v binomické větě

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$$

$$(x + y)^1 = x + y$$

$$(x + y)^2 = x^2 + 2xy + y^2$$

$$(x + y)^3 = x^3 + 3x^2y + 3y^2x + y^3$$

$$(x + y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$$

Pascalův trojúhelník

	$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$
$n = 0$	1				
$n = 1$	1	1			
$n = 2$	1	2	1		
$n = 3$	1	3	3	1	
$n = 4$	1	4	6	4	1

Platí rekurze

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Pascalův trojúhelník v Pythonu

Vypočítejte pole p , tak aby $p[n][k] = \binom{n}{k}$

```
1 def pascal_triangle(N):
2     p=[[1]]
3     for n in range(2,N+1):
4         prev = p[n-2] # předchozí řada
5         p += [[1] + [prev[k-1] + prev[k] for k in range(1,n-1)] + [1]]
6     return p
8 print_2d_matrix(pascal_triangle(5))
```

```
[1]
[1, 1]
[1, 2, 1]
[1, 3, 3, 1]
[1, 4, 6, 4, 1] # -> řádky pole nemusí mít stejnou délku.
```

Část II

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

II. 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

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)

[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
```

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)

[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
```


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)

[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
```

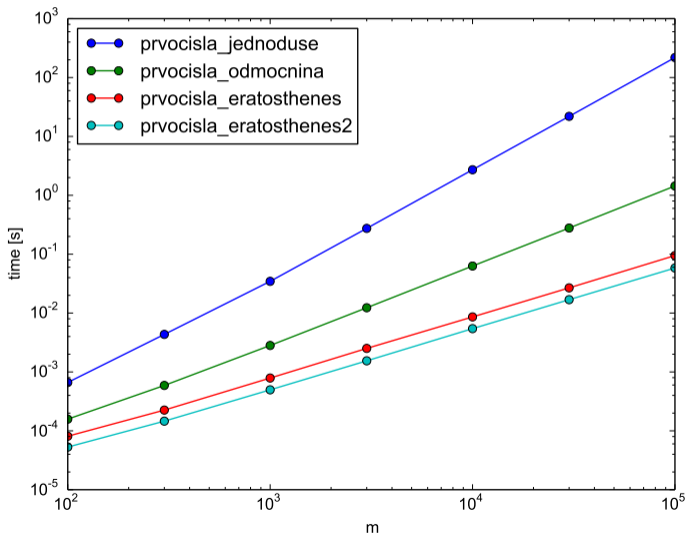
Příklad: Prvočísla

- 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)

[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
```

Prvočísla – graf



II. 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 $n \rightarrow \infty$
 - Pomaleji rostoucí části $T(n)$ zanedbáme

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:

$$f(n) = 456211n + 235166$$

$$O(n)$$

$$f(n) = n(n + 2)/2$$

$$O(n^2)$$

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

$$O(n \log n)$$

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

$$O(n^3)$$

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

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 → různě přesné horní odhady

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

Složitost hledání prvočísel

```
1  for n in range(2,m): # cyklus 2..m-1
2      p=2 # začátek testu
3      while p<n:
4          if n % p == 0:
5              break
6          p+=1
7      if p==n: # n je prvočíslo
8          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

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

Poznámky:

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

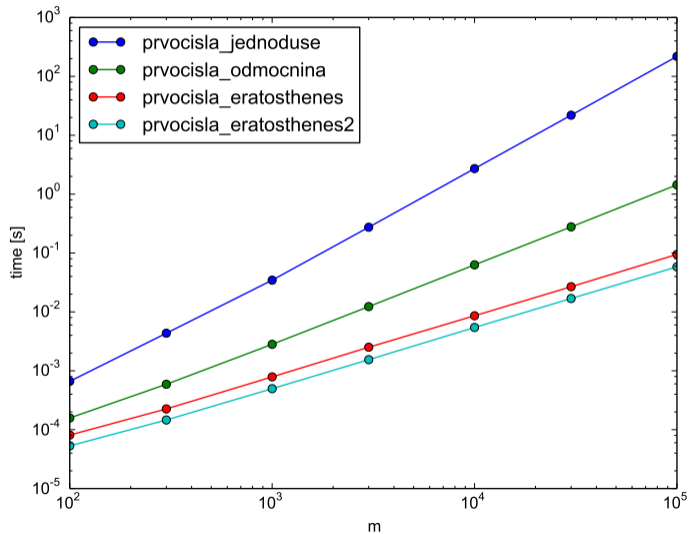
Složitost hledání prvočísel

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

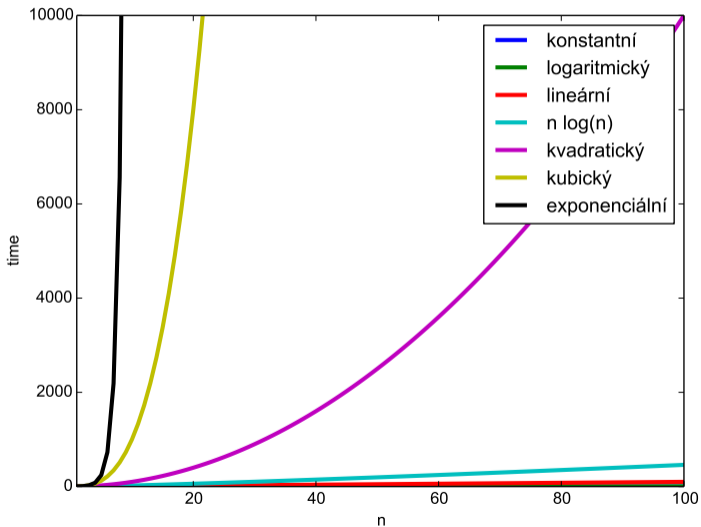
Analýza:

- 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



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 $n \approx 15$



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.

Část III

Vyhledávání

III. 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 čísla
- N čísel: potřebujeme $\log_2 N$ otázek

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

Konkrétní problém

Problém

- Mějme posloupnost (pole) a_0, \dots, 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 $O(\log(n))$

III. Vyhledávání

Motivační příklad

Vyhledávání v Pythonu

Vyhledávací algoritmy

- 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
```

- `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]
```


III. Vyhledávání

Motivační příklad

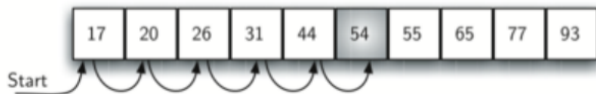
Vyhledávání v Pythonu

Vyhledávací algoritmy

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
```



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 $N=\text{len}(a)$.
- Rychleji to nejde, protože na každý prvek a_i se musíme podívat.

Binární vyhledávání

Vyhledávání v setříděné posloupnosti

- Mějme posloupnost (pole) $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_{N-1} \leq a_N$
- 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.



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 IV

Řazení / Třídění

Terminologická poznámka

- anglicky "sorting algorithms"
- česky používáno: řadící algoritmy nebo třídící algoritmy
- řadící 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=1yZQPjUT5B4>

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.

IV. Řazení / Třídění

Třídění v Pythonu

Třídící algoritmy

Funkce `sorted`

- Funkce `sorted` vrací nové seříděné pole

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

- Metoda `sort` seřídí 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(["pan","paze"])
['pan', 'paze']
```

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

- Další možnosti

```
>>> sorted(s, key=str.lower)
>>> sorted(s, key=len)
```

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)]
```

- 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)]
```


- 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']
```

Příklad – třídící funkce

```
1  from functools import cmp_to_key
3  def letter_cmp(a, b):
4      if a[1] > b[1]:
5          return -1
6      elif a[1] == b[1]:
7          if a[0] > b[0]: return 1
8          else: return -1
9      else: return 1
11 a = [('c', 2), ('b', 1), ('a', 3)]
12 a.sort(key=cmp_to_key(letter_cmp))
13 print(a)
```

```
[('a', 3), ('c', 2), ('b', 1)]
```

- Máme seznam prvků, např. výsledky dotazníku

Oblíbený programovací jazyk :-)

```
["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

```
1 def unique(alist):
2     alist = sorted(alist)
3     # rozdilne chovani od alist.sort() !!
4     result = []
5     for i in range(len(alist)):
6         if i == 0 or alist[i-1] != alist[i]:
7             result.append(alist[i])
8     return result
```

```
1 def unique(alist):
2     return list(set(alist))
```

```
1 def most_common(alist):
2     alist = sorted(alist)
3     max_value, max_count = None, 0
4     current_value, current_count = None, 0
5     for value in alist:
6         if value == current_value: current_count += 1
7         else: current_value = value; current_count = 1
8
9         if current_count > max_count:
10            max_value = current_value
11            max_count = current_count
12    return max_value
14 def most_common(alist):
15    return max(alist, key=alist.count)
```

IV. Řazení / Třídění

Třídění v Pythonu

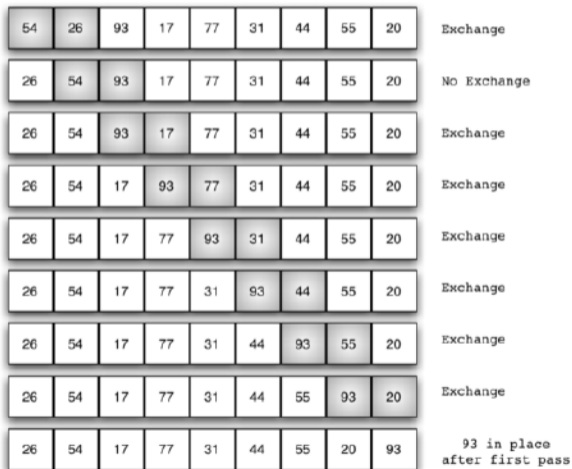
Třídící algoritmy

Třídící algoritmy

- Třídění probubláváním (*bubble sort*)
- Třídění zatřídováním (*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.



Řazení probubláváním – implementace

```
1 def bubble_sort(a):
2     """ sorts array a in-place in ascending order """
3     for i in range(len(a)-1,0,-1):
4         # i=n-1..1. a[i+1:] is already sorted
5         for j in range(i):
6             if a[j]>a[j+1]:
7                 a[j],a[j+1]=a[j+1],a[j] # exchange
```

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. $N^2 \rightarrow$ složitost $O(N^2)$
- Počet výměn je velký, element musí 'probublat' nakonec

Řazení probubláváním – vylepšení

- Pokud neproběhla žádná výměna, je pole setříděné.

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

Třídění probubláváním – kontrola

- Testování na vzorku dat

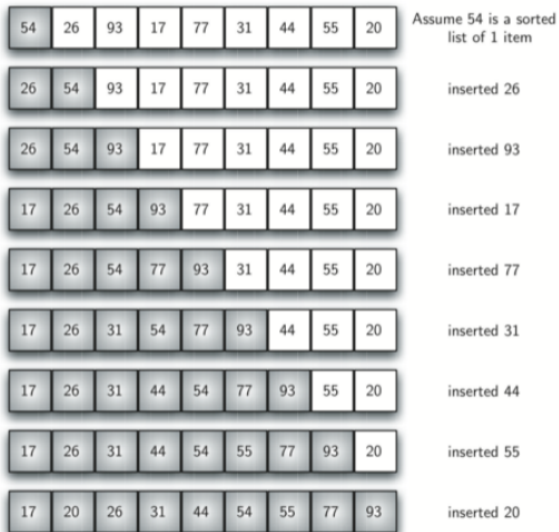
```
1 | from sorting_experiments import *
2 | a=[31, 60, 23, 91, 62, 65, 59, 92, 42, 74]
3 | bubble_sort(a)
4 | print(a)
```

- Správný test je důkladnější:

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

Třídění zatřídováním – insertion sort

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



Třídění zatřídováním – implementace

```
1 def insertion_sort(a):
2     """ sorts array a in-place in ascending order """
3     for i in range(1, len(a)):      # a[0:i] is sorted
4         val = a[i]
5         j = i
6         while j > 0 and a[j-1] > val:
7             a[j] = a[j-1]
8             j -= 1
9             a[j] = val
```

Složitost

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

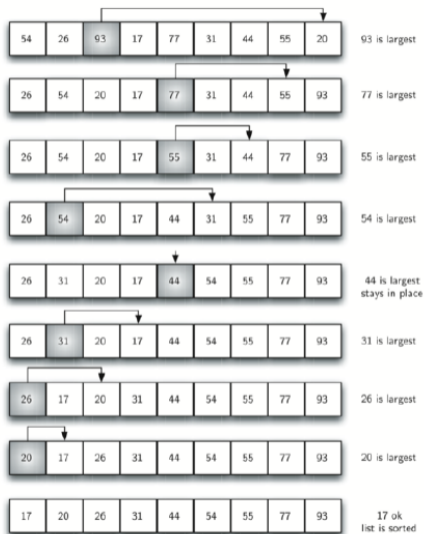
Třídění zatřídováním – kontrola

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



Třídění výběrem – implementace

```
1 def selection_sort(a):
2     """ sorts array a in-place in ascending order """
3     for i in range(len(a)-1,0,-1):
4         # find out what should go to a[i]
5         max_pos=0 # index of the maximum
6         for j in range(1,i+1):
7             if a[j]>a[max_pos]:
8                 max_pos=j
9         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. $N \rightarrow$ složitost $O(N^2)$
- Pouze jedna výměna v každé vnější smyčce

Třídění výběrem – kontrola

```
1  from sorting_experiments import *
3  a=[60, 46, 31, 69, 45, 11, 43, 14, 61, 36]
4  selection_sort(a)
5  print(a)
```

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