

Třídění a vyhledávání

Searching and sorting

Jan Kybic

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

2016–2018



Vyhledávání

Třídění

Třídící algoritmy

Vyhledávání

Searching

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

- ▶ 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 v Pythonu

operátor `in`

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
```

Vyhledávání v Pythonu (2)

`in / not in` funguje i pro řetězce, *n*-tice a podobné typy (*containers*)

```
>>> "omo" in "Olomouc"    # podřetězec
True
>>> "" in "Olomouc"      # prázdný řetězec
True
>>> "olo" in "Olomouc"   # rozlišuje velká/malá písmena
False
>>> 4 in (3,4)
True
>>> 3. in (3,4)          # na typu nezáleží
True
```

Lineární vyhledávání

linear/sequential search

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 algorithms and data structures

Lineární vyhledávání

linear/sequential search

Procházíme postupně a než narazíme na q

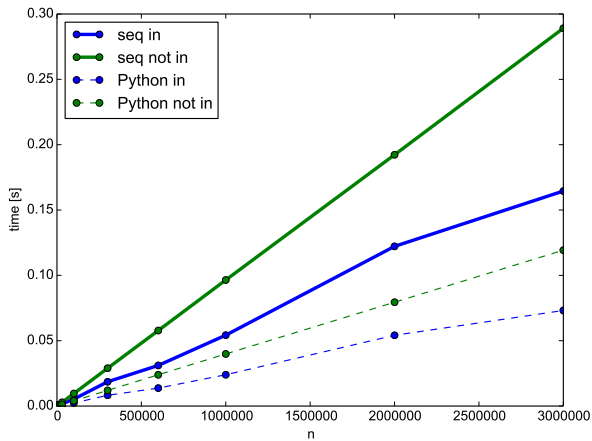
```
def sequential_search(a,q):  
    """ Returns 'True' if 'a' contains 'q' """  
    for x in a:  
        if x==q:  
            return True  
    return False
```

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 najde, protože na každý prvek a_i se musíme podívat.

Lineární vyhledávání — empirická složitost



Soubor `search_experiments.py`

Binární vyhledávání

Binary search

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

Binary search

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.

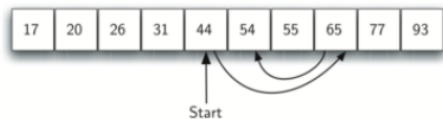


Image from Miller & Ranum: Problem solving with algorithms and data structures

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
```

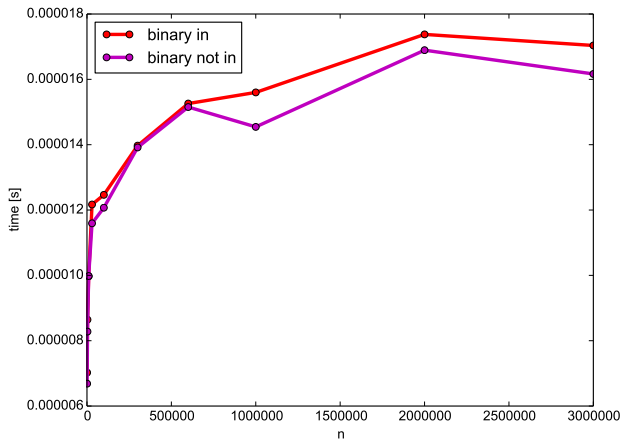
Počet porovnání

- ▶ 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 $d \geq 1$,

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

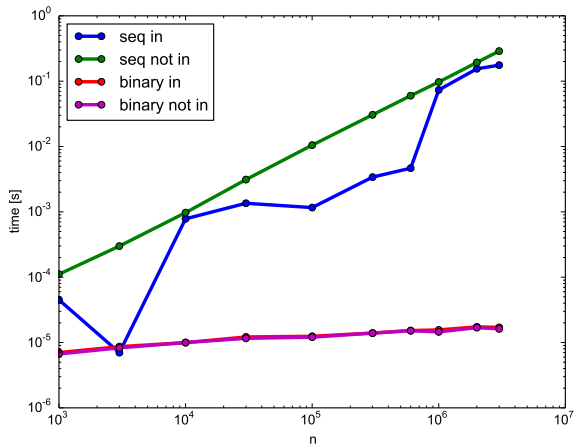
- ▶ Počet porovnání $t \sim \log_2 N$
- ▶ Složitost $O(\log n)$
- ▶ Strategie *rozděl a panuj* (*Divide and conquer*)

Binární vyhledávání — empirická složitost



Soubor `search_experiments.py`

Binární vs. sekvenční vyhledávání



Vyplatí se, pokud mnohokrát vyhledáváme nad stejným a .

Vyhledávání

Třídění

Třídící algoritmy

- ▶ 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.

Třídění v Pythonu

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 v Pythonu

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 řetězců

Lze třídit veškeré porovnatelné typy, například řetězce

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



Třídění není podle českých pravidel

```
print(sorted(["pan", "paze"]))
```

```
['pan', 'paze']
```

```
print(sorted(["pán", "paže"]))
```

```
['paže', 'pán']
```

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)]
```

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)]
```

Uživatelská třídící 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 funkce (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']
```

Vyhledávání

Třídění

Třídící algoritmy

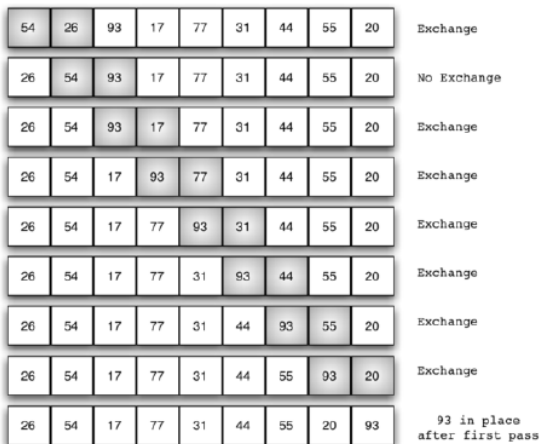
Třídící algoritmy

- ▶ Třídění probubláváním (*bubble sort*)
- ▶ Třídění zatříd'ováním (*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

Bubble sort

```
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
```

Řazení probubláváním — implementace

Bubble sort

```
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
```

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.

```
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

```
>>> a=[31, 60, 23, 91, 62, 65, 59, 92, 42, 74]
>>> bubble_sort(a)
>>> a
[23, 31, 42, 59, 60, 62, 65, 74, 91, 92]
```


Třídění probubláváním — kontrola

```
>>> a=[31, 60, 23, 91, 62, 65, 59, 92, 42, 74]
>>> bubble_sort(a)
>>> a
[23, 31, 42, 59, 60, 62, 65, 74, 91, 92]
```

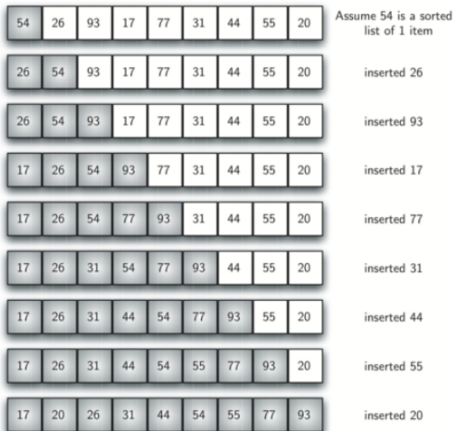
👉 Správný test je důkladnější:

```
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 zatřídováním

Insertion sort

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



Třídění zatřídováním — implementace

Insertion sort

```
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
```

Třídění zatřídováním — implementace

Insertion sort

```
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. $N^2 \rightarrow$ složitost $O(N^2)$.
- ▶ Nepoužívá výměny, ale posun (rychlejší).
- ▶ Přirozeně detekuje setříděné pole.

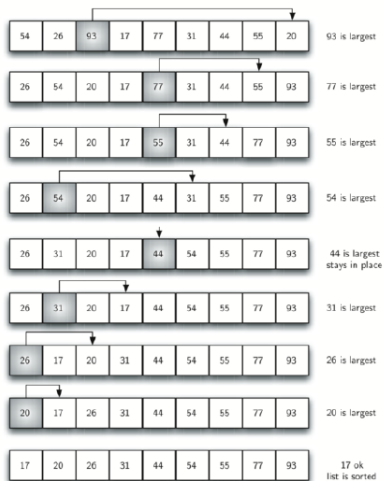
Třídění zatřídováním — kontrola

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

Selection sort

```
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]
```

Třídění výběrem — implementace

Selection sort

```
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

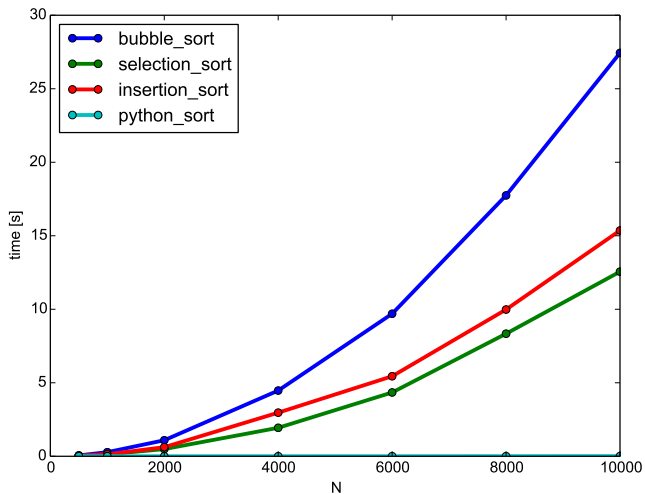
- ▶ Vnější smyčka proběhne $N - 1 \sim N$ -krát
- ▶ Vnitřní smyčka proběhne vždy i -krát, kde $i < N$, tedy max. N -krát
- ▶ Počet porovnání je tedy max. $N \rightarrow$ složitost $O(N^2)$
- ▶ Pouze jedna výměna v každé vnější smyčce

Třídění výběrem — kontrola

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

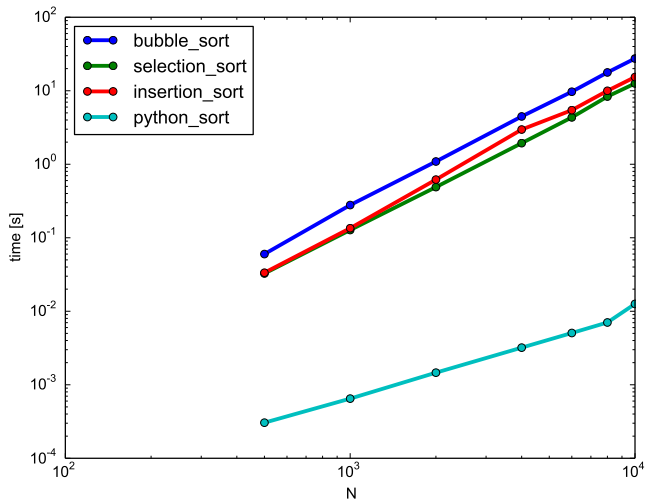
Porovnání rychlosti

Náhodná pole



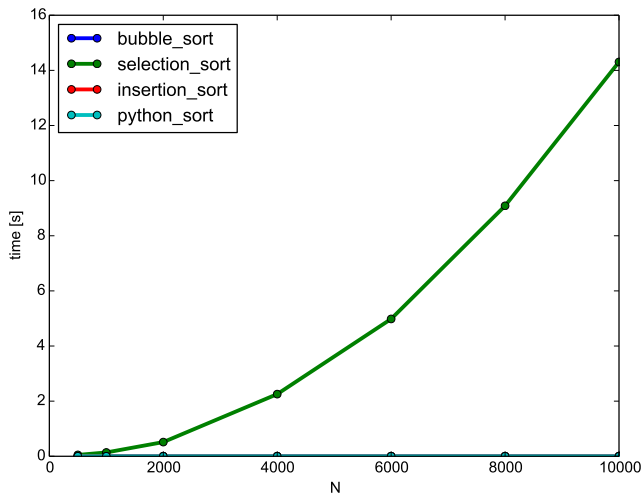
Porovnání rychlosti

Náhodná pole



Porovnání rychlosti

Setříděné pole



Porovnání rychlosti

Setříděné pole

