Algoritmy a programování

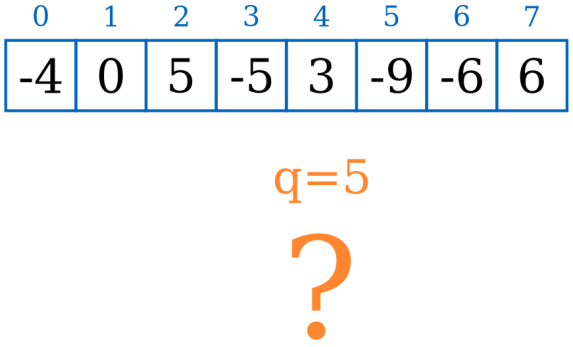
Algoritmy vyhledávání a řazení

Vojtěch Vonásek

Department of Cybernetics
Faculty of Electrical Engineering
Czech Technical University in Prague

Vyhledávání

- Vstupem jsou hodnoty $x_0, x_1, \ldots x_{n-1}$, a dotaz (query) q



- Úkolem je zjistit, jestli existuje $x_i = q$
- Další varianty:
  - zjistit, pro které i platí, že $x_i = q$
  - zjistit nejmenší/největší i pro které platí, že $x_i = q$
  - zjistit všechna i pro které platí, že $x_i = q$
- Vyhledávání je součástí mnoha aplikací
  - Součást algoritmů (prohledávání grafu, stavového prostoru, hraní her...)
  - Má obchod zboží, které hledáte?
  - Hledání v textu, hledání souborů podle názvu, ...

Vyhledávání

Vstup

- Prvky x_i budeme reprezentovat polem `x = [ ... ]`

Výstup

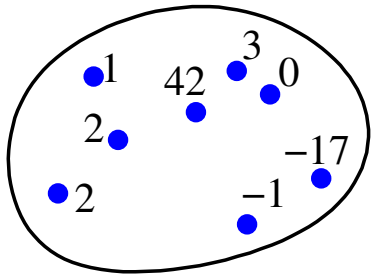
- Hodnota (a datový typ) podle konkrétní varianty
- Zjistit, jestli pro nějaké i platí, že $x_i = q$
  - výstup je datový typ `bool` $\rightarrow$ `True/False`
- Zjistit nejmenší i pro které platí, že $x_i = q$
  - výstup je datový typ `int` — index prvku
- Zjistit všechna i pro které platí, že $x_i = q$
  - výstup je pole indexů

Vyhledávání

- Způsob vyhledávání závisí na tom, co víme o datech

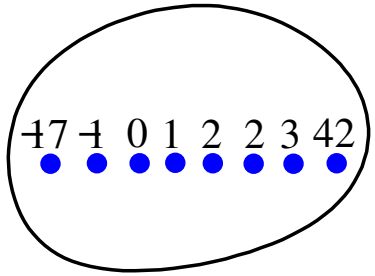
Prvky nejsou organizované (nebo to nevíme)

- Jsou v datové struktuře v libovolném (neznámém) pořadí
- Jediný způsob vyhledávání je projít všechny prvky a porovnat s hledaným
- Lineární (sekvenční) vyhledávání



Prvky jsou organizované (a víme jak)

- Například jsou seřazeny dle velikosti
- Lze použít efektivnější metody, než lineární vyhledávání
- Například Binary search/půlení intervalu



Lineární vyhledávání

- Základní varianta: zjistit, zda existuje $x_i = q$
- Lineární vyhledávání: procházíme jednotlivé buňky dokud nenarazíme na hledaný prvek, nebo na konec pole
- Používáme for cyklus + in (protože není potřeba index prvku)

```
1 def findItem(x,query): #x is list
2     for item in x:
3         if item == query:
4             return True
5     return False
6
7 a = [0,1,0,2]
8 print( findItem(a, 0 ) )
9 print( findItem(a, "0") )
```

```
True
False
```

Lineární vyhledávání

- Zjistit, pro které i platí, že $x_i = q$
- Hledáme index i, použijeme for + range

```
1 def findItemIndex(x,query): #x is list
2     for i in range(len(x)):
3         if x[i] == query:
4             return i
5     return -1 #item not found
6
7 a = [0,1,0,2]
8 print( findItemIndex(a, 0 ) )
9 print( findItemIndex(a, "0") )
```

```
0
-1
```

- Jakou hodnotou indikovat, že hledaný prvek neexistuje?
- Taková hodnota nesmí být zaměnitelná s možnou správnou odpovědí
  - Možná správná odpověd leží v rozsahu $0 \ldots n-1$
  - Pole můžou být různě dlouhá (teoreticky je n neomezené), takže kladná celá čísla nejsou vhodná pro indikaci, že prvek nebyl nalezen
  - Vhodná indikace nenalezení prvku: `-1` nebo `None`

Lineární vyhledávání

- Najít všechna *i* pro které platí, že $x_i = q$

```
1  def findItems(x,query):
2      result = []
3      for i in range(len(x)):
4          if x[i] == query:
5              result.append(i)
6      return result
7
8  a = ["a","b","aa","a","bb","a"]
9  print( findItems(a, "a") )
10 print( findItems(a, "A") )
```

```
[0, 3, 5]
[]
```

- Vrací pole indexů kde leží hledaný prvek, nebo `[]`

Vlastnosti

- Lineární vyhledávání funguje pro obecné pole
  - prvky mohou být v libovolném pořadí
- Můžeme prohledávát pole různých datových typů
  - obecně jakékoliv typy, pro které máme operátor ==
- Složitost $\mathcal{O}(n)$ (přednáška Složitost algoritmů)

```
1 a = [1, "ahoj", None, 4.5/3, "a", True ]
2 print( findItemIndex(a, None) )
3 print( findItemIndex(a, -1) )
```

```
2
-1
```

Vyhledávání v Pythonu

- Python umožňuje zjistit existenci prvku (v poli, listu, stringu, atd . . .) operátorem `in`

```
1 a = [1,2,3]
2 if 1 in a:
3     print("found")
4 else:
5     print ("not␣found")
6
7 print ( "␣abc␣" in "␣ABCabc␣" )
```

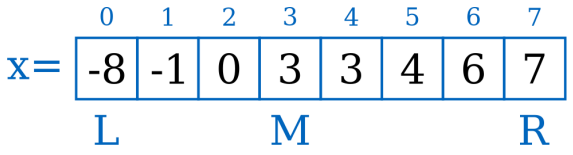
```
found
False
```

- Složitost $\mathcal{O}(n)$ pokud jsou data v poli
- Složitost $\mathcal{O}(1)$ pokud jsou data v dictionary (tzv. slovník)

Binary search

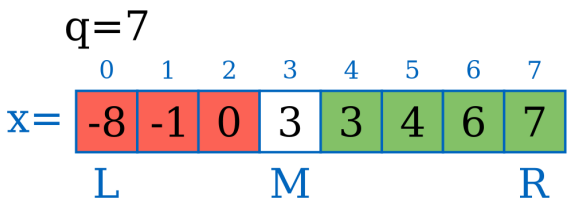
Hledání prvku v seřazeném poli: $x_i \leq x_{i+1}$ pro všechna i

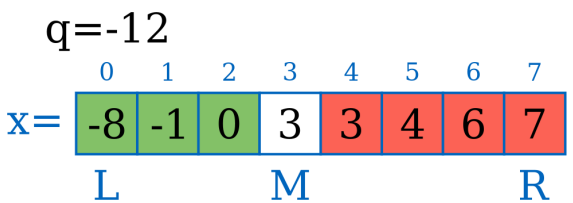
- První (nejmenší) prvek je L (left), poslední (největší) je R (right)
- Určíme prvek mezi nimi: M = (L+R) // 2
- Úlohu nalezení q v intervalu L až R převedeme na úlohu nalezení q v jednom z intervalů L,M nebo M,R



Určení nového intervalu hledání

- Pokud $q = x_M$, našli jsme.
- Pokud $x_M < q$, určitě víme, že nemá smysl hledat v části L,M (neboť tato část obsahuje menší čísla než q), a naopak má smysl hledat v M+1,R
- Pokud $x_M > q$, určitě nemá smysl hledat v části M,R (obsahuje větší prvky než q), ale má smysl hledat v L,M-1



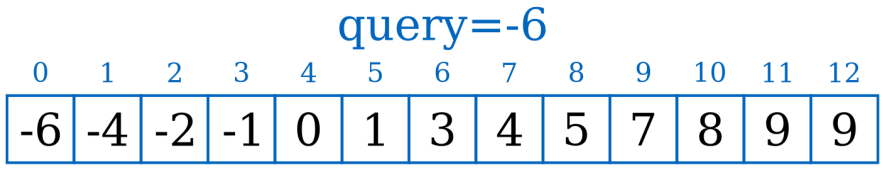


- Úlohu jsme zjednodušili na hledání v polovině původního intervalu

Binary search

- Předpoklad: vzestupně seřazené pole *x*, hledáme prvek query

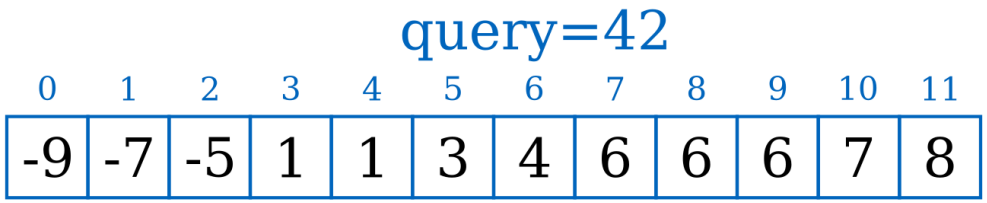
```
1  def binarySearch(x, query):
2      L = 0
3      R = len(x)-1
4      while L <= R:
5          M = (L+R) // 2
6          if x[M] == query:
7              return M
8          if x[M] > query:
9              R = M-1
10         else:
11             L = M+1
12     return -1
```



- Program vrací index nalezeného prvku, nebo -1

Binary search

- Předpokládá seřazené pole
- Aplikace Binary search na neseřazeném pole může dávat špatné výsledky
- Složitost $\mathcal{O}(\log n)$



Řazení

- Vstup: posloupnost x_i a způsob porovnání '$>$' nebo '$<$'
- Výstup:
  - vzestupně seřazená posloupnost, tj. pro všechna i platí $x_i \leq x_{i+1}$
  - sestupně seřazená posloupnost, tj. pro všechna i platí $x_i \geq x_{i+1}$
- Příklad: `[10, 27, -1, 0, 10 ]`
  - Vzestupně: `[-1,0,10,10,27]`
  - Sestupně: `[27, 10,10, 0, -1]`

Terminologie

- Řazení: úprava pořadí prvků tak, aby byly seřazené
- Třídění: rozdělení prvků do skupin dle nějakých atributů
- Pojmy řazení a třídění se v často používají ve významu řazení

Řazení

- Jak zjistíme, že je pole seřazené?

```
1  def isSorted(x):
2      for i in range(len(x)-1):
3          if not x[i] <= x[i+1]:
4              return False
5      return True
6
7  a = [10,-1,2,0,0]
8  print(a, isSorted(a))
9  a.sort()
10 print(a, isSorted(a))
```

```
[10, -1, 2, 0, 0] False
[-1, 0, 0, 2, 10] True
```

- Upravte program pro detekci sestupně seřazeného pole

Odvození základního řazení: BubbleSort

Úkol: seřadit (vzestupně) pole `x = [ ... ]`

Postup

- Nejprve seřadíme dvojici proměnných
- Tento postup rozšíříme na všechny po sobě jdoucí dvojice v poli
- Aplikujeme opakovaně na celé pole
- Zkusíme optimalizovat — odstranit zbytečné operace

Seřazení dvou proměnných

```
1 a = 30
2 b = -1
3 if a > b:
4     a,b = b,a
```

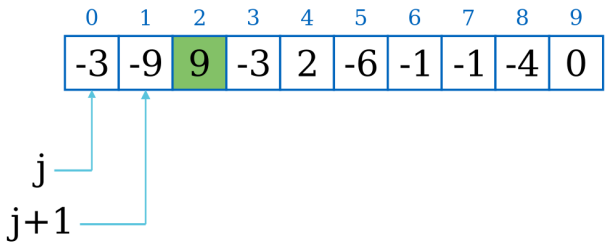
- `a,b = b,a` je tzv. Pythonovská výměna
- `a` obsahuje minimum, `b` obsahuje maximum z obou čísel

Odvození BubbleSort

- Porovnáme prvek x_i s jeho následníkem x_{i+1}, a pokud je větší, tak je vyměníme
- Tento postup provedeme pro všechna i
- Jakou vlastnost má pole po této operaci?

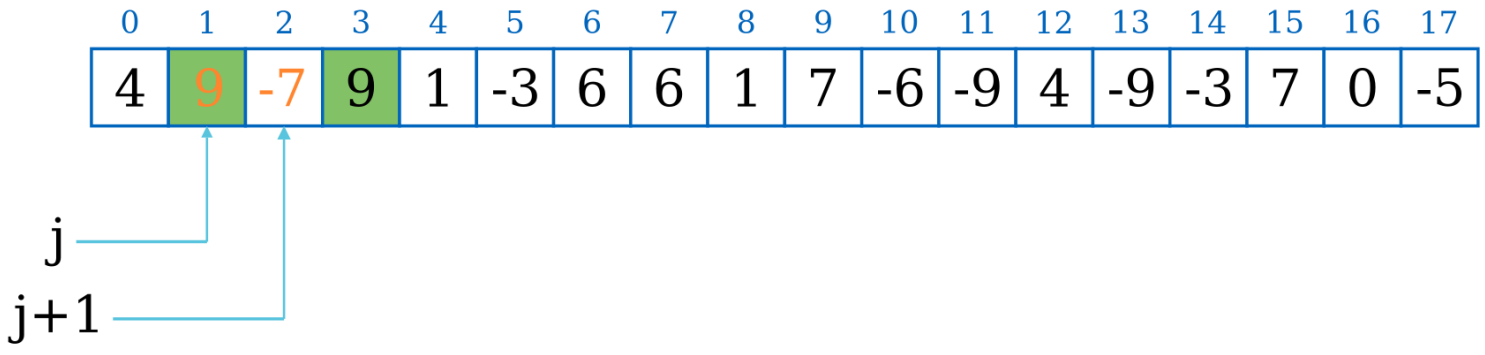
```
1 x = [ 1000,-10,0,1,-3,4,4]
2 print(x)
3
4 for j in range(len(x)-1):
5     if x[j] > x[j+1]:
6         x[j],x[j+1] = x[j+1], x[j]
7
8 print(x)
```

```
[1000, -10, 0, 1, -3, 4, 4]
[-10, 0, 1, -3, 4, 4, 1000]
```



Odvození BubbleSort

- Nejvyšší prvek je (určitě) na konci pole
- Pokud je jich více, je alespoň jeden z nich na konci pole

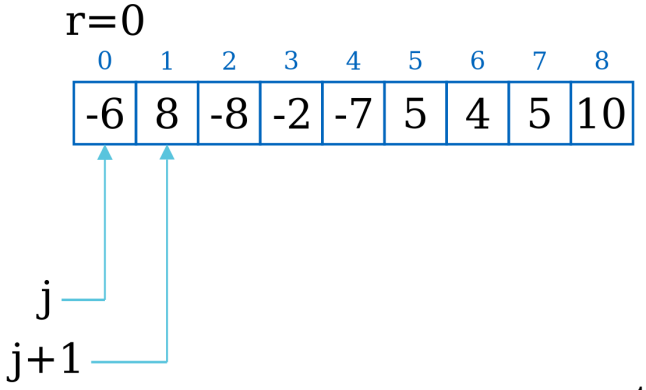


BubbleSort: naive version

- Opakujeme předchozí postup tolikrát, kolik je prvků pole:
  - Seřadíme všechny po sobě jdoucí dvojice
- Výsledkem je seřazené pole

```
1  def bubbleSort(x): #x is list
2      for r in range(len(x)): # r is not used
3          for j in range(len(x)-1):
4              if x[j] > x[j+1]:
5                  x[j],x[j+1] = x[j+1], x[j]
6
7  a = [ 10,-10,0,1,-3,4,4]
8  print(a)
9  bubbleSort(a)
10 print(a)
```

```
[10, -10, 0, 1, -3, 4, 4]
[-10, -3, 0, 1, 4, 4, 10]
```

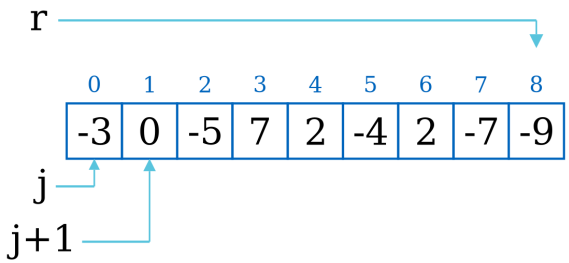


BubbleSort: vylepšení I

- Předchozí postup iteruje pokaždé přes všechny prvky
- Po první iteraci (`r=0`) je na posledním místě nejvyšší prvek
- Po druhé iteraci (`r=1`) je na předposledním místě druhý nejvyšší prvek
- atd ...
- Vylepšení: `j` iterovat do `r`

```
1  def bubbleSort(x): #x is list
2      for r in range(len(x)-1,0,-1): #r is used
3          for j in range(r):  #j = 0..r-1
4              if x[j] > x[j+1]:
5                  x[j],x[j+1] = x[j+1], x[j]
6
7  a = [ 10,-10,0,1,-3,4,4]
8  print(a)
9  bubbleSort(a)
10 print(a)
```

```
[10, -10, 0, 1, -3, 4, 4]
[-10, -3, 0, 1, 4, 4, 10]
```



BubbleSort: vylepšení II

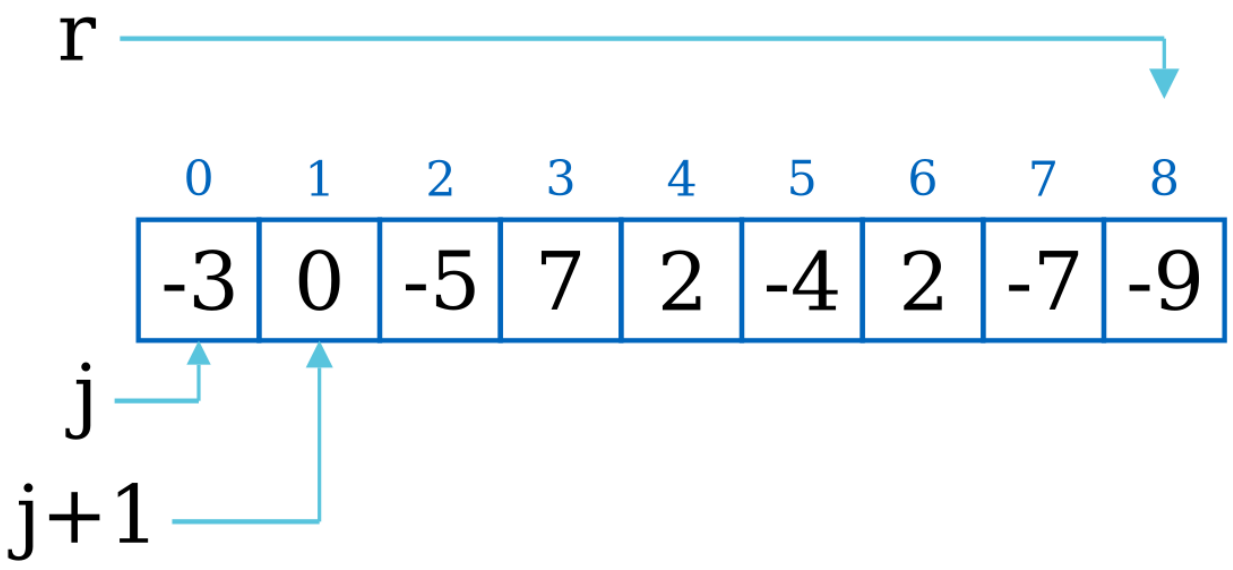
- Pokud nedojde k výměně, je pole seřazené, není třeba dál pokračovat

```
1  def bubbleSort(x): #x is list
2      for r in range(len(x)-1,0,-1): #r is used
3          change = False
4          for j in range(r):  #j = 0..r-1
5              if x[j] > x[j+1]:
6                  x[j],x[j+1] = x[j+1], x[j]
7                  change = True
8          if not change:
9              break
10 a = [ 10,-10,0,1,-3,4,4]
11 print(a)
12 bubbleSort(a)
13 print(a)
```

```
[10, -10, 0, 1, -3, 4, 4]
[-10, -3, 0, 1, 4, 4, 10]
```

Vlastnosti

- Řazení na místě (in-place)
- Prvky ve vstupním poli mohou být v jakémkoliv pořadí
- Směr řazení určuje operátor porovnání (< nebo >)
- Složitost $\mathcal{O}(n^2)$



- Řazení “vkládáním”
- Je založeno na postupném vkládání nových prvků x do již seřazeného pole na pozici, která neporušuje řazení
- Najde se pozice j tak, že

$$x[j-1] \leq x < x[j]$$

- Pokud je takových pozic j více, použije se nejvyšší z nich

0	1	2	3	4	5	6	7	8	9	10
-8	-8	-7	-6	-6	-5	-5	-2	0	1	9

1

InsertionSort

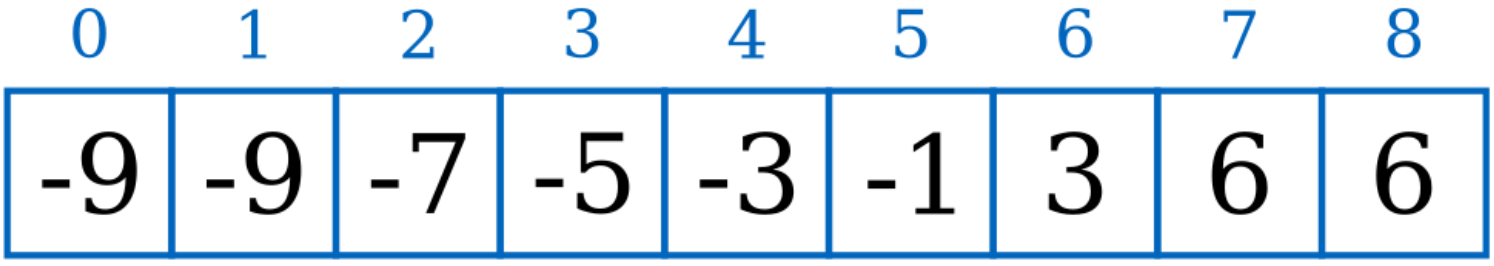

```
1  def insertionSort(x):
2      for r in range(1, len(x)):
3          j = r
4          while j > 0 and x[j-1] > x[j]:
5              x[j-1], x[j] = x[j], x[j-1]
6              j-=1
7
8  a = [10,9,0,0,5,1]
9  insertionSort(a)
10 print(a)
```

```
[0, 0, 1, 5, 9, 10]
```

0	1	2	3	4	5	6	7	8
-3	1	-1	9	3	-4	-1	6	-2

- Opakované vkládání prvků na správné pozice
- r — index prvku, který vkládáme do pole

InsertionSort: seřazené pole



InsertionSort: opačně seřazené pole

0	1	2	3	4	5	6	7	8
6	5	3	1	1	-1	-8	-8	-9

InsertionSort: optimized

- Modifikace InsertionSort, která minimalizuje počet výměn prvků
- Najdeme, na kterou pozici se má vložit hodnota `x[r]`
- Při hledání se neprovádí výměna prvků, pouze jejich posun

```
1  def insertionSortOptimized(x):
2      for r in range(1, len(x)):
3          value = x[r]
4          j = r-1
5          while j >= 0 and x[j] > value:
6              x[j+1] = x[j]
7              j-=1
8          x[j+1] = value
9
10 a = [10,9,0,0,5,1]
11 insertionSortOptimized(a)
12 print(a)
```

```
[0, 0, 1, 5, 9, 10]
```

Vlastnosti

- Řazení na místě (in-place)
- Vhodné, pokud je vstup již částečně seřazené
- Setříděné pole je detekováno v n krocích
- Složitost $\mathcal{O}(n^2)$
- Nepoužívá výměnu prvků jako BubbleSort, ale jejich posun (rychlejší)

```
1  def insertionSortOptimized(x):
2      for r in range(1, len(x)):
3          value = x[r]
4          j = r-1
5          while j >= 0 and x[j] > value:
6              x[j+1] = x[j]
7              j-=1
8          x[j+1] = value
9
10 a = [10,9,0,0,5,1]
11 insertionSortOptimized(a)
12 print(a)
```

```
[0, 0, 1, 5, 9, 10]
```

SelectionSort

- Procházíme prvky zleva doprava, $r = 0, \ldots n-1$
- Prvek `x[r]` musí mít hodnotu $\min(x_r, x_{r+1}, \ldots, x_{n-1})$

```
1  def selectionSort(x): #x is list
2      for r in range(len(x)-1):
3          minidx = r
4          for j in range(r+1,len(x)):
5              if x[j] < x[minidx]:
6                  minidx = j
7          x[minidx], x[r] = x[r], x[minidx]
8
9  a = [7,42,-3,0,5,1,1]
10 selectionSort(a)
11 print(a)
```

```
[-3, 0, 1, 1, 5, 7, 42]
```

0	1	2	3	4	5	6	7	8
9	5	-6	1	-7	-9	-4	2	6

SelectionSort

- Řazení na místě (in-place)
- Složitost $\mathcal{O}(n^2)$
- Prakticky je rychlejší než BubbleSort (používá méně výměn prvků)

0	1	2	3	4	5	6	7	8
9	5	-6	1	-7	-9	-4	2	6

Řazení (v Pythonu)

- Algoritmy řazení používají operátor < pro určení pořadí položek
- V Pythonu pro základní datové typy (int, float, string, pole, . . .)

```
1 print( 1 < 1.2 )
2 print( "Humpolec" < "Pelhrimov" )
3 print( "AAA" < "AAAA" )
4 print( [1,2,3] < [1,2,3,4] )
5 print( "XYZ" == "ABC" )
```

```
True
True
True
True
False
```

Řazení (v Pythonu)

Porovnání stringů: `string1 < string2`

- Prochází oba stringy zleva, porovnává znaky dle pořadí v UTF-8 (`ord()`)
- Pokud jsou znaky stejné, postupuje se na další znak
- Pokud se dojde na konec jednoho stringu, je ten kratší považován za menší
- Case-sensitive
- Nevhodné pro porovnání čísel!
- Pokud víme, že string obsahuje čísla, je nutné pro správné porovnání přetypovat na int nebo float

```
1 print("a" < "b")
2 print("b" < "a")
3 print("aa" < "a")
4 print("pes" < "les")
5 print("Pes" < "les")
6 print("123" < "124")
7 print("0123" < "124")
8 print("1230" < "124") #pozor!
```

```
True
False
False
False
True
True
True
True
```

Řazení

- Defaultní operátor < není vhodný např. pro porovnání českých slov
- Problém např. s písmenem “ch” vs “c”
- Písmeno “ch” je v čestině jedno písmeno, ale v UTF-8/ASCII jsou to dva znaky
- Pokud slovo obsahuje “ch”, je řazeno podle “c”

```
1 print("chleba" < "cihla")
2 a = ["cizinec", "chleba", "cihla"]
3 a.sort()
4 print(a)
```

```
True
['chleba', 'cihla', 'cizinec']
```

Řazení s vlastním porovnáním

- Pokud máme vlastní datové typy, nebo předepsanou formuli řazení
- Je třeba nahradit operátor < porovnávací funkcí

```
1 #case insensitive comparison
2 def isSmaller(a,b): #a,b are strings
3     return a.lower() < b.lower()
```

- Funkci `isSmaller(a,b)` použijeme místo operátoru `a<b` v algoritmech řazení

Řazení s vlastním porovnáním


```
1  #case insensitive comparison
2  def isSmaller(a,b): #a,b are strings
3      return a.lower() < b.lower()
4
5  def bubbleSort(x): #x is list
6      for r in range(len(x)-1,0,-1): #r is used
7          for j in range(r):  #j = 0..r
8              if isSmaller(x[j+1], x[j]):
9                  x[j],x[j+1] = x[j+1], x[j]
10
11 a = ["Dog","cat","fish","alligator"]
12 a.sort() #default python case-sensitive sort
13 print(a)
14 bubbleSort(a) #our case-insensitive
15 print(a)
```

```
['Dog', 'alligator', 'cat', 'fish']
['alligator', 'cat', 'Dog', 'fish']
```

Řazení v Pythonu

- Algoritmy řazení jsou součástí Pythonu

sorted()

- Funkce `sorted()` vrací seřazené pole, původní pole je zachováno
- Je potřeba pamět pro výsledek

```
1 a = [1/1,1/2,1/4,1/5]
2 b = sorted(a)
3 print(a)
4 print(b)
```

```
[1.0, 0.5, 0.25, 0.2]
[0.2, 0.25, 0.5, 1.0]
```

sort()

- In-place řazení, vstupní pole je změněno

```
1 a = [1/1,1/2,1/4,1/5]
2 a.sort()
3 print(a)
```

```
[0.2, 0.25, 0.5, 1.0]
```

Řazení v Pythonu: vlastní porovnání

- `sort()` a `sorted()` lze volat s vlastní metodou, která vrací "klíč" pro porovnání
- Argument `key` nastavíme na jméno funkce s jedním argumentem
- `sort()` zavolá na každou hodnotu vstupního pole tuto funkci
- Hodnoty v poli jsou řazeny na základě výstupů zadané funkce

```
1 def mykey(a): #a is string
2     return a[-1]
3
4 a = ["Dog", "Snake","DOG", "SNAKE", "Albatros"]
5 a.sort() #default python case-sensitive
6 print(a)
7 a.sort(key=mykey)
8 print(a)
```

```
['Albatros', 'DOG', 'Dog', 'SNAKE', 'Snake']
['SNAKE', 'DOG', 'Snake', 'Dog', 'Albatros']
```

Řazení v Pythonu: opačné řazení

- Změna řazení je ovlivněna argumentem `reverse`

```
1 a = ["Dog", "Snake","DOG", "SNAKE", "Albatros"]
2 a.sort(key=len)
3 print(a)
4 a.sort(key=len, reverse=True)
5 print(a)
```

```
['Dog', 'DOG', 'Snake', 'SNAKE', 'Albatros']
['Albatros', 'Snake', 'SNAKE', 'Dog', 'DOG']
```

In-place

- Vytváří výsledek s využitím paměti vstupních dat (plus malá pamět nezávislá na velikosti vstupu pro pomocné proměnné)
- Výhoda: šetření paměti
- Nevýhoda: vstupní data jsou změněna (ne vždy je žádoucí)
- Opakem jsou not-in-place (out-of-place) metody

```
1 a = ["bubble", "insertion", "selection", "quick"]
2
3 a.sort() #in-place sort
4 print(a)
5
6 b = sorted(a,reverse=True) #out-of-place sort
7 print(a)
8 print(b)
```

```
['bubble', 'insertion', 'quick', 'selection']
['bubble', 'insertion', 'quick', 'selection']
['selection', 'quick', 'insertion', 'bubble']
```

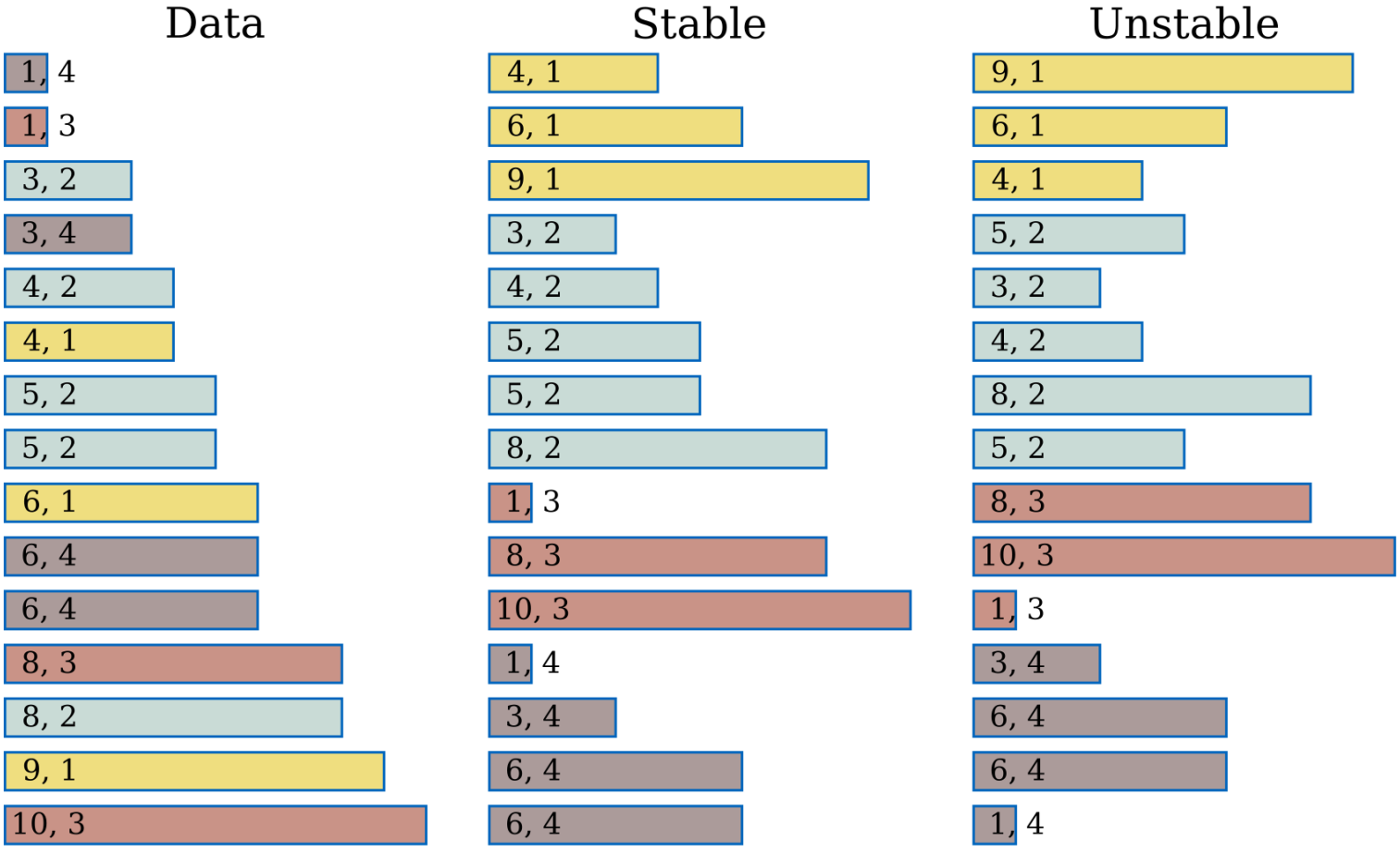
Stabilní vs. nestabilní řazení

- Řadící algoritmus je stabilní, pokud se položky se stejným klíčem objeví na výstupu ve stejném pořadí, jako jsou na vstupu

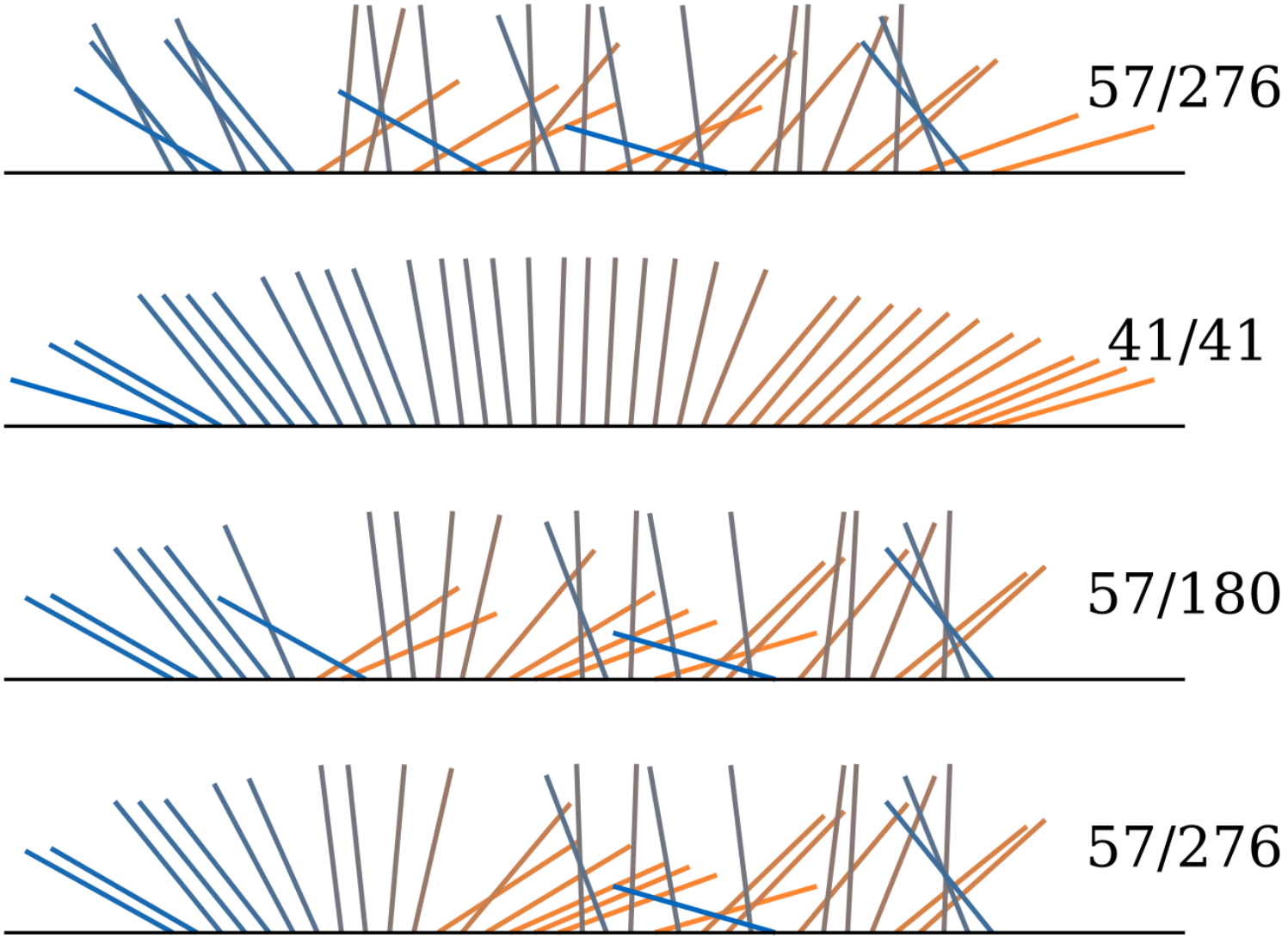
Data	Nestabilní	Stabilní
['Marienville', 2]	['Placida', 1]	['Placida', 1]
['Naxera', 3]	['Wyanet', 1]	['Wyanet', 1]
['Placida', 1]	['Tannersville', 2]	['Marienville', 2]
['Sarahann', 4]	['Verdel', 2]	['Seitz', 2]
['Seitz', 2]	['Wittensville', 2]	['Tannersville', 2]
['Soudersburg', 3]	['Seitz', 2]	['Verdel', 2]
['Tannersville', 2]	['Marienville', 2]	['Wittensville', 2]
['Tokeland', 3]	['Naxera', 3]	['Naxera', 3]
['Tumacacori', 4]	['Soudersburg', 3]	['Soudersburg', 3]
['Uehling', 4]	['Tokeland', 3]	['Tokeland', 3]
['Verdel', 2]	['Yreka', 4]	['Sarahann', 4]
['Wardensville', 4]	['Tumacacori', 4]	['Tumacacori', 4]
['Warners', 4]	['Wardensville', 4]	['Uehling', 4]
['Wittensville', 2]	['Warners', 4]	['Wardensville', 4]
['Wyanet', 1]	['Sarahann', 4]	['Warners', 4]
['Yreka', 4]	['Uehling', 4]	['Yreka', 4]

Stabilní vs. nestabilní třídění

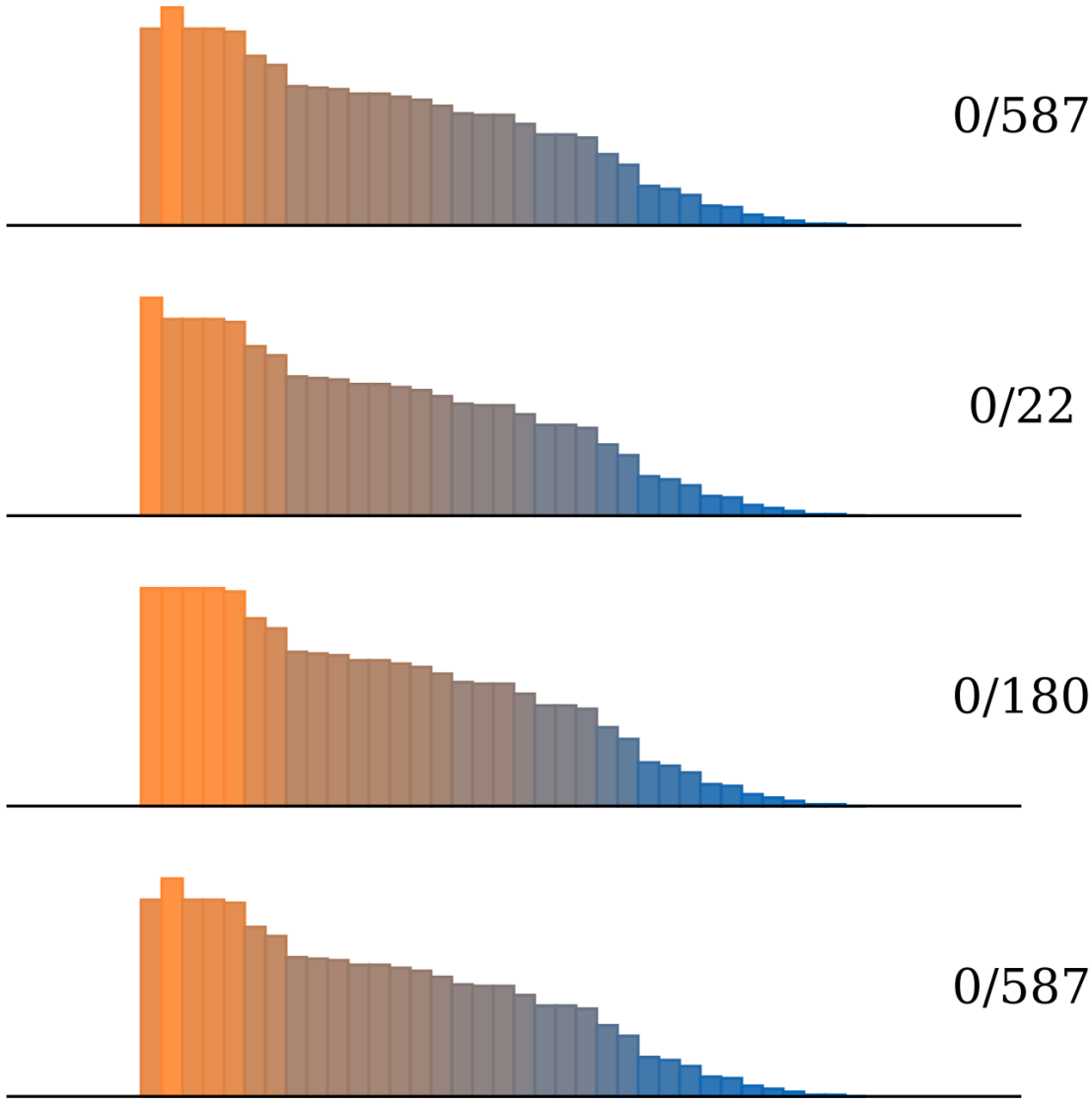
- Řadící algoritmus je stabilní, pokud se položky se stejným klíčem objeví na výstupu ve stejném pořadí, jako jsou na vstupu



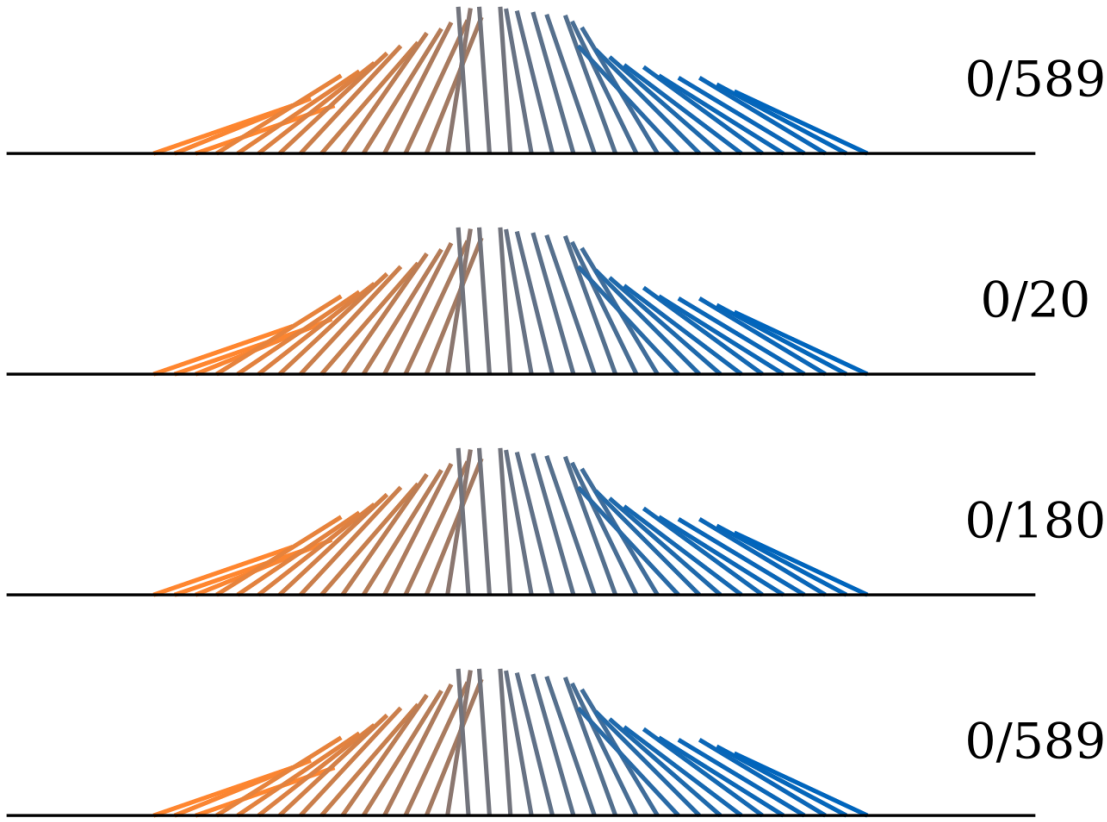
Visualizace



Opačně seřazené pole Poznáte algoritmy podle průběhu?



Opačně seřazené pole Poznáte algoritmy podle průběhu?



Opačně seřazené pole Poznáte algoritmy podle průběhu?