# Třídící algoritmy v jazyce Python

**Teorie (praxe níže):**

## Využití:

* přeorganizování skupiny prvků podle nějakého klíče (primární, sekundární)
* usnadňuje další vyhledávání mezi prvky (binární vyhledávání)
* usnadňuje užití dat pro ruční zpracování (žebříček)
* pro statistické účely (nalezení mediánu)

## Řazení:

* řadíme vzestupně x sestupně
* řazení číselné x lexikografické (abecední)

## Přístup ke třídění:

* algoritmy se odlišují – časovou a prostorovou složitostí, způsobem procházení položek
* lze je dělit podle následujících kritérií:

- výpočetní a časová náročnost

- využití paměti (pomocné proměnné, pole atd.)

- rekurzivnost (využívá sám sebe)

- stabilita (položky se stejným klíčem seřazeny vždy stejně)

- porovnávání třídicího algoritmu

- obecná technika

- přizpůsobivost

* rozlišujeme:

- vnitřní řazení – vyžaduje, aby všechna řazená data byla uložena v operační paměti

- vnější řazení – v operační paměti jen nějaká část dat

* rozlišujeme:

- přirozené – seřazenou posloupnost třídí rychleji než neseřazenou

- nepřirozené – nezáleží na původní posloupnosti

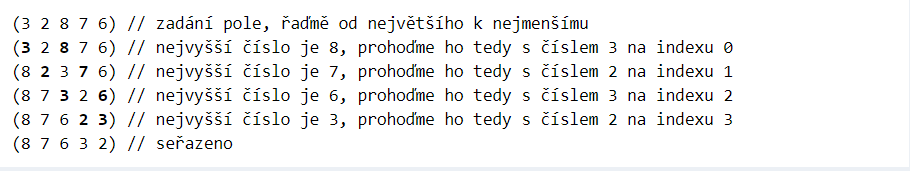
## Typy řazení:

* výběrem – najde se nejmenší (největší) položka, uloží se na konec (začátek).
* vkládáním – postupně se každá položka vloží na správné místo v seřazeném souboru
* záměnou – najde se neseřazená dvojice prvků a zamění se jejich pořadí
* slučováním – soubor se rozdělí na části, které se seřadí, tyto části se pak sloučí

## Select sort

*Select sort* (selection sort) – řazení **výběrem** (viz výše)

- vizualizace <https://commons.wikimedia.org/wiki/File:Selection-Sort-Animation.gif>



## Bubble sort

*Bubble sort* – řazení **záměnou**, porovná vedlejší prvky a v případě nesprávného seřazení je prohodí

- prakticky neefektivní, využívá se pro výukové účely a v nenáročných aplikacích

- snadná implementace

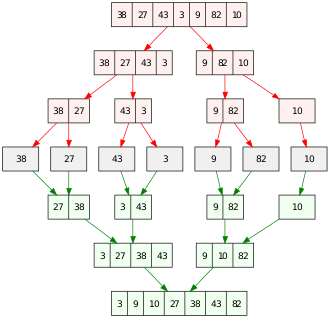
- vizualizace <https://commons.wikimedia.org/wiki/File:Bubble-sort-example-300px.gif>

## Merge sort

- *Merge sort* (řazení **slučováním**) – „rozděl a panuj“

- rozdělí neseřazenou množinu na dvě podmnožiny, ty setřídí a pak slévá dohromady ( rekurze)

- autor John von Neumann



## Insert sort

- *Insert sort* (insertion sort) – řazení **vkládáním** (viz výše)

- u téměř seřazeného pole malá časová složitost

- používán jako doplněk k řadicím algoritmům typu rozděl a panuj

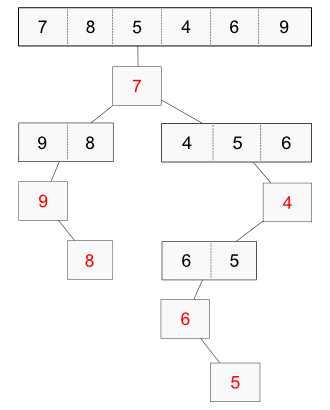
- vizualizace https://www.[youtube](https://www.youtube.com/watch?v=OGzPmgsI-pQ).com/watch?v=OGzPmgsI-pQ

## Quicksort

- *Quicksort*

- zvolíme jeden prvek – pivot, všechny ostatní setřídíme např. nalevo větší než pivot, napravo menší, poté postup zopakujeme pro obě tyto podmnožiny (bez pivota, ten je již správně)

- výkonnost ovlivňuje volba správného pivota – (buď fixně – první, poslední, nebo náhodně)



- vizualizace <https://www.youtube.com/watch?v=PgBzjlCcFvc>

[https://www.itnetwork.cz/movies/quick\_sort.mp4https://www.itnetwork.cz/movies/quick\_sort.mp4](https://www.itnetwork.cz/movies/quick_sort.mp4https:/www.itnetwork.cz/movies/quick_sort.mp4)

- další třídicí algoritmy – Heap sort, Shell sort; Random sort…

**Asymptotická složitost**

Asymptotická složitost je způsob klasifikace počítačových algoritmů. Určuje operační náročnost algoritmu tak, že zjišťuje, jakým způsobem se bude chování algoritmu měnit v závislosti na změně velikosti (počtu) vstupních dat. Zapisuje se pomocí tzv. Landauovy notace, nebo také O notace, např. O(N).

Příklady

Časová složitost **O ( N )** (tzv. lineární) říká, že doba trvání práce algoritmu se zvýší přibližně tolikrát, kolikrát se zvýší velikost vstupu.

Časová složitost **O ( N 2 )** se doba trvání průběhu zvyšuje kvadraticky, tedy pokud se zvýší délka vstupu dvakrát, potřebný čas se zvýší čtyřikrát.

U časové složitosti **O ( 1 )** naopak na délce vstupu vůbec nezáleží a potřebný čas je stále stejný.

Obvykle se používá asymptotická **časová** a **prostorová** složitost.

**Časová složitost**

Důležitou vlastností algoritmu je časová náročnost výpočtů provedené podle daného algoritmu.

Ta se nezískává měřením doby výpočtu pro různá data, **ale analýzou algoritmu**, jejímž výsledkem je časová složitost algoritmu. Nemůžeme např. prohlásit: "Tomuto algoritmu to trvá 5 vteřin". To proto, že mimo jiné **záleží na rychlosti počítače**, na kterém program běží.

Časová složitost algoritmu vyjadřuje **závislost času potřebného pro provedení výpočtu na rozsahu (velikosti) vstupních dat**.

Čas se však neměří v sekundách, ale **počtem provedených operací**, přičemž trvání každé operace se chápe jako bezrozměrná jednotka.

Doba výpočtu obvykle nezávisí jen na rozsahu vstupních dat, ale též na konkrétních hodnotách. Obecně proto rozlišujeme časovou složitost v nejlepším, nejhorším a průměrném případě. Přesné určení počtu operací při analýze složitosti algoritmu však bývá velmi složité. Zvlášť komplikované, ba i nemožné, bývá určení počtu operací v průměrném případě; proto se většinou **omezujeme jen na analýzu nejhoršího případu**.

Zpravidla nás nezajímají konkrétní počty operací pro různé rozsahy vstupních dat n, ale **tendence** **jejich růstu** při zvětšujícím se n.

**Prostorová složitost**

Podobně je tomu i u prostorové složitosti, jen s tou změnou, že **se jedná o potřebné paměťové (prostorové) nároky** v závislosti na délce vstupních dat.

**Stabilita**

O stabilním algoritmu hovoříme v případě, kdy **zachovává předchozí pořadí t**akových prvků, které si jsou podle porovnávacího kritéria rovny. Pokud třídíme pole čísel, nemá to pro nás žádný užitek. Ale ve chvíli, kdy třídíme např. objekty nebo nějaké další kontejnerové struktury, může se nám to velmi hodit. Dejme tomu, že máme zaměstnance seřazené v poli podle abecedy. A my je chceme seřadit podle věku. Pokud řadící algoritmus není stabilní, může se nám stát, že dva stejně staří zaměstnanci (Adam a Zdeněk) budou v pořadí: Zdeněk, Adam. Stabilní algoritmus naopak v případě rovnosti zachová původní pořadí prvků, tedy Adam, Zdeněk. Zaměstnanci se stejným věkem tedy budou ještě seřazeni podle abecedy.

**Praxe:**

## Select Sort

Setřídění pole čísel zadaného uživatelem, a to od nejmenšího po největší číslo.

def selectionSort(nlist):

for fillslot in range(len(nlist)-1,0,-1):

maxpos=0

for location in range(1,fillslot+1):

if nlist[location]>nlist[maxpos]:

maxpos = location

temp = nlist[fillslot]

nlist[fillslot] = nlist[maxpos]

nlist[maxpos] = temp

nlist = [14,46,43,27,57,41,45,21,70]

selectionSort(nlist)

print(nlist)

## Buble Sort

def sort(ip\_arr):

délka = len(ip\_arr)

for i in range(délka - 1):

for j in range (0, délka-i-1):

if ip\_arr[j] > ip\_arr[j+1]:

ip\_arr [j], ip\_arr[j+1] = ip\_arr[j+1], ip\_arr[j]

ip\_arr = [14, 6, 25, 12]

print("Vstupní pole je: ", ip\_arr)

sort(ip\_arr)

print( "Seřazené pole je ")

for i in range(len(ip\_arr)):

print("%d" %ip\_arr[i])

## Insert Sort

def insertionSort(arr):

len(arr)

for i in range ( 1 , len (arr)):

key = arr[i]

j = i - 1

while j>= 0 and key < arr[j] :

arr[j + 1 ] = arr[j]

j-=1

arr[j + 1 ] = key

arr = [ 121 , 11 , 15 , 2 , 7 ]

insertionSort(arr)

print( "Setříděné pole:" )

for i in range ( len (arr)):

print ( "%d" % arr[i])