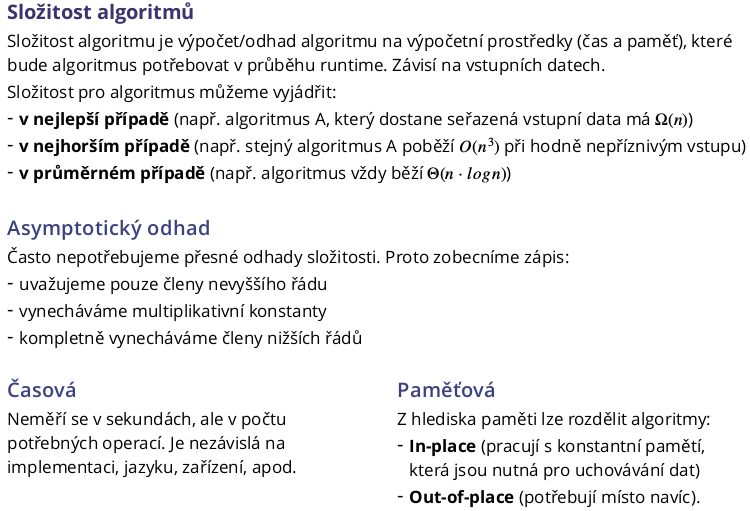
**BI-SPOL-19 Časová a paměťová složitost algoritmů. Algoritmy vyhledávání (sekvenční, půlením intervalu), slučování a řazení (BubbleSort, SelectSort, InsertSort, MergeSort, QuickSort). Dolní mez složitosti řazení v porovnávacím modelu. Řazení v lineárním čase**

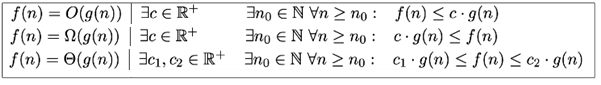
BI-PA1 + BI-AG1

### Časová a paměťová složitost algoritmů

* Abychom byli schopni nějakým způsobem měřit kvalitu programu a byli schopni alespoň odhadnout, kdy program skončí, zavádíme složitost.
* Složitost rozlišujeme na časovou a paměťovou a **vyjadřujeme ji jako funkci (případně odhad) závislou na velikosti vstupních dat**.
* Časovou složitost počítáme počtem provedených operací. – vyjadřuje závislost času potřebného pro provedení výpočtu na rozsahu (velikosti) vstupních dat
* Časová složitost je důležitou vlastností algoritmů – k určení složitosti se používá analýza algoritmu.
* Časová složitost nezávisí na implementaci, programovacím jazyku, hardwaru – je to vlastnost algoritmu
* Přesný výpočet bývá obtížný (průměrný případ je nejobtížnější) – spokojíme se s asymptotickým výpočtem nejhoršího případu – O
* Asymptotické chování – trend chování
  + Zajímá nás proto, že lze předpokládat, že budeme zpracovávat stále více dat
  + Roste výkon HW a zpravidla není problém vyrobit HW 2x 3x výkonnější, ale 100000x už ne



Obsah obrázku text

Popis byl vytvořen automaticky

O(n) – maximálně dosahuje n

Ω(n) – minimálně je to n

Θ(n) – průměrně je to n

### Algoritmy vyhledávání

**Sekvenční**

* jednoduché procházení celé struktury a porovnání každé hodnoty s vyhledávanou
* složitost je O(n), protože při nejhorším musí projít všechny prvky

**Půlením intervalu**

* binární vyhledávání
* vyžaduje seřazenou strukturu
* rychlost je O(log n), protože vždycky dělí velikost intervalu na polovinu
* implementace pomocí rekurze nebo iteračně

Obsah obrázku text

Popis byl vytvořen automaticky

### Algoritmy slučování a řazení

**stabilní/nestabilní**: po seřazení zachovává pořadí prvků se stejným klíčem. Tzn. pokud existují prvky a , tak po seřazení budou ve stejném pořadí

**in-place** – nepotřebují prostor navíc

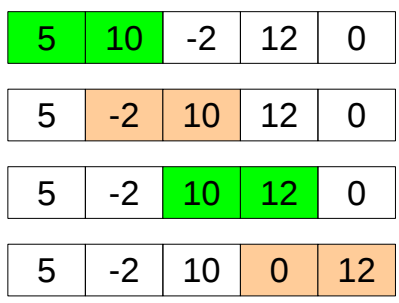
**out-of-place** – při řazení potřebují prostor navíc (pro meziukládání)

**datově citlivý** – množství již seřazených prvků ovlivňuje počet kroků, které musí algoritmus udělat k seřazení celé posloupnosti

**datově necitlivý** – i když dostanou na vstup seřazenou posloupnost, tak provedou stejný počet kroků, jako při jakékoliv jiné permutaci prvků v posloupnosti

#### Bubble sort

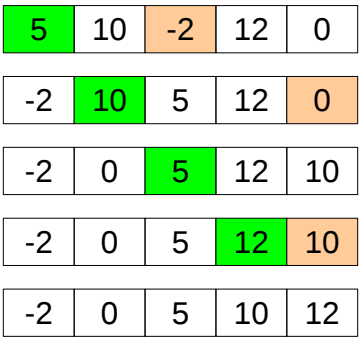
* porovnávání sousedních prvků pole, dokud není seřazený
* postupně zleva doprava (případně opačně) se porovnávají dvojice sousedních prvků a pokud jsou prvky dvojice v nesprávném pořadí, prohodí se
* po prvním průchodu se největší prvek dostane na poslední místo
* pokud se během jednoho průchodu neprohodila žádná dvojice sousedů, už se nikdy žádná neprohodí a řazení skončilo
* vhodný algo. pokud je vstupní posloupnost s velké části seřazená
* časová složitost: O(n^2)
* je in-place, stabilní a datově citlivý

Obsah obrázku text

Popis byl vytvořen automaticky

#### Select sort

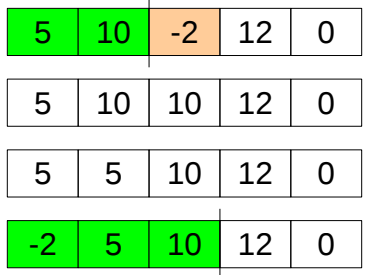
* najdi nejmenší prvek ve zbytku pole
* zaměň ho s prvním (z neseřazené části)
* opakuj s polem o jeden prvek kratším
* Založen na opakovaném vybírání nejmenšího prvku
* Vstupní posloupnost se rozdělí na **levou seřazenou** a **pravou neseřazenou část**
* Na počátku je levá část prázdná a pravá část je celá vstupní posloupnost
* Z neseřazené části vybereme **minimum** a prohodí se s jejím **prvním** prvkem, tj. přímo za seřazenou část
* Po vložení minima na začátek neseřazené části se posune hranice mezi seřazenou a neseřazenou částí o jednu pozici doprava
* Tento postup se provádí, dokud se pravá neseřazená část nezmenší na 1 prvek
* časová složitost: O(n^2)
* je in-place, nestabilní, datově necitlivý

Obsah obrázku text

Popis byl vytvořen automaticky

#### Insert sort

* pole délky 1 je seřazeno
* vezmi první prvek zbytku pole a zařaď ho do již seřazeného pole
* opakuj dokud není seřazeno
* řazení vkládáním
* vstupní posloupnost rozdělíme na levou seřazenou a pravou neseřazenou část
* na počátku tvoří levou část první prvek
* z neseřazené části vždy vezmeme první prvek a vložíme ho na správnou pozici v seřazené části
* po vložení prvku do seřazené části se posune hranice mezi seřazenou a neseřazenou částí o jednu pozici doprava
* to opakujeme, dokud není neseřazená část prázdná
* časová složitost: O(n^2)
* in-place, stabilní, datově citlivý

Obsah obrázku text

Popis byl vytvořen automaticky

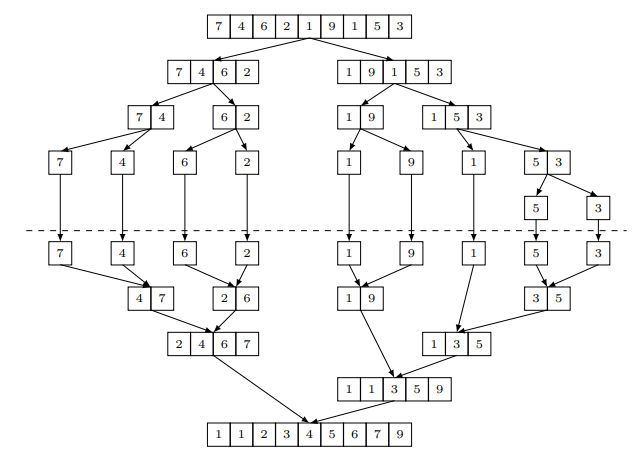
#### Merge sort

* rozděl pole na poloviny a rekurzivně opakuj, dokud ti nezůstane pole o velikosti jednoho prvku (tato pole jsou již seřazená)
* sluč sousední dohromady (stylem vždycky vyber menší prvek ze dvou polí do nového zmergovaného pole)
* založený na slévání seřazených podposloupností
* posloupnost o jednom prvku je seřazená
* časová složitost: Θ(n log n)
* paměťová složitost: Θ(n)
* datově necitlivý, out-of-place, stabilní

Obsah obrázku text

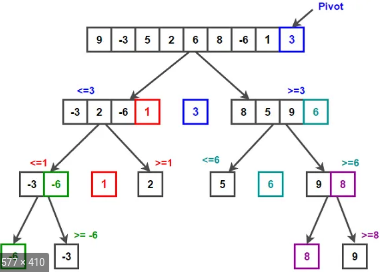
Popis byl vytvořen automatickyObsah obrázku text

Popis byl vytvořen automaticky



#### Quick sort

* Zvolí se **pivot**. Pole se rozdělí na L, S a R, kde
  + L – prvky menší než pivot
  + S – stejně velké
  + R – prvky větší než pivot.
* V seřazené posloupnosti budou vystupovat prvky nejprve z L potom z S a pak z R
* Rekurzivně seřadíme levou část L a pravou část P (S je seřazena)
* Výsledná posloupnost vznikne spojením L, S, P za sebou
* Pivot volíme jako **medián** nebo alespoň **skoromedián**
  + Velikost podproblémů potom klesají exponenciálně
  + Čas. složitost – O(n logn)
* Pokud pivot zvolíme minimum bude složitost O(n^2)
* Pokud zvolíme pivota náhodně – střední hodnota časové složitosti bude O(n logn)
* časová složitost: záleží na volbě pivota, nejhorší O(n^2), jinak O(n log n)
* skoromedián
  + prvek, který leží kdekoliv v prostředních dvou čtvrtinách seřazené posloupnosti
  + bude tedy mít nalevo i napravo vždy nejvýše ¾ \* n prvků
  + nejméně polovina prvků jsou tedy skoromediány
  + pokud vybereme pivota náhodné máme ½ pravděpodobnost, že zvolíme skoromedián
  + pro ověření skoromediánu je potřeba projít celou posloupnost – tedy O(n)



Obsah obrázku text

Popis byl vytvořen automaticky

#### HeapSort

* prvky x1,…,xn vložíme do pole
* no toto pole zavoláme HeapBuild
* Poté n-krát provedeme HeapExtractMin a vrácené hodnoty zapíšeme do pole
* O(n logn)

### Dolní mez složitosti řazení v porovnávacím modelu

**Předpoklady**

* čísla se smí pouze vzájemně porovnat a přesouvat v paměti
* Porovnání cmp(ai , aj) je binární operace, která dokáže porovnat prvky (v O(1))
* algoritmus je deterministický a nepoužívá zdroj náhody a jejich každý krok je jednoznačně určen výsledky kroků předchozích

**Dolní mez vyhledávání**

* problém vyhledávání v porovnávacím modelu: vstupem je číslo n, seřazená posloupnost a1,…,an a hledané x – máme zjistit, jestli se x vyskytuje v posloupnosti
* Dolní mez složitosti problému vyhledávání je Ω(log n) operací porovnání
  + Žádný deterministický algoritmus v porovnávacím modelu nemůže v nejhorším případě nalézt dané číslo v n-prvkové seřazené posloupnosti použitím o(logn) – binární vyhledávání O(logn) je optimální algoritmus pro vyhledávání
  + Důkaz je založen na rozhodovacím stromu – vrcholy mají max. 3 syny s nejméně n + 2 listy a s nejvýše 2n + 1 listy
    - Každý ternární strom hloubky h má nejvýše 3^˛h listů, má strom hloubku nejméně log3n – existuje cesta délky nejméně log3n a proto existuje x, které této cestě odpovídá

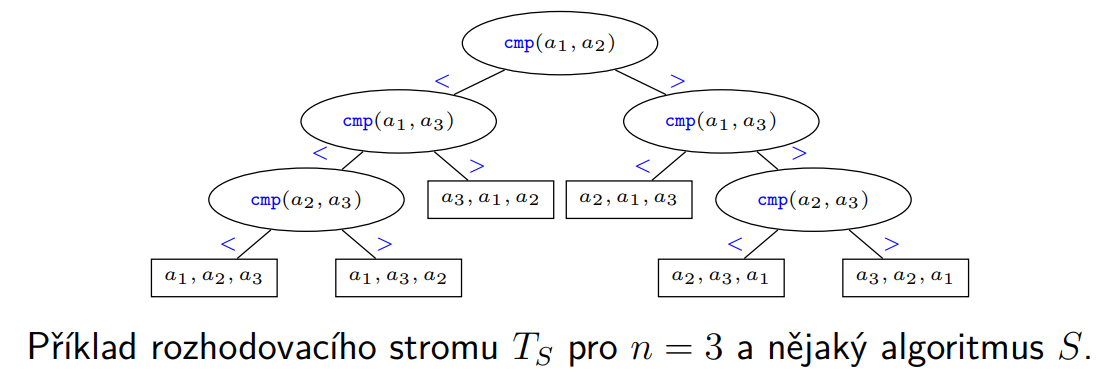
**Dolní mez řazení**

**Obsah obrázku text

Popis byl vytvořen automaticky**

* Každý deterministický algoritmus v porovnávacím modelu, který seřadí n-prvkovou posloupnost, použije v nejhorším případě Ω(n log n) porovnání. – vychází z toho, že řazení je ekvivalentní problému rozpoznání, o kterou permutaci z n! možných se na vstupu jedná – počet potřebných porovnání musí být Ω(n log n) (p9 str.21)
* Obsah obrázku text

  Popis byl vytvořen automaticky
  + Důkaz: postupuje se sestavením rozhodovacího stromu – dva možné výsledky.
    - Strom je binární kořenový strom s n! listy, takže musí mít hloubku nejméně log(n!), tedy Ω(n logn) – proto musí existovat vstupní permutace A, pro kterou S provede Ω(n logn) porovnání



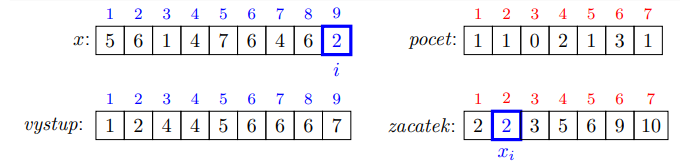
### Řazení v lineárním čase

Tyto modely nepracují v porovnávacím modelu (využívají nějakou speciální vlastnost vstupní posloupnosti)

**Couting sort**

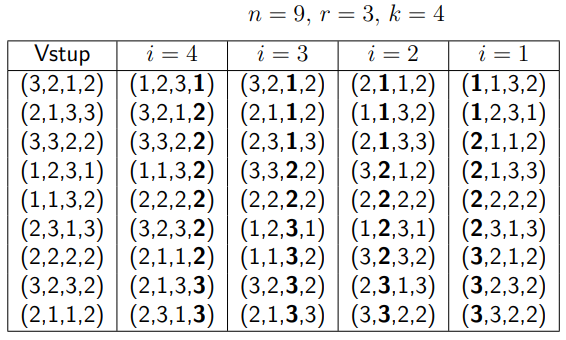
* Algoritmus pro řazení n celých čísel *z* množiny {1,…,r}
* Omezení rozsahu vstupních hodnot je právě tou speciální vlastností
* Nejdříve se projde vstupní pole a spočítá se pro každé číslo z množiny, kolikrát se ve vstupním poli vyskytuje
* Nad tímto polem počtu výskytů se pak vypočte **prefixovaným součtem** pozice, kde budou po seřazení začátky oblastí tvořených stejnými čísli z množiny {1,…,r}
* Jako poslední krok prochází podruhé vstupní pole, pro každý jeho prvek určí pozici, na které se má nacházet po seřazení a na tuto pozici jej přesune
* časová složitost: Θ(n + r) – O(n)
* paměťová složitost: Θ(n + r)
* je stabilní, není in-place a není datově citlivý
* Obsah obrázku text

  Popis byl vytvořen automaticky



**LexCounting sort**

* Úkolem je seřadit k-tice slovníkově (lexikograficky)
* využívá se Counting sort pro řazení poslední pozice
* opakuj s předposlední atd.
* časová složitost: Θ(k · (n + r))
* paměťová složitost: Θ(kn + r)
* na stejném principu je založen RadixSort (řazení víceciferných čísel)
* je stabilní

Obsah obrázku text, snímek obrazovky, osoba

Popis byl vytvořen automaticky