

Třídění/Řazení a vyhledávací algoritmy

Třídění vs. řazení

Třídění je uspořádání objektů podle podobných vlastností. Způsob třídění je vždy závislý na oboru, který s těmito objekty pracuje.

Řazení je způsob uspořádání objektů do specifikovaného pořadí. Řazení může být provozováno podle různých kritérií (abecedně, vzestupně, sestupně).

Oba pojmy bývají často zaměňovány.

Složitost algoritmů

Složitost algoritmů (někdy taky asymptotická složitost) je funkce, která vyjadřuje počet elementárních kroků v závislosti na vstupních datech dané funkce. Značí se O .

Možnosti tříd složitostí: $1 \ll \log(n) \ll n \ll n \cdot \log(n) \ll n^k \ll k^n \ll n! \ll n^n$

Rozdíl mezi jednotlivými třídami složitosti se dá jednoduše pochopit na těchto dvou příkladech. Když máme první algoritmus se složitostí $O(n)$ a druhý algoritmus se složitostí $O(2n)$ stačí nám ten druhý spustit na dvakrát rychlejším stroji a rozdíl je smazán.

Pokud však máme první algoritmus se složitostí $O(n)$ a algoritmus se složitostí $O(n^2)$ bude při různé velikosti stoupat náročnost v závislosti na n , a to několikrát.

Řadicí algoritmy

Bubble sort

Princip:

1. Dostanu pole.
2. Procházím pole, pokud najdu prvek, který je menší než prvek vpravo, prohodím je.
3. Opakuji, dokud není pole seřazeno od největšího po nejmenší (zprava doleva).

Složitost: $O(n^2)$ → za každý prvek pole se pole projde dvakrát.

Ukázka algoritmu:

```
function bubbleSort(array) {
  for (var i = 0; i < array.length - 1; i++) {
    for (var j = 0; j < array.length - 1 - i; j++) {
      if (array[j] < array[j + 1]) {
```

```

        var tmp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = tmp;
    }
}

```

<html>

<script> function test() {

```
var x = [2, 5, 1, 7, 8];
```

```
  bubbleSort(x);
```

}

function bubbleSort(array) {

```

  for (var i = 0; i < array.length - 1; i++) {
    for (var j = 0; j < array.length - 1 - i; j++) {
      printStep(array, step++);
      if (array[j] < array[j + 1]) {
        var tmp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = tmp;
      }
    }
  }
}

```

}

function printStep(array, step) {

```

  var container = document.getElementById("containerbubble");
  container.innerHTML += "Krok " + step + ": "
  for (var index = 0; index < array.length; index++) {
    container.innerHTML += array[index] + " ";
  }
  container.innerHTML += "<br />";
}

```

}

</script>

```

<div id="containerbubble">
</div>
<script src="bubblesort.js"></script>

```

```
<button onclick="test()">Spust' mě!</button>
```

</html>

Insert sort

Princip:

1. Dostanu pole.
2. Procházím pole zleva doprava a vždy každý prvek zařadím na místo podle jeho velikosti.
3. Dostávám pole seřazené zleva doprava (od největšího po nejmenší).

Složitost: Složitost je $O(n^2)$, ale při téměř seřazeném poli se blíží $O(n)$.

Ukázka algoritmu:

```
function insertSort(array) {
  var stepCounter = 0;
  for (var i = 0; i < array.length - 1; i++) {
    var j = i + 1;
    var tmp = array[j];
    while (j > 0 && tmp > array[j - 1]) {
      array[j] = array[j - 1];
      j--;
      array[j] = tmp;
    }
  }
}
```

```
<html> <script> function testInsert() {
```

```
  var x = [1, 5, 6, 7, 8];
```

```
  insertSort(x);
```

```
}
```

```
function insertSort(array) {
```

```
  var stepCounter = 0;
  for (var i = 0; i < array.length - 1; i++) {
    var j = i + 1;
    var tmp = array[j];
    while (j > 0 && tmp > array[j - 1]) {
      printStepIns(array, stepCounter++);
```

```
      array[j] = array[j - 1];
      j--;
      array[j] = tmp;
```

```
    }
```

```
  }
```

```
printStepIns(array, stepCounter++);  
  
}  
  
function printStepIns(array, step) {  
  
    var container = document.getElementById("containerinsert");  
    container.innerHTML += "Krok " + step + ": "  
    for (var index = 0; index < array.length; index++) {  
        container.innerHTML += array[index] + " ";  
    }  
    container.innerHTML += "<br />";  
  
}  
  
</script>
```

```
<div id="containerinsert">  
</div>  
<button onclick="testInsert()">Test me!</button>  
  
</html>
```

Merge sort

Princip:

1. Dostaneme pole.
2. Pole rozdělíme na dvě zhruba stejně velká podpole.
3. Získané podpole dále dělíme až na jednoprvkové pole (získáme n jednoprvkových polí).
4. Jakmile máme jednoprvková pole spojujeme je dohromady tak, aby byly seřazeny.
5. Jakmile máme jenom dvě podmnožiny, porovnáme vždy jednotlivé prvky množiny a vždy ten větší přidáme do finálního pole → postupujeme až máme ve finálním poli prvky od největšího po nejmenší.

Složitost: $O(n * \log(n))$

Ukázka [zde](#).

Quick sort

Princip:

1. Dostaneme pole.
2. Zvolíme si jeden prvek pole (pivot) a rozdělíme zbytek pole na prvky větší než pivot a na prvky menší než pivot (stejně velké prvky mohou být na libovolné straně).
3. Pivota umístíme mezi tyto dvě množiny (pivot je na místě, kam by patřil v seřazeném poli).
4. Kroky opakujeme, dokud nemáme všechny prvky seřazeny.

Složitost: Složitost u quick sortu je hodně závislá na volbě pivotu (resp. pivotů). Pokud je pivot mediánem hodnot, může být složitost až $O(n * \log(n))$, pokud je však pivot největším nebo nejmenším prvkem pole je složitost $O(n^2)$. Pivota můžeme vybrat jako fixní pozici v tabulce (např. vždy poslední, první nebo prostřední prvek) nebo, což se považuje za ideální případ, se vyberou tři hodnoty pole, ze kterých se udělá medián.

Ukázka [zde](#).

Selection sort

Princip:

1. Dostaneme pole.
2. Vyhledáme největší prvek pole a umístíme ho doleva.
3. Toto opakujeme, dokud nemáme seřazeno.

Složitost: Složitost je sice u selection sortu vysoká $O(n^2)$, ale má velmi nízkou paměťovou náročnost.

Ukázka algoritmu

Vyhledávací algoritmy

Lineární hledání (sekvenční hledání)

Princip: Procházíme všechny prvky, dokud nenajdu ten hledaný.

Složitost: $O(n)$

Binární hledání (metoda půlení intervalů)

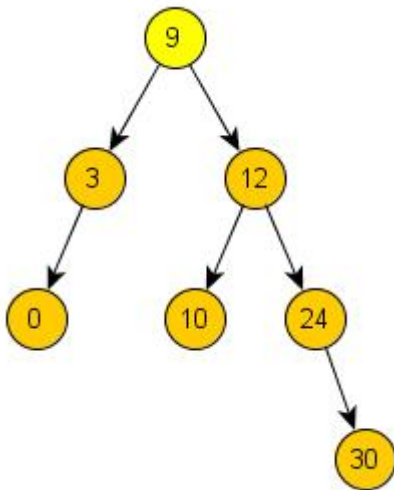
Princip:

1. Pole ve kterém se dá použít půlení intervalů, musí být seřazeno (v tomto případě od největšího po nejmenší).
2. Podívám se na prostřední prvek pole.
3. Pokud je můj hledaný prvek větší opakuj to stejné vpravo, pokud menší tak vlevo.
4. Opakuj, dokud nenajdu hledaný prvek.

Složitost: $O(\log_2(n))$

Metoda binárního vyhledávacího stromu

Princip: Tvořím binární strom (viz obrázek) tak, že vždy v levé větvi jsou menší prvky a v pravé jsou větší prvky. Hledaný prvek hledáme tak, že za ním jdeme po větvi.



Složitost: V závislosti na vyvážení stromu (podle vyváženého počtu větví na obou stranách) může být buď $O(\log(n))$ pro naprosto vyvážený strom, nebo až $O(n)$ pro vůbec nevyvážený strom.

From:

<http://wiki.gml.cz/> - GMLWiki

Permanent link:

<http://wiki.gml.cz/informatika:maturita:22a?rev=1517315866>

Last update: **30. 01. 2018, 13.37**

