15 Základní algoritmy řazení (mergesort, quicksort, heapsort, radixsort) a vyhledávání půlením intervalu, jejich složitost. (A4B36ALG)

15.1 Základní algoritmy řazení

V popiscích jednotlivých algoritmů se vyskytují následující hesla a charakteristiky

Rozděl a panuj
Tato metoda označuje ty algoritmy pro práci s daty, které řeší problém rozdělením řešené úlohy na **dílčí části (podproblémy)**, nad kterými se provádí algoritmická operace. Často se tato metoda implementuje rekurzivně nebo iterativně a původní úloha se dělí na stále menší části.

Stabilní/nestabilní algoritmus
Stabilním algoritmus je ten, který zachovává pořadí stejných prvků, tak jak byly na vstupu v seznamu. Seřadí je na odpovídající místo, ale v rámci stejných prvků zůstanou ve stejném pořadí. U nestabilního algoritmu toto nezaručíme.

15.1.1 Mergesort

Grafické zpracovaní mergesortu z předmětu A4B36ALG spolecna/15/mergesort.pdf

Mergesort je **stabilní** řadicí algoritmus typu **rozděl a panuj** s asymptotickou složitostí $O(n * \log(n))$. Merge sort pracuje na bázi **slévání** již seřazených částí pole.

15.1.1.1 Merge

Proces merge nebo-li slévání probíhá následovně. Máme dva seznamy, u kterých víme, že jsou seřazeny ve stejném pořadí. Postupně je procházíme a díváme se, který z právě iterovaných členů je větší (případně menší, záleží, jak srovnávám). Ten větší (resp. menší) z dvojice vložíme do pomocného seznamu. Takto se dostanem do fáze, kdy jeden ze seznamů, kterými iterujem, je už prázdný. Můžeme tedy zbytek druhého vzít a celý ho zkopírovat na konec do pomocného seznamu.

Ve vypsaném kódu je první `while` cyklus na řádku 13 porovnávání. Z následujících dvou cyklů (21 a 25) se provede jen jeden a to ten, který zkopíruje do pomocného seznamu zbývající členy jednoho ze dvou iterovaných seznamů.

```
1  /**
2   * Slevani pro Merge sort
3   * @param array pole k serazeni
4   * @param aux pomocne pole (stejne velikosti jako razene)
5   * @param left prvni index, na ktery smim sahnout
6   * @param right posledni index, na ktery smim sahnout
7   */
8  private static void merge(int[] array, int[] aux, int left, int
       right) {
9      int middleIndex = (left + right)/2;
10     int leftIndex = left;
11     int rightIndex = middleIndex + 1;
12     int auxIndex = left;
13     while(leftIndex <= middleIndex && rightIndex <= right) {
14         if (array[leftIndex] >= array[rightIndex]) {
15             aux[auxIndex] = array[leftIndex++];
16         } else {
17             aux[auxIndex] = array[rightIndex++];
18         }
19         auxIndex++;
20     }
21     while (leftIndex <= middleIndex) {
22         aux[auxIndex] = array[leftIndex++];
23         auxIndex++;
24     }
25     while (rightIndex <= right) {
26         aux[auxIndex] = array[rightIndex++];
27         auxIndex++;
28     }
29 }
```

15.1.1.2 Průběh algoritmu

Algoritmu dodáme array, který postupným půlením rekurzivně rozdělíme do nejmenších skupin, tedy dvojic a případně lichého členu. Na řádku 14 aplikujeme proces merge na tyto malé jednotky, a ve stacku, který se nám během rekurze vytvořil, skočíme o úroveň výš, kde sléváme podseznamy délky 4, případně 3. Takto skáčeme až do chvíle, kdy se ocitneme v hlavní smyčce programu a sléváme levou a pravou polovinu seznamu, který nám byl vložen na vstupu.

```java
/**
 * Razeni slevanim (od nejvyssiho)
 * @param array pole k serazeni
 * @param aux pomocne pole stejne delky jako array
 * @param left prvni index na ktery se smi sahnout
 * @param right posledni index, na ktery se smi sahnout
 */
public static void mergeSort(int[] array, int[] aux, int left,
    int right) {
    if(left == right) return;
    int middleIndex = (left + right)/2;
    mergeSort(array, aux, left, middleIndex);
    mergeSort(array, aux, middleIndex + 1, right);

    merge(array, aux, left, right);
}
```

15.1.2 Quicksort

Grafické zpracovaní quicksortu z předmětu A4B36ALG spolecna/15/quicksort.pdf

Quicksort je velmi rychlý **nestabilní** řadící algoritmus na principu **rozděl a panuj** s asymptotickou složitostí O(n^2) a s očekávanou složitostíO($n * \log(n)$).

15.1.2.1 Průběh algoritmu

1. Levý index se nastaví na začátek zpracovávaného úseku pole, pravý na jeho konec, zvolí se pivot, nejjednodušeji tak, že se vybere prvek na začátku zpracovávaného úseku. Levý index se pohybuje doprava a zastaví se na prvku vetším nebo rovném pivotovi. Pravý index se pohybuje doleva a zastaví se na prvku menším nebo rovném pivotovi.

2. Pokud je levý index ještě před pravým, príslušné prvky se prohodí, a oba indexy se posunou o 1 ve svém směru. Jinak pokud se indexy rovnají, jen se oba posunou o 1 ve svém směru.

3. Cyklus se opakuje, dokud se indexy neprekříží, tj. pravý se dostane pred levého.

4. Následuje rekurzivní volání (zpracování „malých" a „velkých" zvlášť) na úsek od začátku do pravého indexu včetně a na úsek od levého indexu včetně až do konce, má-li príslušný úsek délku větší než 1.

```
1  private static void qSort(int a[], int low, int high) {
2    int iL = low, iR = high;
3    int pivot = a[low];
4    do {
5      while (a[iL] < pivot) iL++;
6      while (a[iR] > pivot) iR--;
7      if (iL < iR) {
8        swap(a,iL, iR);
9        iL++; iR--; }
10     else
11       if (iL == iR) { iL++; iR--;}
12   } while( iL <= iR);
13   if (low < iR)  qSort(a, low, iR);
14   if (iL < high) qSort(a, iL, high);
15 }
```

15.1.3 Heapsort

Grafické zpracovaní heapsortu z předmětu A4B36ALG spolecna/15/heapsort.pdf

Heapsort (řazení haldou) je jedním z nejefektivnějších řadících algoritmů založených na porovnávání prvků s asymptotickou složitostí $O(n * \log(n))$. Heapsort je **nestabilní**.

15.1.3.1 Vlastnosti haldy

Halda je binární strom s rekurzivní vlastností býti haldou. Rekurzivita vlastnosti značí, že i každý podstrom haldy je taktéž haldou. V každé haldě také platí, že otec má vždy vyšší hodnotu než jeho potomci. Při její implementaci polem pak také platí, že pokud je otec na indexu i, tak jsou jeho potomci na indexech $2i + 1$ a $2i + 2$ (indexováno od 0). Z tohoto uspořádání plyne, že tvar haldy je pyramida s částečně useknutou pravou stranou základny.

15.1.3.2 Průběh algoritmu

Budu se odkazovat na zdrojový kód na následující stránce.

1. Postavme haldu nad zadaným polem. To se děje ve `for-cyklu` na řádku 7. Je důležité povšimnout si, že cyklus probíhá od poloviny a jde pozpátku. Proč? Protože seznam znázorňuje haldu, to znamená, že rodičem posledního členu bude prostřední člen. Tohle je potřeba pochopit, mrkněte na spolecna/15/heapsort.pdf.

2. Z našeho neuspořádaného seznamu jsme udělali haldu. Je to pořád seznam, array, ale splňuje charakteristiku haldy. Na řádku 10 začíná řazení.

3. Začneme tím, že prvek na vrcholu prohodíme s i-tým prvkem seznamu (řádek 11). Tím dosáváme na konci seznamu již seřazené prvky.

4. Prohozením se ale na vrcholu haldy může ocitnout prvek, který tam nemá co dělat. Proto na řádku 12 dojde k opravení haldy.

Hlavní část algoritmu je v metodě repairTop. Ta zařadí vrchol haldy na správné místo (řádek 38, proměnná topIndex se nastaví ve `while-cyklu`) a nahradí ho prvkem, který má být na vrcholu (řádek 31).

```java
/**
 * Heapsort - razeni haldou
 * @param array pole k serazeni
 * @param descending true, pokud ma byt pole serazeno sestupne, false pokud vzestupne
 */
public static void heapSort(Comparable[] array, boolean descending) {
    for (int i = array.length / 2 - 1; i >= 0; i--) {
        repairTop(array, array.length - 1, i, descending ? 1 : -1);
    }
    for (int i = array.length - 1; i > 0; i--) {
        swap(array, 0, i);
        repairTop(array, i - 1, 0, descending ? 1 : -1);
    }
}

/**
 * Umisti vrchol haldy na korektni misto v halde (opravi haldu)
 * @param array pole k setrizeni
 * @param bottom posledni index pole, na ktery se jeste smi sahnout
 * @param topIndex index vrsku haldy
 * @param order smer razeni 1 == sestupne, -1 == vzestupne
 */
private static void repairTop(Comparable[] array, int bottom, int topIndex, int order) {
    Comparable tmp = array[topIndex];
    int succ = topIndex * 2 + 1;
    if (succ < bottom && array[succ].compareTo(array[succ + 1]) == order) {
        succ++;
    }

    while (succ <= bottom && tmp.compareTo(array[succ]) == order) {
        array[topIndex] = array[succ];
        topIndex = succ;
        succ = succ * 2 + 1;
        if (succ < bottom && array[succ].compareTo(array[succ + 1]) == order) {
            succ++;
        }
    }
    array[topIndex] = tmp;
}
```

15.1.4 Radixsort

Grafické zpracovaní radixsortu z předmětu A4B36ALG spolecna/15/radixsort.pdf

Princip radix sortu vychází přímo z definice stabilního řazení – řadicí algoritmus je stabilní, pokud zachovává pořadí klíčů, které mají stejnou hodnotu (tj. pokud stuktury seřadíme napřed dle klíče **A**, poté podle klíče **B**, tak jsou seřazeny podle **B**, a kde jsou si hodnoty **B** rovny, tam jsou struktury v pořadí daném klíčem **A**).

Popis algoritmu vychází z následujícího kusu kódu, který je ale zkrácen. Celá kopie je v `spolecna/15/radixsort.java`. Názvy proměnných vychází z označení v `spolecna/15/radixsort.pdf`, je proto dobré procházet si kód zároveň s tímto pdf.

1. Vytvoříme pomocné tabulky

   - **z** - první výskyt znaků
   - **k** - poslední výskyt znaků
   - **d** - tabulka odkazů na další výskyt znaku

   a to z posledního znaku slova (poslední klíč při řazení, řádek 4). Funkce `initStep` projde pole všech slov na vstupu (15), zjistí poslední písmeno (16), pokud ještě není v tabulce prvních výskytů znaků **z**, přidá ho do ní a zároveň do tabulky posledního výskytu znaku **k** (18). Pokud už nějaký záznam s klíčem **c** existuje, algoritmus uloží na jeho pozici v tabulce **d** odkaz na právě iterovaný prvek (20), tento prvek se stane i posledním výskytem daného znaku (21).

2. Poté začneme iterovat přes délku slov na vstupu (5) - to je důvod, proč musí být všechny řetězce různé délky uvedeny na stejnou délku připojením "nevýznamných znaků", např. mezer.

3. Funkce `radixStep` se podívá na aktuální pomocné tabulky, které určují pořadí prvků podle aktuálního klíče. Při prvním průchodu funkcí to tedy bude podle posledního znaku.

   Vlastní funkce `radixStep` začíná iterací přes pole prvních výskytů znaků (28). V ní je vnořen další cyklus určen pomocnou tabulkou **d** a **k**. Pokud se aktuální prvek rovná koncovému výskytu znaku (39), tento vnitřní cyklus skončí. Během tohoto cyklu se testuje to samé jako ve funkci `initStep` s tím rozdílem, že tentokrát ukládáme informace do pomocných tabulek **z1**, **k1**, **d1** (34, 36, 37), takže si nepřepíšeme aktuálně iterované **z**, **k**, **d**. Přepsání proběhne až v hlavním cyklus programu (7, 8, 9).

```
1  void radix_sort (String [] a) {
2    int len   = ...; // number of chars used (2^16?)
3    int [] z  = new int [len]; int [] k  = new int [len]; int []
        z1 = new int [len]; int [] k1 = new int [len]; int [] d =
         new int [a.length]; int [] d1 = new int [a.length]; int
        [] aux;
4    initStep(a, z, k, d);
5    for (int p = a[0].length()-2; p >= 0; p-- ) {
6      radixStep(a, p, z, k, d, z1, k1, d1);
7      aux = z; z = z1; z1 = aux;
8      aux = k; k = k1; k1 = aux;
9      aux = d; d = d1; d1 = aux;
10   }
11 }
12
13 void initStep (String[] a, int[] z, int [] k, int[] d){
14   int pos = a[0].length()-1;
15   for (int i = 0; i < a.length; i++) {
16     c = (int) a[i].charAt(pos);
17     if (z[c] == -1)
18       k[c] = z[c] = i;
19     else {
20       d[k[c]] = i;
21       k[c]    = i;
22   }
23   }
24 }
25
26 void radixStep(String[]a, int pos, int[] z, int[] k,
27   int[] d, int[] z1, int[] k1, int[] d1){
28   for (int i = 0; i < z.length; i++)
29     if (z[i] != -1) {
30       j = z[i];
31       while (true) {
32         c = (int) a[j].charAt(pos);
33         if (z1[c] == -1)
34           k1[c]= z1[c] = j;
35         else {
36           d1[k1[c]] = j;
37           k1[c2] = j;
38         }
39       if (j == k[i]) break;
40       j = d[j];
41     }
42   }
43 }
```

15.1.5 Vyhledávání půlením intervalu

Metoda půlení intervalu (nebo také binární vyhledávání) je vyhledávací algoritmus typu **rozděl a panuj** na nalezení zadané hodnoty v **uspořádaném seznamu** pomocí zkracování seznamu o polovinu v každém kroku.

Binární vyhledávání najde medián, porovná s hledanou hodnotou a na základě výsledku porovnání se rozhodne o pokračování v horní nebo dolní části seznamu a rekurzivně pokračuje od začátku. Binární vyhledávání je algoritmus s logaritmickou časovou složitostí O(log(n)). Přesněji, je potřeba iterací na získání výsledku. Je značně rychlejší než lineární vyhledávání, které má časovou složitost O(n). Nicméně vyžaduje, aby data byla setříděna, je tudíž vhodný jen pro určitou množinu problémů.

15.1.6 Srovnání složitostí

Grafické zpracovaní složitostí vypracované z předmětu A4B36ALG spolecna/15/benchmark.pdf

	Nejhorší případ	Nejlepší případ	Průměrný případ	Stabilní
Quick sort	$\Theta(n^2)$	$\Theta(nlog(n))$	$\Theta(nlog(n))$	Ne
Merge sort	$\Theta(nlog(n))$	$\Theta(nlog(n))$	$\Theta(nlog(n))$	Ano
Heap sort	$\Theta(nlog(n))$	$\Theta(nlog(n))$	$\Theta(nlog(n))$	Ne
Radix sort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	Ano