09. Řazení

Obsah

- Typy řazení (vnitřní, vnější)
- Stabilní řazení
- Přímé metody řazení
- Řazení pomocí haldy (Heap Sort)
- Rekurzivní řadící algoritmy
- Algoritmy s lineární složitostí
- Efektivita a složitost řadících algoritmů
- Vhodné užití a příklady

Typy řazení

Vnitřní řazení

- Vychází z předpokladu, že **všechna data se vejdou do operační paměti**
- Data jsou uložena ve formě datového pole
- Rychlost algoritmu závisí na počtu porovnání a záměn
- Příklady: Selection Sort, Insertion Sort, Bubble Sort, Quick Sort, Heap Sort, Merge Sort

Vnější řazení

- Používá se, když objem dat je tak veliký, že **nestačí vnitřní paměť** a musí se využít vnější paměť (disk)
- Rychlost algoritmu je ovlivněna počtem přístupů k vnější paměti
- Typicky využívá slučování seřazených částí (merge)
- Příklad: Víceprůchodové přihrádkové řazení, Přímé slučování

Stabilní řazení

Stabilní řazení je takové, které zachovává vzájemné pořadí prvků se stejným klíčem.

Příklad stability

- Máme množinu prvků M
- Pro každé dva prvky R a S o stejném klíči platí:
  - Pokud byl prvek R v neseřazené množině před prvkem S, pak je i v seřazené posloupnosti prvek R před prvkem S

Stabilní algoritmy

- Bubble Sort
- Insertion Sort
- Merge Sort
- Counting Sort
- Radix Sort
- Bucket Sort

Nestabilní algoritmy

- Selection Sort (standardně, lze upravit na stabilní)
- QuickSort
- HeapSort

Přímé metody řazení

Selection Sort (Řazení přímým výběrem)

Princip: 1. Najdeme prvek s nejmenší hodnotou v posloupnosti 2. Zaměníme ho s prvkem na první pozici 3. Na první pozici se nyní nachází správný prvek, zbytek posloupnosti se uspořádá opakováním pro zbylých n-1 prvků

```
void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int min_idx = i;
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        // Záměna prvků
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Vlastnosti: - Časová složitost: $O(n^2)$ ve všech případech - Velmi jednoduchá implementace - Nestabilní (standardně) - Vhodný pro malé množství dat

Insertion Sort (Přímé zatřiďování)

Princip: 1. Procházíme prvky postupně 2. Každý další nesetříděný prvek zařadíme na správné místo do již setříděné posloupnosti

```
void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Vlastnosti: - Časová složitost: $O(n^2)$ v průměrném a nejhorším případě, $O(n)$ v nejlepším případě - Efektivní na malých množinách a částečně seřazených datech - Stabilní - Přirozený (efektivnější na částečně seřazených datech)

Bubble Sort (Bublinkové řazení)

Princip: 1. Opakovaně procházíme seznam 2. Porovnáváme každé dva sousedící prvky 3. Pokud nejsou ve správném pořadí, prohodíme je 4. Největší prvky "probublávají" na konec seznamu

```
void bubbleSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        for (int j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Záměna prvků
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
```

```
            }
        }
    }
}
```

Vlastnosti: - Časová složitost: $O(n^2)$ v průměrném a nejhorším případě, O(n) v nejlepším případě - Jednoduchá implementace - Stabilní - Přirozený - Pro praktické účely neefektivní

Řazení pomocí haldy (Heap Sort)

Halda (Heap) je speciální binární strom, který splňuje tyto podmínky: - Je to **kompletní binární strom** (všechny úrovně kromě poslední jsou plně obsazeny, poslední úroveň je zaplněna zleva) - Pro každý uzel platí, že hodnota v něm uložená je větší (Max-Heap) nebo menší (Min-Heap) než hodnoty v jeho potomcích

Princip Heap Sort: 1. Vytvoříme z pole haldu (operace "heapify") 2. Opakovaně odebíráme kořen haldy (největší prvek) a umisťujeme ho na konec pole 3. Po každém odebrání znovu provedeme "heapify" pro zbytek haldy

```
void heapSort(int[] arr) {
    int n = arr.length;

    // Vytvoření haldy
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // Postupné odebírání pruků z haldy
    for (int i = n - 1; i > 0; i--) {
        // Přesun kořene na konec
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Oprava haldy po odebrání kořene
        heapify(arr, i, 0);
    }
}

void heapify(int[] arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // Porovnání s levým potomkem
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // Porovnání s pravým potomkem
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // Pokud není kořen největší, prohodíme ho
    if (largest != i) {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;
```

```
        // Rekurzivně opravíme narušenou část haldy
        heapify(arr, n, largest);
    }
}
```

Vlastnosti: - Časová složitost: O(n log n) ve všech případech - Prostorová složitost: O(1), řadí data na místě - Nestabilní - Efektivní pro velké množství dat

Rekurzivní řadící algoritmy

Quick Sort

Princip (metoda "rozděl a panuj"): 1. Vybereme prvek jako pivot 2. Přesuneme prvky menší než pivot před něj a větší za něj (partitioning) 3. Rekurzivně seřadíme obě části

```
void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        // Rozdělovací index (pivot)
        int pi = partition(arr, low, high);

        // Rekurzivní řazení před a po pivotu
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;

            // Záměna prvků
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Záměna s pivotem
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}
```

Vlastnosti: - Časová složitost: O(n log n) v průměrném případě, O(n^2) v nejhorším případě - Velmi rychlý v praxi díky efektivní práci s cache - Nestabilní - Řazení na místě - Nevhodný pro již seřazená nebo téměř seřazená data

Merge Sort

Princip (metoda "rozděl a panuj"): 1. Rozdělíme pole na dvě poloviny 2. Rekurzivně seřadíme obě poloviny 3. Sloučíme obě seřazené poloviny do jednoho seřazeného pole

```
void mergeSort(int[] arr, int l, int r) {
    if (l < r) {
        // Nalezení středového bodu
        int m = l + (r - l) / 2;

        // Seřazení první a druhé poloviny
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Sloučení seřazených polovin
        merge(arr, l, m, r);
    }
}

void merge(int[] arr, int l, int m, int r) {
    // Velikosti dočasných polí
    int n1 = m - l + 1;
    int n2 = r - m;

    // Vytvoření dočasných polí
    int[] L = new int[n1];
    int[] R = new int[n2];

    // Kopírování dat do dočasných polí
    for (int i = 0; i < n1; ++i) {
        L[i] = arr[l + i];
    }
    for (int j = 0; j < n2; ++j) {
        R[j] = arr[m + 1 + j];
    }

    // Sloučení dočasných polí
    int i = 0, j = 0;
    int k = l;

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Kopírování zbývajících pruků L[]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Kopírování zbývajících pruků R[]
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
```

```
    }
}
```

Vlastnosti: - Časová složitost: O(n log n) ve všech případech - Stabilní - Vyžaduje dodatečnou paměť O(n) - Vhodný pro řazení spojových seznamů - Efektivní pro vnější řazení velkých datových souborů

Algoritmy s lineární složitostí

Radix Sort (Přihrádkové řazení)

Princip: 1. Řadíme čísla podle jednotlivých řádů (jednotky, desítky, stovky, ...) od nejnižšího k nejvyššímu 2. Pro každý řád použijeme stabilní řazení (typicky Counting Sort)

Omezení: - Použitelný pouze pro celá čísla - Hodnoty musí být z předem známého, ne příliš velkého rozmezí

```
void radixSort(int[] arr) {
    // Nalezení maximální hodnoty pro určení počtu průchodů
    int max = Arrays.stream(arr).max().getAsInt();

    // Řazení podle každého řádu
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSortByDigit(arr, exp);
    }
}

void countingSortByDigit(int[] arr, int exp) {
    int n = arr.length;
    int[] output = new int[n];
    int[] count = new int[10]; // 0-9 číslic

    // Inicializace počítacího pole
    Arrays.fill(count, 0);

    // Počítání výskytů číslic
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    // Kumulativní počty
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Vytvoření výstupního pole
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Kopírování seřazeného pole
    System.arraycopy(output, 0, arr, 0, n);
}
```

Vlastnosti: - Časová složitost: O(d * (n + k)), kde d je počet řádů, n je počet prvků a k je rozsah hodnot - Stabilní - Vyžaduje dodatečnou paměť - Efektivní pro celá čísla s omezeným počtem řádů

Efektivita a složitost řadících algoritmů

Algoritmus	Průměrná složitost	Nejhorší složitost	Paměťová náročnost	Stabilita
Bubble Sort	$O(n^2)$	$O(n^2)$	O(1)	Stabilní
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)	Nestabilní
Insertion Sort	$O(n^2)$	$O(n^2)$	O(1)	Stabilní
Heap Sort	O(n log n)	O(n log n)	O(1)	Nestabilní
Quick Sort	O(n log n)	$O(n^2)$	O(log n)	Nestabilní
Merge Sort	O(n log n)	O(n log n)	O(n)	Stabilní
Radix Sort	O(d * (n + k))	O(d * (n + k))	O(n + k)	Stabilní

Vhodné užití a příklady

Selection Sort

- Vhodný pro: malé množství dat, situace s omezenou pamětí
- Příklad: Řazení seznamu pěti studentů podle věku

Insertion Sort

- Vhodný pro: malé množství dat, téměř seřazená data
- Příklad: Řazení karet v ruce při karetních hrách, průběžné řazení nově příchozích dat

Bubble Sort

- Vhodný pro: edukativní účely, jednoduché implementace
- Příklad: Výukové aplikace demonstrující princip řazení

Heap Sort

- Vhodný pro: velké množství dat, situace vyžadující garantovanou složitost
- Příklad: Systémy zpracovávající prioritní fronty, operační systémy

Quick Sort

- Vhodný pro: většinu praktických situací, velké objemy dat
- Příklad: Implementace standardních knihovních funkcí pro řazení, databázové systémy

Merge Sort

- Vhodný pro: spojové seznamy, vnější řazení, paralelní zpracování
- Příklad: Řazení velkých souborů dat na disku, paralelní algoritmy

Radix Sort

- Vhodný pro: celá čísla s omezeným počtem číslic, řetězce stejné délky
- Příklad: Řazení identifikačních čísel, poštovních směrovacích čísel