**LAPORAN**

**STRUKTUR DATA**

**(TIFNJK130704)**

**MINGGU 12**

**SEMESTER III**

**Logo, icon

Description automatically generated**

**(Tugas Sorting)**

**Oleh:**

Syafrizal Wd Mahendra (E41222719)/C2

**PROGRAM STUDI TEKNIK INFORMATIKA KAMPUS 3 NGANJUK**

**JURUSAN TEKNOLOGI INFORMASI**

**POLITEKNIK NEGERI JEMBER**

**TAHUN 2023**

1. **Implementasi Fungsi-fungsi Metode Sorting**

* **Fungsi Main**

public static void main(String[] args) {

int[] arr = {33, 12, 45, 6, 78, 23, 90, 1, 8, 56, 72};

System.out.println("Original Array: ");

printArray(arr);

// Bubble Sort

bubbleSort(arr.clone());

// Selection Sort

selectionSort(arr.clone());

// Insertion Sort

insertionSort(arr.clone());

// Quick Sort

quickSort(arr.clone(), 0, arr.length - 1);

}

* **Method untuk Menampilkan Array Awal**

// Print Array

public static void printArray(int[] arr) {

for (int num : arr) {

System.out.print(num + " ");

}

System.out.println();

}

* **Metode Bubble Sort**

// Bubble Sort Method

public static void bubbleSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// swap temp and arr[i]

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

System.out.println("After Bubble Sort: ");

printArray(arr);

}

* **Metode Selection Sort**

// Selection Sort Method

public static void selectionSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

// swap the found minimum element with the first element

int temp = arr[minIndex];

arr[minIndex] = arr[i];

arr[i] = temp;

}

System.out.println("After Selection Sort: ");

printArray(arr);

}

* **Metode Insertion Sort**

// Insertion Sort Method

public static void insertionSort(int[] arr) {

int n = arr.length;

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1] that are greater than key to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

System.out.println("After Insertion Sort: ");

printArray(arr);

}

* **Metode Quick Sort**

// Quick Sort

public static void quickSort(int[] arr, int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

// Recursively sort elements before and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

System.out.println("After Quick Sort: ");

printArray(arr);

}

// Helper method for Quick Sort to find the partition position

public static int partition(int[] arr, int low, int high) {

int pivot = arr[high];

int i = (low - 1); // index of smaller element

for (int j = low; j < high; j++) {

// If the current element is smaller than the pivot

if (arr[j] < pivot) {

i++;

// swap arr[i] and arr[j]

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

// swap arr[i+1] and arr[high] (or pivot)

int temp = arr[i + 1];

arr[i + 1] = arr[high];

arr[high] = temp;

return i + 1;

}

* **Output**

run:

Original Array:

33 12 45 6 78 23 90 1 8 56 72

After Bubble Sort:

1 6 8 12 23 33 45 56 72 78 90

After Selection Sort:

1 6 8 12 23 33 45 56 72 78 90

After Insertion Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 33 23 12 45 56 72 78 90

After Quick Sort:

1 6 8 33 23 12 45 56 72 78 90

After Quick Sort:

1 6 8 33 23 12 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

After Quick Sort:

1 6 8 12 23 33 45 56 72 78 90

BUILD SUCCESSFUL (total time: 0 seconds)

1. **Hasil Pengukuran Kinerja dari Masing-masing Metode Sorting**

* Kompleksitas Waktu

Bubble sort, selection sort, dan insertion sort semuanya memiliki kompleksitas waktu O(n^2) dalam kasus rata-rata dan terburuk. Artinya, jika ukuran inputnya menjadi dua kali lipat (2n), waktu eksekusinya akan menjadi empat kali lipat (4 \* waktu sebelumnya).

Quick Sort memiliki kompleksitas waktu O(n log n) dalam kasus rata-rata karena pada setiap tingkat rekursifnya, algoritma membagi array menjadi dua bagian sekitar setengah dari ukuran aslinya dan melakukan pemrosesan di setiap bagian. Kasus terburuk terjadi ketika partisi yang dipilih selalu merupakan elemen terkecil atau terbesar di array. Dalam situasi ini, setiap tingkat rekursif hanya mengurangi satu elemen dari array, dan algoritma melakukan partisi pada array yang lebih kecil dan lebih kecil pada setiap tingkatnya. Sebagai hasilnya, dalam kasus terburuk, jumlah operasi yang diperlukan dapat tumbuh secara kuadrat, menghasilkan kompleksitas waktu O(n^2).

* Kecepatan

1. Bubble Sort :

**Kecepatan Rata-rata:** Lambat. Memiliki kompleksitas waktu O(n^2), yang membuatnya tidak efisien untuk data set yang besar. Menggunakan pertukaran berulang hingga data terurut.

**Kecepatan Terbaik:** O(n) pada kondisi terbaik ketika array sudah terurut.

1. Selection Sort :

**Kecepatan Rata-rata:** Lambat. Kompleksitas waktu O(n^2), yang membuatnya tidak efisien untuk data set yang besar. Melibatkan pertukaran dan seleksi elemen minimum pada setiap iterasi.

**Kecepatan Terbaik:** O(n^2) pada semua kondisi.

1. Insertion Sort :

**Kecepatan Rata-rata:** Cukup baik untuk data yang kecil atau hampir terurut. Memiliki kompleksitas waktu O(n^2), tetapi lebih efisien daripada Bubble Sort dan Selection Sort dalam beberapa kasus.

**Kecepatan Terbaik:** O(n) pada kondisi terbaik ketika array sudah hampir terurut.

1. Insertion Sort :

**Kecepatan Rata-rata:** Cepat. Memiliki kompleksitas waktu O(n log n) dalam kasus rata-rata, yang membuatnya efisien untuk data set yang besar dan umumnya dianggap sebagai salah satu metode pengurutan yang paling cepat.

**Kecepatan Terbaik:** O(n log n) pada kondisi terbaik.