Sorting Algorithm Performance Comparison

2024–2025 Spring

CME2204 Algorithm Analysis

Assignment 1

Erol Çelik

2022510164

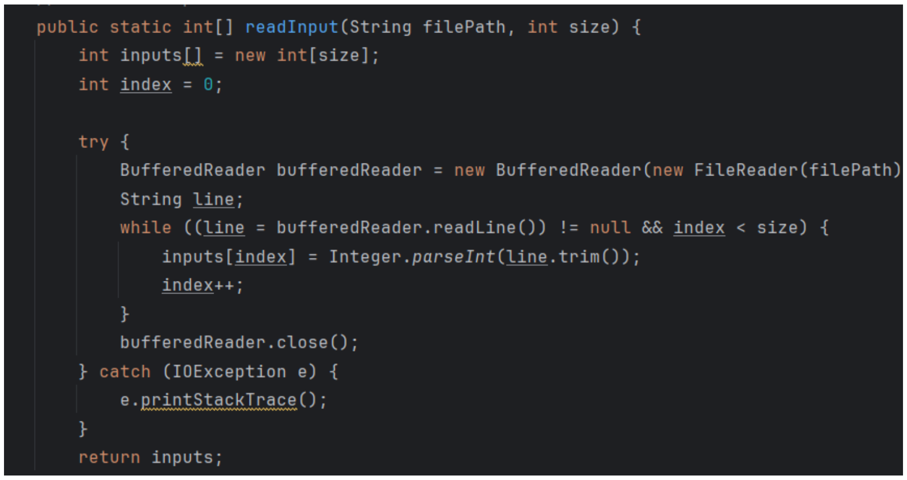
May 07, 2025

Implementation Details

The project includes four basic sorting algorithms: Insertion Sort, Merge Sort, Heap Sort, and Quick Sort. All of them are written inside a single Java class called Sorting. Each sorting algorithm is written as a separate static method and may include sub-methods.

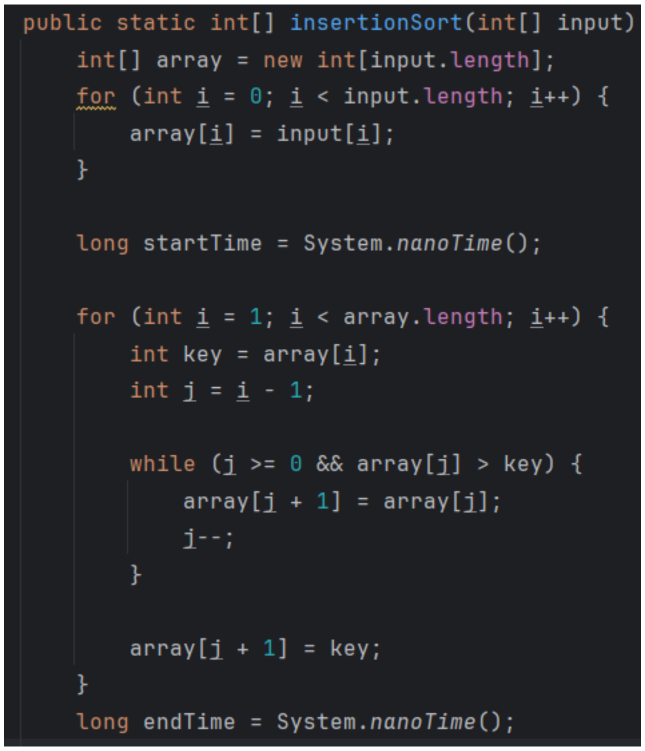
The input arrays are tested with different types of data: increasing order, decreasing order, and random numbers. Each type has three sizes: 1K, 10K, and 100K elements.

The input data is read from .txt files in the inputs folder. After sorting, the output data is saved into the inputs folder.The execution time of each sorting algorithm is measured in milliseconds using the System.nanoTime() function.



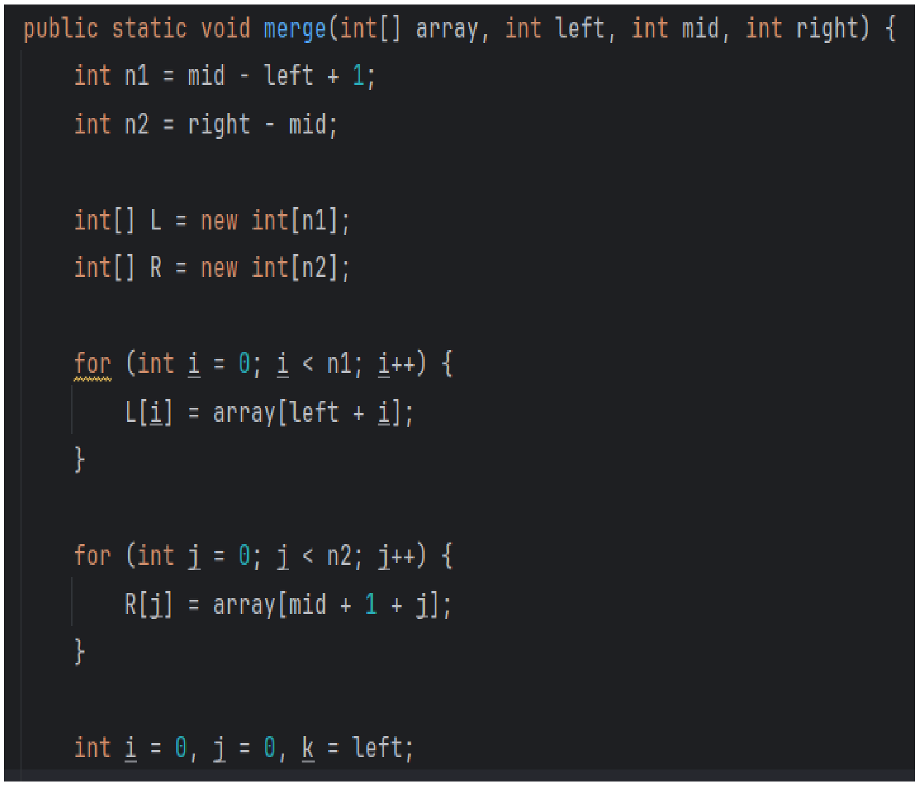
```java
public static int[] readInput(String filePath, int size) {
    int inputs[] = new int[size];
    int index = 0;

    try {
        BufferedReader bufferedReader = new BufferedReader(new FileReader(filePath)
        String line;
        while ((line = bufferedReader.readLine()) != null && index < size) {
            inputs[index] = Integer.parseInt(line.trim());
            index++;
        }
        bufferedReader.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return inputs;
```

reads the the input file from given file path.used for reading random input files.

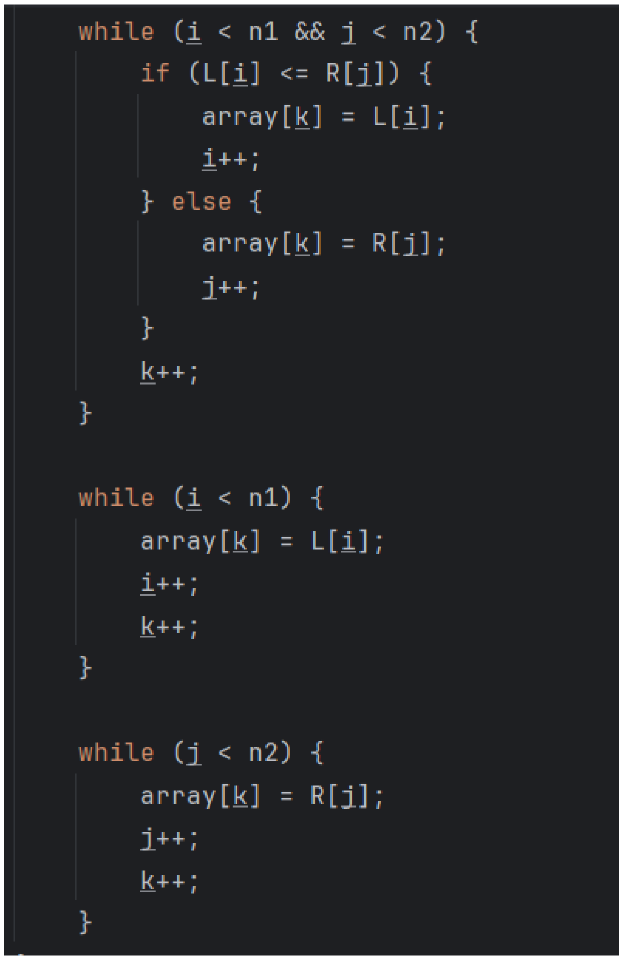


```java
public static int[] insertionSort(int[] input)
    int[] array = new int[input.length];
    for (int i = 0; i < input.length; i++) {
        array[i] = input[i];
    }

    long startTime = System.nanoTime();

    for (int i = 1; i < array.length; i++) {
        int key = array[i];
        int j = i - 1;

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

        array[j + 1] = key;
    }
    long endTime = System.nanoTime();
```

Sorts array by comparison each element one by one.

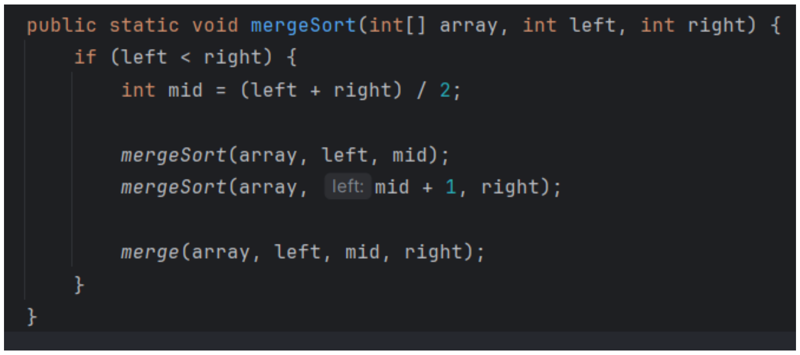


```java
public static void merge(int[] array, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int[] L = new int[n1];
    int[] R = new int[n2];

    for (int i = 0; i < n1; i++) {
        L[i] = array[left + i];
    }

    for (int j = 0; j < n2; j++) {
        R[j] = array[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;
```



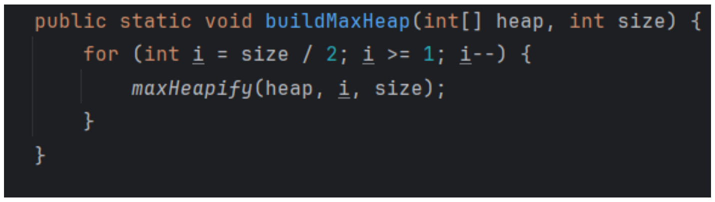
```java
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            array[k] = L[i];
            i++;
        } else {
            array[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        array[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        array[k] = R[j];
        j++;
        k++;
    }
```

First, it calculates the sizes of the left and right parts, and then it copies the values from the main array into two new temporary arrays. After that, it compares the elements of these two arrays one by one and puts the smaller value back into the original array. If one of the arrays finishes earlier, it copies the remaining elements from the other array. In the end, the selected part of the main array becomes sorted.



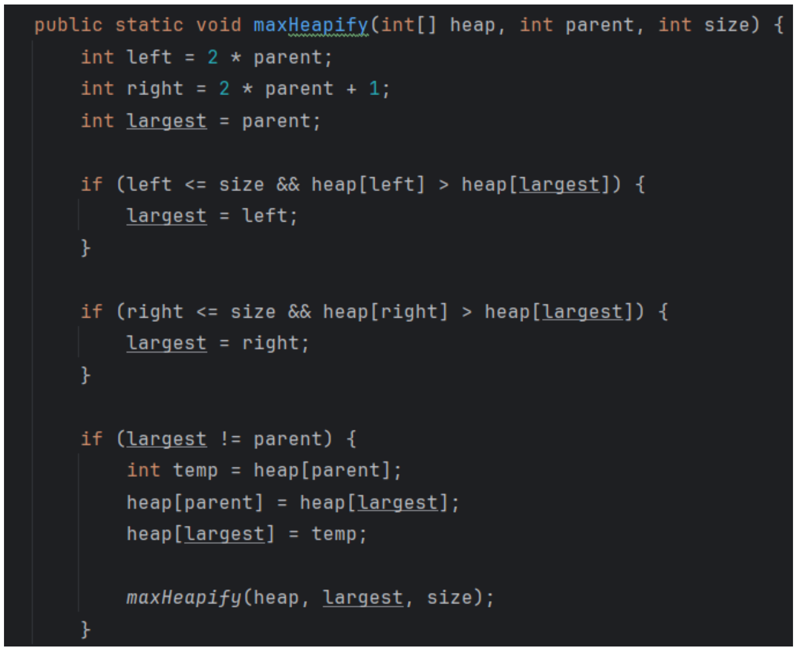
```java
public static void mergeSort(int[] array, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        mergeSort(array, left, mid);
        mergeSort(array, left: mid + 1, right);

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

Main part of Merge Sort. It works by dividing the array into two halves again and again until each part has only one element. It first finds the middle point, then it calls itself for the left and right parts. After both sides are sorted, it calls the merge function to join them into one sorted part. This process continues until the whole array is sorted.



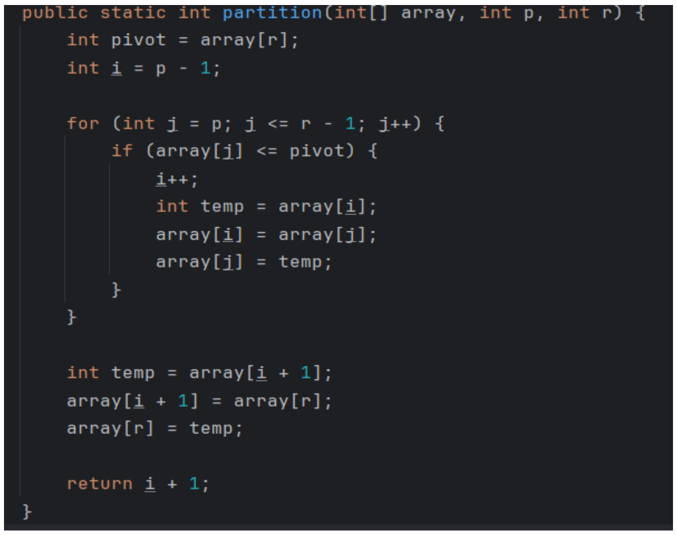
```java
public static void buildMaxHeap(int[] heap, int size) {
    for (int i = size / 2; i >= 1; i--) {
        maxHeapify(heap, i, size);
    }
}
```

Turns an unsorted array into a max heap. It starts from the middle of the array and moves to the beginning, calling maxHeapify for each parent. This makes sure the heap rules are followed before sorting starts.



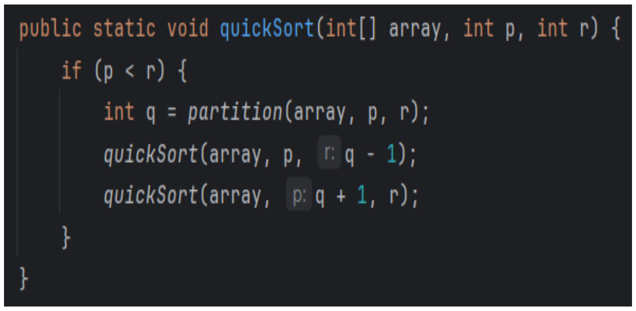
```java
public static void maxHeapify(int[] heap, int parent, int size) {
    int left = 2 * parent;
    int right = 2 * parent + 1;
    int largest = parent;

    if (left <= size && heap[left] > heap[largest]) {
        largest = left;
    }

    if (right <= size && heap[right] > heap[largest]) {
        largest = right;
    }

    if (largest != parent) {
        int temp = heap[parent];
        heap[parent] = heap[largest];
        heap[largest] = temp;

        maxHeapify(heap, largest, size);
    }
```

Checks if a parent node is smaller than its children. If one of the children is bigger, it swaps the parent with the larger child. Then it calls itself again on the new position of the parent. This keeps the max-heap property.



```java
public static int partition(int[] array, int p, int r) {
    int pivot = array[r];
    int i = p - 1;

    for (int j = p; j <= r - 1; j++) {
        if (array[j] <= pivot) {
            i++;
            int temp = array[i];
            array[i] = array[j];
            array[j] = temp;
        }
    }

    int temp = array[i + 1];
    array[i + 1] = array[r];
    array[r] = temp;

    return i + 1;
}
```

Chooses the last element in the array as the pivot. Then it checks each number in the array. If a number is smaller than or equal to the pivot, it is moved to the left side. At the end, the pivot is placed between the smaller and larger numbers.



```java
public static void quickSort(int[] array, int p, int r) {
    if (p < r) {
        int q = partition(array, p, r);
        quickSort(array, p, r: q - 1);
        quickSort(array, p: q + 1, r);
    }
}
```

Main part of the Quick Sort algorithm. It works by choosing a pivot using the partition function and then dividing the array into two parts: one smaller than the pivot, and one larger. After that, it sorts both sides by calling itself again. This keeps going until all parts are sorted.

Hardware Configuration

- CPU: 11th Gen Intel(R) Core(TM) i7-1165G7
- RAM: 16 GB
- Operating System: Windows 11 64-bit
- Java Version: OpenJDK 17+

Performance Evaluation Table

	RANDOM INTEGERS			INCREASING INTEGERS			DECREASING INTEGERS		
N / ms	1,000	10,000	100,000	1,000	10,000	100,000	1,000	10,000	100,000
Insertion Sort	2,841	42,876	2749,946	0,002	0,020	0,129	0,337	17,473	2078,221
Merge Sort	0,911	2,425	18,437	0,090	0,836	6,871	0,079	0,684	7,145
Heap Sort	0,341	1,532	14,883	0,114	0,666	13,594	0,110	0,668	11,152
Quick Sort	0,491	1,494	20,304	1,203	76,166	7462,133	0,843	60,717	6705,873

Analysis and Discussion

According to the results, Insertion Sort worked very well on already sorted (increasing) arrays because its best case is O(n). But it showed weak performance on large random and decreasing arrays. Merge Sort and Heap Sort gave stable O(n log n) performance for all data types because they use the divide and conquer approach. However, Merge Sort is not in-place, so it uses more memory than Heap Sort. Quick Sort was fast on most random inputs, but it performed badly on decreasing arrays because of poor pivot selection. Also, in large arrays, too much recursion caused a stack overflow error.

This project shows that choosing the right sorting algorithm depends on the structure of the input data. Merge Sort and Heap Sort are generally suitable for large datasets, while Insertion Sort is a good choice for small or nearly sorted arrays.

Algorithm Comparison Summary

	Average Case Complexity	Best Case Complexity	Worst Case Complexity	Is Stable?	Is in Place?	Comparison Based or Divide and Conquer
Insertion Sort	O(n^2)	O(n)	O(n^2)	Yes	Yes	Comparison Based
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Yes	No	Divide and Conquer
Heap Sort	O(n log n)	O(n log n)	O(n log n)	No	Yes	Comparison Based
Quick Sort	O(n log n)	O(n log n)	O(n^2)	No	Yes	Divide and Conquer