

Selection Sort

```
import java.util.Random;
public class main {
    public static void main(String[] args) {
        int n = 1000;
        int[] randomArray = new int[n];
        Random rand = new Random();
        for (int i = 0; i < n; i++) {
            randomArray[i] = rand.nextInt(1000);
        }
        long start = System.currentTimeMillis();
        sort(randomArray);
        long finish = System.currentTimeMillis();
        long timeElapsed = finish - start;
        System.out.println(timeElapsed);
    }
    static void sort(int arr[]) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
            int min_idx = i;
            for (int j = i + 1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;

            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
}
```

Used Java's Random Library in order to fill our array.

This code implements the array in Selection Sort algorithm. Also shows the elapsed time while sorting the array in algorithm.

Selection Sort method

Table of showing the array sizes and the sorting time in the Selection Sort algorithm:

Array Size (integers)🔗	1000	10000	100000	1000000	3133456
Selection Sort Elapsed Time (millisecond)🔗	3	47	4471	336587	723675

Heap Sort

```
import java.util.Random;
public class HeapSort {
    public static void main(String[] args) {
        int n = 1000;
        int[] randomArray = new int[n];
        Random rand = new Random();
        for (int i = 0; i < n; i++) {
            randomArray[i] = rand.nextInt(1000);
        }
        long start = System.currentTimeMillis();
        sort(randomArray);
        long finish = System.currentTimeMillis();
        long timeElapsed = finish - start;
        System.out.println(timeElapsed);
    }
    public static void sort(int arr[])
    {
        int N = arr.length;

        for (int i = N / 2 - 1; i >= 0; i--)
            heapify(arr, N, i);

        for (int i = N - 1; i > 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

            heapify(arr, i, 0);
        }
    }
    static void heapify(int arr[], int N, int i)
    {
        int largest = i;
        int l = 2 * i + 1;
        int r = 2 * i + 2;

        if (l < N && arr[l] > arr[largest])
            largest = l;

        if (r < N && arr[r] > arr[largest])
            largest = r;

        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
            heapify(arr, N, largest);
        }
    }
}
```

Used Java's Random library in order to fill our array.

This code implements the array in Heap Sort algorithm. Also show the elapsed time while sorting the array in algorithm.

This function implements the Heap Sort algorithm. It first uses the heapify function to turn the array into a maximum heap. Then, it repeatedly takes the maximum element as the root and swaps it with the element at the end of the array, followed by another heapify operation. This step sorts the array.

This function is used to transform a specific node of an array into a maximum heap of a given size. It checks the left and right child nodes of the node in question. It identifies the largest node, and if it's not the root node, it swaps them and repeats the process, ensuring the maximum heap property.

Table of showing the array sizes and the sorting time in the Heap Sort algorithm:

Array Size (integers)🔗	1000	10000	100000	1000000	3133456
Heap Sort Elapsed Time (millisecond)🔗	1	2	15	180	632

Merge Sort

```

import java.util.Random;
public class TheLargestArray {
    public static void main(String[] args) {
        int n = 1000;
        int[] randomArray = new int[n];
        Random rand = new Random();
        for (int i = 0; i < n; i++) {
            randomArray[i] = rand.nextInt(1000);
        }
        long start = System.currentTimeMillis();
        mergeSort(randomArray);
        long finish = System.currentTimeMillis();
        long timeElapsed = finish - start;
        System.out.println("Time elapsed: "+timeElapsed+" millisecond.");
    }

    private static void mergeSort(int[] inputArray) {
        int inputLength = inputArray.length;

        if (inputLength < 2) {
            return;
        }

        int midIndex = inputLength / 2;
        int[] leftHalf = new int[midIndex];
        int[] rightHalf = new int[inputLength - midIndex];

        for (int i = 0; i < midIndex; i++) {
            leftHalf[i] = inputArray[i];
        }
        for (int i = midIndex; i < inputLength; i++) {
            rightHalf[i - midIndex] = inputArray[i];
        }

        mergeSort(leftHalf);
        mergeSort(rightHalf);
        merge(inputArray, leftHalf, rightHalf);
    }

    private static void merge (int[] inputArray, int[] leftHalf, int[] rightHalf) {
        int leftSize = leftHalf.length;
        int rightSize = rightHalf.length;

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

        while (i < leftSize && j < rightSize) {
            if (leftHalf[i] <= rightHalf[j]) {
                inputArray[k] = leftHalf[i];
                i++;
            }
            else {
                inputArray[k] = rightHalf[j];
                j++;
            }
            k++;
        }

        while (i < leftSize) {
            inputArray[k] = leftHalf[i];
            i++;
            k++;
        }

        while (j < rightSize) {
            inputArray[k] = rightHalf[j];
            j++;
            k++;
        }
    }
}

```

Used Java's Random library in order to fill our array.

This code implements the array in Merge Sort algorithm. Also shows the elapsed time while sorting the array in algorithm.

This function implements the Merge Sort algorithm for sorting an array. It first checks if the length of the input array is less than 2, and if so, it returns because an array with 0 or 1 elements is already considered sorted. If the array has more than one element, it calculates the middle index and divides the array into two halves, leftHalf and rightHalf. It then recursively calls mergeSort on both halves. Finally, it merges the two sorted halves back into the original array using the merge function.

This function is responsible for merging two sorted arrays (leftHalf and rightHalf) into a single sorted array (inputArray). It maintains three pointers, i for leftHalf, j for rightHalf, and k for inputArray. It compares elements at the i and j positions of leftHalf and rightHalf, respectively, and places the smaller element into the inputArray at position k. It increments i, j, and k accordingly. This process continues until either the leftHalf or rightHalf is exhausted. After one of the halves is exhausted, the function copies the remaining elements from the other half into the inputArray. The end result is a merged and sorted inputArray.

Table of showing the array sizes and the sorting time in the Merge Sort algorithm:

Array Size(integers) -->	1000	10000	100000	1000000	3133456
Merge Sort Elapsed Time (millisecond)🕒	1	2	45	181	520

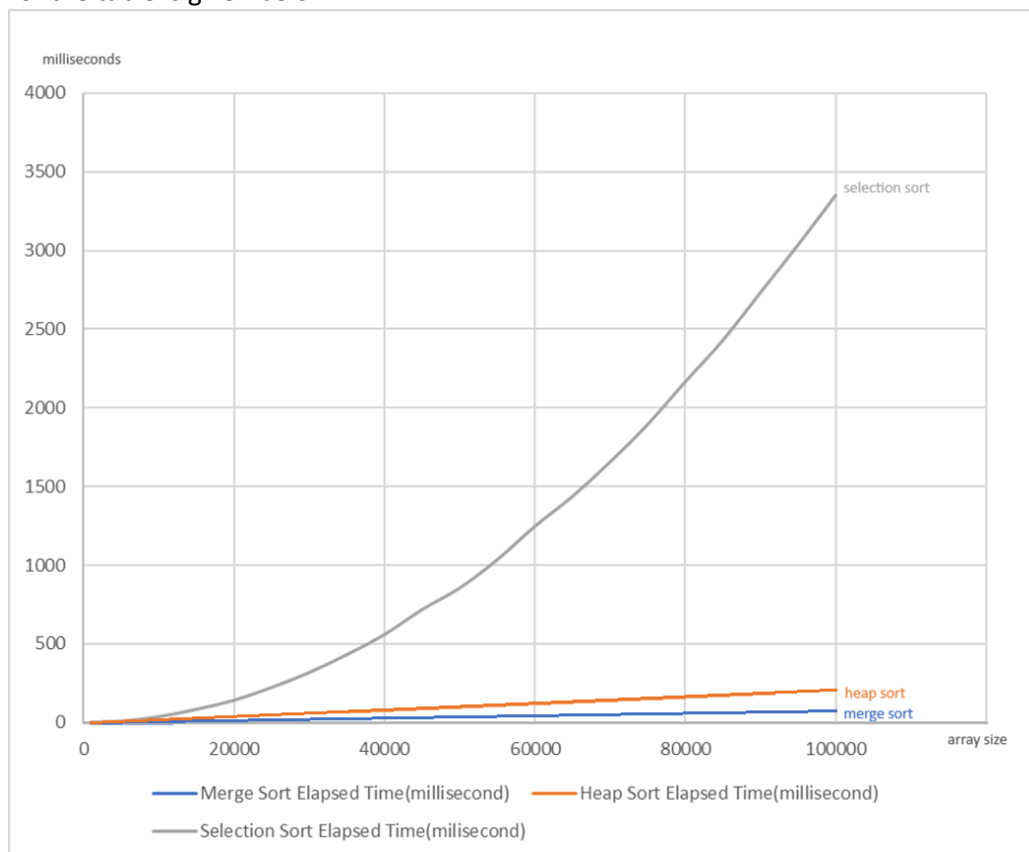
Comparison of the three sorting algorithms

Array size (integers)	1000	10000	100000	1000000	3133456
Selection Sort Elapsed Time (millisecond)	3	47	4471	336587	723675
Heap Sort(millisecond)	1	2	15	180	632
Merge Sort (millisecond)	1	2	45	181	520

In order to make a comprehensive comparison and make the graph it is possible to narrow the array sizes and make a better inference.

Array Size (integers)	1000	5000	10000	15000	20000	25000	30000	35000	40000	45000	50000	55000	60000	65000	70000	75000	80000	85000	90000	95000	100000
Merge Sort Elapsed Time(millisecond)	1	1	2	3	3	4	4	5	6	6	6	7	8	8	9	9	10	10	11	11	11
Heap Sort Elapsed Time(millisecond)	1	1	2	3	5	5	6	9	9	9	9	9	10	12	13	13	14	16	17	17	18
Selection Sort Elapsed Time(millisecond)	2	10	38	85	142	224	319	432	560	718	855	1035	1247	1437	1656	1894	2165	2429	2732	3035	3349

Graph of the table is given below:



Comparison and Conclusion:

As it easily can be seen on the graph, Selection Sort takes too much time compared with Merge and Heap Sort algorithms while sorting large sized arrays. Also selection sort is inferior in terms of performance compared to the other two algorithms. On the other hand, Heap and Merge Sort algorithms has a small differences between each other. Both algorithms have $O(n \log n)$ complexities, but the difference occurs since Merge Sort uses extra memory because it creates new subarrays and merges them during the sorting process. However, Heap sort would be more efficient in terms of extra memory usage, because it only converts the data into heap structure.

Best Performing 🏆 Merge Sort

Worst Performing 🏆 Selection Sort

References:

GeeksforGeeks. (n.d). Selection Sort. <https://www.geeksforgeeks.org/selection-sort/>

GeeksforGeeks. (n.d). Heap Sort. <https://www.geeksforgeeks.org/heap-sort/>

GeeksforGeeks. (n.d). Merge Sort. <https://www.geeksforgeeks.org/merge-sort/>

Ted University CMPE223 Sorting Algorithms Lecture Notes (n.d)