# Sorting Algorithms

Dhruba Das

Spring Fall 2023

## Introduction

In this assignment, we are tasked with the creation and benchmarking of 4 sorting algorithms: Selection sort, Insertion sort, Merge sort and Quick sort.

## Selection sort

This was the easiest sorting algorithm to implement, as we only had to start from the element in the first index, find the index for the element which is the smallest compared to it, swap the two elements, and move on. Below is the code snippet for the algorithm:

```
public static void swap(int[] array, int i, int j){
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;

    }

    public static void sort(int[] array){
        for(int i = 0; i < array.length - 1; i++){
            int cand = i;
            for(int j = i + 1; j < array.length; j++){
                if(array[i] > array[j]){
                 cand = j;
                }
            }
            swap(array, i, cand);
        }
    }
```

The "swap" method is the same for both Selection and Insertion sort. The time complexity of this algorithm O($n^2$)

## Insertion sort

This was a little more complicated to implement, but still relatively easier than both Merge and Quick sort. Here, if the item at the index position is smaller than the item before, we move it towards the beginning (i.e. insert); when inserted at the right place, we move the index forward. Below is the code snippet for this algorithm:

```
public static void sort(int[] array){
        for(int i = 0; i < array.length; i++){
            for(int j = i; j > 0 && (array[j] < array[j - 1]); j--)
             swap(array, j, j-1);
        }
    }
```

The time complexity of this algorithm O($n^2$), but when it becomes O($n$).

## Merge Sort

Reading the guidelines thoroughly before implementing the different methods helped quite a bit with this task.

### Sort

Below I have added the code snippet for the second sort method which I implemented:

```
private static void sort(int[] org, int[] aux, int lo, int hi) {
        if (lo != hi) {
        int mid = (lo + hi)/2;
        sort(org, aux, lo, mid);

        sort(org, aux, mid+1, hi);
        merge(org, aux, lo, mid, hi);
        }
    }
```

The above is a method that, given an original array and a temporary array, will sort the original array from lo to hi.

### "merge" method

Below I have added the code snippet for the "merge" method, including comments which explain each step.

```
private static void merge(int[] org, int[] aux, int lo, int mid, int hi) {
        // copy all items from lo to hi from org to aux
        for (int i = lo; i <= hi; i++) {
            aux[i] = org[i];
        }
        // let's do the merging
        int i = lo; // the index in the first part
        int j = mid+1; // the index in the second part
        // for all indices from lo to hi
        for (int k = lo; k <= hi; k++) {
        // if i is greater than mid then
        // move the j'th item to the org array, update j
        if(i > mid) org[k] = aux[j++];
        // else if j is greate than hi then
        // move the i'th item to the org array, update i
        else if(j > hi) org[k] = aux[i++];
        // else if the i'th item is smaller than the j'th item,
        else if(aux[i] < aux[j])
        // move the i'th item to the org array, update i
            org[k] = aux[i++];
        else
        // move the j'th item to the org array, update j
            org[k] = aux[j++];
        }
    }
```

The time complexity of this algorithm O($nlog(n)$).

## Optimised Merge sort (OMerge)

The optimisation was quite easy to implement actually; I had to move to the code for copying of the original array, to the public sort method. After that, I had to switch the arguments of the second sort method, and the order of the arguments in the merge method call. Code snippet for the changes made to the code:

```
 public static void sort(int[] org) {
        if (org.length == 0)
        return;
        int[] aux = new int[org.length];
        int lo = 0;
        int hi = aux.length - 1;
        for (int i = lo; i <= hi; i++) {
            aux[i] = org[i];
```

```
        }
        sort(aux, org, 0, org.length -1);

    }
...
private static void sort(int[] org, int[] aux, int lo, int hi) {
        if (lo != hi) {
        int mid = (lo + hi)/2;
        sort(aux, org, lo, mid);

        sort(aux, org, mid+1, hi);
        merge(aux, org, lo, mid, hi);
        }
    }
```

## Benchmarks

I have represented the difference in performances between the 4 algorithms in a table below:

| **Size** | **Select (ms)** | **Insert (ms)** | **Merge (ms)** | **OMerge (ms)** |
|---|---|---|---|---|
| 100 | 157 | 117 | 26 | 61 |
| 200 | 603 | 464 | 81 | 72 |
| 300 | 1363 | 1033 | 121 | 97 |
| 400 | 2353 | 1796 | 188 | 153 |
| 500 | 3654 | 2832 | 249 | 204 |
| 600 | 5297 | 4011 | 302 | 280 |
| 700 | 7286 | 5496 | 365 | 328 |
| 800 | 9476 | 7170 | 419 | 385 |
| 900 | 11992 | 9161 | 484 | 459 |
| 1000 | 14979 | 11377 | 559 | 539 |
| 1100 | 18172 | 13956 | 647 | 577 |
| 1200 | 21576 | 16011 | 691 | 658 |
| 1300 | 25198 | 18800 | 776 | 724 |
| 1400 | 29327 | 21784 | 833 | 794 |
| 1500 | 33562 | 25613 | 926 | 834 |
| 1600 | 38883 | 28696 | 998 | 918 |

Table 1: (Each algorithm had 10 loops, with 10 tries each loop; runtime in microseconds)

As it can be seen, the optimisation has little effect on the algorithm, with only VERY SLIGHT improvements compared to the initial Merge sort.