# COMP3506 Homework 2: Q2

Joel Foster

February 19, 2024

## Solution

Observations:
As shown by figures 1, 3 and 5, selection sort was consistently the longest running for all array sizes. This was followed by selection sort then mergesort and finally the fastest method was quick sort.
Notable outliers are:

- Insert sort took longer than Selection Sort when the arrays were pre-sorted in descending.
- Quick Sort was significantly more efficient when used on a pre-sorted array.
- An outlier was seen when 5 items were used in selection sort on an unsorted array.
- An outlier deviation was seen when 5 items were used in insertion sort on an sorted ascending array.
- Quick Sort was slightly slower than Insertion Sort and Merge Sort when used on a sorted ascending array.

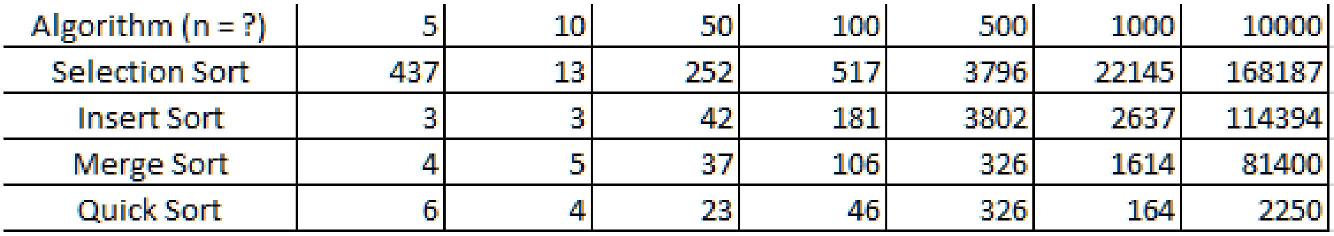
Selection Sort O($n^2$): The increase of compute time as n increases mimics the expected asymptopic bound. A single outlier was found which could have been influenced by several factors such as background process. This behaviour was consistent over the three types of arrays. The worst case was observed when the array was unsorted and a best case when the array was already sorted.

Insertion Sort O($n^2$): The trend of compute time vs array size mimics the expected asymptopic bound. In terms of boundary cases, it does take significantly more time when the array is pre-sorted in descending order as expected. It also takes the least amount of time when the array is already sorted.

Merge Sort O($nLog(n)$): The trend of compute time vs n mimics the expected asymptopic bound. As expected it takes the least amount of time on already sorted array. Both other cases had similar runtimes with the array sorted in descending order taking slightly more time.

Quick Sort Expected Runtime O($nLog(n)$): As shown by the three previously mentioned figure, the implemented quick sort method has a projected runtime of at most O($nLog(n)$). The actual worst case of the algorithm was not observed in the three tested examples however the algorithm took slightly longer to sort unsorted arrays which is expected due to the nature of the algorithm. The runtimes for sorted arrays were similar in progression as n grew.

When comparing the runtime of the algorithms in the three cases, their performances, compared with each other, are what is expected.



| Algorithm (n = ?) | 5 | 10 | 50 | 100 | 500 | 1000 | 10000 |
|---|---|---|---|---|---|---|---|
| Selection Sort | 437 | 13 | 252 | 517 | 3796 | 22145 | 168187 |
| Insert Sort | 3 | 3 | 42 | 181 | 3802 | 2637 | 114394 |
| Merge Sort | 4 | 5 | 37 | 106 | 326 | 1614 | 81400 |
| Quick Sort | 6 | 4 | 23 | 46 | 326 | 164 | 2250 |

Figure 1: Table of Running Time with Unsorted Arrays

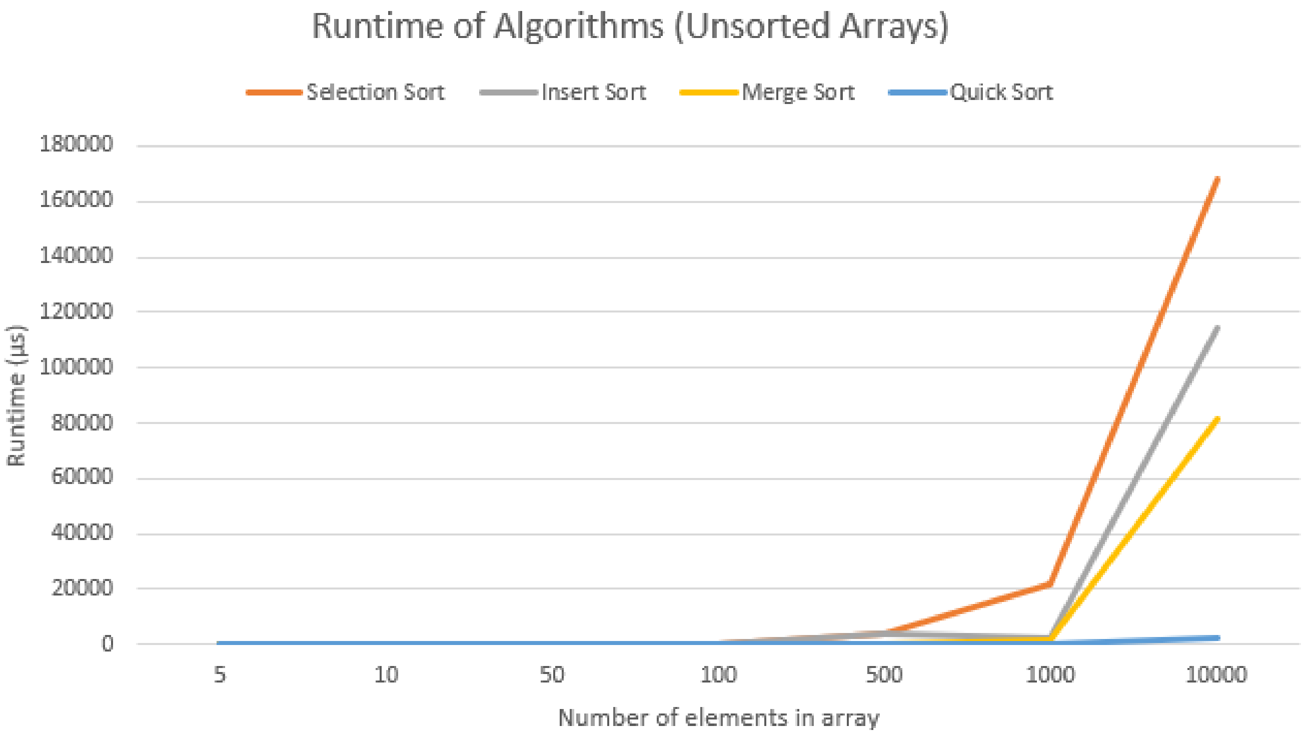
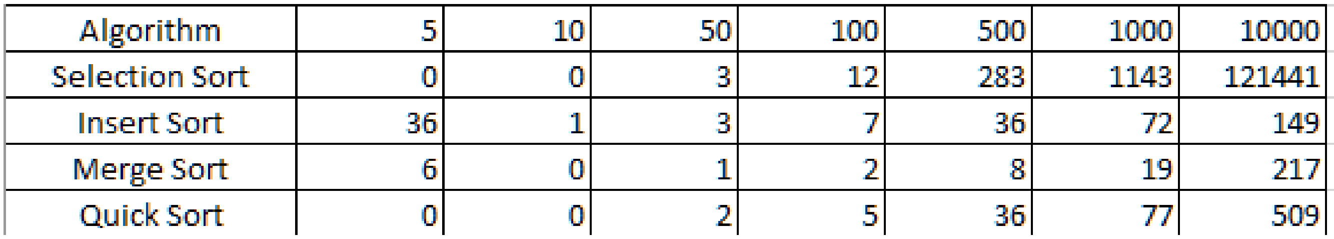


Figure 2: Graph of Running Time with Unsorted Arrays



| Algorithm | 5 | 10 | 50 | 100 | 500 | 1000 | 10000 |
|---|---|---|---|---|---|---|---|
| Selection Sort | 0 | 0 | 3 | 12 | 283 | 1143 | 121441 |
| Insert Sort | 36 | 1 | 3 | 7 | 36 | 72 | 149 |
| Merge Sort | 6 | 0 | 1 | 2 | 8 | 19 | 217 |
| Quick Sort | 0 | 0 | 2 | 5 | 36 | 77 | 509 |

Figure 3: Table of Running Time with Sorted Arrays (ascending)

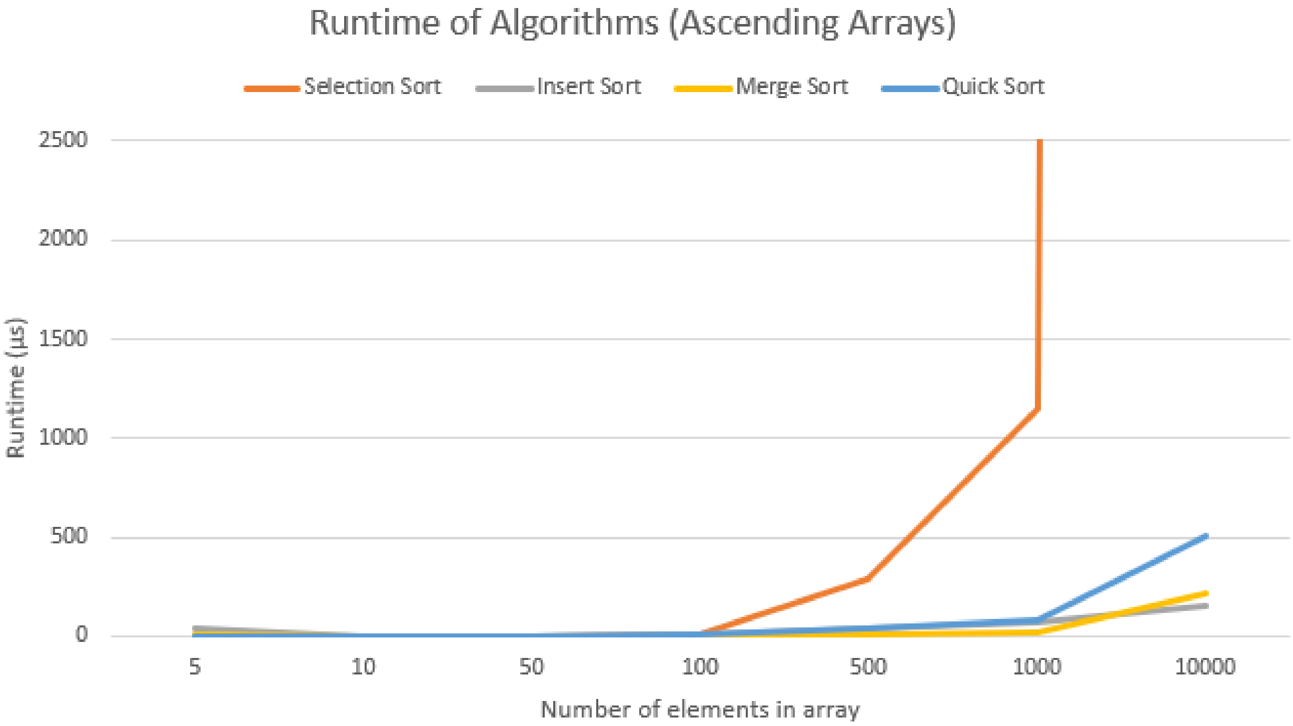
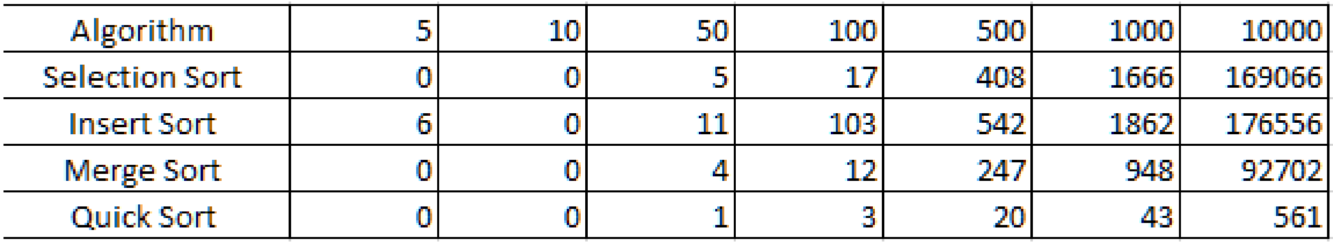


Figure 4: Graph of Running Time with Sorted Arrays (ascending)



| Algorithm | 5 | 10 | 50 | 100 | 500 | 1000 | 10000 |
|---|---|---|---|---|---|---|---|
| Selection Sort | 0 | 0 | 5 | 17 | 408 | 1666 | 169066 |
| Insert Sort | 6 | 0 | 11 | 103 | 542 | 1862 | 176556 |
| Merge Sort | 0 | 0 | 4 | 12 | 247 | 948 | 92702 |
| Quick Sort | 0 | 0 | 1 | 3 | 20 | 43 | 561 |

Figure 5: Table of Running Time with Sorted Arrays (descending)

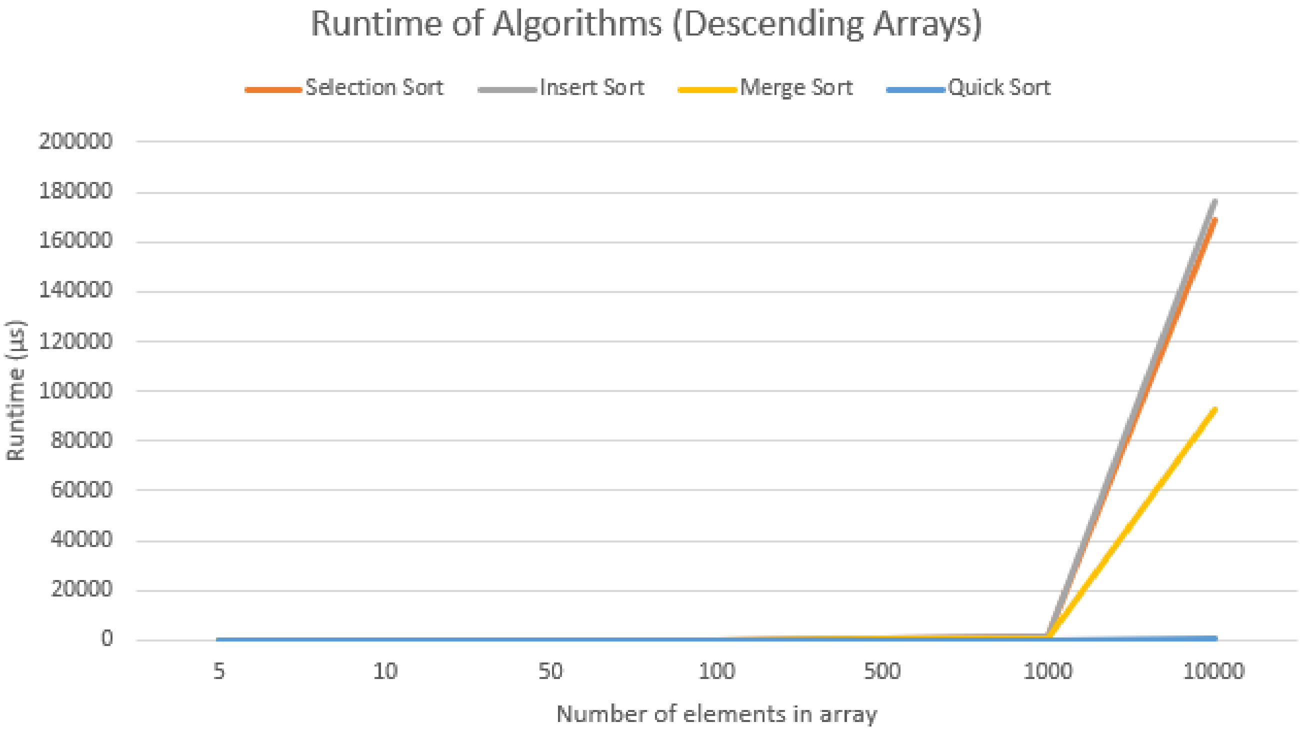


Figure 6: Graph of Running Time with Sorted Arrays (descending)

**Timing Code:**

```
import java.util.Random;

public class TimeAlgorithms {

public static void main(String[] args) {
        int lengths[] = {5, 10, 50, 100, 500, 1000, 10000};
        long[][] unsortedResults = unsortedTests(lengths);
        long[][] sortedAscendingResults = sortedAscendTests(lengths);
        long[][] sortedDescendingResults = sortedDescendTests(lengths);
        return;
}

private static long[][] unsortedTests(int[] lengths) {
        long[][] unsortedResults = new long[lengths.length][4];
        int count = 0;
        for (int n : lengths) {
                // Select Forward
                Integer[] arr = initialiseRandomArray(n);
                unsortedResults[count][0] = timeSelectForward(arr);
                // Insert Forward
                arr = initialiseRandomArray(n);
                unsortedResults[count][1] = timeInsertForward(arr);
                // Merge Forward
                arr = initialiseRandomArray(n);
                unsortedResults[count][2] = timeMergeForward(arr);
                // Quick Forward
                arr = initialiseRandomArray(n);
                unsortedResults[count][3] = timeQuickForward(arr);
                count++;
        }
        return unsortedResults;
}

private static long[][] sortedAscendTests(int[] lengths) {
        long[][] sortedAscendingResults = new long[lengths.length][4];
        int count = 0;
        for (int n : lengths) {
                // Select Forward
                Integer[] arr = initialiseAscendingArray(n);
                sortedAscendingResults[count][0] = timeSelectForward(arr);
                // Insert Forward
                arr = initialiseAscendingArray(n);
                sortedAscendingResults[count][1] = timeInsertForward(arr);
                // Merge Forward
                arr = initialiseAscendingArray(n);
                sortedAscendingResults[count][2] = timeMergeForward(arr);
                // Quick Forward
                arr = initialiseAscendingArray(n);
                sortedAscendingResults[count][3] = timeQuickForward(arr);
                count++;
        }
        return sortedAscendingResults;
}

private static long[][] sortedDescendTests(int[] lengths) {
        long[][] sortedDescendingResults = new long[lengths.length][4];
        int count = 0;
        for (int n : lengths) {
            // Select Forward
            Integer[] arr = initialiseDescendingArray(n);
            sortedDescendingResults[count][0] = timeSelectForward(arr);
            // Insert Forward
            arr = initialiseDescendingArray(n);
            sortedDescendingResults[count][1] = timeInsertForward(arr);
            // Merge Forward
            arr = initialiseDescendingArray(n);
            sortedDescendingResults[count][2] = timeMergeForward(arr);
            // Quick Forward
            arr = initialiseDescendingArray(n);
            sortedDescendingResults[count][3] = timeQuickForward(arr);
            count++;
        }
        return sortedDescendingResults;
    }

    private static long timeSelectForward(Integer arr[]) {
        long start = System.nanoTime();
        SortingAlgorithms.selectionSort(arr, false);
        long end = System.nanoTime();
        return (long) ((end - start) * 0.001);
    }

    private static long timeInsertForward(Integer arr[]) {
        long start = System.nanoTime();
        SortingAlgorithms.insertionSort(arr, false);
        long end = System.nanoTime();
        return (long) ((end - start) * 0.001);
    }

    private static long timeMergeForward(Integer arr[]) {
        long start = System.nanoTime();
        SortingAlgorithms.mergeSort(arr, false);
        long end = System.nanoTime();
        return (long) ((end - start) * 0.001);
    }

    private static long timeQuickForward(Integer arr[]) {
        long start = System.nanoTime();
        SortingAlgorithms.quickSort(arr, false);
        long end = System.nanoTime();
        return (long) ((end - start) * 0.001);
    }

    private static Integer[] initialiseRandomArray(int length) {
        Integer[] arr = new Integer[length];
        Random rd = new Random();
        for (int i = 0; i < length; i++) {
            arr[i] = rd.nextInt(length);
        }
        return arr;
    }
```

```
private static Integer[] initialiseAscendingArray(int length) {
        Integer[] arr = new Integer[length];
        for (int i = 0; i < length; i++) {
                arr[i] = i;
        }
        return arr;
}

private static Integer[] initialiseDescendingArray(int length) {
        Integer[] arr = new Integer[length];
        int j = 0;
        for (int i = length - 1; i >= 0; i--) {
                arr[j] = i;
                j++;
        }
        return arr;
        }
}
```