Sorting Algorithms

Aditya Gupta

September 2022

Introduction

In this assignment, we focus on the efficiency of the different sorting algorithms: selection sort, insertion sort and merge sort. We sort the arrays passed by the user to the function for the different sorting algorithms. Where we sort in the first two algorithms(selection and insertion sort) using an iterative approach and the final algorithm using a recursive approach for the the merge sort algorithm. Here we start with a the most basic algorithm with not much efficiency and we try and progress towards a more efficient approach. We benchmark all these algorithms and then tabulate and graph the data to compare the efficiency of the different algorithms, like how we did in the previous tasks. Here I have chosen to deviate a bit in terms of how I wrote the code for some of the algorithms as I found my way of implementing it somewhat more intuitive than the way it was asked of us to do in the assignment.

Selection Sort

```
public static void selectionSort(int [] inputArray){
        int min, t;
        for (int i = 0; i < inputArray.length - 1; i++) {
            min = i;
            for (int j = i; j < inputArray.length; j++) {
                if(inputArray[min]>inputArray[j]){
                    min = j;
                }
                if(min != i){
                    t = inputArray[min];
                    inputArray[min] = inputArray[i];
                    inputArray[i] = t;
                }
            }
        }
    }
```

In this sorting algorithm, we iterate through the array of elements and try to locate the smallest element and swap the current element with the smallest one. At first, we just set the 1st element as the index of the temporary minimum value and then we keep going from there. If the index for the current temporary minimum value also happens to be the smallest in the array, it continues to remain in its place, otherwise, we swap this value with the smallest value that we come across while iterating. We do this by using 2 nested for loops, one to set the current minimum and then the other for loop to compare and swap the values at these indices. After this, we check if the minimum index we had set still holds the same value or not. If it does not have the same value as before we swap the elements at those two positions. Upon iterating over the entire array and following this procedure, we manage to sort it.
The data we get upon benchmarking the above algorithm for different-sized arrays have been tabulated below:

Case	n	Time(in microseconds)
1	800	59
2	1600	196
3	2400	416
4	3200	715
5	4000	1088
6	4800	1715
7	5600	2356
8	6400	2830

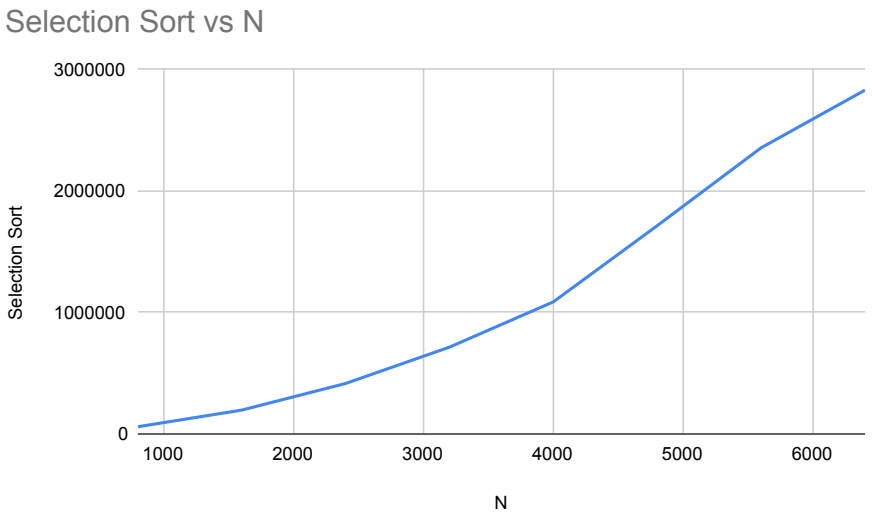

Figure 1: Graph illustrating the Selection Sort Algorithm

Insertion Sort

```
public static void insertionSort(int [] inputArray){
    for (int i = 1; i <inputArray.length ; i++) {
        for (int j = 0; j < i ; j++) {
            if(inputArray[i] < inputArray[j]){
                int t = inputArray[i];
                inputArray[i] = inputArray[j];
                inputArray[j] = t;
            }
        }
    }
}
```

Now, we proceed toward implementing the insertion sort algorithm which is slightly better than the selection sort algorithm. Here the strategy is to compare the values at all indices lower than the current index and swap the values if any of the values on indices lower than the current index is greater than the value at the current index. Here we use the outer loop to specify the current index and the inner loop is used for retrieving values from all indices lower than the current index. This way we quickly manage to sort the array.

The data we get upon benchmarking the above algorithm for different-sized arrays have been tabulated below:

Case	n	Time(in microseconds)
1	800	78
2	1600	194
3	2400	411
4	3200	709
5	4000	1082
6	4800	1713
7	5600	2348
8	6400	2844

Merge Sort

In the previous two algorithms we used an iterative approach to the problem, while in this case, we are going to approach the problem in a recursive fashion. Recursion can be simply defined as an approach to writing functions that call themselves inside their definition. My approach to creating this algorithm deviates from the approach that was suggested in the assignment. I did this as I realised that even though this approach of writing the merge sort algorithm is not the most memory efficient it is a lot faster at executing the task of sorting compared to the suggested approach of creating this algorithm. Aside from this I also found it to be a lot more intuitive for me

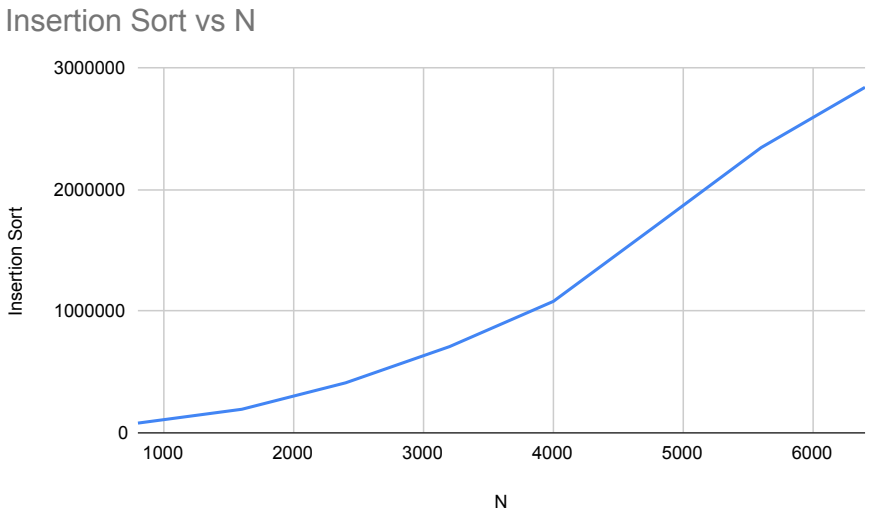

Figure 2: Graph illustrating the Insertion Sort Algorithm

to understand how this algorithm works this way. This implementation comprises of two functions i.e. `mergeSort(int [] inputArray)` and `merge(int [] inputArray, int [] leftHalf, int [] rightHalf)`. In this algorithm, we use the `mergeSort()` function to split the input array into two halves and populate them with the values from the original array's corresponding halves. After this we recursively call the function twice; once with the left half as the argument and after that using the right half as the argument. We do this as long as they're at least 2 elements in the array passed as the input. Thus this algorithm continues to recurse until its splits the original array into smaller arrays of length 1. After we are done with this we merge the values in the two arrays in a manner that the merged array is sorted. We do this by calling the `merge()` function. In that, we compare the first elements of both the arrays and after that, we start adding them into the input array depending on which value is smaller. We insert values this way and increment the index in the same array so that we can start to compare the next value and so on. After adding an element to the input array, we increment its counter so that a value can be added to that index. This way after inserting nearly all the elements into the input array there will be a few leftover elements that we insert by checking the position of the corresponding indices in that smaller array. This approach is not memory efficient compared to the suggested implementation as each time we split the input array into smaller arrays. But it is quite a bit faster than the suggested implementation. Thus from this, we can say that it can be seen as a trade-off, either we can get a fast algorithm which is not space efficient otherwise it's a slower algorithm that does not take up as much memory.

The data we get upon benchmarking the above algorithm for different-sized arrays have been tabulated below:

Case	n	Time(in microseconds)
1	800	30
2	1600	50
3	2400	54
4	3200	58
5	4000	79
6	4800	95
7	5600	116
8	6400	142

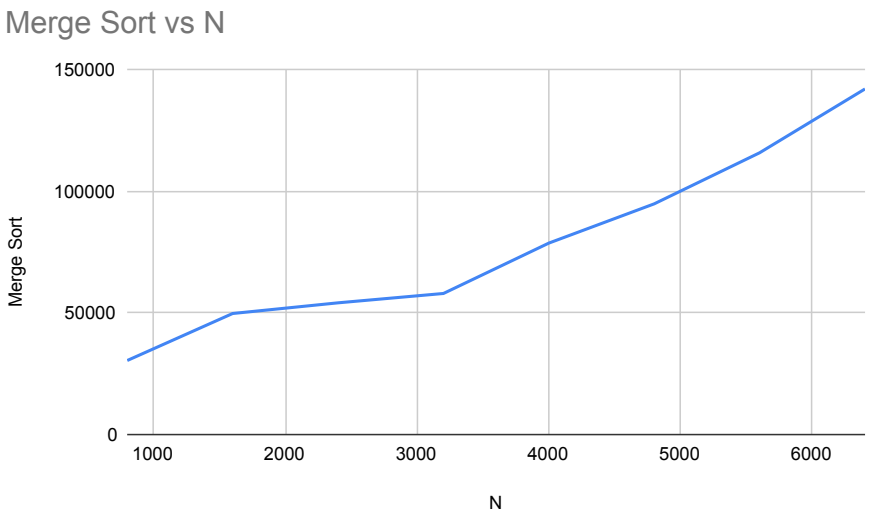

Figure 3: Graph for the search sorted algorithm.

All Data

Here we put all the data from all the previous algorithms that we looked at and compare them using the data tabulated below:

Case	n	Time for Selection Sort (us)	Time for Insertion Sort (us)	Time for Merge Sort (us)
1	800	59	78	30
2	1600	196	194	50
3	2400	416	411	54
4	3200	715	709	58
5	4000	1088	1082	79
6	4800	1715	1713	95
7	5600	2356	2348	116
8	6400	2830	2844	142

We graph the above data, and we can clearly see that the merge sort algorithm is clearly superior to the selection and insertion sort. The data for all the algorithms were retrieved by iterating over all of them 1000 times. The benchmarking was done on arrays of sizes 800 to 6400. From the table above and the graph below we can clearly see that selection sort and insertion sort have a very similar time trend. While the merge sort algorithm is significantly faster. In the graph, we can see that the values for the selection sort and insertion sort happen to overlap, this is because their values are extremely similar. From the graphs, we can deduce the time complexity for the selection sort and insertion sort which should be $O(n^2)$. While for the case of merge sort it should be $O(nlog(n))$ (Where n is the size of the array that we are operating upon).

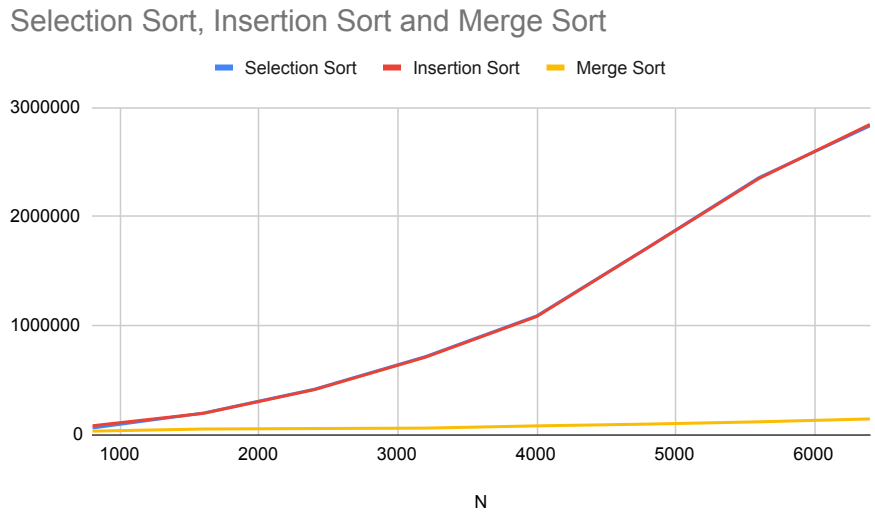

Figure 4: Graph illustrating the Merge Sort Algorithm