ANALYSIS OF ALGORITHMS 1
BLG335E
PROJECT 1

ANALYSIS OF INSERTION SORT AND MERGE SORT

PREPARED BY:

150150058

HASAN H. EROGLU

Table of Contents

Asymptotic Upper Bound on the Running Time

Insertion Sort

Insertion sort's asymptotic upper bound on the running time is **O(n^2)**(assumed that there are n elements in the array).

Insertion sort algorithm starts from the second element of the array and traverses to the end of the array. For every selected element, it compares all of the previous elements, that come before the selected element, with the selected element and if the selected element is less than previous elements their positions are exchanged. To apply the algorithm two **for loop**s are necessary. In the worst-case scenario, both loops loop for **n-1 times**.

As a result, asymptotic upper bound on the running time becomes proportional to **n^2**.

Here is a code snippet (insertion sort function) from my program:

```
void InsertionSort(std::vector<std::string> &rows, std::vector<float>
&numberstosort){
    std::cout<<"Insertion sort is initiated."<<std::endl;
    int k;
    for(int i=1; i < rows.size(); i++){
        k = i;
        for(int j=i-1; j >= 0; j--){
        //Selected element's value is less than previous element's value then
exchange their position
            if(numberstosort[k] < numberstosort[j]){
                float f_temp;
                std::string s_temp;
                //Assign temporary variables
                f_temp = numberstosort[j];
                s_temp = rows[j];
                //Exchange elements
                numberstosort[j] = numberstosort[k];
                rows[j] = rows[k];
                numberstosort[k] = f_temp;
                rows[k] = s_temp;
                k--;
            }
            else{
                break;
            }
        }
    }
    std::cout<<"Insertion sort is completed. Terminating the
program..."<<std::endl;
}
```

Figure 1. Insertion Sort Function

Outer loop always loops for **n-1 times** (which makes the best-case scenario O(n-1) → O(n)). Inner loop loops only 1 time for all selected in the best-case scenario and n-1 times in the worst-case scenario (it can loop for n-1 times for nth element of the array). In my insertion sort function, outer loop traverses from 1 to n-1 and inner loop traverses from n-2 to 0.

That proves that my insertion sort function's asymptotic upper bound on the running time is **O(n^2)**.

Merge Sort

Merge sort's asymptotic upper bound on the running time is **O(nlgn)**.

Merge sort algorithm starts by dividing the array to two sub-arrays. The array is divided as many as possible which means until it gets 1 element sub-arrays. This operation constructs a binary tree. If there are **n** elements in the binary tree, binary tree's depth will be **lgn**. After the dividing part, comparison and merge part will begin. In this part, every 2 sub-arrays, which construct upper sub-array, merges with each other. For comparison, elements are selected from one of these sub-arrays and according to comparison, algorithm puts sub-arrays' element to another array and this array becomes merged array. This operation continues until n-element array is constructed. For every elements' comparison number of iterations in the **for loop** needed is proportional to **n**.

When divide and merge parts combined merge sort's asymptotic upper bound on the running time becomes **O(nlgn)**.

Here is a code snippet (merge sort function) from my program:

```
void MergeSort(std::vector<std::string> &rows, std::vector<float> &numberstosort){
    if(rows.size() <= 1){
        return;
    }

    //Separate vectors to low and high vectors
    std::vector<std::string> rows_lo(rows.begin(), rows.begin() + (rows.size()/2));
    std::vector<std::string> rows_hi(rows.begin() + (rows.size()/2), rows.end());

    std::vector<float> numberstosort_lo(numberstosort.begin(), numberstosort.begin() +
numberstosort.size()/2);
    std::vector<float> numberstosort_hi(numberstosort.begin() + numberstosort.size()/2,
numberstosort.end());


    MergeSort(rows_lo, numberstosort_lo);
    MergeSort(rows_hi, numberstosort_hi);


    int j = 0, k = 0;
    for(int i=0; i < rows.size(); i++){
        if(j >= numberstosort_lo.size()){
            numberstosort[i] = numberstosort_hi[k];
            rows[i] = rows_hi[k];
            k++;
        }
        else if(k >= numberstosort_hi.size()){
            numberstosort[i] = numberstosort_lo[j];
            rows[i] = rows_lo[j];
            j++;
        }
        else{
            if(numberstosort_lo[j] <= numberstosort_hi[k]){
                numberstosort[i] = numberstosort_lo[j];
                rows[i] = rows_lo[j];
                j++;
            }
            else{
                numberstosort[i] = numberstosort_hi[k];
                rows[i] = rows_hi[k];
                k++;
            }
        }
    }
}
```

Figure 2. Merge Sort Function

"MergeSort(rows_lo, numberstosort_lo);" and "MergeSort(rows_high, numberstosort_high);" lines are **recursive calls**. They construct a binary tree with depth **lgn**.

Also, there is a **for loop,** which always has number of iterations proportional to **n,** for compare and merge operation.

That proves my merge sort function's asymptotic upper bound on the running time **O(nlgn)**.

Note: For merge sort it does not matter whether array is in the state of the best case or the worst case. It always divides array as many as possible and uses a for loop to merge and compare elements. Therefore, its cost always proportional to **nlgn**.

Observations

Sorted by the Last Price

N\Algorithm	Insertion Sort	Merge Sort
1000	0.0480 sec	0.0055 sec
5000	1.0176 sec	0.0278 sec
10000	4.4422 sec	0.0587 sec
20000	17.3816 sec	0.1268 sec
40000	68.2978 sec	0.2655 sec
60000	238.298 sec	0.4075 sec
80000	421.226 sec	0.5418 sec
100000	518.6587 sec	0.7094 sec
300000	6237.3 sec	2.2231 sec
500000	14214.7 sec	3.8609 sec
700000	30107.6 sec	5.5911 sec
916722	51182.8 sec	7.7690 sec

Table 1. Average cost of time while sorting by the last price

Note: When preparing table for the last price, I could not get more than one sample for data, that has greater than 60000 elements, for insertion sort since it takes too much time to sort.

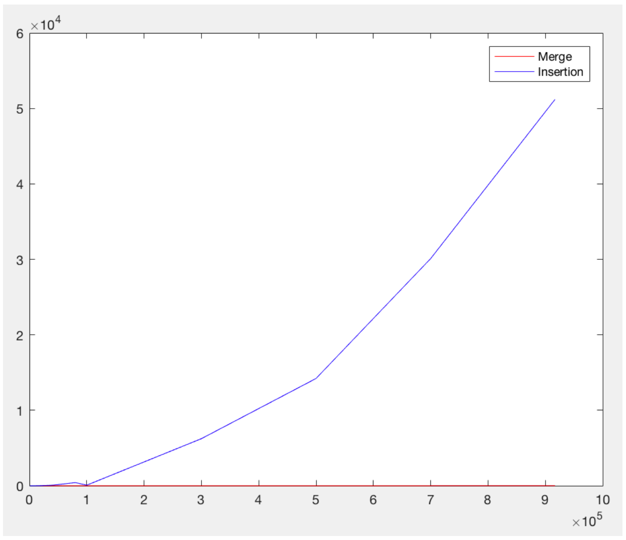

Figure 3. Cost of time while sorting by the last price. Insertion sort vs. Merge Sort

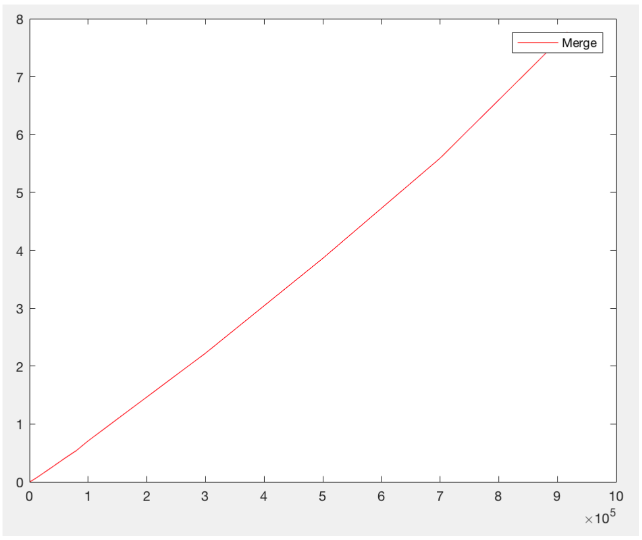

Figure 4. Cost of time while sorting by the last price. Merge Sort

The last price column does not need to be sorted. Then, it could be the worst-case scenario. Upper bound of algorithms in the worst-case scenario:
Merge sort → O(nlgn)
Insertion sort → O(n^2)
In the Figure 3, Insertion sort's time cost too high so that merge sort's curve looks like flatline. But you can see merge sort's curve clearly, in Figure 4.

Sorted by the Timestamp

N\Algorithm	Insertion Sort	Merge Sort
1000	3.6900e-05 sec	0.0049 sec
5000	9.2800e-05 sec	0.0229 sec
10000	0.0148 sec	0.0560 sec
20000	0.0465 sec	0.0997 sec
40000	0.0540 sec	0.2107 sec
60000	0.0769 sec	0.3495 sec
80000	0.1008 sec	0.4428 sec
100000	0.1298 sec	0.5689 sec
300000	0.3573 sec	1.8154 sec
500000	0.5636 sec	3.2024 sec
700000	0.8164 sec	4.4224 sec
916722	1.1030 sec	6.2343 sec

Table 2. Cost of time while sorting by the timestamp.

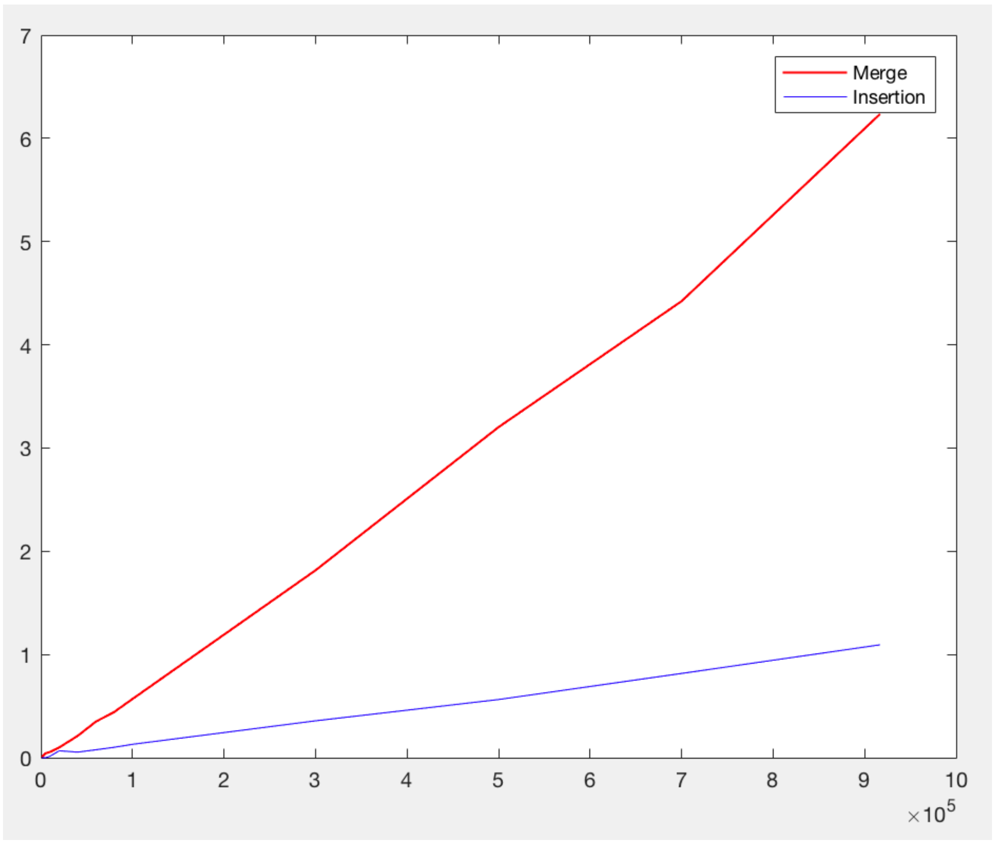

Figure 5. Cost of time while sorting by the timestamp. Insertion sort vs. Merge Sort

Timestamps are already sorted in the array which makes insertion sort faster than merge sort. Already sorted array is the best-case scenario. Upper bound of algorithms in the best-case scenario:
Merge sort → O(nlgn)
Insertion sort → O(n) (for the best-case scenario)
From the Figure 4, it is easy to see that insertion sort's plot is almost linear.
nlgn > n for all positive n values. Therefore, in the Figure 5 merge sort's plot is always above the insertion's sort plot.

Putting an element to the end of the sorted array

An element is already sorted (Best-case)

If the element is already sorted, which means its value greater other element in the array, insertion sort will sort faster than merge sort. Reason for that is insertion sort's cost depends on the array (input order). If the array is already sorted (the best-case scenario for insertion sort), insertion sort's cost will be **O(n)**. Since merge sort cost does not depend on the scenario, it will be **O(nlgn)**.

An element is not sorted (Worst-case)

If the element is not sorted, my preference will be merge sort. The element can be the smallest value of the array, which is the worst case, then insertion sort's cost will be **$O(n^2)$** but merge sort's cost still will be **O(nlgn)** (because it does not depend on input). Insertion will sort faster than merge sort, again. Actually, I expected to see merge sort faster than insertion sort. Since **$O(n^2) = c_1n^2$** and **$O(nlgn) = c_2nlgn$**, **c_1** and **c_2** can differ. When an element, that is not sorted, added to the dataset, since it is only an element: $c_1 < c_2$. These are the results I got when I put an element to the end of the array:

	Worst-Case	Best-Case
Merge Sort	5.4853 sec	6.42868 sec
Insertion Sort	0.210106 sec	0.011588 sec

Table 3. Comparison of worst-case and best-case for merge sort and insertion sort. (an element added)

In the table 3, you can see that insertion sort performs better than merge sort in both cases. It is because, we added only an element. If we add 40 elements that are the smallest elements of the dataset, we will get table below:

	Worst-Case
Merge Sort	7.07148 sec
Insertion Sort	7.09934 sec

Table 4. Comparison of worst-case for merge sort and insertion sort. (40 elements added)

As you can see cost of time of insertion sort increases faster than merge sort's cost of time. If we add 100 elements, we will get table below:

	Worst-Case
Merge Sort	6.45405 sec
Insertion Sort	23.0729 sec

Table 5. Comparison of worst-case for merge sort and insertion sort. (100 elements added)

After adding 100 elements, we can easily see insertion sort is not stable while merge sort's cost of time stays between fixed range.

Conclusion

Real world is complicated and unexpectable. For this report, I have worked with the dataset with 916722 elements. In the future, it is possible for me to work with datasets larger than this dataset. Insertion sort could be faster than merge sort for some cases, but merge sort is more reliable and stable than insertion sort. Therefore, my preference will be merge sort for both cases, even insertion sort works faster than merge sort in the best-case.

Notes for the code

I have used some functions, that are supported with C++11. Hence, you should use this command to compile with no errors and warnings:

```
g++ main.cpp -std=c++11
```