BLG335E - Analysis of Algorithms

Mareva Zenelaj

150150906

Project 1 - Report

A.

The asymptotic upper bound on the running time of **merge sort** is said to be $\boldsymbol{O(n \log_2 n)}$. To prove the statement empirically I have included the running time of merge sort with two different number of instances:

If n = 10,000, runtime = 70,000; if n = 1,000,000, runtime = 14,840,000; After calculations it is seen that $n \log_2 n = 10{,}000 * \log_2 10{,}000 = 92{,}103.40$ which is indeed larger than 70,000. This proves that $n \log_2 n$ is an asymptotically upper bound for merge sort. Also, as seen $n \log_2 n = 1{,}000{,}000 * \log_2 1{,}000{,}000 = 13{,}85{,}510.56$, which again proves the statement since nlog2n for n = 1,000,000 is larger than the actual runtime.

Likewise, regarding insertion sort for n = 1,000 the runtime resulted to be 60,000 and for n = 10,000, the runtime was 5,400,000. Since the asymptotic upper bound of **insertion sort** is said to be $\boldsymbol{O(n^2)}$. For n = 1,000, n^2 = 1,000,000 which is larger than the actual runtime it takes for insertion sort to finish sorting. For n = 10,000, n^2 = 100,000,000 and again this is larger than the obtained runtime.

Considering the comparisons for both merge sort and insertion sort, it can be said that the implementations of the algorithms result in values that fit to the stated upper bounds.

If we take into consideration insertion sort whose implementation is shown in Figure 1, the worst

```
void insertion_sort(Items array_[], int n, char parameter){
    Items key;
    int i;
    for (int j = 2; j < n; j++){
        key = array_[j];
        i = j - 1;
        while (i >= 0 && !compare_class_objects(array_[i], key, parameter)){
            array_[i + 1] = array_[i];
            i--;
        }
        array_[i + 1] = key;
    }
}
```

Figure 1

case scenario would be if the array is sorted in a descending order and both the for and while loops would have to parse n elements making a total of n*n iterations.

Whereas in merge sort as shown in Figure 2, *merge_sort* is called recursively which makes the

```
void merge_sort(Items array_[], int low, int high, char parameter){
    int mid;
    if (low < high){
        mid = (low + high) / 2;
        merge_sort(array_, low, mid, parameter);
        merge_sort(array_, mid + 1, high, parameter);
        merge_sort(array_, low, mid, high, parameter);
    }
}
```

Figure 2

calculations of its complexity more difficult than insertion sort. However, for each subarray that is called there's a complexity of log2n + 1, since we have n of such arrays then consequently the overall upper bound would $n(\log_2 n)$ which is again what the worst-case complexity of merge sort is said to be.

B.

Both algorithms were run with ten times for different number of inputs with criterion to be p. The following table was created from the obtained results when executing merge sort:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**avg**
1000	0.01	0	0.01	0.01	0.01	0.01	0.01	0	0.01	0.01	0.008
5000	0.03	0.03	0.03	0.03	0.03	0.02	0.03	0.03	0.03	0.03	0.029
10000	0.06	0.06	0.06	0.06	0.05	0.07	0.06	0.05	0.06	0.07	0.06
50000	0.36	0.41	0.42	0.33	0.4	0.37	0.39	0.42	0.4	0.4	0.39
100000	0.94	0.85	0.89	0.86	0.91	0.92	0.94	0.89	0.88	0.9	0.898

Whereas for insertion sort:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**avg**
1000	0.5	0.6	0.6	0.6	0.5	0.6	0.5	0.6	0.5	0.5	0.55
5000	1.33	1.33	1.33	1.36	1.33	1.34	1.31	1.34	1.34	1.33	1.334
10000	6.09	5.49	5.47	5.5	5.62	5.6	5.42	5.5	5.18	5.46	5.533
50000	170.64	167.34	213.66	226.14	187.76	175.49	186.03	178.52	202.52	200.95	190.91

For insertion sort, I tried to run the algorithm with an input of 100000, yet the process was killed and I could not retrieve an output.

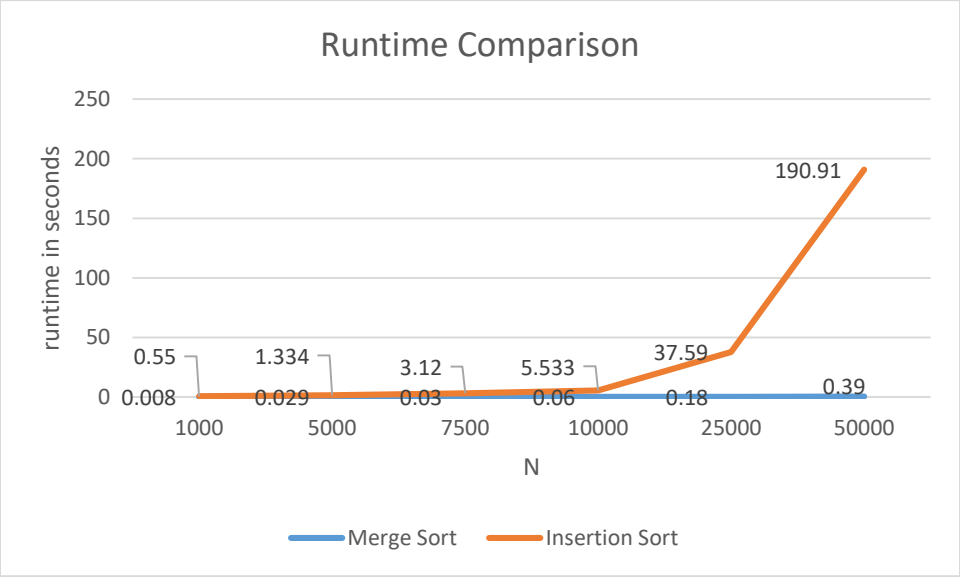
While getting the results after running the code in ssh with the required parameters and N to be less than 1000, some of the runtime results were 0. The result is merely the difference between start time before entering the sorting algorithm and the end time after finishing that is calculated as such:

cout << ((double)after_time - (double)before_time)/CLOCKS_PER_CYCLE << endl;

Note: To retrieve the execution time for both algorithms where the criterion is set to be timestamp, I used the output sorted.csv because the data would be shuffled and timestamp would not be already sorted as it is in log_inf.csv.

C.

The data retrieved from running both algorithms with different values of N was converted to the graph below:



As known the upper bound for merge sort is $O(n\log_2 n)$ whereas for insertion sort it is $O(n^2)$. These upper bounds are seen in the graph where insertion sort has a roughly parabolic growth and merge sort's is somewhat logarithmic. The numbers retrieved are also explained in part A.

D.

For this last part, I got a line from log_inf.csv at around the 650,000 entry and set it to a variable named new_data which will be added to the sorted array. After that, I run the new code for both insertion sort and merge sort algorithms with N to be 100,000 and the results about runtime were 0.69s for merge sort and 0.03 for insertion sort. This is excepted since insertion sort would parse through the array until it finds the position of new_data and this process would be of a complexity of n, which is also the best case scenario for insertion sort, meaning the algorithm cannot perform better than O(n). Whereas in merge sort, it is said to behave exactly the same way for any kind of input. Therefore, in this case, insertion sort is more efficient than merge sort which can be also seen in the difference of run times that is of a factor of 23.

For an unsorted array of N = 100000, insertion sort killed the process, whereas for a sorted version it completes within 0.03 seconds.