1.Merge Sort and Insertion Sort Program

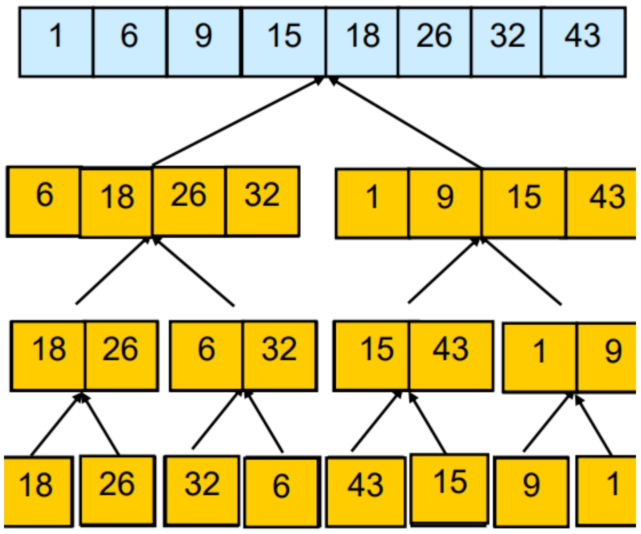
a. Merge Sort Pseudo Code (Arr[1~n])

I. If start index GTE(greater than or equal to) last index, then done.

II. (else) Recursively divide the array. (Arr[1 ~ n/2], Arr[n/2+1 ~ n]).

III. Merge the recursively divided array in ascending order.

(i.e.) Sorting the recursively divided array in merging process



b. Insertion Sort Pseudo Code (Arr[1~n])

I. For(·j = 2~n)

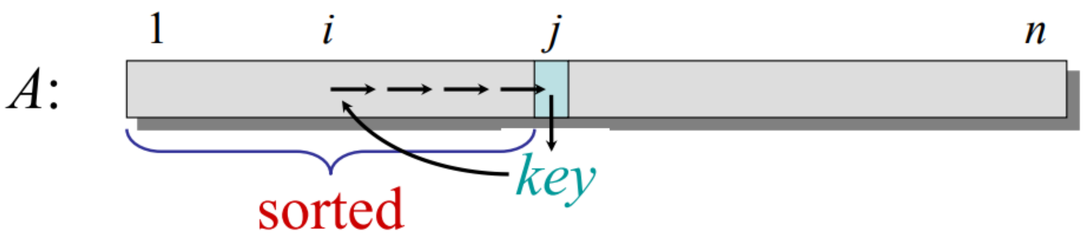
II. In the first loop, key = Arr[j].

III. For(I = j-1~1, and key < Arr[i])

IV. In the second loop, Arr[i+1] = Arr[i]

V. Out of the second loop and in the first loop, Arr[i+1] = key

(Ex) The process of insertion sort



2. Merge Sort vs Insertion Sort Running Time analysis

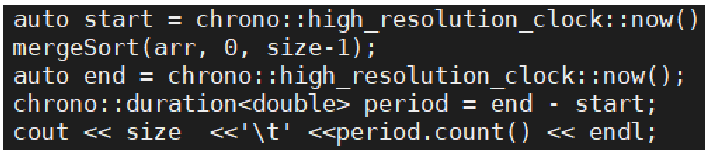
<How I modified the code to collect the running time>

In order to collect the running time, I used 'chrono' C++ library. 'chrono' library is a library which enables a flexible collection of types that track time with varying degrees of precision. I used 'high_resolution_clock' which is the clock with the shortest tick period available in order to measure the execution time of the sorting function as possible as precise.

(From: https://en.cppreference.com/w/cpp/chrono)

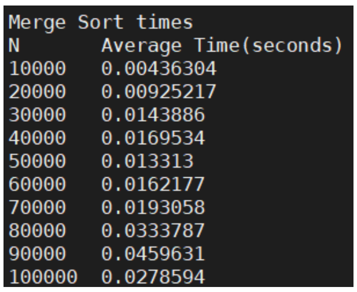
I set the start variable a line before I call the merge(or insertion) sort function, and then call the end variable after a line after I call the merge sort function. Then I subtract the start variable from the end variable then assigned it to period variable.

- Merge sort



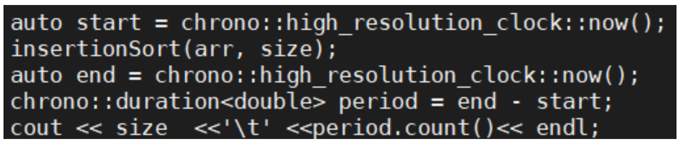
```
auto start = chrono::high_resolution_clock::now()
mergeSort(arr, 0, size-1);
auto end = chrono::high_resolution_clock::now();
chrono::duration<double> period = end - start;
cout << size  <<'\t' <<period.count() << endl;
```

Output:



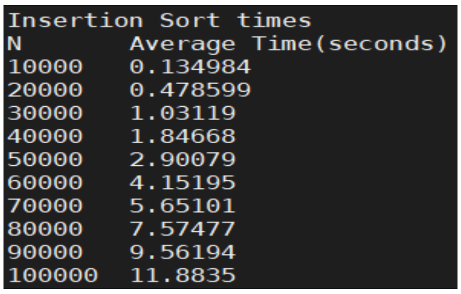
```
Merge Sort times
N       Average Time(seconds)
10000   0.00436304
20000   0.00925217
30000   0.0143886
40000   0.0169534
50000   0.013313
60000   0.0162177
70000   0.0193058
80000   0.0333787
90000   0.0459631
100000  0.0278594
```

- Insertion sort



```
auto start = chrono::high_resolution_clock::now();
insertionSort(arr, size);
auto end = chrono::high_resolution_clock::now();
chrono::duration<double> period = end - start;
cout << size  <<'\t' <<period.count()<< endl;
```

Output:



```
Insertion Sort times
N        Average Time(seconds)
10000    0.134984
20000    0.478599
30000    1.03119
40000    1.84668
50000    2.90079
60000    4.15195
70000    5.65101
80000    7.57477
90000    9.56194
100000   11.8835
```

3. Written Report and Data analysis

a. Collect Running Times

For collecting running times, I ran each algorithm 10 times for each size of array and made the average running times. The reason is that I wanted to collect the running time as general as I can by executing the algorithm for 100 times, but it takes too much time to collect the data. Therefore, I decided to execute the algorithm 10 times for each size of n.

Moreover, I used 10 different array size starting from 10,000 to 100,000.

- Insertion Sort Running Time collection

	Average	Best	Worst
10000	0.1117	0.0001287	0.2929
20000	0.4724	0.0002503	0.9899
30000	1.0056	0.0003869	2.073
40000	1.812	0.0005063	3.683
50000	2.824	0.0006321	5.888
60000	4.003	0.0007376	8.450
70000	5.463	0.0008580	11.467
80000	7.107	0.0009756	14.614
90000	9.003	0.001121	18.639
100000	11.520	0.001219	23.579

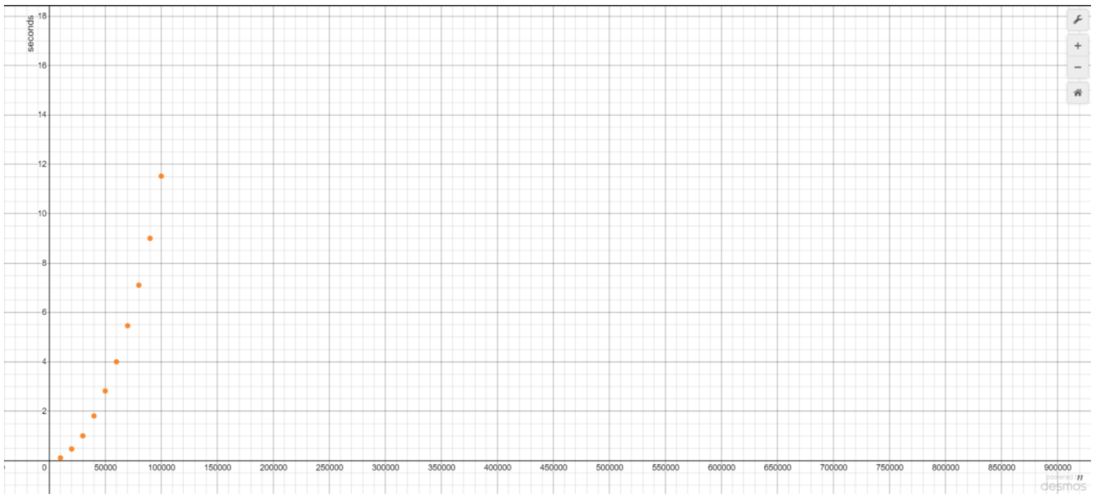
- Merge Sort Running Time collection

	Average	Best	Worst
10000	0.00235	0.001629	0.001627
20000	0.00496	0.003370	0.003332
30000	0.00780	0.005159	0.005148
40000	0.01025	0.006993	0.007178
50000	0.01311	0.008870	0.008957
60000	0.01609	0.01083	0.01090
70000	0.01885	0.01315	0.01295
80000	0.02195	0.01482	0.01445
90000	0.02452	0.01688	0.01632
100000	0.02738	0.01876	0.01880

(Average Case: randomly generated numbers, Best Case: Already Sorted, Worst Case: Reversely sorted)

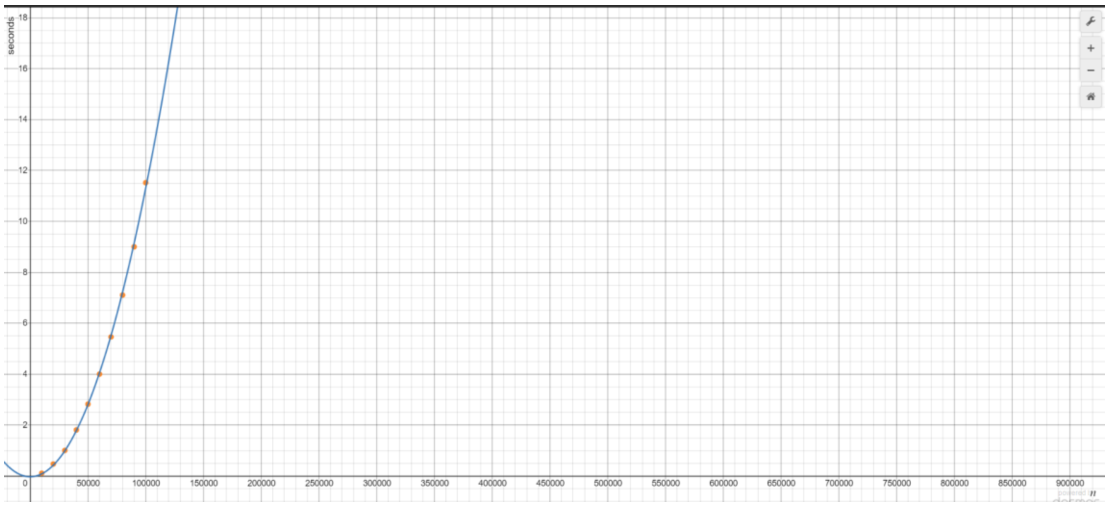
b. Plot and fit a curve

i.Insertion Sort



(Scatter Plot of average case of insertion sort)

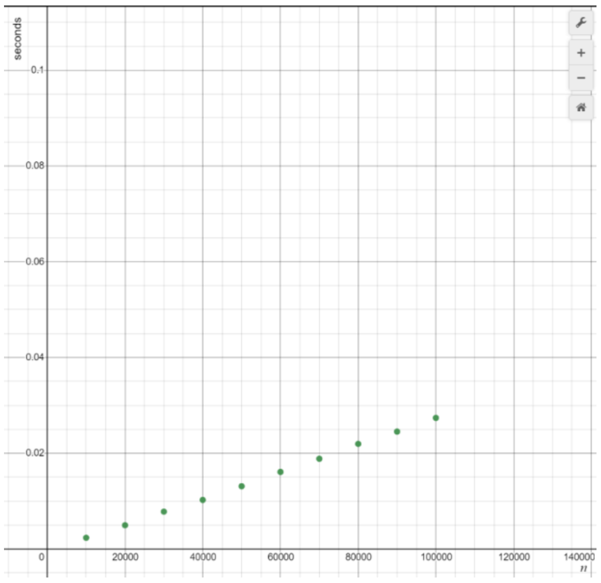
In case of insertion sort, quadratic curve best fits for the data set. The reason is that the time complexity of insertion sort is $O(n^2)$. The equation of curve is $T(n) = 1.1328 * 10^{-9} * n^2 - 0.0291642$.



(The curve equation: $T(n) = 1.1328 * 10^{-9} * n^2 - 0.0291642$)

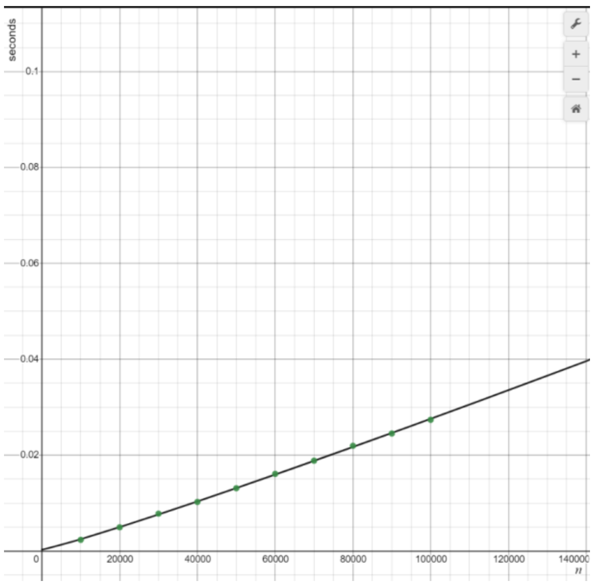
Although not all of experimental running times exactly fits theoretical running times(the curve), but all of the running times are approximate to theoretical running times(the curve).

ii. Merge Sort



(Scatter Plot of average case of merge sort)

Logarithmic curve best fits the data set. The reason is that the time complexity of merge sort is $O(nlgn)$. The equation of the curve is $T(n) = 1.5203 * 10^{-8} * nlog_2n + 0.015603$.

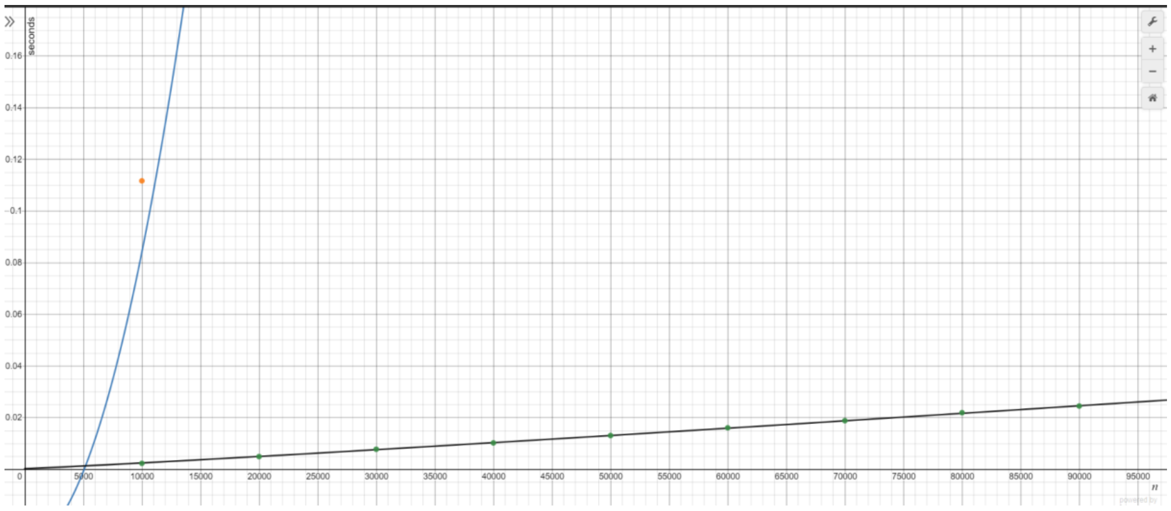


(The equation of curve : $T(n) = 1.6415 * 10^{-8} * nlog_2n + 0.00031243$)

Although not all of experimental running times exactly fit theoretical running times(the curve), all of the running times are approximate to theoretical running times(the curve).

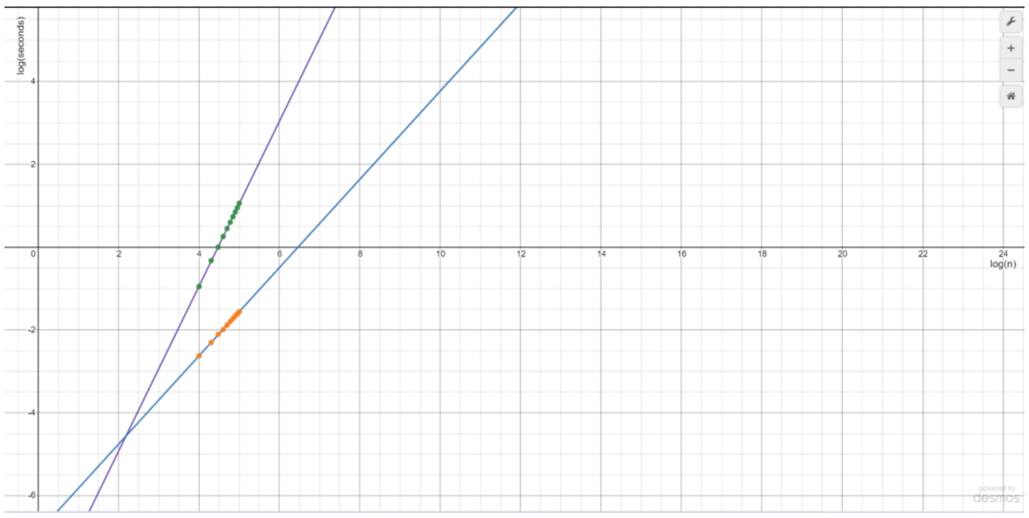
c. Combine

<The combine of Insertion Sort Running Time and Merge Sort Running Time>



(Black line with green dots: Merge sort data, Blue lines with orange dots: Insertion sort data)

<The combine of Insertion Sort Running Time and Merge Sort Running Time in log-log plot>



(Purple line with green dots: Insertion sort plot, Blue line with orange dots: Merge sort plot)

d. Prediction

I. Insertion Sort

Since the theoretical running time of insertion sort is $T(n) = 1.1328 * 10^{-9} * n^2 - 0.0291642$, the running time when the size of n is 500,000 is T(500,000) = 283.1708358. Thus, the theoretical prediction of running time when the size of n is 500,000 is about 283 seconds in three significant figures.

II. Merge Sort

Since the theoretical running time of merge sort is $T(n) = 1.6415 * 10^{-8} * nlog_2 n + 0.00031243$, the running time when the size of n is 500,000 is T(500,000) = 0.155693279. Therefore, the theoretical prediction of running time when the size of n is 500,000 is about 0.16 seconds in two significant figures.