AUBURN UNIVERSITY

CPSC 3273 Programming Assignment 3

Objectives of this assignment:

- To compare the running time performance of InsertSort and MergeSort,

What you need to do:

1. (20 points) Implement the *InsertSort* and *MergeSort* algorithms to sort an array.
2. (5 points) Collect the execution time T(n) as a function of n for the two algorithms
3. (20 points) Plot on the same graph the running time of the two algorithms.
4. (15 points) Using a **pertinent graph/plot** with your data (**hint**: look at previous programming assignments), show/illustrate what the time complexity of *InsertSort* is.
5. (15 points) Using a **pertinent graph/plot** with your data (**hint**: look at previous programming assignments), show/illustrate what the time complexity of *MergeSort* is.
6. (25 points) Discuss the results comparing the two algorithms.

Objective:

The objective of this programming assignment is to implement in Java the *InsertSort* and *MergeSort* algorithms presented in the lectures to sort a list of numbers. We are interested in comparing the two algorithms. For this exploration, you will collect the execution time T(n) as a function of n and plot on the same graph the execution times T(n) of the two algorithms. Finally, discuss your results.

Program to implement

```
collectData()
      Generate an array G of HUGE length L (as huge as your language allows)
      with random values capped at 0x7ffffffe. for n = 4000 to L (with step
      1,000)
            for each algorithm InsertSort and MergeSort do
                  copy in Array A n first values from Array G
                  Start timing // We time the sorting of Array A of length n
                  Sort A using one of the two algorithms.
                  Store the value n and the value T(n) in a file F where T(n)
                  is the execution time
                  //Think here about value(s) to collect for the pertinent
                  graph/plot for questions 4, 5, and 6
```

Data Analysis

Use any plotting software (e.g., Excel) to plot the values T(n) in File F as a function of n. File F is the file produced by the program you implemented. Discuss your results based on the plots.

Report

- Write a report that will contain, explain, and discuss the plot. The report should not exceed one page.
- In addition, your report must contain the following information:
  - whether the program works or not (this must be just ONE sentence)
  - the directions to compile and execute your program
- Good writing is expected.
- Recall that answers must be well written, documented, justified, and presented to get full credit.

AUBURN UNIVERSITY

CPSC 3273 Programming Assignment 3

Grading

Directions for compiling and executing the program:

1. Type `javac ProgrammingAssignment3.java` – *this will compile the program*
2. Type `java ProgrammingAssignment3` and this will execute the program – *it will take some time to execute*
3. Type `ls` once the program is done executing and you should see the files `ProgrammingAssignment3.java`, `ProgrammingAssignment3.class`, `programming3merge.csv`, and `programming3insert.csv`

1. (**20 points**) Implement the *InsertSort* and *MergeSort* algorithms to sort an array.

Code is included in `ProgrammingAssignment3.java`. The program `ProgrammingAssignment3.java` works as intended and provided valid data points for graphing.

2. (**5 points**) Collect the execution time T(n) as a function of n for the two algorithms

The execution time T(n) as a function of n for the two algorithms was collected and shown in the graph below:

3. (**20 points**) Plot on the same graph the running time of the two algorithms.
(**25 points**) Discuss the results comparing the two algorithms.

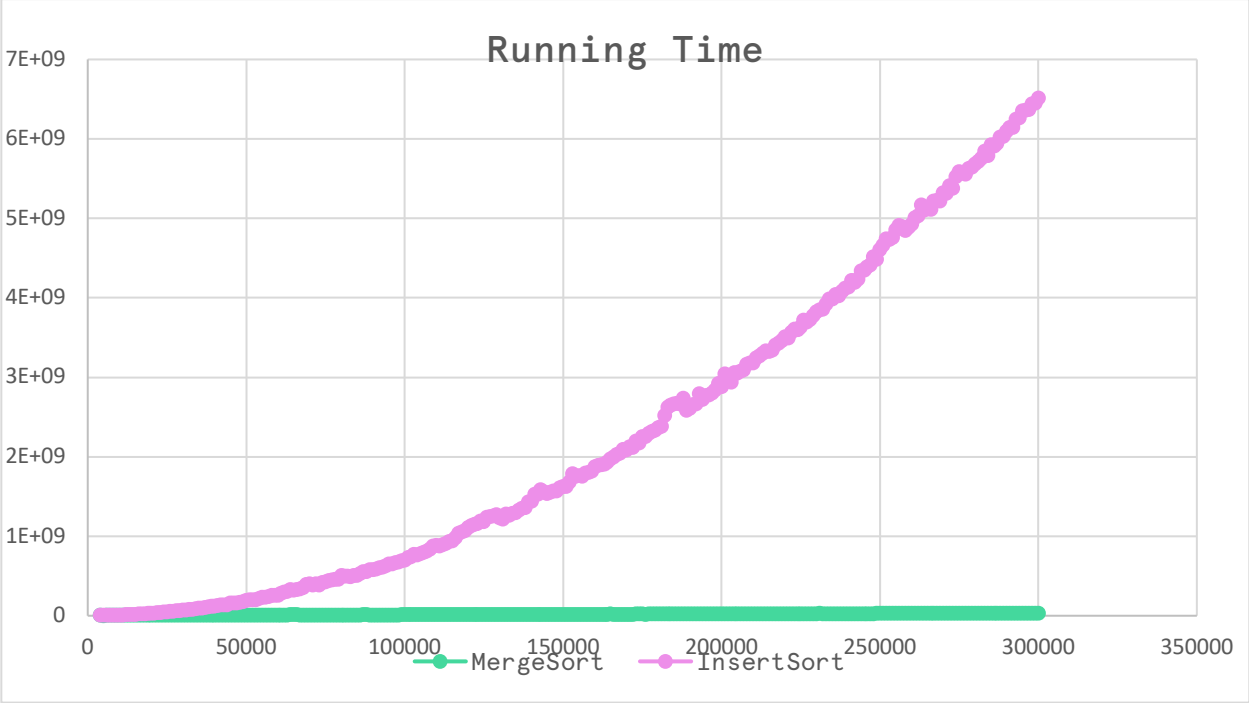

Figure 1

This is the graph for the running time ***T(n)*** of `MergeSort` and `InsertSort`. The X-axis represents the input size ***n*** and Y-axis represent ***T(n).*** Let $T_1(n)$ represent `MergeSort` and $T_2(n)$ represent `InsertSort`.

As we have learned throughout this course, the time complexity of $T_1(n)$ is $\theta(nlog(n))$ for the best, average and worst case. We also know that the time complexity of $T_2(n)$ is $\theta(n^2)$ for the worst and average case, and $\theta(n)$ for the best case. Since we input an array of randomly chosen numbers, $T_2(n)$ will neither be worst nor best case. What we do know, is that whatever the case may be, $T_2(n)$ will always grow faster and higher than $T_1(n)$. This means that the graph of $T_2(n)$ should be an upward curving line, while $T_1(n)$ will appear horizontal due to the difference in time complexity. Such comparisons have been made throughout our lectures and quizzes, comparing n^2 to $nlog(n)$.

As we can observe in **Figure 1**, $T_2(n)$ dwarfs $T_1(n)$ because the running time of $T_2(n)$ grows faster and higher. This means that $T_1(n)$ is a better algorithm than $T_2(n)$ for sorting, in terms of time complexity. This also confirms our prediction that the graph for `InsertSort` will move in a curved upward trend while `MergeSort` will appear horizontal because of the difference in time complexity. In the graphs below, we will show and confirm that this comparison of `MergeSort` and `InsertSort` is accurate, and that `MergeSort` is a better algorithm than `InsertSort`, no matter the case.

4. (**15 points**) Using a **pertinent graph/plot** with your data (**hint**: look at previous programming assignments), show/illustrate what the time complexity of *InsertSort* is.

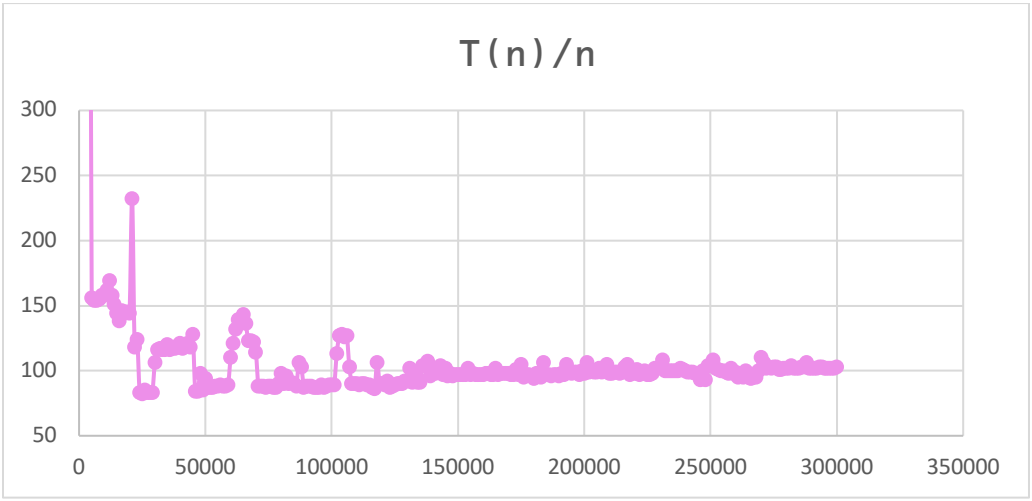

Figure 2

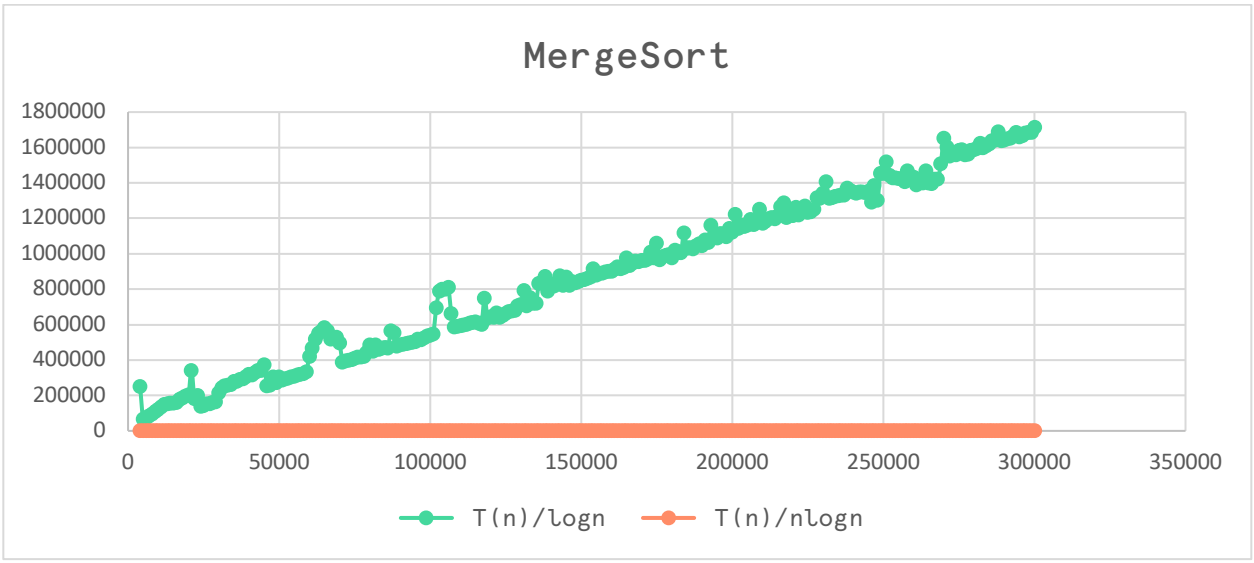

Figure 3

As we have learned throughout this course, the time complexity of `MergeSort` is $\theta(nlog(n))$ for the best, average and worst case, which means that $T(n) \in \theta(nlog(n))$. For large values of n, $T(n) \approx Knlog(n)$ where K is some constant. Therefore:

$\frac{T(n)}{n} \approx K.\log(n)$ this graph is log, so it would look pretty much horizontal, but slightly increasing for larger values of n.
$\frac{T(n)}{\log(n)} \approx K.n$ this graph would be a line sloping upwards with slope K (goes higher than the log graph).
$\frac{T(n)}{nlog(n)} \approx K$this graph would be a horizontal line where $y = K$.

Now let's take a look at the results from our `MergeSort` graph. Let $f_1(n)$ represent $\frac{T(n)}{n}$, $f_2(n)$ represent $\frac{T(n)}{\log(n)}$, and $f_3(n)$ represent $\frac{T(n)}{nlog(n)}$. The graphs for $f_1(n)$ $f_2(n)$ and $f_3(n)$ all confirms our predictions above. For $f_1(n)$ it means that $T(n)$ grows faster than n, exemplifying that the time complexity is not n. For $f_2(n)$ it means that $T(n)$ grows faster than $log(n)$, exemplifying that the time complexity is not n. For $f_3(n)$ it means that $T(n)$ grows at a constant rate K when compared to $nlog(n)$ verifying that the time complexity is ***nlog(n)***.

5. (**15 points**) Using a **pertinent graph/plot** with your data (**hint**: look at previous programming assignments), show/illustrate what the time complexity of *MergeSort* is.

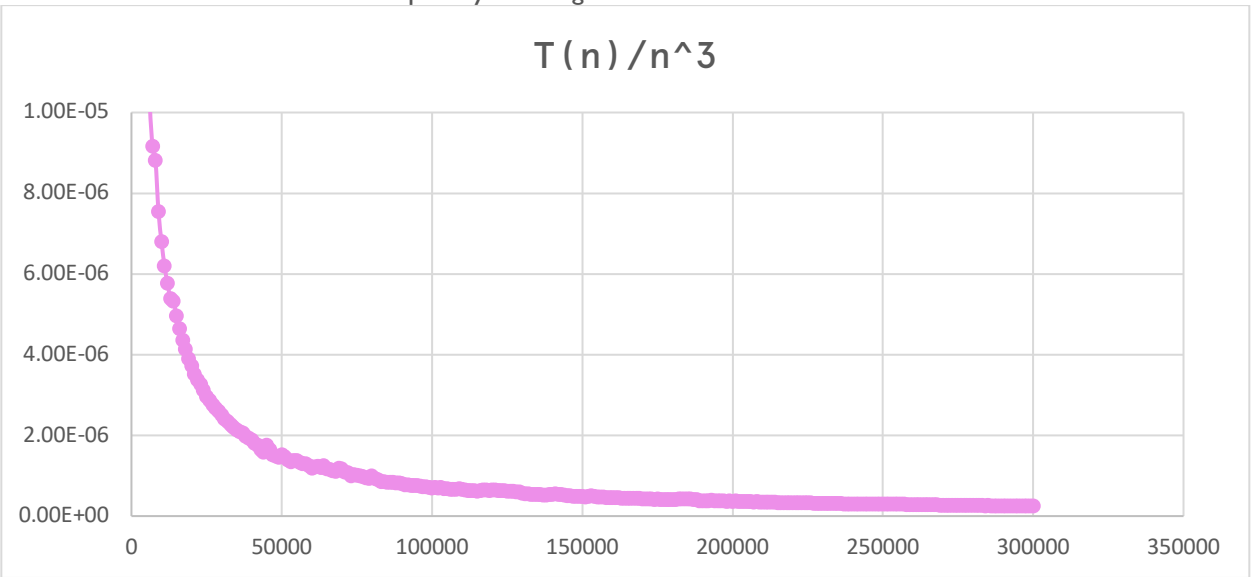

Figure 4

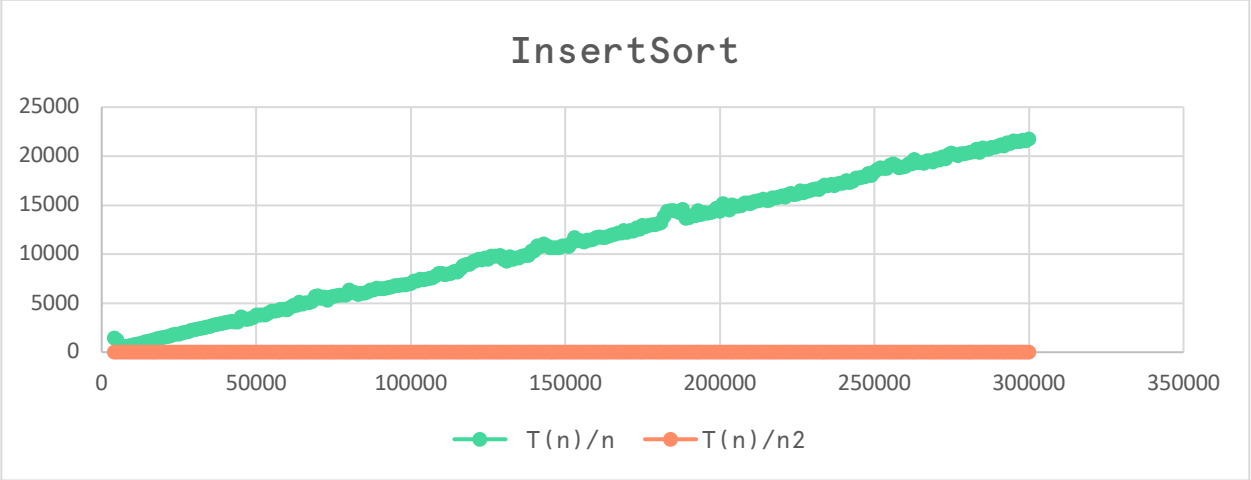

Figure 5

As we have learned in our most recent module, the time complexity of `InsertSort` is $\theta(n^2)$ for the average and worst case, and $\theta(n)$ for the best case. Since our input is an array of random numbers, it is most likely that it will be neither the best nor worst case, so we will approach this as an average case scenario, which means that $T(n) \in \theta(n^2)$. For large values of n, $T(n) \approx K(n^2)$ where K is some constant. Therefore:

$\frac{T(n)}{n^3} \approx K.\frac{1}{n}$ this graph would be pretty much horizontal, but for large values of n, it will slope downwards slightly, but never reach 0.
$\frac{T(n)}{n} \approx K.n$ this graph would be a line sloping upwards with slope K.
$\frac{T(n)}{n^2} \approx K$ this graph would be a horizontal line where $y = K$.
Now let's take a look at the results from our `InsertSort` graph. Let $f_1(n)$ represent $\frac{T(n)}{n^3}$, $f_2(n)$ represent $\frac{T(n)}{n}$, and $f_3(n)$ represent $\frac{T(n)}{n^2}$. The graphs for $f_1(n)$ $f_2(n)$ and $f_3(n)$ all confirm our predictions above. For $f_1(n)$ it means that n^3 grows faster than $T(n)$, exemplifying that the time complexity is not n^3. For $f_2(n)$ it means that $T(n)$ grows faster than n, exemplifying that the time complexity is not n. Based on those results, the time complexity **has** to be in between n and n^3. Further confirming, for $f_3(n)$ it means that $T(n)$ grows at a constant rate K when compared to n^2 verifying that the time complexity is $\boldsymbol{n^2}$.