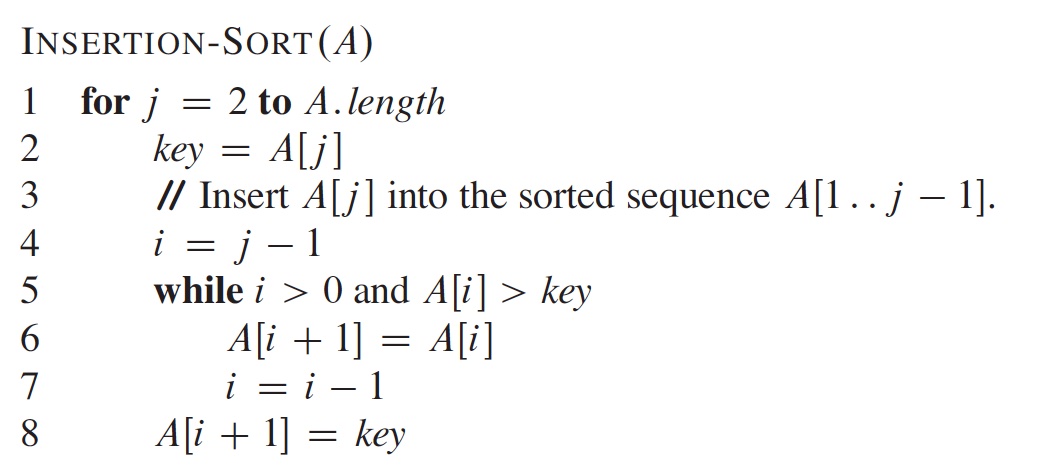
**Objective**:

The objective of this programming assignment is to implement in your preferred language (**already available on Tux machines**) the *InsertSort* and *QuickSort* 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. You must propose **pertinent (relevant)** plots to **validate** the theoretical time complexity of each algorithm and compare them. The theoretical time complexity is the time complexity we determine through analysis. Finally, discuss your results.

Below is the pseudocode of *InsertSort* is:



* USE THIS FILE AS THE STARTING DOCUMENT YOU WILL TURN IN. **DO NOT DELETE ANYTHING FROM THIS FILE:** JUST **INSERT** EACH ANSWER **RIGHT AFTER ITS QUESTION/PROMPT**.
* IF USING HAND WRITING (STRONGLY DISCOURAGED), **USE THIS FILE** BY CREATING SUFFICIENT SPACE AND WRITE IN YOUR ANSWERS.
* FAILING TO FOLLOW TURN IN DIRECTIONS /GUIDELINES WILL COST **A 30% PENALTY.**

What you need to do (*Insert in this file your answers*):

1. (30 points) Implement the *InsertSort* and *QuickSort*algorithms to sort an array.

*submit your source code in your preferred language (as long as it is supported on Tux machines).*

*Provide here the instructions to compile and execute your code.*

*First I have both my sorting algorithms and all the additional functionality code added to a file I named programming\_assignment\_3.py on my TUX machine. You can see this part in the screenshot I included below. Then you should simply be able to run the python code by hitting “python programming\_assignment\_3.py” in the terminal which I also did in the screenshot. At this point the program should run and ask for the user to hit enter to randomly generate a 10 element array. I do this to help show my algorithm sorts work properly. It will will then request you to press enter an additional time to start collecting data on the larger array and will do this in steps of 5000 starting at 10000 up to “L” which I set to 200,000 in my collectData() function. Because of how long it takes to finish each iteration of “L” I added additional functionality to show it stepping through each iteration. I included screenshots showing my program at each step and after it exits. At this time there should be another file that gets created in the current directory named “performance\_data.csv”. I have include again this file I got from my results after the program ran from the TUX machine.*

*Actions to complete:*

*1) ssh into a* ***Tux*** *machine*

*2) clear the screen on the Tux machine (use the command clear)*

*3) Display the current date (use the command date)*

*4) Compile your program (InsertSort and QuickSort)*

*5) Execute your program showing that your algorithm sorts correctly an array with 10 elements. We must see the original array and the sorted array. We must see this for both sorting algorithms*

*6) Take a screenshot of the Tux terminal and insert here (below). Your screenshot must be as readable as this template screenshot (we should see the username, the date, the commands typed, and the results):*



.... (10 points for the screenshot if your program works and produces correct data) Insert here YOUR screenshot with Tux terminal. Without screenshot with all required information 30 points will be taken off.

...(20 points if screenshot **PLUS** program works and produces correct data). No credit if the above tasks are not completed on a Tux machine.

*A screenshot of a computer program

Description automatically generated*

*I am also going to include in my screenshot here actions of what happens after hitting enter for starting to collect data on the larger array input values. These scale all the way to 200,000 for “L”. This took me about 90 minutes to complete after I started. I had the extra functionality that was not necessarily required for the assignment in hopes of making it easier for the grader to see how my code works and to verify the results easier.*

*A screenshot of a computer program

Description automatically generated*

*Upon completion of the program (after about 90 minutes) all the data should be completed and stored in the file I named above. Here is small screenshot of just the lines that happen upon completion.*

*A computer screen shot of a message

Description automatically generated*

1. ( 5 points) Collect the execution time T(n) as a function of n for the two algorithms

*No need to submit a table of the values you collect. Just state here if you collected data and submit a cvs file containing the values n, TInsertSort(n), and TQuickSort(n) where TInsertSort(n), and TQuickSort(n) are respectively the execution time of* InsertSort and QuickSort for an array of size n.

*The python file I provided when I turned in this assignment in fact does generate a csv file that a got to name “performance\_data.csv” after each part of the program finishes and exits. I have added multiple console messages to help with representing the part it is currently in.*

1. (15 points) Plot on the **same** graph the running time *TInsertSort (n), and TQuickSort(n)* of each algorithm.

*Insert here the plot...*

*Discuss the plots and conclude as which algorithm is more efficient.*

*A graph with a green line

Description automatically generated*

*I will say it very obvious after the waiting time and the ultimate difference in milliseconds between the times it took the Array to sort using the Insert sort algorithm that was described above and I implemented versus the quicksort time. Since the time complexity of Insertion-Sort is N2 as the value of n increases so does the time required to sort. This exponential growth of Insertion-Sort next to the logarithmic growth of Quicksort shows how much more efficient quicksort is as sorting algorithm. Again, the running time of QuickSort is considerably less than that of Insertion Sort across all tested array sizes. This efficiency becomes particularly noticeable as the size of the array increases, making QuickSort a preferable choice for sorting larger datasets.*

1. (25 points) Using a **pertinent/relevant graph/plot** with your data (**hint**: look at previous programming assignments how we can determine the shape of T(n) and its asymptotic growth), **validate** the theoretical time complexity of *InsertSort*.

*Insert here the plot...*

*Discuss here the pertinent plot that validates/confirms the theoretical asymptotic growth of InsertSort* *A graph with a line going up

Description automatically generated*

With this I used python to create my plot showing the asymptotic growth pattern next to a quadratic function. For whatever unexplained reason there is part of array size within my csv data that shows faster growth pattern, within the ranges of 75000 and 110000. Other than this portion of the graph it maintains a very similar growth pattern as a quadradic function as expected from an in-place sorting algorithm. The worst case will include n2 time complexity, which is not very efficient. The consistency of the experimental data with the quadratic model not only validates the theoretical complexity of Insertion Sort but also highlights the limitations of this algorithm in practical applications. This quadratic growth indicates that while Insertion Sort may be practical for small datasets, its performance degrades significantly with larger arrays, making it inefficient for large-scale sorting tasks due to the sharp increase in execution time.

1. (25 points) Using a **pertinent/relevant graph/plot** with your data (**hint**: look at previous programming assignments), validate the theoretical time complexity of *QuickSort*.

*Insert here the plot...*

*Discuss here the pertinent plot that validates/confirms the theoretical asymptotic growth of QuickSort*

The graph clearly shows that QuickSort's execution times are closely aligned with the n.logn curve, especially as the size of the input array increases. The plot emphasizes QuickSort’s efficiency, where the growth rate of execution time is significantly slower than quadratic, which is typically observed with algorithms like Insertion Sort. This confirms that QuickSort is better suited for handling larger data sets where performance is a critical factor. This consistency under various conditions underscores the reliability and stability of QuickSort in practical applications. The plot not only validates the theoretical efficiency of QuickSort but also highlights its practical applicability, providing a robust solution for real-world sorting challenges where large volumes of data must be processed efficiently.

A graph with a green line and blue line

Description automatically generated

**Program to implement**

collectData()

Generate an array G of **HUGE** length L (as huge as your language and available time allows) with **random** values capped at 0x7ffffffe.

for n = 10,000 to L (with step 5,000)

for each algorithm **Alg** from *InsertSort* to *QuickSort do*

copy in Array A **n** first values from Array G

Start timing // We time the sorting of Array A of length n

// Use nanoseconds resolution if possible.

Sort A using Algorithm **Alg**.

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/relevant** graph/plot for questions 4 and 5.

**How large should L be?** L must be as large as possible such that 1) you collect enough data to produce meaningful plots, 2) it would not take too much time to collect data, and 3) your machine can handle. **L should not be less than 200,000**.

**Data Analysis**

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

**Report**

* Your report is this file in which you inserted your answers
* Good writing is expected.
* Recall that answers must be well written, documented, justified, and presented to get full credit.

**What you need to turn in:**

* Electronic copy of your source program (standalone) that collects data
* csv file containing n, *TInsertSort(n), and TQuickSort(n)*
* Electronic copy of the report (this file including your answers) (standalone). Submit the file as a Microsoft Word or PDF file.

**Grading**

1. (**30 points**) Implement the *InsertSort* and *QuickSort* algorithms to sort an array A.
2. ( **5 points**) Collect the execution time T(n) as a function of n for the two algorithms. Submit the csv files containing n, *TInsertSort(n), and TQuickSort(n)*
3. (**15 points**) Plot on the same graph the running time of the two algorithms. Compare the two algorithms.
4. (**25 points**) Using a **pertinent/relevant graph/plot** with your data (**hint**: look at previous programming assignments), validate the theoretical time complexity of *InsertSort*.
5. (**25 points**) Using a **pertinent/relevant graph/plot** with your data (**hint**: look at previous programming assignments), validate the theoretical time complexity of *QuickSort*.