## Objectives of this assignment:

- to explore time complexity and "real time" of a well-known algorithm

## What you need to do:

1. Implement the ***Merge-Sort*** algorithm to sort an array. (See Appendix for the *Merge-Sort* algorithm)
2. Collect the execution time T(n) as a function of n
3. Plot the functions $T(n)/\log_2(n)$, $T(n)/n.\log_2(n)$, and $T(n)/n$ on the same graph. *If you cannot see clearly the shape of the plots, feel free to separate plots.*
4. In Module 4 (next module), we will establish that the running time T(n) of *Merge-Sort* is $\Theta(n.\log(n))$. Discuss T(n) in light of the graph you plotted above.

**Objective**: The objective of this programming assignment is to design and implement in Java the Merge-Sort algorithm presented in the lecture to sort a list of numbers. We are interested in exploring the relationship between the time complexity and the "real time". For this exploration, you will collect the execution time T(n) as a function of n and plot the functions $T(n)/\log_2(n)$, $T(n)/n.\log_2(n)$, and $T(n)/n$ on the same graph (*If you cannot see clearly the shape of the plots, feel free to separate plots.*). Finally, discuss your results.

## Program to implement

Tux Directions for **MAC:**

Compiling & executing the program on a Tux machine:

1. After you have transferred the files into a directory, follow the instructions below.
2. Open a new XQuartz or Mac Terminal
3. Type `ssh username@gate.eng.auburn.edu`
4. Enter password
5. When it asks `Please enter the name of an Engineering host <anywhere>:` just press **ENTER**
6. Type `yes` when it asks you if you want to continue
7. Enter password again
8. Type `cd directoryName` – *this will be the directory that you transferred the .java file to*
9. Type `ls` – *You will now see the file* `ProgrammingAssignment2.java`
10. Type `javac ProgrammingAssignment2.java` – *this will compile the program*
11. Type `java ProgrammingAssignment2` and this will execute the program – *it will take some time to execute*
12. Type `ls` again once the program is done executing and you should see the files `ProgrammingAssignment2.java, ProgrammingAssignment2.class, and programming2.csv`

Once you see the 3 files in the directory you created, it means that the program successfully compiled and executed on the Tux machine.

```
collectData()
    Generate an array G of HUGE length L (as huge as your language allows)
    with random values capped at some max value (as supported by your chosen
    language).
    for n = 1,000 to L (with step 1,000)
        copy in Array A n first values from Array G // (declare Array A
only ONCE out of the loop)

        Take current time Start // We time the sorting of Array A of length n
        Merge-Sort(A,0,n-1)
        Take current time End // T(n) = End - Start
        Store the value n and the values T(n)/log2(n), T(n)/n.log2(n), and
        T(n)/n in a file F where T(n) is the execution time
```

The implementation for program ProgrammingAssignment2.java worked perfectly and produced the required data for graphing in file programming2.csv.

**Advice:**

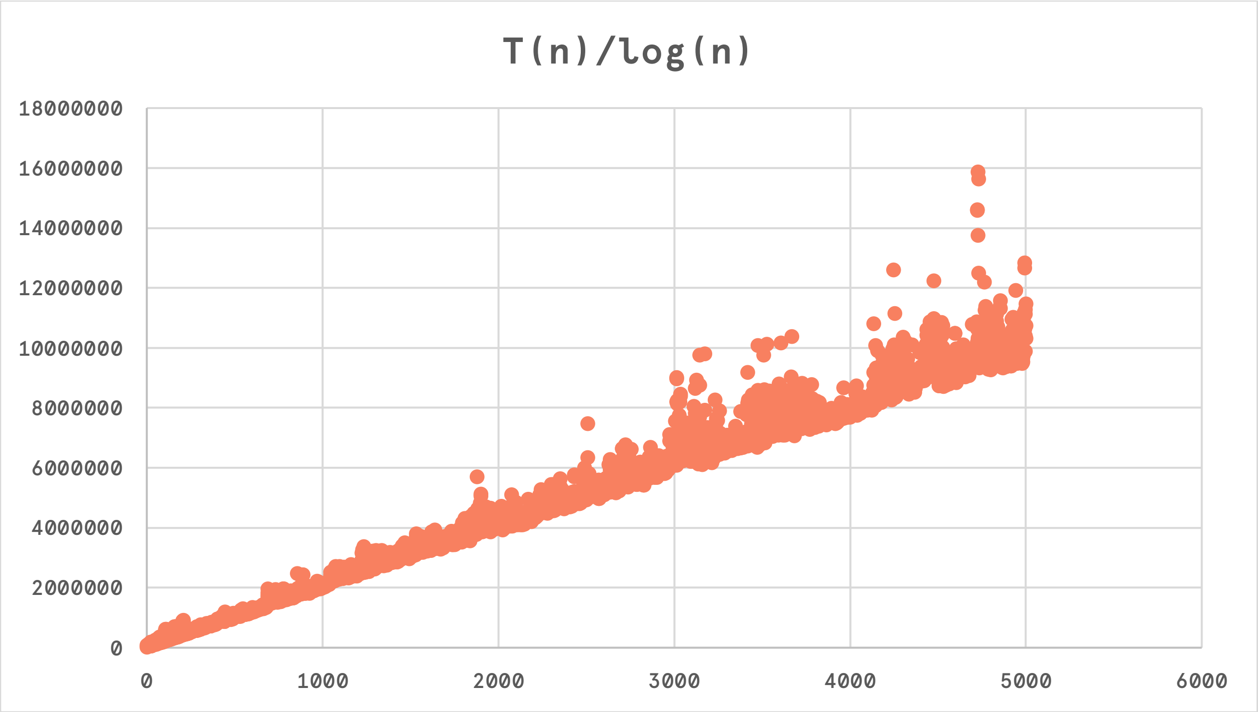
**1)** The pseudocode assumes arrays that start with index 1. So, an array A with n elements is an array A[1], A[2]..., A[n-1], A[n]. With most programming languages, an array A with n elements is an array A[0], A[2]..., A[n-1], A[n-1]. When implementing pseudocode that uses some array A with $n$ elements, I advise you to declare an array with $n + 1$ elements and just ignore (not use) A[0]. This way, you can directly implement the algorithm without worrying about indices changes.

**2)** When plotting, **ignore the first values of n= 1000, 2000, 3000, and 4000**. When a program starts, there will be some overhead execution time not related to the algorithms. That overhead may skew T(n).

**Data Analysis**

Use any plotting software (e.g., Excel) to plot the values T(n)/log$_2$(n), T(n)/n.log$_2$(n), and T(n)/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. (**Hint**: is T(n) closer to K. log$_2$(n), K. n.log$_2$(n), or K. n where K is a constant?)

The graphs are plotted separately to get a more accurate and clear view.




## Figure 1

Above is **Figure 1**, the graph for $f_1(n) = T(n)/log_2(n)$. The graph for $f_1(n)$ has a linear runtime, grows in an upward trend, and is the highest and fastest growing function out of the 3. This graph exemplifies that $T(n)$ grows faster than $n$, meaning that the time complexity does **not** grow as $\log(n)$, and $T(n)$ is not the closest to $K.log_2(n)$, where $K$ is a constant.

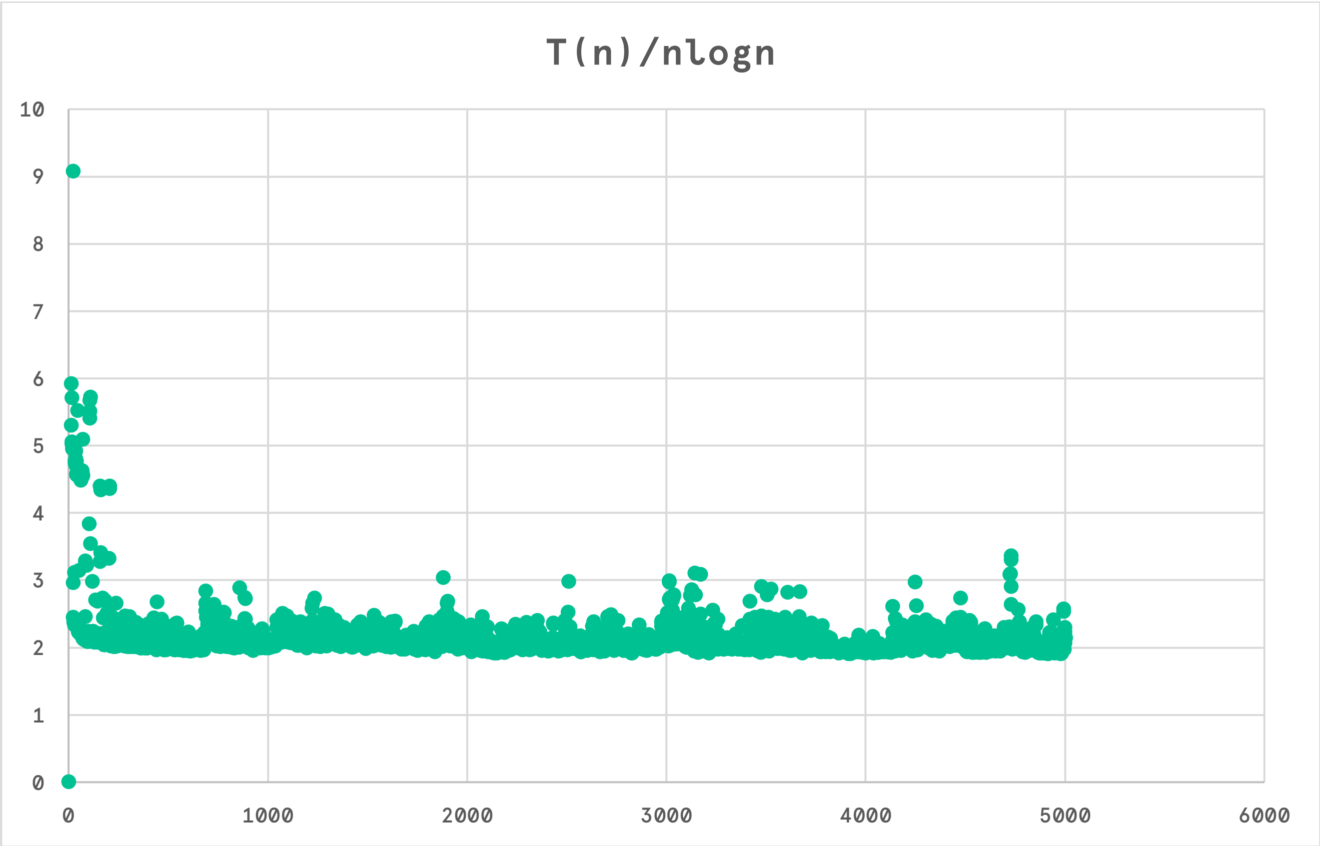
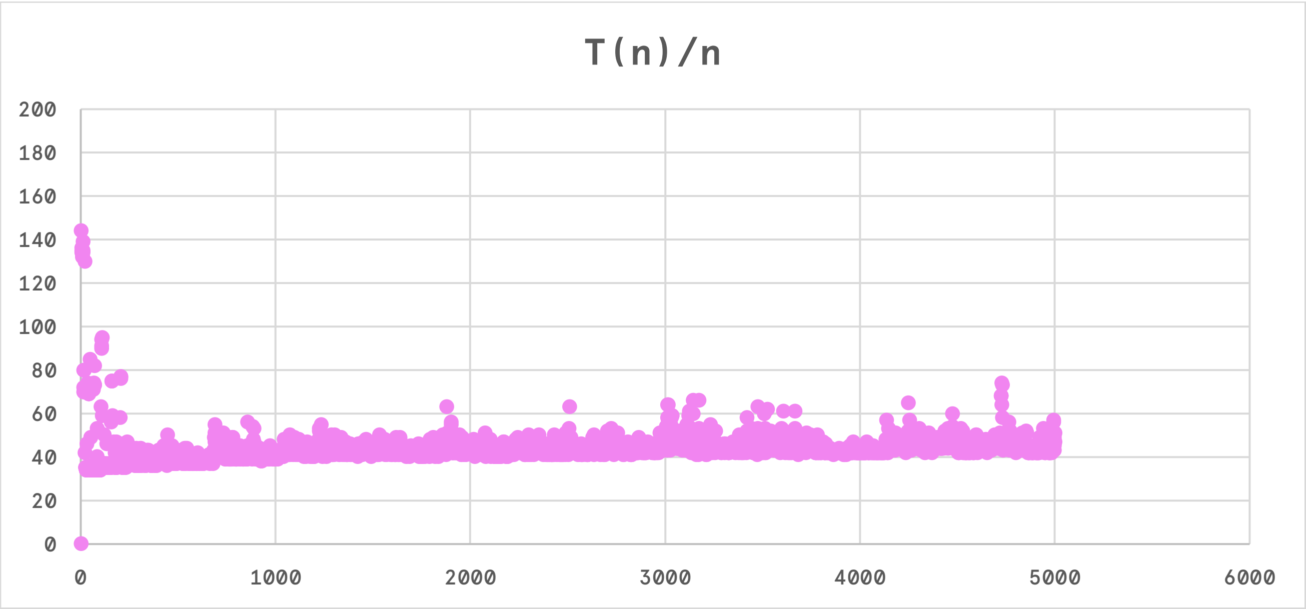



**Figure 2**

Above is **Figure 2**, the graph for $f_2(n) = T(n)/nlog_2(n)$. The graph for $f_2(n)$ is growing at a constant rate around **2**. This graph exemplifies that $T(n)$ grows at a constant rate around $2$ when compared to $nlog(n)$, meaning that the time complexity is $nlog\ (n)$, because $T(n)$ scales at the same rate as $nlog(n)$ and is the closest to $K.nlog_2(n)$, where $K$ is a constant.




## Figure 3

Above is **Figure 3**, the graph for $f_3(n) = T(n)/n$. The graph for $f_3(n)$ is growing at a constant rate around $\mathbf{40}$. Although this graph is growing at a constant rate, it is not the closest to $T(n)$. Since it is growing at a constant rate around $\mathbf{40}$ and not a rate closer to $\mathbf{1}$, it means that $T(n)$ grows faster than $n$. This exemplifies that the time complexity is **not** $n$, because $T(n)$ does not grow as $n$ and is not the closest to $K.n$, where $K$ is a constant, solidifying the conclusion that the time complexity of $T(n)$ is $nlog(n)$**.**

**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.

**What you need to turn in:**

- Electronic copy of your source program
- Electronic copy of the report (including your answers) (standalone). Submit the file as a Microsoft Word or PDF file.

**Grading**

- Program is worth 30% if it works and provides data to analyze
- Quality of the report is worth 70% distributed as follows: good plot (25%), explanations of plot (10%), discussion and conclusion (35%).

  Appendix: Merge-Sort Algorithm.

  At this stage, you do NOT need to understand Merge-Sort (It will be presented and explained in Module 4)). Implement Merge-Sort exactly the way it is described below. Replace the infinity value (∞) with 0xffffffff.

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

MERGE$(A, p, q, r)$

$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
**for** $i = 1$ **to** $n_1$
    $L[i] = A[p + i - 1]$
**for** $j = 1$ **to** $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
**for** $k = p$ **to** $r$
    **if** $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    **else** $A[k] = R[j]$
        $j = j + 1$