UserActivityLogManager

Experiment Report

Team Members:

Brendan Viscount

CSC-316-002

Brendan Viscount

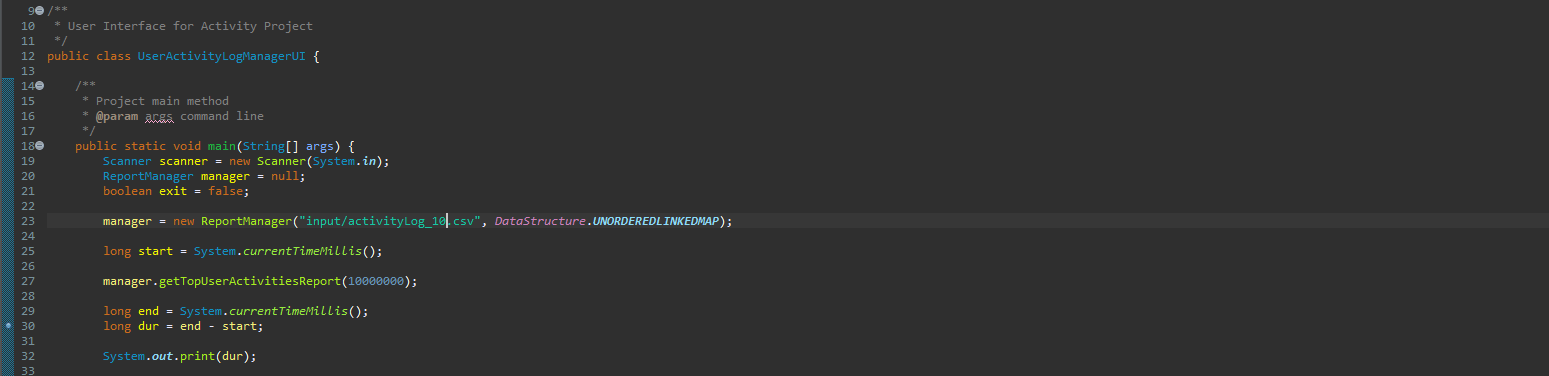
10/30/2024

Implementation provided by:

*Brendan Viscount*

# **Methodology**

**Measuring Runtimes**



In order to test the actual runtimes of each data structure’s performance with various input sizes, I instantiated a new manager object that specified the input file to read and the data structure to be used. I utilized the System.currentTimeMillis() command to attain the time before and after running the algorithm. I then took the difference between the two times to find the duration of the algorithm and printed it to the terminal for accessibility.

## **Results**

**Hardware**

We conducted the experiment using the following hardware:

● Operating system version: \_\_\_\_\_\_\_\_\_\_\_\_\_Windows 10 Home\_\_\_\_\_\_\_\_\_\_\_\_\_\_

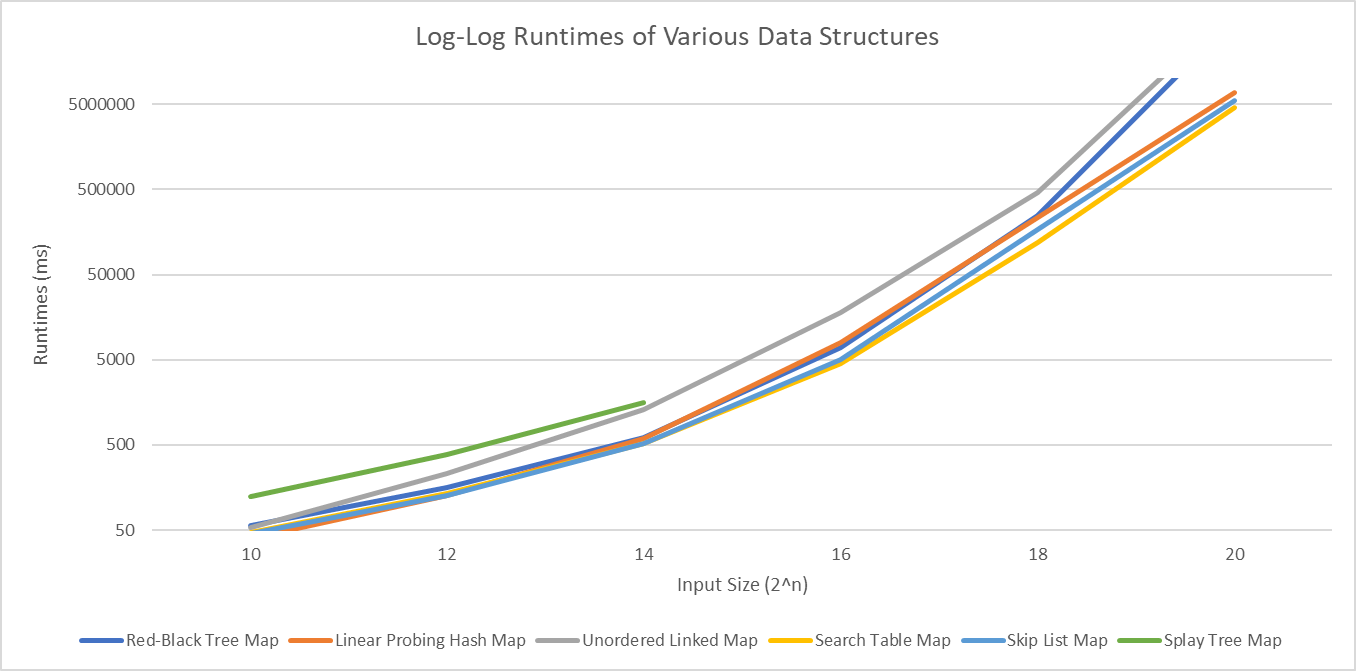
● Amount of RAM: \_\_\_\_\_\_\_\_\_\_\_\_\_16GB\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

● Processor Type & Speed: \_\_\_\_\_\_\_\_Intel(R) Core i5-9400F 2.9 GHz\_\_\_\_\_\_\_\_\_\_

## **Table of Actual Runtimes**

| **Input Size (2x)** | **Unordered Linked Map (ms)** | **Search Table Map (ms)** | **Skip List Map (ms)** | **Splay Tree Map (ms)** | **Red-Black Tree Map (ms)** | **Linear Probing Hash Map (ms)** |
| --- | --- | --- | --- | --- | --- | --- |
| 10 | 54 | 47 | 46 | 124 | 57 | 40 |
| 12 | 228 | 135 | 128 | 382 | 159 | 127 |
| 14 | 1303 | 526 | 514 | 1571 | 609 | 592 |
| 16 | 17848 | 4499 | 4979 | ERROR | 6920 | 7963 |
| 18 | 456555 | 119755 | 167787 | ERROR | 248958 | 231597 |
| 20 | 64673239 | 4583694 | 5513326 | ERROR | 50579768 | 6917529 |

## **Chart 1: Log-Log Chart of Actual Runtimes**



## 

## 

## 

## 

## 

## 

## 

## 

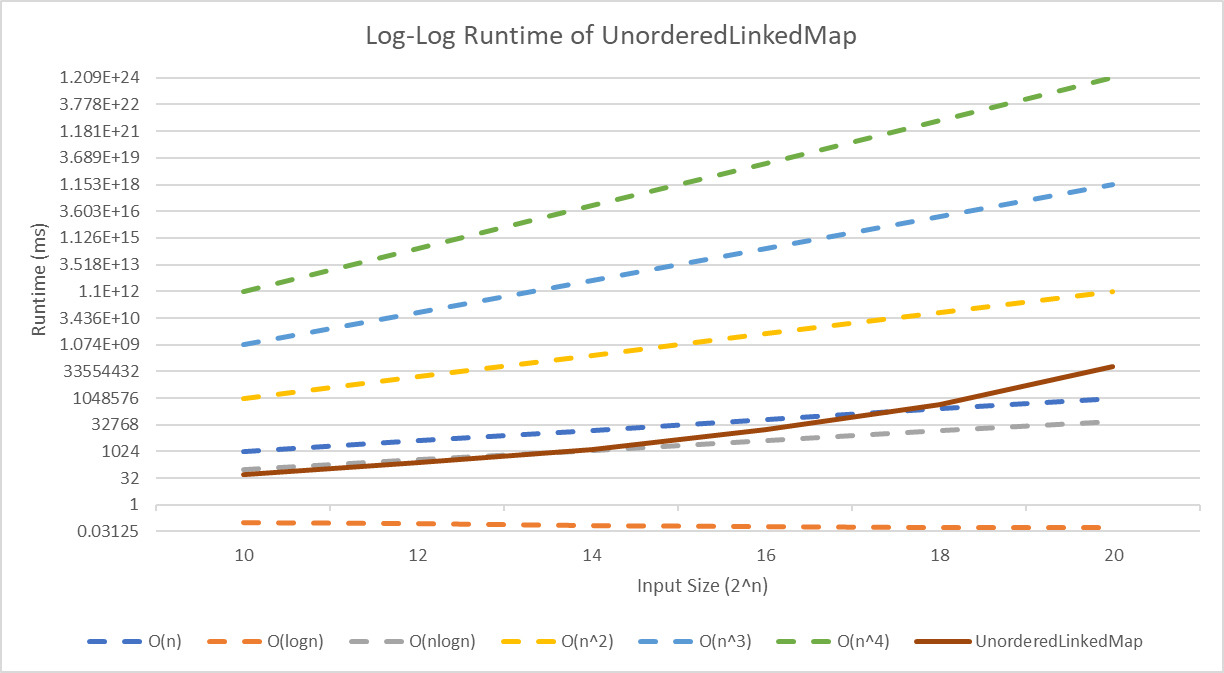
## 

## 

## 

## 

## **Chart 2: Log-Log Chart of Actual Runtimes Unordered Linked List-based Map**



## 

## 

## 

## 

## 

## 

## 

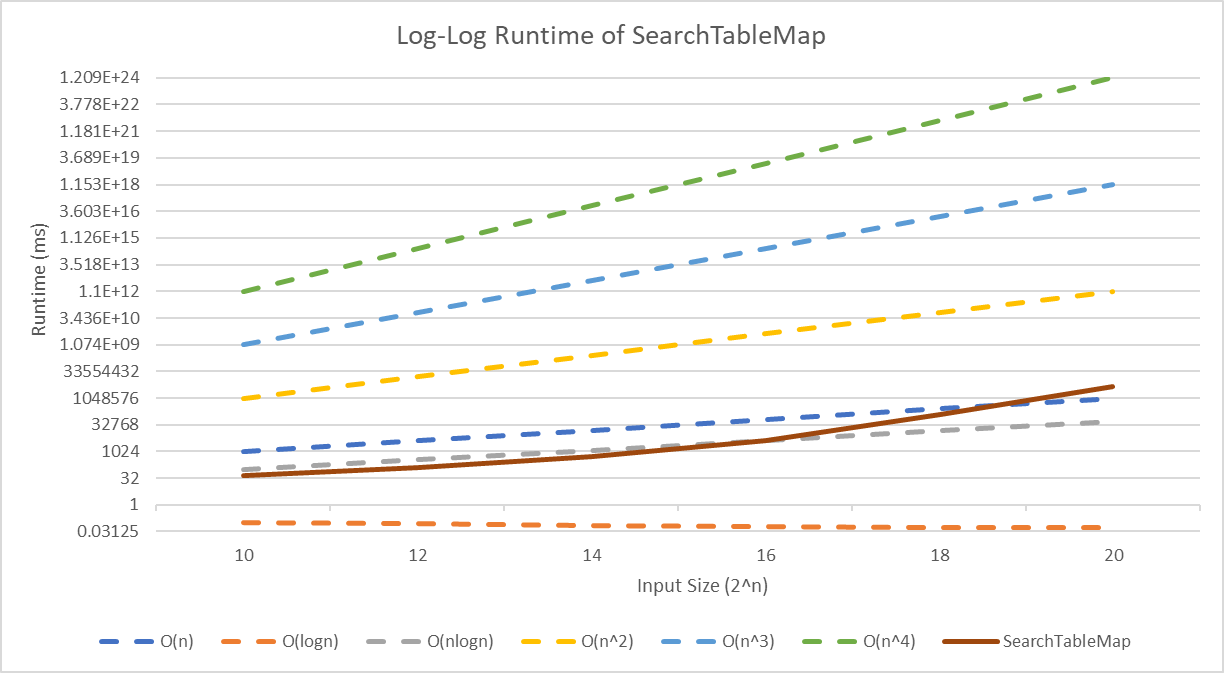
## 

## 

## 

## 

## **Chart 3: Log-Log Chart of Actual Runtimes for Search Table Map**



## 

## 

## 

## 

## 

## 

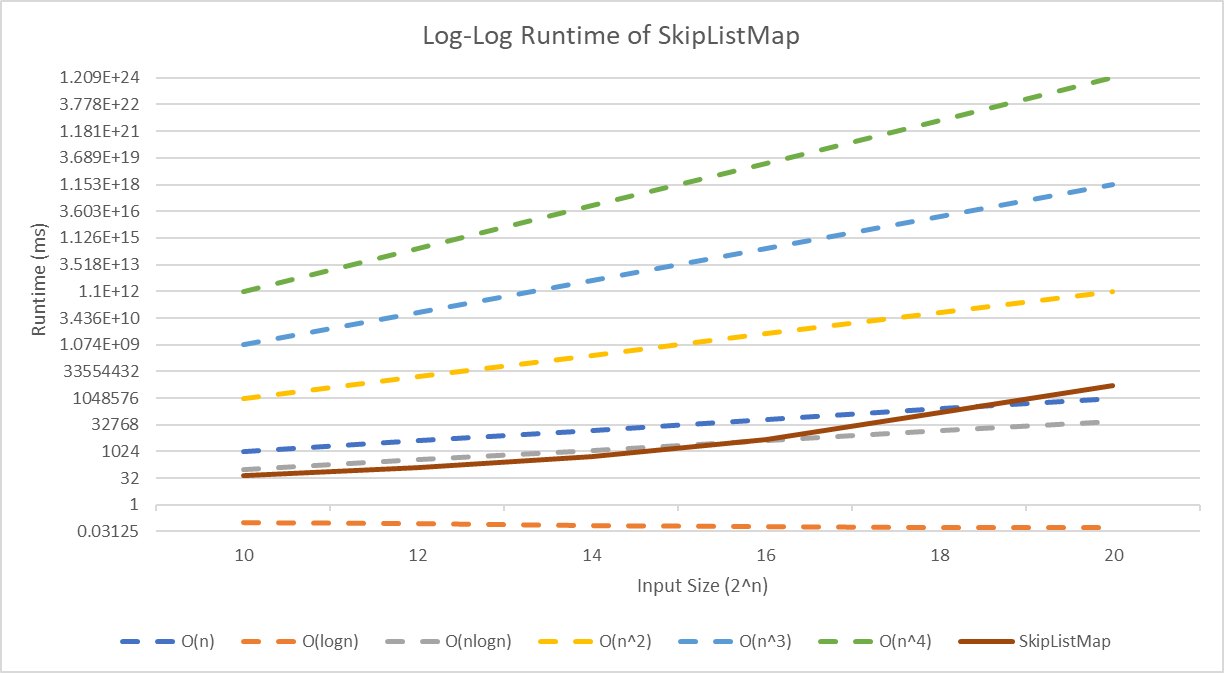
## 

## 

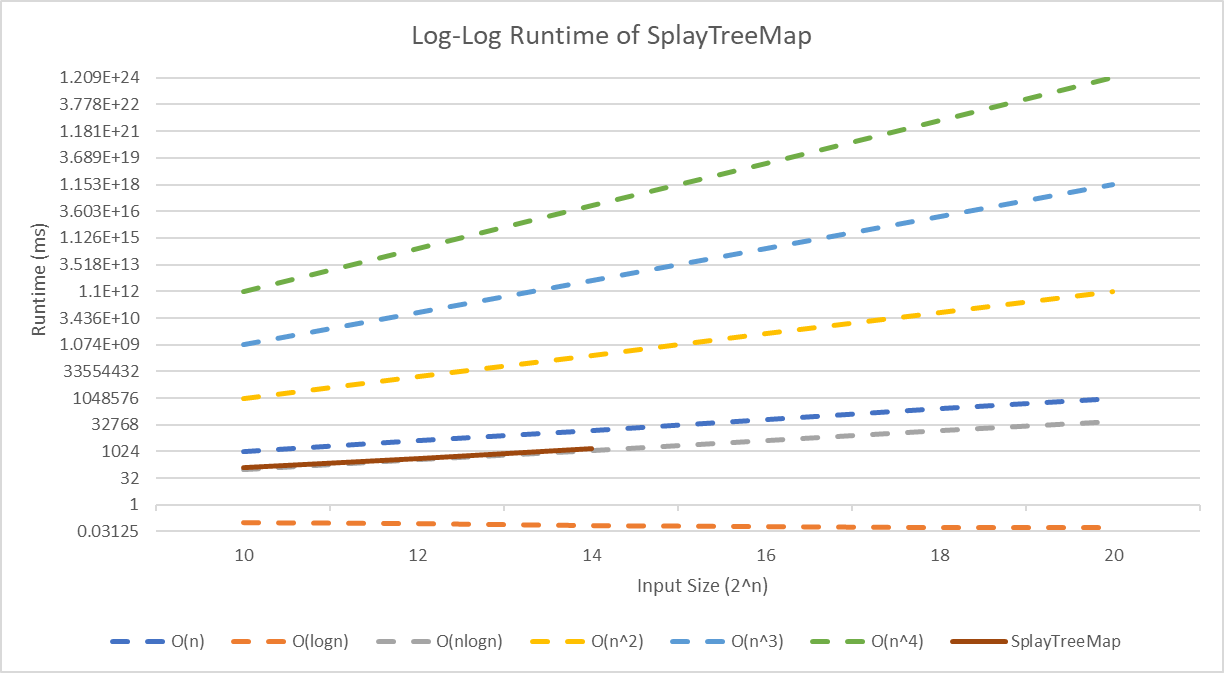
## 

## 

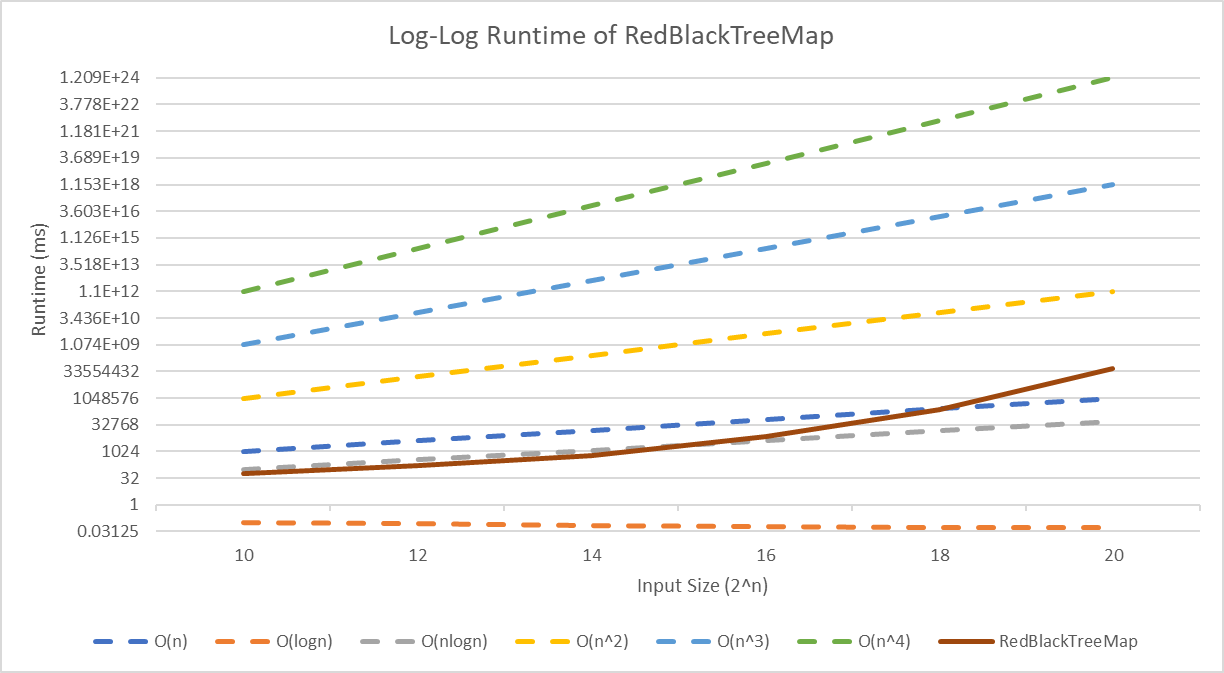
## **Chart 4: Log-Log Chart of Actual Runtimes for Skip List Map**



**Chart 5: Log-Log Chart of Actual Runtimes for Splay Tree Map**



## **Chart 6: Log-Log Chart of Actual Runtimes for Red-Black Tree Map**



## 

## 

## 

## 

## 

## 

## 

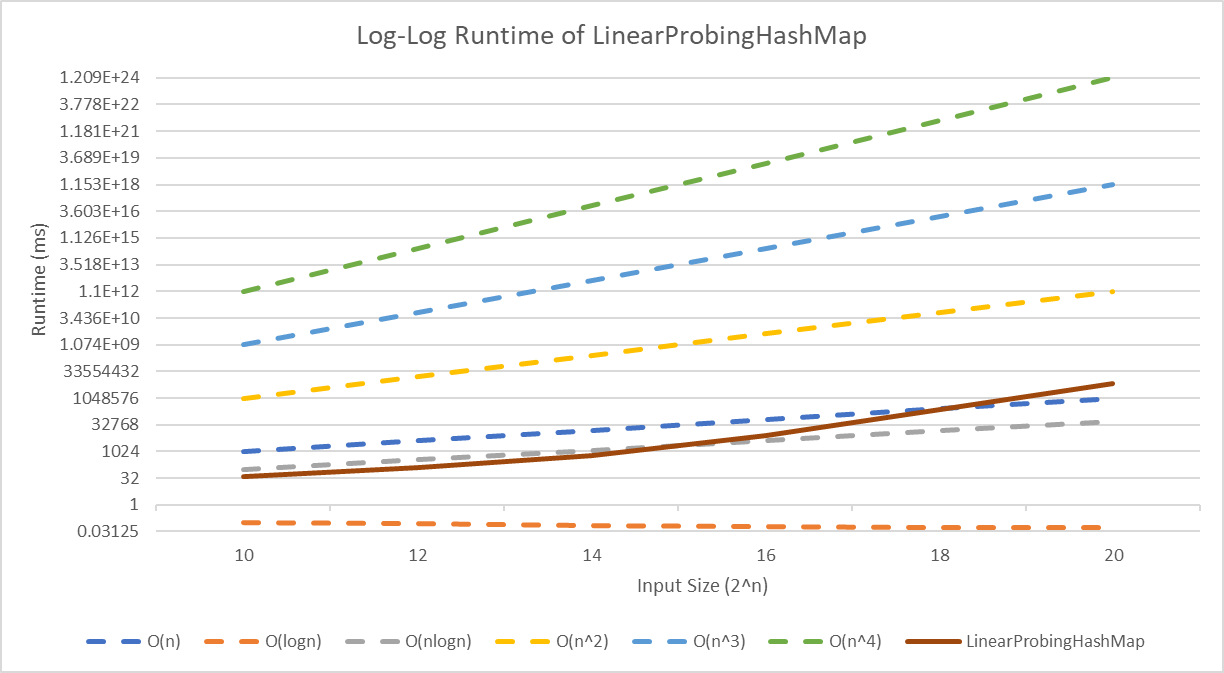
## 

## 

## 

## 

## **Chart 7: Log-Log Chart of Actual Runtimes for Linear Probing Hash Map**



# **Discussion**

## **Reflection on Theoretical Analysis**

In the project proposal, my algorithm runtime analysis estimated the runtime would be O(n). This was because the most complex operation within the algorithm was O(1), and that was executed in a loop that iterated *n* times. Experimental results showed that while the algorithm initially adhered to the O(n) runtime estimate, performance degraded as input size increased. When the algorithm was dealing with the smaller relative input samples it performed below the O(n) line, but as the sample size grew to half-a-million/a million inputs the graph line for the algorithm’s runtime can be seen crossing above the O(n) projection line. I believe that the discrepancies between the estimated and actual runtime of the algorithm has to do with the implementation strategy of the algorithm from the project proposal having to be altered to accommodate the requirements of the project. These accommodations included sorting the log entries within their top activity lists beyond how they were separated in the proposal. Also there were several edge cases that had not been addressed in the proposal’s implementation, such as actions with different resources and sorting alphabetically for equally frequented activities which could also increase the runtime and time complexity of the algorithm.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## **Reflection on Data Structure Selection**

In my initial proposal, I selected an array-based list and queue to manage and sort log entry data, largely because they allow for indexed access and straightforward iteration. However, this choice was initially based on an incomplete understanding of the requirements. I expected a fixed set of actions and designed my approach around this assumption. These data structures were not the best choice for implementing this algorithm because in order to create the sorted action map, I needed to have a way of creating a new action list when a unique action and resource were encountered. The strategy I created in the proposal used queues for this situation because I initially thought there would be a set number of actions encountered, which was not the case. Beyond this, using a queue to store the data gave an efficient approach to adding logs to their respective action lists, but it became inefficient when trying to sort the logs within each list by their timestamp. While these structures allowed for basic data storage and retrieval, a more efficient map-based approach would have been better suited to creating dynamically ordered lists by action and resource. If it had been an option, using a hash map that used double-hashing based on the action and timestamp would have allowed more efficient retrieval of specific logs and significantly improved performance.

## **Improving Efficiency**

In my implementation, there are several places where efficiency could be improved. The strategy I implemented relied on three helper methods, getActionList, getLargest, and getReturnArray. Each method had some sort of iteration through a specific list structure. While this is good for code readability and debugging, it is inefficient. If I streamlined these operations to minimize the amount of iterations through the data and used more clever methods to format the data properly without needing to create instances of it for each helper method to be able to work with, that would surely be a significant improvement of the efficiency of the code.

The algorithm could employ early stopping when encountering pre-sorted data which would reduce unnecessary operations. This would also lead to a significant improvement to the efficiency of the code, but only in the cases where this happens to occur. I don’t believe this would be a useful addition to the code because the chances of this situation occurring are slim, and if it can be assumed that the input was already sorted in some manner, other accommodations could be made to improve efficiency in a more drastic manner. If the input file was already sorted by timestamp or activity, several operations could be optimized. For example, the getActionList method would not be necessary if the entries have already been sorted by action. Also, the algorithm would not have to sort the entries by timestamp if the proper ordering is provided. Both of these alterations could work to improve the efficiency of the code’s execution. A pre-sorted data set could be obtained if the log entry data was stored in an Excel sheet that provides a sorting mechanism for each.

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## **Lessons Learned**

A key lesson I learned is the importance of minimizing unnecessary loops. In my getTopActivities method, I used four separate loops to process the dataset, which increased inefficiency. This could have been avoided by structuring the data flow more effectively. By combining steps, such as joining activity frequencies and sorting in a single pass, I could have reduced the number of iterations and improved performance. This experience highlighted the need to plan data processing more efficiently, minimizing redundant operations to optimize both time and space complexity. Another lesson I took away from this experiment is the importance of selecting and correctly utilizing efficient data structures. The experiment provided examples of how choosing different data structures can impact the overall performance of an algorithm. The difference in efficiency was most apparent in the data that was collected when working with over a million entries. For example, the runtime when choosing a search table map data structure to implement the algorithm was quicker than the unordered linked list’s runtime by over 15 hours. This proved that choosing efficient data structures to implement an algorithm can work to save a significant amount of time during execution. These lessons will guide my approach in future projects and encourage me to prioritize efficient data structure selection and remain conscious when working with the flow of data processing.