Project 2 – Written Analysis of Project 1

Jade Pearl

CMSC 451-7380

Professor Dennis Didulo

Table of Contents

[Introduction 3](#_Toc152767132)

[Pseudocode 3](#_Toc152767133)

[Big -Θ 3](#_Toc152767134)

[JVM Warm Up 4](#_Toc152767135)

[Critical Operation 4](#_Toc152767136)

[Analysis 5](#_Toc152767137)

[Graph 5](#_Toc152767138)

[Performance Comparison 6](#_Toc152767139)

[Critical Operation Results and Time Measurements 6](#_Toc152767140)

[Coefficient of Variation 7](#_Toc152767141)

[Results vs Big-Θ 8](#_Toc152767142)

[Conclusion 8](#_Toc152767143)

# Introduction

The two algorithms chosen for Project 1 in comparison were Heap sort, which heapifies an array until it is sorted, and Quick sort, which partitions the array into smaller subarrays, sorting them recursively using pivot elements.

## Pseudocode

The high-level pseudocode for heapsort that was used in my program is as follows:

Heapsort(array):  
 heapify(array)  
 for i from n-1 to 0 do  
 swap (array[0], array[i])  
 heapify(array, i, 0)

For the Quick sort:

Quicksort(array, low, , high):

if low < high then

partitionIndex = partition(array, low, high)

QuickSort(array, low, partitionIndex - 1)

QuickSort(array, partitionIndex + 1, high)

## Big -Θ

Heap sort, in the worst-case scenario, is a Θ(n log n) because the given array is heapified for every element in the array regardless of the input distribution. Hence, the worst-case is consistent. In the best-case, the array is already sorted. Regardless of that, the heap sort still requires the building of the heap and heapifying for each element in the array, so the time complexity in the best-case is still Θ(n log n). Finally, the average-case holds true to the Θ(n log n) time complexity because of the heaping nature of the sorting algorithm. The time complexity has proved consistent all around, no matter the case.

The worst-case in Quick sort occurs when the resulting subarrays have barely reduced in size due to unbalanced partitioning. This results in a time complexity of Θ(n2). In the best-case, the pivot element consistently partitions the array into nearly equal halves which leads into fewer recursive calls, so the time complexity would be Θ(n log n). Lastly, the average-case of the sort would occur when the partitioning is balanced and the data is more randomized rather than already sorted. The time complexity remains Θ(n log n).

## JVM Warm Up

JVM warm-up can create obstacles in the accurate measurement of algorithm performance, especially with sorting algorithms. In the BenchmarkSorts.java where the main method resides, the runWarmUp() method was called to execute a series of warm-up runs for both of the sorting algorithms separately. This allows for JVM optimization before actual benchmarking of the sorts because the runWarmUp() method sorts randomly generated data with 100 elements. The purpose of sorting a smaller data set of 100 elements before actual benchmarking given input primes the JVM to optimize sorting as the program runs.

## Critical Operation

The critical operation used for the Heap sort algorithm was the swapping of elements in the heapify() method. It occurs when adjusting the heap structure and maintaining the minimum or maximum heap. I had chosen the swap operation because of its maintenance of the heap structure and the number of swaps in the array impacts the efficiency of the algorithms performance.

Similarly, the critical operation chosen in the Quick sort was element swapping in the partition() method. It occurs when reordering elements based on the pivot element and the array divides into smaller sections. I chose the swap in Quick sort as well because rearranging elements around the pivot element is crucial to the divide-and-conquer nature that the Quick sort utilizes.

The resulting critical count of these operations provides a quantifiable measurement that reflects the efficiency of the two sorting algorithms.

# Analysis

## Graph

Due to difficulty with the number of variables in Excel, I had to make two separate line graphs for comparing Heap sort and Quick sort. One graph is for comparing the average critical count (which analyzes the critical operation of swaps in each sort):

The other graph compares their average times:

## Performance Comparison

According to my findings and the previously shown graphs, the heap sort and quick sort both have a relatively consistent increase in their use of critical operations (swaps) as the size of the array increases. And there was a relatively stagnant execution time that slightly increased with array size.

## Critical Operation Results and Time Measurements

The critical operations for Heap and Quick sort are both the same, hence the overlapping lines in the first graph. They both increase in the number of critical operations as the size of the array increases. In time measurement, Quick sort appears to run at a somewhat faster rate than the Heap sort. In this case, it can be concluded that the Heap sort is better suited for smaller array sizes whereas Quick sort may be quicker with the larger data sets (over 100,000 elements) since it runs at a higher speed than Heap sort.

## Coefficient of Variation

The coefficient of variation was applied to the project and is significant because it provides insight into consistency and stability of an algorithm’s performance across data set sizes. A lower value coefficient of variation typically points to more consistent performance results. Below are screenshots of tables from the output of Project 1 which show the coefficients for count and time for each algorithm represented in percentages:

Heap Sort  
A screenshot of a computer

Description automatically generated

Quick Sort  
A screenshot of a computer

Description automatically generated

The coefficient of variance for counts (critical operations) is the same for both sorting algorithms (0.0%) which indicates that neither sort is more stable or efficient than the other when it comes to the number of swaps. The coefficient of variance for time in the Quick sort is higher than that of the Heap sort but is still low in percentage; Heap sort is still constantly at 0.0%. This could be due to rounding since the execution time does in fact change, but possibly too small for the program to calculate. These results show that Heap sort is more stable in its efficiency than Quick sort as data sizes increase which may be plausible since in the table, the execution time goes up and down a few times whereas it barely changes in Heap sort.

## Results vs Big-Θ

As stated previously, Heap sort always has a Big-Θ time complexity regardless of the size of the data set. This is supported by the results in both the line graph and the coefficient of variance observed in the above table. The critical operation and execution time are consistently increasing as the data set grows. Therefore, the results of the project support and prove the Big-Θ analysis of the Heap sort algorithm.

It was stated that Quick sort has a worst-case Big-Θ time complexity of Θ(n2) and an average and best-case of Θ(n log n). The results from the project also supports the Big-Θ analysis of Quick sort because even though the critical operation was consistent, the execution time had slightly inconsistent changes as the data set size increased.

# Conclusion

This study carried out from the program results of Project 1 analyzed the efficiency and compared the two sorting algorithms, Heap sort and Quick sort. Heap sort demonstrated a consistent behavior in critical operations and in execution time and the results aligned with its Big-Θ analysis of Θ(n log n). With Quick sort having a consistent best and average-case of Θ(n log n) but slightly inconsistent worst-case of Θ(n2) which was also supported by the results, it still has a degree of stability with increasing data set sizes due to its low coefficient of variance. The performance and efficiency comparisons paired with the coefficient of variation showed that both sorts are efficient and fast, with the Quick sort being faster at higher data set sizes. Heap sort may be better with small-medium data set sizes with its high execution time but may be more reliable over all with its 0.0% coefficient of variance. Quick sort may be better with larger sized arrays but may be a bit unstable in its execution time. Ultimately, choosing Heap sort versus Quick sort falls down to specific needs of a program and the data set sizes and the trade off between efficiency and stability.