**CMSC 451 Project 2 - QuickSort**

QuickSort is a divide and conquer algorithm that makes use of a pivot element for sorting elements of a given input. The critical operation used in QuickSort is a method called partition, which moves all values less than the pivot value before the pivot (all element larger than the pivot stay as they are already correctly after the pivot element.) There are two implementations in this project: iterative and recursive. The iterative implementation uses a stack to keep track of partitioned elements and it loops until the stack is empty. The recursive version requires an extra method in this implementation in order to keep track of number of operations and the amount of time taken to run. The way the recursive version works is by calling itself on the half that is less than or equal to the pivot element and then again on the half that is greater than the pivot element. The pivot element is chosen by the pivot method in both implementations.

1. The Sorting Algorithm

Pseudocode

Iterative

iterativeSort(start, end, data)

create a stack

add start to stack

add end to stack

while (stack is not empty)

end = stack.pop()

start = stack.pop()

pivot = partition(start, end, data)

if pivot - 1 > start

add start to stack

add pivot – 1 to stack

if pivot + 1 < end

add pivot + 1 to stack

add end to stack

Recursive

recursiveSort(start, end, data)

if start > end

return

partition = partition(start, end, data)

recursiveSort(start, partition – 1, data)

recursiveSort(partition + 1, end, data)

Big-Θ Analysis

Iterative

[analysis]

Recursive

[analysis]

JVM Warm-Up Technique

In order to avoid anomalous results, especially for the earlier test runs, it was necessary to use some technique to warm up the JVM. My approach was to simply call the sorting algorithms in a loop 100 times and then finally record the results one last call after this warm up. I chose this technique because it was easy to implement and based on my research it seemed like it would work. While experimenting, I found that running the sorting algorithms just a few times yielded consistent results without anomalies in the data, and so I chose 100 as it does not require too long of a wait either.

Chosen Critical Operation

The critical operation chosen to benchmark was the while loop for the iterative implementation and the recursive function that calls itself for the recursive implementation. Even though partition is the critical operation in QuickSort, it is called from within each function. This way we are also able to get a more complete picture for comparing the performance of the recursive and iterative implementations.

1. The Results

Critical Operations Count

Execution Times

Comparisons

[a comparison of the performance of the two versions of the algorithm ]

[a comparison of the critical operation results and the actual execution time measurements ]

Coefficient of Variance

[a discussion of the significance of the coefficient of variance results and how it reflects the data sensitivity of your algorithm ]

Results vs Big-Θ analysis

[how your results compare to your Big-Θ analysis ]

1. Conclusion

[A conclusion that summarizes the important observations of your study ]