**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

The worst case is when the pivot chosen is the largest or smallest value in the input every iteration which leads to O(n2) runtime since each successive iteration will compare n – i – 1 elements to the pivot. The best case is when the pivot is the middle-most value each iteration as this will perfectly halve the elements into their respective groups. The best-case runtime is Ω(n log n). The worst-case running time is unlikely to occur since it relies on a particular element being chosen as the pivot each iteration. However, the worst-case is more likely if the input is sorted or almost sorted since the last element in the sub-array is chosen as the pivot element.

The average case is Θ(n log n) for both the iterative and recursive implementation. We know this because both implementations call partition n log times and partition is Θ(n). Partition is Θ(n) because it has a single for-loop that runs up to n times. Specifically, the for-loop in partition will run one time for each element in the sub-array given as input. Both the recursive and iterative implementation call the same partition method.

The iterativeSort method has a while-loop that will run until the stack is empty. The elements to the left or right of the pivot are then pushed to the stack. Then each iteration, two elements are removed from the stack. While doing this, partition is called as many times as the while-loop runs which leads to Θ(n log n) = Θ(log n) \* Θ(n).

The recursive method calls partition and calls itself twice (left side of partition and right side of partition.) Each recursive calls is, on average, less than half of n input size (less because pivot element is sorted each call.) So we have T(n) = T(m) + T(n – m – 1) + Θ(n) = Θ(n log n), where m is the elements on the left of the pivot, -1 for the pivot element, and Θ(n) for the partition method. The average case is Θ(n log n) much like the best case is n log n. The reason that the average case tends towards the best case rather than the worst case is that the pivot element tends toward the middle of the sub-array rather than each subsequent largest element left in the input.

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. Partition is the critical operation in QuickSort and it is called from within each function. By counting the loops and recursive calls, we are able to capture the amount of times each QuickSort algorithm runs in addition to how many times partition is called. The reason that the loop and recursive function itself were chosen is that they are, with respect to each of their implementations, the driving factor in run time of the algorithm and will best serve as a way to compare results to theoretical Big-Θ analysis.

1. The Results

Critical Operations Count

Execution Times

Comparisons

It is clear that the iterative implementation of QuickSort outperforms the recursive implementation, especially as the dataset size grows. The number of operations and the amount of time that the recursive implementation takes is nearly double that of the iterative version regardless of the input size.

The number of critical operations and the total execution time show similar trends for both implementations on each data set. As the input size doubles, the number of operations approximately doubles as well as the execution time, with one exception: data set number 5 where n = 160. Based on other results from running this program, this is an anomaly that was observed just one time, on the results we are currently analyzing. Since both iterative and recursive versions do have a worst case of n2, it does leave room for an anomaly such as this to occur. Additionally, it could have been a hardware issue while running. The anomaly was reflected in both the execution time and number of critical operations ran by both implementations.

Coefficient of Variance

The coefficient of variance is the relative standard deviation or the ratio of the standard deviation to the average. Since each benchmark size had ten different datasets, the average results were used. The coefficient of variance would indicate precision or the lack of precision in those calculations. If there was a lower coefficient of variance, one could reasonably expect similar results on another run of the program. Higher coefficient of variance would indicate that the results are more likely to change the next time the program is executed.

The coefficient of variance decreased as the input size increased, as expected.

Results vs Big-Θ analysis

The growth rates of both the iterative and recursive implementations fall in line with the Big-Θ analysis of Θ(n log n). Although the recursive implementation had execution times and counts that were twice that of the iterative implementation, the growth rates of each were Θ(n log n).

1. Conclusion

In conclusion, QuickSort is among the elite class of efficient sorting algorithms. My iterative implementation was more efficient than the recursive version, which was unexpected since most implementations I have ever studied were recursive. The partition subroutine is where the sorting really happens as the elements are moved around the pivot point. This makes the partition subroutine doubly important because it is also, where the pivot element is chosen. An improvement that could be made on my current implementation of this algorithm would be to choose the pivot element at random rather than choosing the last element in the sub-array. This would help with arrays that are sorted or mostly sorted.

The iterative implementation was overall more efficient than the recursive implementation as far as results are concerned, but they both have Θ(n log n) runtime. Also, the probability of the worst case O(n2) occurring decreases as n increases since partition will choose a new pivot and the pivot needs to be the worst case each time. The data suggests that this may have been a possibility at n=160, but as n increased the execution times and operation counts grew closer to a rate that was more expected based on the Big-Θ analysis.