**Project 2:**

**Merge Sort is a Good Sort**

Zachary Finnegan

CMSC 451

**Introduction**

Merge Sort is one of the three Θ (n Log n) sorting algorithms. The other two are Heap Sort and Quick Sort. In all cases Heap and Merge Sort are Θ (n Log n), however Merge Sort is slightly faster but with a higher memory overhead. Quick Sort is faster than both except in the worst case where it is Θ(n^2).

Merge Sort works as a divide and conquer algorithm. An array is taken as an input and then it is split in half repeatedly until only one value arrays remain. Then the algorithm takes the arrays and begins combining them so that they are ordered. From single arrays to double arrays to quadruple arrays, and on and on until all the chunks are recombined into one array. Due to the need for an additional array, Merge Sort has a high memory overhead.

During the combining process, the array chunks are checked to see if any rearranging of values is needed. If there is a need the values are swapped and then the two array chunks are combined. If all the values in the array are in order and all the values in the left array are lower than the values in the right array the two arrays are just merged into one array without swapping values.

This is a very efficient sorting algorithm and is useful for large data sets. If there is a strong possibility that the data is already sorted this algorithm is slower than other options such as Modified Bubble Sort or Insertion Sort. These two algorithms are about Θ(n) for an already sorted set.

**Recursive**

The pseudocode for the recursive Merge Sort is as follows:

1. If the array length is greater than 1
   1. Divide the array into two equal sized arrays, left and right.
   2. If the array length is not greater than 1 go to step 4
2. Recursively call Merge Sort on the left array
   1. Return to step 1
3. Recursively call Merge Sort on the right array
   1. Return to step 1
4. Compare the left and right arrays.
   1. If values are not in order, swap the values to create the correct order.
   2. Merge the values in the left and right arrays into the main array in sequential order.
5. Add remaining sorted elements to the final array.

**Iterative**

Pseudocode for the iterative Merge Sort is as follows:

1. Divide the array into chunks of 1 then 2 then 4 then 8, etc.
   1. Send the position information of each chunk in the array to the merge method
2. Put the values of each chunk into temporary arrays left and right.
3. Compare the values in the left and right arrays.
   1. If values are not in order, swap the values to create the correct order.
   2. Merge the values in the left and right arrays into the main array in sequential order.
4. Add remaining sorted elements into the final array.
5. Return to step 1 until all values are sorted.

**Big-Θ Analysis**

For both the recursive and iterative implementation of Merge Sort the time complexity Big-Θ is (n log n) in all cases. This provides the algorithm with an excellent consistency. The iterative implementation took much longer for the same sized array on the computer using the program as implemented. The iterative implementation took about 67 times longer to run than the recursive implementation.

This was confusing and required further investigation. After re-examining the code and restarting the computer the data came out much more evenly. After a fresh restart, the iteration implementation of Merge Sort took only slightly more time than the recursive implementation.

The memory overhead is higher than some other sorting algorithms as it requires an additional array. This gives Θ(2n) for memory needs.

How to get Θ (n log n):

Merge Sort time complexity is T(n) = 2T(n1/2) + Θ(n)

Based on Master Method 2:

If f(n)=Θ(nc) where c=logba then T(n) = Θ (nc log n)

So, c=1 and logba = log22=1

Since both values equal 1 then the time complexity of Merge Sort can be expressed by Θ (n log n)

**Warm-Up**

To “warm up” the JVM so that the program would be run optimally, and all of the necessary methods and variables were locked and loaded I called a method called “warmUp().” This method creates a random array of 1000 values. Then the method enters a 1000 iteration for loop that calls recursive merge sort and iterative merge sort on that array.

Based on the data with and without the warm-up method being called this method worked quite well to get everything going properly.

**Critical Operations**

The critical operation I chose to count was the comparison between two elements. In my code it looks like (left[i] < right[j]). To me this made the most sense as counting the number of divisions would always be the same because of math. It would not tell me anything useful. Counting every time two arrays were merged did not seem like an effective way of getting a more specific efficiency count number as it counts all merging operations. This again is math. If you split up an array into so many chunks, then so many chunks will get merged.

The aspect of the algorithm that was most likely to vary and show the algorithms efficiency was the comparison to determine if swapping values was necessary. This was the only aspect that seemed likely to vary between implementations.

**Graphs**

****

Graph of the Critical Operations Count Based on Array Size



Graph of the Average Execution Time Based on Array Size

Both graphs appear linear. Which is pretty much what (n log n) looks like. Especially given the inconsistency with which the programs are run. Which will lead into the Coefficient of Variance later in the paper.

**Performance**

The performance between the two algorithms in relation to the critical operations count and the execution time is significant. On average the Recursive Merge Sort implementation needs 59% of the comparison operations and 72% of the execution time as the Iterative Merge Sort implementation. This is a marked increase in inefficiency from the recursive to the iterative implementations. It is a little surprising honestly. Most of the time when comparing recursion and iteration, iteration is faster. Merge Sort appears to be one of the few times were that is not the case.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Recursion | | | | |
| Size | Avg Count | Coeff Count | Avg Time | Coeff Time |
| 500 | 2274.33 | 0.30% | 35174.8 | 24.51% |
| 1000 | 5147.13 | 0.27% | 64300.4 | 21.21% |
| 1500 | 8250.91 | 0.25% | 114803.8 | 24.88% |
| 2000 | 11598.29 | 0.22% | 130442.8 | 19.91% |
| 2500 | 15005.34 | 0.19% | 201278.8 | 25.72% |
| 3000 | 18723.06 | 0.21% | 217290 | 26.32% |
| 3500 | 22247.44 | 0.21% | 244902 | 24.00% |
| 4000 | 25945.68 | 0.19% | 268494.2 | 17.29% |
| 4500 | 29726.41 | 0.17% | 317642.4 | 20.68% |
| 5000 | 33860.3 | 0.17% | 341115.4 | 16.05% |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | | | | |
| Size | Avg Count | Coeff Count | Avg Time | Coeff Time |
| 500 | 3859.14 | 0.32% | 51637 | 32.89% |
| 1000 | 8718.13 | 0.21% | 89238.4 | 14.55% |
| 1500 | 14094.03 | 0.16% | 152445.6 | 18.03% |
| 2000 | 19438.08 | 0.13% | 185155.2 | 25.67% |
| 2500 | 25906.99 | 0.12% | 266040.6 | 18.18% |
| 3000 | 31192.63 | 0.09% | 295987.6 | 20.90% |
| 3500 | 37084.73 | 0.08% | 324853.8 | 5.56% |
| 4000 | 42874.75 | 0.09% | 390965.8 | 13.75% |
| 4500 | 51539.57 | 0.08% | 431213.2 | 9.83% |
| 5000 | 56820.34 | 0.07% | 475403.4 | 6.35% |

The results in the tables above show that the findings I stated above. For the average counts the iteration values are close to double the recursion average counts. Execution times appear to be about a third more in the iteration table than the recursion table.

**Coefficient of Variance**

Coefficient of variance is a measure of how spread out the data points are within a similar group aka relative standard deviation. When running the Merge Sort programs each time an array is sorted the time it takes, and the number of critical operations performed vary. This can be because of the state of the computer the program is running on (memory is almost full or CPU is being more heavily used at the time), the array that is sorted (size, how sorted it already is), if there is a warm-up, etc.

What the coefficient of variance will indicate about the iteration and recursion implementations of Merge Sort are “What was the spread of critical operations for each size of array? What was the spread of execution time for each size of array?” The larger the percentage the greater the spread or the less consistent the results are. The smaller the percentage the less spread or more consistent the results are.

As can be seen in the tables above the average counts of both programs have very small coefficient of variance. This means that the number of comparisons that are needed based on an array of size n is consistent. This makes sense as it is based on math. An array of size n will on average require m comparisons before it is sorted.

The coefficient of variance for execution time on the other hand can vary wildly for both implementations of Merge Sort. This also makes sense in that the time needed to perform the sorting is heavily reliant on the performance of the computer and the access the program as to the CPU. If another program has higher priority and interrupts the Merge Sort program, then the execution time will be greater. If the Merge Sort program has greater access it will take less time to finish. If the JVM is not “warmed-up” then the program will have to load in more of the methods and values into faster memory locations. The variation in the number of critical operations can also play a role in the variation of execution time. Finally, how sorted the array already is can lead to longer or shorter execution times. Rearranging an array takes time. If the program just needs to compare values and determine no changes are necessary, the execution time will be greatly lowered.

**Conclusion**

I would say that the results of my program pretty accurately reflect the actual Big-Θ analysis. The execution time graph does look fairly linear, however, so does (n log n). I had to google that because I was a little freaked out.

The most poignant thing I learned during this project is how massively different the execution time can be for unknown reasons. Like I said above the first few sets of data for the iteration implementation had massively greater execution times than the recursive implementation. This was with a warm-up. I still do not really know what caused the difference. My guess would be something with the load my CPU and memory were under at the time.

The coefficient of variance for execution time lends support to the idea that such a massive difference in execution times between similar programs is possible. Because time is so closely related to clock speed, computer hardware, CPU and memory load the variability in time can be quite large. The number of critical operations not varying much also makes sense. The number of critical operations relies on math, not the capabilities or condition of the system the program is running on. A given array size should have a consistent average number of comparisons before it is sorted.

The Merge Sort algorithm itself is a wonderful algorithm for sorting. Due to the consistency in execution time as n grows larger is a benefit for large data sets or data sets of unknown size. As the Big-Θ stays consistent in the best, average and worst case there will not be any nasty surprises for time. This combined with Merge Sort being one of the fastest sorting algorithms makes it a solid choice for many situations.

Interesting project.

**References**

Agrawal. S. (n.d.). *Iterative merge sort.* Geeks for Geeks. <https://www.geeksforgeeks.org/iterative-merge-sort/?ref=rp>

Allain. A. (2019). *Sorting algorithm comparison.* Cprogramming.com.<https://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html>

Geeks for Geeks. (n.d.). *Merge sort.* Geeks for Geeks. <https://www.geeksforgeeks.org/merge-sort/>

Statistics How To. (n.d.). *How to find a coefficient of variation.* Statistics How To. <https://www.statisticshowto.com/probability-and-statistics/how-to-find-a-coefficient-of-variation/>

TutorialsPoint. (n.d.). *Data structures – Merge sort algorithm.* TutorialsPoint.com. <https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm>

Raj. R. (n.d.). *Analysis of algorithms.* Standford University. <http://www-cs-students.stanford.edu/~rashmi/projects/Sorting.pdf>