Hayden Walpole
U1210998
6/11/2024

Analysis Document — Assignment 5

1. My Partner for this assignment was Zijia (Harry) Xie, and they submitted our assignment on the GradeScope group.

2. The pair programming experience went good for us. We wrote all the code together over the course of about 6 hours for the initial code, and 6 additional hours debugging and writing our timing program. We would try to switch roles every 40 minutes, but occasionally lost track of time. I was able to drive closer to 50% this time, and I noticed during debugging Harry was more willing to switch and try both our ideas to solve the problem. We have found our strengths working together better, and I would anticipate continuing to work together in the future.

3. The expected growth rates of Merge sort for the average and worst case are O(N^2logN). The Best case will be O(NlogN). This is because regardless of the order of the list mergeSort will still create the same amount of sublists as it breaks down the list to base case subarrays(Log(N)). Each base case subarray will employ an insertion sort which will either have a big O of O(N) or O(N^2) depending on if the sub array is already sorted or not.

4. We invoked insertion sort by putting an if statement at the beginning of our recursive method taking the first index, and last index of the subarray and check if the difference was smaller than our threshold limit value. If it was longer, we would split the subarray again, but when the subarray size is smaller, we would use insertion sort to sort the array.

5. In order to find the best threshold value for merge Sort we ran several timing experiments with a starting size of 10,000 elements in the array, and incremented the size by 10,000 ten times. Our times to repeat value for these timing experiments was 1,000 times for each value N, taking the Average. For these experiments we used a randomly sorted array from the method in our class method generatePermuted. The results are shown in Figure 1 below. We can see a threshold value of 10-40 is the best. 10 appears to be the fastest, but as we approach 100,000, the threshold value of 40 appears to become faster. Setting the Threshold value to 1 has the same behavior as a full mergesort for our experiment.

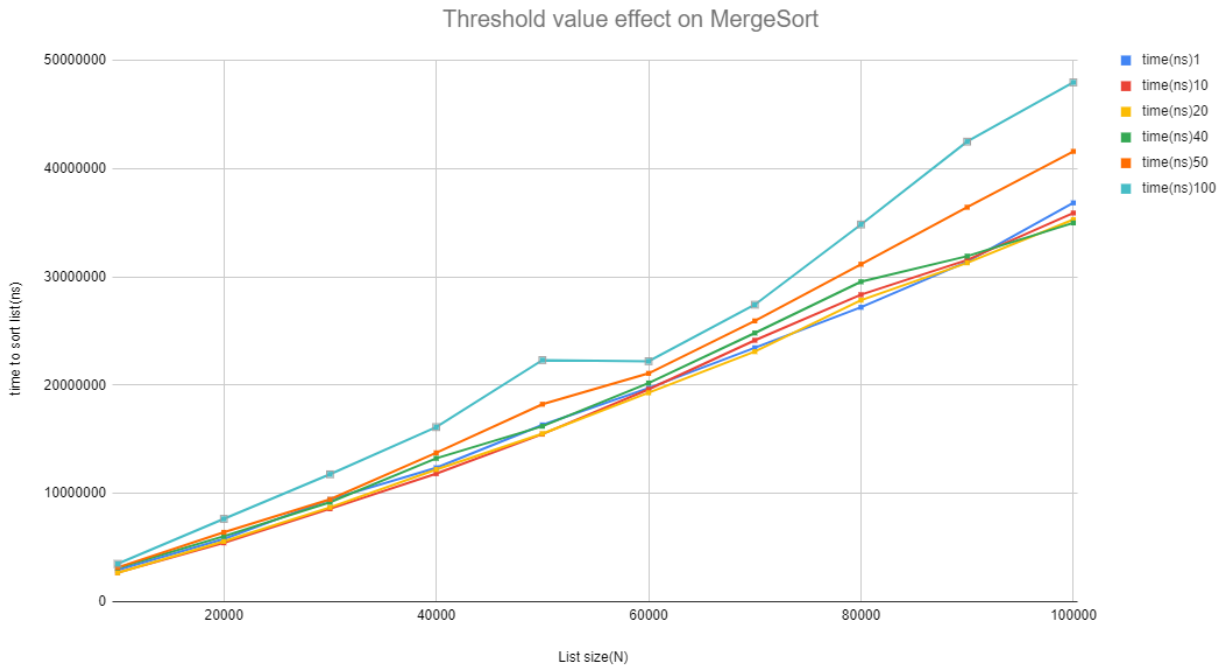

Figure 1

6. The expected growth rates of Quick sort are O(NlogN) for the average and best case, but worst case is O(N^2). The order of the elements will play a small impact on the run time if we choose a good partition strategy, but the way we select the pivot point plays a much more important role. The ideal situation is when we can choose a pivot point that has a value that is close to the median of the list values, splitting the array in half. If we choose the lowest value in the array we have a behavior of O(N^2) as the sorting algorithm slowly goes through each item swapping them back and forth until everything is sorted. This is why it is important to choose a strategy that will find a value that is a median of the values in the list size.

7. **Choose First Value:** Our first strategy is to use the value that was at the first index of our group. This strategy alone has a big-o of O(1) as it will always just return the first index. If the value is in the middle range of the values in our data set the overall notation is O(N^2logN), but if the value is one of the high or lows of our data set it will be O(N^2). This strategy is worse the more sorted our list is because more sorted lists will have a higher or lower value at the first index

   Choose Random: Our second strategy is to use the value that was in a random index at the list. This Strategy alone also has a big-o of O(1) as it will always create a random object and return the first int that is in bounds of the array indexes. If the value is in the middle range of the values in our data set the overall Notation will be o(NlogN), but if the value is one of the high or lows of our data set O(N^2). This strategy is better than our first because if the array is sorted, or not sorted, it has less detrimental impact on the sorting time when we are choosing a value at a random index.

Rule of Thirds: Third was to choose the values at the first, last, and middle index, and take the median of those three values. This strategy alone is also O(1) as there are no for loops, and the code will always compare the values the same way. Overall, with this method we should get O(NlogN) regardless of if the list is sorted or not sorted. If the array is sorted we will take the middle value, and if the array is unsorted the value we return will still be close to the median of the data set.

8. In order to find the best partition strategy for quicksort we ran several timing experiments with a starting size of 10,000 elements in the array and incremented the size by 10,000 ten times. Our times to repeat value for these timing experiments was 1,000 times for each value N, taking the Average. For these experiments we used a randomly sorted array from the method in our class method generatePermuted. The results are shown in Figure 2 below. We can see that using a randomly sorted array choosing the first value and using the rule of thirds are close to the same, but using our rule of thirds strategy is slightly better.

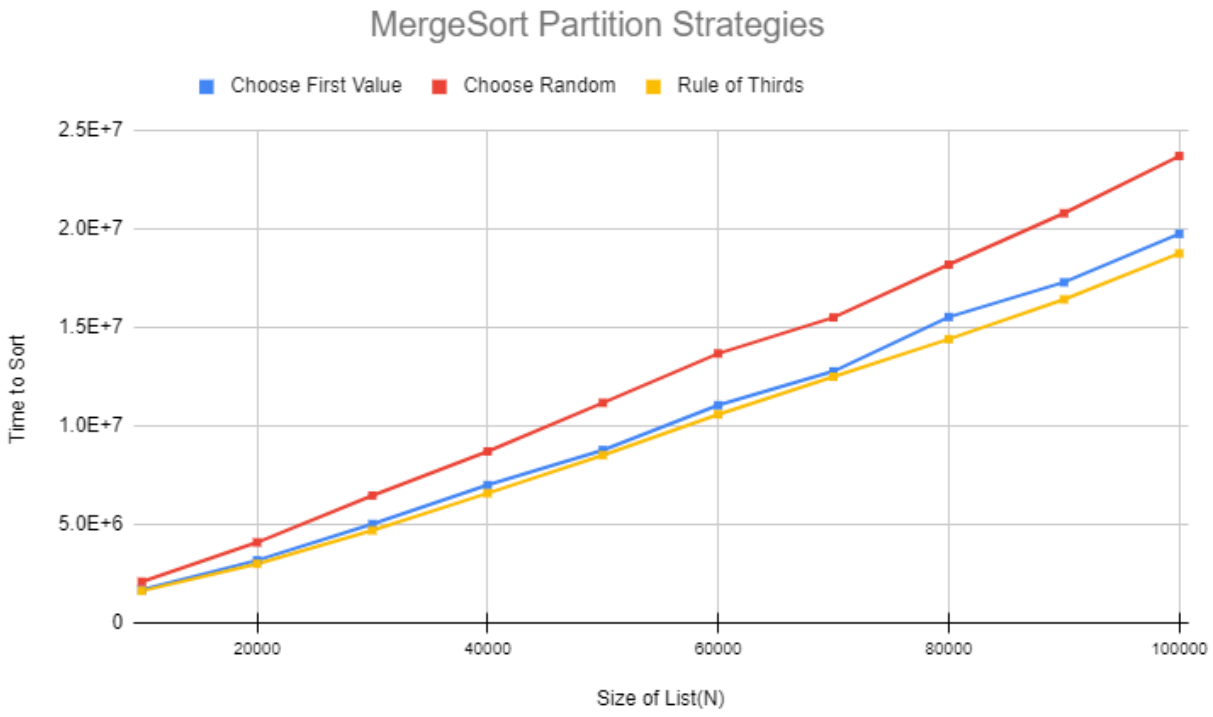

Figure 2

9. To find the best sorting method to use based on the order of the list we ran several timing experiments with a starting list size of 10,000 elements in the array and incremented the size by 10,000 ten times. Our times to repeat value for these timing experiments was 1,000 times for each value N, taking the Average. However, we encountered Stack overflow errors on quicksort at these sizes for ascending and descending arrays, and changed our array sizes to 10,000 to 28,000 incrementing by 2,000 and were able to run our timing tests. We used a threshold value of 40 in our MergeSort, and our third strategy, the rule of thirds, for QuickSort. The results are

shown in the graphs below, Figure 3,4,5. We found that when the list was sorted MergeSort did a lot better, but when the list was permuted, QuickSort was faster than MergeSort.

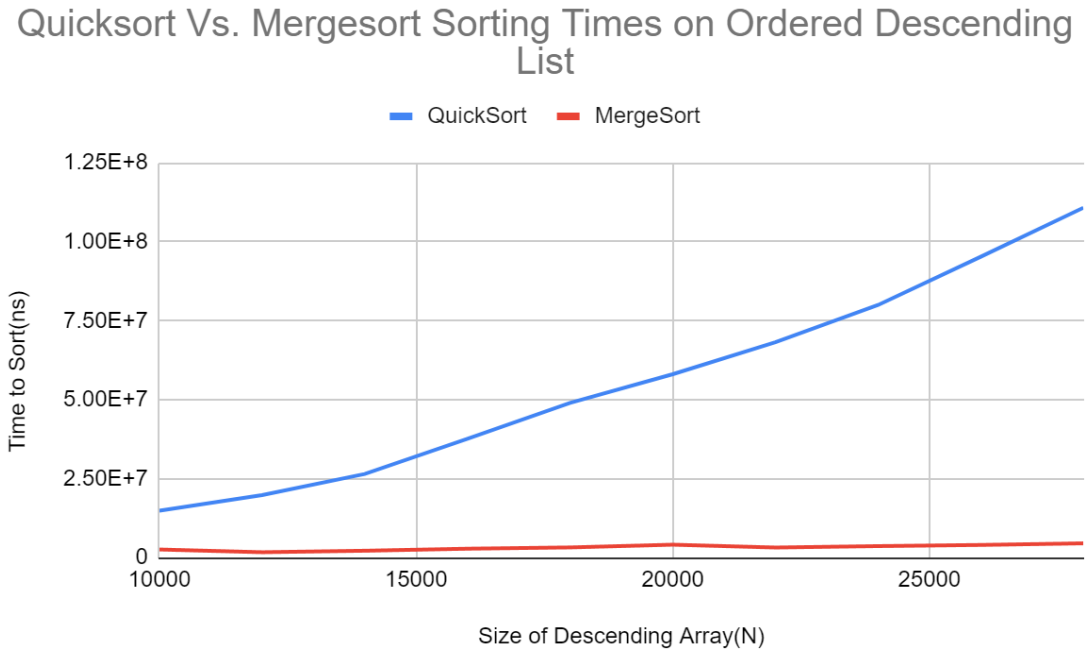

Figure 3

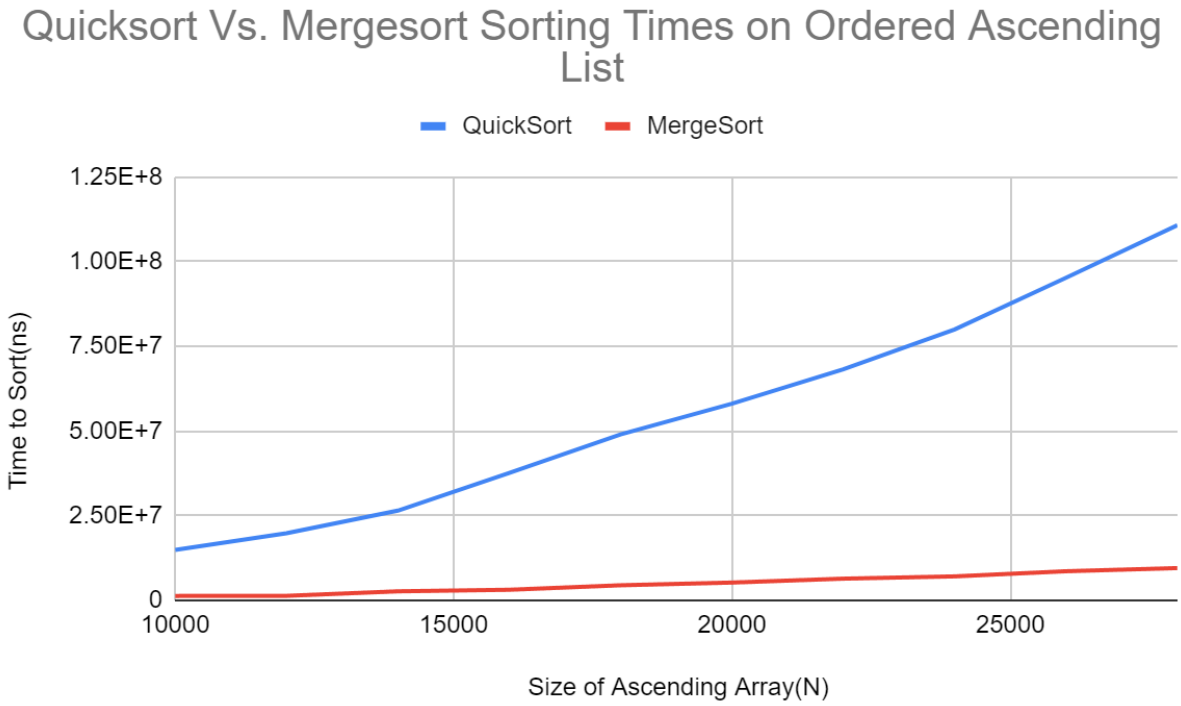

Figure 4

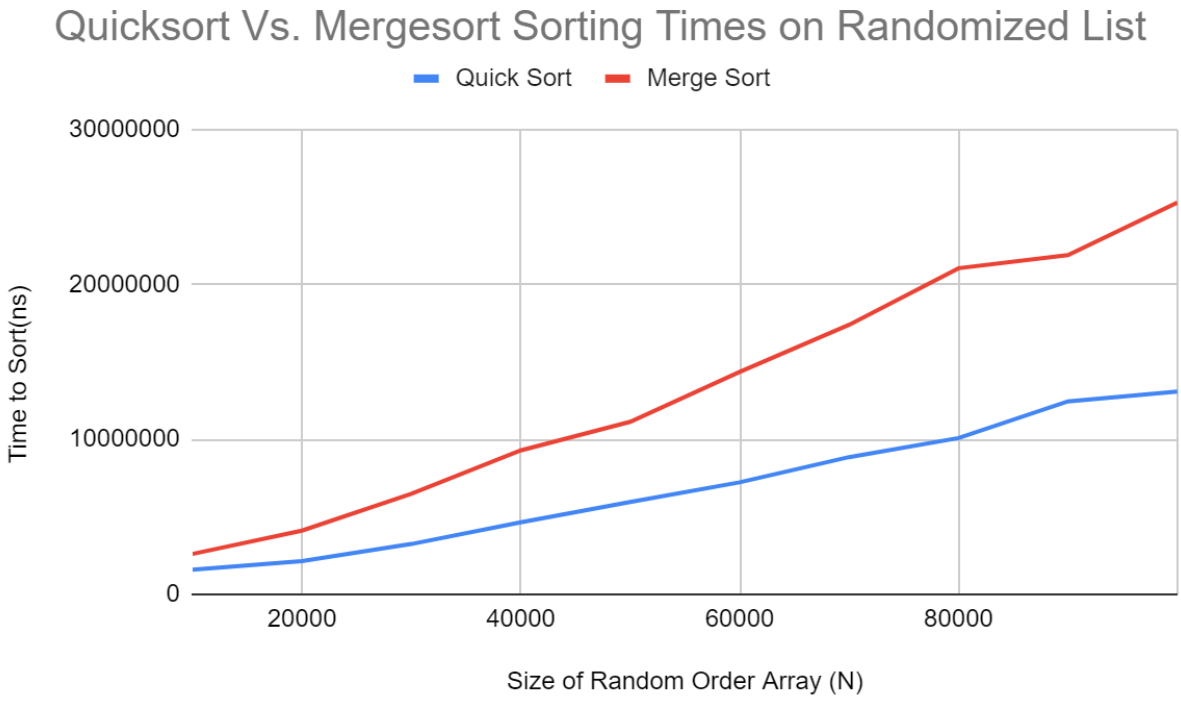

Figure 5

10. The actual run times of our sorting methods exhibit the expected growth rates in most cases. Using the Convergence of T(N)/F(N) we can confirm this. When sorting a list that is sorted ascending or descending Quicksort is O(N^2Log(N)) and MergeSort best fits O(NLog(N)). This makes sense for mergesort, since an ordered list would be a best case for the insertion sort, but it seems a little strange that our Quicksort is modeling a worse case situation and is not the O(NLogN) behavior we would have expected. This indicates that something is probably wrong with our Rule of Thirds partition, and it is choosing a higher or low value in the list as our pivot point. When faced with a permuted list both Lists become closer to O(N^2Log(N)), but quick sort has a lower coefficient, making it faster. This is the behavior we would expect for both these methods with a permuted list.