HW08 Quick Sort and Radix Sort: Time Efficiency and Complexity Reflection and Analysis

Cathy Guang, Wenlai Han

This paper examines the efficiency and complexity of quick sort and radix sort in regard of their interaction with insertion sort. We would directly present our data through the form of a graph and analyze factors that affect the running time of these two sorting methods. The potential factors might be the size of the unsorted list, the level of sortedness of the list, and the repetition of numbers in the list.

1. **Insertion Sort in Quick Sort**

Hypothesis: Insertion sort has a Big Oh of O(n^2) as worst case and O(n) as best case, whereas quicksort has a Big Oh of O(nlogn) as the average case. We propose that change to use insertion sort after we partitioned to lists of small size would help improve time efficiency, because higher level of sortedness in a small list is more suitable for insertion sort which has O(n) as best case, as it reduces time spending on sorting recursively to lists of size 1 in quick sort algorithm.

   a. **Running time vs. MIN_SIZE of arrays using fixed length arrays**

First, we looked into how the average running time of the quick sort changes when we vary the value of MIN_SIZE, which is the size of partition for the sorting program to start using insertion sort. We ran the program with a group of fixed-length arrays containing 10000000 random numbers, but with different MIN_SIZE values. We first chose a small range (3-200) of MIN_SIZE values to see how the average running time varies. We tested this small range four times and averaged out the result to avoid anomalies of the random array generated by computer and other uncontrollable factors. Figure 1 shows the zoomed-in data from 3 to 70, deriving an optimal MIN_SIZE range from 30 to 50. This U-shaped curve supports our hypothesis perfectly with the runtime reaching minimum when MIN_SIZE is relatively small. In order to test what happens for larger values of MIN_SIZE, we tested the algorithm with larger MIN_SIZE (50000-1000000) and found that the run time increases linearly, as shown in Figure 2. The runtime is

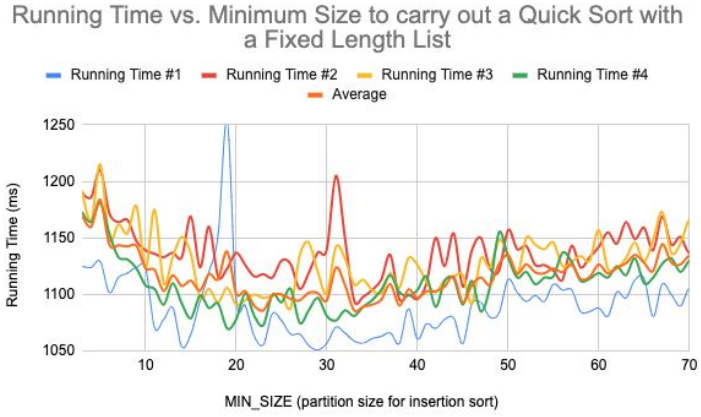

Figure 1. Run Time vs. Relatively Large MIN_SIZE

not approaching the minimum runtime in any sense when MIN_SIZE grows this large.

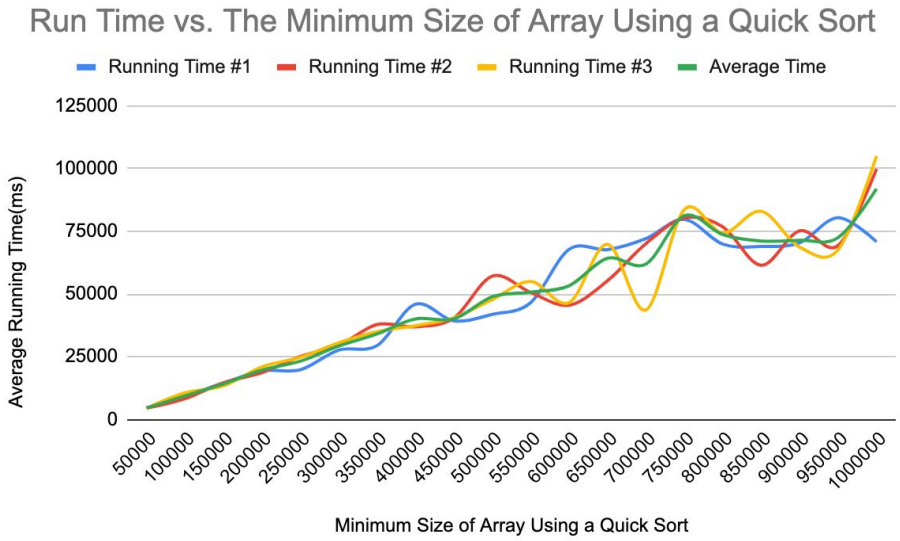

Figure 2. Run Time vs. Relatively Large MIN_SIZE

b. Running time vs. the size of unsorted array within the optimized MIN_SIZE range

In order to look into whether the size of the original array of numbers can impact the right time to switch, we chose the array of length 100000, 200000, 500000, and 1000000 and run the code for each of these length. We chose to focus on the optimal range of the MIN_SIZE values (5, 15, 25, 30, 35, 50, 60, 70, 100, and 500) and ran the code for each different values. We ran the algorithm three times for each and averaged them out to make sure we have avoided anomalies for this fixed MIN_SIZE and factors we cannot control.

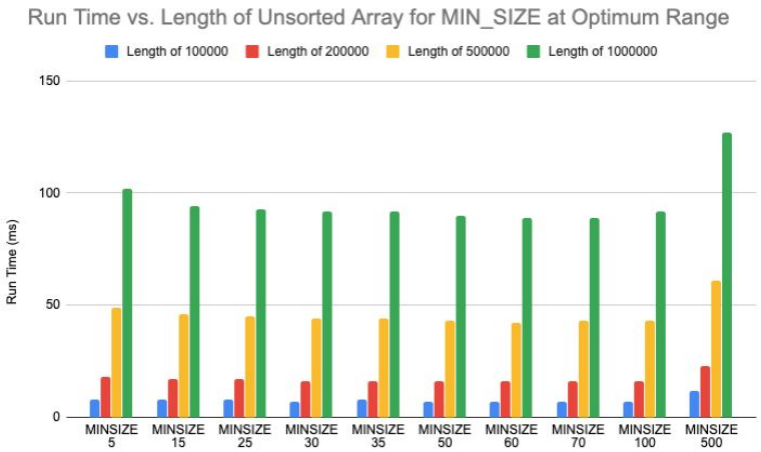

Figure 3. Running time vs. optimized range of MIN_SIZEs with different sizes of unsorted arrays.

From Figure 3, we can see that as the length of array increases, the run Time increases. Also, as MIN_SIZE increases, the average run time decreases first, approaching the lowest value at about 50-70, and then increases as the MIN_SIZE goes

up. This further supports our general hypothesis about using insertion sort inside quick sort algorithm would facilitate the efficiency of standard quick sort. Also, with size changing within each group of MIN_SIZE, we discovered a linear increase of runtime with regard to the length of lists, which runs counter towards our knowledge of the Big Oh (nlogn) of quick sort. This requires further investigation in the future.

c. **Running time vs. the repetition of elements in the unsorted array**
We ran the program with a variation of fillAndShuffle method to make repetition of numbers by taking the remainder after divisions (as shown in comments of our codes). We discovered significant decreases of running times in each level of repetitiveness of numbers. We propose that more repetition of elements in the unsorted list would trigger less swap during quick sort, thus, making the efficiency better.

2. **Efficiency of Radix Sort (msdRadixSort() vs. radixSort())**
Hypothesis: We think that msdRadixSort is generally faster than RadixSort because msd Radix Sort first put all words into buckets of their initial letters and then sort each bucket independently. For radix sort, we need to create buckets for n times where n represents the length of the longest word in the list, so if a very long word appears in the list, the whole running time would increase a lot. On the other hand, when we use msdRadixSort, we use radix sort independently for each list with the same initial letter, so the running time for each list won't be affected by a very long word in a different initial letter list.

a. Average runtime of Radix Sort vs. msdRadixSort for lists of various length
To collect our data, we shuffled the whole “words” list first. To change the length of original “words” lists, we used a for loop in the main method to iterate for 20 times. We created a sublist of “words” lists, increasing the length of it by 10000 each time using the for loop. We ran and recorded the Average Run Time of Radix Sort and that of msdRadixSort for three times and took average to plot in our graph.
From the graph, we can see that our hypothesis is true since the red line is always above the blue line. Also, the gap between the two lines increases as list size increases, which means the difference between running time of the two sorting increased. We think this is because as the list size increases, the efficiency of msd Radix Sort further differs from Radix Sort as the chance of a very long word appearing in the list increases.

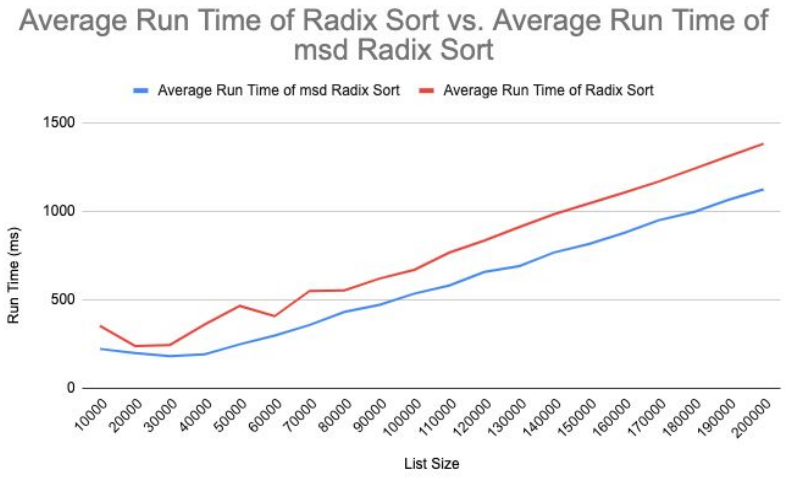

Figure 4. Average Run Time of Radix Sort vs. Average Run Time of msd Radix Sort

b. Run Time vs. Number of Different Random Elements in an 100000-Long List

We further investigated whether the number of repeated elements in the unsorted list affects the running time of Radix sort and msdRadixSort. With a fixed length list (length of 100000), we generated random index to select words that were duplicated for this repetition test. As we see from the graph below, the time efficiency gets significantly better as we further repeat the element in the list. Also, the general running time for radix sort is almost double of msd Radix Sort's. This is because when using more repetition in the list, radix sort would create less buckets to store the values, and more likely to use simpler words with fewer letters to be sorted.

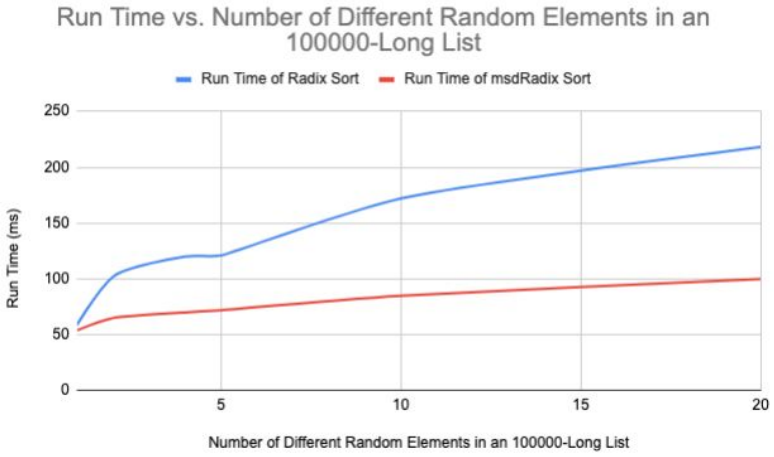

Figure 5. Run Time vs. Number of Different Random Elements in an 100000-Long List

Conclusion

We used line and column graphs to support our hypotheses that insertion sort would be helpful when running quick sort at a relatively small MIN_SIZE range from 30 to 70 and that msdRadixSort outperforms standard radixSort algorithm because of msdRadixSort carries out radixSort in a smaller scope with lower chance of getting the longest item within the whole list.