Thais Chloe Campanac-Climent

NetID: tcc180002

Data Algorithms Assignment 1 Report

**Defining the probes used for the experiment:**

* Movement probe – This probe increments for each element in the array that is moved to a different position (swaps counts as two since two elements are moving from initial position)
* Comparison probe – This probe increments for each comparison made inside the algorithm (i.e. a while() loop and if()/if() else() statement)
* Time probe – This probe records the amount of time each algorithm takes in order to sort the given list
* Memory probe – This probe records the amount of memory each sorting algorithm takes in order to sort the given list

**Defining What Is the Winning Algorithm:**

My winning sort algorithm was based both on my memory and time probe. It will test for the quickest algorithm and if there two algorithms with the same time then the memory probe will be the tie breaker. During debugging/testing my program, I have noticed that even if the list does not change, both the time and memory probe will change for all sorting algorithms therefore all data presented came from multiple trials for accuracy.

**Approach**:

The first part of the program that was completed with the implementation of how to create the different types of lists: in order, reverse order, almost in order, and random order and ask the user to input which list they wish to create and the size of that list. For the in order list, I created a list that starts at 0 and ends at the size of the list minus one. Reverse order list was created in a similar fashion as the in order list, however the list started at the size of the array to one. For the almost in order list, I did the same as in order list but then selected 20% of the list to become randomized. Lastly, random list was randomly selected. Next, I asked the user to input which algorithm they wish to test then the program would execute that algorithm. I then took all probes back into my main class in order to analysis the winning algorithm as defined above. Afterwards I had to reset the lists and objects from other classes in order to allow to user to reuse them. However, the list will not be reset until the user asks to create another list. Lastly, I implemented a GUI into my program with one additional probe in order to better understand my program. I, later on, conducted multiple tests and started to improve upon the efficiency of sorting lists with my best algorithm class.

**Challenges:**

There were a few challenges that I have encountered which was first making the movement probe to accurately represent movement in the list for quick sort, heap sort, and radix sort since this was my first time implementing those algorithms. I feel as though the largest challenge I faced was deciding what will determine the winning algorithm which I have talked about in the Defining the Winning Algorithm section in which was largely part of me overthinking about the solution. I then wanted to make a creative different portion of this project which took a while to have a reasonable idea which will be shown after the Results section below. The last challenge I have encountered was creating a working GUI that both executes the creation of lists and the sorting algorithms while printing the results since this was my first GUI.

**Results:**

**In Order**

**Analysis**: Based on time, insertion sort would be the best algorithm to sort in order lists. For pre-sorted lists, Insertion Sort has the best case scenario since Insertion Sort will only require one comparison per element inside the list and no movement for the list. On the other hand, sorting algorithms like Quick, Merge, Heap, and Radix all involves either building new data structures, require a large amount of recursion, or is based on a counting structure which all takes time to accomplish.

**Almost Order**

**Analysis:** Overall, Quick Sort would be the best type of algorithm to implement for almost in order lists. Although Radix Sort was close to Quick Sort in time, Quick Sort utilizes comparison and divide-and-conquer strategy (based on the partition function) which provides a better sorting algorithm at times then the counting structure that Radix provides. In addition, Quick Sort doesn’t create another structure nor require memory in order to sort the list unlike Merge Sort. For Insertion Sort, the amount of movement is higher then others since the algorithm for Insertion Sort relies heavily on swapping elements and for Selection Sort, it relies heavily on comparing elements.

**Reverse Order**

**Analysis:** Quick Sort, once again, is proven to be the quickest sort to implement for reverse ordered lists. The other two algorithms that came close to Quick Sort are Radix Sort and Merge Sort. Comparing these three sorts, Quick Sort has the least amount of movement required for the lists, but looking at the time graph it seems as though if the lists size were bigger then Merge Sort would have been more efficient then Quick Sort.

**Random Order**

**Analysis:** Radix Sort is shown to be the quickest algorithm for randomly ordered lists and I believe that the reason as to why radix sort is the best is because radix sort is not a comparison sort in which is the opposite case for the five other algorithms. I believe that for randomly sorted lists, comparing elements slow down the algorithm execution instead of comparing values of the elements then sorting based on a counting algorithm. In addition, I noticed that for Selection Sort is exponentially increasing at a |n^2| rate for comparing elements while Insertion Sort is exponentially increasing at a |n^2| rate for the movement of elements.

**Conclusion of Experiments and Future Improvements to the GUI:**

I have noticed after implementing my memory probe that certain algorithms like Merge Sort and Heap Sort would not get a constant value for memory which leads me to believe that either the sorts don’t sort in a consistent fashion or that my memory probe is unreliable to test the amount of memory used to sort the lists. In addition, with exception of in ordered lists, if the size of the lists became larger then Merge Sort would have been the most efficient sorting algorithm. I added an error system for if I forgotten to create a list. I also decided to implement a best algorithm selector. In the first version of the selector, the algorithm decided what type of list was created by the user and then runs the sorting algorithm best suited for that list. I then tried to make improvements on the best sorting algorithm, for example, the algorithm can determine if a list is already sorted and not resort the list. For when the algorithm is mostly reversed, I had implemented a function that flipped the list which decreased the amount of time required for the list to be In Order (Look Below For Graph). I started to implement this feature of flipping a mostly reverse list to Quick and Radix Sort to see if it would decrease the time it takes to sort. However, it seems as though to take a longer time in order to flip the list rather than letting the sorting algorithm sort by itself.

**Appendix:**

Check the Excel Document for data trials and results.