Matthew Wen

Only one of the sorting algorithms is complete. I have finished bubble sort, and there are many improvements that still have to be made before moving on to the next task. I have also finished the load and save file function. Later on this week, my goal is to have a rough draft of shell insertion sort finished.

The load file function works by reading the number of integers the file contains. This is done by looking at the \n it recognizes from the file pointer. Then, it allocates the number of elements it recognizes. Because the file pointer is already at the back of the file, it reads each number backwards and puts the number in order starting from the last index to the first index.

The save files first allocates a string by calculating trying to fit the maximum value for a long data type. If this number is the same for all, I will put it inside a pound defined next time. It then uses the sprintf function to put the numbers into an array, and then fwrite to the file to put the integers inside of it. For each number, it puts it inside of the array, and then fwrites to save the files. This is effective because it keeps the time complexity to 1 without the need to allocate the entire file inside of memory.

For the improved bubble sort, I tried to remove the complexity of the function by minimizing the number of code. In total, the method contains 22 lines inside of the sort function. The method works well with small data types, but I did not work well with the biggest data type. I believe this is because I tried to do the swap method without the addition of more memory. This could be a mistake because the sum of two numbers could go above the maximum number that could be stored inside a long. Further testing needs to be done to determine what is the best action so the 1000000.txt file can do the improved bubble sort within a reasonable time.

As of this time, the time complexity for the improved bubble sort is still O(n^2). The amount of memory used is just O(1). The time complexity for shell sort is expected to be O(n^1.5) and the amount of memory should be O(1). This could change if the function needs to malloc based off of the K value for dividing the list into columns and rows.

There is no runtime for the shell sort, but there are results for some of the bubble-sorts. For 1001 elements, it took the computer 0.006735 seconds, with 330595 comparison and 250115 moves. For 10001 elements, it took the computer 0.305403 seconds, with 33185211 comparison and 24816134 moves. For the 100001 elements, it took the computer 30.277511 with 3328464621 comparisons. For 1000001, it is indeterminate at the moment.

For the improved bubble sort, it contains only 1 for loop, with a boolean value that tells the array to either go left or right. It also stores the last position where an array performs a swap. In total, it takes 17 additional bytes inside the stack to complete the bubble sort.

The approach is not entirely correct. I made some silly mistakes by not using the shellshort method inside the improved bubble sort method. This, as mentioned above, makes the program runs a lot faster. The logical error I ran into is not following the directions where I should of worked on the shell short method first before working on the improved bubble sort method. For the shell short, I am planning to get the memory complexity to 1, therefore there is no need for me to malloc. Therefore, instead of comparing pairs that are adjacent, I would be comparing by different increments. My goal is for the improved bubble sort to use the code from the shell short correctly.