ECE368 Project 1 Report

1 - Description of Algorithms

A concise description of your algorithms, the structure of your program, and any optimizations that you have performed.

The first function found, Load_File, simply read line by line using fscanf, and created an array of longs to store the numbers. And then returned the final pointer to the array. The Save_File functioned in a very similar way, iteratively printing the array to a file.

My Shell_Insertion_Sort function worked by first creating an array of the gaps to be used in the actual sorting. It first checked how deep into the sequence it would have to reach by finding the largest 2^n number which was still smaller than the Size of the array we were inputted. This would ensure that any other combinations of 2^p*3^q would not be missed. It then created an array based of the sequences generated from this. From there it generates the combinations of 2 and 3 checking to make sure that the generated number is less than the Size of the array we have inputted. Then it uses quick-sort to sort the generated sequence in descending orders to ensure the gaps are in proper order before we start the sort. Then it uses each gap we created, iterating through and using these gaps and an insertion sort to sort the rest of the array. All the way down to a gap of one.

My improved bubble sort algorithm takes a similar approach to the use of gaps in shell sort. In a standard bubble sort two elements are always compared when they are next to each other (a gap of 1), however here we can use a larger gap. This helps by quickly dealing with small values which are near the end of a list, which poses the major problem in the efficiency of a standard bubble sort. The gaps decrease each iteration through, and eventually to 1 until the list is fully sorted. By the time the gap is to 1 most of the elements which would take very long to sort have been dealt with, making for an efficient final bubble sort.

The functions to generate the sequences for the functions were described in the paragraphs on the improved bubble sort and shell insertion sort. The last function I have is a comparison function utilized by quick-sort. The function is setup so quick-sort returns a descending array of numbers.

2 - Space and Time Complexity of Sequence Algorithms

An analysis of the time- and space- complexity of your algorithm to generate the two sequences.

Space complexity of the shell insertion sequence algorithm is O(log(N)^2). This is caused by the fact that we need to generate the gap before beginning the sort, and the size of the array is determined by how many times N can be divided by 2 (referred to as *P*) and then by summing the sequence 1+2+3+4...*P*. This series has a total sum of (n*(n+1))/2 which after plugging in *P*

of log(N) simplifies to O(log(N)^2). The worst time complexity is O(N*log(N)^2) due to the nested loops based on the size of the input needed.

For the improved bubble sort the space complexity is O(log(N)) as the size of the array comes out to be the number of times 1.3 needs to be multiplied by itself until it is greater than the input size. After taking this relationship and simplifying you'll receive a relation of log(N)/log(1.3), which yields O(log(N)) for the space complexity. The worst time complexity to generate this sequence is O(log(N)) as the loop will only iterate as many times through as the size of the sequence.

3 - Algorithm Comparisons

A tabulation of the run-time, number of comparisons, and number of moves obtained from running your code on a few sample input files for both Shell Sort and Improved Bubble Sort. You should comment on how the run-time, number of comparisons, and number of moves appear to grow as the problem size increases.

In the table below, the number of comparisons, moves, and run-times of the algorithms can be seen and compared. The improved bubble sort algorithm was much quicker than our shell insertion sort, and the number of comparisons and moves for the improved bubble sort only increased by a magnitude of 1 for each increase of 1 magnitude to the size of the input. The shell sort the number of moves and comparisons increases much quicker. Same can be said for the run times of each.

Algorithm	**Size of Input**	**Run-time (s)**	**Moves**	**Comparisons**
Shell Sort	1,000	0.000000	66,221	30,955
Improved Bubble Sort	1,000	0.000000	13,197	21,704
Shell Sort	10,000	0.010000	1,166,240	550,711
Improved Bubble Sort	10,000	0.000000	186,408	306,727
Shell Sort	100,000	0.090000	18,089,535	8,605,411
Improved Bubble Sort	100,000	0.010000	2,438,928	3,966,742
Shell Sort	1,000,000	0.550000	259,760,828	123,987,151
Improved Bubble Sort	1,000,000	0.370000	30,177,258	50,666,761

4 - Space Complexity of Sorting Routines

A summary of the space complexity of your sorting routines, i.e., the complexity of the additional memory required by your routines.

The space complexity of the improved bubble sort routine is O(1). Due to the fact that the gap can be continuously updated and did not need to be pre-determined allowed this to happen. No additional array of any *N* size needs to be created, keeping the space complexity at 1.

For the shell insertion sort, the space complexity was O(log(N)^2). This is caused by the fact that we need to generate the gap before beginning the sort, and the size of the array is determined by how many times N can be divided by 2 (referred to as *P*) and then by summing the sequence 1+2+3+4...*P*. This series has a total sum of (n*(n+1))/2 which after plugging in *P* of log(N) simplifies to O(log(N)^2).