Stephen Corya
ECE 368
Project 1 Report
10/7/14

For this project, I wrote an algorithm to load a specific file, save a similar file, run a Shell Sort on an array, run an improved version of bubble sort on an array, generate a Pratt sequence, generate a second sequence, save a Pratt sequence to a file, and save the second sequence to a file. The functions are contained in the file “sorting.c”, and their headers are contained in “sorting.h.”

My algorithm to generate the first sequence is of space complexity O(n log(n)), where the maximum value in the sequence must be less than 'n'. This same algorithm is of time complexity O(log(n)), where 'n' corresponds to the same value. The logarithmic time complexity is based on the fact that the algorithm approaches the max value at a greater and greater rate as the algorithm progresses. I may have been able to reduce the space complexity to O(log(n)) by not initializing the sequence within an array of 'n' elements, rather calculating a value to use a the size based on a value less than 'n' using logarithms. I was unsure if this would compile properly on the Shay machine, so I elected to leave the space complexity at O(n log(n)). The time complexity is O(log(n)). My algorithm to generate the second sequence is also of O(n log(n)) space complexity and O(log(n)) time complexity. Again, the space complexity could likely be further optimized.

The functions containing both these algorithms produce output arrays of size O(log(n)), but still require O(n log(n)) space to run. These output arrays represent the gap sequences used by the sorting algorithms. Producing outputs like this allows the sorting routines to both use O(log(n)) additional space complexity, as each sorting routine uses a sequence of the same magnitude relative to its input size. Each sorting routine uses O(1) additional space other than the sequences they must utilize.

The Shell insertion sort runs with a time complexity of O(n log(n)). The insertion sorts that utilize gap spaces run in logarithmic time relative to the size of the entire array, leaving a nearly sorted array to be 1-sorted in linear time after they all run. Similarly, the improved bubble sort runs in

logarithmic time with large gaps, but as the gaps diminish the time nears O(n^2) time complexity that is consistent with bubble sorting. The array must also be 1-sorted using a bubble sort, which requires O(n^2) time as well; hence, the run time for this improved bubble sort is O(N^2).

Shell Insertion Sort

Input size	**Run-time (s)**	**Num. comparisons**	**Num. moves**
1,000	0.000212	35,266	66,221
10,000	0.007962	615,529	1,166,240
100,000	0.058340	9,484,124	18,089,535
1,000,000	0.524954	135,697,411	259,684,562

Improved Bubble Sort

Input size	**Run-ime (s)**	**Num. comparisons**	**Num. moves**
1,000	0.004244	1,376,990	13,095
10,000	0.141474	139,956,550	187,818
100,000	13.971682	794,952,497	2,448,540
1,000,000	[overflow]	[overflow]	[overflow]

These quantitative values appear to correspond to my calculations of time complexity. These results (run time, number of comparisons, and number of moves), increase with time complexity for their respective algorithms. Run-times for the improved bubble sort on arrays of sizes on the order of one million elements was on the order of many minutes. If I were to continue to work on this project, I would try to learn more about library linking, logarithms, and try to make my algorithms more efficient.