ECE 368 Project 1

Report

Asheem Chhetri

1. The Shell_Insertion_Sort function was implemented to loop every gap in order of $2p'3q'$ gaps. Each iteration it goes through, it divides the gap by 3 and multiplies it by 2. Other than that, the algorithm itself inside the shell sort is very similar to insertion sort. The Improved_Bubble_Sort function iterated each loop through and divided by 1.3, except the case when the gap was 9 or 10, where it was replaced by 11. Other than that, inside the shell sort, the algorithm is similar to a bubble sort – using 2 for loops to check each possible instances.

2. Sequence 1:

   It seems that the time complexity of the seq_1 is o(n), after ignoring the constants and looking at how many time the line of codes are running within while loop, and performing summation. Thus for worse case will be O(n), while for best case it can be O(1), assuming none of the while loop is encountered.

   Regarding the space complexity, it do not take any extra space, thus it is O(1).

   Sequence 2:

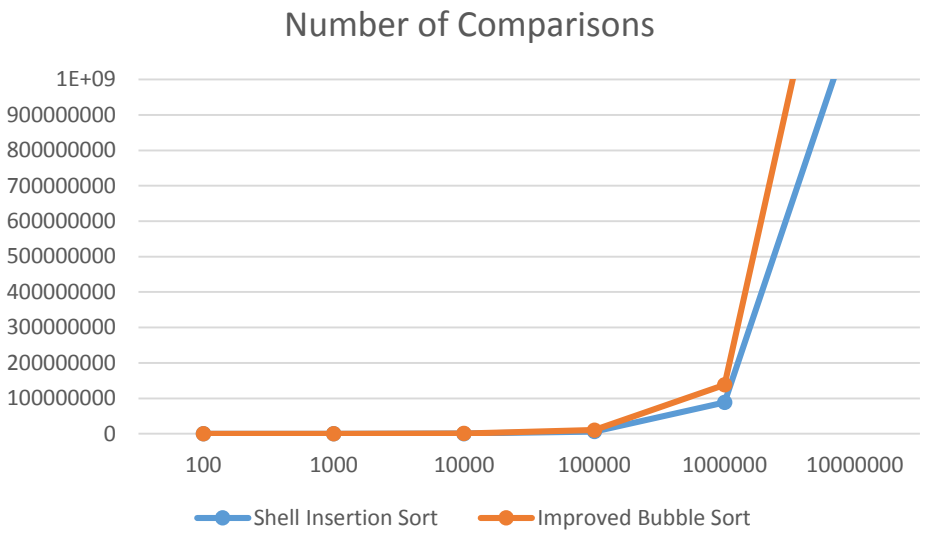
   It seems that the time complexity of the seq_2 is also O(n), using same argument used for seq_1, while the space complexity seems to be again O(1), as no extra space is used to generate seq_2.

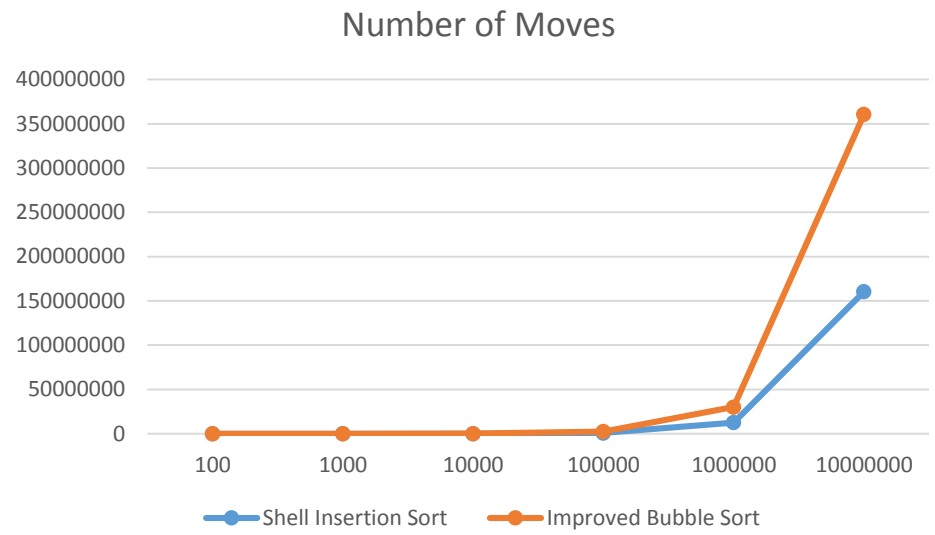
3. Tabulation of the run time, number of comparisons and number of moves.

	Shell Insertion Sort			Improved Bubble Sort		
Array Size	Run Time in seconds	Number of Comparisons	Number of Moves	Run Time In seconds	Number of Comparisons	Number of Moves
100	0.0	1,199	242	0.0	1,294	786
1,000	0.0	24,975	4,743	0.0	21,704	13,197
10,000	0.0	421,499	70,236	0.0	306,727	186,408
100,000	0.6	6,338,375	972,318	0.3	3,966,742	2,438,928
1,000,000	0.75	88,624,899	12,708,296	0.4	49,666,762	30,144,183
10,000,000	9.05	1,178,542,175	160,508,072	4.31	606,666,765	360,671,748

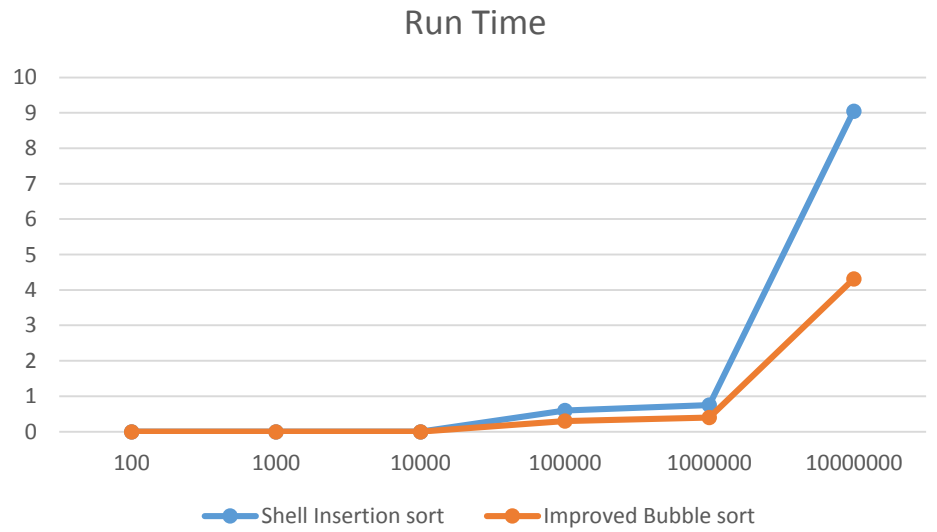
By looking at the table, we can say that as the size of the array increased the time required by both algorithms increased, and also the number of comparisons and moves increased substantially with increasing size of problem. The improved bubble sort algorithm was much quicker than our shell insertion sort when the problem size increased for "number of comparisons", while "number of moves" was lesser for shell sort. Regarding the run-time, they both take same time of 0.0 till array size of 10,000, but after that as the size increases rapidly, improved bubble sort takes almost half the time as that of shell sort, as it is expected because it incorporates both shell and bubble technique to sort the data.

As we can see from the charts on next page:





We can see that Shell Sort and improved bubble sort are substantially increasing number of comparisons closely as the problem size increases but still the improved bubble sort is doing better here, while for the number of moves, improved bubble sort is increasing much rapidly then improved bubble sort.



As array size increases, shell sort takes more time then improved bubble sort.

4. 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 predetermined 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).