Matthew Wen

For Shell_Insertion_Sort, it first calculates the number of times 3 can be multiplied to 1 to equal to a value greater than N, which is the number of elements inside the array. Then it allocates an array which stores. At index 0 of the array, it stores 3 ^ 0 , while the element at index x stores 3 ^ x. Then, for each index, multiply it by 2 until it is just about greater than N. Afterword, get the biggest element from the array to get the current gap sequence, use the gap sequence, then divide it by 2. the element at that index is not divisible by 2, then the index contains a value which is a power of 3. That means that 3 ^ index * 2 ^ a, where is a value from 0 to log (N / 3 ^ index) / log (2). For each gap sequence, it runs insertion sort.

For Improved_Bubble_Sort, it stores an element N inside a variable. Then it calculates the gap sequence by dividing that global variable by 1.3. For each gap sequence, it performs shell short to get a sorted sequence.

I improved the optimizations of the program by creating multiple local variables. If the algorithm only deals with data types that are close to each other, it does not require additional time to reach a value that is not in locality. Also, I limited the amount of time it incremented N_COMP and N_MOVE, therefore counting the number of comparisons and moves does not take up runtime.

The time complexity to develop 2^p * 3^q would be O(logN) because it would take Log N / log (3) to determine the value of q if p was 0, and then it would take log(N) / log (2) to determine the p value if q was a certain quantity. The space complexity is O(log(N)) because it allocates Log(N) / Log(3) bytes. It also For printing the sequence, by default, it allocate N spaces, and then uses a similar method to develop the same sequence used for Shell Sort Insertion.

The time complexity to develop the bubble sort gap sequence would be O(logN) because the number of gaps is determined by Log(N) / Log(1.5). The space complexity in the sequence is one because to find the next gap, I divide my current gap by 1.3. For storing the array, it is O(N) because it preallocate N to store the sequence.

	Shell Sort Insertion.			Improved Bubble Sort		
Input Size	# CMP	# SWAPS	TIME (s)	# CMP	# SWAPS	TIME (s)
1000	35306	4311	0	24071	13095	0
10000	615596	64818	0	349335	187818	0
100000	9484225	878713	0.02	4482925	2448540	0.02
1000000	135697553	11710260	0.28	55746098	30237996	0.19

The time increased by a multiple of 10 from 100000 to 1000000 inputs. The big O of the algorithm is O(n * (log(n))^2) because it perform insertion log(n) * log (n) / (log (2) * log (3)) with an average with a complexity of sqrt(n). The time for bubble sort increased by 100, but the number of comparisons and swaps increased by a multiple of 10. The comparison and swaps for both shell insertion and bubble is both linear as the input size increased by each increment of 10. This is noticeable because each column increase by a constant factor.

The space complexity of the Shell_Sort algorithm is O(log(N)), because it allocates only Log(N) / 3 bytes for an array that gets gap sequences by dividing it by 2 from each index of the array. Then, because insertion sort have a big O(1), the total space complexity of Shell Sort is O(log(N)).

The space complexity of the improved bubble sort algorithm is O(1) because it did not require to allocate more memory to store the sorting algorithm; I divided each gap by 1.3 to get the newest gap sequence.

For saving the sequence, I used O(log(N)) to preallocate the array for both sequences. Then it prints the indexes that are used inside the array.