The entire program contains 7 parts:

1. A *main* function provided by the professor;
2. *Load_File* allocates the memory for the input array and read the Size as well as the data.
3. *Save_File* function opens an external text file to output the size and the array.
4. *Shell_Insertion_Sort* calls *seq_gen_1* function to generate the sequence. Then, sort it in descending order. After that, the function uses the sequence to do shell insertion sort.
5. *Seq_gen1* function assigns 1 to seq1[0] at first, then have two pointers, pointer2, and pointer3 pointing to seq[1] at first. The function compares the value at pointer2 multiplies 2 with the value at pointer3 multiplies 3. It saves the smaller value to the next index, and increments that pointer. If values are the same, it saves the value and increments both pointers. If the value generated is greater than Size, the function returns.
6. *Improved_Bubble_Sort* calls the *Seq_gen2* function to generate the sequence. Then, the function divides the array into several small vertical arrays. The function sorts each vertical array by doing the following:
   - Comparing the 1st element of **first vertical array** with 2nd element of that array (the 0th and the Kth of the original array). Swap if necessary. Continue for other vertical arrays.
   - Comparing and swapping the 2nd element of first vertical array with the 3rd element of that array. Continue for all vertical arrays, then proceed to comparing the 3rd elements and the 4th elements of each array, until the greatest value is located in the last index.
   - Reduce the vertical array size by excluding the last element of **each** array. Repeat the vertical bubble sorting, until all the vertical arrays have been sorted.

   Then, read the 2nd K value from seq2, repeat the process of vertical sorting.
7. *Seq_gen_2* function generates sequence 2 by dividing the Size by 1.3^n, where n is increasing from 1 to the point where the result is 1. Then, change value from float to int. If the result is 9 or 10, round it to 11.

*Both sort functions free the sequence at the end.

The time complexity and space complexity of generating the 1st sequence are both O(n). The time complexity and space complexity of generating the 2nd sequence are both O(n) as well. Both bubble sort and shell insertion sort do not allocate extra memory other than the temp variable and less than four pointers. The space complexity is O(1).

For bubble sort: number of comparisons has O(n^1.2) number of moves has O(n^1.2), increasing of time is less than O(n^1.2).

For shell insertion sort, number of comparisons has O(n^1.3) number of moves has O(n^1.3), and increasing of time is less than O(n^1.3).

Case	Bubble			Insertion		
	Time	Comp	Moves	Time	Comp	Moves
1000	0	1.97E+04	4.38E+04	0	3.44E+04	6.62E+04
10000	0	2.87E+05	6.36E+05	0	6.04E+05	1.16E+06
100000	0.03	3.67E+06	8.15E+06	0.04	9.38E+06	1.81E+07
1000000	0.24	4.57E+07	1.01E+08	0.64	1.34E+08	2.60E+08