BBM204: Software Laboratory 1
Assignment 1 Report

Süleyman Ekmekçi, 21985921

April 14, 2021

1 Software Design Notes

1.1 Problem & Idea

The most important reason for analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application. Moreover, the analysis of an algorithm can help us understand it better, and can suggest informed improvements. Algorithms tend to become shorter, simpler, and more elegant during the analysis process.

1.2 Approach

1.2.1 Design

I created TestObject in order to append and sort the lists. This class has two attributes, randomString and randomInteger. All sorting algorithms are implemented in a class named SortAlgorithms. I created double variables in order to calculate time for each sorting. All sorting algorithms are generic type so any list can be sorted (only the objects have to have compareTo method).
I created lists with size of 2^1 to the 2^{12}. For each list, program calls the sort algorithms for given time (I gave 50). First n (I gave 3) sorting times are not included to the result as it can ruin the results because of the first compile times. Program calculates average of the time and prints the results.
In order to calculate and compare best and worst cases for each algorithm, I created a class named WorstCase. It basicly runs every algorithm 1000 times for the lists with size of 2^1 to the 2^{13} . It creates best, random and worst case inputs for each algorithm and returns the average times. You can see the overall results in the last page.

1.2.2 Number & String Generation

As said above, TestObject has two attributes, randomString and randomNumber. For random cases, I created a function named generateList it takes size of the list as parameter, returns TestObject list. For each element, I created random double and converted it to long bits, after that it is converted to hexString and here our random generated string. For the numbers, I just used Random class.
For the best cases (mostly sorted lists), I created sorted lists for the given list size.
For the worst cases (mostly sorted in descending order), I created sorted lists in descending order for the given list size.

1.2.3 Stability

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. In my approach, I created new class to find stabilities of the algorithms. Program intializes a list with 18 elements. 5 of the elements are randomly generated. Other 13 elements' strings are named as test1-test2..test13. However, test7 is named as intersection and numbers of these elements are 0. With done that, program adds whole elements to another list but in opposite order. After that, program sorts two lists. If the intersection objects intersect in the same index, it means that the algorithm is stable. This algorithm is not %100 guarantees the result but increasing the number of the elements and compiling more than one time will increase the chance. It becomes almost %100 correct. Let's see this process.
Assuming that we have 3 objects, they will be named as test1, intersection, test3 with the number 0. List1 will be like this: [test1 0, intersection 0, test3 0]. As explained, List 2 should be like this: [test3 0, intersection 0, test1 0]. Program will sort both of the lists. Since the numbers are 0, their appearance will not change so both intersections will be located at index 1. If any sorting algorithm can preserve this sequence, it means that it is stable.

General stabilities:
Bitonic : Not stable
Shaker : Stable
Stooge : Not stable
Comb : Not stable
Gnome : Stable

1.3 Time and Space Complexity Analysis

1.3.1 Comb Sort Analysis

Combsort iterates through the list multiple times, swaps elements that are not ordered. It has a special iterator which looks at elements with a certain distance called the gap. The gap is decreased as the algorithm continues.
The main sequences of the algorithm are like this
1 - Set the gap (input size is the great start for this)
2 - While elements are not sorted
2.a - Shrink the gap
2.b - Beginning from start to the end, compare two pairs which have n gaps between them
2.c - Swap the numbers in the pairs.

Time Complexity: Our comb sort implementation has actually $O(n * log_{1,3}n)$ iterations both in worst and best case since number of loops depends on the gap, which decreasing by 1,3 about n times.However, swap operation is an important factor in determining the difference between the best case and the worst case as program executes nested while loops in any case. We can easily say that in the worst case, program should make the most swap operations.
I created two inputs one is randomly generated list and the other is sorted list. As you understand, in sorted list there won't be any swap operation. On the other hand, randomly generated list most probably will have a lot of swap operations. These swap operations causes the time difference as you can see in the graph. I also plotted n^2 so that we can see the relation easily.

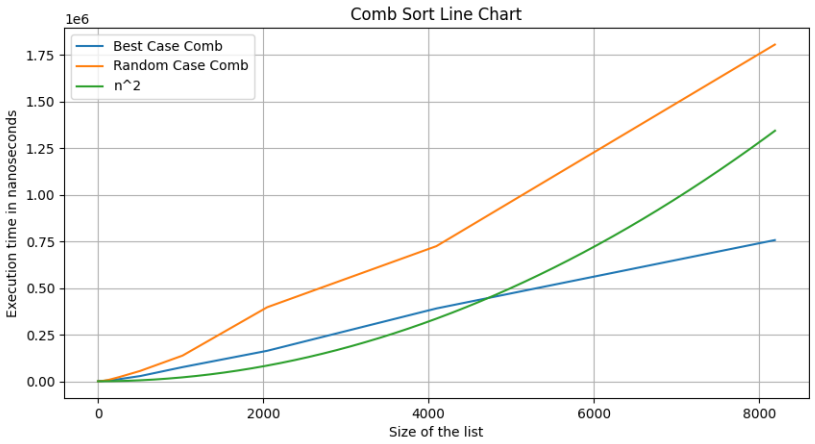

Figure 1: Comb Sort difference between best case input and input which is almost worst case.

Space Complexity: This sorting algorithm mostly based on iterations. It does not save any extra value or data. So its space complexity is constant O(1).

1.3.2 Gnome Sort Analysis

Main idea of the gnome sort to swap ajdacent elements if they are not ordered as wanted to sort the whole list. It is similar to Insertion sort however, gnome sort swaps only adjacent elements.
The main sequences of the algorithm are like this
1- Start iterating the list from second element of the array
2- Compare it with the left element of the current element
2.a - If the current element is greater than or equal to the left element of the current element, go one step right
2.b - If the current element is less than the left element of the current element, swap these two values and go one step left.
3- Repeat these steps till reach to the last element.

Time Complexity: Gnome sort is based on iterations which depend the initial sequence of the input. If the given input is already sorted, program won't need to go backwards so it will just iterate the list. It will be the best case and time complexity of the best case is O(n). Average and worst case has $O(n^2)$ since it is possible to go backward for each element while sorting. As you can see in the graph, random and worst case are nearly equal and grows exponentially. On the other side, best case is works really faster so it is almost being neglected. I also plotted n^2 and n so that we can see the relation easily.

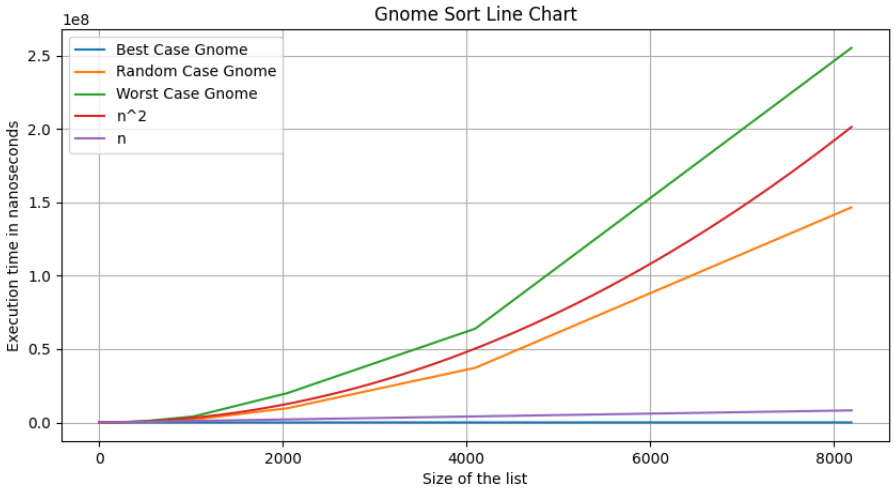

Figure 2: Comb Sort execution time / size of the list graph.

Space Complexity: This sorting algorithm mostly based on iterations. It does not save any extra value or data. So its space complexity is constant O(1).

1.3.3 Shaker Sort Analysis

Shaker Sort traverses through a given array in both directions alternatively while it is sorted. It first starts from left to right then right to left and each loop it swaps the given adjacent elements if the items are not ordered.
The main sequences of the algorithm are like this
1- Start iterating elements from left to right and compare each adjacent elements. If the left value is greater than the right value, swap values. At the end, largest number will be located at the end of the array.
2- Iterate elements from right to left and compares each adjacent elements. If necessary, adjacent items are swapped.

Time Complexity: Shaker sort needs to iterate each element from left to right and after that it needs to iterate from right to left. In average and worst case, its time complexity is $O(n^2)$ as it needs to iterates both directions and program needs to do this till the list is sorted. If the input is already sorted, it just needs to iterates one times. So in the best case, time complexity will be $O(n)$. I also plotted n^2 and n so that we can see the relation easily.

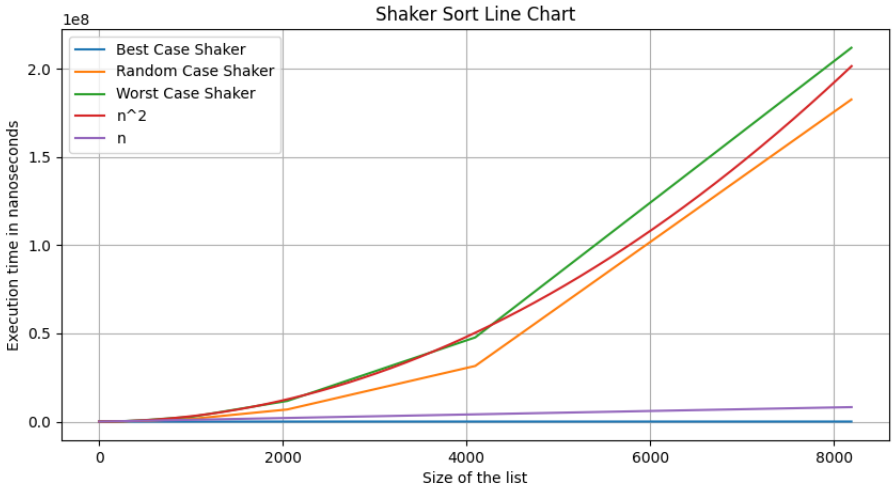

Figure 3: Shaker Sort execution time / size of the list graph.

Space Complexity: This sorting algorithm mostly based on iterations. It does not save any extra value or data. So its space complexity is constant O(1).

1.3.4 Stooge Sort Analysis

Stooge sort is based on recursive calls. It divides the array into two parts every parts' size is 2/3 of the array. Then it starts sorting in first 2/3 part and then it starts sorting in last 2/3 part. After that, sorting is done on first 2/3 part to ensure the array is sorted.
The main sequences of the algorithm are like this
1- If the first element of the array is greater than the last element of the array, swap them.
2- Call recursive sort for the first 2/3 part of the array.
3- Call recursive sort for the last 2/3 part of the array.
4- Again call recursive sort for the first 2/3 part of the array.

Time Complexity: Stooge sort has $O(n^{log3/log1.5})$ as every time it recursively calls itself, each time on an array whose size is 2/3 of the original array's size. As you can see in the graph, its worst, average and best cases are almost same. I also plotted $n^{log3/log1.5}$ so that we can see the relation easily.

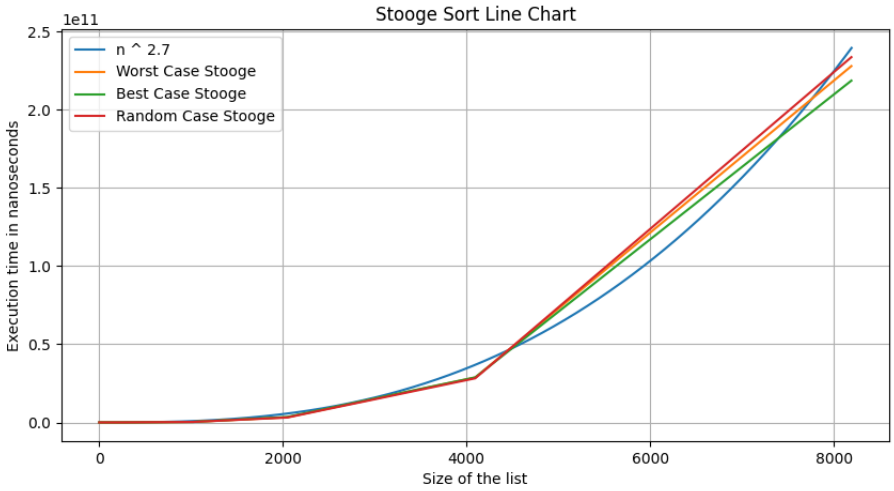

Figure 4: Stooge Sort execution time / size of the list graph.

Space Complexity: This sorting algorithm mostly based on recursive calls. It basicly stores the array and gives as parameter for each recursive calls. Program needs extra space to do it. So its space complexity is O(n).

1.3.5 Bitonic Sort Analysis

Bitonic sort is a comparison based recursive sorting algorithm. It can only be done if the number of elements to sort are 2^n. The procedure of bitonic sequence fails if the number of elements are not as stated.
The main sequences of the algorithm are like this
1- Form a Bitonic sequence from the given random sequence. We can consider it as the first step in our procedure. After this step, we get a sequence whose first half is sorted in ascending order while second step is sorted in descending order.
2- Compare first element of first half with the first element of the second half, then second element of the first half with the second element of the second half and so on. Swap the elements if any element in the second half is smaller.
3- After the step 2, we got all the elements in the first half smaller than all the elements in the second half. The compare and swap results into the two sequences of n/2 length each. Repeat the process performed in the second step recursively onto every sequence until we get single sorted sequence of length n.

Time Complexity: Bitonic sort has $O(n * log^2 n)$ if it is runned non-parallel environment. As you can see in the graph, its worst and average cases are almost same. I plotted a graph of the $O(n * log^2 n)$. Their behavior is almost the same.

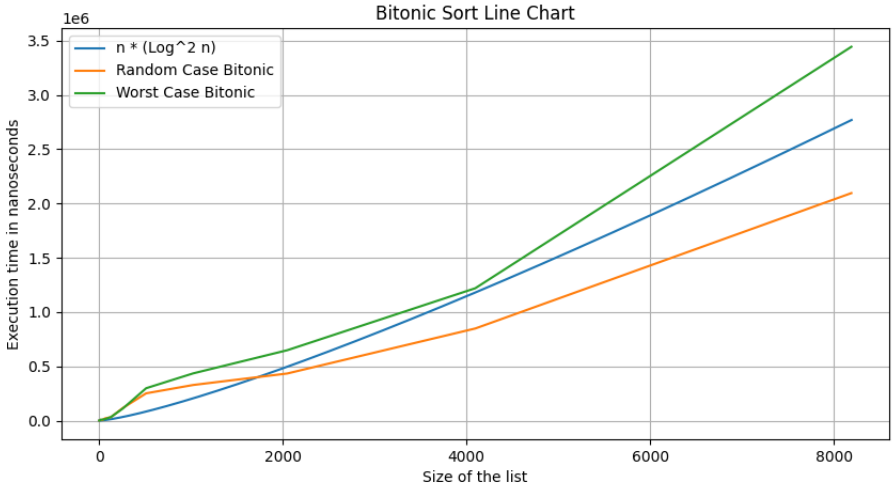

Figure 5: Bitonic Sort execution time / size of the list graph.

Space Complexity: This sorting algorithm mostly based on recursive calls. It basicly stores the array and gives as parameter for each recursive calls. Program needs extra space to do it. So its space complexity is $O(n * log^2 n)$.

1.3.6 Overall Analysis

Stooge sort is slower than rest. It really affects other algorithms in the graph so I decided to put two graphs. With stooge and without stooge. Overall we can see that Stooge > Shaker > Gnome > Bitonic > Comb. Stooge is the slowest and comb is the fastest algorithm.

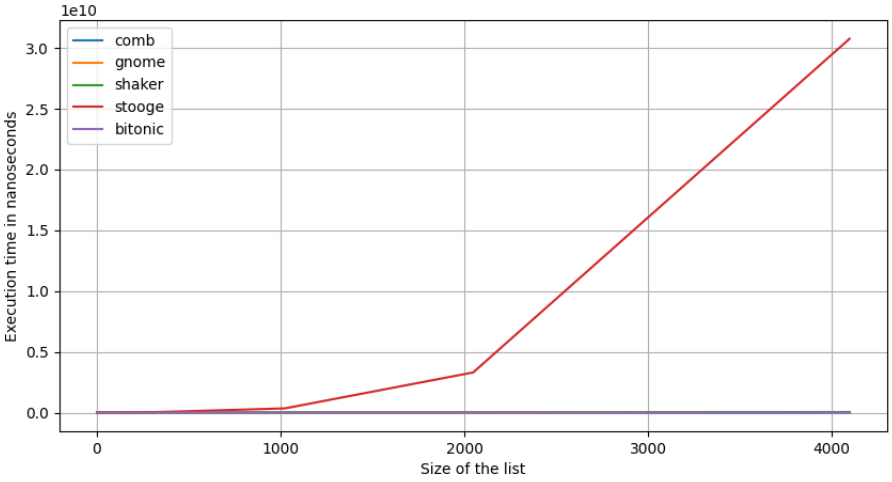

Figure 6: Overall results .

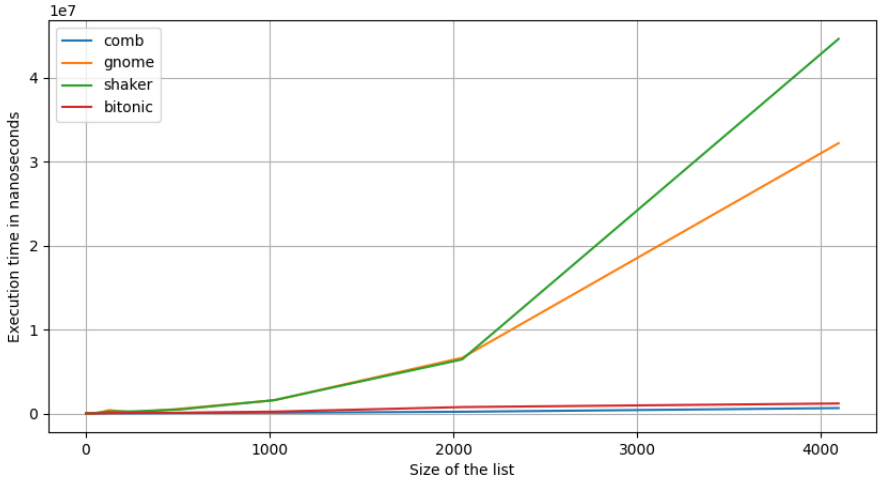

Figure 7: Overall results (stooge excluded).

Algorithms/n	2	4	8	16	32	64	128	256	512	1024	2048	4096
Comb Sort	582	2600	2938	10440	4698	13474	52382	15858	58852	83074	198896	636852
Gnome Sort	214	1272	5182	8148	9646	49946	395390	112268	551010	1564812	6629934	3,22E+14
Shaker Sort	144	1880	2818	10724	10158	89896	225602	242440	472800	1573904	6431478	4,46E+14
Stooge Sort	104	612	3254	27908	174446	739424	1483238	1,39E+13	1,18E+16	3,46E+16	3,30E+17	3,08E+20
Bitonic Sort	354	3368	3218	2606	3542	8112	26690	69006	84714	212944	756640	1183346

EXECUTION TIMES IN NANOSECONDS

Figure 8: Overall table results.