Hacettepe University

Computer Engineering Department

BBM204 Software Lab II - 2021 Spring

Assignment 1
Sort Algorithms and Analysis

March 23, 2021

Student Name:
Uğurcan ERDOĞAN

Student Number:
b21827373

1. Objective

In this assignment, we were asked to implement various sort algorithms, explain how they work, and make inferences about runtimes. With the help of experimental analysis, we were expected to have proved the complexity graphs of algorithms with real-time trials.

2. Implementation

In my implementation, I mainly used Java's class constructs and the concept of *Polymorphism*. After implementing the desired sort algorithms with various classes, I gathered them all in a common *interface*. With the **sort** method in the interface, I have overridden a common sorting function in each algorithm.

As a theme, I named each of my objects forest. My classes have two attributes: number of trees(integer) and forest name(string). Basically, I used the number of trees value when sorting. For the stability control, I used the forest name string. I kept all the objects in the forest arrays.

First of all, I created two arrays with a length of numerical value that the experimenter will give. I filled one of them with random integer values using Java's random library. I filled the other array with the desired length with the help of a function that can fill an array in descending order.

After the filling process, I sorted the clones of both arrays with the desired algorithms in the *printAvgElapsingTime* method. During sorting, I skipped the first of the desired repetition value and measured the average working time in milliseconds with the help of the stopwatch class that I created. Then I did the process of showing this to the experimenter.

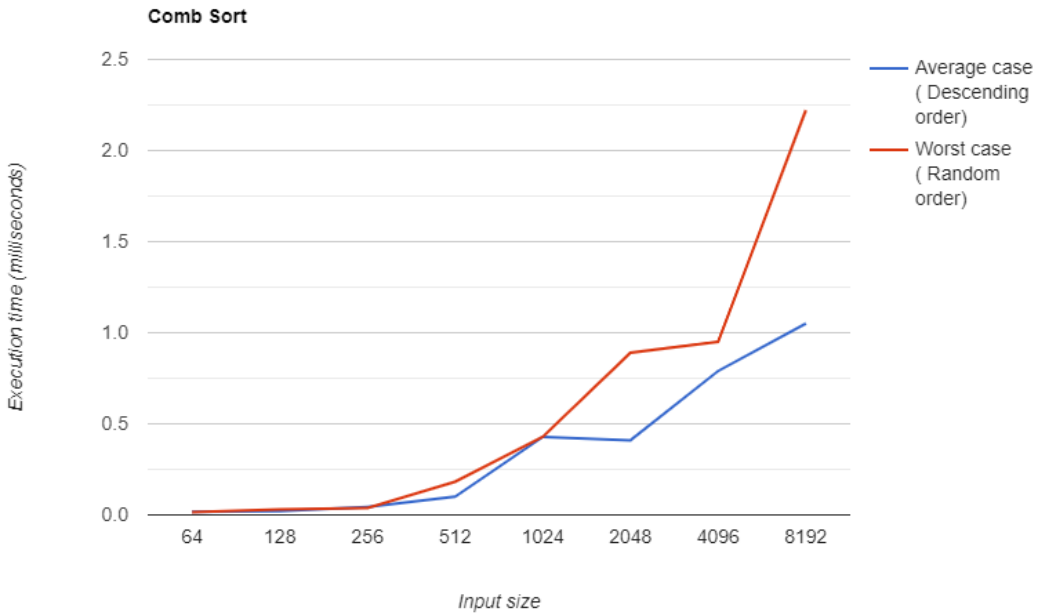
3. Analysis

- **Comb Sort**

Comb sort is an enhanced version of the Bubble Sort. Bubble sort checks adjacent indexes, while combsort checks indexes with a certain gap value. It controls a narrower range by dividing the gap by 1.3 (shrink factor) each time. Thus, the number of inversion counts decreases.

I used random array for worst case over comb sort. Based on the observations, the descending order array gave a better results and allowed us to get better results than worst case on this experiment for a random array on comb sort algorithm.

It works better than bubble sort, but worst case remains O (n^2) in Comb Sort. (proving this is a bit tricky, but has been proved using a method based on Kolmogorov complexity) Average case is the same as worst case. The best case is O (n logn). Comb sort is an in-place algorithm, space complexity is O(1). Also it is not a stable algorithm.



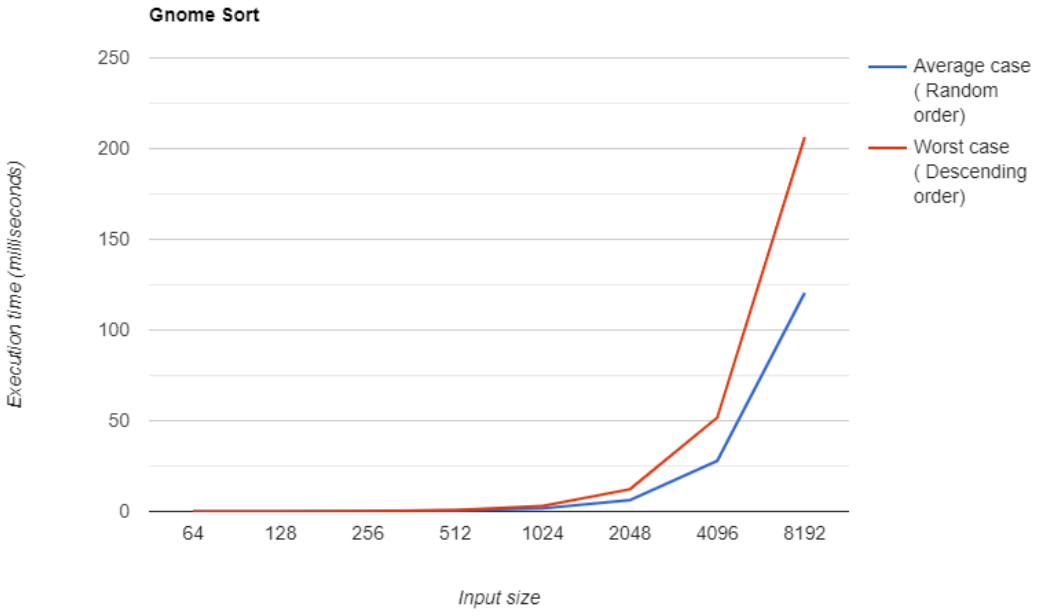
Execution time - input size graph (comb sort)

Comb Sort / Input size	Average Case	Worst Case
64	0,017	0,015
128	0,02	0,03
256	0,044	0,0386
512	0,1005	0,1818
1024	0,4285	0,4285
2048	0,41	0,89
4096	0,79	0,95
8192	1,05	2,22

- **Gnome Sort**

The gnome sort algorithm works based on the theme of garden gnome arranging flower pots. Each time it looks at adjacent indexes and swaps it, it goes back one step. If not, an index goes forward until the pots are finished.

Since there are no nested loops, the time complexity is $O(n^2)$, although it looks like O(n). Because index does not always increase, sometimes it may decrease as in the explanation. Best case is O(n). Since gnome sort is an in-place algorithm, space complexity is O(1). This is a stable algorithm.



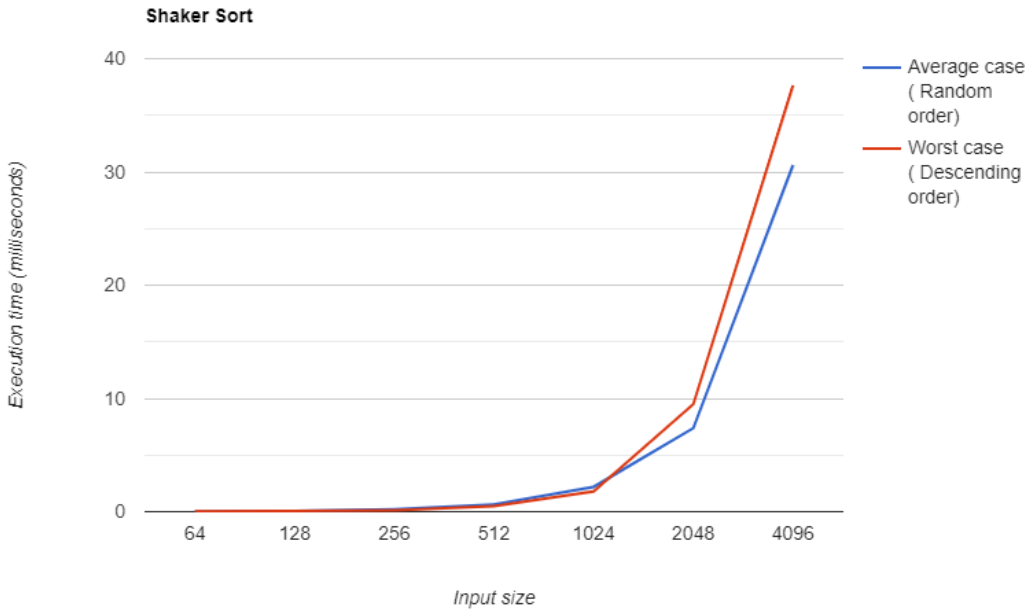
Execution time - input size graph (gnome sort)

Gnome Sort / Input size	Average Case	Worst Case
64	0,03	0,01
128	0,04	0,05
256	0,12	0,24
512	0,42	0,92
1024	1,71	3,09
2048	6,12	12,21
4096	27,84	51,68
8192	120,44	206,21

- **Shaker Sort**

Shaker(cocktail) sorting (bi-directional bubble sort), unlike bubble sorting, sorts the array in both directions. So that, each iteration of the algorithm consists of two stages. In the first, the smallest bubble rises to the end of the array, and in the second stage, the biggest bubble lands at the beginning of the array. Although the time complexities are the same, the shaker sort works better than bubble sort. Shaker sort works faster, a little less than 2 times on average.

Worst and average case time complexity of shaker sort is $O(n^2)$. Best case occurs when the array is already sorted. This is an in-place algorithm so that space complexity is O(1). Also shaker sort is a stable sorting algorithm.



Execution time - input size graph (shaker sort)

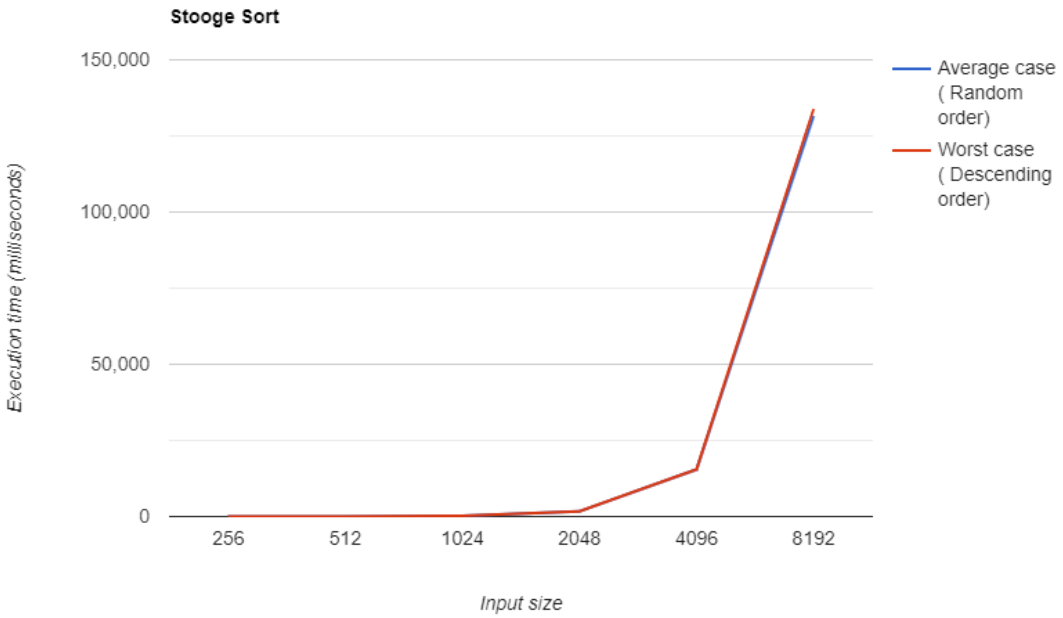
Shaker Sort / Input size	Average Case	Worst Case
64	0,01	0,03
128	0,06	0,05
256	0,20	0,13
512	0,64	0,49
1024	2,19	1,78
2048	7,39	9,48
4096	30,59	37,64
8192	145,35	149,96

- **Stooge Sort**

The stooge sort algorithm works recursively. Named for the Three Stooges, where Moe would repeatedly slap the other two stooges, much like stooge sort repeatedly sorts 2/3 of the array multiple times.

It swaps if the first and last elements of the array are not sorted. If there are more than 2 elements in the array, we apply the same operation to the first 2/3 part of the array, then the last 2/3 part is the same.

This algorithm differs from others for its exceptionally bad time complexity of $O(n^{\log 3 / \log 1.5}) = O(n^{2.7095} ...)$ **This value is valid for all 3 cases**. Worst case space complexity is O(n) because of recursions. Stooge sort is not a stable algorithm.



Execution time - input size graph (stooge sort)

Stooge Sort / Input size	Average Case	Worst Case
64	0,30	0,29
128	0,79	0,74
256	6	6
512	52,64	52,99
1024	204	204
2048	1663,84	1675,48
4096	15478,78	15446,50
8192	131461,67	133838,25

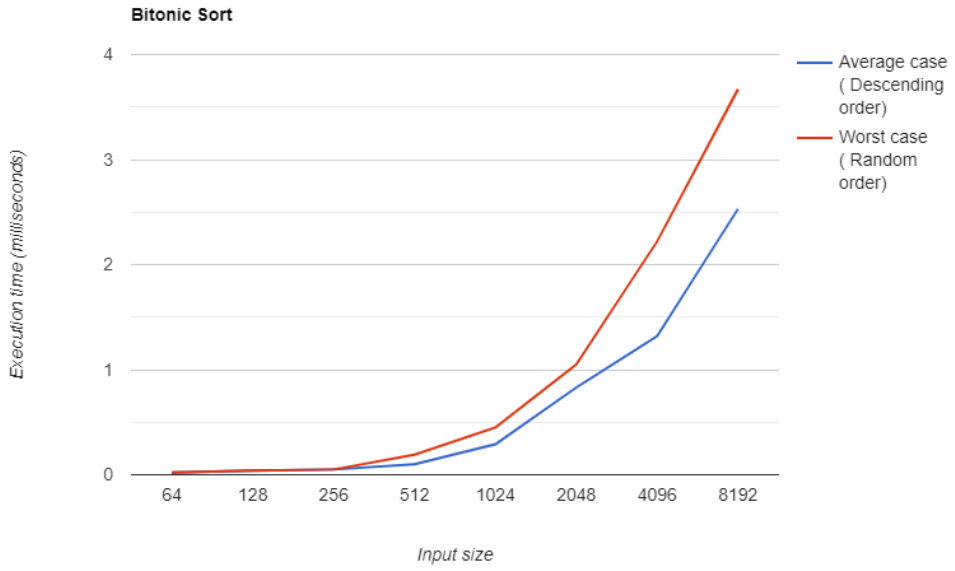
- **Bitonic Sort**

A sequence is a bitonic series if it initially increases up to a certain index and then decreases. Bitonic sort first converts an array to a bitonic series. Thus the first half is in ascending order while the second half is in descending order. It compares first element of first half with first element of second half, then second element of first half with second element of second and so on. It exchanges elements if an element of first half is smaller. This process continues until the length of each half-array is 1. Since all these bitonic sequence are sorted and every bitonic sequence has one element, we get the sorted array.

A logn step is required to obtain two n / 2-dimensional arrays from an array of n dimensions. We can use the equation T (n) = log (n) + T (n / 2) to sort all arrays. Solving this reccurence equation we get:

$$T(n) = \log(n) + \log(n) - 1 + \log(n) - 2 + \ldots + 1 = \log(n)(\log(n) + 1) / 2 .$$

Since each step contains n / 2 comparisons, we get $(n\log^2 n)$ comparisons in total. All cases in parallel time take $O(\log^2(n))$ time. Space complexity is $O(n\log^2(n))$ in non parallel time. It should be noted that for the bitonic sort to work, the array size must be a multiple of 2 exponentials. Bitonic sort is not stable.



Execution time - input size graph (bitonic sort)

Bitonic Sort / Input size	Average Case	Worst Case
64	0,02	0,02
128	0,04	0,04
256	0,05	0,05
512	0,10	0,19
1024	0,29	0,45
2048	0,83	1,05
4096	1,32	2,22
8192	2,53	3,67

4. Tables

- **Worst Cases**

Input size	Comb Sort	Gnome Sort	Shaker Sort	Stooge Sort	Bitonic Sort
64	0,015	0,01	0,03	0,29	0,02
128	0,03	0,05	0,05	0,74	0,04
256	0,0386	0,24	0,13	6	0,05
512	0,1818	0,92	0,49	52,99	0,19
1024	0,4285	3,09	1,78	204	0,45
2048	0,89	12,21	9,48	1675,48	1,05
4096	0,95	51,68	37,64	15446,50	2,22
8192	2,22	206,21	149,96	133838,25	3,67

- **Average Cases**

Input size	Comb Sort	Gnome Sort	Shaker Sort	Stooge Sort	Bitonic Sort
64	0,017	0,03	0,01	0,30	0,02
128	0,02	0,04	0,06	0,79	0,04
256	0,044	0,12	0,20	6	0,05
512	0,1005	0,42	0,64	52,64	0,10
1024	0,4285	1,71	2,19	204	0,29
2048	0,41	6,12	7,39	1663,84	0,83
4096	0,79	27,84	30,59	14629,00	1,32
8192	1,05	120,44	145,35	131461,67	2,53

5. Stabilities

Advantage of stability is when we want to sort by multiple attributes. For example, to sort an array of forest objects with number of trees / forest name, you might sort first by the number of trees, and then by the forest name. If the sort was not stable, then we would lose the benefit of the first sort.

First of all, I manually created an unsorted array. Later, I saved the sorted places of forest objects in this array by noting their second index (forest name) in a string array. After sorting operation, I compared the values in this string array with the second attributes of the forest objects in the sorted array to check whether they are stable or not. **stabilityCheck** function prints:

```
***--- Sort algorithms and stabilities ---***
Comb sort: is NOT stable.
Gnome sort: is stable.
Shake sort: is stable.
Stooge sort: is NOT stable.
Bitonic sort: is NOT stable.
```

6. Conclusion

If we have enough memory we can use bitonic sort. Because it is the fastest running algorithm, but space complexity is inefficient. The most effective algorithm is gnome sort because it is stable, fast, and space complexity's the best. The worst algorithm is definetely stooge sort.

Algorithm	Worst Case	Average Case	Worst Case (Space)	Stability
Comb Sort	$O(n^2)$	$O(n^2)$	O(1)	NO
Gnome Sort	$O(n^2)$	$O(n^2)$	O(1)	YES
Shaker Sort	$O(n^2)$	$O(n^2)$	O(1)	YES
Stooge Sort	$O(n^3)$	$O(n^3)$	O(n)	NO
Bitonic Sort	$O(\log^2(n))$	$O(\log^2(n))$	$O(n\log^2(n))$	NO