BBM204: Software Laboratory 1
Assignment 1 Report

Tuana Cetinkaya, 21985164

March 24, 2021

1 Approach Analysis

1.1 Problem and Idea Behind this Assignment

Main idea behind this experiment is to analyze the sorting algorithms. That's useful for choosing the proper efficient algorithm for possible problems. Some problems require stability over execution time where others might require time efficiency. In this assignment we compare five sorting algorithms with respect to their scalability among the increase in the execution time by the increase in input size.

1.2 Experiment Design

1.2.1 Algorithm Hierarchy

In my assignment all sorting algorithms extends from the abstract class SortingAlgorithm that implements Comparable in order to sort objects. SortingAlgorithm class defines a sort() method and all other sorting algorithms implements that.

That hierarchy is useful for not implementing the same swap method or attributes for each sorting class.

1.2.2 Case Testing

CaseTester class was build to test best, average and worst cases. It generates a random TestRat array by using RandomFactory's randomize method. TestRat's has 2 properties which is an ID and an experimentNumber.
Case tester basically has 3 functionalities:

1. Best Case Tester : This generates an array of sorted order
2. Average Case Tester : This generates an array of random order
3. Worst Case Tester : This generates an array of reversed sorted order

Not all algorithms has the same worst case therefore Comb Sort is using a randomly generated array for simplicity. Then the CaseTester uses TimeDecorator to time and record each sorting algorithm and prints a record of each result in nanoseconds.

1.2.3 Stability Testing

Java's Collections.sort() is a stable sorting algorithm since it's using Merge Sort. I used that fact to my advantage and compared my sorted list results with Collections.sort() result. Console outputs the instabilities. Initial method decides the array size to test stability. For simplicity I recorded size 8 arrays in this experiment to make it easier to read. However increasing the number of the array provides a clearer answer if that sort is truly stable since the unstable algorithms might coincidentally provide stable outcomes. For that purpose I run the experiment 10 times with different sized arrays and eliminate the ones I got unstable results at least one times.

2 Algorithm Stability Analysis

2.1 What is advantages and disadvantages of stability?

Stability is defined by keeping the order of the equal elements as they existed in the array in the first place. Stability provides a stable output in cases of multiple key sorting situations. When you sort an array by the second key then sort again with the first, it guaranties you the first key's equal elements will be sorted by their second key.
On the other hand a stable sorting algorithm requires way more comparisons than an unstable one. So if we only need to sort the list by one key and we does not care the order of the other properties we should choose an unstable one to gain time efficiency.

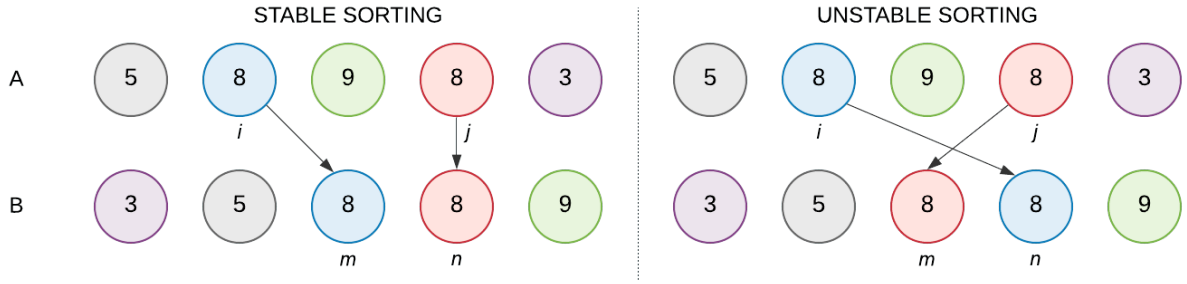

Figure 1: Comparison of Stable and Unstable Algorithm Behaviors

2.2 Stability Analysis

COMB SORT [Figure 2] : is NOT Stable
GNOME SORT [Figure 3]: is Stable
SHAKER SORT [Figure 4]: is Stable
STOOGE SORT [Figure 5]: is NOT Stable
BITONIC SORT [Figure 6]: is NOT Stable

This analysis has been repeated ten times with different sized inputs. For most resulted similar as below.

```
================== STABILITY CHECKS ==================
            ========= COMB SORT =========
Sorted Array for That Algorithm is:
-8  : Number of Experiments for rat ID#2
-6  : Number of Experiments for rat ID#4
-5  : Number of Experiments for rat ID#5  <--- Instability! Stable output had to be: -5 : Number of Experiments for rat ID#2
-5  : Number of Experiments for rat ID#2  <--- Instability! Stable output had to be: -5 : Number of Experiments for rat ID#5
0   : Number of Experiments for rat ID#7
3   : Number of Experiments for rat ID#8
4   : Number of Experiments for rat ID#6  <--- Instability! Stable output had to be: 4  : Number of Experiments for rat ID#5
4   : Number of Experiments for rat ID#5  <--- Instability! Stable output had to be: 4  : Number of Experiments for rat ID#6
false
```

Figure 2: Comb Sort is an unstable algorithm since it contains instabilities from the initial array.

```
            ========= GNOME SORT =========
Sorted Array for That Algorithm is:
-8  : Number of Experiments for rat ID#2
-6  : Number of Experiments for rat ID#4
-5  : Number of Experiments for rat ID#2
-5  : Number of Experiments for rat ID#5
0   : Number of Experiments for rat ID#7
3   : Number of Experiments for rat ID#8
4   : Number of Experiments for rat ID#5
4   : Number of Experiments for rat ID#6
true
```

Figure 3: Gnome Sort is a stable algorithm.

```
          ========== SHAKER SORT ==========
Sorted Array for That Algorithm is:
-8   : Number of Experiments for rat ID#2
-6   : Number of Experiments for rat ID#4
-5   : Number of Experiments for rat ID#2
-5   : Number of Experiments for rat ID#5
0    : Number of Experiments for rat ID#7
3    : Number of Experiments for rat ID#8
4    : Number of Experiments for rat ID#5
4    : Number of Experiments for rat ID#6
true
```

Figure 4: Shaker Sort is a stable algorithm.

```
          ========== STOOGE SORT ==========
Sorted Array for That Algorithm is:
-8  : Number of Experiments for rat ID#2
-6  : Number of Experiments for rat ID#4
-5  : Number of Experiments for rat ID#2
-5  : Number of Experiments for rat ID#5
0   : Number of Experiments for rat ID#7
3   : Number of Experiments for rat ID#8
4   : Number of Experiments for rat ID#6  <--- Instability! Stable output had to be: 4  : Number of Experiments for rat ID#5
4   : Number of Experiments for rat ID#5  <--- Instability! Stable output had to be: 4  : Number of Experiments for rat ID#6
false
```

Figure 5: Stooge Sort is an unstable algorithm since it contains instabilities from the initial array.

```
          ========== BITONIC SORT ==========
Sorted Array for That Algorithm is:
-8  : Number of Experiments for rat ID#2
-6  : Number of Experiments for rat ID#4
-5  : Number of Experiments for rat ID#5  <--- Instability! Stable output had to be: -5 : Number of Experiments for rat ID#2
-5  : Number of Experiments for rat ID#2  <--- Instability! Stable output had to be: -5 : Number of Experiments for rat ID#5
0   : Number of Experiments for rat ID#7
3   : Number of Experiments for rat ID#8
4   : Number of Experiments for rat ID#6  <--- Instability! Stable output had to be: 4  : Number of Experiments for rat ID#5
4   : Number of Experiments for rat ID#5  <--- Instability! Stable output had to be: 4  : Number of Experiments for rat ID#6
false
```

Figure 6: Bitonic Sort is an unstable algorithm since it contains instabilities from the initial array.

3 Complexity and Algorithm Analysis

3.1 Comb Sort

3.1.1 Algorithm Analysis

Comb sort implements the following procedure:

1. Sets a gap, in our implementation it's initially set to array length
2. It compares each element with a distance of that gap and swaps if needed
3. Repeats the procedure and shrinks the gap until the gap is equal to 1
4. When it traverse the list again with a gap of 1 the array is sorted.

Comb sort is an improvement over Bubble sort where the gap is a constant 1.

3.1.2 Time Complexity Analysis

Comb sort is shrinking with a factor of 1.3 by a best tested shrink factor found by the authors Stephen Lacey and Richard Box by testing Comb sort on over 200,000 random lists.
(Source: https://en.wikipedia.org/wiki/Comb_sort)

So in best case, where the inputs are already sorted, each gap runs once. By the harmonic sequence:

$a_n = n * 1/1.3 \quad a_(n-1) = n * 1/(1.3)^2 \quad ... \quad a_(n-i) = n * 1/(1.3)^i$
we get: $O(n * log(n))$

Since comb sort's using an approach to check the list from both ends, feeding a reversed sorted array will result with a better performance than it's random feeded run. Thus, for this algorithm both average and worst cases are using randomly created arrays.

The average and worst case runtime of Comb sort (or Dobosiewicz sort) are $O(n^2)$. It has been proved using a method based on Kolmogorov complexity (e.g. Survey by Vitanyi, page 16)

BEST CASE

$$O(n * log(n))$$

AVERAGE and WORST CASES

$$O(n^2)$$

//(Source: https://hurna.io/assets/files/algorithms/sort/sorting.pdf)

3.1.3 Space Complexity Analysis

Comb sort uses no excessive space other than some constants which makes it space

$$O(1)$$

3.1.4 Graphs and Tables

As expected from the complexity, when the input size doubled for best case [by the data calculation in Fig.7] result is a little higher than doubled, which correlates with the best case of $O(n * log(n))$
And similarly the exponential trend line proves the average time complexity O(n)

	COMB SORT		
INPUT SIZE	BEST CASE	AVERAGE CASE	WORST CASE
16	223713	217766	308476
32	19431	21519	29313
64	14587	23164	20699
128	33969	60722	68297
256	72383	94623	129077
512	120684	98193	284079
1024	93032	176143	313862
2048	167468	398659	419999
4096	359594	827007	867234
8192	945838	1885573	2051302

Figure 7: Comb Sort Data Table

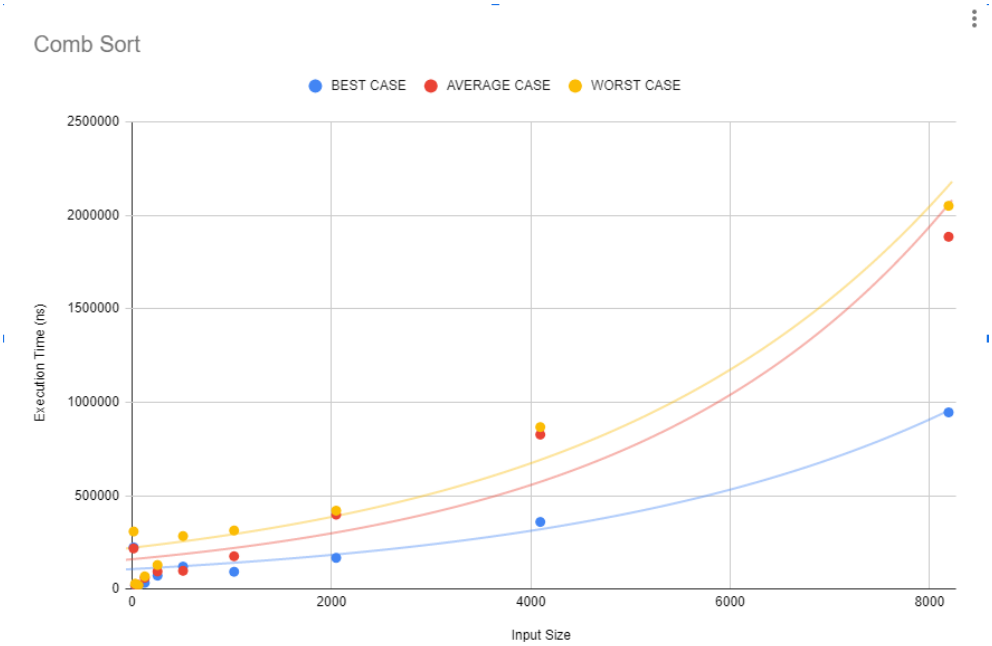

Figure 8: Comb Sort Execution time with respect to Input Size

3.2 Gnome Sort

3.2.1 Algorithm Analysis

Gnome sort is a different implementation of Insertion sort, it do not make a drastic change among the elements thus it's stable by nature. Gnome sort implements the following approach:

1. It compares the adjacent elements from the beginning of the list starting the second element.
2. It keeps moving right as long as it finds properly placed pairs
3. When it found an unsorted pair it swaps and moves left.

Gnome Sort behaves like insertion sort only when it's moving an element into its proper index. Unlike insertion sort, gnome sort makes comparisons even after placing an element in its proper spot. If it encounters a value that is out of place, it behaves like insertion sort again. Moving backwards instead of forwards provide some efficiency since it does not necessarily traverse through the whole list as much as Insertion sort does.

3.2.2 Time Complexity Analysis

In best case, where the list is initially sorted, gnome sort traverse the list only once. Which makes it's best case complexity $O(n)$

In average and worst cases of size n Gnome Sort has the complexity $O(n^2)$ since an element at the beginning of the list is expected to be an average of $n/2$ distance away from its sorted location. An item in the middle of the list is expected to be an average of $n/4$ distance away from its sorted location. With the total of n elements, number of traversals needs to be minimum $n * (n/4)$ with a time complexity of $O(n^2)$. Which is the reason it's almost adjacent to the x-axis when put in the same graph along with it's average and worst case execution times.

Especially in the worst case, where the elements are reversed sorted, Gnome sort has to traverse the whole array from index to beginning for each element. At that matter, it behaves exactly like the insertion sort, with again a complexity of $O(n^2)$.

BEST CASE:

$$O(n)$$

WORST AND AVERAGE CASES

$$O(n^2)$$

3.2.3 Space Complexity Analysis

Since Gnome Sort requires no other space than an index, we can clearly say it has the space complexity of

$$O(1)$$

3.2.4 Graphs and Tables

	GNOME SORT		
INPUT SIZE	BEST CASE	AVERAGE CASE	WORST CASE
16	68292	49772	51631
32	13695	25518	23444
64	8821	48633	89856
128	11017	270313	216530
256	15013	323975	519226
512	14945	818022	1140312
1024	19249	2920071	5477762
2048	20548	11810540	23296761
4096	33820	47672390	95880909
8192	73520	190250085	407164103

Figure 9: Gnome Sort Data Table

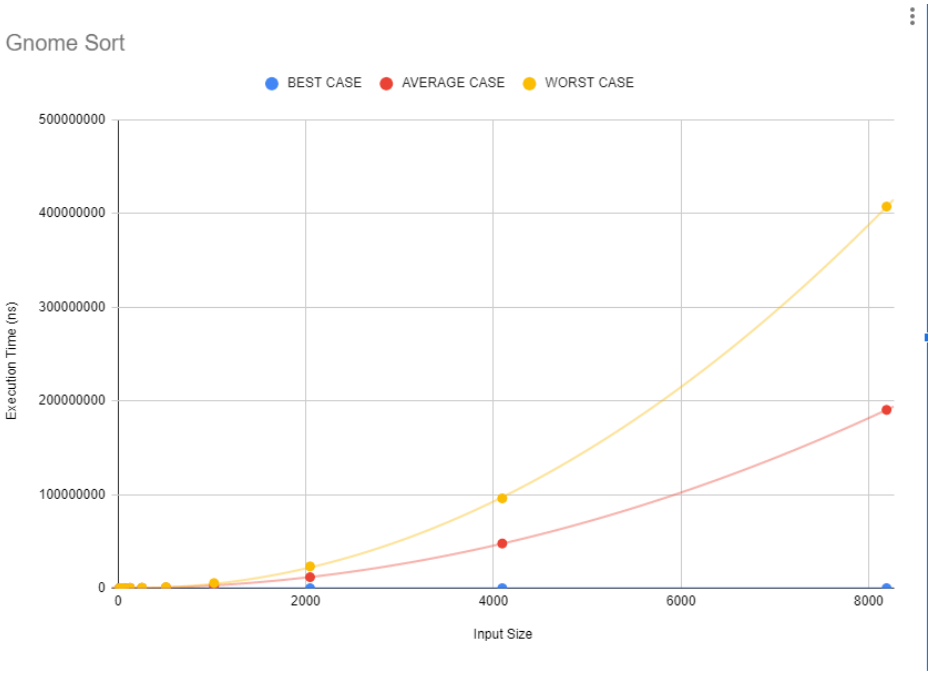

Figure 10: Comb Sort Execution time with respect to Input Size

3.3 Shaker Sort

3.3.1 Algorithm Analysis

Shaker sort is a variation of the Bubble Sort. Unlike Bubble Sort, Shaker Sort traverse the array in both directions. While carrying the biggest element to the end, it also carries the smallest element to the beginning. Basic steps are as below:

1. Starting from the first index, compare and carry the biggest element to the last, which fixes the position of the largest element to the end of the array.
2. On the way back check and carry the smallest element back to beginning. That fixes the smallest element to beginning.
3. Rinse and repeat when there's a swap was made in the last run.

In shaker sort that operation ends up in the middle of the list when bubble sort ends up at the beginning.

3.3.2 Time Complexity Analysis

In best case where everything is sorted, Shaker Sort traverse the list twice to see there's nothing to swap. Then halts the procedure with $2n$ array traversals.

In worst case where the items are reversed sorted, Shaker Sort needs to carry each element all the way to the other side with an n comparisons of n times that results in n^2 operations. Similar case also applies to average random lists.

Where the average and worst cases are similar to Bubble Sort, Shaker Sort is less than two times faster than bubble sort.

BEST CASE COMPLEXITY:

$$O(n)$$

WORST AND AVERAGE CASE COMPLEXITIES:

$$O(n^2)$$

3.3.3 Space Complexity Analysis

Similarly Shaker sort only uses an index to traverse through the list. So the space complexity is a constant

$$O(1)$$

3.3.4 Graphs and Tables

	SHAKER SORT		
INPUT SIZE	BEST CASE	AVERAGE CASE	WORST CASE
16	42915	80595	77677
32	11957	31327	23075
64	8835	74973	78605
128	11926	307309	220220
256	14193	415760	518751
512	13873	997760	1072429
1024	16632	3501764	5819545
2048	18767	13683388	23741285
4096	31173	56359764	98254723
8192	67524	241469737	417111308

Figure 11: Shaker Sort Data Table

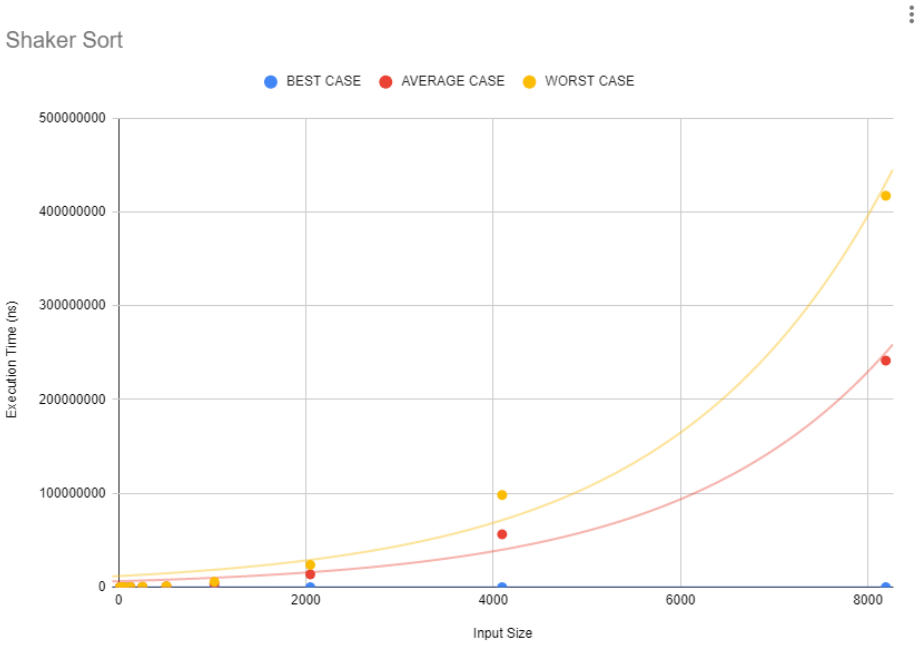

Figure 12: Shaker Sort Execution time with respect to Input Size

3.4 Stooge Sort

3.4.1 Algorithm Analysis

Stooge sort sorts the array in groups of the 2/3's of the size as below:

1. It swaps the last and the first elements if they're out of order in a given interval,
2. Sorts the array's first 2/3
3. Sorts the last 2/3
4. Then sorts the first 2/3 again
5. Algorithm repeats the first four steps to sort each part recursively until the split size of the parts less than 2.

Main reason this approach works is to collect all great elements of the first 2/3 in the middle where they can reach to the last 2/3. When the array is sorted for the second time with the last 2/3 all the big elements are placed at the end of the array and away from the first part. That second sort also places the small elements within the reach of the first part. So when the array is sorted for the last time with first 2/3, all the elements are put in place.

3.4.2 Time Complexity Analysis

Each time Stooge Sort splits the array in two parts another level is being added to the recursive tree. Since each branch is sorting an array that's 2/3 size of the previous call the running time complexity of stooge sort can be written as,

$$T(n) = 3T(3n/2)+?(1)$$

Solution of above recurrence is

$$O(n * (log3/log1.5)) = O(n * 2.709)$$

hence it is slower than even bubble sort(n^2).

BEST, AVERAGE AND WORST TIME COMPLEXITIES:

$$O(n)^{(log(3)/log(1.5))}$$

(Source: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. pp. 161–162. ISBN 0-262-03293-7.)

3.4.3 Space Complexity Analysis

Since Stooge sort is a recursive sort, each call allocates some space in the memory. With a maximum of n calls, we can say the worst case space complexity is:

$$O(n)$$

3.4.4 Graphs and Tables

In this experiment I used input sizes of 2^n because of the limitations due to Bitonic sort. Due to array splits of stooge sort, there should be some breaking points in the graph that we could not observe on a dataset with a great gap such as ours. Regular graph (Fig. 13) for Stooge Sort is also given below along with our vaguely distributed dataset.

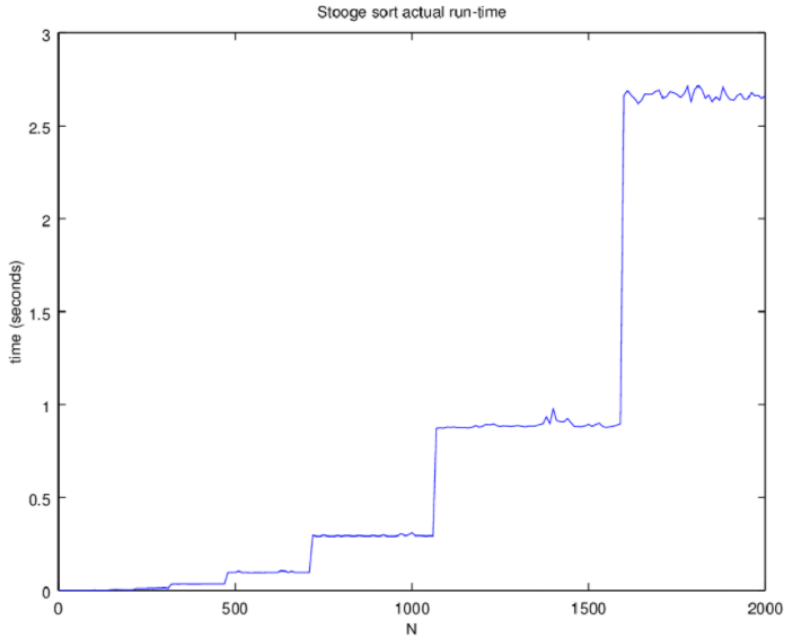

Figure 13: Stooge Sort Data Table with More Inputs to Observe Breaking Points

	STOOGE SORT		
INPUT SIZE	BEST CASE	AVERAGE CASE	WORST CASE
16	41087	524020	483678
32	5376	192080	174037
64	35023	372080	439587
128	165767	1518180	1209589
256	1057480	15945300	14342917
512	7548124	118370920	111511967
1024	22449734	278891690	269658288
2048	197193277	2404445200	2303673718
4096	1794363984	22061055780	20659710006
8192	18806106360	188938866150	186510236960

Figure 14: Stooge Sort Data Table

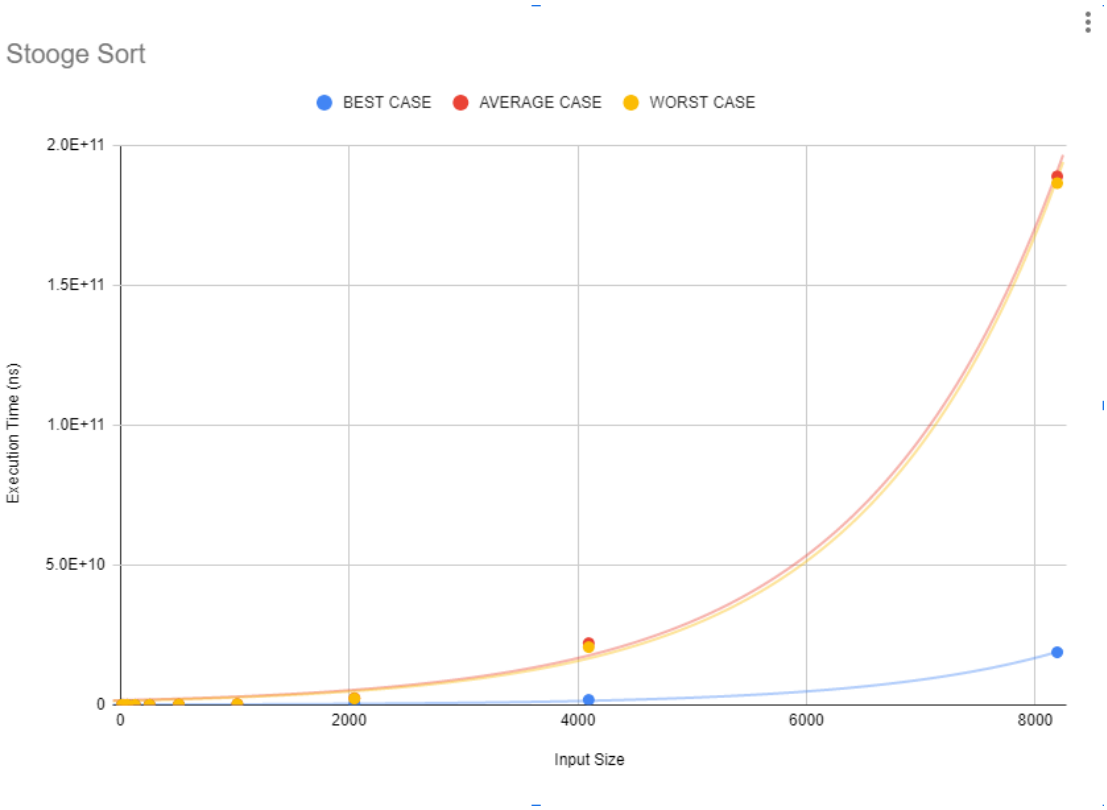

Figure 15: Stooge Sort Execution time with respect to Input Size

3.5 Bitonic Sort

3.5.1 Algorithm Analysis

Bitonic is a recursive sorting algorithm that only sorts the arrays of sized 2^n that's why this tests was made with array sizes of that format. Bitonic sort functions as below:

1. Form the array into a bitonic sequence.
   (This step is skipped for simplicity since I gave all inputs in bitonic sizes)
2. Sorts the first half in ascending matter
3. Sorts the second half in descending matter
4. Compare two arrays with each other from the beginning to the end and swap if the first array has a bigger element than the second.
5. That swap operation provides two $n/2$ length arrays that each element of the first half is smaller than each element in the second half .
6. When this process repeats again we get $n/4$ sized 4 arrays and we repeat until we have 1 sized n arrays. At that point since we have 1 elements in each list, all bitonic sequences are internally sorted thus the whole array is sorted.

This algorithm can run in parallel to reduce time complexity.

3.5.2 Time Complexity Analysis

In each comparison we split the array into two pieces until we get n lists with 1 elements. That means this operation runs $log(n)$ times. then we have to repeat this for each time we split the list, so complexity of $BitonicSort(n/2)$ adds onto the previous operation. We can represent a recursive formula as

$$T(n) = log(n) + T(n/2)$$

Solution of this recursive equation for one run is

$$T(n) = log(n)(log(n) + 1)/2$$

In that case the overall complexity of of Bitonic Sort is independent from the initial array placement and same for BEST, AVERAGE and WORST CASES:

$$O(log^2(n))$$

3.5.3 Space Complexity Analysis

With each recursive call, Bitonic Sort allocates another array with a size of $n/2$ and since this process is repeated for $log^2(n)$ times our space complexity becomes

$$O(n * log^2(n))$$

3.5.4 Graphs and Tables

	BITONIC SORT		
INPUT SIZE	BEST CASE	AVERAGE CASE	WORST CASE
16	49497	50130	43728
32	25306	24958	15805
64	20478	34816	27728
128	140172	68921	33998
256	70342	78136	67887
512	93784	145253	97258
1024	188528	277667	196009
2048	391990	605008	427810
4096	861277	1272551	904538
8192	2335743	2791158	2061923

Figure 16: Bitonic Sort Data Table

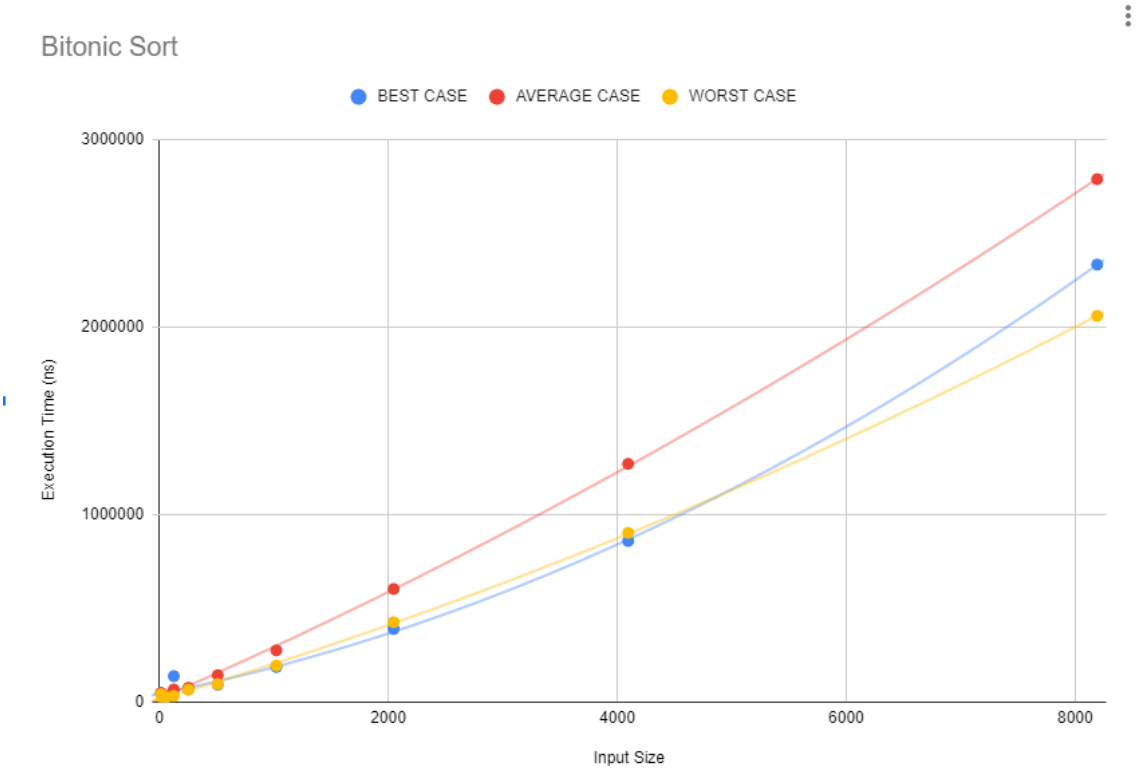

Figure 17: Bitonic Sort Execution time with respect to Input Size

4 Conclusion

If we do not need a stable algorithm, by the result of the percentage distribution of average and worst cases we can compare the algorithms as below

$$Stooge > Gnome >= Shaker > Bitonic > Comb$$

When looking for a stable algorithm we can choose:

$$Gnome > Shaker$$

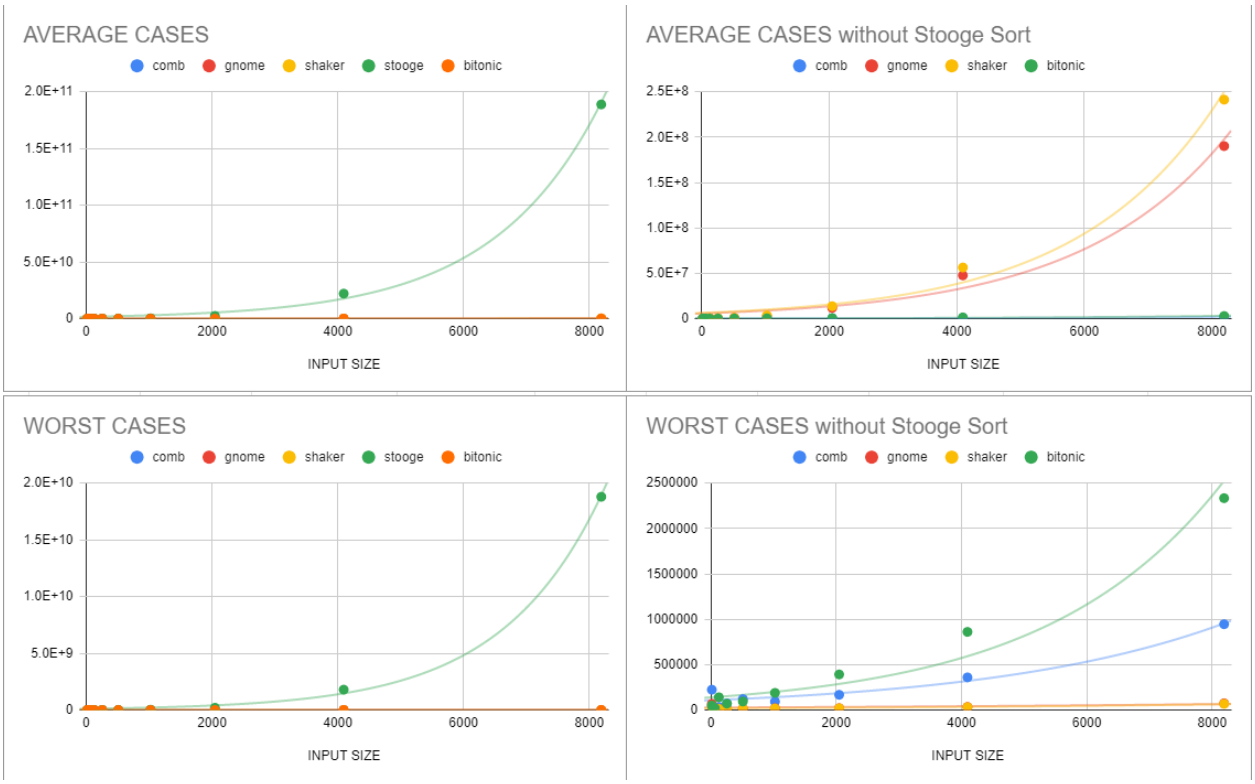

Figure 18: %100 Stacked Column Chart for Execution Times For Average and Worst Cases

4.1 Data Tables

Since the Stooge Sort is overwhelmingly slow and is effecting other algorithms I choose to exclude it from the Fig18 Below, there's a percentage distribution chart to clearly see the Stooge Sort's effect on other algorithm times as the input increases.

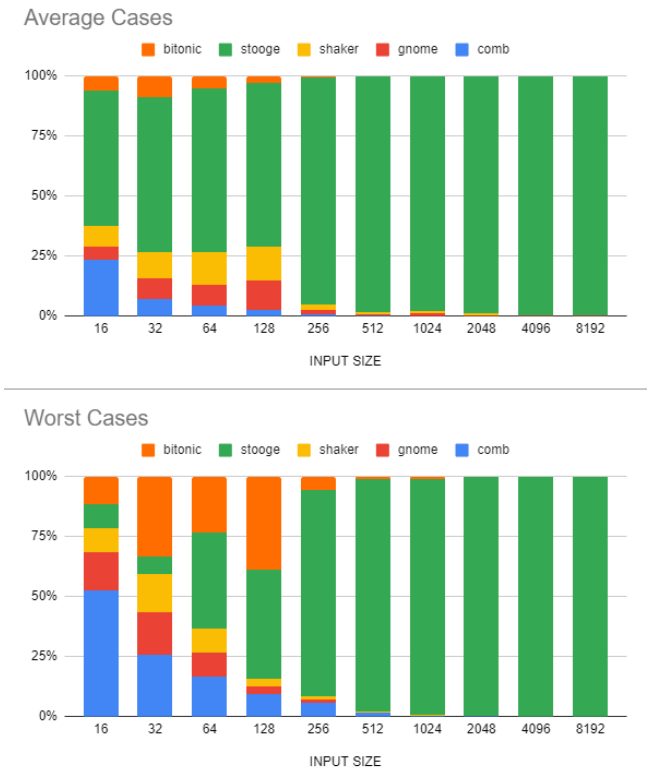

Figure 19: %100 Stacked Column Chart for Execution Times For Average and Worst Cases

BEST CASES					
INPUT SIZE	comb	gnome	shaker	stooge	bitonic
16	223713	68292	42915	41087	49497
32	19431	13695	11957	5376	25306
64	14587	8821	8835	35023	20478
128	33969	11017	11926	165767	140172
256	72383	15013	14193	1057480	70342
512	120684	14945	13873	7548124	93784
1024	93032	19249	16632	22449734	188528
2048	167468	20548	18767	197193277	391990
4096	359594	33820	31173	1794363984	861277
8192	945838	73520	67524	18806106360	2335743

Figure 20: Best Case Data Tables for each Sorting Algorithm

AVERAGE CASES					
INPUT SIZE	comb	gnome	shaker	stooge	bitonic
16	217766	49772	80595	524020	57352
32	21519	25518	31327	192080	25186
64	23164	48633	74973	372080	28967
128	60722	270313	307309	1518180	59785
256	94623	323975	415760	15945300	100792
512	98193	818022	997760	118370920	179371
1024	176143	2920071	3501764	278891690	373521
2048	398659	11810540	13683388	2404445200	608648
4096	827007	47672390	56359764	22061055780	1315564
8192	1885573	190250085	241469737	188938866150	2874592

WORST CASES					
INPUT SIZE	comb	gnome	shaker	stooge	bitonic
16	223713	68292	42915	41087	49497
32	19431	13695	11957	5376	25306
64	14587	8821	8835	35023	20478
128	33969	11017	11926	165767	140172
256	72383	15013	14193	1057480	70342
512	120684	14945	13873	7548124	93784
1024	93032	19249	16632	22449734	188528
2048	167468	20548	18767	197193277	391990
4096	359594	33820	31173	1794363984	861277
8192	945838	73520	67524	18806106360	2335743

Figure 21: Average and Worst Case Data Tables for each Sorting Algorithm