BBM 204: Software Laboratory II Name: Umut Bugrahan Ozkan

Assignment 1 — March 26th, 2020

ID: 21727628 *Section: 1*

1 Problem Definition

Analyzing various sorting algorithms and compare their running times and space requirements with different sized inputs, visualizing the results using tables and plots.

2 Solution Approach

- Implement all algorithms into java code.
- Determine the time required for each basic operation.
- For average case create different random arrays (25 for each size, 25*6=150 total).
- While analyzing worst and best case, create specific arrays for each algorithm separately.
- Run each algorithm with these arrays (25 average, 1 best and 1 worst) using **time** command.
- Convert seconds to milliseconds then calculate **average** execution time.
- Visualize analyses using chart and plots.
- Compare algorithms with their space and time requirements.

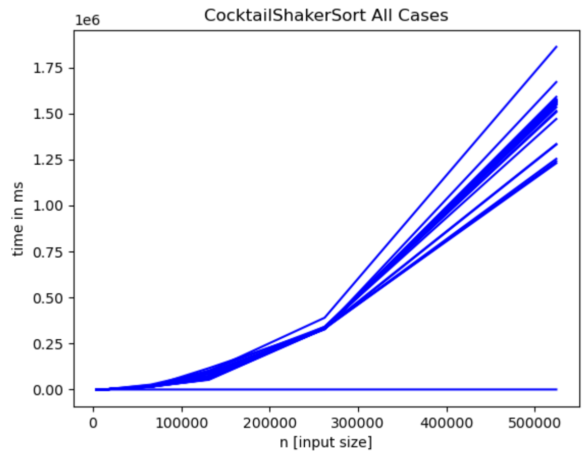
3 Analyses

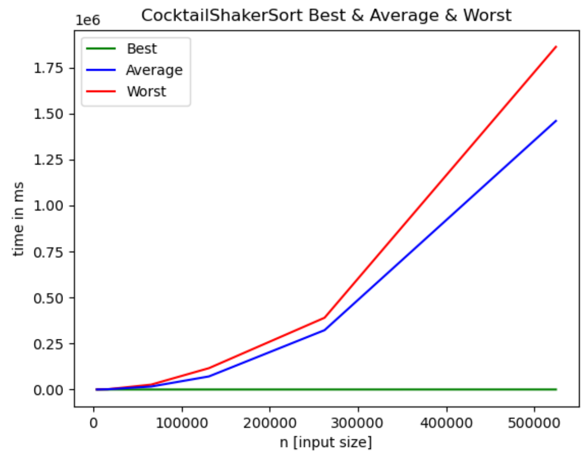
3.1 Cocktail Shaker Sort

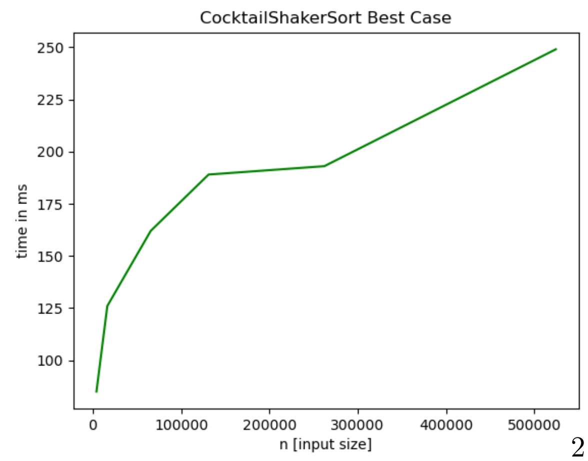
Cocktail Shaker Sort is a comparison based sorting algorithm. Works like the bubble sort algorithm but instead traversing only one way, it traverses the array bidirectional. So theoretically cocktail shaker sort is two times faster than bubble sort algorithm. Cocktail Shaker Sort's **best case** is that the array is already sorted (traverses the array only 1 time and it takes $0(N)$ time complexity) ; **worst case** is that the array is reverse sorted (traverses the array N-1 times and it takes $O(N^2)$ time complexity).

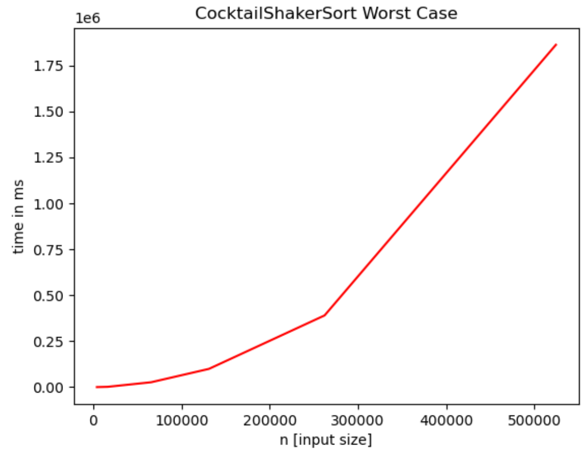
Algorithm 1 Cocktail Shaker Sort

```
procedure COCKTAILSHAKERSORT(A : list of sortable items)
    $swapped \leftarrow true$ ........................................$1$
    while swapped do ....................................$N$
        $swapped \leftarrow false$ ....................................$N$
        for $i$ in 0 to $length(A)$ - 2 do ..............$2N^2 - 2N$
            if $A[i] > A[i+1]$ then ....................$3N^2 - 6N$
                $temp \leftarrow A[i]$ ................................$2N^2 - 4N$
                $A[i] \leftarrow A[i+1]$ ............................$3N^2 - 6N$
                $A[i+1] \leftarrow temp$ ...........................$2N^2 - 4N$
                $swapped \leftarrow true$ .........................$N^2 - 2N$
        if not $swapped$ then ..............................$N$
            $break$
        $swapped \leftarrow false$ .....................................$N$
        for $i$ in $length(A)$ - 2 to 0 do ...............$2N^2 - 2N$
            if $A[i] > A[i+1]$ then ....................$3N^2 - 6N$
                $temp \leftarrow A[i]$ ................................$2N^2 - 4N$
                $A[i] \leftarrow A[i+1]$ ............................$3N^2 - 6N$
                $A[i+1] \leftarrow temp$ ...........................$2N^2 - 4N$
                $swapped \leftarrow true$ .........................$N^2 - 2N$
    ..........................................................................$26N^2 - 44N + 1 -> O(N^2)$
```









As we can see from top right graph, we are approaching $O(N)$ line when the input becomes better, best case is that the array is already sorted. Best case line looks like a flat line because of the scale. You can see best case line at *CocktailShakerSort Best Case* plot.
When the input becomes worse, we are approaching $O(N^2)$ curve and for the average case this curve becomes wider. Worst case of the algorithm is reverse sorted array.
Algorithm takes constant extra space to sort the array.

Table 1: Execution Times[ms] of Cocktail Shaker Sort

Cocktail Shaker Sort/n	*4096*	*16384*	*65536*	*131072*	*262144*	*524288*
Best Execution Time	85	126	162	189	193	249
Average Execution Time	302	1516	16283	71175	322738	1459829
Worst Execution Time	728	1766	26567	116124	390132	1862315

- **Best Time Complexity:** $O(N)$
- **Average Time Complexity:** $O(N^2)$
- **Worst Time Complexity:** $O(N^2)$
- **Extra Space:** $O(1)$

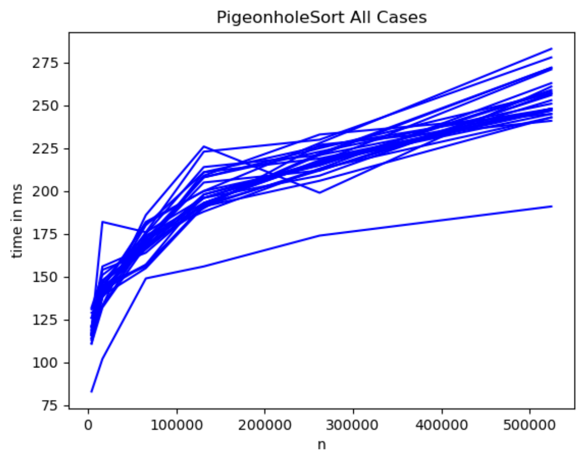
3.2 Pigeonhole Sort

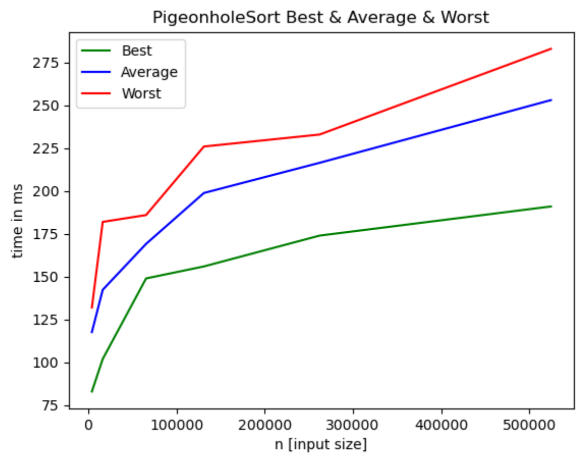
Pigeonhole Sort is an algorithm that do not use comparison while sorting. It has a procedure written in below:

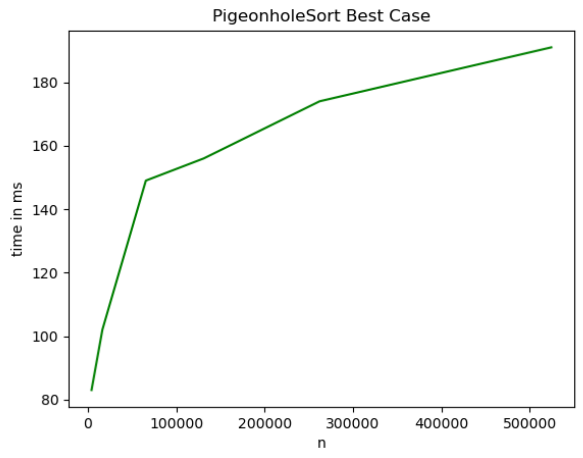
- Make an array of lists, one for each possible key. Each of these lists should be able to hold an arbitrary number of items of the same type as those in the list to be sorted. Initialize each pigeonholes to an empty list.
- Iterate through the list to be sorted; insert each item at the end of the pigeonhole corresponding to its value.
- Copy items in each pigeonhole, in order, back to the original list

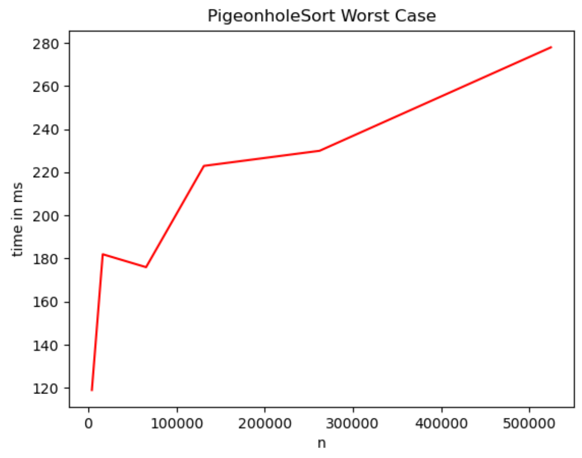
Algorithm 2 Pigeonhole Sort

```
1: procedure PIGEONHOLESORT(A : list of sortable items)
2:     $max \leftarrow getMax(A)$ ........................................$n+1$
3:     $min \leftarrow getMin(A)$ ........................................$n+1$
4:     $size \leftarrow max - min + 1$ .................................$1$
5:     $holes \leftarrow [0] * size$ .......................................$N+1$
6:     for $x$ in $A$ do ................................................$n$
7:         $holes[x - min]{+}{+}$ .....................................$2n$
8:     $i \leftarrow 0$ ...........................................................$1$
9:     for $count$ in 0 to $size$ do ................................$N$
10:        while $holes[count] > 0$ do ........................$2N$
11:            $holes[count]--$ ..................................$2N$
12:            $A[i] \leftarrow count + min$ .............................$2N$
13:            $i{+}{+}$ ................................................$N$
       ............................................................$9N + 5n + 5 -> O(N+n)$
```









From top right graph, we are moving away from $O(N)$ line when the input becomes better or worse. If we say average case is approximately *f(x)* = x then best case line looks like *f(x)* = x - c and worst case line looks like *f(x)* = x + c lines.

Best case of this algorithm is that the array is full of the same integers. In best case, algorithm takes **O(n + 1)** time complexity and **O(1)** extra space.
Worst case of the algorithm is that the array is full of 2 different integers (e.g. first one is 0 and the other one is 999999999) and the order is reverse. In this case it takes **O(n)** extra space and **O(N + n)** time complexity where **N** is number of pigeonholes and **n** is size of the array.

Table 2: Execution Times[ms] of Pigeonhole Sort

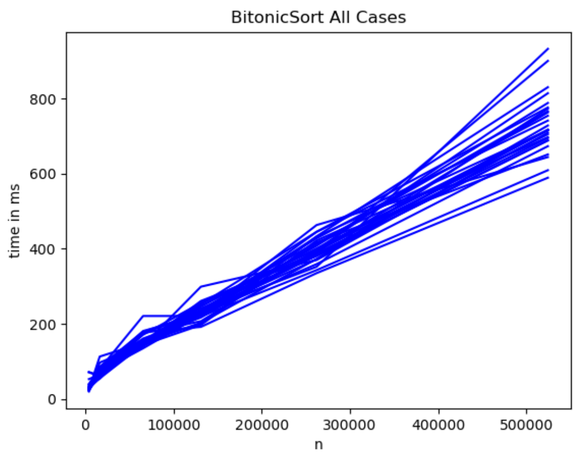
Pigeonhole Sort/*n*	*4096*	*16384*	*65536*	*131072*	*262144*	*524288*
Best Execution Time	83	102	149	156	174	191
Average Execution Time	117	142	169	198	216	253
Worst Execution Time	132	182	186	226	233	283

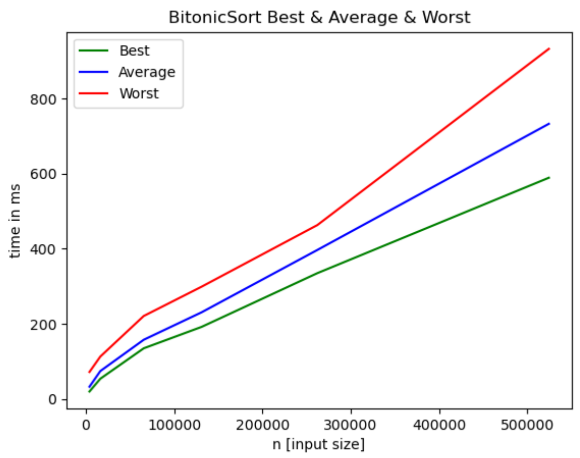
- **Best Time Complexity:** $O(N + n)$ where N is pigeonholes and n is size of the array
- **Average Time Complexity:** $O(N + n)$
- **Worst Time Complexity:** $O(N + n)$
- **Extra Space:** $O(n)$

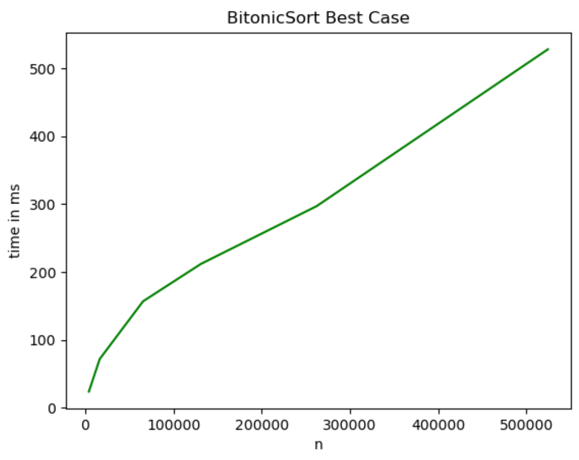
3.3 Bitonic Sort

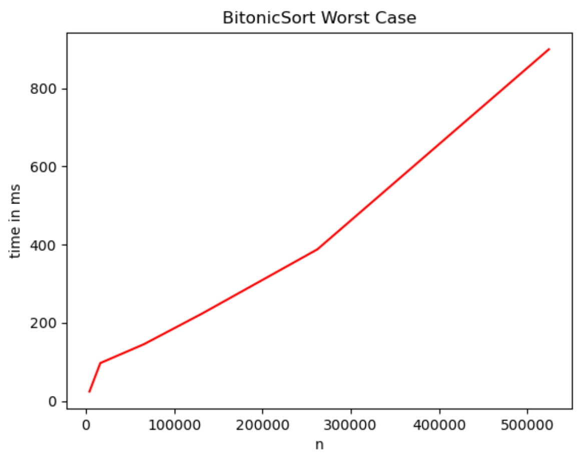
Bitonic sort is a comparison-based sorting algorithm. First, splits the array into two half recursively. In left part, it sorts increasing order and the right part, sorts decreasing order. Then we compare first element of left part with first element of right part, then second element of left part with second element of second and so on. We swap the elements if an element of left part is smaller.

Base case *T(1)* is 0 since we do not need a comparison.
T(n)* = log(n) + *T(n/2)
***T(n)* = log(n) + log(n) - 1 + ... + 1**
***T(n)* = log(n) * (log(n) + 1) / 2**
T(n) $= (log^2(n) + log(n))/2$
->$O(log^2(n))$









Best case of the algorithm is *$O(log^2(n))$ with an array is already sorted.* ***Worst case*** *of the algorithm is $O(log^2(n))$ too. I tried worst case with an array like:*

4096, 4094, ..., 0, 4095, 4093, ..., 3, 1

Left part is reverse sorted array with only even integers and right part is reverse sorted with only odd integers.

From top right graph, we are moving away from *$O(log^2(n))$* curve when the input becomes better or worse. If we say average case is approximately f(x) = $log^2(x) * log(c * x)$ then best case graph looks like f(x) = $log^2(x) * log(m * x)$ and worst case graph looks like f(x) = $log^2(x) * log(k * x)$ graphs where $m < c < k$ and $m > 1$

Table 3: Execution Times[ms] of Bitonic Sort

Bitonic Sort/n	*4096*	*16384*	*65536*	*131072*	*262144*	*524288*
Best Execution Time	22	57	129	194	297	528
Average Execution Time	32	74	157	230	396	732
Worst Execution Time	72	113	221	299	463	932

- **Best Time Complexity:** $O(log^2(N))$
- **Average Time Complexity:** $O(log^2(N))$
- **Worst Time Complexity:** $O(log^2(N))$
- **Space Complexity:** $O(Nlog^2(N))$

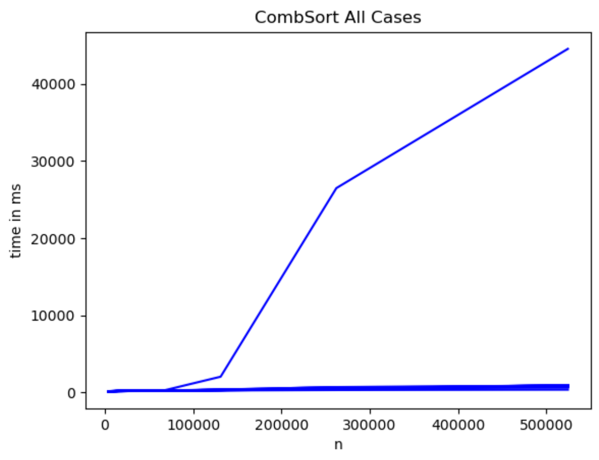
3.4 Comb Sort

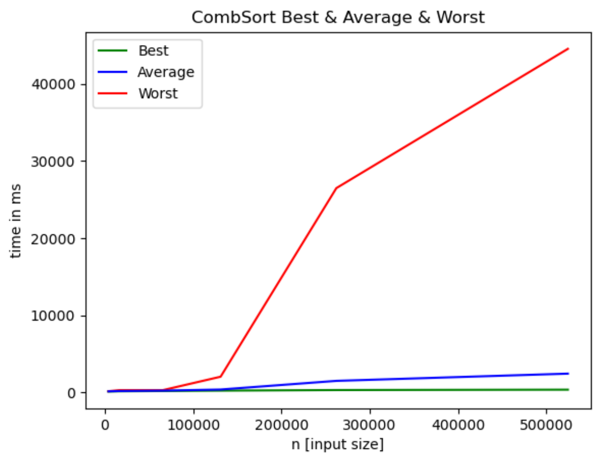
Comb Sort is the advance form of *Bubble Sort* and *Cocktail Shaker Sort*. Bubble and Cocktail Shaker Sort compare all the adjacent values while Comb Sort removes all the small values near the end of the list. Has a procedure written below:

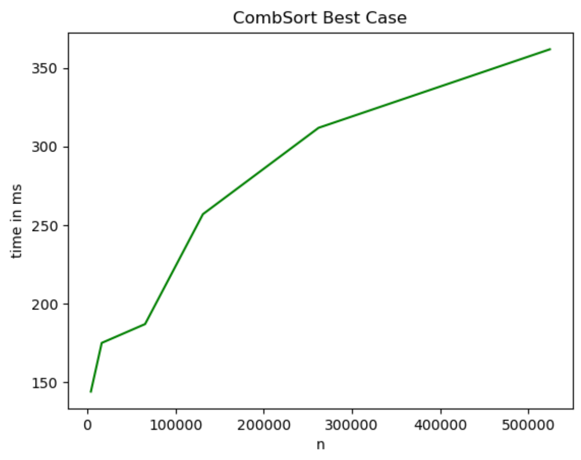
- **1.** Calculate the gap value (calculated as gap/1.3 and initially equal to size of the array). If gap value is 1 then go to fourth step. Otherwise go to second step.
- **2.** Iterate over data set and compare each item with gap item then go to third step.
- **3.** Swap the element if left one is bigger than the right one. Otherwise go to first step.
- **4.** Stop

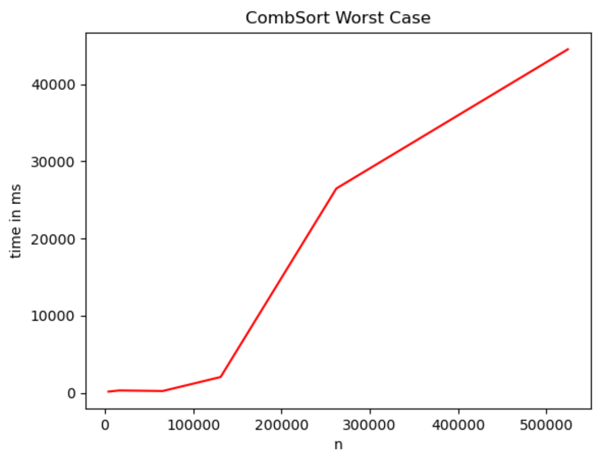
Algorithm 3 Comb Sort

```
procedure COMBSORT(A : list of sortable items)
    $gap \leftarrow length(A)$ ................................2
    $sorted \leftarrow false$ ................................1
    while $!sorted$ do ................................$N$
        $gap \leftarrow gap/1.3$ ................................$2N$
        if $gap == 1$ then ................................$N$
            $sorted \leftarrow true$ ................................$N$
        $i \leftarrow 0$ ................................$N$
        while $i + gap < length(A)$ do ................................$N^2$
            if $A[i] > A[i+gap]$ then ................................$3N^2$
                $swap(A[i], A[i+1])$ ................................$5N^2$
                $sorted \leftarrow false$ ................................$N^2$
            $i{+}{+}$ ................................$N^2$
    ................................$11N^2 + 6N + 3 -> O(N^2)$
```









Best case of the algorithm is that the array is already sorted and best case takes $O(NlogN)$ time complexity. **Worst case** of the algorithm takes $O(N^2)$ time complexity. For the worst case I used an array explained below:

My worst case is an ordered array with the largest values in odd positions. In this sequence it makes maximum number of swaps.

0, 256, 1, 257, 2, 258, ..., 255, 512

Algorithm takes constant extra space to sort the given array. For **average case** I run the algorithm with 25 different input sets and it takes similar to $O(NlogN)$ time complexity.

Table 4: Execution Times[ms] of Comb Sort

Comb Sort/n	*4096*	*16384*	*65536*	*131072*	*262144*	*524288*
Best Execution Time	123	152	187	238	312	362
Average Execution Time	146	186	235	381	1501	2439
Worst Execution Time	161	299	331	2040	26492	44519

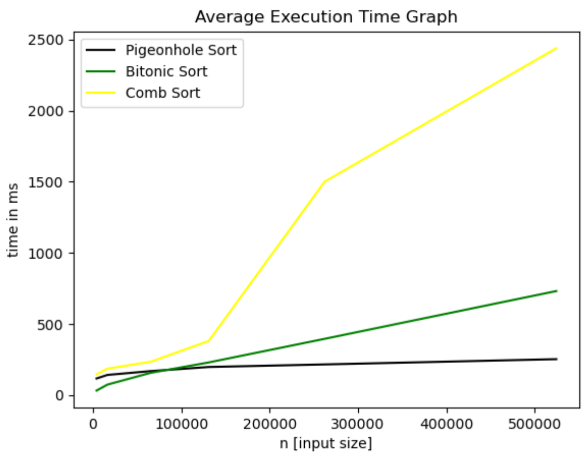
- **Best Time Complexity:** $O(NlogN)$

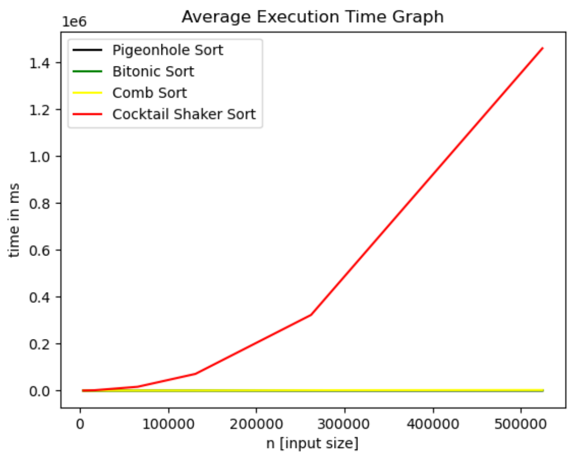
- **Average Time Complexity:** $O(NlogN)$
- **Worst Time Complexity:** $O(N^2)$
- **Extra Space:** $O(1)$

4 Conclusion

Table 5: Average Execution Times [ms] of Algorithms

Algorithms/n	*4096*	*16384*	*65536*	*131072*	*262144*	*524288*
Cocktail Shaker Sort	302	1516	16283	71175	322738	1459829
Pigeonhole Sort	117	142	169	198	216	253
Bitonic Sort	32	74	157	230	396	732
Comb Sort	146	186	235	381	1501	2439

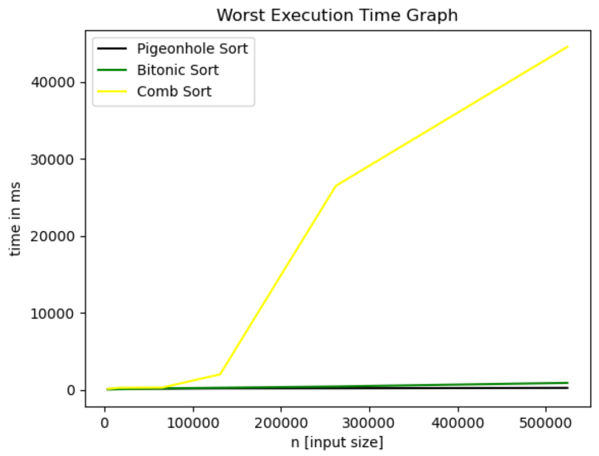


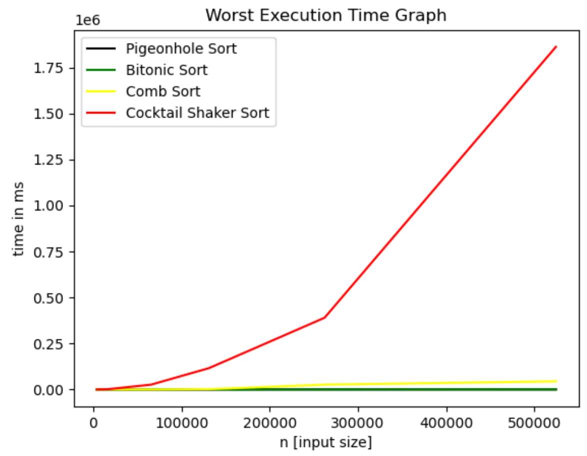


In **Average case** and **worst case** *Cocktail Shaker Sort* performs worst in terms of *time complexity*. Range between 0 to 16384 is acceptable but when the input size becomes larger execution time increases exponentially.
In range of 0 to 131072 *Comb Sort* sorts nearly same as *Bitonic Sort* and *Pigeonhole Sort*, but after n=131072, execution time increased larger than the other two.

Table 6: Worst Execution Times [ms] of Algorithms

Algorithms/n	*4096*	*16384*	*65536*	*131072*	*262144*	*524288*
Cocktail Shaker Sort	728	1766	26567	116124	390132	1862315
Pigeonhole Sort	132	182	186	226	233	283
Bitonic Sort	72	113	221	299	463	932
Comb Sort	161	299	331	2040	26492	44519





Pigeonhole Sort and *Bitonic Sort* behave fine even on **worst case** scenario. *Cocktail Shaker Sort* behaves badly on **worst case** scenario. *Comb Sort* behaves badly on **worst case** scenario too but not worse than *Cocktail Shaker Sort*.

Table 7: Time Complexities of Algorithms

Algorithms/Case	*Best*	*Average*	*Worst*
Cocktail Shaker Sort	$O(N)$	$O(N^2)$	$O(N^2)$
Pigeonhole Sort	$O(N+n)$	$O(N+n)$	$O(N+n)$
Bitonic Sort	$O(log^2(N))$	$O(log^2(N))$	$O(log^2(N))$
Comb Sort	$O(N*log(N))$	$O(N*log(N))$	$O(N^2)$

References

[1] *Sorting Algorithms*,
`http://www.cs.bilkent.edu.tr/~ugur/teaching/cs202/lectures/L2_SortingAlgorithms.pdf`

[2] *Bitonic Sort*,
`https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Mullapudi-Spring-2014-CSE633.pdf`

[3] *Algorithm Analysis*,
`https://cs.lmu.edu/~ray/notes/alganalysis/`

[4] *Bitonic Sort*,
`https://www.geeksforgeeks.org/bitonic-sort/`