Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2025 Spring

Programming Assignment 1

March 21, 2025

Student name:
Arda AGACDELEN

Student Number:
b2220765009

1 Problem Definition

In this report, five different sorting algorithms (Shaker Sort, Comb Sort, Insertion Sort, Radix Sort, Shell Sort) will be analyzed. It will be observed if they give the expected asymptotic-time complexities for three datasets: random dataset, ascending dataset, and descending dataset. Their space complexities will be discussed. The algorithms for these three datasets will be compared with plotting comparison graphs. Additionally, for all algorithms, it will be shown which data set is more suitable in order to obtain a shorter running time. At the end of the report, the reader will know how the size of a data set affects the running time.

Dataset (TrafficFlowDataset.csv) includes more than 250,000 captures of communication packets sent in a bidirectional manner between senders and receivers over a certain period of time. In order to be able to perform a comparative analysis of the performance of the given sorting algorithms over different data sizes, work will consider several smaller partitions of the dataset, that is, its subsets of sizes 500, 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000, and 250000 starting from the beginning of the file. the records will be sorted based on the Flow Duration feature given in the 3rd column (Flow Duration), which is of type int.

2 Solution Implementation

Important Note: I had implemented this work using generics for sorting methods, objects of my custom Record class for data points, and a Record array for the dataset. However, as the code cache size is limited for the JVM, at a certain point, the cache was getting full. Increasing the code cache size hadn't worked. As a result, at a certain point, the code was slowing down, and the graphs were corrupted. Therefore, I changed the Record array to a long array without changing any logic, and the problem was solved. I will attach the ideal code for this test in an unused java file. Additionally, I will keep the corrupted results saved in case you want to see them. You can see the slowdown and corrupted linearity in the plots.

2.1 Radix Sort

- A non-comparative algorithm. It does not apply direct comparison among elements.
- Uses counting sort as a helper function
- Takes maximum number of digits in data elements as parameter d.
- It sorts the numbers digit by digit, from right digit to left digit.

```
1    public static Long[] radixSort(Long[] array, int d) {
2        for (int pos = 1; pos <= d; pos++) {
3            array = countingSort(array, pos);
4        }
5        return array;
6    }
```

```
7
8       private static Long[] countingSort(Long[] array, int pos) {
9           int size = array.length;
10          long digitNumber = (long) Math.pow(10, pos - 1);
11          int[] count = new int[10];
12          Long[] output = new Long[size];
13
14          for (int i = 0; i < size; i++) {
15              int digit = getDigit(array[i], digitNumber);
16              count[digit]++;
17          }
18
19          for (int i = 1; i < 10; i++) {
20              count[i] += count[i - 1];
21          }
22
23          for (int i = size - 1; i >= 0; i--) {
24              int digit = getDigit(array[i], digitNumber);
25              output[count[digit] - 1] = array[i];
26              count[digit]--;
27          }
28          return output;
29      }
30
31      private static int getDigit(Long number, long digitNumber) {
32          return (int) ((number / digitNumber) % 10);
33      }
```

2.2 Shell Sort

- An in-place, comparison-based sorting algorithm.
- It sorts elements by comparing and moving them based on a decreasing gap sequence.
- When ‘gap = 1‘, the algorithm works like insertion sort, ensuring full sorting. Takes advantage of insertion sort’s efficiency on partially sorted data.

```
34      public static void shellSort(Long[] array) {
35          int n = array.length;
36          int gap = n / 2;
37          int j;
38          Long key;
39
40          while (gap > 0) {
41
42              for (int i = gap; i < n; i++) {
43                  key = array[i];
```

```
                j = i;

                while (j >= gap && array[j - gap] > key) {
                    array[j] = array[j - gap];
                    j -= gap;
                }
                array[j] = key;
            }
            gap /= 2;
        }
    }
```

2.3 Comb Sort

- An in-place, comparison-based sorting algorithm.
- Improves on bubble sort: By using a larger step between elements, reduces the total number of comparisons and swaps.
- Uses swap() as a helper function.

```
    public static void combSort(Long[] array) {
        int gap = array.length;
        float shrink = 1.3f;
        boolean sorted = false;
        while (!sorted) {
            gap = Math.max(1, (int) Math.floor(gap / shrink));
            sorted = (gap == 1);
            for (int i = 0; i < array.length - gap; i++) {
                if (array[i] > array[i + gap]) {
                    swap(array, i, i + gap);
                    sorted = false;
                }
            }
        }
    }
```

2.4 Shaker Sort

- An in-place, comparison-based sorting algorithm.
- Input sensitive: It identifies a sorted list on the first iteration.
- Bidirectional version of bubble sort: Shaker sort is an improved version of Bubble Sort that sorts in both directions, improving efficiency.

- Uses swap() as a helper function.

```
70      public static void shakerSort(Long[] array) {
71          boolean swapped = true;
72          while (swapped) {
73              swapped = false;
74              for (int i = 0; i < array.length - 1; i++) {
75                  if (array[i] > array[i + 1]) { % \label{line:if}
76                      swap(array, i, i + 1);
77                      swapped = true;
78                  }
79              }
80              if (!swapped) {
81                  break;
82              }
83              swapped = false;
84              for (int i = array.length - 2; i >= 0; i--) {
85                  if (array[i] > array[i + 1]) {
86                      swap(array, i, i + 1);
87                      swapped = true;
88                  }
89              }
90          }
91      }
```

2.5 Insertion Sort

- An in-place, comparison-based sorting algorithm.
- Input sensitive: It identifies a sorted list on the first iteration.
- Insertion Sort builds the sorted array in the beginning of the same array, inserting each new element into its correct position in the already sorted portion.
- It works efficiently for small datasets or nearly sorted data

```
92      public static void insertionSort(Long[] array) {
93          Long key;
94          int i;
95
96          for (int j = 1; j < array.length; j++) {
97              key = array[j];
98              i = j - 1;
99
100             while (i >= 0 && array[i] > key) {
101                 array[i + 1] = array[i];
```

```
                    i = i - 1;
                }
                array[i + 1] = key;
            }
        }
```

3 Results, Analysis, Discussion

The provided tables show the execution times (in milliseconds) for five sorting algorithms (Shell Sort, Comb Sort, Radix Sort, Shaker Sort, and Insertion Sort) with different input sizes and dataset conditions (Random, Sorted, and Reverse Sorted). The changes in performance were recorded based on the dataset properties and the efficiency of the algorithm.

Table 1: Results of the running time tests performed for varying input sizes (in ms).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Comb Sort	0.3010	0.0810	0.1148	0.2061	0.5656	1.0946	2.5222	5.6509	13.1403	27.5476
Shaker Sort	0.7290	1.2795	2.3404	8.6016	32.5871	140.1211	512.4504	2100.8397	9275.3544	57066.1965
Radix Sort	0.2020	0.0755	0.1362	0.3806	0.2666	0.5410	1.0776	2.1814	4.8941	9.3690
Insertion Sort	0.3947	0.5083	0.8550	3.5199	12.9113	56.0324	209.5737	824.0374	3431.6431	14745.0590
Shell Sort	0.2314	0.1187	0.1745	0.3523	0.6467	1.4800	3.4960	8.7608	20.2112	39.8679
	Sorted Input Data Timing Results in ms									
Comb Sort	0.0058	0.0125	0.0337	0.0905	0.1638	0.3816	0.8541	1.9176	4.3973	9.4214
Shaker Sort	0.0005	0.0006	0.0011	0.0022	0.0047	0.0092	0.0192	0.0407	0.0833	0.1463
Radix Sort	0.0175	0.0354	0.0686	0.1390	0.3006	0.6071	1.2184	2.4530	4.7729	9.5264
Insertion Sort	0.0037	0.0014	0.0032	0.0063	0.0108	0.0206	0.0417	0.0856	0.1669	0.3351
Shell Sort	0.0050	0.0232	0.0250	0.0654	0.1246	0.2675	0.5803	1.2650	2.7075	5.5527
	Reversely Sorted Input Data Timing Results in ms									
Comb Sort	0.0116	0.0305	0.0374	0.0824	0.1901	0.4610	1.0124	2.1675	5.0546	10.6544
Shaker Sort	0.2748	1.0261	3.9000	14.7886	61.0720	248.8056	990.2220	4052.5036	16238.3947	64908.2212
Radix Sort	0.0176	0.0424	0.0680	0.1329	0.2821	0.6084	1.1754	2.4365	5.0502	10.1667
Insertion Sort	0.1399	0.4286	1.6292	6.3705	25.1611	104.0799	417.7233	1687.4097	6985.5538	27427.3754
Shell Sort	0.0138	0.0171	0.0378	0.0799	0.1788	0.3914	0.8197	1.7849	3.7481	8.0429

Table 2: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Comb Sort	$\Omega(nlogn)$	$\Theta(n^2/2^p)$	$O(n^2)$
Shaker Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Radix Sort	$\Omega(n*d)$	$\Theta(n*d)$	$O(n*d)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Shell Sort	$\Omega(nlogn)$	$\Theta(n^{1.5})$	$O(n^2)$

- Comb Sort:

Line 6: In each iteration of the while loop, the gap is reduced by a factor of approximately 1.3. The number of iterations is proportional to $(nlogn)$.
Line 8-10: The for loop runs (n) times for each iteration of the while loop. Since the number of while loop iterations is $(nlogn)$, the overall time complexity is $(nlogn)$.

- Insertion Sort

Line 2: The outer loop runs (n) times, where n is the length of the array.
Line 5-7: In the worst case, the inner while loop might shift every element in the array. This leads to (n) comparisons and shifts for each iteration of the outer loop, so the overall time complexity is (n^2).

- Shaker Sort

Line 3: The while loop runs as long as a swap is made, which in the worst case can happen up to (n) times.
Line 5-9: The inner for loop runs (n) times in the worst case.
Line 15-19: Similarly, the second inner for loop also runs (n) times.
The time complexity of Shaker Sort is (n^2), as it involves two nested loops with a maximum of (n) iterations each.

- Shell Sort

Line 3: The gap starts at $n/2$ and is halved each time, which results in $(logn)$ iterations for the while loop.
Line 5-9: For each gap value, the inner for loop runs (n) times in the worst case.
Line 9-12: The inner while loop for shifting elements runs (n) in the worst case for each gap size.
The overall time complexity of Shell Sort is $(nlogn)$ - $(n^{1.5})$ with a specific choice of gaps, but in the worst case, it can be (n^2).

- Radix Sort

Line 2: The outer loop runs for (d) digits, where d is the number of digits.
Line 3: The counting sort runs in (n), where n is the number of elements.
Therefore, the overall time complexity of Radix Sort is $(n*d)$

Table 3: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Comb sort	$O(1)$
Shaker sort	$O(1)$
Radix sort	$O(n+k)$
Insertion sort	$O(1)$
Shell sort	$O(1)$

- Comb Sort: Uses only in-place swaps.
Line 10: swap(A[i], A[i+gap]) (No extra memory used, swaps in-place)

- Insertion Sort: Shifts elements within the same array.
Line 6: A[i + 1] ← A[i] (Shifts elements, no extra memory)
Line 9: A[i + 1] ← key (Uses existing space, no additional memory)

- Shaker Sort: Uses only in-place swaps.
Line 7: swap(A[i], A[i+1]) (No extra memory used, swaps in-place)
Line 17: swap(A[i], A[i+1]) (No extra memory used, swaps in-place)

- Shell Sort: Uses shifting like insertion sort.
Line 9: A[j] ← A[j - gap] (Shifts elements, no extra memory)
Line 12: A[j] ← temp (Uses existing space, no additional memory)

- Radix Sort: Uses extra arrays for counting sort.
Line 8: count ← array of 10 zeros (O(k) extra memory for count array)
Line 9: output ← array of the same length as A (O(n) extra memory for sorted output)

3.1 Comparison of Sort Algorithms

3.1.1 Comparison on Random Data

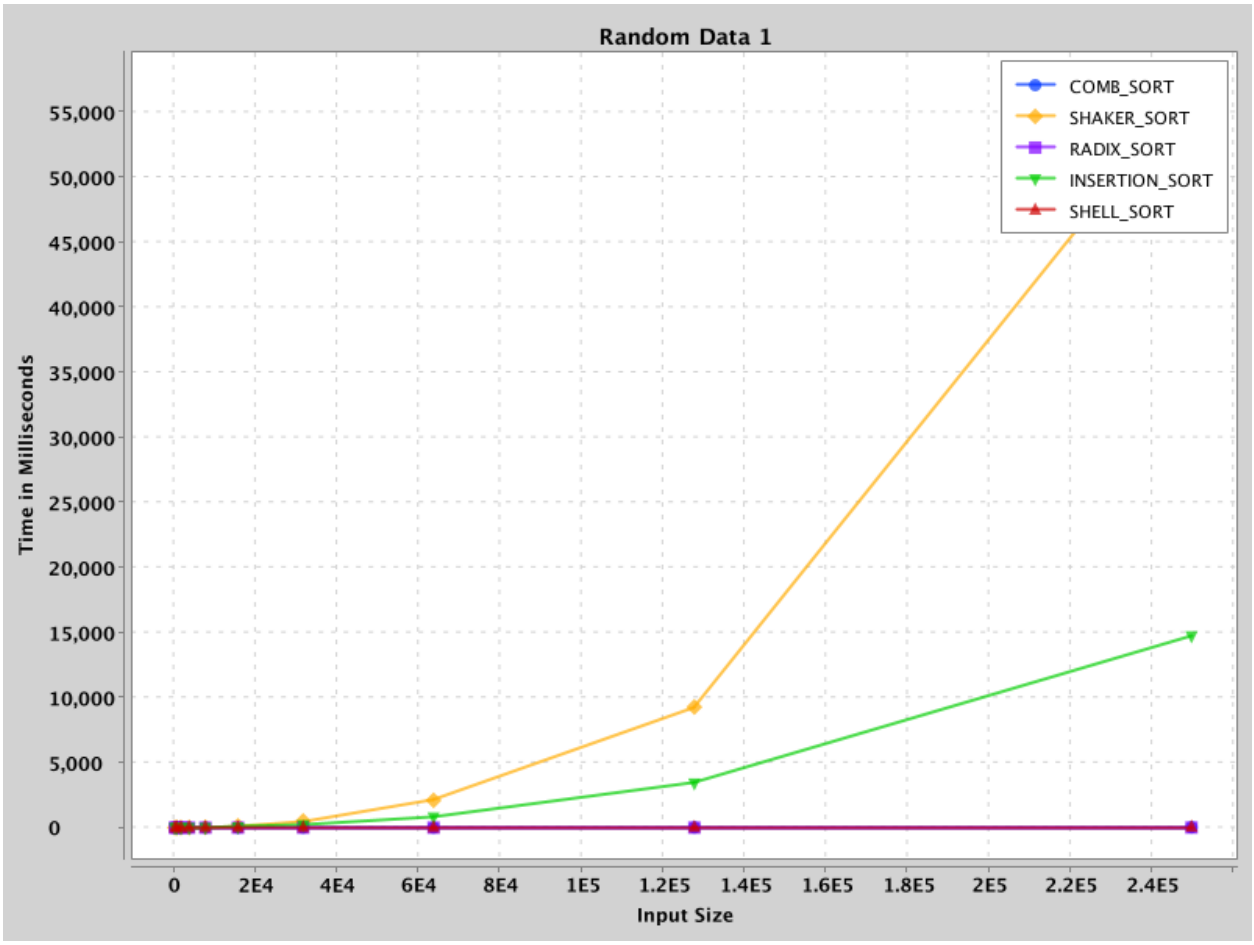

Figure 1: Comparison plot of the sort algorithms on random data.

When dataset is small, all algorithms are fast and do not show significant differences. However, shaker and insertion sort shows quadratic pattern, and significantly differs in the large amount of input data. Since shaker and insertion sort does not implement big steps for outlier data in the wrong ends, large datasets are not suitable for them. To be able to compare other algorithms, a graph without these two algorithms will be plotted.

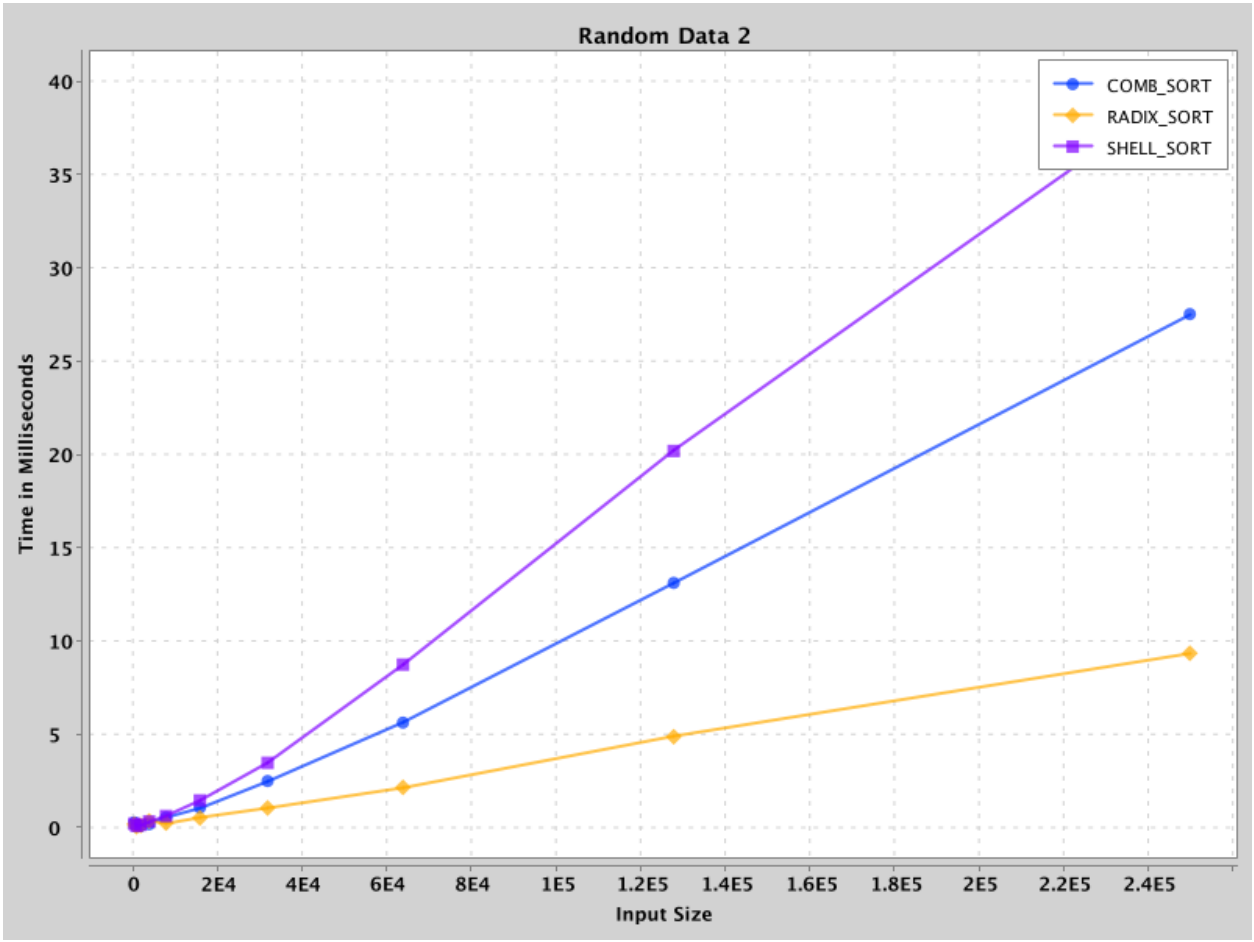

Figure 2: A closer look at the algorithms in the lower part of Figure 1.

From the figure 1 and the figure 2, it can be seen that shell, comb, and radix sort performs well on large datasets. Radix sort shows linear pattern due to constant number of digits, which makes it very efficient for random datasets among sort algorithms. Shell and comb sort show that the changes on insertion and bubble sort provided significant improvement. They almost followed a log-linear time sequence.

3.1.2 Comparison on Ascending Sorted Data

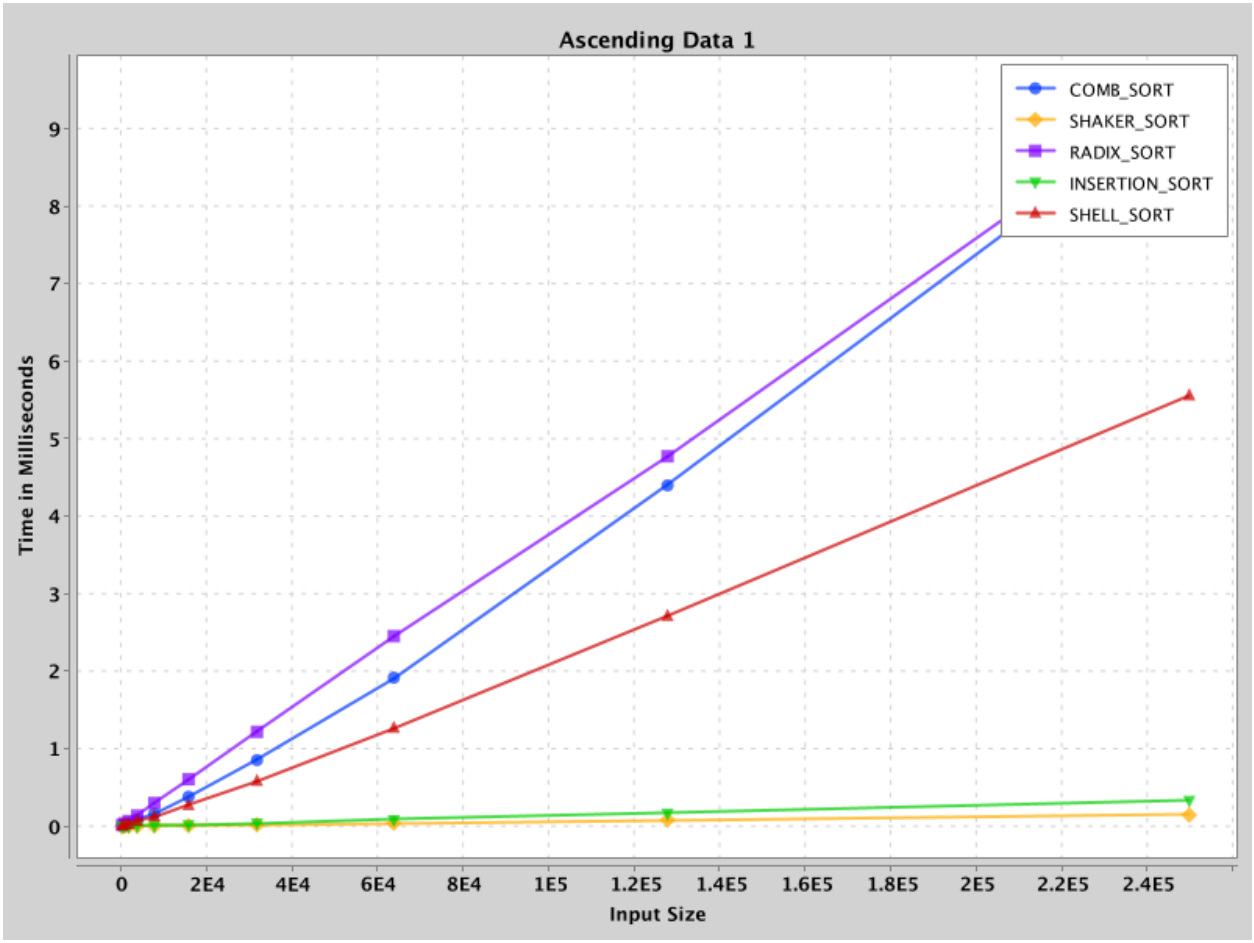

Figure 3: Comparison plot of the sort algorithms on ascending sorted data.

Radix short remains same because it has a non-comparison based structure. Rest of the algorithms show improvement on running time. Insertion and shaker sort show their input sensitive nature and perform best. They show linear time and stay under 1 milliseconds even for the largest dataset. Since the swap operation does not take place, the iterations of the shell and comb sort are faster now, and the total time almost shows a linear pattern.

3.1.3 Comparison on Descending Sorted Data

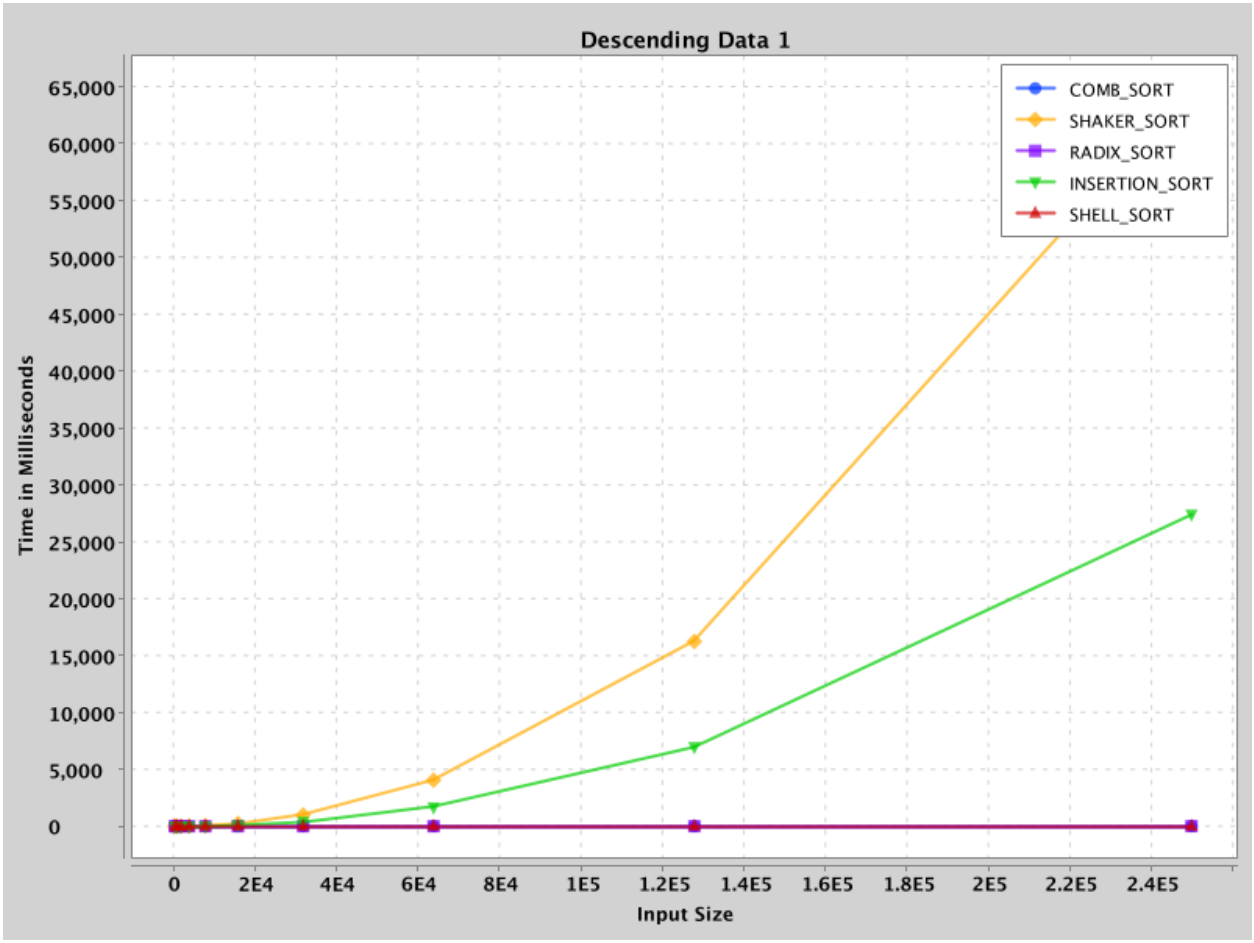

Figure 4: Comparison plot of the sort algorithms on descending sorted data.

Shaker sort and insertion sort suffer from the disadvantage of small steps and show significant running time increase on large datasets. They follow the worst case scenario and shows a quadratic pattern. To be able to compare other algorithms, a graph without these two algorithms will be plotted.

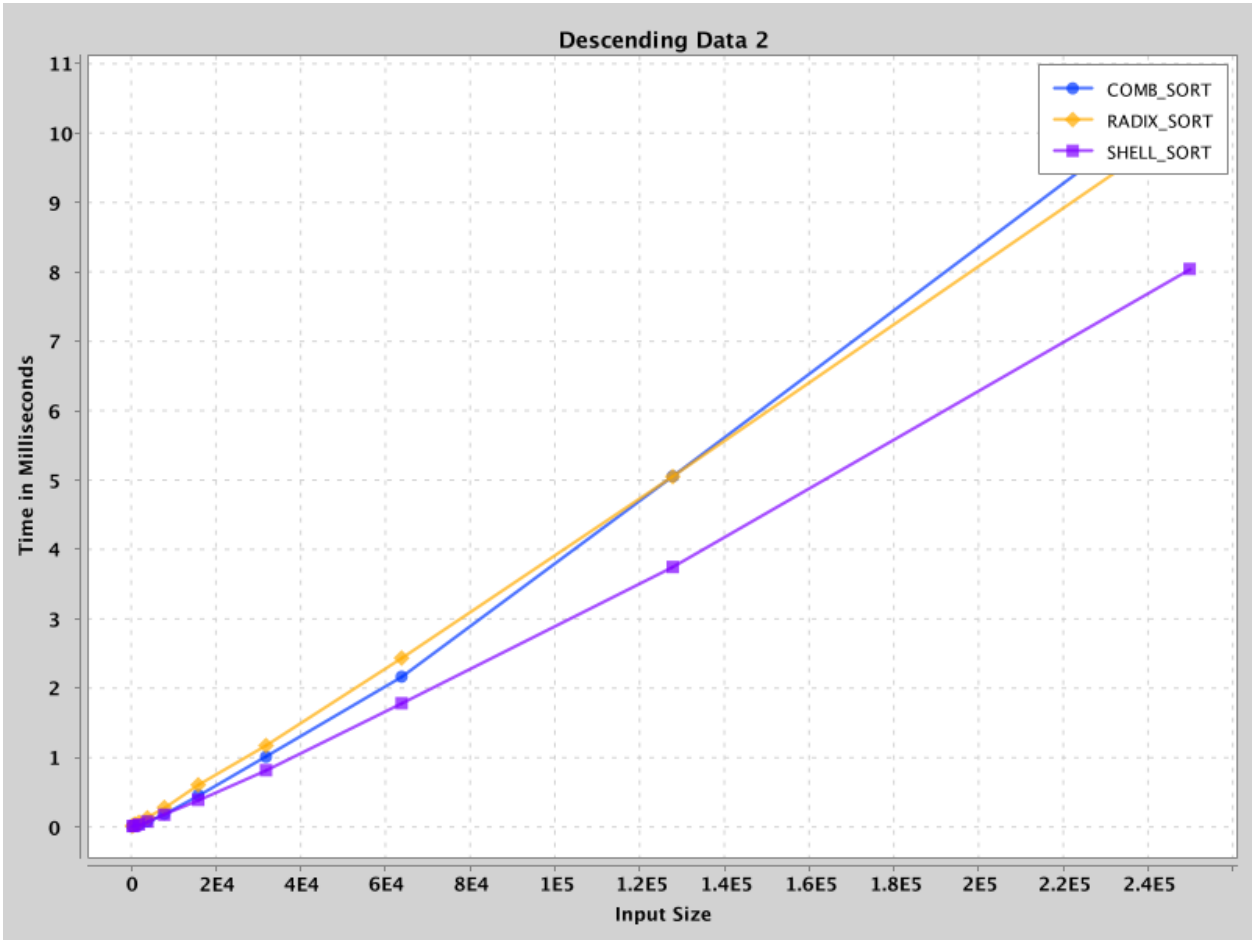

Figure 5: A closer look at the algorithms in the lower part of Figure 4.

Again, radix short remains same because it has a non-comparison based structure. The big steps in the beginning of sort operations for shell and comb sort appears to be beneficial. They show significant improvement on insertion and bubble sort. For reversely-sorted data, it can be said that these three algorithms remain efficient.

3.2 Performance Observation of Algorithms on Random, Sorted, and Reverse-Sorted Data

3.2.1 Comb Sort

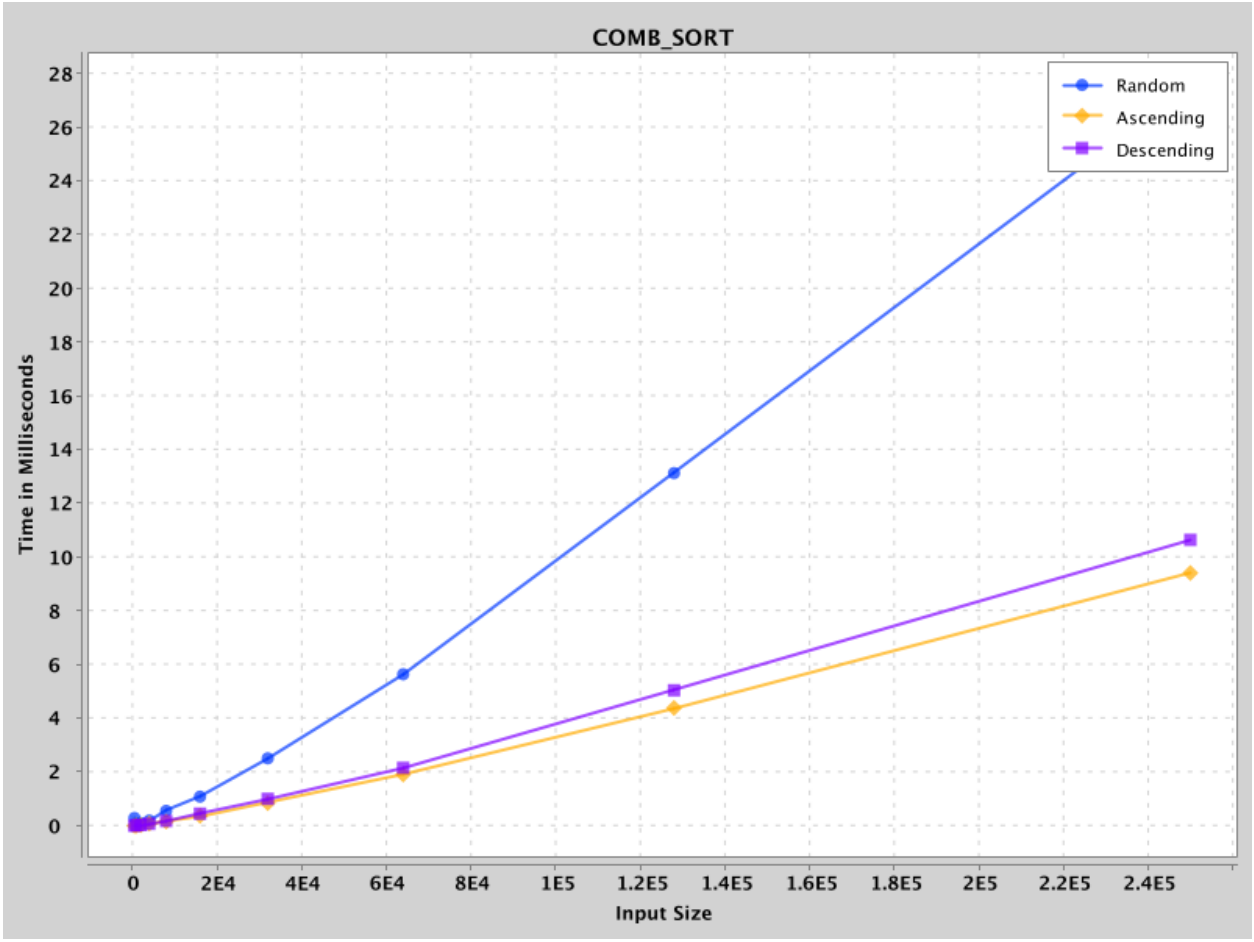

Figure 6: Comparison plot of the Comb Sort algorithm on three datasets: random, ascending, and descending, showing its performance under different input conditions.

- Results in efficient time for all data.
- Shows a significant improvement on bubble sort.
- Almost follows a log-linear pattern for random data and reversely sorted data.
- Almost follows a linear pattern for sorted.

- Test results show that the computational complexities have a loose bound from above in some scenarios. Which is normal, when the worst and best case scenarios are far from each other, algorithm can follow a non-asymptotic pattern.

3.2.2 Shaker Sort

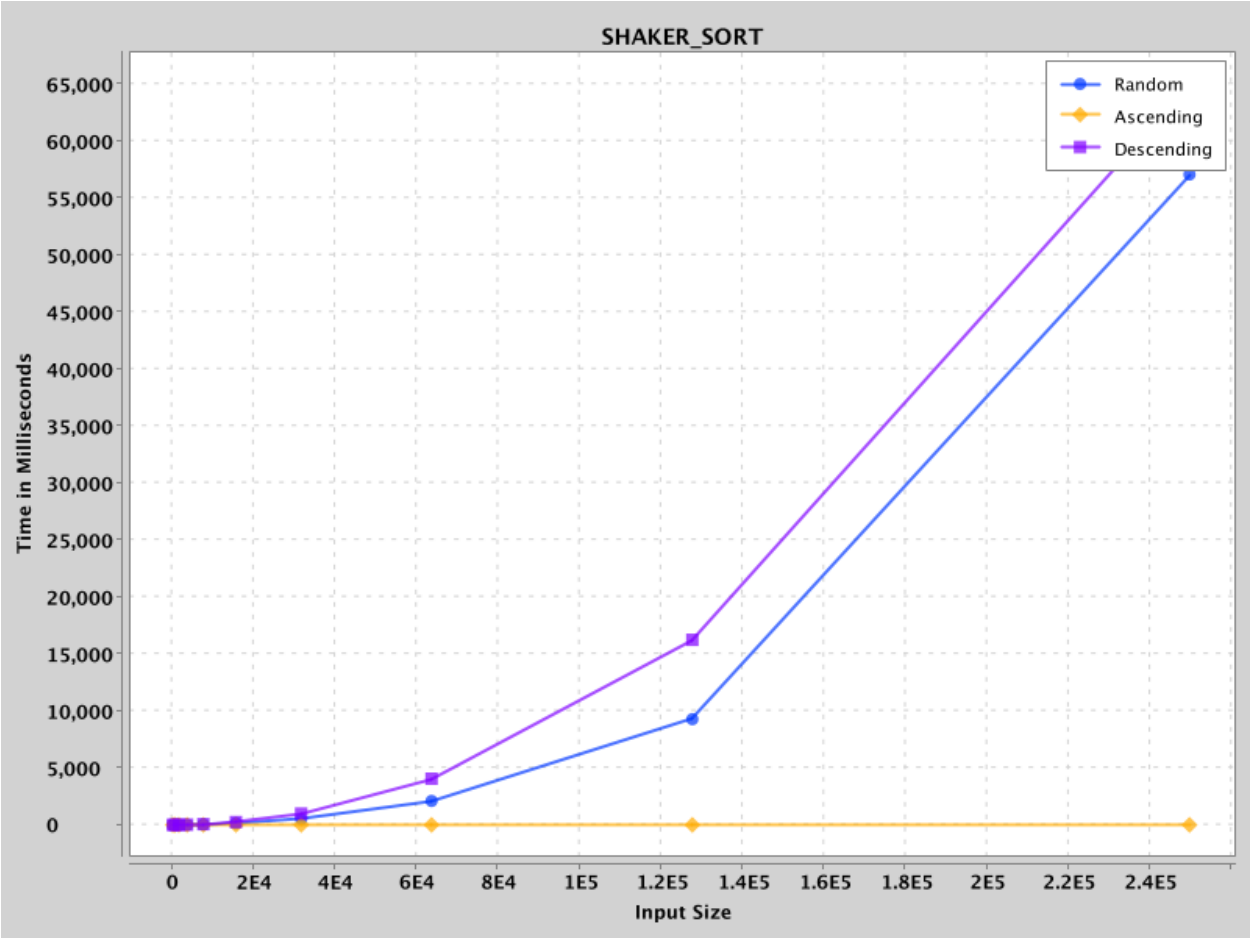

Figure 7: Comparison plot of the Shaker Sort algorithm on three datasets: random, ascending, and descending, showing its performance under different input conditions.

- Results in inefficient time for real life data.
- Shows a little improvement on bubble sort.
- Almost follows a quadratic pattern for random data and reversely sorted data.
- Follows a linear pattern for sorted data. (One whole iteration of the list, so fast with today's technology)
- Test results show that the computational complexities have a tight bound from above in all scenarios.

3.2.3 Radix Sort

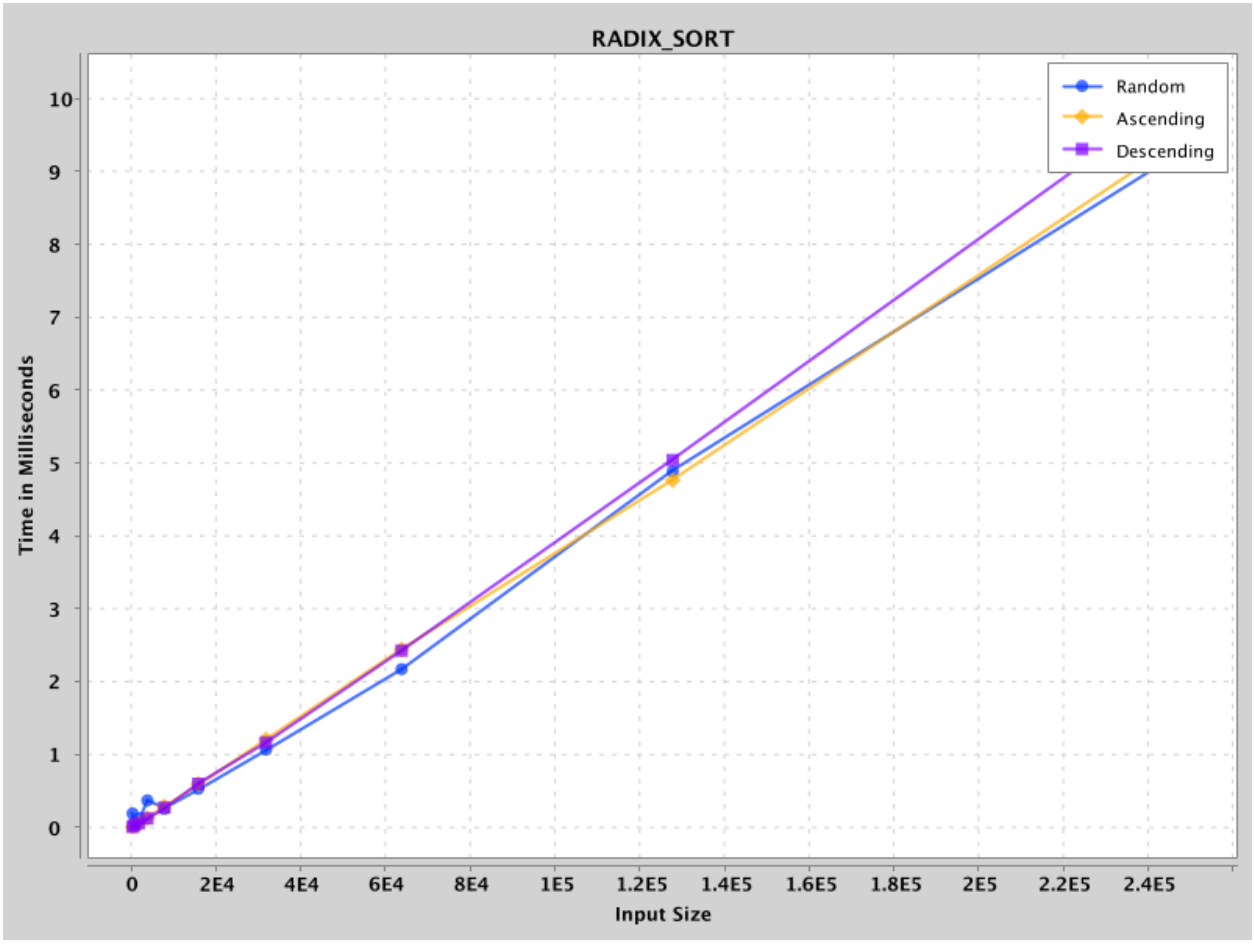

Figure 8: Comparison plot of the Radix Sort algorithm on three datasets: random, ascending, and descending, showing its performance under different input conditions.

- Results in efficient time for all data.
- Shows non-sensitive nature for input data and runs at the same speed in all cases.
- Follows a linear pattern for all data (It is because d:number of digits were constant for every size of the dataset)
- Radix Sort efficiently handles large numerical ranges by sorting digits individually using Counting Sort, with time complexity dependent on the number of digits rather than the value size, and the number of passes is fixed regardless of the range size.

- Test results show that the computational complexities have a tight bound from above in all scenarios.

3.2.4 Insertion Sort

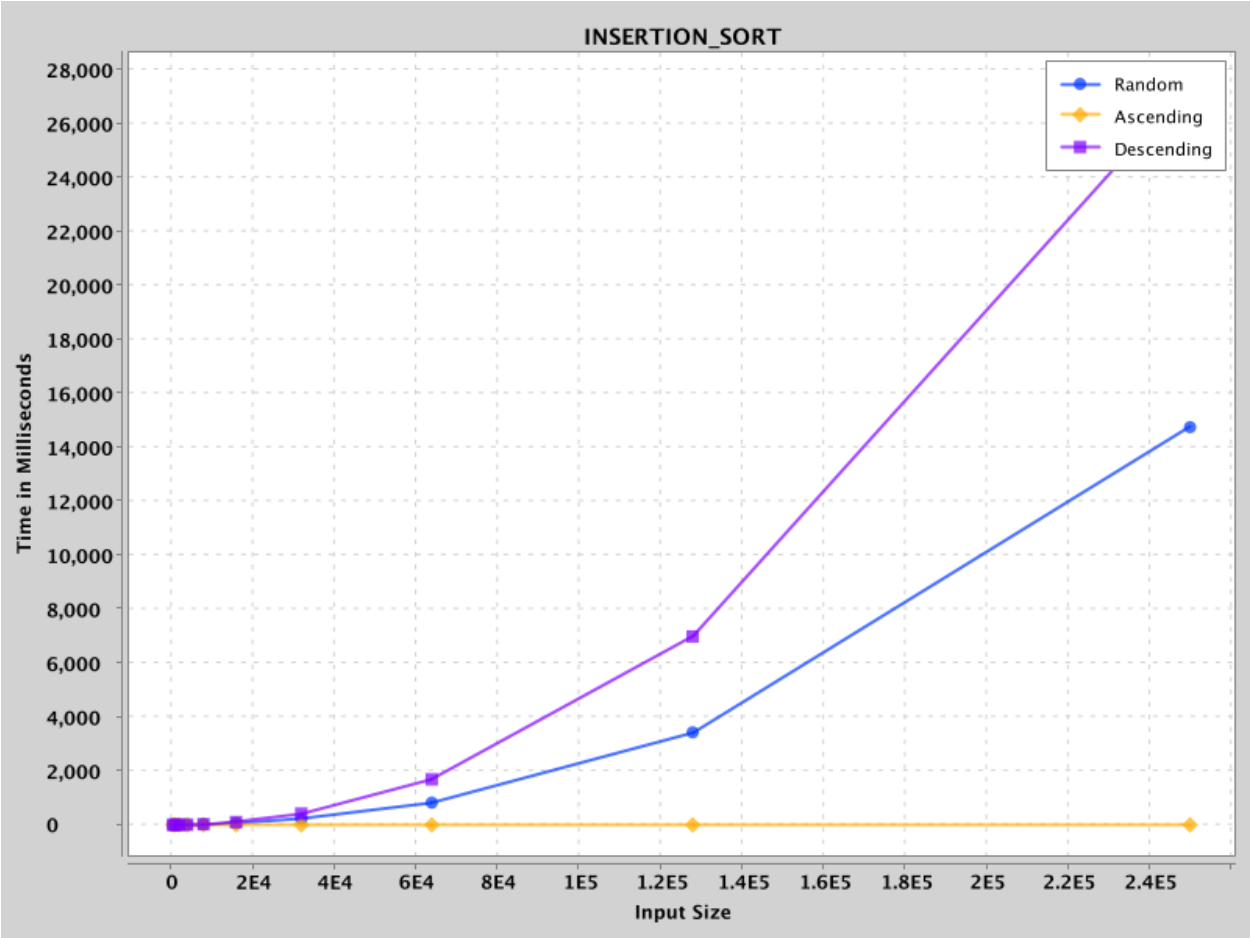

Figure 9: Comparison plot of the Insertion Sort algorithm on three datasets: random, ascending, and descending, showing its performance under different input conditions.

- Results in inefficient time for real life data.
- Shows sensitive nature for input data and runs faster on sorted data.
- Works fast for partially sorted and sorted data.
- Follows a linear pattern for sorted data. (one comparison per element, so fast with today's technology)
- Almost follows a quadratic pattern for random and reversely-sorted data. (Reversely-sorted data is worst case as it requires maximum number of changes in array.)

- Test results show that the computational complexities have a tight bound from above in all scenarios.

3.2.5 Shell Sort

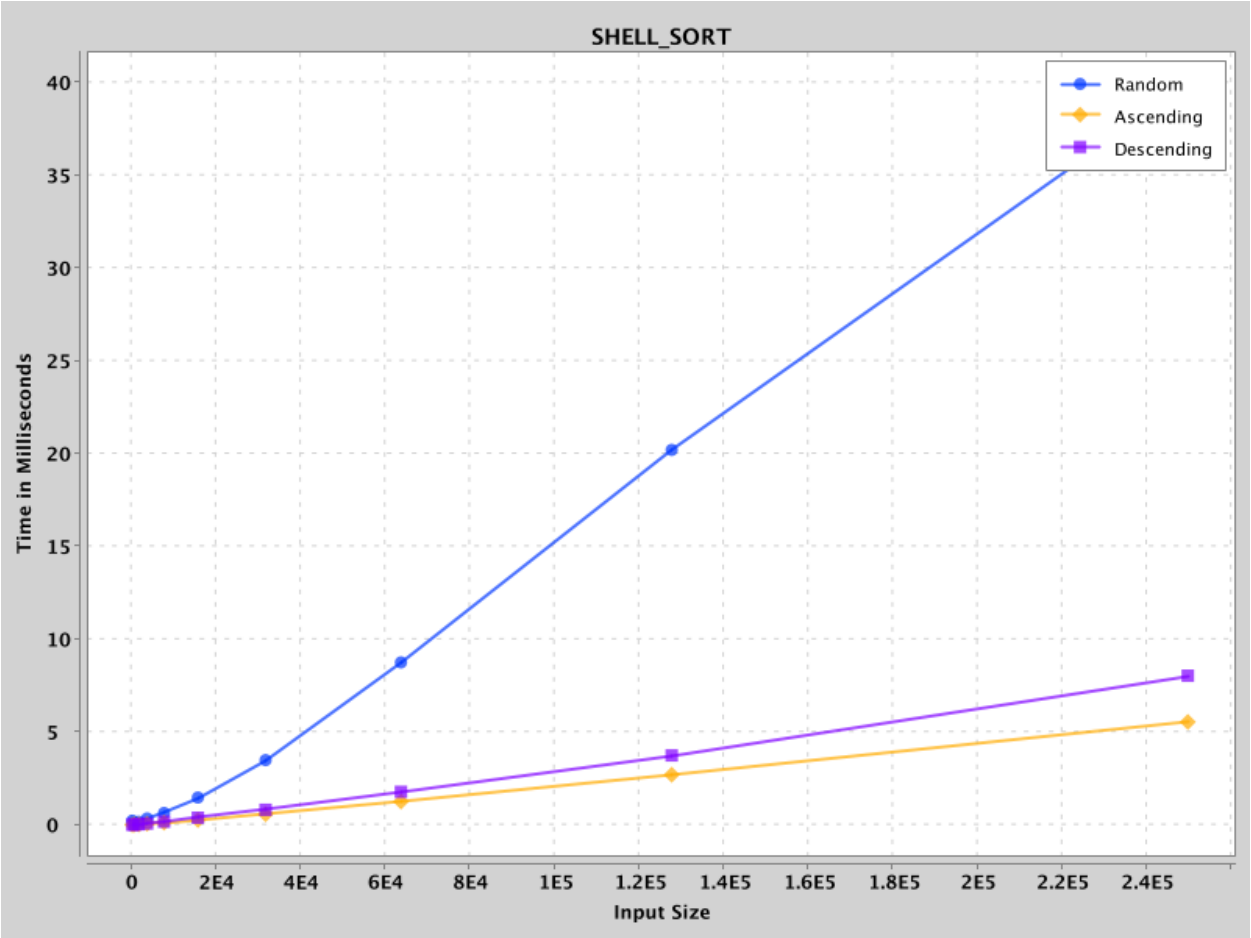

Figure 10: Comparison plot of the Shell Sort algorithm on three datasets: random, ascending, and descending, showing its performance under different input conditions.

- Results in efficient time for all data.
- Shows sensitive nature for input data and runs faster on sorted and reversely-sorted data.
- Having benefit of large steps, performs better than insertion sort on partially-sorted, sorted, and reversely-sorted data.
- Almost follows a log-linear pattern for sorted data and reversely-sorted data.
- Almost follows a quadratic pattern for random.

- Test results show that the computational complexities have a tight bound from above in all scenarios.

4 Notes

Table 4: System Specifications Used for Performance Evaluation

Component	**Details**
Operating System	macOS Sequoia 15.3.2
Processor	Apple M3, 10-core CPU
RAM	16GB
Programming Language	Java 21
Graphing Library	XChart v3.8.8

• The execution time of each sorting algorithm was recorded and averaged over ten runs to minimize fluctuations.

• Results may vary on different hardware or JVM configurations.

References

References

[1] Python Read, "Shell Sort," Available at: https://pythonread.github.io/dsa/shell-sort.html. [Accessed: March 2025].

[2] Wikipedia, "Shellsort," Available at: https://en.wikipedia.org/wiki/Shellsort. [Accessed: March 2025].

[3] Wikipedia, "Sorting Algorithm," Available at: https://en.wikipedia.org/wiki/Sorting_algorithm. [Accessed: March 2025].

[4] GeeksforGeeks, "Shell Sort," Available at: https://www.geeksforgeeks.org/shell-sort/. [Accessed: March 2025].

[5] GeeksforGeeks, "Time and Space Complexity of Radix Sort Algorithm," Available at: https://www.geeksforgeeks.org/time-and-space-complexity-of-radix-sort-algorithm/. [Accessed: March 2025].