CSE331 - Assignment #1

Kim MinKyo (20221044)
UNIST
South Korea
mindol@unist.ac.kr

1 PROBLEM STATEMENT

Sorting is the process of arranging data in a specific order based on a certain rule. Efficient sorting is important not only for sorting itself, but also because it helps improve other algorithms such as searching and merging, which often rely on sorted data to work correctly.

Many different sorting algorithms have been developed, each with its own strengths and weaknesses. In this project, we implemented and analyze 12 sorting algorithms. These include 6 basic sorting algorithms that have been studied for a long time, and 6 advanced or modern sorting algorithms. The goal of this project is to understand how each algorithm works, analyze its time and space complexity, and compare their performance with different types of input data.

We test each algorithm with random, sorted, reverse-sorted, and partially sorted inputs. We then compare the results based on speed, memory usage, and stability. You can find my implementation and input dataset in my github[1].

2 BASIC SORTING ALGORITHMS

There are many advanced sorting algorithms now, but sill, basic sorting algorithms, become the foundation of advanced sorting algorithms. They are very useful for learning how sorting works and understanding algorithmic thinking.

In this section, we will study about the six well-known basic sorting algorithms: Merge Sort, Heap Sort, Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort.

2.1 Merge Sort

Merge Sort [2] is a **Divide-and-Conquer** sorting algorithm. It works by dividing the array into two part, sorting each part, and then merging them. Because they need to store the part of the array temporarily in merging step, it is **not in-place**.

Merge Sort always divides the array into two parts until each subarray has only one element. Since a single element is already sorted, sorting is not needed for them. Then it merges the subarrays back in sorted order.

The steps of Merge Sort are as follows:

- **Divide:** Split the array into two part.
- **Conquer:** Recursively apply Merge Sort to both part.
- **Combine:** Merge the two sorted part into one sorted array. This step takes extra space.

The time complexity of Merge Sort in the best, average, and worst cases is $O(n \log n)$. This is because the algorithm consistently divides the array into half and then merges them in linear time. Table 1 shows the overall property of the Merge Sort, including time complexity, space complexity, and Stability.

[1]https://github.com/mindolii/Algorithm

Best	**Average**	**Worst**	**Space**	**Stable**
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes

Table 1: Time and Space Complexity of Merge Sort

2.2 Heap Sort

Heap Sort [2] is a sorting algorithm that uses a special data structure called a **heap**. A heap is a complete binary tree where each parent node is greater than (or equal to) its children (in a max heap), or smaller than or equal to its children (in a min heap).

Heap Sort first builds a heap from the input data. If a max heap is used, the root node will always be the largest element. Then they swaps the root with the last element in the array. After this, the size of the heap is reduced by one, and the heap is rebuilt for the remaining elements. This process is repeated until all elements are sorted and the heap is empty.

Best	**Average**	**Worst**	**Space**	**Stable**
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No

Table 2: Time and Space Complexity of Heap Sort

Heap Sort has a time complexity of $O(n \log n)$ in the best, average, and worst cases. This is because the Heap Sort relies on the height of the tree. It is not stable and do not require additional space.

2.3 Bubble Sort

Bubble Sort [2] is very simple sorting algorithm. It works by repeatedly going through the array and comparing each pair of adjacent elements. If the elements are not sorted, it swaps them. This process is repeated until the array is fully sorted. The reason it is called "Bubble Sort" is because the movement of elements resembles bubbles rising to the surface.

Best	**Average**	**Worst**	**Space**	**Stable**
$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes

Table 3: Time and Space Complexity of Bubble Sort

Bubble Sort performs best when the input is already sorted, because it doesn't have to swap anything, resulting in $O(n)$ time. However, in the average and worst cases, it requires $O(n^2)$ time due to repeated pairwise comparisons and swaps. It is a stable, in-place sorting algorithm with constant space complexity.

2.4 Insertion Sort

Insertion Sort [2] is a simple sorting algorithm that works by inserting element one by one. It starts from the second element and compares it with the elements before it. If the element is smaller than the one before, it keeps moving it backward until it finds the correct position. This process is repeated for all elements until the array is completely sorted.

Best	Average	Worst	Space	Stable
$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes

Table 4: Time and Space Complexity of Insertion Sort

Insertion Sort runs in $O(n)$ time in the best case, which occurs when the input is already sorted. In the average and worst cases, it requires $O(n^2)$ time because it need to perform many comparisons and shifts. It is a stable, in-place algorithm with $O(1)$ space complexity and performs well on small or nearly sorted datasets.

2.5 Selection Sort

Selection Sort [5] is a simple sorting algorithm that works by finding the smallest element from the unsorted part of the array and putting it in the correct position. It keeps dividing the array into two parts: sorted part and unsorted part. At each step, it selects the minimum value from the unsorted part and swaps it (i.e., change the position) with the first unsorted element in the unsorted part. This process continues until the whole array is sorted.

Best	Average	Worst	Space	Stable
$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No

Table 5: Time and Space Complexity of Selection Sort

Selection Sort performs consistently in all cases with a time complexity of $O(n^2)$, as it always scans the whole unsorted portion of the array to find the minimum element. It makes fewer swaps compared to Bubble Sort, but it is not stable. The algorithm is in-place and uses only $O(1)$ additional space.

2.6 Quick Sort

Quick Sort [2] is **Divide-and-Conquer** sorting algorithm. It works by selecting a pivot element from the array and then putting all smaller elements to the left of the pivot and all larger elements to the right of the pivot. This process is called "partitioning." Here, the pivot can be chosen randomly, or the element at the left (or right) end can be chosen. And when the partitioning is over, the pivot goes to the right position by swapping. Then, Quick Sort is repeatedly called on the left and right parts of the array. It keeps doing this until the whole array is sorted.

The steps of Quick Sort are as follows:

- **Divide:** Choose a "pivot" element from the array. Partition the array into two parts – elements smaller than the pivot and elements greater than the pivot.
- **Conquer:** Recursively apply Quick Sort to the left and right parts of the array.
- **Combine:** Nothing to do, since the elements are sorted in place.

Best	Average	Worst	Space	Stable
$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	No

Table 6: Time and Space Complexity of Quick Sort

Quick Sort has an average and best case time complexity of $O(n\log n)$ due to efficient partitioning. However, in the worst case, such as when the pivot is the smallest or largest element, it degrades to $O(n^2)$. It is not stable and in-place with $O(\log n)$ auxiliary space due to recursion.

2.7 Summary of Basic Sorting Algorithms

Name	Best	Average	Worst	Space	Stable
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	Yes
Heap	$O(n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	No
Quick	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n)$	No

Table 7: Summary of Time and Space Complexities of Classical Sorting Algorithms

3 ADVANCED SORTING ALGORITHMS

While basic sorting algorithms are useful for understanding how sorting works, they are often too slow or inefficient. To solve this, many advanced sorting algorithms have been developed that improve performance through more efficient techniques.

In this section, we will study about 6 relatively new and advanced sorting algorithms: Library Sort, Tim Sort, Cocktail Shaker Sort, Comb Sort, Tournament Sort, and IntroSort.

These algorithms use different ideas such as insertion with gaps, bidirectional passes, tournament trees, or hybrid strategies that combine multiple sorting techniques. Some of them are used in real-world systems, including Python and C++ standard libraries.

3.1 Library Sort

Best	Average	Worst	Space	Stable
$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(n)$	No

Table 8: Time and Space Complexity of Library Sort

Library Sort [1] is an improved version of Insertion Sort. In standard Insertion Sort, inserting a new element may require shifting many elements, which makes the algorithm inefficient. Library Sort solves this problem by having gaps between the sorted elements, just like how librarians leave empty space on bookshelves. It's not always effective, but it sorts faster with a high probability by inserting gaps between elements. Refer Algorithm 1.

Algorithm 1 LibrarySort

```
1:  function LIBRARYSORT(input_array, size)
2:      $cap \leftarrow 2 \cdot size + 1$
3:      Create sorted_array of size $cap$ filled with GAP
4:      $count \leftarrow 0$
5:      for each $x$ in input_array do
6:          INSERTWITHGAP(sorted_array, count, x, cap)  ▷ Insert $x$ using nearest gap
7:          if $count \cdot 2 > cap$ then
8:              REBALANCE(sorted_array, cap, count)  ▷ Double array size and spread elements
9:          end if
10:     end for
11:     return all non-GAP values from sorted_array
12: end function
```

With high probability, each insertion takes $O(\log n)$ time, leading to an overall time complexity of $O(n \log n)$. The algorithm uses extra space proportional to $O(n)$ to maintain these gaps and is not stable due to possible shifts during insertion. It is particularly effective when input elements are inserted in random order. The advantage of this algorithm is that it improves insertion sort with high probability. And the disadvantage of this library sort is that it requires additional memory and it is not stable.

3.2 Tim Sort

Tim Sort [4] is a hybrid sorting algorithm used in real-world programming languages such as Python. It combines the simplicity of Insertion Sort with the power of Merge Sort. It is based on the idea that in reality, the data will not be truely random, but partially sorted.

The algorithm first divides the array into small chunks called *runs*. Each run is sorted using Insertion Sort. The reason they use Insertion Sort here, is because Insertion sort is efficient for small or nearly sorted arrays. After that, these runs are merged using the Merge Sort algorithm until the entire array is sorted. The steps of Tim Sort are as follows:

- **Divide:** Split the array into small runs of fixed size (typically 32 or 64 elements).
- **Conquer:** Sort each run using Insertion Sort.
- **Combine:** Repeatedly merge the sorted runs using Merge Sort.

Detailed explanation of the algorithms is described in Algorithm 2.

Best	Average	Worst	Space	Stable
$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes

Table 9: Time and Space Complexity of TimSort

Algorithm 2 TimSort

```
1:  function TIMSORT(arr, size)
2:      for $i \leftarrow 0$ to $size$ step RUN do
3:          $right \leftarrow \min(i + \text{RUN} - 1, size - 1)$
4:          INSERTIONSORT(arr, $i$, $right$)
5:      end for
6:      for $s \leftarrow$ RUN to $size$ step $s \times 2$ do
7:          for $left \leftarrow 0$ to $size$ step $2 \cdot s$ do
8:              $mid \leftarrow \min(left + s - 1, size - 1)$
9:              $right \leftarrow \min(left + 2 \cdot s - 1, size - 1)$
10:             if $mid < right$ then
11:                 MERGE(arr, $left$, $mid$, $right$)
12:             end if
13:         end for
14:     end for
15: end function
```

In the best case, when the input is already sorted, TimSort performs in $O(n)$ time. On average and in the worst case, it guarantees $O(n \log n)$ performance. The algorithm is stable and uses $O(n)$ auxiliary space for merging operations. The advantage of this algorithm is that it is efficient for real-world data, and the disadvantage of this algorithm is that it requires additional memory for merging.

3.3 Cocktail Shaker Sort

Cocktail Shaker Sort [3] is a variation of Bubble Sort that sorts in both directions. It passes through the list forward to push the largest element to the end, and then backward to bring the smallest element to the beginning. The process repeats until no swaps are needed. Details are described in Algorithm 3.

Algorithm 3 BubbleSort

```
1: function BUBBLESORT(arr, size)
2:     for $i \leftarrow 0$ to $size - 1$ do
3:         for $j \leftarrow size - 1$ down to $i + 1$ do
4:             if $arr[j] < arr[j-1]$ then
5:                 swap $arr[j]$ and $arr[j-1]$
6:             end if
7:         end for
8:     end for
9: end function
```

The bidirectional approach helps move both large and small elements toward their correct ends more efficiently. While it can achieve $O(n)$ time in the best case when the data are already sorted, its average and worst case performance remain $O(n^2)$. The algorithm is stable and in-place, requiring only $O(1)$ auxiliary space. The advantage of this algorithm is that it is very easy to implement, and disadvantage of this algorithm is that it is still inefficient to handle large datasets.

Best	Average	Worst	Space	Stable
$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes

Table 10: Time and Space Complexity of Cocktail Shaker Sort

3.4 Comb Sort

Best	Average	Worst	Space	Stable
$O(n \log n)$	$O(n^2/2^p)$	$O(n^2)$	$O(1)$	No

Table 11: Time and Space Complexity of Comb Sort

Comb Sort [3] is an improvement of Bubble Sort. The algorithm starts with a very large gap between two elements which will be compared, and then gradually shrinks the gap in each pass. The gap is divided by a constant shrink factor (usually 1.3) on each iteration. When the gap reaches 1 (or becomes smaller than 1), the algorithm behaves like Bubble Sort, but with fewer comparisons remaining. Refer Algorithm 4.

Algorithm 4 CombSort

```
function CombSort(arr, size)
    $gap \leftarrow size$
    $shrink \leftarrow 1.3$
    $swapped \leftarrow$ false
    while $gap > 1$ or $swapped$ do
        $gap \leftarrow \lfloor gap/shrink \rfloor$
        if $gap < 1$ then $gap \leftarrow 1$
        end if
        $swapped \leftarrow$ false
        for $i \leftarrow 0$ to $size - gap - 1$ do
            if $arr[i] > arr[i + gap]$ then
                swap $arr[i]$ and $arr[i + gap]$
                $swapped \leftarrow$ true
            end if
        end for
    end while
end function
```

The average case complexity depends on the shrink factor p, commonly set to around 1.3, resulting in better performance than Bubble Sort. However, in the worst case, it still performs at $O(n^2)$. Comb Sort is not stable but is an in-place algorithm requiring only $O(1)$ extra space. The advantage of this algorithm is that it is simple to implement and the disadvantage of this algorithm is that it is not stable and has quadratic worst case running time.

3.5 Tournament Sort

Best	Average	Worst	Space	Stable
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	No

Table 12: Time and Space Complexity of Tournament Sort

Tournament Sort [5] is a tree-based sorting algorithm that simulates a tournament among the input elements. Pairs of elements compete and the smaller one becomes the parent node. The smallest become the root of this tree. After removing the winner(i.e., the smallest), the tree is updated to find the next smallest element. This process continues until all elements are sorted. Tournament Sort is a variation of Heap Sort. Detailed informations are in Algorithm 5.

Algorithm 5 TournamentSort

```
function TournamentSort(arr, size)
    Create an array tree of size $2 \cdot size$
    Copy input values to leaf nodes: tree[size+i] = arr[i]
    for $i \leftarrow 2 \cdot size - 1$ downto 2 step 2 do
        $k \leftarrow i/2$, $j \leftarrow i - 1$
        tree[k] ← index of smaller between tree[i] and tree[j]
    end for
    for $i \leftarrow 0$ to $size - 1$ do
        $winner \leftarrow tree[1]$
        $output[i] \leftarrow tree[size + winner]$
        Replace tree[size + winner] with $\infty$
        Update winner path from leaf to root
    end for
end function
```

The algorithm guarantees $O(n \log n)$ time in all cases and requires $O(n)$ space to maintain the tree. It is not in-place, not stable algorithm. The advantage of this algorithm is that it is quite fast for overall cases, and the disadvantage of this algorithm is that it uses additional spaces.

3.6 IntroSort

Introsort [6] is a hybrid sorting algorithm that begins with Quick Sort and switches to Heap Sort when the recursion depth exceeds certain limit. This algorithm is used in C++ for sorting.

Algorithm 6 Introsort

```
function Introsort(arr, begin, end, maxDepth)
    if $end - begin < 16$ then
        InsertionSort(arr, begin, end)
    else if $maxDepth == 0$ then
        HeapSort(arr, begin, end)
    else
        $pivot \leftarrow$ Partition(arr, begin, end)
        Introsort(arr, begin, pivot, maxDepth - 1)
        Introsort(arr, pivot + 1, end, maxDepth - 1)
    end if
end function
```

Best	Average	Worst	Space	Stable
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	No

Table 13: Time and Space Complexity of IntroSort

Switches to Heap Sort if the recursion depth exceeds a predefined threshold, prevents worst-case degradation to $O(n^2)$. This guarantees $O(n \log n)$ time complexity in all cases. The algorithm is not stable and operates in-place with $O(\log n)$ auxiliary space. The advantage of this algorithm is that it is fast, and the disadvantage of this algorithm is that it is not stable and not easy to implement.

3.7 Summary of Advanced Sorting Algorithms

Algorithm	Best	Average	Worst	Space	Stable
Library	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	No
Tim	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Cocktail	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Comb	$O(n \log n)$	$O(n^2/2^p)$	$O(n^2)$	$O(1)$	No
Tournament	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	No
IntroSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$	No

Table 14: Summary of Time and Space Complexities of Advanced Sorting Algorithms

4 EXPERIMENTAL RESULTS AND ANALYSIS

Assumptions. All sorting algorithms were implemented in C++ to closely follows their theoretical definitions, without introducing aggressive optimizations. The goal was to preserve the canonical behavior of each algorithm as described in standard references.

The input data was generated using Python, and all sorting algorithms were evaluated under the assumption that the elements to be sorted are integers.

4.1 Execution Time

4.1.1 Input Generation. To evaluate each sorting algorithm in various situations, we tested them with the input of size 1,000,000(1M) with four different types:

- **Sorted:** Elements arranged in increasing order.
- **Reverse Sorted:** Elements arranged in decreasing order.
- **Partially Sorted:** Half of the array is sorted; the other half is shuffled.
- **Random:** All elements are randomly generated.

Each input type was generated programmatically to ensure consistency across experiments.

4.1.2 Measurement. Execution time was measured using the `clock()` function from the C++ standard library. This method captures only the time consumed by the sorting algorithm itself, excluding any time spent generating input or printing output.

```
clock_t start=clock();
mergeSort(numbers,0,size-1);
clock_t end=clock();

double duration=(double)(end-start)/CLOCKS_PER_SEC;
```

This approach ensures precise measurement of the algorithm's runtime.

4.1.3 Experimental Procedure. For each combination of sorting algorithm and input type, the experiment was repeated 10 times to account for system-level fluctuations. The final reported runtime is the average of these 10 trials.

All tests were performed under the same hardware and software environment to maintain fairness. Memory allocations, input generation, and result handling were kept outside of the measurement interval to isolate sorting time as accurately as possible.

4.1.4 Experimental Result. Table 15 summarizes the average execution time (in seconds) of each sorting algorithm across four input types: Random, Sorted, Reverse Sorted, and Partially Sorted input of size 1,000,000.

Algorithm	Random	Sorted	Reverse	Partial
Merge	0.185	0.123	0.126	0.156
Heap	0.222	0.153	0.146	0.198
Bubble	2426.16	1432.02	1707.13	1657.23
Insertion	701.76	0.003	1379.93	535.24
Selection	1130.58	1130.63	1140.21	1130.33
Quick	0.130	1747.54	1488.34	0.129
Library	2407.57	0.150	2622.59	1715.15
Tim	0.156	0.080	**0.119**	**0.119**
CocktailShaker	1990.40	**0.0025**	3034.71	1677.33
Comb	0.217	0.112	0.120	0.216
Tournament	0.322	0.130	0.131	0.216
Intro	**0.126**	0.287	0.263	0.126

Table 15: Execution Time(sec) for Each Algorithm on Different Input Types

Overall Performance: Hybrid algorithms such as **Tim Sort** and **Intro Sort** showed strong and consistent performance across all input types. Notably, Tim Sort ranked first on two of the four input types — reverse sorted and partially sorted arrays — highlighting its adaptability and robustness.

Best Performers: Algorithms like **Insertion Sort** and **Cocktail Shaker Sort** showed exceptional performance on already sorted input. Cocktail Sort was the fastest (0.0025 sec), followed closely by Insertion Sort (0.003 sec). However, their performance declined dramatically on other input types, particularly reverse sorted data, where both exhibited near-worst-case behavior. **Tim Sort** and **Intro Sort** showed best on Random input, and Partially Ordered input, each.

Poor Performers: Traditional $O(n^2)$ algorithms such as **Bubble Sort**, **Selection Sort**, and **Insertion Sort** (except in the best case) were consistently the slowest. **Library Sort**, although theoretically efficient, performed poorly on random and reverse sorted input in practice. Additionally, **Quick Sort**, while excellent on random input, showed suboptimal performance on already sorted and reverse sorted data due to its pivot selection strategy.

Observations:

- Tim Sort was the most robust overall, consistently ranking among the top three across all input types.

- Insertion Sort and Cocktail Shaker Sort excelled only in best-case scenarios and were inefficient for more complex or unfavorable input orders.
- Quick Sort was highly efficient on random and partially sorted input, but its performance deteriorated on sorted and reverse sorted arrays.

These results emphasize the importance of selecting a sorting algorithm that aligns with the characteristics of the input data. As demonstrated by the performance of Cocktail Shaker Sort and Insertion Sort, an algorithm that performs poorly in general may become the best choice under specific conditions. Understanding the structure of the input is essential to choosing an efficient sorting strategy.

4.2 Memory Consumption

4.2.1 Input Generation. We used the same dataset as in 4.1.

4.2.2 Measurement. To measure memory consumption, we used the `heaptrack` tool, which records detailed heap usage of a process during its execution. While `heaptrack` introduces a slight overhead due to sampling and system interrupts, it was consistently applied to all sorting algorithms, ensuring fair and uniform measurement conditions. Additionally, memory usage for input and output handling is also included in the measurement; however, since the same logic was used across all algorithms and the input size was fixed, this contributes equally to each case. Therefore, the evaluation focuses on relative memory usage rather than absolute values.

4.2.3 Experimental Procedure. Unlike running time, which can be significantly change by situation, memory usage is generally less sensitive to such variations. Therefore, unlike the experiments described in Section 4.1, we performed a single memory usage test per input type and algorithm. Furthermore, since the memory consumption of each algorithm did not differ noticeably across different input types, we averaged the memory usage over the four input types and used this value as the representative memory consumption for that algorithm.

4.2.4 Experimental Result. Table 16 summarizes the average memory usage(MB) of each sorting algorithm.

Algorithm	Mem Use (MB)	Theoretical
Merge	**315.31**	$O(n)$
Heap	16.39	$O(1)$
Bubble	16.43	$O(1)$
Insertion	16.48	$O(1)$
Selection	16.35	$O(1)$
Quick	16.47	$O(\log n)$
Library	**20.16**	$O(n)$
Tim	16.22	$O(n)$
Cocktail Shaker	16.43	$O(1)$
Comb	16.53	$O(1)$
Tournament	**20.32**	$O(n)$
Intro	16.39	$O(\log n)$

Table 16: Memory Usage(MB) for Each Algorithm

Observations:

- Although both Merge Sort and TimSort require $O(n)$ auxiliary space in theory, their actual memory usage differed significantly in our measurements. This is likely due to implementation details: Merge Sort allocates temporary arrays more aggressively during recursion, while TimSort reuses buffers and manages memory more efficiently.
- All algorithms of space complexity $O(n)$, like MergeSort, Library Sort, and Tournament Sort, showed slightly more consumption of memory than $O(1)$ except TimSort.
- The additional space usage of $O(\log n)$ is small and does not significantly differ from $O(1)$. This is precisely why such $O(\log n)$ algorithms are often classified as in-place.

4.3 Stability

4.3.1 Input Generation. While the 1M size input used in Sections 4.1 and 4.2 was initially considered for stability testing, it was found to be difficult to generate repeated values due to the wide range of numbers. To clearly evaluate the stability of sorting algorithms, a smaller dataset of 1,000 elements was generated with the upper bound limited to 100. This setting increased the frequency of duplicate values, making it more clear to observe whether sorting algorithms preserve the original order of identical elements.

To track the relative order among duplicate values, each number was represented as a `float`, where the integer part indicates the actual key and the decimal part indicates its order among same values. For example, two identical keys such as 42 are stored as 42.000000 and 42.000001.

4.3.2 Measurement. As illustrated in Section 4.3.1, the input values were represented as `float`s, where the integer part indicates the key and the decimal part indicates the original order among duplicates. During the sorting process, only the integer part of each float was used for comparison.

After sorting, the original float values were used to verify stability. The result was considered *stable* if the sequence of float values formed a strictly non-decreasing order. Conversely, if any pair of identical keys appeared out of their original input order (i.e., if they are not strictly increasing), the result was considered as *not stable.*

4.3.3 Experimental Procedure. Following the input generation process described in Section 4.3.1, a dataset consisting of 1,000 randomly generated elements with frequent duplicate keys was created. This input was used uniformly across all sorting algorithms.

Each algorithm was executed once on the same input, and the result was analyzed to determine whether it preserved the relative order among elements with equal keys. To eliminate the influence of external factors, all non-sorting logic such as input/output handling was standardized across implementations.

4.3.4 Experimental Result. Refer Table 17.

Observations:

- The stability test results were found to be fully consistent with theoretical expectations.
- This confirms that the experimental setup was appropriately designed to distinguish between stable and unstable behaviors, and that the tagging method used in input encoding effectively revealed such distinctions.

Algorithm	Stable	Theoretical
Merge	Yes	Stable
Heap	No	Not Stable
Bubble	Yes	Stable
Insertion	Yes	Stable
Selection	No	Not Stable
Quick	No	Not Stable
Library	No	Not Stable
Tim	Yes	Stable
Cocktail Shaker	Yes	Stable
Comb	No	Not Stable
Tournament	No	Not Stable
Intro	No	Not Stable

Table 17: Stability of Sorting Algorithms

4.4 Overall Discussion

The experimental results aligned well with the theoretical expectations in terms of stability, time complexity, and memory usage.

Among all sorting algorithms, **TimSort** stood out as the most well-rounded option. It achieved the fastest execution time on two types of input, maintained low memory usage, and, preserved stability. These results closely reflect the strengths that have led to TimSort's adoption in widely-used programming languages such as Python and Java.

The experiment confirms that TimSort's hybrid strategy, combining merge sort with insertion sort,enables it to achieve both performance and stability. The algorithm not only meets theoretical expectations but also proves its real-world effectiveness in practice.

Another surprising result was that both **Cocktail Sort** and **Insertion Sort** performed exceptionally well on already sorted data, finishing significantly faster than most other algorithms. This demonstrates that despite their relatively poor worst-case complexity, these algorithms can still be very efficient under favorable input conditions.

These results suggest that the choice of sorting algorithm should be based not only on theoretical complexity but also on practical factors such as data characteristics, required stability, and available memory.

References

[1] Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. 2006. Insertion Sort is $o(n \log n)$. *Theory of Computing Systems* 39 (2006), 391–397.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd ed.). MIT Press.

[3] Ahmad H. Elkalhout and Ashraf Y. A. Maghari. 2017. A Comparative Study of Sorting Algorithms: Comb, Cocktail and Counting Sorting. *International Research Journal of Engineering and Technology (IRJET)* 4, 1 (January 2017). Available at www.irjet.net.

[4] Mohammad Rizal Hanafi, Muhammad Azfa Faadhilah, Deden Pradeka, and Muhammad Taufik Dwi Putra. 2022. Comparison Analysis of Bubble Sort Algorithm with Tim Sort Algorithm Sorting Against the Amount of Data. *Journal of Computer Engineering, Electronics and Information Technology (COELITE)* 1, 1 (April 2022), 9–13.

[5] Donald E. Knuth. 1998. *The Art of Computer Programming: Sorting and Searching.* Vol. 3. Addison-Wesley Professional.

[6] David R. Musser. 1997. Introspective Sorting and Selection Algorithms. *Software: Practice and Experience* 27, 8 (1997), 983–993.