CSE331 - Assignment #1

Seonguk Choi (20211331)
UNIST
South Korea
csuzzim@unist.ac.kr

1 PROBLEM STATEMENT

Sorting is a fundamental operation in computer science that arranges elements in a specific order, typically numerically or lexically. In this paper, we analyze and compare twelve sorting algorithms, including six traditional comparison-based algorithms (merge sort, heap sort, bubble sort, insertion sort, selection sort, and quick sort) and six modern algorithms (library sort, team sort, cocktail shaker sort, comb sort, tournament sort, and intro sort).

The goal of this paper is to comprehensively evaluate the theoretical basis and empirical performance for various data distributions and input sizes. After examining the theoretical time complexity, space requirements, and stability properties of each algorithm, we experimentally evaluate them using datasets with various characteristics, such as sorted data (ascending and descending), random data, and partially sorted data. For each dataset type, we measure the running time over input sizes ranging from $1,000$ to $1,000,000$, and repeat each test at least 10 times to ensure statistical stability.

Through this systematic analysis, we will analyze factors such as data characteristics, memory constraints, and performance requirements to select the appropriate sorting algorithm in various situations, ultimately providing a deep understanding of each algorithm, identifying the optimal algorithm for a specific environment, and considering additional improvements.

2 BASIC SORTING ALGORITHMS

2.1 Merge Sort

[1]The core idea of merge sort is to divide a list into several parts, sort them partially, and then merge them back together. Merge sort is a representative algorithm that uses the divide and combine method, and can be analyzed by dividing it into three parts: divide, combine, and combine.

The divide part calculates the middle part of the array, so it takes $\Theta(1)$. The combine part sorts the subarrays that are divided into two halves, so it takes $2T(n/2)$. Finally, the combine part compares the elements of the entire array one by one and sorts them, so it takes $\Theta(n)$. Therefore, the time taken is expressed as follows:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & \text{if } n > 1. \end{cases} \tag{1}$$

And through this formula, we can calculate the time complexity. Using the recursion tree, the total sum of the combine part of each level is all cn. And since the number of levels of the entire tree is lg(n) + 1, the total cost is $cn(\lg n + 1) = cn \lg n + cn$. Therefore, the total time complexity is $T(n) = \Theta(n \lg n)$. [2]

2.2 Heap Sort

The core idea of heap sort is to use a heap (nearly binary complete tree) data structure to store each value in a node and sort it using the heap property. I implemented it using the max heap property. The algorithm can be largely divided into three: Max Heapify(), Build Heap(), and Heap Sort().

Max Heapify() is a recursive algorithm that checks whether a specific index i in an array satisfies the max heap property and repeats until the condition is satisfied. The worst case is when the size of the left child's subtree is $2n/3$, and since one exchange is sufficient to maintain the max heap property of three nodes, it is $\Theta(1)$. Therefore, $T(n) \leq T(2n/3) + \Theta(1)$ [2], and according to the mater theorem, $T(n) = O(\lg n)$. Build Max Heap() is an algorithm that iterates over all elements of a given array to satisfy the max heap property. At this time, for n elements, height is $\lfloor \lg n \rfloor$, and the maximum number of nodes with height h is $\lceil n/2^{h+1} \rceil$, so the formula is as follows:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}) = O(\sum_{h=0}^{\infty} \frac{h}{2^h}) = O(n)$$

And Heap Sort() is an algorithm that performs Build Max Heap and then performs Max Heapify() $n-1$ times by utilizing the characteristic that the root node is the maximum. Since Build Max Heap() is $O(n)$ and Max Heapify() is $O(lgn)$, therefore

$$T(n) = O(n) + nO(\lg n) = O(n \lg n). \tag{2}$$

2.3 Bubble Sort

The core idea of bubble sort is to compare the sizes of two elements, put them in the correct order, and repeat this for all indices until the end of the array. It consists of a double loop, where i is from 0 to $n-2$, and j is from 0 to $n-i-2$. Therefore, $T(n) = n-1+n-2+\cdots+1$, so

$$T(n) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2} = O(n^2). \tag{3}$$

2.4 Insertion Sort

The core idea of insertion sort is to repeatedly compare all elements in an array from the beginning to the end with the already sorted array, find their positions, and insert them. Compared to bubble sort and selection sort, both algorithms compare all elements from the beginning to the end, but insertion sort can stop the iteration once it finds its position, so it is slightly faster even though it has the same time complexity of $O(n^2)$. Also, I implemented insertion sort separately for left and right in order to use it in intro sort. Assuming $left = 0, right = n-1$, the time complexity of the best case and worst case are different.

Best case: If all elements are already sorted, since comparison operation is performed only once while i is iterated from 0 to $n-1$, $T(n) = n - 1 = O(n)$.

Worst case: If all elements are sorted in reverse order, while i is iterated from 0 to n˘1, comparison and insertion increase from 1 to

n˘1. Therefore,

$$T(n) = \sum_{k=1}^{n-1} k = O(n^2) \tag{4}$$

2.5 Selection Sort

The core idea of selection sort is to find the minimum value in an array, move that value to the beginning, and then repeat the array excluding that value as the next array. Selection sort always has a time complexity of $O(n^2)$ like bubble sort, but selection sort takes less time than bubble sort because it only exchanges values once per iteration.

As i increases from 0 to $n-1$, j in the inner loop also increases from $i+1$ to $n-1$. Therefore,

$$T(n) = \sum_{k=1}^{n-1} k = O(n^2).$$

2.6 Quick Sort

The core idea of quicksort is to divide the array into two parts, one with larger numbers and the other with smaller numbers, similar to mergesort, using the divide and conquer method, and recursively sort each part again. Therefore, quicksort can also be analyzed by dividing it into three parts: divide, conquer, and combine.

[2]The divide part divides $A[p..r]$ into $A[p..q..r]$ for the pivot q. At this time, $A[p..q-1]$ is less than or equal to $A[q]$, and $A[q]$ is less than or equal to $A[q+1..r]$. Then, the conquer part recursively calls divide on the two subarrays $A[p..q-1]$ and $A[q+1..r]$ to sort all the elements. Finally, combine is not necessary because the array is already sorted.

Quicksort consists of two functions: Quicksort() and Partition(). First, in the case of Quicksort(), if $p < r$, it calls Partition() to find the pivot q, and recursively calls the function for the left subarray smaller than q and the right subarray greater than q. Partition() iterates over j from p to $r-1$, and if $A[j]$ is less than or equal to the pivot, it swaps the values of $A[i]$ and $A[j]$.

Here, quicksort shows $O(n \lg n)$ performance in average situations, but has time complexity of $\Theta(n^2)$ in worst cases. If the array is sorted and the last element is always set as the pivot, then other values except the pivot are always smaller than the pivot, so $T(n) = T(n-1) + \Theta(n)$. At this time, $T(n-1) = T(n-2) + \Theta(n)$, and if we repeatedly substitute this, $T(n) = \Theta(n^2)$.

Also, in the best case, Partition() divides it into two subarrays with almost half the size. This case is the most stable case, and $T(n) = 2T(n/2) + O(n)$. Using the Master theorem, we can see that $T(n) = O(n \lg n)$.

However, in the average case, which is neither the worst nor the best, it has a time complexity of $O(n \lg n)$, and the process is as follows. First, let's think that Partition() always divides the subarray in a ratio of 1 to 9. In this case, we can obtain the following formula. $T(n) = T(9n/10) + T(n/10) + cn$. When we think of the recursion tree corresponding to this formula, the highest height is $\log_{10/9} n$, and the sum of each level is all equal to or greater than cn, so the time complexity will be $O(n \lg n)$. Also, in this case, even if we divide it 1 to 99, it will show a result of $O(n \lg n)$, so we can say that the time complexity of quicksort is

$$T(n) = O(n \lg n) \tag{5}$$

in the average case.

3 ADVANCED SORTING ALGORITHMS

3.1 Library Sort

Algorithm 1 LibrarySort(input)

```
n ← length(input)
Initialize sortedArr of size 2n + 1, filled with GAP
sortedArr[1] ← input[0]
for i ← 1 to ⌊log2(n − 1)⌋ + 1 do
    Rebalance(sortedArr, 1, 2^(i−1))
    for j ← 2^(i−1) to min(2^i, n) − 1 do
        pos ← BinarySearch(sortedArr, 1, 2^i, input[j])
        Insert(sortedArr, pos, input[j], 2^i)
    end for
end for
return all elements in sortedArr that are not GAP
```

Algorithm 2 BinarySearch(A, begin, end, value)

```
left ← begin,    right ← end
while left ≤ right do
    mid ← ⌊(left + right)/2⌋
    if A[mid] = GAP then
        /* If GAP encountered, adjust left/right based on nearby
        non-GAP */
    else if A[mid] < value then
        left ← mid + 1
    else
        right ← mid − 1
    end if
end while
return left
```

Algorithm 3 Rebalance(A, begin, end)

```
r ← end,    w ← 2 · end
while r ≥ begin do
    A[w] ← A[r]
    A[w − 1] ← GAP
    r ← r − 1,    w ← w − 2
end while
```

Algorithm 4 Insert(A, pos, value, end)

if $pos > end$ **then**
  /** Shift elements left from $pos - 1$ until a GAP is found, then insert *value* at the last valid position **/
  return
end if
if $A[pos]$ == GAP **then**
  /** Directly insert *value* at position *pos* **/
  $A[pos] \leftarrow value$
  return
end if
/** Otherwise, search for the first GAP starting at index *pos* **/
Let i be the smallest index $\geq pos$ such that $A[i]$ is GAP.
if $i > end$ **then**
  /** Shift elements left from *pos* until a GAP is found, then insert *value* at position $pos - 1$ **/
  return
else
  /** Shift elements right from index i down to $pos + 1$, then insert *value* at *pos* **/
  return
end if

3.2 Tim Sort

Algorithm 5 TimSort(input)

$n \leftarrow$ length(input)
$run \leftarrow 32$
$i \leftarrow 0$ {/* Sort each subarray of size run using insertion sort */}
while $i < n$ **do**
  InsertionSort(input, i, $\min(i + run - 1,\ n - 1)$)
  $i \leftarrow i + run$
end while
while $size < n$ **do**
  while $left < n$ **do**
    $mid \leftarrow left + size - 1$
    $right \leftarrow \min(left + 2 \cdot size - 1,\ n - 1)$
    if $mid < right$ **then**
      MergeSort.merge(input, $left$, mid, $right$)
    end if
    $left \leftarrow left + 2 \cdot size$
  end while
  $left \leftarrow 0$
  $size \leftarrow 2 \cdot size$
end while

[6]Library sort is a variation of insertion sort. It works similarly to insertion sort, but inserts gaps between each element, resulting in an overall array size of $(1+\epsilon)n$. Additionally, after inserting as many elements as already exist in the array, rebalancing is performed every $\lg n$ rounds. The insertion position is found using binary search by scanning until a gap is encountered. At the end of each round, gaps are inserted between elements to rebalance the array. Therefore, the algorithm consists of binary search, insertion, and rebalancing.

Advantages: By pre-allocating empty spaces between the array elements, unnecessary shifting during insertion can be reduced. This brings the average sorting time closer to $O(n \lg n)$. Since the algorithm is based on the simple idea of adding gaps to insertion sort, it is easy to understand. Moreover, if the input data is nearly sorted, similar to insertion sort, the overall sorting time can be greatly reduced.

Disadvantages: Library sort requires additional space, about $\epsilon \cdot n$ extra size compared to insertion sort. Therefore, it is less efficient than in-place sorting algorithms when memory is constrained. The rebalancing step, which redistributes the elements and inserts gaps, incurs additional overhead. This extra cost depends on the value of ϵ. Although the algorithm exhibits average-case performance of $O(n \lg n)$, in the worst-case scenario — when gaps are not efficiently utilized — its performance may degrade to $O(n^2)$. Hence, it is less stable compared to more reliable algorithms such as mergesort or heapsort.

[7]Tim sort is a hybrid sorting algorithm derived from merge sort and insertion sort, which leverages the already partially sorted runs present in the overall array. When a run is shorter than a prescribed minimum length, insertion sort is used to extend that run. These runs are merged according to specified criteria using a stack to maintain roughly balanced merges. This algorithm exhibits good adaptability because it efficiently exploits the pre-existing sorted order in the data. Therefore, in the worst and average situations, it shows $O(n \lg n)$, and in the best situation, it shows $O(n)$, since it is an ordered situation.

Advantages: Since the algorithm takes advantage of the already sorted portions, the longer these segments are, the closer the overall sorting time can approach nearly linear performance. Furthermore, as Tim sort is widely used as a sorting algorithm in real systems such as Python and Java, stability is ensured. It adaptively handles a variety of inputs by applying insertion sort to small arrays and merge sort to larger arrays.

Disadvantages: Tim sort is structurally more complex than traditional algorithms, and features such as the galloping mode—as implemented in Python—are difficult to implement and understand. In addition, the merge process requires extra memory, which may make it less suitable for environments with limited memory. Moreover, in the case of a completely random array where there is almost no existing sorted order, although the time complexity remains the same, the increased constant factors can reduce efficiency. Finally, since the run size is determined in advance, the optimal parameter may vary from system to system.

3.3 Cocktail Shaker Sort

Algorithm 6 CocktailShakerSort(input)

```
swapped ← true
start ← 0, end ← length(input) − 1
while swapped do
    swapped ← false
    for i = start to end − 1 do
        if input[i] > input[i + 1] then
            Swap(input[i], input[i + 1])
            swapped ← true
        end if
    end for
    end ← end − 1
    if ¬swapped then
        break
    end if
    swapped ← false
    for i = end downto start + 1 do
        if input[i] < input[i − 1] then
            Swap(input[i], input[i − 1])
            swapped ← true
        end if
    end for
    start ← start + 1
end while
```

[4]Cocktail Shaker Sort is a bidirectional implementation of bubble sort. While bubble sort compares and swaps numbers in one direction—from the beginning to the end—Cocktail Shaker Sort repeatedly traverses the array in both directions, gradually sorting it. In the forward pass, the algorithm repeatedly compares adjacent numbers, similar to standard bubble sort, moving the largest element to the very end of the array. Then, in the backward pass, excluding the element that was just moved, it traverses from the end to the beginning and moves the smallest element to the very front. This process is repeated until the entire array is sorted. The absolute time complexity is the same as bubble sort. In the best case, it is $O(n)$, but in the worst, average case, it is $O(\lg n)$.

Advantages: Since Cocktail Shaker Sort can be implemented with a very simple logic similar to bubble sort, it is easy to implement. Moreover, if optimized to remember the positions where swaps actually occur, it can achieve faster performance than the basic bubble sort.

Disadvantages: Nevertheless, on average it performs slower than other sorting algorithms such as merge sort or quick sort, which makes it less practical for real-world applications. Consequently, for large datasets its speed is even further reduced compared to alternative algorithms.

3.4 Comb Sort

[3]Comb sort, like Cocktail Shaker Sort, is a sorting algorithm derived from bubble sort. It repeatedly compares and swaps values between two indices that are separated by a gap, similar to bubble sort, while using the idea of gradually reducing this gap by dividing it by a shrink factor after each pass. In bubble sort, small values at the end of the list (often referred to as the "tortoise" problem) can slow down the sorting process significantly. Comb sort eliminates this issue by continuously shrinking the gap and finally performing a bubble sort on an array that is largely sorted. If no swaps occur during a pass, the algorithm concludes that the array is sorted. It is similar to CocktailShaker sort in that it improves on bubble sort, but the worst case is the same as $O(n^2)$, the best case is $\Theta(n \lg n)$, and the average case is $\Omega(n^2/2^p)$ when p is an increasing number.

Algorithm 7 CombSort(input)

```
gap ← length(input)
shrink ← 1.3
sorted ← false
while ¬sorted do
    gap ← integer(gap/shrink)
    if gap > 1 then
        sorted ← false
    else
        gap ← 1
        sorted ← true
    end if
    i ← 0
    while i + gap < length(input) do
        if input[i] > input[i + gap] then
            Swap(input[i], input[i + gap])
            sorted ← false
        end if
        i ← i + 1
    end while
end while
```

Advantages: By addressing the "tortoise" problem present in bubble sort, comb sort greatly improves efficiency—under optimal conditions, its performance can approach $O(n \lg n)$. Like bubble sort and cocktail shaker sort, it is simple to implement and works well when the input data is partially sorted or when dealing with small datasets.

Disadvantages: The worst-case time complexity of comb sort remains $O(n^2)$, which makes it less effective for sorting large datasets compared to other algorithms. Additionally, since the performance is sensitive to the choice of the shrink factor, a careful selection of this parameter is necessary.

3.5 Tournament Sort

Algorithm 8 MATCH(input, p1, p2)

```
if input[p1] ≤ input[p2] then
    return p1
else
    return p2
end if
```

[8]Tournament sort is a sorting algorithm that uses a modified structure similar to heap sort, functioning like a tournament in which the winner of a sports competition is determined. It constructs a binary tree in which every element from the input is

Algorithm 9 BUILD_TREE(input)

```
n ← length(input)
Create an array tree of size 2 × n
for i ← 0 to n − 1 do
    tree[n + i] ← i
end for
for i ← n − 1 downto 1 do
    tree[i] ← MATCH(input, tree[2i], tree[2i + 1])
end for
return tree
```

Algorithm 10 UPDATE_TREE(input, tree, p)

```
tree_idx ← p + length(input)
while tree_idx/2 ≥ 1 do
    tree_idx ← tree_idx/2
    tree[tree_idx] ← MATCH(input, tree[2 · tree_idx], tree[2 · tree_idx + 1])
end while
```

Algorithm 11 TOURNAMENT_SORT(input)

```
tree ← BUILD_TREE(input)
Create an array result of size length(input)
for i ← 0 to length(input) − 1 do
    min_idx ← tree[1]
    result[i] ← input[min_idx]
    Set input[min_idx] ← INF
    UPDATE_TREE(input, tree, min_idx)
end for
input ← result
```

placed as a leaf node, and the internal nodes are set to store the smaller value among their two child nodes. When the tournament is conducted from the leaf nodes upward to the root, the root node holds the minimum value. This minimum value is then replaced with infinity to exclude it from further competition, and the tree is updated by backtracking. By repeating the tournament on the updated tree, the elements stored in the resulting array eventually become sorted. Also, it shows $O(n \lg n)$ in the worst and average cases.

Advantages: Since tournament sort operates similarly to heap sort, its logical structure is easy to understand and implement. Moreover, with both worst-case and average-case performance of $O(n \lg n)$, it yields reliable results even in the worst-case scenarios. Additionally, because the minimum value can be obtained in $O(1)$ time in each iteration, this feature is beneficial when such fast retrieval is required.

Disadvantages: Tournament sort requires an additional array to store the tournament tree structure (approximately twice the size of the input), making it a non in-place algorithm and reducing memory efficiency. Although its time complexity is $O(n \lg n)$ like other efficient algorithms such as quicksort, the constant factors involved may be higher, which can result in lower efficiency when processing large datasets compared to other advanced sorting algorithms.

3.6 Introsort

Algorithm 12 IntroSort(input)

```
max_depth ← 2 · int( log2 (length(input)) )
call QuickSort.sort_intro_version(input, 0, length(input) − 1, max_depth)
call InsertionSort.sort(input, 0, length(input) − 1)
```

Algorithm 13 QuickSort.sort_intro_version(input, p, r, max_depth)

```
if p < r then
    if max_depth = 0 then
        call HeapSort.sort_intro_version(input, r − p + 1)
        return
    end if
    q ← partition(input, p, r)
    QuickSort.sort_intro_version(input, p, q − 1, max_depth − 1)

    QuickSort.sort_intro_version(input, q + 1, r, max_depth − 1)
end if
```

[5]IntroSort is a hybrid sorting algorithm that combines the best features of three sorting algorithms: Quick Sort, Heap Sort, and Insertion Sort. It begins with Quick Sort and, to prevent excessive recursion depth due to poor pivot selections, sets a maximum recursion depth (e.g., $2 \cdot \log_2(n)$). When this depth limit is reached, the algorithm switches to Heap Sort, which guarantees a worst-case time complexity of $O(n \lg n)$ and thereby ensures overall stability. Finally, in the later stages of the sort, Insertion Sort is employed because it performs very quickly on small arrays, making it well-suited for the final cleanup phase. Therefore, the best, worst, and average all show $O(n \lg n)$.

Advantages: Since IntroSort is designed by combining only the best aspects of these three algorithms, it effectively avoids worst-case scenarios and guarantees a time complexity of $O(n \lg n)$ in both the worst-case and average-case, thus providing robust stability. Moreover, if the input data is already nearly sorted, the efficiency of Insertion Sort can further enhance performance.

Disadvantages: The implementation of IntroSort is more complex than that of simpler sorting algorithms, and incorporating additional optimization techniques can further increase this complexity. The algorithm's performance is sensitive to the chosen value of the recursion depth limit parameter, so a careful selection is required. Additionally, because it combines three different sorting approaches, it may actually exhibit decreased performance on datasets with a small number of elements.

4 EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Dataset Generation and Experimental Setup

For our experiments, we systematically evaluated the performance of various sorting algorithms by generating input datasets with diverse characteristics and configuring a robust experimental environment.

Dataset Generation:
The input data is categorized into several types:

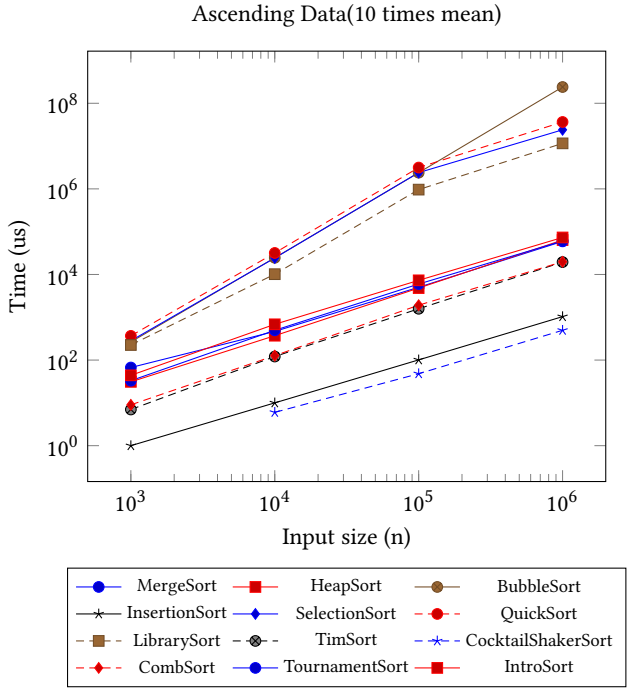
- **Sorted Data:** Data that is pre-sorted in ascending order as well as descending order.
- **Random Data:** Data generated in a completely random order.
- **Partially Sorted Data:** Data where a certain percentage of the elements are already sorted, enabling analysis of the impact of initial order on algorithm performance.

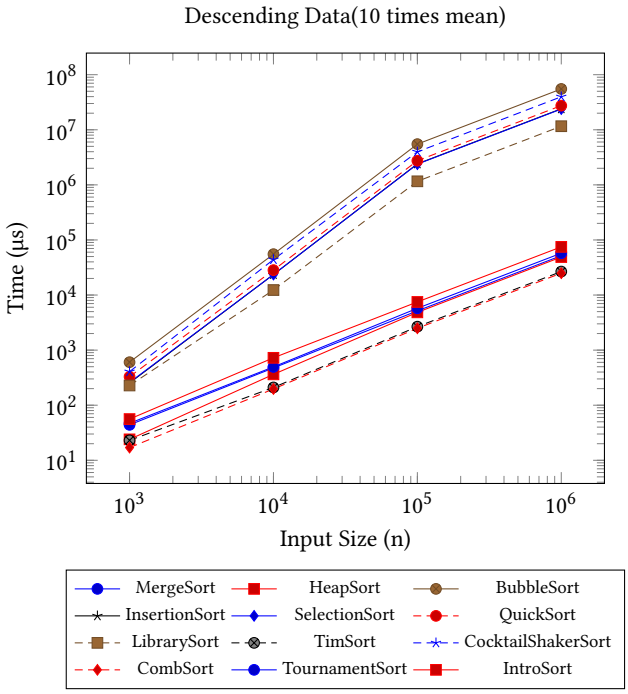
Each dataset size ranges from 1,000 to 1,000,000 elements, and each test case is executed at least 10 times to compute precise average running times.

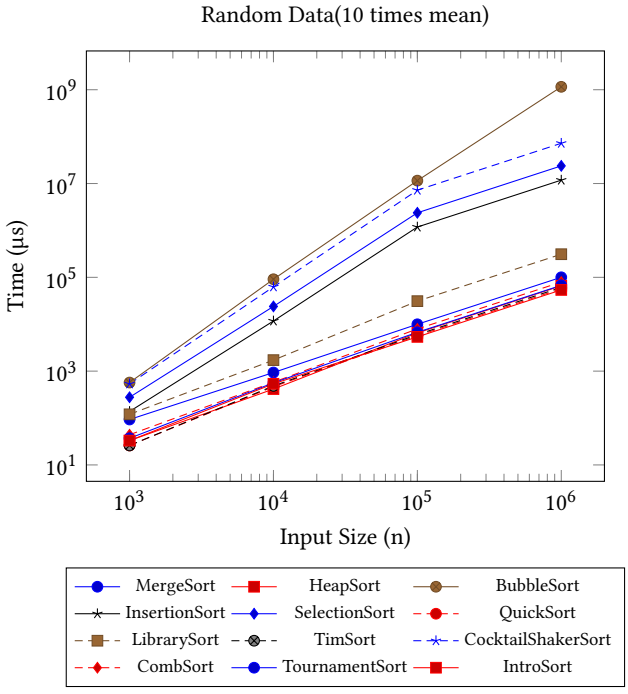
Experimental Environment:
The experiments were conducted using C++ implementations of the sorting algorithms. Execution times were measured under different compiler optimization settings and hardware configurations. Additionally, each algorithm's performance was evaluated based on criteria including execution time, memory consumption, and sorting stability.

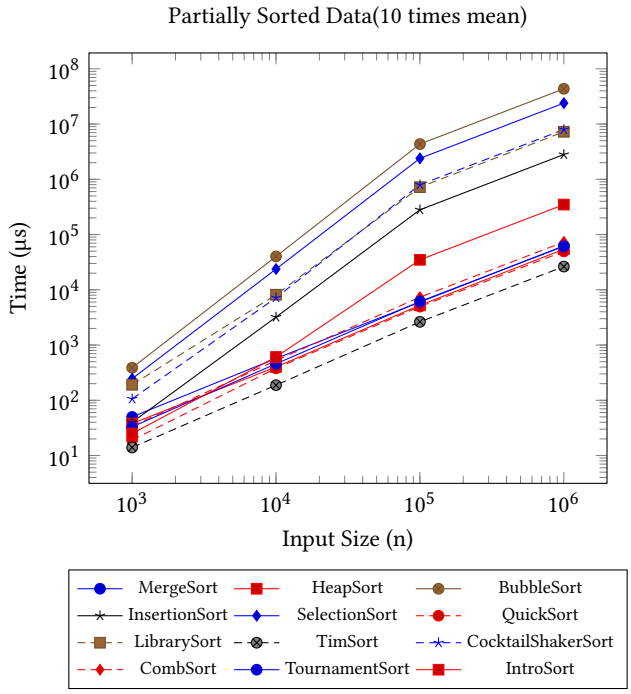
This experimental setup, combined with the diverse dataset generation, enables an in-depth analysis of the inherent characteristics and real-world performance of each sorting algorithm.

4.2 Performance Comparison









4.3 Analysis and Discussion

Table 1: Assessment of Sorting Algorithms

Algorithm	Execution Time	Memory Consumption	Stability
MergeSort	$O(n \log n)$ (avg & worst)	n	Yes
HeapSort	$O(n \log n)$ (avg & worst)	1	No
BubbleSort	$O(n^2)$	1	Yes
InsertionSort	Best: $O(n)$, Worst: $O(n^2)$	1	Yes
SelectionSort	$O(n^2)$	1	No
QuickSort	Avg: $O(n \log n)$, Worst: $O(n^2)$	AVG: $\log n$, Worst: n	No
LibrarySort	$O(n \log n)$	n	Yes
TimSort	Worst: $O(n \log n)$, Nearly sorted: $O(n)$	n	Yes
CocktailShakerSort	$O(n^2)$	1	Yes
CombSort	$O(n^2)$	1	No
TournamentSort	$O(n \log n)$	n	No
IntroSort	$O(n \log n)$	$\log n$	No

4.3.1 *Performance Patterns Based on Data Characteristics.*

- **Ascending sorted data:** InsertionSort and CocktailShakerSort show excellent performance because they have $O(n)$ time complexity for already sorted data.
- **Descending data:** Efficient algorithms (MergeSort, HeapSort, CombSort, TimSort) maintain consistent performance, while QuickSort shows performance close to its worst-case $O(n^2)$ time complexity.
- **Random data:** QuickSort, IntroSort, and HeapSort operate most efficiently.
- **Partially sorted data:** TimSort demonstrates very fast performance because it was originally designed with an algorithm that efficiently utilizes runs in partially sorted data. Additionally, InsertionSort and CocktailShakerSort show slightly improved performance compared to random data.

4.3.2 *Scalability with Input Size.*

- **$O(n^2)$: $O(n^2)$** algorithms such as BubbleSort, SelectionSort, and InsertionSort show dramatic performance degradation when input size exceeds 100,000.
- **$O(n \log n)$:** These algorithms maintain relatively consistent performance even with large datasets.
- **Hybrid algorithms:** Hybrid algorithms like TimSort and IntroSort demonstrate excellent adaptability across various input sizes, especially with partially sorted data.

4.3.3 *Algorithm-Specific Characteristics.*

- **Performance sensitivity based on sorting state:** InsertionSort's performance varies greatly depending on the sorting state of the data—very fast with ascending order, very slow with descending order.
- **Consistency:** HeapSort, MergeSort, and TournamentSort show the most consistent performance across all datasets, making them the most suitable algorithms for general situations.
- **Efficiency of hybrid algorithms:** Hybrid algorithms like TimSort and IntroSort demonstrate strong performance in various situations because they combine only the advantages of multiple algorithms.

4.3.4 *Practical Application Considerations.*

- **Performance in various situations:**
  - **Small-scale data ($n < 20$):** Just as IntroSort uses InsertionSort for small datasets, InsertionSort is simple to implement and efficient for small datasets.
  - **Large-scale data:** For large-scale data, which needs to be cache-friendly and generally have good performance, TimSort, IntroSort, and MergeSort provide the most reliable performance.
  - **Memory-constrained environments:** Among the algorithms we studied, HeapSort is the fastest algorithm that provides stable performance without additional memory usage.
  - **Already partially sorted data:** TimSort can demonstrate fast performance by utilizing runs, so it performs better when partially sorted lengths are longer.
- **Performance trade-offs:**
  - When considering the balance between time efficiency and memory usage, HeapSort can guarantee reasonably good results in both space efficiency and time efficiency.
  - However, when stability is important, MergeSort, TimSort, and LibrarySort could be good choices.

4.3.5 *Conclusion.*

In real-world applications, modern algorithms like TimSort and IntroSort show the best performance. However, since these are algorithms that combine several existing sorting algorithms such as InsertionSort, HeapSort, and MergeSort, a deeper understanding of these fundamentals would be even more necessary to develop better performing sorting algorithms in the future. Also, since the optimal algorithm varies depending on whether the dataset characteristics are sorted, completely random, or partially sorted, and also changes depending on the memory environment and stability requirements, proper analysis of these algorithms is essential for appropriate utilization in the right context.

References

[1] Seonguk Choi. 2025. GitHub Repository for Sorting Algorithms. https://github.com/CSUISFINE/algorithm_project_1. Accessed: April 15, 2025.

[2] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[3] Wikipedia contributors. 2024. Comb sort — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Comb_sort&oldid=1230345145 [Online; accessed 15-April-2025].

[4] Wikipedia contributors. 2025. Cocktail shaker sort — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Cocktail_shaker_sort&oldid=1267360789 [Online; accessed 15-April-2025].

[5] Wikipedia contributors. 2025. Introsort — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Introsort&oldid=1274700328 [Online; accessed 15-April-2025].

[6] Wikipedia contributors. 2025. Library sort — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Library_sort&oldid=1270400628 [Online; accessed 15-April-2025].

[7] Wikipedia contributors. 2025. Timsort — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Timsort&oldid=1285075086 [Online; accessed 15-April-2025].

[8] Wikipedia contributors. 2025. Tournament sort — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Tournament_sort&oldid=1268246236 [Online; accessed 15-April-2025].