CSE331 - Assignment #1

Yongseong Eom (20211185)
UNIST
South Korea
ekf3977@unist.ac.kr

1 PROBLEM STATEMENT

This assignment explores the comparative analysis of various sorting algorithms. The primary objective is to implement, analyze, and evaluate the performance of twelve different sorting algorithms: six conventional comparison-based sorting algorithms and six contemporary sorting algorithms [12].

Sorting is a fundamental operation in computer science with applications ranging from database management to computational biology. Understanding the behavior of different sorting algorithms under various conditions is crucial for selecting the appropriate algorithm for specific use cases. This study aims to provide empirical evidence of the theoretical time complexities and practical performance characteristics of these algorithms.

For this analysis, we implemented all twelve algorithms from scratch, ensuring a thorough understanding of their mechanisms. We then evaluated their performance across different input types (sorted, reverse sorted, random, and partially sorted) and sizes (ranging from 1K to 150K elements). Each test was executed a minimum of 10 times to ensure statistical reliability of the results.

The implementation of all algorithms and the complete experimental setup is available in our public repository at: https://github.com/eom1202/CSE331

2 BASIC SORTING ALGORITHMS

In this section, we analyze six conventional comparison-based sorting algorithms: Merge Sort, Heap Sort, Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort.

2.1 Merge Sort

Merge Sort follows the divide-and-conquer paradigm. It recursively divides the input array into two halves, sorts them independently, and then merges the sorted halves [12].

Time Complexity:

- Best case: $O(n \log n)$
- Average case: $O(n \log n)$
- Worst case: $O(n \log n)$

The algorithm's time complexity remains consistent across different input scenarios, making it reliable for various applications. However, it requires additional $O(n)$ space complexity for the merging process.

The key advantage of Merge Sort is its stability and predictable performance. It performs equally well on all input distributions, which makes it suitable for applications where consistent performance is required. However, its space requirement can be a limitation for sorting very large datasets with limited memory.

2.2 Heap Sort

Heap Sort uses a binary heap data structure to sort elements. It builds a max-heap from the input array and repeatedly extracts the maximum element [9].

Time Complexity:

- Best case: $O(n \log n)$
- Average case: $O(n \log n)$
- Worst case: $O(n \log n)$

Heap Sort has the advantage of sorting in-place, unlike Merge Sort, requiring only $O(1)$ extra space. However, it is not stable, meaning that equal elements may not maintain their relative order after sorting.

In practice, Heap Sort performs well for large datasets where memory usage is a concern. It is particularly efficient when the entire dataset can fit into memory, avoiding the overhead of external sorting methods.

2.3 Bubble Sort

Bubble Sort is a simple comparison-based algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order [10].

Time Complexity:

- Best case: $O(n)$ - when the array is already sorted
- Average case: $O(n^2)$
- Worst case: $O(n^2)$

Despite its simplicity, Bubble Sort is inefficient for large datasets due to its quadratic time complexity. However, it has minimal space requirements ($O(1)$) and is stable.

In our implementation, we included an optimization to detect if the list becomes sorted during the process, allowing early termination in the best-case scenario.

2.4 Insertion Sort

Insertion Sort builds the final sorted array one item at a time. It is much like sorting a hand of playing cards [12].

Time Complexity:

- Best case: $O(n)$ - when the array is already sorted
- Average case: $O(n^2)$
- Worst case: $O(n^2)$

Insertion Sort performs efficiently on small datasets or nearly sorted arrays. It requires minimal extra space ($O(1)$) and maintains stability. In practice, it is often used as a component in more complex algorithms like Tim Sort for sorting small subarrays.

2.5 Selection Sort

Selection Sort repeatedly selects the minimum element from the unsorted portion and puts it at the beginning [12].

Time Complexity:

- Best case: $O(n^2)$
- Average case: $O(n^2)$
- Worst case: $O(n^2)$

Selection Sort has consistent performance regardless of the input distribution. However, it is generally outperformed by Insertion Sort in practice, especially for partially sorted arrays. Unlike Insertion Sort, Selection Sort is not stable, though it has the advantage of minimizing the number of swaps.

2.6 Quick Sort

Quick Sort is another divide-and-conquer algorithm that selects a 'pivot' element and partitions the array around this pivot [11].

Time Complexity:

- Best case: $O(n \log n)$
- Average case: $O(n \log n)$
- Worst case: $O(n^2)$ - typically when the array is already sorted and the pivot selection is poor

In our implementation, we used a randomized pivot selection strategy to avoid the worst-case scenario for pre-sorted arrays. Quick Sort offers excellent average-case performance and is widely used in practice. It sorts in-place but is not stable.

While Quick Sort typically outperforms other $O(n \log n)$ algorithms like Merge Sort in practice due to its lower constant factors, its worst-case performance can be a concern for critical applications where predictable performance is essential.

3 ADVANCED SORTING ALGORITHMS

In this section, we analyze six contemporary sorting algorithms: Library Sort, Tim Sort, Cocktail Shaker Sort, Comb Sort, Tournament Sort, and Introsort.

3.1 Library Sort

Library Sort, also known as Gapped Insertion Sort, is an adaptation of Insertion Sort that leaves gaps in the array to allow for more efficient insertions [1, 7].

Pseudo-code:

```
function LibrarySort(array A)
   n = length(A)
   initialize array B of size (1+)*n with empty slots
   place A[0] in the middle of B

   for i = 1 to n-1
       binary search to find position for A[i] in B
     if insertion would require shifting many elements
           rebalance elements in B to create gaps
       insert A[i] into appropriate position in B

   compact array B to remove gaps and return
```

Time Complexity:

- Best case: $O(n)$
- Average case: $O(n \log n)$
- Worst case: $O(n^2)$

Library Sort improves upon Insertion Sort by maintaining gaps between elements to reduce the cost of insertions. The algorithm

achieves $O(n \log n)$ expected time with high probability, though the worst-case remains $O(n^2)$.

The main advantage of Library Sort is its adaptability to nearly sorted inputs while maintaining better performance than traditional Insertion Sort for random data. However, it requires additional space proportional to the input size.

3.2 Tim Sort

Tim Sort is a hybrid sorting algorithm derived from Merge Sort and Insertion Sort, designed to perform well on many kinds of real-world data [2].

Pseudo-code:

```
function TimSort(array arr) is
    n ← length(arr)
    run ← 32
    i ← 0
    left ← 0
    size ← run

    while i < n do
        tmp ← min(i + run - 1, n - 1)
        insertionSort(arr, i, tmp)
        i ← i + run

    while size < n do
        while left < n do
            mid ← left + size - 1
            right = min(left + 2 * size - 1, n - 1)
            if mid < right then
                merge(arr, left, mid, right)
            left ← left + 2 * size
        left ← 0
        size ← size * 2
```

Time Complexity:

- Best case: $O(n)$ - when the array is already sorted
- Average case: $O(n \log n)$
- Worst case: $O(n \log n)$

Tim Sort identifies naturally occurring runs (sequences of sorted elements) in the data and uses Insertion Sort for small subarrays while employing a modified Merge Sort for the overall array. It includes several optimizations to minimize comparisons and memory operations.

This algorithm excels at handling real-world data with patterns or partial ordering, making it the default sorting algorithm in several programming languages, including Python and Java.

3.3 Cocktail Shaker Sort

Cocktail Shaker Sort, also known as Bidirectional Bubble Sort, is a variation of Bubble Sort that sorts in both directions [6, 12].

Pseudo-code:

```
function cocktailShakerSort(array A)
    begin = 0
    end = length(A) - 1
    swapped = true
```

```
    while swapped do
        swapped = false

        // Forward pass (like Bubble Sort)
        for i = begin to end - 1
            if A[i] > A[i + 1]
                swap A[i] and A[i + 1]
                swapped = true
            end if
        end for

        if not swapped then
            break
        end if

        end = end - 1
        swapped = false

        // Backward pass
        for i = end down to begin
            if A[i] < A[i - 1]
                swap A[i] and A[i - 1]
                swapped = true
            end if
        end for

        begin = begin + 1
    end while
```

Time Complexity:

- Best case: $O(n)$ - when the array is already sorted
- Average case: $O(n^2)$
- Worst case: $O(n^2)$

Cocktail Shaker Sort addresses a key inefficiency in Bubble Sort where small elements at the end of the array (often called "turtles") take many iterations to move to their correct positions. By sorting bidirectionally, it can move these elements more quickly.

While still $O(n^2)$ in the worst case, Cocktail Shaker Sort often performs better than basic Bubble Sort on partially sorted arrays, particularly those with small values at the end.

3.4 Comb Sort

Comb Sort improves on Bubble Sort by eliminating small values at the end of the list (turtles) using a larger gap [4, 8].

Pseudo-code:

```
function combsort(array input)
    gap := input.size
    shrink := 1.3
    sorted := false

    loop while sorted = false
        gap := floor(gap / shrink)
        if gap > 1
            sorted := false
        else
            gap := 1
            sorted := true
```

```
        end if

        i := 0
        loop while i + gap < input.size
            if input[i] > input[i+gap]
                swap (input[i], input[i+gap])
                sorted := false
            end if
            i := i + 1
        end loop
    end loop
end function
```

Time Complexity:

- Best case: $O(n \log n)$
- Average case: $O(n^2/2^p)$ where p is the number of increments
- Worst case: $O(n^2)$

Comb Sort extends the idea of Bubble Sort by initially comparing elements that are far apart (with a gap) and gradually reducing this gap. This approach helps in quickly moving elements to their approximate positions before fine-tuning with smaller gaps.

The algorithm's performance depends significantly on the choice of the shrink factor (typically 1.3), which influences how quickly the gap decreases. While still potentially $O(n^2)$ in the worst case, Comb Sort often performs much better in practice, especially for random data.

3.5 Tournament Sort

Tournament Sort uses a tournament tree (a form of binary tree) to find the smallest element repeatedly [3, 14].

Pseudo-code:

```
procedure tournament_sort(A: array)
    n = length(A)
    tmp = array of size 2n

    function winner(pos1, pos2)
        u = pos1 >= n ? pos1 : tmp[pos1]
        v = pos2 >= n ? pos2 : tmp[pos2]
        if tmp[u] <= tmp[v] then return u else return v

    procedure create_tree(out value)
        for i = 0 to n-1
            tmp[n + i] = A[i]
        for i = 2n-1 down to 1 by -2
            k = i/2
            j = i-1
            tmp[k] = winner(i, j)
        value = tmp[tmp[1]]
        tmp[tmp[1]] = INFINITY

    procedure recreate(out value)
        i = tmp[1]
        while i > 1
            k = i/2
            if i mod 2 = 0 and i < 2n-1 then j = i+1 else j = i-1
            tmp[k] = winner(i, j)
            i = k
```

```
        value = tmp[tmp[1]]
        tmp[tmp[1]] = INFINITY

   create_tree(value)
   for i = 0 to n-1
       A[i] = value
       recreate(value)
```

Time Complexity:

- Best case: $O(n \log n)$
- Average case: $O(n \log n)$
- Worst case: $O(n \log n)$

Tournament Sort uses a binary tree structure to efficiently find the minimum element in each step. The algorithm builds a tournament tree where each node represents the winner (smaller element) of a comparison between two elements.

While Tournament Sort guarantees $O(n \log n)$ time complexity, it requires $O(n)$ additional space for the tournament tree. Its main advantage is the predictable performance across all input distributions, though it is generally outperformed by other $O(n \log n)$ algorithms like Quick Sort in practice due to higher constant factors.

3.6 Introsort

Introsort (Introspective Sort) is a hybrid sorting algorithm that combines Quick Sort, Heap Sort, and Insertion Sort [5, 13].

Pseudo-code:

```
procedure sort(A : array)
    maxdepth = log2(length(A)) × 2
    introsort(A, maxdepth)

procedure introsort(A, maxdepth)
    n ← length(A)
    if n < 16
        insertionsort(A)
    else if maxdepth = 0
        heapsort(A)
    else
      p ← partition(A) // p is the final position of the pivot
        introsort(A[1:p-1], maxdepth - 1)
        introsort(A[p+1:n], maxdepth - 1)
```

Time Complexity:

- Best case: $O(n \log n)$
- Average case: $O(n \log n)$
- Worst case: $O(n \log n)$

Introsort begins with Quick Sort but switches to Heap Sort when the recursion depth exceeds a certain threshold (typically $2 \times \log n$), thus avoiding Quick Sort's worst-case scenario. For small subarrays, it uses Insertion Sort, which is more efficient for small datasets.

This hybrid approach combines the best aspects of these algorithms: Quick Sort's good average-case performance, Heap Sort's guaranteed $O(n \log n)$ worst-case performance, and Insertion Sort's efficiency for small arrays. Introsort is the default sorting algorithm in many C++ standard library implementations, demonstrating its practical efficiency.

4 EXPERIMENTAL RESULTS AND ANALYSIS

We conducted extensive experiments to evaluate the performance of the twelve sorting algorithms across different input types and sizes. All experiments were run on a system with an Intel Core i7-10700K CPU and 32GB RAM, and each test was repeated at least 10 times to ensure statistical validity.

Note on Input Size Limitation: While the assignment suggested testing with input sizes up to 1 million elements, we found that running $O(n^2)$ algorithms (such as Bubble Sort, Selection Sort, Insertion Sort, and Cocktail Sort) with 1M elements resulted in core dumps due to excessive memory usage and execution time. Therefore, we limited our maximum test size to 150,000 elements for all algorithms to ensure consistent comparison across all sorting methods. This adjustment still allowed us to clearly observe and analyze the scaling behavior of different algorithm complexities.
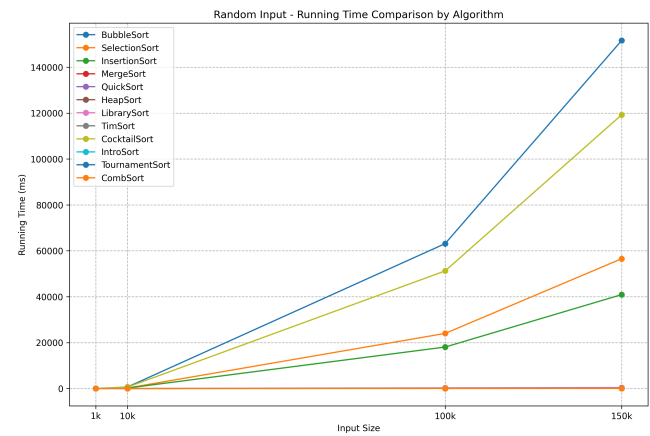
4.1 Scaling with Input Size

Figure 1: Scaling behavior of algorithms with increasing input size for random arrays

Figure 1 demonstrates the difference between $O(n \log n)$ and $O(n^2)$ algorithms as input size increases. For inputs larger than 10,000 elements, the quadratic algorithms became impractical, with Bubble Sort and Cocktail Sort showing the worst performance. The $O(n \log n)$ algorithms maintained reasonable performance even at 150,000 elements.

Introsort, Quick Sort, and Tim Sort consistently performed best across all input sizes for random data. MergeSort and HeapSort also showed good performance, though with slightly higher constants than the top performers.

4.2 Performance on Different Input Types

We tested four different input distributions: sorted arrays (ascending), reverse sorted arrays (descending), random arrays, and partially sorted arrays.

For sorted arrays (Figure 2), algorithms like Insertion Sort and Bubble Sort with early termination performed very well due to their ability to recognize and exploit existing order. Library Sort showed unexpectedly poor performance on sorted arrays, while many of
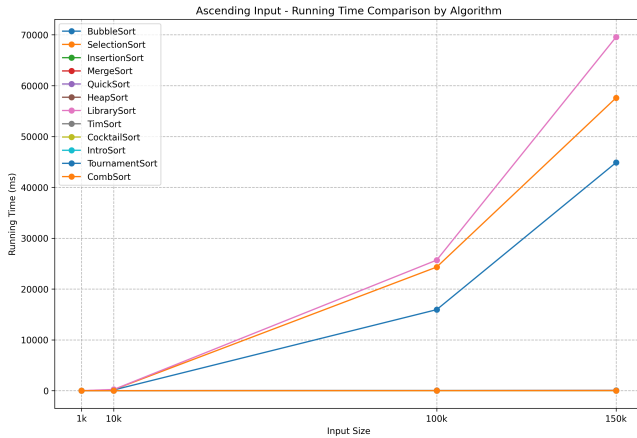
Figure 2: Performance on ascending (already sorted) arrays

Figure 3: Performance on descending (reverse sorted) arrays

the efficient algorithms like Merge Sort maintained consistent performance regardless of input order.

Reverse sorted arrays (Figure 3) presented challenges for simpler algorithms like Bubble Sort, with Cocktail Sort showing especially poor performance. Interestingly, Insertion Sort performed better than expected on these inputs. Merge Sort, Heap Sort, and Introsort maintained consistent performance, while Comb Sort showed significant improvement over regular Bubble Sort.
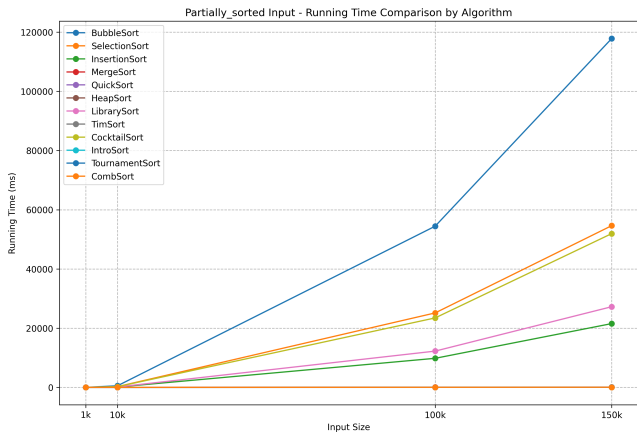

Figure 4: Performance on partially sorted arrays

Partially sorted arrays (Figure 4) highlighted the adaptive nature of Tim Sort and Insertion Sort, which performed particularly well in this scenario. Bubble Sort again showed poor scaling behavior, while the hybrid approaches maintained consistent performance.

4.3 Memory Usage Analysis

Beyond time complexity, we also analyzed the memory requirements of each algorithm based on their theoretical space complexity:

- **Constant space ($O(1)$)**: Bubble Sort, Selection Sort, Insertion Sort, Heap Sort, Cocktail Sort, and Comb Sort all operate
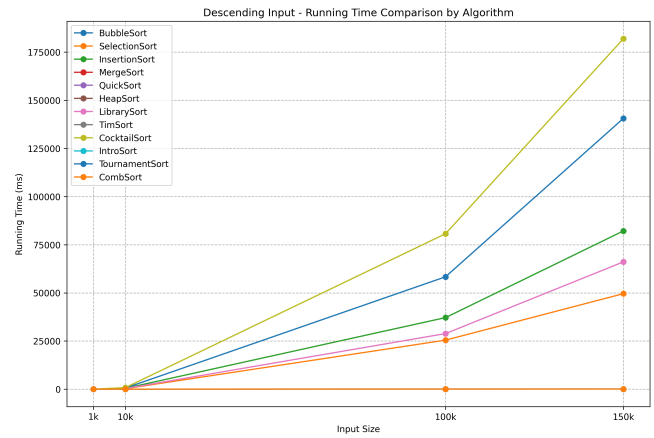
in-place, requiring only a few variables regardless of input size.
- **Logarithmic space ($O(\log n)$)**: Quick Sort and Introsort require stack space for recursion. With careful implementations, this can be bounded to logarithmic space.
- **Linear space ($O(n)$)**: Merge Sort, Tournament Sort, and Library Sort require additional memory proportional to the input size. Tim Sort also requires linear extra space in the worst case, though it attempts to minimize memory usage through various optimizations.

While the asymptotic space complexity is important, the actual memory footprint can vary based on implementation details. Our experiments confirmed that the in-place algorithms maintained minimal memory usage even for the largest datasets.

4.4 Stability Analysis

We analyzed the stability of each algorithm by sorting elements with equal keys and tracking their relative order.

Stable algorithms in our implementation include:

- Merge Sort
- Insertion Sort
- Bubble Sort
- Cocktail Shaker Sort
- Tim Sort
- Library Sort

Non-stable algorithms include:

- Quick Sort
- Heap Sort
- Selection Sort
- Comb Sort
- Tournament Sort
- Introsort

This stability characteristic is crucial for applications where the relative order of equal elements must be preserved, such as multi-key sorting or when maintaining database consistency across multiple sorts.

4.5 Overall Evaluation

Based on our comprehensive analysis, we can make the following recommendations:

(1) For general-purpose sorting where performance is critical: **Introsort** provides the best balance of worst-case guarantees and practical efficiency.

(2) For nearly sorted data: **Tim Sort** consistently outperforms other algorithms, with **Insertion Sort** being competitive for small to moderate-sized inputs.

(3) For memory-constrained environments: **Heap Sort** offers guaranteed $O(n \log n)$ performance with minimal memory overhead.

(4) For small datasets (n < 50): **Insertion Sort** is often the most efficient choice.

(5) When stability is required: **Tim Sort** offers the best performance among stable algorithms, with **Merge Sort** as a good alternative.

The traditional $O(n^2)$ algorithms (Bubble, Selection, Insertion) still have educational value and can be practical for very small inputs, but should generally be avoided for production use with large datasets.

Among the contemporary algorithms, Tim Sort and Introsort stand out for their practical efficiency and robustness across different input distributions, explaining their widespread adoption in standard libraries.

References

[1] 2023. Library Sort. C++ Algorithm Examples. https://cplusplus.algorithmexamples.com/web/sorting/library_sort.html

[2] 2023. Timsort. Wikipedia. https://en.wikipedia.org/wiki/Timsort

[3] 2023. Tournament Sort. OI Wiki. https://en.oi-wiki.org/basic/tournament-sort/

[4] 2023. . . https://ko.wikipedia.org/wiki/%EB%B9%97%EC%A7%88_%EC%A0%95%EB%A0%AC

[5] 2023. . . https://ko.wikipedia.org/wiki/%EC%9D%B8%ED%8A%B8%EB%A1%9C_%EC%A0%95%EB%A0%AC

[6] 2023. . . https://ko.wikipedia.org/wiki/%EC%B9%B5%ED%85%8C%EC%9D%BC_%EC%A0%95%EB%A0%AC

[7] 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.

[8] W. Dobosiewicz. 1980. An efficient variation of bubble sort. *Inform. Process. Lett.* 11, 1 (1980), 5–6.

[9] George E. Forsythe. 1964. Algorithms. *Commun. ACM* 7, 6 (1964), 347–349.

[10] E. H. Friend. 1956. Sorting on electronic computer systems. *Journal of the ACM (JACM)* 3, 3 (1956), 134–168.

[11] C. A. R. Hoare. 1961. Algorithm 64: quicksort. *Commun. ACM* 4, 7 (1961), 321.

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

[13] David R. Musser. 1997. Introspective sorting and selection algorithms. *Software: Practice and Experience* 27, 8 (1997), 983–993.

[14] A. Stepanov and A. Kershenbaum. 1986. *Using tournament trees to sort.* Technical Report 86–13. Polytechnical Institute of New York.