CSE331 - Assignment #1

Seungchan Choi (20231393)
UNIST
South Korea
unknown@unist.ac.kr

1 PROBLEM STATEMENT

Sorting is one of the most fundamental and extensively studied problems in computer science. Its importance stems from the fact that a large number of computational tasks — from data analysis and database queries to searching and scheduling — rely on efficient sorting as a critical subroutine. Despite its familiarity, discovering new or more efficient sorting methods remains a topic of ongoing research, particularly for large-scale datasets and specialized use cases. As data volumes continue to grow, refining and exploring advanced sorting techniques remains highly relevant, both in academic research and in practical applications.

With this motivation in mind, we present an assignment focusing on the implementation and analysis of various sorting algorithms. Specifically, we will:

- Implement and analyze **6 conventional sorting algorithms**.
- Implement and analyze **6 contemporary sorting algorithms**.
- Measure and compare the average running times for each algorithm with varying input datasets.

The goal is to deepen our understanding of how each algorithm performs under different conditions and data characteristics. Finally, this report will discuss both theoretical complexity and empirical results obtained through extensive experiments.

2 BASIC SORTING ALGORITHMS

This section covers **Merge Sort, Heap Sort, Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort**. These are some of the most fundamental comparison-based sorting algorithms in computer science.

2.1 Merge Sort

- **Algorithm Description**
  Merge Sort utilizes the *divide-and-conquer* method. The array is divided into smaller subarrays, each of which is recursively sorted. During the *merge* step, two sorted subarrays are combined by repeatedly comparing their front (leftmost) elements and placing the smaller one into the output array, thereby producing a globally sorted sequence.
- **Time Complexity : $O(n \log n)$**
  At each level of recursion, the array is split into two parts, leading to a recursion tree of height $O(\log n)$. Each level processes all n elements once (during merging), so the total running time is $O(n \log n)$ (same for the average case and the worst case).
- **Stability : stable**
  Merge Sort is commonly implemented as a **stable** algorithm. Specifically, if two values in the left and right subarrays are equal, the element from the left subarray is chosen first, preserving the relative order of identical elements from the original array.
- **Additional Memory Space**
  When merging two subarrays, temporary array is needed. So it uses $O(n)$ additional space.
- **Additional Experiments** 4.2
  - **Sentinel vs. No Sentinel:** Compare the performance and implementation complexity between using sentinel values (e.g., ∞) in the temporary arrays and checking array boundaries explicitly.
  - **Partition Groups (2-Way vs. 3-Way):** Investigate the effect of splitting the array into two parts (traditional Merge Sort) versus three or more partitions, examining both running time and code complexity.

2.2 Heap Sort

- **Algorithm Description**
  Heap Sort first constructs a *max-heap*, ensuring the largest element is placed at the root. Then, in each iteration, the root (the current largest) is swapped with the last element in the unsorted region, and the size of the unsorted portion is reduced by one. A call to `MaxHeapify` restores the max-heap property among the remaining elements.
- **Time Complexity : $O(n \log n)$**
  The procedure runs in $O(n)$, because for each node in height k, it takes h-k+1 steps in operation. Since there are 2^{k-1} nodes in height k, the total running time would be $\sum_{k=1}^{h}(2^{k-1} \cdot (h-k+1))$ which is $O(n)$ (more detailed proof in the textbook). In the sorting phase, we perform $(n-1)$ swaps and call (which costs $O(\log n)$) each time. Hence, the overall running time of Heap Sort is $O(n \log n)$.
- **Stability : not stable**
  Heap Sort is **not stable** by default, because elements with equal keys may get reordered when they are swapped within the heap.

2.3 Bubble Sort

- **Algorithm Description**
  Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order. In each outer iteration, the largest element among the unsorted portion "bubbles up" to its correct position at the end of the array.
- **Time Complexity : $O(n^2)$**
  In the worst and average case, the algorithm requires $O(n^2)$ comparisons and swaps. The nested loops each range up to n, so the total number of comparisons is proportional to $n \times (n-1)/2$.
- **Stability : stable**
  Bubble Sort is **stable**, because it only swaps adjacent elements when needed and thus does not reorder elements with equal keys.

2.4 Insertion Sort

- **Algorithm Description**
  Insertion Sort inserts one item by one into the sorted array. In each iteration, the current element (key) is compared with the elements before it, shifting one position to the right if it is larger than the key, until the correct spot for the key is found.
- **Time Complexity : $O(n^2)$**
  In the worst and average cases, each element may be compared with all elements to its left, leading to $\boldsymbol{O(n^2)}$ time complexity. In the best case (when the array is already sorted), the algorithm only performs $\boldsymbol{O(n)}$ comparisons.
- **Stability : stable**
  Insertion Sort is **stable**, because when two elements are equal, the new element (the key) does not move past another equal element inserted before it.
- **Additional Experiments** 4.3
  - **Standard vs Using binary search:** Compare the performance of a standard Insertion Sort versus an optimized version that employs *binary search* to find the correct insertion position of the key element.

2.5 Selection Sort

- **Algorithm Description**
  Selection Sort finds the minimum element in the unsorted part of the array and swaps it with the element at the beginning of that part. This process is repeated for each position, gradually expanding the sorted portion from left to right.
- **Time Complexity : $O(n^2)$**
  For every iteration i, the algorithm performs a linear scan to find the minimum in the subarray $A[i \dots n-1]$. Hence, the overall complexity is $O(n^2)$, regardless of the array distribution.
- **Stability : not stable**
  In its basic form, Selection Sort is **not stable**, because it may swap non-adjacent elements and thereby change the relative order of equal keys.

2.6 Quick Sort

- **Algorithm Description**
  We present two versions of Quick Sort:
  - **CLRS Approach:** The pivot is chosen as the *last* element of the subarray. Elements smaller or equal to the pivot are moved to one side, while larger elements remain on the other side. The pivot is then placed in its correct position.
  - **Data Structures (DS) Approach:** The pivot is chosen as the *first* element of the subarray. Two pointers (`i` from the left and `j` from the right) move inwards, i detecting those greater than pivot and j detecting those less than pivot and swaps them. One thing to know is that this algorithm also detects the same element with the key. After the pointers cross, the pivot is placed between them.
- **Time Complexity : $O(n \log n)$**
  In general, both versions of Quick Sort achieve an average time complexity of $O(n \log n)$. However, in the **worst case**, standard Quick Sort can degrade to $O(n^2)$ (e.g., if the pivot is repeatedly chosen as the smallest or largest element).

Algorithm 1: Library Sort

```
Input  : Array $A$, real parameter $\epsilon$ (e.g., 0.5)
Output : Sorted array in ascending order
$n \leftarrow \text{length}(A)$;
$S\_size \leftarrow \lfloor (2 + 2 \times \epsilon) \times n \rfloor$;
Initialize array $S$ of size $S\_size$ with None;
$S[0] \leftarrow A[0]$;
$positions \leftarrow [\,0\,]$; // tracks indices of filled cells in $S$
$m \leftarrow 1$;
while $m < n$ do
    $next\_m \leftarrow \min(2 \times m, n)$;
    for $i \leftarrow m$ to $(next\_m - 1)$ do
        attempt_insert($S$, $positions$, $A[i]$);
    $new\_active\_length \leftarrow \lfloor (2 + 2 \times \epsilon) \times next\_m \rfloor$;
    $(S, positions) \leftarrow$
      rebalance($S$, $positions$, $next\_m$, $new\_active\_length$);
    $m \leftarrow next\_m$;
Initialize result as an empty list;
for $i \leftarrow 0$ to $(length(S) - 1)$ do
    if $S[i] \neq$ None then
        append $S[i]$ to result
return result;
```

- **Differences in All-Equal Data Sets.**
  If the input array consists of all identical elements:
  - **CLRS version:** The partition procedure iterates through each element and swaps it into the same region, causing a partition index that barely moves, effectively leading to $O(n^2)$ time.
  - **DS version:** Thanks to how the two-pointer approach works, the pointers converge quickly when all elements are the same, maintaining $O(n \log n)$ time.
- **Stability : not stable**
  Generally, in-place Quick Sort is **not stable**. Standard partitioning schemes (including both CLRS and DS) can reorder elements with equal values relative to each other.
- **Additional Experiments** 4.4
  We plan three separate comparisons:
  - **DS vs. CLRS:** Observing how the two partition strategies behave in various cases.
  - **Ordinary Quick Sort vs. Randomized Quick Sort:** Evaluating whether randomizing the pivot selection improves average performance and avoids the worst-case scenario.
  - **Ordinary Quick Sort vs. Dual-Pivot Quick Sort:** Investigating whether using two pivots instead of one provides a noticeable performance gain.

3 ADVANCED SORTING ALGORITHMS

For more sophisticated use cases or optimizations, we explore **Library Sort, Tim Sort, Cocktail Shaker Sort, Comb Sort, Tournament Sort, and Introsort**. These algorithms often combine ideas from multiple sorting methods or introduce additional space or heuristics to improve performance in practice.

Algorithm 2: `attempt_insert` (Sub)

```
Input  : Array $S$, list $positions$, a key $k$ to insert
Output : No direct return; modifies $S$ and $positions$ in-place
$posIndex \leftarrow$ binary_search_positions($S$, $positions$, $k$);
$insertIdx \leftarrow$ find_insert_pos($positions$, $posIndex$);
if $S[insertIdx]$ = None then
    $S[insertIdx] \leftarrow k$;
else
    $pos \leftarrow insertIdx$;
    while $pos < length(S)$ and $S[pos] \neq$ None do
        $pos \leftarrow pos + 1$;
    for $i \leftarrow pos$ to $(insertIdx + 1)$ step $-1$ do
        $S[i] \leftarrow S[i-1]$;
    for $i \leftarrow 0$ to length(positions)-1 do
        if $positions[i] \geq insertIdx$ and $positions[i] \leq pos$ then
            $positions[i] \leftarrow positions[i] + 1$;
    $S[insertIdx] \leftarrow k$;
insert_in_list($positions, posIndex, insertIdx$);
```

Algorithm 3: `rebalance` (Sub)

```
Input  : Array $S$, list $positions$, integer $m$, integer $new\_active\_length$
Output : New array $S$ and updated list $positions$
Initialize $newS$ as an array of the same length as $S$ filled with None;
$newPositions \leftarrow [\ ]$;
$spacing \leftarrow \frac{new\_active\_length}{m}$;
for $i \leftarrow 0$ to $(m-1)$ do
    $oldIdx \leftarrow positions[i]$;
    $val \leftarrow S[\ oldIdx\ ]$;
    $placeIndex \leftarrow \lfloor(\ i + 0.5\ ) \times spacing\rfloor$;
    if $placeIndex \geq length(newS)$ then
        $placeIndex \leftarrow$ length($newS$) $- 1$;
    $newS[\ placeIndex\ ] \leftarrow val$;
    Append $placeIndex$ to $newPositions$;
return ( $newS$, $newPositions$ );
```

3.1 Library Sort

- **Algorithm Description**
  Library Sort was introduced as an improved version of *insertion-sort*, aiming to reduce the frequent shifting of elements that occurs in a naive Insertion Sort. It achieves this by placing gaps between inserted elements in a larger "sparse" array (S) and periodically *rebalancing* them so that future insertions can be done with fewer shifts on average.
  However, it is important to note that the implementation presented here relies on a `positions` array which must be updated after every insertion and during rebalancing. Because we use `positions` to locate the insertion index (via binary search) and adjust elements, the overhead of maintaining this structure can negate the potential performance benefit. As a result, this version strictly underperforms a straightforward *binary search-based insertion sort.* A more advanced approach would require using only the sparse array S to handle insertion points and rebalancing, eliminating the need for a **separate index structure**.
- **Time Complexity : $O(n \log n)$**
  The average-case time complexity of Library Sort is $\boldsymbol{O(n \log n)}$. If new elements are placed into a sufficiently larger array (with parameter ϵ) and rebalanced at appropriate intervals, the overall insertion process can be amortized to yield $O(\log n)$ steps per insertion. As a result, the total running time reaches $O(n \log n)$. More detailed proof can be found in the work of Bender *et al.* [1]. However, our particular implementation relies on a `positions` array that introduces extra overhead for tracking and shifting, so it may not consistently outperform Insertion Sort in practice.
- **Additional Memory Space**
  The size of the array is $(2 + 2\epsilon)n$ and the `positions` array is also $O(n)$. Also in rebalancing, it uses additional temporary array (`new_S, new_Positions`). So it uses $\boldsymbol{O(n)}$ additional space.
- **Stability : stable**
  It may differ from the way how it is implemented, our algorithm behaves as **stable** sorting algorithm. Our approach preserves the relative order of duplicates by:
  - Always inserting new elements *after* any existing elements of the same value.
  - *Rebalncing* elements according to their order in `positions`, ensuring that duplicates remain in the same sequence as when they were inserted.
- **Pros and Cons**
  - Insertion sort takes a long time since it has to shift lots of elements. Library sort can solve this problem.
  - Due to our poor implementation, there is too much overhead. It cannot be faster than binary-search insertion sort.
- **Additional Experiments** 4.3
  - **Naive Insertion Sort vs. Library Sort vs. Binary-search-based Insertion Sort:** Since library sort is introduced as an improved version of insertion sort, try comparing with insertion sort.

3.2 Tim sort

- **Algorithm Description**
  Timsort is a hybrid algorithm that combines ideas from *Merge Sort* and *Insertion Sort.* It first scans the array to identify or create small *runs* (subarrays that are either non-decreasing or non-increasing) using *insertion sort.* In the final stage, these runs are merged together (using *merge sort*) until the entire array is sorted.
  The motivation is to handle real-world data efficiently, leveraging naturally occurring runs (which might be already partially sorted) and a simpler insertion-based approach for small segments.
- **Time Complexity : $O(n \log n)$**
  In practical scenarios, Timsort operates in $O(n \log n)$ time. Already-sorted or partially sorted data can lead to linear or near-linear performance, since finding and merging sorted runs is cheaper. The `insertion_sort` step is applied only to short subarrays (below `minRun`), where $O(k^2)$ cost is negligible compared to the overall $O(n \log n)$ cost.
- **Additional Memory Space**
  Since we need to store *runs*, it needs $O(n)$ additional space.
- **Stability : not stable**
  Since Tim sort is a combination of two stable algorithms, it can be thought as stable. But, as we sometimes reverse the array (in

Algorithm 4: Timsort

```
   Input    : Array $A$ of length $n$
   Output   : The array $A$ is sorted in ascending order
 2 if $n \le 1$ then
 3     return
 5 $minRun \leftarrow$ calc_min_run($n$);
 7 Initialize list of runs, runs $\leftarrow$ [ ];
 9 $i \leftarrow 0$;
11 while $i < n$ do
12     $runStart \leftarrow i$;
13     $i \leftarrow i + 1$;
14     if $i < n$ then
15         if $A[i] < A[i-1]$ then
16             while $i < n$ and $A[i] \le A[i-1]$ do
17                 $i \leftarrow i + 1$;
18             Reverse the subarray $A[runStart \ldots (i-1)]$;
19         else
20             while $i < n$ and $A[i] \ge A[i-1]$ do
21                 $i \leftarrow i + 1$;
22     $runEnd \leftarrow i - 1$;
23     $runLen \leftarrow runEnd - runStart + 1$;
24     if $runLen < minRun$ then
25         $end \leftarrow \min(runStart + (minRun - 1), n - 1)$;
26         insertion_sort($A$, $runStart$, $end$);
27         $runEnd \leftarrow end$;
28         $i \leftarrow end + 1$;
29     Append ( $runStart$, $runEnd$) to runs;
31 $size \leftarrow$ length(runs);
33 while $size > 1$ do
34     Initialize merged_runs $\leftarrow$ [ ];
35     $j \leftarrow 0$;
36     while $j < size$ do
37         if $j = size - 1$ then
38             Append runs[$j$] to merged_runs;
39             $j \leftarrow j + 1$;
40         else
41             ( $leftStart$, $leftEnd$) $\leftarrow$ runs[$j$];
42             ( $rightStart$, $rightEnd$) $\leftarrow$ runs[$j+1$];
43             merge($A$, $leftStart$, $leftEnd$, $rightEnd$);
44             Append ( $leftStart$, $rightEnd$) to merged_runs;
45             $j \leftarrow j + 2$;
46     runs $\leftarrow$ merged_runs;
47     $size \leftarrow$ length(runs);
```

line 18), the order of identical elements can be changed. So our implementation is **not stable**.

- **Pros and Cons**
  - Using insertion sort in short subarrays, overall performance can be enhanced.
  - Since it detects the decreasing or non-decreasing part, Tim sort outperforms in datasets which have sorted part.
- **Additional Experiments** 4.5
  - **Merge Sort vs. Tim Sort:** Since tim sort is an improved version of merge sort, try to compare it with merge sort.

Algorithm 5: Cocktail Shaker Sort

```
Input    : An array $A$ of length $n$
Output   : The array $A$ sorted in ascending order
$start \leftarrow 0$,   $end \leftarrow n - 1$;
$swapped \leftarrow$ True;
while $swapped$ do
    $swapped \leftarrow$ False;
    for $i \leftarrow start$ to $(end - 1)$ do
        if $A[i] > A[i+1]$ then
            swap $A[i] \leftrightarrow A[i+1]$;
            $swapped \leftarrow$ True;
    if $\neg swapped$ then
        break ; // No swaps = fully sorted
    $end \leftarrow end - 1$;
    $swapped \leftarrow$ False;
    for $i \leftarrow end$ to $(start + 1)$ step $-1$ do
        if $A[i] < A[i-1]$ then
            swap $A[i] \leftrightarrow A[i-1]$;
            $swapped \leftarrow$ True;
    $start \leftarrow start + 1$;
```

3.3 Cocktail Shaker Sort

- **Algorithm Description**
  Cocktail Shaker Sort, also known as *Bidirectional Bubble Sort*, is a variation of Bubble Sort that traverses the list in both directions. In each forward pass, it "bubbles up" the largest element to the end, and in the subsequent backward pass, it "bubbles down" the smallest element toward the front. This bidirectional approach can reduce the number of passes needed compared to a standard one-direction Bubble Sort, especially if smaller elements are originally placed near the end.
- **Time Complexity : $O(n^2)$**
  Like Bubble Sort, the worst-case and average-case time complexity of Cocktail Shaker Sort is $O(n^2)$. Each iteration performs a forward pass and a backward pass, leading to a similar count of comparisons and swaps. However, in best-case scenarios (e.g., partially sorted data), the algorithm may stop early if no swaps occur during a pass, giving near $O(n)$ behavior.
- **Stability : stable**
  Cocktail Shaker Sort is **stable** under the same principle as Bubble Sort: it only swaps adjacent elements, so equal elements are never rearranged out of order.
- **Pros and Cons**
  - Since it keeps filling the both side of the array, it is slightly faster than bubble sort.
  - Since it moves big elements to the back and small elements to the front, it takes too long in reverse_sorted datasets.
- **Additional Experiments** 4.6
  - **Bubble Sort vs. Cocktail Shaker Sort:** Try comparing it with Bubble Sort, observing whether the bidirectional scans significantly reduce the total number of passes in various datasets.

3.4 Comb Sort

- **Algorithm Description**
  Comb Sort is often regarded as a variation of Bubble Sort that

Algorithm 6: Comb Sort

```
Input  : An array $A$ of length $n$
Output : The array $A$ sorted in ascending order
$gap \leftarrow n$;
shrink $\leftarrow$ 1.3;
sortedFlag $\leftarrow$ False;
while $\neg$sortedFlag do
    $gap \leftarrow \lfloor gap/\text{shrink} \rfloor$;
    if $gap \le 1$ then
        $gap \leftarrow 1$;
        sortedFlag $\leftarrow$ True;
    $i \leftarrow 0$;
    while $i + gap < n$ do
        if $A[i] > A[i + gap]$ then
            swap $A[i] \leftrightarrow A[i + gap]$;
            sortedFlag $\leftarrow$ False;
        $i \leftarrow i + 1$;
```

initially uses a larger *gap* to compare distant elements. The gap size is reduced in each iteration by a constant `shrink` factor (commonly 1.3). Once the gap reaches 1, the algorithm behaves similarly to Bubble Sort, potentially requiring a final pass to ensure everything is fully sorted.

- **Time Complexity : $O(n^2)$**
  On average, Comb Sort tends to perform better than a standard Bubble Sort, but in the worst case it remains $O(n^2)$.
- **Stability : not stable**
  Comb Sort is generally considered **not stable**, since it can swap elements that are far apart, potentially reordering duplicates.
- **Pros and Cons**
  - Even though the time complexity is $O(n)$, it shows performance closer to $O(n \log n)$.
  - There can be the worst case like an array with the elements alternating between small and large values.
- **Additional Experiments** 4.6
  - **Bubble Sort vs. Comb Sort:** Try comparing it with Bubble Sort, since comb sort is an improved version of bubble sort.

3.5 Tournament Sort

- **Algorithm Description**
  Tournament Sort constructs a tree structure (similar to a *segment tree*) where the leaves represent the array elements. Each internal node stores the "winner" (in this case, the smallest value) between its two children, effectively forming a tournament bracket. After building this tree, we repeatedly select the root (global minimum), append it to the output, and set that leaf to ∞, causing the tree to "referee" the next smallest element. This process continues until all elements have been extracted in ascending order.
- **Time Complexity : $O(n \log n)$**
  Building the initial tree from n elements requires $O(m)$ assignments where m is the smallest power of two $\ge n$. Each of the n extractions requires updating the path from a leaf to the root, which costs $O(\log m) \approx O(\log n)$. Hence, the overall complexity is $O(n \log n)$.

Algorithm 7: Tournament Sort

```
Input  : An array $A$ of length $n$
Output : A new array containing the elements of $A$ in ascending
         order
$m \leftarrow 1$;
while $m < n$ do
    $m \leftarrow m \times 2$;
Initialize array tree of size $2 \times m$ with None;
for $i \leftarrow m$ to $(m + n - 1)$ do
    tree[i] $\leftarrow (A[i - m],\ i - m)$;
for $i \leftarrow (m + n)$ to $(2 \times m - 1)$ do
    tree[i] $\leftarrow (\infty,\ -1)$;
for $i \leftarrow (m - 1)$ to 1 step $-1$ do
    $left \leftarrow$ tree$[2 \times i]$;
    $right \leftarrow$ tree$[2 \times i + 1]$;
    tree[i] $\leftarrow$ ( if $left[0] \le right[0]$ then $left$ else $right$);
Initialize result as an empty list;
for _ $\leftarrow$ 1 to $n$ do
    $winner \leftarrow$ tree$[1]$;
    Append $winner[0]$ to result;
    $pos \leftarrow (winner[1]) + m$;
    tree[pos] $\leftarrow (\infty,\ winner[1])$;
    while $pos \ge 1$ do
        $left \leftarrow$ tree$[2 \times pos]$;
        $right \leftarrow$ tree$[2 \times pos + 1]$;
        tree[pos]
          $\leftarrow$ ( if $left[0] \le right[0]$ then $left$ else $right$);
        $pos \leftarrow \lfloor pos/2 \rfloor$;
return result;
```

Algorithm 8: Intro Sort

```
Input  : Array $A$ of length $n$
Output : $A$ is sorted in ascending order
if $n \le 16$ then
    insertion_sort_range($A$, 0, $(n - 1)$);
    return $A$;
$depthLimit \leftarrow 2 \times \lceil \log2(n) \rceil$;
quick_sort_range($A$, 0, $(n - 1)$, $depthLimit$);
insertion_sort_range($A$, 0, $(n - 1)$);
return $A$;
```

- **Additional Memory Space**
  To make a tree structure, $O(n)$ space is additionally needed.
- **Stability : stable**
  In this particular implementation, if two values are equal, the tree logic selects the *left* one (via $\le$ comparison). So, duplicates outputs in the same order they appeared in original array, making the sort **stable** for equal keys.
- **Pros and Cons**
  - Regardless of the data variety, it always show $O(n \log n)$ performance.

3.6 Intro Sort

- **Algorithm Description**
  Intro sort is an improved version of Quick sort. It begins like a

```
Algorithm 9: quick_sort_range
Input  : Array $A$, indices $left$, $right$, integer $depth$
Output : Recursively sorts $A[left..right]$, switching to other
         methods if depth is exceeded
if $left < right$ then
    if $depth = 0$ then
        if $(right - left + 1) > 16$ then
            heap_sort_range($A$, $left$, $right$);
        else
            insertion_sort_range($A$, $left$, $right$);
        return;
    $q \leftarrow$ partition($A$, $left$, $right$);
    quick_sort_range($A$, $left$, $(q-1)$, $(depth-1)$);
    quick_sort_range($A$, $(q+1)$, $right$, $(depth-1)$);
```

typical Quick Sort but monitors the recursion depth. If the depth limit (proportional to $\log n$) is exceeded, it switches to a different method (Heap Sort or Insertion Sort) to avoid Quick Sort's worst-case $O(n^2)$. In practice, small subarrays (e.g., of size ≤ 16) are handled by Insertion Sort for efficiency.

- **Time Complexity : $O(n \log n)$**
  Initially, it uses Quick Sort which is typically $O(n \log n)$ on average. When it detects any risk of degenerating into $O(n^2)$, it switchs to heap sort (or insertion sort), having $O(n \log n)$ or better complexity in the relevant range. So, it is $O(n \log n)$ even in the worst case.
- **Stability : not stable**
  Since it consists of not stable algorithms, it is not stable.
- **Pros and Cons**
  - It shows high performance consistently.
  - It is in-place algorithm.
- **Additional Experiments** 4.7
  - **Quick Sort vs. Intro Sort:** Since it is an improved version of quick sort, try comparing it with quick sort.

4 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we present the experimental setup and the results of various sorting algorithm tests. All implementations are written in Python, and the full source code and all the data is available on GitHub:

https://github.com/Chan-py/Sorting-Algorithm-Analysis

4.1 Experimental Setup

We used a standard desktop environment with Python 3.9, running on a machine with 16 GB of RAM and an Intel Core i7 processor. Each experiment was run multiple times($10 \sim 100$), and average running times were recorded. We made testcases used it ourself. Size varies from 1K to 1M and various distributions are used (random, sorted, reverse_sorted, partially_sorted, same_element). The unit time of result is s(seconds).

4.2 Experiment 1: Variations of Merge Sort

Our first experiment focuses on **Merge Sort** and explores two main variations. To simplify the experimental setup, I fixed the dataset size to 100K in the first experiment and used random distributions in the second, as neither factor significantly affected the results.

- **Sentinel vs. No Sentinel:**

	With Sentinel	Without Sentinel
random	0.422	0.448
sorted	0.259	0.258
reverse_sorted	0.213	0.211
partially_sorted	0.223	0.225

Table 1: Sentinel vs. No Sentinel

  Using sentinels showed a slight advantage in terms of time efficiency in the common case. The reason is presumed to be that using sentinels **eliminates the need to check array bounds** in every iteration.
  However, in cases like sorted or reversely sorted, where one subset contains elements that are all smaller than or equal to those in the other subset, the number of comparisons is reduced, and the version without sentinels sometimes outperformed.
- **2-Way vs. 3-Way Splitting:**

	2-Way	3-Way
1K	0.0015	0.0013
10K	0.018	0.016
100K	0.388	0.322
1M	5.762	4.971

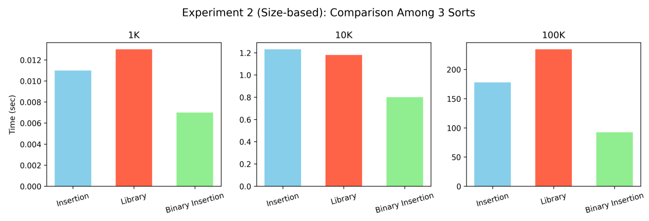
Table 2: 2-Way vs. 3-Way

  In our experiment, the **3-way** merge showed better efficiency compared to the standard 2-way merge. However, this does not necessarily imply that splitting into more parts always leads to better performance. In a k-way split, each iteration requires $\binom{k}{2}$ comparisons to merge the k subarrays. On the other hand, the overall recursion depth becomes $\log_k n$. Therefore, as k increases, the number of steps per iteration grows, while the total number of recursive steps decreases—resulting in a **trade-off** structure.

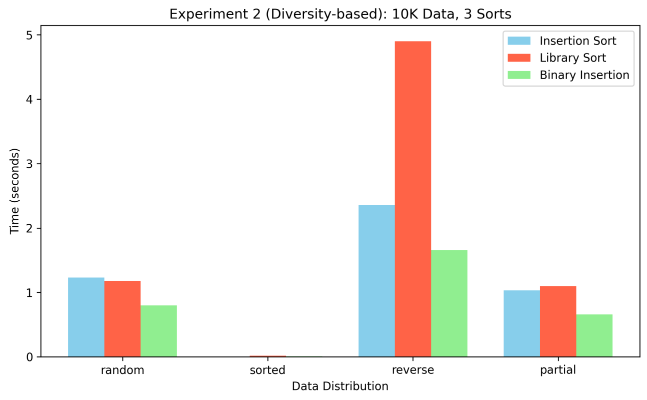
4.3 Experiment 2: Insertion Sort vs. Library Sort

In this experiment, we compare three insertion-based sorting algorithms (naive insertion sort, library sort, binary-search-based insertion sort).

- **Size-based Comparison (Random Data)** - Figure 1a
  Binary-search based insertion sort always outperforms the other two since it takes only $O(\log n)$ in searching. Also, except for 10K dataset, library sort was even slower than insertion sort.
- **Distribution-based Comparison (Fixed 10K)** - Figure 1b
  Only in the *sorted* case, the naive insertion sort exceeded the binary search-based insertion sort. In the *reverse* distribution, every insertion sort-based algorithms were slow since it is the worst case. As mentioned above, library sort was always slower than binary search-based insertion sort. In some cases, library sort was even slower than insertion sort. The reason is estimated to be the extra overhead in code implementation.



(a) Size-based comparison



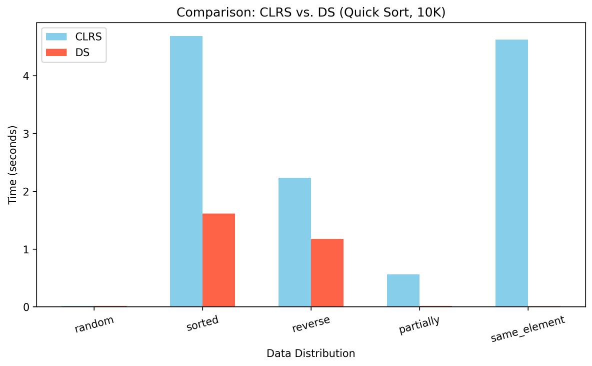
(b) Distribution-based comparison

Figure 1: Insertion sort vs. Library sort

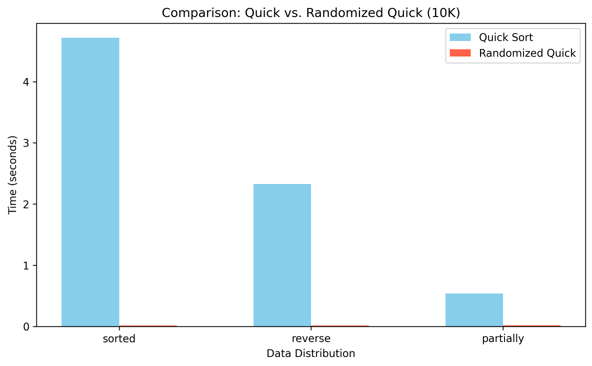
4.4 Experiment 3: Variations of Quick Sort

In this experiment, we explore different implementations or variants of Quick Sort.

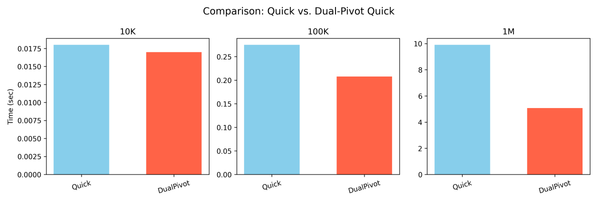
- **Implementation method (CLRS vs. DS)** - Figure 2a
  - DS is generally faster than CLRS. The DS approach tends to perform fewer swaps and handle duplicates more efficiently than the CLRS approach.
  - In extreme cases like `sorted` or `reverse_sorted`, quick sort can degrade toward $O(n^2)$ performance. (worst case pivot selection)
  - In the `same_element` dataset, DS maintained an $O(n \log n)$-like behavior, while CLRS degrade toward $O(n^2)$ performance.
- **Randomized Quick Sort** - Figure 2b
  - On the data like `sorted` or `reverse_sorted`, the randomized partition significantly reduced the chance of a bad pivot, yielding behavior near $O(n \log n)$ instead of $O(n^2)$.
  - We observed that randomized Quick Sort kept running time low and stable across multiple input distributions.
- **Dual-Pivot Quick Sort** - Figure 2c
  - As the dataset grew (e.g., up to 1M elements), dual-pivot sorting outperformed the single-pivot version. The height of recursion differ larger in a big dataset.
  - While adding a second pivot helps improving performance, increasing the pivot count does not guarantee improving performance. More pivots can mean more complex partitioning logic and overhead, so there is a balance (trade-off) to be made.



(a) CLRS vs. DS



(b) Quick vs. Randomized



(c) Dual-Pivot Quick

Figure 2: Different Quick Sort Variants.

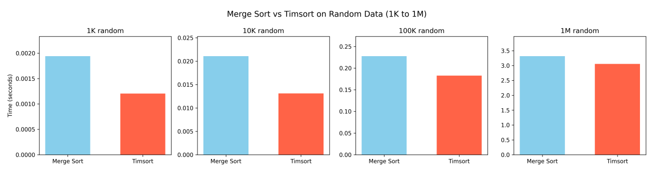
4.5 Experiment 4: Merge Sort vs. Timsort

We compared Merge Sort and Timsort on both varying sizes of random data and different distributions at fixed size (100K).

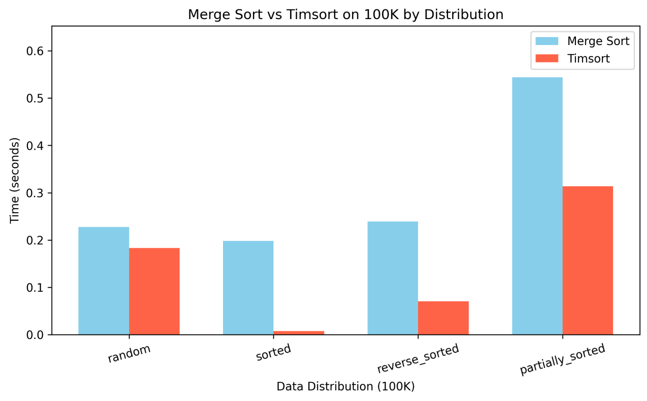
- **Size-based (Random data)** - Figure 3a
  Timsort consistently outperforms Merge Sort at every scale since it optimizes sorting short subarrays.
- **Distribution-based (100K)** - Figure 3b
  Timsort gains a particularly large advantage on partially sorted inputs, thanks to its run detection and insertion-sort on small runs.

4.6 Experiment 5: Bubble vs. Cocktail Shaker vs. Comb Sort

We evaluated three quadratic-time algorithms on both random data of increasing size and on 100K inputs of various distributions.

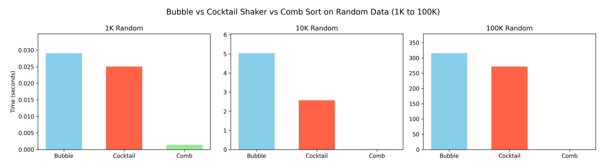


(a) Random data (1K, 10K, 100K, 1M)

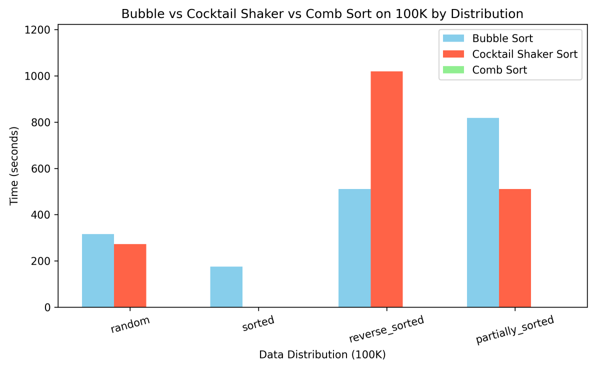


(b) 100K data by distribution

Figure 3: Merge Sort vs. Timsort



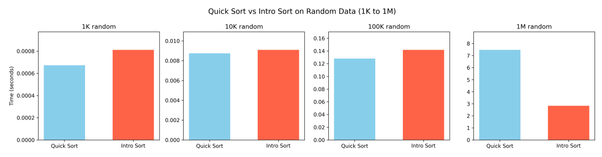
(a) Random data (1K, 10K, 100K)



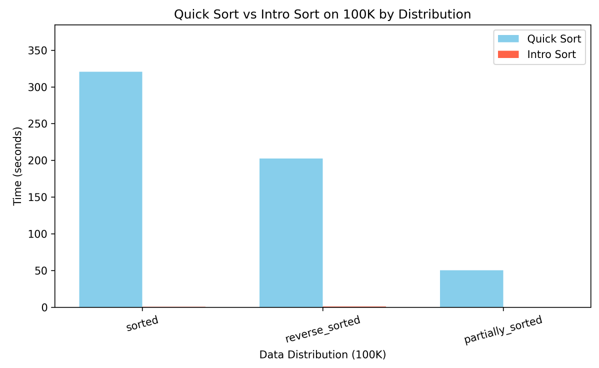
(b) 100K data by distribution

Figure 4: Bubble vs. Cocktail Shaker vs. Comb Sort

- **Size-based (Random data)** - Figure 4a
  Comb Sort is the fastest in every case follows by Cocktail shaker sort.
- **Distribution-based (100K)** - Figure 4b
  Comb Sort again leads, while Cocktail Shaker Sort underperforms especially on reverse-sorted data, even slower than bubble sort (the worst case of cocktail shaker sort). Maybe this is because of the locality of reference (cache).



(a) Random data (1K, 10K, 100K, 1M)



(b) 100K data by distribution

Figure 5: Quick Sort vs. Intro Sort

4.7 Experiment 6: Quick Sort vs. Intro Sort

Finally, we compared standard Quick Sort and Intro Sort on random data of varying size and on 100K inputs with different distributions.

- **Size-based (Random data)** - Figure 5a
  Quick sort is faster on small to medium arrays, but on 1M elements intro sort is faster than quick sort.
- **Distribution-based (100K)** - Figure 5b
  Intro Sort avoids Quick Sort's worst-cases on sorted or reverse-sorted inputs thanks to the depth limit logic.

4.8 Overall Key Points

- Tim sort and Intro sort was the fastest in the random datasets.
- Tim sort, Cocktail shaker sort, and Insertion sort was the fastest in the sorted datasets.
- Tim sort was the fastest in the reverse_sorted datasets.
- Tim sort and Intro sort was the fastest in the partially_sorted datasets.
- **Hybrid and adaptive strategies showed good performance.** Algorithms that combine simple sorts on small runs (Timsort, Intro Sort) or detect existing order (Timsort's run detection) consistently showed good performance overall.
- **Quadratic sorts diverge quickly.** Bubble, Cocktail Shaker, and Comb Sort all exhibit $O(n^2)$ scaling, but small engineering tweaks (gap sequences in Comb Sort) can yield order-of-magnitude speedups even within the same complexity class.

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.