CSE331 - Assignment #1

Deokhyeon Kim (20201032)
UNIST
South Korea
ejrgus1404@unist.ac.kr

1 PROBLEM STATEMENT

Sorting is one of the most fundamental and widely used algorithms due to its numerous practical applications. The ability to organize data in a specific order is a powerful tool—for instance, searching for a specific element in a sorted array can be significantly more efficient.

In many cases, sorting algorithms serve as subroutines for more complex algorithms, such as *Kruskal's Algorithm*. Interestingly, there is no universally optimal sorting algorithm. This means that an algorithm that performs well in one situation might not be the best choice in another. For example, while quick sort generally outperforms insertion sort in terms of average-case time complexity, insertion sort can surpass quick sort when the input data is nearly or fully sorted.

In this paper, we aim to analyze *six basic sorting algorithms* and *six advanced sorting algorithms* from both theoretical and experimental perspectives. The goal is to provide useful insights and guidelines for choosing the most appropriate sorting algorithm based on the characteristics of the input data.

2 BASIC SORTING ALGORITHMS

In this section, we introduce six basic sorting algorithms: Merge Sort, Heap Sort, Bubble Sort, Insertion Sort, Selection Sort, and Quick Sort. We then analyze the features and time complexity of each algorithm.

2.1 Merge Sort

Merge Sort is a $\Theta(n \log n)$ sorting algorithm based on the divide-and-conquer paradigm. Its operation can be summarized as follows: the input array is recursively divided into halves until each subarray contains only one element (a base case, as a single-element array is trivially sorted). Then, two consecutive subarrays are merged recursively by selecting the smaller element from the two.

Since the merge process requires additional memory to combine subarrays, Merge Sort is not an in-place algorithm. However, it is a stable sorting algorithm, as elements with equal values can consistently be chosen from the left subarray during merging.

The recurrence relation for Merge Sort is given by:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Applying the *Master Theorem* to this recurrence yields the time complexity $\Theta(n \log n)$.

Due to its design, Merge Sort performs well when one of the subarrays is exhausted early during merging. Thus, already sorted or reverse-sorted inputs often yield the best performance, whereas randomized input tends to represent the worst case.

I also tested a variation known as k-way Merge Sort, which divides the array into k subarrays in each recursive step instead of just two. The time complexity of k-way Merge Sort is $\Theta(n \log n)$ which can be derived in a similar manner to that of the basic Merge Sort. Interestingly, 3-way Merge Sort demonstrated better performance than the standard 2-way version. The key difference lies in the total number of recursive calls. Since k-way Merge Sort divides the array into k parts, the recursion depth becomes $\log_k n$, and the total number of recursive calls can be expressed as:

$$\sum_{i=0}^{\log_k n} k^i = \frac{kn-1}{k-1}$$

Here, the coefficient of n is $\frac{k}{k-1}$, which decreases as k increases, suggesting fewer recursive calls with larger k.

Now, consider the extreme case where $k = n$. This scenario effectively becomes Selection Sort, which has a time complexity of $O(n^2)$. Therefore, the function

$$f(k) = \text{execution time of } k\text{-way Merge Sort}$$

forms a concave uni-modal function. Experimental results show that 5-way Merge Sort yields the best performance among the variations tested. For instance, on a partially sorted input of size 1 million, the standard Merge Sort took approximately 650 milliseconds, whereas the 5-way Merge Sort completed in just 456 milliseconds.

One might ask: "What if we reduce the overhead of recursion by using a stack-based implementation?" Another experiment reveals that, for both basic and 3-way Merge Sort, the recursive implementation outperforms the stack-based version in terms of execution time.

2.2 Heap Sort

The main idea of Heap Sort is to first construct a heap from the input array and then repeatedly extract the maximum element to produce a sorted sequence. For ascending order sorting, a max-heap is used. The bottom-up heap construction takes $\Theta(n)$ time, and restoring the heap property after each extraction (pop operation) takes $O(\log n)$ time. Therefore, the overall time complexity of Heap Sort is $O(n \log n)$.

Heap Sort is an in-place algorithm, as both heap construction and pop operation can be performed without using additional memory. However, it is not a stable sorting algorithm because the heap structure does not preserve the relative order of elements with equal keys.

The operation to restore the heap property—commonly known as *heapify*—is typically implemented using recursion. However, an iterative version using a single loop significantly reduces the overhead of recursive calls. Experimental results demonstrate that, for an input of size 1 million with randomized values, the recursive version took approximately 809 milliseconds, while the non-recursive version completed in 733 milliseconds. This performance gap tends to grow as the input size increases.

Another interesting aspect of Heap Sort is the indexing scheme. Most implementations use a 0-based heap, where the left and right child indices of a node at index i are $2i + 1$ and $2i + 2$, respectively. In contrast, a 1-based heap allows for more efficient bitwise calculations: the left and right child indices become $2i$ and $2i + 1$, which can be computed using $(i \ll 1)$ and $(i \ll 1 \mid 1)$, while the parent index is $(i \gg 1)$.

Although this seems to offer an advantage, experimental results show otherwise. For the same 1 million-element randomized dataset, the 1-based heap sort took 1208 milliseconds, compared to just 809 milliseconds for the 0-based implementation. The performance degradation is likely due to the additional overhead involved in adjusting indices to accommodate the 1-based structure.

Therefore, in practice, the 0-based, non-recursive version of Heap Sort proves to be the most efficient among the variations tested.

Time Complexity Analysis. Heap Sort consists of two main phases: building a max-heap from the input array and repeatedly extracting the maximum element to sort the array.

1. Building the Max-Heap. The `BUILD-MAX-HEAP` procedure constructs a max-heap from an unsorted array using a bottom-up approach. It calls `MAX-HEAPIFY` for all non-leaf nodes in reverse level order. Although a single `MAX-HEAPIFY` operation can take up to $O(\log n)$ time, most nodes are located near the bottom of the tree and require much less work.

Let us analyze the tighter upper bound on the cost of building the heap. For a node at height h, the cost of heapifying is $O(h)$. Since the number of nodes at height h is at most $\frac{n}{2^{h+1}}$, the total cost is:

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} \cdot O(h) = O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h}\right)$$

The summation $\sum_{h=0}^{\infty} \frac{h}{2^h}$ converges to a constant, so the overall cost is $O(n)$. Therefore, `BUILD-MAX-HEAP` runs in $\Theta(n)$ time.

2. Heap Sort Phase. After building the heap, Heap Sort performs $n - 1$ calls to `MAX-HEAPIFY` as it repeatedly extracts the maximum element and re-heapifies the remaining elements. Each call to `MAX-HEAPIFY` takes $O(\log n)$ time in the worst case, leading to a total cost of:

$$(n - 1) \cdot O(\log n) = O(n \log n)$$

Total Time Complexity. Combining both phases:

$$T(n) = \Theta(n) + O(n \log n) = O(n \log n)$$

Thus, the overall time complexity of Heap Sort is $O(n \log n)$ in the worst case.

2.3 Bubble Sort

Bubble Sort is the simplest, yet the most inefficient sorting algorithm discussed in this paper. It has a time complexity of $\Theta(n^2)$ and is both an in-place and stable sorting algorithm. Bubble Sort operates by repeatedly swapping adjacent elements if they are out of order.

More precisely, in the i-th iteration ($1 \le i \le n-1$), the algorithm compares each pair $(\texttt{arr}[j], \texttt{arr}[j+1])$ for $1 \le j \le n - i$. If $\texttt{arr}[j] > \texttt{arr}[j+1]$, the two elements are swapped. After each complete iteration over the array, the largest unsorted element is moved to its correct position.

The total number of comparisons and potential swaps is proportional to:

$$1 + 2 + 3 + \cdots + (n-2) + (n-1) = \frac{n(n-1)}{2}$$

Hence, the time complexity of Bubble Sort is $\Theta(n^2)$.

2.4 Insertion Sort

Insertion Sort divides the input array into two parts: a sorted subarray and an unsorted subarray. At each iteration, the first element of the unsorted portion is moved to its correct position within the sorted portion. Typically, the correct position is found through a linear search, which begins from the end of the sorted subarray. If the current position is not suitable for insertion, the element at that position is shifted one step to the right to make room. Once the correct position is found, the insertion is performed, and the iteration ends.

Because each iteration may terminate early when the correct position is found quickly, the best-case time complexity for already sorted input is $O(n)$. In contrast, the worst-case time complexity, which occurs when the input is in reverse order, is $O(n^2)$. Insertion Sort is both an in-place and stable sorting algorithm. It only uses swapping (or shifting) within the input array, and the relative order of equal elements is preserved.

A notable variation of this algorithm is called *Binary Insertion Sort*, which replaces the linear search with binary search to find the insertion position. Although binary search reduces the number of comparison operations to $O(\log n)$ per iteration, the algorithm still requires up to $O(n)$ shift operations in the worst case to make space for insertion. As a result, the overall worst-case time complexity remains $O(n^2)$.

Nevertheless, Binary Insertion Sort is more efficient when comparisons are expensive, as it reduces the number of comparisons significantly. To ensure that the best-case performance remains $O(n)$, the algorithm can be enhanced to detect whether the input is already sorted in ascending or descending order, allowing early termination of unnecessary iterations.

2.5 Selection Sort

The main idea of Selection Sort is to repeatedly select the smallest element from the input array and place it in its correct position. It has a time complexity of $\Theta(n^2)$ and is an in-place sorting algorithm. However, it is not stable. This is because, once the smallest element is found, it is swapped with the leftmost element in the unsorted portion of the array. This swapping operation can reverse the original order of equal elements, thereby breaking stability.

Since each iteration scans the remaining unsorted portion of the array, whose size decreases by one each time, the total number of comparisons is quadratic. Therefore, the time complexity of Selection Sort is $\Theta(n^2)$.

2.6 Quick Sort

As its name implies, Quick Sort is the fastest among the *six basic sorting algorithms* discussed in this paper. Although its worst-case

time complexity is $O(n^2)$, it achieves an average-case performance of $O(n \log n)$, and typically has the smallest constant factor among sorting algorithms in this complexity class.

Quick Sort operates as follows: first, a *pivot* element is selected from the input array. There are various pivot selection strategies, and in this implementation, we choose a pivot randomly, which guarantees an average-case time complexity of $O(n \log n)$ with high probability.

After selecting the pivot, the array is partitioned into two subarrays—one containing elements less than or equal to the pivot, and the other containing elements greater than the pivot. The same process is recursively applied to each subarray until the base case of a single-element array is reached, similar to Merge Sort. However, unlike Merge Sort, Quick Sort does not require an explicit merging step, since the partitioning process inherently ensures that each subarray becomes sorted.

As stated earlier, Quick Sort demonstrated the best performance among the basic sorting algorithms. For a randomized input of size one million, it completed in 495 milliseconds, whereas the second fastest algorithm, 5-way Merge Sort, required 572 milliseconds.

Average-case Time Complexity Analysis. To analyze the average-case running time of Quick Sort, we focus on the number of comparisons performed during its execution. Let X be the total number of comparisons made by partitioning procedures. Then, the running time of Quick Sort is $O(n + X)$, where n is the size of the input array.

Let the input array be $A = \{z_1, z_2, \ldots, z_n\}$, where z_i denotes the i-th smallest element. Define the indicator random variable $X_{i,j}$ as follows:

$$X_{i,j} = \begin{cases} 1 & \text{if } z_i \text{ is compared to } z_j \text{ during execution} \\ 0 & \text{otherwise} \end{cases}$$

The total number of comparisons can then be expressed as:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$$

Taking expectation on both sides:

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{i,j}]$$

It can be shown that:

$$\mathbb{E}[X_{i,j}] = \Pr\{z_i \text{ is compared to } z_j\} = \frac{2}{j - i + 1}$$

Thus:

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

Changing the index by setting $k = j - i$ gives:

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1}$$

This double sum is bounded by:

$$\mathbb{E}[X] = O(n \log n)$$

Therefore, the expected number of comparisons, and hence the average-case time complexity of Quick Sort, is $O(n \log n)$.

3 ADVANCED SORTING ALGORITHMS

In this section, we introduce six advanced sorting algorithms: Library Sort, Tim Sort, Cocktail Shaker Sort, Comb Sort, Tournament Sort, and Intro Sort. As in the previous section, we analyze the core characteristics and time complexity of each algorithm, considering both theoretical and experimental aspects.

3.1 Library Sort

Library sort, also known as gapped insertion sort, is a sorting algorithm that improves upon basic insertion sort by leaving gaps between sorted elements. This gap-based structure helps reduce the cost of shifting elements during insertion. The algorithm begins by randomly shuffling the input array to avoid worst-case scenarios, and processes at most $\log n$ rounds. In each round, it distributes k previously sorted elements into a $2k$-sized array, placing one gap to the right for each element. Then, it inserts next k unsorted elements one by one, using binary search to locate the appropriate position among the sorted elements and scanning for the nearest gap to insert the current element. If a nearby gap is found, it shifts elements to make space and inserts the element in its correct position while maintaining sorted order.

The key idea is to use binary search for efficient position lookup and sparse placement to minimize the number of shifts. This makes the algorithm significantly faster than basic insertion sort on average. The expected time complexity of Library Sort is $O(n \log n)$, and although the worst case is $O(n^2)$, such cases rarely occur due to randomization and the presence of gaps. Since it requires additional space for the gaps, the algorithm is not in-place. It is also not stable, as the relative order of equal elements may change during random shuffling.

In experiments, Library Sort demonstrated significantly better performance on randomized data compared to basic insertion sort. For an input of size one million, it completed in 1,556 seconds, while basic insertion sort required 5,943 seconds.

3.2 Tim Sort

The motivation behind Tim sort is to efficiently sort real-world data, which is often partially sorted and involves expensive comparisons. Tim sort is a hybrid sorting algorithm that combines the ideas of *binary insertion sort* and *merge sort*. Since it is based on merge sort, its worst-case time complexity is $O(n \log n)$, while the best-case time complexity is $O(n)$, as it terminates early when the input array is already sorted or reverse sorted. Tim sort is widely used in practical applications, including the built-in sort functions in Python and Java. The key idea is to divide the array into small runs, sort each run using binary insertion sort, and then merge them using several optimization techniques.

Tim sort is not an in-place algorithm, as the merge operations require additional memory. However, it is a stable sorting algorithm. This makes Tim sort suitable for real-world applications where stability is important.

In our implementation, a *run* is defined as a subsequence that is either increasing or decreasing. If the run is decreasing, it is reversed before sorting. The minimum size of a run is evaluated from the input size to ensure that all runs have a minimum length, and helps balance the merge process. Each run is sorted using binary insertion

Algorithm 1 Library Sort

Input: Iterators `begin`, `end` to an integer vector

$n \leftarrow$ `end` − `begin`
if $n \leq 1$ **then**
  return
end if
Randomly shuffle the range `[begin, end)`
$range \leftarrow 1$
Declare array $temp[2n]$
Declare vector `available` to store (`value`, `index`) pairs
while $range < n$ **do**
  Clear `available`
  $range \leftarrow 2 \cdot range$
  for $i \leftarrow 0$ to $range - 1$ **do**
    if i is odd or $i/2 \geq n$ **then**
      $temp[i] \leftarrow$ `EMPTY`
    else
      $temp[i] \leftarrow$ `begin`$[i/2]$
      Add (`temp`$[i], i$) to `available`
    end if
  end for
  for $i \leftarrow range/2$ to $\min(range, n) - 1$ **do**
    $val \leftarrow$ `begin`$[i]$
    Use binary search on `available` to find insertion index *ins*
    Probe left/right from *ins* to find nearest slot in *temp* where $temp[pos]$ = `EMPTY`
    Shift elements if necessary to make space
    Insert *val* into *temp* and update `available`
  end for
  Write back all non-`EMPTY` values from *temp* to `begin`
end while

sort and pushed onto a stack. The stack is managed to maintain certain invariants that determine how and when to merge runs, ensuring that the overall merge process remains efficient.

One of the key optimization in Tim sort is *galloping mode*, which accelerates merging when one of the two runs has a long sequence of smaller elements. If the same run wins three comparisons in a row, the algorithm switches to exponential search followed by binary search to quickly skip over multiple elements. This significantly reduces the number of comparisons when the input data is partially sorted.

Experimental results confirm the efficiency of Tim sort, especially on partially sorted data. On a data of size one million that is partially sorted, Tim sort completed in 192ms, significantly outperforming both quick sort (431ms) and 5-way merge sort (456ms). This result demonstrates that Tim sort's *galloping mode* and *run detection* is highly suitable for real-world data.

On randomized data of the same size, Tim sort completed in 601ms. While this is slightly slower than quick sort (495ms) and 5-way merge sort (572ms). This shows that although Tim sort may not outperform specialized algorithms like quick sort in purely random cases, it remains competitive. Moreover, its adaptability to ordered inputs gives it a distinct advantage in practice.

Procedure: get_minrun

Input: Integer n
Output: Integer *minrun*

$r \leftarrow 0$
while $n \geq 32$ **do**
  $r \leftarrow r \mid (n \bmod 2)$
  $n \leftarrow n \div 2$
end while
return $n + r$

Procedure: tim_sort_merge

Input: Two runs `[b_1, e_1)`, `[b_2, e_2)` as iterators

Use `upper_bound` on $[b_1, e_1)$ and `lower_bound` on $[b_2, e_2)$ to trim merge range
Let the smaller of the two trimmed ranges be copied to temporary buffer
Initialize $cnt_1 \leftarrow 0$, $cnt_2 \leftarrow 0$
while both ranges are non-empty **do**
  if element at current index in range 1 ≤ element at current index in range 2 **then**
    $cnt_1 \leftarrow cnt_1 + 1$, $cnt_2 \leftarrow 0$
    if $cnt_1 = 3$ **then**
      Perform galloping search on range 1 (exponential + binary search)
      Copy galloped block to output
      $cnt_1 \leftarrow 0$
    else
      Copy one element from range 1
    end if
  else
    $cnt_2 \leftarrow cnt_2 + 1$, $cnt_1 \leftarrow 0$
    if $cnt_2 = 3$ **then**
      Perform galloping search on range 2 (exponential + binary search)
      Copy galloped block to output
      $cnt_2 \leftarrow 0$
    else
      Copy one element from range 2
    end if
  end if
end while
Copy remaining elements from both ranges to output

3.3 Cocktail Shaker Sort

Cocktail Shaker Sort, also known as bidirectional Bubble Sort, improves upon Bubble Sort by scanning the array in both directions. It alternates between forward and backward passes—pushing the largest unsorted element to the end and the smallest to the front—thereby narrowing the unsorted range from both sides.

This bidirectional approach offers slight performance gains over Bubble Sort, but its overall time complexity remains $\Theta(n^2)$. It is both in-place and stable, as it is derived from bubble sort.

In the experiment, Cocktail Shaker Sort took 9259 seconds on a 1M-element randomized dataset, while Bubble Sort took 9893 seconds. A similar performance gap was observed across other types

Algorithm 2 Tim Sort

Input: Iterators `begin`, `end` to an integer vector

$n \leftarrow$ `end` $-$ `begin`
if $n \leq 1$ **then**
  return
end if
$minrun \leftarrow$ `get_minrun`(n)
Initialize empty stack `st`
$b \leftarrow$ `begin`
while $b \neq$ `end` **do**
  Identify run from b to e (length at least $minrun$)
  Sort run $[b, e)$ using binary insertion sort (with reversal if necessary)
  Push $[b, e)$ onto stack
  while stack size constraints are violated **do**
    Pop top runs and merge them using `tim_sort_merge`
    Push merged run back onto stack
  end while
  $b \leftarrow e$
end while
while stack size > 1 **do**
  Pop top two runs and merge
  Push merged run back onto stack
end while

of input data. Although Cocktail Shaker Sort is slightly faster than Bubble Sort, it does not demonstrate any significant improvement in practice.

Algorithm 3 Cocktail Shaker Sort

Input: Iterators `begin`, `end` to an integer vector

$flag \leftarrow$ `true`
while `begin` $+ 1 \neq$ `end` **do**
  if $flag$ **then**
    `end` $\leftarrow$ `end` $- 1$
    for $it \leftarrow$ `begin` to `end` $- 1$ **do**
      if $*it > *(it + 1)$ **then**
        Swap $*it$ and $*(it + 1)$
      end if
    end for
  else
    for $it \leftarrow$ `end` $- 1$ down to `begin` $+ 1$ **do**
      if $*(it - 1) > *it$ **then**
        Swap $*(it - 1)$ and $*it$
      end if
    end for
    `begin` $\leftarrow$ `begin` $+ 1$
  end if
  $flag \leftarrow \neg flag$
end while

3.4 Comb Sort

Comb Sort is an enhanced version of bubble sort that addresses one of its main weaknesses: when small values are located near the end of the array, they take many steps to reach their correct position near the beginning. This significantly slows down the sorting process. Comb Sort improves this by comparing and swapping elements that are far apart, rather than just adjacent ones. In each iteration, the algorithm compares elements that are a fixed gap apart. This gap gradually decreases, allowing misplaced values to reach their correct positions more quickly.

Initially, the gap is set to the length of the input array and is repeatedly divided by a shrink factor (1.3 in our implementation) after each iteration. When the gap becomes 1, the algorithm performs a final iteration equivalent to bubble sort. If this iteration completes without any swaps, the algorithm terminates, indicating that the array is sorted.

Comb Sort is an in-place algorithm, as it requires no extra memory aside from a few variables. However, it is not stable, since equal elements may change their relative order during swaps.

In the experiment, Comb Sort completed in **992ms** on 1M-element randomized data and **892ms** on partially sorted data—comparable to $O(n \log n)$ algorithms like Merge Sort or Heap Sort. This unexpectedly strong performance can be theoretically explained by its *gap-based comparison strategy*, which reduces the number of inversions more effectively than Bubble Sort.

In Bubble Sort, elements are only compared with adjacent neighbors, so a small value near the end (often called a "turtle") must pass through $\Theta(n)$ swaps to reach the front. This leads to a worst-case time complexity of $\Theta(n^2)$.

In contrast, Comb Sort compares elements that are *gap* positions apart, where the initial gap is n and shrinks each round by a factor of 1.3. The total number of passes is approximately:

$$\log_{1.3} n = O(\log n) \tag{1}$$

During each pass, the number of comparisons is $O(n)$, so the expected time complexity becomes:

$$O(n \log n) \tag{2}$$

This analysis reflects the average-case behavior, especially for inputs with many evenly spread inversions. Since most "turtles" are resolved early during large-gap passes, Comb Sort avoids the repetitive adjacent swaps that dominate Bubble Sort's cost.

Thus, although Comb Sort has a worst-case complexity of $O(n^2)$, its effective gap-based operation allows it to behave similarly to $O(n \log n)$ algorithms in practical scenarios.

3.5 Tournament Sort

Tournament Sort is a selection-based sorting algorithm that repeatedly finds the minimum element and removes it from the data structure. It is typically implemented using a complete binary tree, where each leaf represents an element of the array and each internal node stores the minimum of its two children. This structure resembles a single-elimination tournament, where the winner (the minimum value) propagates to the top.

We implemented Tournament Sort using a segment tree. The input array is inserted into the leaves of the tree from left to right, and the remaining leaves are initialized to `INF`. Each internal node stores the minimum of its two children, so the root of the tree always holds the minimum value in the array. After extracting the

Algorithm 4 Comb Sort

Input: Iterators `begin`, `end` to an integer vector

$n \leftarrow$ `end` $-$ `begin`
if $n \leq 1$ **then**
  return
end if
$gap \leftarrow n$
$shrink \leftarrow 1.3$
$sorted \leftarrow$ `false`
while not $sorted$ **do**
  $gap \leftarrow \lfloor gap/shrink \rfloor$
  if $gap \leq 1$ **then**
    $gap \leftarrow 1$
    $sorted \leftarrow$ `true`
  end if
  for $i \leftarrow 0$ to $n - gap - 1$ **do**
    if `begin`$[i]$ > `begin`$[i + gap]$ **then**
      Swap `begin`$[i]$ and `begin`$[i + gap]$
      $sorted \leftarrow$ `false`
    end if
  end for
end while

minimum, the corresponding leaf is marked as `INF`, and the tree is updated upward to maintain correctness.

Tournament Sort is not an in-place algorithm, as it requires $O(n)$ additional memory to maintain the segment tree. However, it is a stable sorting algorithm, since we can always choose the leftmost element among those with the same minimum value. The time complexity is $\theta(n \log n)$, which can be derived through an analysis similar to heap sort.

Surprisingly, Tournament Sort showed the best performance among all sorting algorithms tested in this paper on both randomized and partially sorted inputs. Although heap sort is theoretically faster due to its in-place nature and similar tree structure, Tournament Sort outperformed it in practice. On one million-element randomized and partially sorted data, Tournament Sort took only 263ms and 126ms respectively, while 0-based non-recursive heap sort took 733ms and 691ms. Additionally, the second fastest algorithms were Quick Sort (495ms) for randomized data and Tim Sort (192ms) for partially sorted data. The performance gap between Tournament Sort and the second-best algorithms remains significant in both cases.

This performance advantage originates from our segment tree implementation, which uses a single-loop update procedure and is heavily optimized with bitwise operations. In contrast, heap sort's heapify operation is more complex and introduces greater overhead in practice.

3.6 Intro Sort

Intro Sort is a hybrid sorting algorithm that begins with quick sort and switches to heap sort when the recursion depth exceeds a certain threshold. The algorithm is designed to provide the fast average-case performance of quick sort while ensuring worst-case $O(n \log n)$ time complexity like heap sort.

Procedure: init

Input: Iterators `begin`, `end` to an integer vector

Compute $size \leftarrow$ `end` $-$ `begin`
Choose smallest power-of-two size $sz \geq size$ based on $size$ (up to 2^{20})
Fill $tree[sz \ldots sz + size - 1]$ with input values
Fill remaining leaves with `INF`
for $i \leftarrow sz - 1$ to 1 **do**
  $tree[i] \leftarrow \min(tree[2i], tree[2i + 1])$
end for

Procedure: remove_root

Output: Minimum value in current segment tree

$x \leftarrow 1, ret \leftarrow tree[1]$
while $x < sz$ **do**
  if $tree[x] == tree[2x]$ **then**
    $x \leftarrow 2x$
  else
    $x \leftarrow 2x + 1$
  end if
  Set $tree[x \div 2] \leftarrow$ `INF` (mark removed)
end while
$tree[x] \leftarrow$ `INF`
while $x > 1$ **do**
  $x \leftarrow x \div 2$
  $tree[x] \leftarrow \min(tree[2x], tree[2x + 1])$
end while
return ret

Algorithm 5 Tournament Sort

Input: Iterators `begin`, `end` to an integer vector

Call `init(begin, end)` on segment tree
for $it \leftarrow$ `begin` to `end` $- 1$ **do**
  $*it \leftarrow$ `remove_root()` from segment tree
end for

Our implementation starts by calling a recursive function that tracks the recursion depth. If the depth exceeds $\log n$, the algorithm switches to heap sort to avoid quick sort's worst-case behavior. However, heap sort is only applied when the size of the current partition is larger than a predefined threshold; otherwise, the algorithm skips sorting for that partition. If the current partition size is small from the beginning, insertion sort is used instead due to its low overhead. Lastly, one additional insertion sort is performed on the entire array, which runs very quickly because the array is almost completely sorted at that point.

Intro sort is an in-place algorithm, as all sorting steps operate directly on the input array without requiring significant extra memory. It is not a stable algorithm, since heap sort does not preserve the relative order of equal elements. Nonetheless, it is widely used in practice, including the C++ STL, due to its strong performance guarantees and adaptability to various input characteristics.

On randomized input of size one million, Intro Sort demonstrated robust performance, completing in 521ms. This is slightly slower than pure Quick Sort, which finished in 495ms, but significantly

faster than 0-based non-recursive Heap Sort, which required 733ms. These results highlight the hybrid nature of Intro Sort: while it starts as Quick Sort and typically follows its efficient behavior, it incurs a slight overhead from tracking recursion depth and switching to Heap Sort when necessary. Nonetheless, the performance difference between Intro Sort and Quick Sort is minimal, and the worst-case safety net provided by Heap Sort makes Intro Sort a more reliable choice in practice, especially when input characteristics are unknown.

Procedure: intro_sort_rec(begin, end, rec_depth, max_depth)

Input: Iterators `begin`, `end`; integers `rec_depth`, `max_depth`

1: $n \leftarrow$ `end` $-$ `begin`
2: **if** $rec\_depth > max\_depth$ **then**
3: **if** $n > 16$ **then**
4: Run heap sort on `[begin, end)` (0-based, non-recursive)
5: **end if**
6: **return**
7: **end if**
8: **if** $n \leq 16$ **then**
9: Run insertion sort on `[begin, end)`
10: **return**
11: **end if**
12: $mid \leftarrow$ partition point of `[begin, end)`
13: Call `intro_sort_rec(begin, mid, rec_depth + 1, max_depth)`
14: Call `intro_sort_rec(mid + 1, end, rec_depth + 1, max_depth)`

Algorithm 6 Intro Sort

Input: Iterators `begin`, `end` to an integer vector

1: $n \leftarrow$ `end` $-$ `begin`
2: **if** $n \leq 1$ **then**
3: **return**
4: **end if**
5: $max\_depth \leftarrow \lceil \log_2(n) \rceil$
6: Call `intro_sort_rec(begin, end, 1, max_depth)`
7: Run insertion sort on range `[begin, end)`

4 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we evaluate execution time and sorting accuracy of all sorting algorithms discussed. All programs are implemented in standard C++ and compiled with `g++` in Windows 10. All experiments are performed on a machine with Intel i7-9750H 2.6GHz CPU. Codes and results are available on GitHub[1].

4.1 Experiment Settings

- **Datasets**: We generated integer-valued input data with varying sizes and characteristics. Four sizes were used: 1K, 10K, 100K, and 1M, and each size was tested with four types of input characteristics: sorted, reverse sorted, randomized, and partially sorted. As a result, each algorithm was tested on 16 datasets, covering every combination of size and characteristic. Randomized data were generated using the `mt19937` engine, producing integers uniformly between -10^9 and 10^9. Partially sorted data were created by starting from a sorted array and performing a number of random swaps equal to 1% of the input size.
- **Evaluation**: For simplicity, we only implemented and tested the ascending-order version of each sorting algorithm. However, since all of the algorithms are comparison-based, they can easily be extended to support other sorting orders. The execution time of each algorithm on each dataset was measured by running the algorithm 10 times on the same input and taking the average as the final result.
  However, for algorithms with time complexity $O(n^2)$, executing a single iteration on 1M-sized data took an excessively long time. For instance, bubble sort required approximately three hours to process a 1M-element reverse sorted array once. Therefore, for these algorithms on 1M-sized inputs, we reduced the number of iterations to two. The affected algorithms are: Bubble Sort, Insertion Sort, Binary Insertion Sort, Selection Sort, Library Sort, and Cocktail Shaker Sort.
  We also evaluated the sorting accuracy of each implementation using two methods. The first method checks whether the resulting array satisfies ascending order. The second method verifies that all values in the original input are preserved in the sorted output. For example, if there were three occurrences of the number 1 in the input, the same should hold true in the output. To validate this, we used two simple commutative hash functions: the sum of all elements and the XOR of all elements. If both hash values remain the same before and after sorting, we consider the result to be accurate. Both validation methods can be performed in $O(n)$ time, making them efficient for large inputs.

4.2 Experiment Results

Tables 1 and 2 present the execution time (in milliseconds) for each sorting algorithm on 1M-element inputs across four types of data characteristics. Several notable observations can be drawn from these results.

First, among all Merge Sort variations, the 5-way Merge Sort consistently achieved the best overall performance. As the number of divisions (k) increases, the recursion depth decreases, reducing the overhead of recursive calls. However, when k becomes too large, the cost of merging multiple subarrays outweighs this benefit. This trade-off results in a concave unimodal performance curve, with the 5-way Merge Sort striking the optimal balance. For instance, on both randomized and partially sorted data, it clearly outperforms the 3-way and even the 6-way Merge Sort. Interestingly, the 4-way Merge Sort also achieved nearly the same performance as the 5-way variation.

Another interesting result is that Comb Sort, despite its $O(n^2)$ time complexity, performs much better than expected in practice. Its gap-based comparison strategy allows it to resolve inversions more efficiently than traditional quadratic algorithms, resulting in

[1] https://github.com/dh28be/CSE331_Assignment1

performance close to that of several $O(n \log n)$ algorithms on large inputs.

In contrast, algorithms like Bubble Sort and Cocktail Shaker Sort scale very poorly on large inputs, taking several hours to complete on reverse sorted data. Although Library Sort has an expected time complexity of $O(n \log n)$, its performance proved unstable and significantly worse in practice due to the overhead of managing gaps and insertions.

Among the advanced algorithms, Tournament Sort stands out as the fastest on both randomized and partially sorted inputs. Quick Sort and Tim Sort also show excellent overall performance, with Tim Sort especially outstanding on partially sorted data, as expected through its design motivation.

Algorithm	1M Sorted (ms)	1M Reverse Sorted (ms)
Merge Sort	622	636
Non-recursive Merge Sort	722	729
3-way Merge Sort	440	453
Non-recursive 3-way Merge Sort	523	541
4-way Merge Sort	386	400
5-way Merge Sort	369	403
6-way Merge Sort	384	419
0-based Heap Sort	768	735
0-based Non-recursive Heap Sort	692	666
1-based Heap Sort	1170	1106
1-based Non-recursive Heap Sort	1092	1043
Bubble Sort	5338733	11460081
Insertion Sort	14	11934454
Binary Insertion Sort	11	19
Selection Sort	4035989	3942698
Quick Sort	425	439
Library Sort	1563922	1579371
Tim Sort	12	19
Cocktail Shaker Sort	5721133	11536298
Comb Sort	555	638
Tournament Sort	122	120
Intro Sort	461	493

Table 1: Execution time (ms) on 1M Sorted and Reverse Sorted data

Algorithm	1M Random (ms)	1M Partially Sorted (ms)
Merge Sort	738	650
Non-recursive Merge Sort	857	758
3-way Merge Sort	614	499
Non-recursive 3-way Merge Sort	680	581
4-way Merge Sort	570	460
5-way Merge Sort	572	456
6-way Merge Sort	617	479
0-based Heap Sort	809	771
0-based Non-recursive Heap Sort	733	691
1-based Heap Sort	1208	1167
1-based Non-recursive Heap Sort	1140	1086
Bubble Sort	9893667	5729008
Insertion Sort	5943698	159364
Binary Insertion Sort	2825038	74836
Selection Sort	4185059	4097804
Quick Sort	495	431
Library Sort	1556609	1533919
Tim Sort	601	192
Cocktail Shaker Sort	9259677	5811422
Comb Sort	992	892
Tournament Sort	263	126
Intro Sort	521	472

Table 2: Execution time (ms) on 1M Random and Partially Sorted data

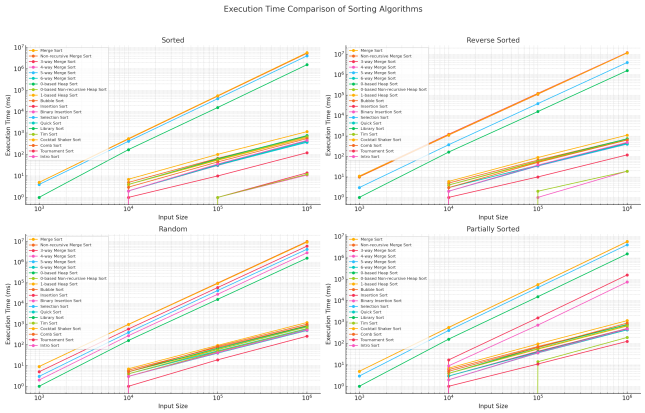

Figure 1: Execution time comparison of sorting algorithms across four data types (Sorted, Reverse Sorted, Random, Partially Sorted).

The trends observed in Figure 1 validates our earlier findings. As expected, algorithms with $O(n^2)$ complexity show quadratic growth, quickly becoming impractical for large input sizes. Among them, Comb Sort stands out for its unexpectedly efficient behavior. On the other hand, $O(n \log n)$ algorithms maintain reasonable scalability, with 5-way Merge Sort, Tim Sort, Quick Sort, and Tournament Sort demonstrating the best practical performance across various input characteristics.

These results clearly demonstrate the importance of selecting an appropriate sorting algorithm based on the characteristics of the input data. While theoretical complexity serves as a useful guideline, empirical performance often deviates due to practical factors such as implementation overhead and hardware behavior, which can significantly impact actual runtime.

References

[1] Naver D2. 2016. Introduction to Tim Sort. https://d2.naver.com/helloworld/0315536. Accessed: 2025-04-14.
[2] JusticeHui. 2019. IntroSort. https://justicehui.github.io/medium-algorithm/2019/03/24/IntroSort/. Accessed: 2025-04-14.
[3] JusticeHui. 2020. Heavy-Light Decomposition. https://justicehui.github.io/hard-algorithm/2020/01/24/hld/. Accessed: 2025-04-14.
[4] Nanarin. 2019. Cocktail Shaker Sort. https://nanarin.tistory.com/104. Accessed: 2025-04-14.
[5] ST-Lab. 2020. Java Sorting - Binary Insertion Sort. https://st-lab.tistory.com/262. Accessed: 2025-04-14.
[6] ST-Lab. 2020. Java Sorting - Tim Sort. https://st-lab.tistory.com/276. Accessed: 2025-04-14.
[7] Wikipedia contributors. 2024. Cocktail Shaker Sort. https://en.wikipedia.org/wiki/Cocktail_shaker_sort. Accessed: 2025-04-14.
[8] Wikipedia contributors. 2024. Comb Sort. https://en.wikipedia.org/wiki/Comb_sort. Accessed: 2025-04-14.
[9] Wikipedia contributors. 2024. Library Sort. https://en.wikipedia.org/wiki/Library_sort. Accessed: 2025-04-14.
[10] Wikipedia contributors. 2024. Tournament Sort. https://en.wikipedia.org/wiki/Tournament_sort. Accessed: 2025-04-14.

[1–10]