CSE331 - Assignment #1

Doyeol Oh (20211187)
UNIST
South Korea
ohdoyoel@unist.ac.kr

1 PROBLEM STATEMENT

This assignment involves the implementation and evaluation of a variety of sorting algorithms. The objectives are as follows:

(1) **Implement Six Conventional Sorting Algorithms**
- Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Heap Sort, Merge Sort
- Describe the design rationale and analyze the time complexity

(2) **Implement Six Contemporary Sorting Algorithms**
- Cocktail Shaker Sort, Comb Sort, Tournament Sort, Library Sort, Intro Sort, Tim Sort
- Provide pseudocode and each algorithm's strengths, limitations, and unique characteristics

(3) **Conduct Experimental Analysis**
- Use input data types: random, sorted, reverse sorted, and partially sorted
- Measure execution time, memory usage, and stability
- Test input sizes ranging from 1K to 1M, with 10 repetitions per case to average results

Full source code is available at:

https://github.com/ohdoyoel/unist_cse331_assignment_1/

2 BASIC SORTING ALGORITHMS

2.1 Bubble Sort

Algorithm 1 Bubble Sort

```
1:  procedure BubbleSort(A, n)
2:      for i = 1 to n − 1 do
3:          for j = 1 to n − i do
4:              if A[j] > A[j + 1] then
5:                  swap A[j] and A[j + 1]
6:              end if
7:          end for
8:      end for
9:  end procedure
```

Bubble Sort repeatedly compares and swaps adjacent elements, gradually moving larger values toward the end in each pass [3, 6].

2.1.1 *Design Rationale.*

- **In-Place and Stable:** Requires no additional memory and maintains relative order of equal elements.
- **Educational Simplicity:** Serves as a foundational algorithm for illustrating sorting principles.

2.1.2 *Complexity Analysis.*

- **Worst Case:**
  - Comparisons: $O(n^2)$ — all elements scanned in each pass.
  - Swaps: $O(n)$ — swaps occur when out-of-order elements are found.
- **Best Case:**
  - Comparisons: $O(n^2)$ — all comparisons still occur.
  - Swaps: $O(1)$ — none or minimal swaps if already sorted.
- **Average Case:**
  - Comparisons: $O(n^2)$ — full scans of unsorted portion.
  - Swaps: $O(n)$ — typically one per pass.
- **Space:** $O(1)$ auxiliary (in-place)

2.2 Selection Sort

Algorithm 2 Selection Sort

```
1:  procedure SelectionSort(A, n)
2:      for i = 1 to n − 1 do
3:          jMin ← i
4:          for j = i + 1 to n do
5:              if A[j] < A[jMin] then
6:                  jMin ← j
7:              end if
8:          end for
9:          if jMin ≠ i then
10:             swap A[i] and A[jMin]
11:         end if
12:     end for
13: end procedure
```

Selection Sort repeatedly selects the smallest element from the unsorted portion and places it in its correct position [6].

2.2.1 *Design Rationale.*

- **Minimum Select:** Selects the smallest element in each pass.
- **In-Place and Simple:** Requires no extra memory; follows a straightforward approach.
- **Few Swaps:** Performs at most $n - 1$ swaps, efficient when writes are costly.

2.2.2 *Complexity Analysis.*

- **Worst Case:**
  - Comparisons: $O(n^2)$ — full scan of unsorted part per pass.
  - Swaps: $O(n)$ — one swap per pass.
- **Best Case:**
  - Comparisons: $O(n^2)$ — all comparisons still made.
  - Swaps: $O(1)$ — swaps skipped if elements are already in correct positions.
- **Average Case:**
  - Comparisons: $O(n^2)$ — each element scanned to find minimum element.
  - Swaps: $O(n)$ — typical one-per-pass behavior.

- **Space:** $O(1)$ auxiliary (in-place)

2.3 Insertion Sort

Algorithm 3 Insertion Sort

```
1: procedure InsertionSort(A, n)
2:     for i = 2 to n do
3:         j ← i
4:         while j > 1 and A[j − 1] > A[j] do
5:             swap A[j] and A[j − 1]
6:             j ← j − 1
7:         end while
8:     end for
9: end procedure
```

Insertion Sort incrementally builds a sorted list by placing each element into its correct position [1, 6]. It is stable, in-place, and efficient on nearly sorted data, making it suitable for small inputs or hybrid algorithms.

2.3.1 *Design Rationale.*

- **Incremental Sorting:** Inserts each item into the sorted portion on the left.
- **In-Place and Stable:** Requires no extra space and preserves equal-value order.
- **Efficient on Nearly Sorted Input, adaptive behavior:** Performs well with minimal disorder.

2.3.2 *Complexity Analysis.*

- **Worst Case:**
  - Comparisons: $O(n^2)$ — every element compared with all sorted predecessors.
  - Shifts: $O(n^2)$ — maximum movement when array is reverse sorted.
- **Best Case:**
  - Comparisons: $O(n)$ — one per element in sorted input.
  - Shifts: $O(1)$ — no movement needed, negligible.
- **Average Case:**
  - Comparisons: $O(n^2)$ — moderate scanning per element.
  - Shifts: $O(n^2)$ — elements often need repositioning.
- **Space:** $O(1)$ auxiliary (in-place)

2.4 Quick Sort

Quick Sort recursively partitions the array around a pivot and places all the smaller elements on its left and the larger ones on its right [5]. It is in-place, efficient on average, and widely used in practice.

2.4.1 *Design Rationale.*

- **Divide and Conquer:** Recursively splits array into subproblems using a pivot.
- **In-Place Sorting:** Requires no auxiliary array; it requires only stack space.
- **Pivot Flexibility:** Pivot selection can be randomized or optimized. This will be discussed later in Experiment.

Algorithm 4 Quick Sort

```
1: procedure Partition(A, low, high)
2:     pivot ← A[high], i ← low − 1
3:     for j = low to high − 1 do
4:         if A[j] ≤ pivot then
5:             i ← i + 1
6:             swap A[i] and A[j]
7:         end if
8:     end for
9:     swap A[i + 1] and A[high]
10:    return i + 1
11: end procedure
12: procedure QuickSort(A, low, high)
13:    if low < high then
14:        p ← Partition(A, low, high)
15:        QuickSort(A, low, p − 1)
16:        QuickSort(A, p + 1, high)
17:    end if
18: end procedure
```

2.4.2 *Complexity Analysis.*

- **Worst Case:**
  - Comparisons: $O(n^2)$ — unbalanced partitions (e.g., sorted input with poor pivot).
  - Space: $O(n)$ auxiliary — due to deep recursion.
- **Best Case:**
  - Comparisons: $O(n \log n)$ — ideal partitioning.
  - Space: $O(\log n)$ auxiliary — shallow recursion.
- **Average Case:**
  - Comparisons: $O(n \log n)$ — reasonably balanced partitions.
  - Space: $O(\log n)$ auxiliary — typical recursion depth.

2.5 Heap Sort

Heap Sort "repeatedly extract the largest element and place it at the end [6]. It is in-place, not stable, and guarantees $O(n \log n)$ time complexity.

2.5.1 *Design Rationale.*

- **Heap-Based Selection:** Efficiently finds the largest element using a max-heap.
- **In-Place and Predictable:** Requires no extra memory.
- **Consistent Performance:** Always runs in $O(n \log n)$, regardless of input order.

2.5.2 *Complexity Analysis.*

- **Worst Case:**
  - Comparisons: $O(n \log n)$ — each element heapified with depth $\log n$.
  - Swaps: $O(n \log n)$ — per extraction and restore.
- **Best Case:**
  - Comparisons: $O(n \log n)$ — heap properties must still be enforced.
  - Swaps: $O(n)$ — if no reordering needed (e.g., equal elements).
- **Average Case:**
  - Comparisons: $O(n \log n)$ — balanced heap operations.

Algorithm 5 Heap Sort

```
procedure HEAPIFY(A, n, i)
    largest ← i, l ← 2i, r ← 2i + 1
    if l ≤ n and A[l] > A[largest] then
        largest ← l
    end if
    if r ≤ n and A[r] > A[largest] then
        largest ← r
    end if
    if largest ≠ i then
        swap A[i] and A[largest]
        HEAPIFY(A, n, largest)
    end if
end procedure
procedure HEAPSORT(A, n)
    for i = ⌊n/2⌋ downto 1 do
        HEAPIFY(A, n, i)
    end for
    for i = n downto 2 do
        swap A[1] and A[i]
        HEAPIFY(A, i − 1, 1)
    end for
end procedure
```

  - Swaps: $O(n \log n)$ — per heapify and rebuild.
- **Space:** $O(1)$ — operates entirely in-place

2.6 Merge Sort

Algorithm 6 Merge Sort

```
procedure MERGE(A, left, mid, right)
    Split A into L = A[left..mid], R = A[mid + 1..right] with sentinels
    Merge L and R back into A[left..right]
end procedure
procedure MERGESORT(A, left, right)
    if left < right then
        mid ← ⌊(left + right)/2⌋
        MERGESORT(A, left, mid);
        MERGESORT(A, mid + 1, right)
        MERGE(A, left, mid, right)
    end if
end procedure
```

Merge Sort recursively divides the input array into two halves, sorts each half, and then merges them into a fully sorted array [6]. It guarantees a worst-case time complexity of $O(n \log n)$ and is stable, preserving the relative order of equal elements.

2.6.1 Design Rationale.

- **Divide and Conquer:** Recursively splits and merges.
- **Stable Sorting:** Preserves relative order of equal elements.
- **Consistent Performance:** Always runs in $O(n \log n)$.
- **External Sorting Friendly:** Well-suited for external sorting on large datasets.

2.6.2 Complexity Analysis.

- **Worst Case:**
  - Comparisons: $O(n \log n)$ — across all levels of merging.
  - Moves: $O(n \log n)$ — all element copied at each merge step.
- **Best Case:**
  - Comparisons: $O(n)$ — naturally ordered input (e.g., in natural merge sort).
  - Moves: $O(n)$ — minimal when merging sorted runs.
- **Average Case:**
  - Comparisons: $O(n \log n)$ — balanced recursion and merges.
  - Moves: $O(n \log n)$ — all elements involved in every merge.
- **Space:** $O(n)$ auxiliary — requires temporary arrays for merging.

3 ADVANCED SORTING ALGORITHMS

3.1 Cocktail Shaker Sort

Algorithm 7 Cocktail Shaker Sort

```
procedure COCKTAILSHAKERSORT(A, n)
    start ← 1, end ← n, swapped ← true
    while swapped do
        swapped ← false
        for i = start to end − 1 do
            if A[i] > A[i + 1] then
                swap and set swapped ← true
            end if
        end for
        break if not swapped
        end ← end − 1, swapped ← false
        for i = end − 1 downto start do
            if A[i] > A[i + 1] then
                swap and set swapped ← true
            end if
        end for
        start ← start + 1
    end while
end procedure
```

Cocktail Shaker Sort is a bidirectional variant of Bubble Sort [3] that moves both large and small elements.

3.1.1 Design Rationale.

- **Bidirectional Scanning:** Alternates forward and backward passes to accelerate convergence.
- **Early Exit:** Ends immediately if array becomes sorted.
- **In-Place and Stable:** No extra memory required; preserves order of equal elements.

3.1.2 Complexity Analysis.

- **Worst Case:** $O(n^2)$ comparisons and swaps — reverse sorted input.
- **Best Case:** $O(n)$ comparisons, $O(1)$ swaps — already sorted.
- **Average Case:** $O(n^2)$ comparisons and swaps.
- **Space:** $O(1)$ auxiliary (in-place).

3.1.3 *Comparative Discussion.*

- **Distinct Characteristics:**
  - Performs forward and backward passes per iteration.
  - Stops early when no swaps occur in a full pass.
- **Strengths:**
  - Typically faster than Bubble Sort on nearly sorted data.
  - Simple to implement and stable.
- **Limitations:**
  - Still has $O(n^2)$ time in most cases.
  - Not suitable for large-scale inputs due to poor scalability.
- **Comparison to Bubble Sort:**
  - Both maintain stability and work in-place.
  - Cocktail Shaker Sort tends to complete in fewer passes because of its two-way traversal.

3.2 Comb Sort

Algorithm 8 Comb Sort

```
procedure CombSort(A, n)
    gap ← n, shrink ← 1.3, swapped ← true
    while gap > 1 or swapped do
        gap ← max(1, ⌊gap/shrink⌋)
        swapped ← false
        for i = 1 to n − gap do
            if A[i] > A[i + gap] then
                swap A[i] and A[i + gap], swapped ← true
            end if
        end for
    end while
end procedure
```

Comb Sort improves Bubble Sort by comparing elements at a shrinking gap, allowing faster movement of small values [6].

3.2.1 *Design Rationale.*

- **Gap-Based Comparison:** Moves small values forward early by comparing distant elements ("turtles").
- **In-Place:** No extra memory required.

3.2.2 *Complexity Analysis.*

- **Worst Case:** $O(n^2)$ comparisons and swaps.
- **Best Case:** $O(n \log n)$ comparisons, $O(n)$ swaps.
- **Average Case:** Typically sub-quadratic; $O(n \log n)$ swaps.
- **Space:** $O(1)$ auxiliary (in-place).

3.2.3 *Comparative Discussion.*

- **Distinct Characteristics:**
  - Shrinks gap each pass; final pass equals to Bubble Sort.
- **Strengths:**
  - Faster than Bubble or Cocktail Sort, especially on disordered or reverse-sorted inputs.
  - Simple and in-place.
- **Limitations:**
  - Performance can degrade if the shrink factor is not tuned properly (e.g., too small slows convergence).
- **Comparison:**
  - Outperforms Bubble and Cocktail Sort on average.
  - Simpler but less stable.

3.3 Tournament Sort

Algorithm 9 Tournament Sort

```
procedure TournamentSort(A, n)
    m ← smallest power of 2 ≥ n, build tree T[1 . . . 2m − 1]
    Fill leaves T[m . . . m + n − 1] ← A[1 . . . n], rest with ∞
    for i = m − 1 downto 1 do
        T[i] ← min(T[2i], T[2i + 1])
    end for
    for k = 1 to n do
        A[k] ← T[1]
        Locate and remove leaf with T[1], update path to root
    end for
end procedure
```

Tournament Sort simulates a winner tree to repeatedly extract the minimum value [8]. It performs well in scenarios requiring repeated selection, such as k-way merging.

3.3.1 *Design Rationale.*

- **Winner Tree:** Uses a binary tree to repeatedly identify the minimum efficiently.
- **Log-Time Update:** After removing a winner, only the path to the root is updated, avoiding full re-evaluation.
- **Deterministic Behavior:** Reuses past comparisons for consistency; not stable unless tie-breaking logic is added.

3.3.2 *Complexity Analysis.*

- **Worst / Best / Average Case:**
  - Comparisons: $O(n \log n)$ — each of the n extractions requires a $\log n$ update.
  - Moves: $O(n)$ — each element is moved once.
- **Tree Initialization:** $O(n)$ to fill leaves, $O(n)$ to compute internal nodes.
- **Space:** $O(n)$ auxiliary — due to the explicit binary tree.

3.3.3 *Comparative Discussion.*

- **Distinct Characteristics:**
  - Simulates tournament-style selection, reusing previous comparisons.
- **Strengths:**
  - Guarantees $O(n \log n)$ comparisons.
  - Particularly effective in external sorting or k-way merges where repeated minimum selection is required.
- **Limitations:**
  - Not in-place; requires $O(n)$ additional memory.
  - Does not preserve the relative order of equal elements unless extended with tie-breaking logic.
  - More complex than heap-based approaches.
- **Comparison:**
  - **vs. Selection Sort:** Faster due to reduced redundant comparisons; logarithmic updates vs linear scans.
  - **vs. Heap Sort:** Avoids repeated comparison of the same elements but consumes more memory.

3.4 Library Sort

Algorithm 10 Library Sort

```
1: procedure LibrarySort(A, n)
2:     Initialize sparse array S[1 ... 2n] ← ∞
3:     S[1] ← A[1]
4:     for each round i do
5:         Rebalance S over first 2^(i−1) elements
6:         for new elements do
7:             Locate insertion index using binary search (skip gaps)
8:             Insert element at nearest available position
9:         end for
10:    end for
11:    Copy valid entries from S to A, which is non-empty
12: end procedure
```

Library Sort uses a gapped array and binary search to reduce shifting costs during insertions [1]. It balances speed and stability via amortized techniques.

3.4.1 *Design Rationale.*

- **Gapped Insertion:** Avoids full shifts during insert.
- **Binary Search:** Quickly finds insertion points.
- **Amortized Rebalancing:** Spreads movement cost over multiple operations.

3.4.2 *Complexity Analysis.*

- **Worst:** $O(n \log^2 n)$ comparisons and moves, due to binary search + rebalancing overhead and dense insertions.
- **Average:** $O(n \log n)$ for both comparisons and moves.
- **Best:** $O(n \log n)$ comparisons, $O(n)$ moves.
- **Space:** $O(n)$ total, $O(\varepsilon n)$ extra.

3.4.3 *Comparative Discussion.*

- **Distinct Characteristics:**
  - Uses gaps in the array to reduce shifts during insertion.
  - Periodic rebalancing keeps data evenly distributed.
- **Strengths:**
  - Outperforms Insertion Sort on large or random data by reducing shifts.
  - Simple; can be stable with proper gap handling.
- **Limitations:**
  - Requires additional space and is less cache-friendly.
  - More complex to implement than traditional insertion-based approaches.
- **Comparison to Insertion Sort:**
  - Avoids full-element shifts using gaps.
  - Scales better with larger input sizes, though both can be stable with proper handling.

3.5 Intro Sort

Intro Sort integrates Quick Sort, Heap Sort, and Insertion Sort [7]. It starts with Quick Sort for speed, switches to Heap Sort to avoid degenerate recursion, and uses Insertion Sort for small partitions. This hybrid design guarantees fast and reliable performance.

Algorithm 11 Intro Sort

```
1: procedure IntroSortRecursive(A, lo, hi, depthLimit)
2:     if hi − lo ≤ typically 16 then
3:         InsertionSort(A, lo, hi)
4:     else if depthLimit = 0 then
5:         HeapSort(A, lo, hi)
6:     else
7:         p ← Partition(A, lo, hi)
8:         IntroSortRecursive(A, lo, p − 1, depthLimit − 1)
9:         IntroSortRecursive(A, p + 1, hi, depthLimit − 1)
10:    end if
11: end procedure
12: procedure IntroSort(A, n)
13:    depthLimit ← 2 · ⌊log2(n)⌋
14:    IntroSortRecursive(A, 0, n − 1, depthLimit)
15: end procedure
```

3.5.1 *Design Rationale.*

- **Hybrid Logic:** Adapts strategy based on depth and size.
- **Fallback Safety:** Heap Sort ensures worst-case bound.
- **Practicality:** Used in `std::sort()` (C++ STL).

3.5.2 *Complexity Analysis.*

- **Worst:** $O(n \log n)$ comparisons and moves.
- **Best:** $O(n \log n)$ comparisons, $O(n)$ moves.
- **Average:** $O(n \log n)$ for both.
- **Space:** $O(\log n)$ for stack.

3.5.3 *Comparative Discussion.*

- **Distinct Characteristics:**
  - Uses Quick Sort by default, Heap Sort when needed.
  - Insertion Sort optimizes small inputs.
- **Strengths:**
  - Robust: avoids Quick Sort's pitfalls.
  - Fast in most cases, reliable in worst-case.
- **Limitations:**
  - Slightly complex logic.
  - Requires depth monitoring.
- **Comparison:**
  - **Quick Sort:** Intro Sort adds safety net.
  - **Heap Sort:** Only used when needed.
  - **Insertion Sort:** Enhances performance on small parts.

3.6 Tim Sort

Tim Sort is a stable hybrid of Merge Sort and Insertion Sort [2], optimized for real-world data by detecting natural runs and using efficient merging techniques like galloping.

3.6.1 *Design Rationale.*

- **Run Detection:** Finds pre-sorted segments to reduce effort.
- **Hybrid Merge Strategy:** Insertion Sort for short runs, and Merge Sort for combining runs, including galloping mode to accelerate merging of long runs.
- **Stable:** Maintains original order of equal elements.

3.6.2 *Complexity Analysis.*

Algorithm 12 Tim Sort

```
procedure TimSort(A, n)
    RUN ← typically 32
    for i = 0 to n − 1 by RUN do
        InsertionSort(A, i, min(i + RUN − 1, n − 1))
    end for
    for size = RUN to n by 2 · size do
        for left = 0 to n − 1 by 2 · size do
            mid ← min(left + size − 1, n − 1)
            right ← min(left + 2 · size − 1, n − 1)
            if mid < right then
                Merge(A, left, mid, right)
            end if
        end for
    end for
end procedure
```

- **Worst:** $O(n \log n)$ comparisons and moves.
- **Best:** $O(n)$ — when input consists of long natural runs (e.g., nearly sorted).
- **Average:** $O(n \log n)$ overall.
- **Space:** $O(n)$ auxiliary for merging.

3.6.3 *Comparative Discussion.*

- **Distinct Characteristics:**
  - Detects and merges natural runs adaptively.
  - Applies Insertion Sort to short segments.
- **Strengths:**
  - Excellent on partially sorted data.
  - Stable and adaptive; used in Python and Java.
- **Limitations:**
  - Requires extra memory.
  - Implementation is non-trivial.
- **Comparison:**
  - **Merge Sort:** Tim Sort is input-aware.
  - **Insertion Sort:** Used for small, local optimizations.

4 EXPERIMENTAL RESULTS AND ANALYSIS

4.1 Experimental Design and Implementation

I implemented a modular C++ benchmarking framework to evaluate sorting algorithms [2]. Each algorithm sorts an array of `TaggedValue` structures:

```
struct TaggedValue {
    int value;
    int originalIndex;
};
```

This enables correctness and stability checks, along with performance evaluation:

- **Correctness:** Verifies non-decreasing order by scanning adjacent elements.
- **Stability:** [4] Confirms equal elements retain their original relative order based on stored indices.
- **Time:** Execution time is measured using `std::chrono`, then computing the duration between before and after the sorting.
- **Memory:** Memory usage is tracked using `Windows API`, calling `GetProcessMemoryInfo()` before and after the sorting.

For the full implementation of these measurement functions and the implementation below, please refer to the GitHub repository.

4.1.1 *Input Generation.*

I tested four data patterns—random, sorted, reverse sorted, and partially sorted—across sizes 10^3, 10^4, 10^5, 10^6. Each configuration was repeated 10 times.

- **File Naming:** Format — `type_size_trial.txt` (e.g., `random_1000_4.txt`).
- **Total Files:** 160 input sets (4 types × 4 sizes × 10 trials).

4.1.2 *Benchmarking Framework.*

- **Batch Execution:**
  Automatically parses and runs grouped test files.
- **Metrics:**
  Execution Time, Memory Usage, Correctness, and Stability.
- **Averaging:**
  10 trials per Type and Size, summarized per algorithm.
- **Formatted Output:**
  Clean per-group display of metrics.

4.1.3 *Special Attention in Quick Sort.*

The standard Quick Sort uses the Lomuto scheme with the last element as pivot, which led to stack overflow or severe slowdowns on sorted and reversed inputs due to unbalanced partitions.

To mitigate this, I implemented **Quick2**, which adopts **median-of-three pivoting**—choosing the pivot as the median of the first, middle, and last elements. This resulted in more balanced splits and significantly improved stability.

4.2 Experimental Results

To evaluate algorithmic behavior, I conducted benchmarks across four input types (random, sorted, reverse sorted, partially sorted) and input sizes ranging from 10^3 to 10^6.

4.2.1 *Correctness and Stability.*

All algorithms consistently produced **correct**, non-decreasing output across all datasets.

Stability was verified by checking whether elements with equal values preserved their original order. If a single violation was observed in any trial, the algorithm was classified as unstable.

Results:

- **Stable:** Bubble Sort, Insertion Sort, Merge Sort, Cocktail Shaker Sort, Library Sort, Tim Sort
- **Unstable:** Selection Sort (8/16), Quick Sort (11/16), Heap Sort (12/16), Comb Sort (8/16), Tournament Sort (5/16), Intro Sort (12/16)

4.2.2 *Execution Time.*

The following four tables respectively show the execution time of each algorithm for **Random Input**, **Sorted Input**, **Reverse Sorted Input**, and **Partially Sorted Input**, across different input sizes (10^3, 10^4, 10^5, and 10^6).

Bold and † denote the top two performers in each column, with **blue indicating the best** and *red italic representing the worst*.

Algorithm	10^3	10^4	10^5	10^6
		Random		
Bubble	*0.00246*	*0.32273*	*36.43039*	*3942.56713*
Selection	0.00125	0.14129	10.86712	1117.92214
Insertion	0.00158	0.15556	15.71373	1843.35003
Quick	0.00020	0.00159	0.01954	0.30220
Quick2	0.00030	0.00159	**0.01517**†	0.32493
Heap	**0.00010**†	0.00232	0.02964	0.58660
Merge	0.00020	0.00247	0.03454	0.33370
Cocktail	0.00218	0.24523	27.53963	3017.76145
Comb	0.00015	**0.00151**†	0.02015	0.23406
Tournament	0.00073	0.12257	9.53680	1028.36774
Library	0.00030	0.00561	0.04549	8.62014
Intro	0.00032	**0.00116**†	**0.01604**†	**0.23182**†
Tim	**0.00009**†	0.00169	0.01832	**0.19882**†
		Sorted		
Bubble	0.00105	0.09788	10.29538	*1229.06407*
Selection	0.00118	0.11514	*10.80210*	1223.53202
Insertion	**0.00000**†	**0.00000**†	**0.00030**†	**0.00230**†
Quick	*0.00307*	*0.36235*	*overflow*	*overflow*
Quick2	**0.00009**†	0.00092	0.00766	0.10766
Heap	0.00010	0.00227	0.02000	0.37884
Merge	0.00010	0.00200	0.02892	0.23155
Cocktail	**0.00000**†	**0.00000**†	**0.00041**†	**0.00209**†
Comb	0.00010	0.00072	0.00911	0.09816
Tournament	0.00092	0.10457	10.40200	1149.62712
Library	0.00090	0.07347	8.79863	831.32801
Intro	0.00039	0.00449	0.04623	0.57818
Tim	**0.00000**†	**0.00067**†	0.00786	0.10932
		Reverse		
Bubble	*0.00287*	0.32841	33.64382	*3840.68543*
Selection	0.00115	0.11601	11.70575	1251.84812
Insertion	0.00277	0.31612	32.24314	3768.46693
Quick	0.00184	0.25916	*overflow*	*overflow*
Quick2	0.00010	**0.00111**†	**0.01472**†	0.24183
Heap	0.00010	0.00173	0.01989	0.34510
Merge	0.00016	0.00202	0.03002	0.23102
Cocktail	0.00279	*0.34173*	*34.20896*	3063.47829
Comb	**0.00000**†	**0.00098**†	**0.01019**†	**0.11948**†
Tournament	0.00076	0.10831	9.56592	1060.79032
Library	0.00152	0.08883	9.72443	1045.398545
Intro	0.00035	0.00323	0.03700	0.49643
Tim	**0.00009**†	0.00175	0.01924	**0.20652**†
		Partial		
Bubble	*0.00203*	*0.21153*	*12.34734*	1096.56131
Selection	0.00106	0.10805	11.05071	*1206.44366*
Insertion	0.00100	0.09400	1.48714	2.15748
Quick	**0.00000**†	0.00155	*overflow*	*overflow*
Quick2	**0.00000**†	0.00149	**0.01032**†	**0.11037**†
Heap	0.00047	0.00229	0.02380	0.35624
Merge	0.00036	0.00248	0.03624	0.27291
Cocktail	0.00122	0.12514	4.45257	46.04605
Comb	0.00025	**0.00137**†	0.01174	0.12076
Tournament	0.00105	0.11025	10.01889	1127.62456
Library	0.00042	0.01033	6.67287	799.21829
Intro	**0.00000**†	0.00137	0.05661	0.63857
Tim	**0.00020**†	**0.00111**†	**0.01031**†	**0.10683**†

Table 1: Execution Time (Sec) for Each Algorithm

4.2.3 *Memory Usage.*

Memory usage per algorithm is shown below. Entries with noise, zero values, or inconsistency were omitted for clarity and space. *Red italic indicates the worst among remaining valid values.*

Algorithm	Input Type	10^3	10^4	10^5	10^6
Quick	Random	409	0	0	0
	Sorted	0	4505	*overflow*	*overflow*
	Reverse	*2867*	41779	*overflow*	*overflow*
	Partial	0	0	*overflow*	*overflow*
Quick2	Random	0	0	0	0
	Sorted	0	0	0	0
	Reverse	0	0	0	0
	Partial	409	0	0	0
Merge	Random	0	29491	53657	0
	Sorted	0	51200	67993	0
	Reverse	0	45875	72908	0
	Partial	2457	39731	296960	*305561*
Tournament	Random	0	25804	0	409
	Sorted	0	51609	0	1638
	Reverse	0	51609	0	409
	Partial	819	77824	0	0
Library	Random	0	162201	287539	0
	Sorted	0	*180224*	300236	5734
	Reverse	0	*180224*	287539	0
	Partial	2048	0	*313753*	0
Intro	Random	0	0	0	0
	Sorted	0	0	0	0
	Reverse	0	0	0	0
	Partial	0	0	0	0
Tim	Random	0	70680	74956	112230
	Sorted	0	60211	24166	139264
	Reverse	0	68403	32768	139264
	Partial	0	73318	67993	106496

Table 2: Memory Usage (Byte) Across Input Types and Sizes

4.3 Experimental Interpretation

4.3.1 *Execution Time Analysis.*

1. Complexity Trends

Quadratic sorts (Bubble, Selection, etc.) degrade sharply past 10^4. Log-linear sorts (Quick2, Tim, Intro, etc.) scale smoothly up to 10^6. (e.g., Bubble Sort took over 3900s at 10^6 vs. Tim Sort under 0.2s)

2. Algorithm Highlights

- **Insertion:** Performs well on sorted data.
- **Quick:** Unstable on ordered inputs; overflows.
- **Quick2:** Median-of-three pivot ensures stability.
- **Comb:** Fastest among simple sorts on reverse data.
- **Intro:** Balanced and consistent across inputs.
- **Tim:** Excels on structured input with run detection.

3. Best by Input Type

- **Random:** Tim, Quick2, Intro
- **Sorted:** Insertion > Cocktail > Tim
- **Reverse:** Comb > Tim, Quick2
- **Partial:** Tim > Quick2, Intro, Comb

4. Practical Summary

For general-purpose use, Tim, Quick2, and Intro offer high reliability and performance. Comb is an elegant upgrade to basic sorts, especially effective on reverse or disordered input. Classic Quick should be avoided without pivot enhancements.

5. Scalability

Only Tim and Quick2 completed all trials for 10^6 elements without failure. Intro and Comb maintained good scaling, particularly in non-worst-case distributions.

4.3.2 *Memory Usage Analysis.*

1. Overall Patterns

Intro and Quick2 use near-zero memory via in-place sorting. Merge, Library, and Tim show higher usage due to auxiliary buffers. (e.g., temporary arrays for merging or rebalancing.) Quick and Merge risk overflows on structured input.

2. Algorithm Highlights

- **Quick:** Stack overflow on sorted/reverse input.
- **Quick2:** Minimal and stable.
- **Merge:** Usage grows with input size.
- **Library:** Memory-heavy on structured data.
- **Tim:** Moderate to high, depending on input.

3. Input Sensitivity

Ordered and partially sorted inputs caused spikes in Merge and Library Sort; random input showed stable usage.

4. Takeaways

Intro and Quick2 are memory-efficient. Tim is balanced. Merge and Library need careful use.

5. Measurement Limitations

Some results from `GetProcessMemoryInfo()` could not be trusted, due to irregular behavior in memory tracking, such as zero values or implausible reductions.

4.3.3 *Can We Do Better? TimSort with Parallel Merge.*

To enhance performance, I implemented **Tim2**, a multithreaded variant of Tim Sort. However, it underperformed compared to standard Tim Sort (e.g., for random input of size 10^3, Tim: 0.00242s, Tim2: 0.02085s).

Issues and Fixes:

Thread Overhead Excessive thread creation slowed execution. *Fix:* Apply thresholds and use thread pooling.

Instability Failures occurred around $n = 10^4$ due to premature parallelism. *Fix:* Introduce granularity control.

Memory Spikes Merging caused sharp memory usage increases. *Fix:* Reuse buffers where possible.

Conclusion: Without careful granularity and resource management, parallelism can hinder rather than help.

5 Conclusion

This assignment explored twelve sorting algorithms through implementation and empirical benchmarking. Tests spanned various input types (random, sorted, reverse, partial) and sizes (from 10^3 to 10^6), with metrics including execution time, memory usage, correctness, and stability.

5.1 Summary of Findings

- **Tim Sort** consistently outperformed others on sorted and partially sorted data, thanks to its adaptive merging.
- **Intro Sort** and **Quick2** showed strong general performance and robustness against unbalanced partitioning.
- **Quadratic-time algorithms** (e.g., Bubble, Selection, Cocktail) were only feasible at small scales [6].
- **Merge and Library Sorts** showed increased memory usage; Library peaked at 300KB for 10^5 inputs.

5.2 Limitations and Improvement Directions

During experimentation, several challenges emerged:

- **Quick Sort crashes** occurred on sorted/reverse inputs due to unbalanced recursion with the Lomuto scheme. Switching to median-of-three pivoting (Quick2) mitigated this.
- **Memory measurements** using `GetProcessMemoryInfo()` showed noise due to background processes and granularity.

To address these:

- Use stack size limits or tail recursion optimizations to prevent Quick Sort crashes.
- Isolate benchmarking using dedicated environments or containerized sandboxes for cleaner memory readings.

5.3 Strategic Takeaway

Algorithm performance is shaped not just by complexity but also input patterns and implementation. Tim and Intro Sorts emerge as safe defaults, while Comb Sort offers a space-efficient option. Avoid naive Lomuto Quick Sort; Library Sorts may strain memory.

5.4 Recommendations

- **Random Input:** Tim or Intro Sort.
- **Sorted Input:** Tim or Insertion Sort.
- **Reverse sorted Input:** Comb or Tim Sort; avoid Lomuto Quick, due to unbalanced partitioning.
- **Partially sorted Input:** Tim or Quick with median-of-three.

5.5 Future Work

- Multithreaded sorting via OpenMP or TBB (see Tim2)
- Profiling cache behavior and exploring SIMD variants

References

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

[2] Jon Louis Bentley and M. Douglas McIlroy. 1993. Engineering a Sort Function. *Software: Practice and Experience* 23, 11 (1993), 1249–1265.

[3] Witold Dobosiewicz. 1980. An Efficient Variation of Bubble Sort. *Inform. Process. Lett.* 11, 1 (1980), 5–6.

[4] Edward H. Friend. 1956. Sorting on Electronic Computer Systems. *J. ACM* 3, 3 (1956), 134–168.

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

[6] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley.

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

[8] Alexander Stepanov and Andrew Kershenbaum. 1986. *Using Tournament Trees to Sort.* CATT Technical Report 86–13. Polytechnical Institute of New York.