Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

Laboratory work 2: Empirical Analysis of Comparison-Based and Non-Standard Sorting Algorithms

Author:
st. gr. FAF-241 Moisei Daniel
Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Contents

ALGORITHM ANALYSIS

Objective

Study, implement, and analyze the performance of various sorting algorithms including MergeSort, QuickSort, HeapSort, and PatienceSort under different input conditions and data types.

Tasks:

1. Implement MergeSort, QuickSort (Hoare), HeapSort, and PatienceSort;
2. Analyze QuickSort partitioning: comparing Lomuto and Hoare schemes;
3. Evaluate algorithms against Random, Sorted, Reversed, and Duplicate data;
4. Analyze execution time complexity for Best, Average, and Worst cases;
5. Test algorithm robustness against extreme cases (Negatives, Doubles);
6. Deduce conclusions based on empirical graphs and theoretical knowledge.

Theoretical Notes:

Empirical analysis is a critical tool for understanding how algorithms behave on real hardware, often revealing nuances that Big O notation overlooks. Sorting algorithms are traditionally categorized by their average and worst-case complexities.

In this analysis, we focus on:

1. **Divide and Conquer:** Breaking problems into sub-problems (MergeSort, QuickSort).
2. **Heap Data Structures:** Utilizing priority-based ordering (HeapSort).
3. **Patience Sorting:** A non-standard approach inspired by the card game Solitaire, often used to find the Longest Increasing Subsequence.

Execution time is the primary metric here, as it captures the overhead of recursive calls, memory allocation, and the impact of input ordering on pivot selection or pile creation.

Introduction:

Sorting is one of the most fundamental operations in computer science. While simple algorithms like Bubble Sort offer $O(n^2)$ complexity, more advanced algorithms aim for

$O(n \log n)$. However, the actual performance can vary wildly based on the initial state of the data.

In this laboratory, we implement four distinct algorithms. **MergeSort** is known for its stability and guaranteed $O(n \log n)$ time. **QuickSort** is often faster in practice but can degrade to $O(n^2)$ if the pivot selection is poor. **HeapSort** provides a reliable $O(n \log n)$ performance without the extra memory requirements of MergeSort. Finally, **PatienceSort** offers a unique perspective on sorting by organizing data into "piles" and merging them.

Comparison Metric:

The comparison metric for this laboratory work is the execution time measured in milliseconds (ms) using the `Stopwatch` class in C#. Each test is averaged over multiple runs to minimize system noise.

Input Format:

To thoroughly stress the algorithms, four types of input arrays were generated:

- **Random:** Elements in no particular order.
- **Sorted:** Elements already in ascending order (tests best/worst cases).
- **Reversed:** Elements in descending order.
- **Duplicates:** Arrays containing many repeating values.

The array sizes range from 10 to 15,000 elements to observe the growth curves.

IMPLEMENTATION & SCENARIO ANALYSIS

The algorithms were implemented in C#. Below we detail the implementation logic, complexity scenarios, and how each handles extreme data cases.

MergeSort:

MergeSort is a stable, divide-and-conquer algorithm that guarantees performance by splitting the data into halves and merging them.

Extended Pseudocode:

Listing 1: MergeSort Recursive Structure and Merge Logic

```
MergeSort(A, p, r):
    if p < r:
        mid = (p + r) / 2
        MergeSort(A, p, mid)
        MergeSort(A, mid + 1, r)
        Merge(A, p, mid, r)

Merge(A, p, q, r):
    L = A[p...q], R = A[q+1...r] // Create copies
    i = 0, j = 0, k = p
    while i < L.length and j < R.length:
        if L[i] <= R[j]: A[k] = L[i]; i++
        else: A[k] = R[j]; j++
        k++
    // Append remaining elements
```

Implementation:

```csharp
public static class MergeSort
{
    public static void Sort(int[] arr, int left, int right)
    {
        if (left < right)
        {
            int mid = left + (right - left) / 2;
            Sort(arr, left, mid);
            Sort(arr, mid + 1, right);
            Merge(arr, left, mid, right);
        }
    }

    public static void Sort(int[] arr)
    {
        Sort(arr, 0, arr.Length - 1);
    }

    private static void Merge(int[] arr, int l, int m, int r)
    {
        int[] leftArr = arr[l..(m + 1)];
        int[] rightArr = arr[(m + 1)..(r + 1)];
        int i = 0, j = 0, k = l;

        while (i < leftArr.Length && j < rightArr.Length)
        {
            arr[k++] = (leftArr[i] <= rightArr[j]) ? leftArr[i++] : rightArr[j++];
        }

        while (i < leftArr.Length) arr[k++] = leftArr[i++];
        while (j < rightArr.Length) arr[k++] = rightArr[j++];
    }
}
```

Figure 1: MergeSort implementation in C#

Complexity Scenarios:

- **Best/Average/Worst Case:** Always $O(n \log n)$. Because the split and merge process is deterministic and doesn't depend on the values, the timing is extremely stable.

Extreme Cases:

- **Negatives/Doubles:** Works perfectly. Since it uses standard comparison operators, floating point precision or negative signs do not affect the logic.
- **Large Duplicates:** Highly efficient. Stability ensures equal values stay in their relative positions.

```
--- MERGESORT ---
Random Array (15000):
  Average: 5.130 ms  [ 2.936, 2.711, 9.742 ]
  Sorted: ✓ Yes
Sorted Array (15000):
  Average: 1.485 ms  [ 1.557, 1.458, 1.441 ]
  Sorted: ✓ Yes
Reversed Array (15000):
  Average: 1.496 ms  [ 1.517, 1.482, 1.488 ]
  Sorted: ✓ Yes
Duplicates Array (15000):
  Average: 1.964 ms  [ 1.986, 1.939, 1.967 ]
  Sorted: ✓ Yes
```

Figure 2: Execution times for 15,000 elements across various input types

Results:

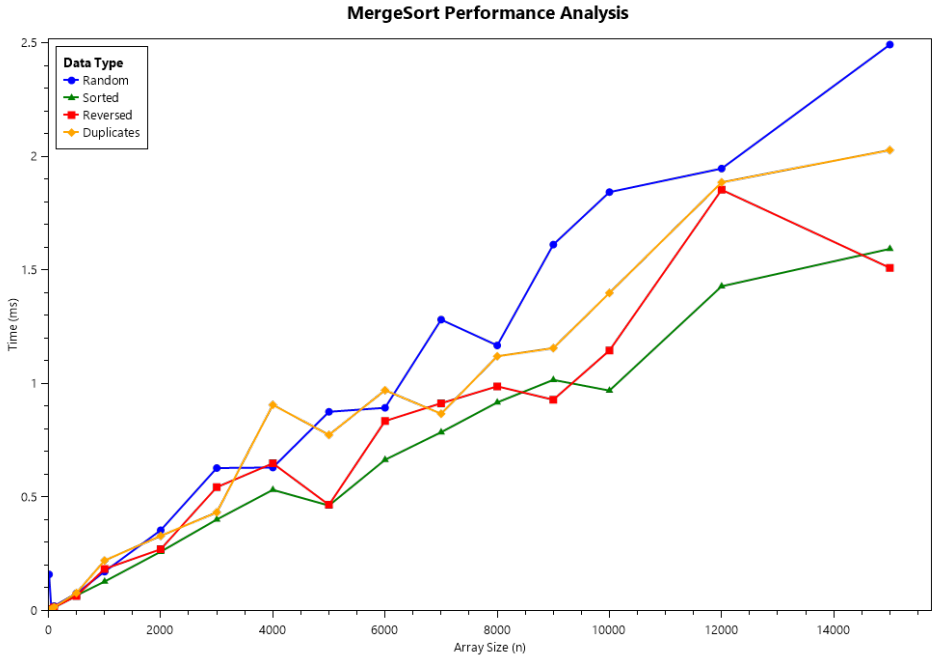


Figure 3: MergeSort Performance Graph

QuickSort: Lomuto vs Hoare Partitioning

QuickSort's efficiency depends entirely on its partitioning scheme. While both schemes achieve $O(n \log n)$ average time, their performance differs in practice.

1. Lomuto Partition (Standard):

Listing 2: Lomuto Partition Scheme

```
LomutoPartition(A, p, r):
    pivot = A[r] // Pivot is the last element
    i = p - 1
    for j = p to r - 1:
        if A[j] <= pivot:
            i = i + 1
            swap A[i], A[j]
    swap A[i + 1], A[r]
    return i + 1
```

2. Hoare Partition (Optimized):

Listing 3: Hoare Partition Scheme

```
HoarePartition(A, p, r):
    pivot = A[p] // Pivot is the first element
    i = p - 1
    j = r + 1
    while true:
        do i++ while A[i] < pivot
        do j-- while A[j] > pivot
        if i >= j: return j
        swap A[i], A[j]
```

3. QuickSort Recursive Structure:

Listing 4: QuickSort Recursive Implementation

```
QuickSort(A, left, right):
    if left < right:
        // Partition the array and get the split point
        pivotIndex = HoarePartition(A, left, right)

        // Recursively sort the two halves
        QuickSort(A, left, pivotIndex)
        QuickSort(A, pivotIndex + 1, right)
```

Implementation:

```
public static class QuickSort
{
    public static void Sort(int[] arr, int left, int right)
    {
        if (left < right)
        {
            int pivotIndex = HoarePartition(arr, left, right);
            Sort(arr, left, pivotIndex);
            Sort(arr, pivotIndex + 1, right);
        }
    }

    public static void Sort(int[] arr)
    {
        Sort(arr, 0, arr.Length - 1);
    }

    private static int HoarePartition(int[] arr, int left, int right)
    {
        int pivot = arr[left]; //Choosing first element as pivot
        int i = left - 1;
        int j = right + 1;
        while (true)
        {
            //Move i to the rigth until finding element >= pivot
            do
            {
                i++;
            } while (arr[i] < pivot);

            //Move j to the left until finding element <= pivot
            do
            {
                j--;
            } while (arr[j] > pivot);

            //If pointers crossed, return partition index
            if (i >= j)
                return j;

            //Swap elements at i and j
            Swap(arr, i, j);
        }
    }
```

Figure 4: QuickSort with Hoare Partition in C#

Why Hoare is faster: Hoare's scheme is more efficient because it performs, on average, three times fewer swaps than Lomuto. Furthermore, Hoare's pointers move towards each other, naturally handling duplicate elements and already-sorted segments with fewer operations. Lomuto often performs redundant swaps even when elements are already in the correct side of the pivot.

Complexity Scenarios:

- **Best Case:** $O(n \log n)$ - When the pivot consistently lands in the middle.

- **Average Case:** $O(n \log n)$.
- **Worst Case:** $O(n^2)$ - Occurs when the pivot is the smallest or largest element (Sorted or Reversed data).

Extreme Cases:

- **Negatives/Doubles:** Handles them well.
- **Stability:** QuickSort is **unstable**; relative order of equal elements (like doubles with different metadata) is not preserved.

```
--- QUICKSORT ---
Random Array (15000):
  Average: 1.655 ms  [ 1.889, 1.537, 1.538 ]
  Sorted: ✓ Yes
Sorted Array (15000):
  Average: 109.609 ms  [ 110.621, 109.104, 109.101 ]
  Sorted: ✓ Yes
Reversed Array (15000):
  Average: 111.189 ms  [ 111.674, 110.650, 111.244 ]
  Sorted: ✓ Yes
Duplicates Array (15000):
  Average: 1.204 ms  [ 1.380, 1.127, 1.106 ]
  Sorted: ✓ Yes
```

Figure 5: Execution times for 15,000 elements across various input types

Results:

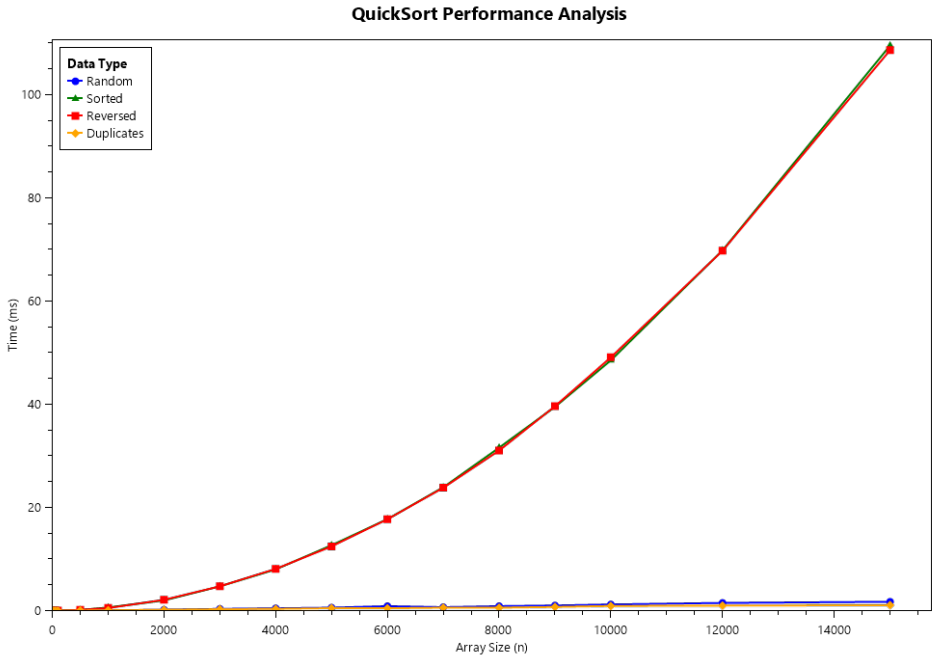


Figure 6: QuickSort Performance Graph

HeapSort:

HeapSort transforms the array into a Max-Heap structure. It is the only $O(n \log n)$ algorithm here that sorts strictly in-place with $O(1)$ extra memory.

Extended Pseudocode:

Listing 5: HeapSort Logic: Build-Heap and Extract

```
HeapSort(A):
    n = A.length
    // Phase 1: Build Max-Heap
    for i = (n/2 - 1) down to 0:
        Heapify(A, n, i)

    // Phase 2: Extract elements from heap
    for i = (n - 1) down to 1:
        swap A[0], A[i] // Move current root to end
        Heapify(A, i, 0) // Max-heapify the reduced heap

Heapify(A, heapSize, rootIndex):
    largest = rootIndex
    l = 2 * rootIndex + 1
    r = 2 * rootIndex + 2

    if l < heapSize and A[l] > A[largest]: largest = l
    if r < heapSize and A[r] > A[largest]: largest = r

    if largest != rootIndex:
        swap A[rootIndex], A[largest]
        Heapify(A, heapSize, largest)
```

Implementation:

```
public static class HeapSort
{
    public static void Sort(int[] arr)
    {
        int n = arr.Length;

        //Build the max heap
        for (int i = n/2 -1; i >= 0; i--)
        {
            Heapify(arr, n, i);
        }

        //Extract elements from heap one by one
        for (int i= n -1; i > 0; i--)
        {
            //Move current root to end
            Swap(arr, 0, i);

            //Heapify the reduced heap
            Heapify(arr, i, 0);
        }

    }
    private static void Heapify(int[] arr, int heapSize, int rootIndex)
    {
        int largest = rootIndex;
        int left = 2 * rootIndex + 1;
        int right = 2 * rootIndex + 2;

        //If left child is larger than root
        if(left < heapSize && arr[left] > arr[largest])
        {
            largest = left;
        }

        // If right child is larger than largest so far
        if (right < heapSize && arr[right] > arr[largest])
        {
            largest = right;
        }

        //If largest is not root
        if (largest != rootIndex)
        {
            Swap(arr, rootIndex, largest);
            //Recursively heapify the affected sub-tree
            Heapify(arr, heapSize, largest);
        }

    }
```

Figure 7: HeapSort implementation in C#

Complexity Scenarios:

- **Best/Average/Worst Case:** $O(n \log n)$. Like MergeSort, the structure of the binary heap ensures that the "height" of the work is always logarithmic, making it immune to the $O(n^2)$ pitfalls of QuickSort.

Extreme Cases:

- **Doubles/Negatives:** Fully compatible.

- **Memory-Constrained Systems:** This is the best choice for extreme cases where memory is limited, as it doesn't use the recursion stack or auxiliary arrays of other sorts.

```
--- HEAPSORT ---
Random Array (15000):
  Average: 3.825 ms  [ 4.234, 3.635, 3.604 ]
  Sorted: ✓ Yes
Sorted Array (15000):
  Average: 3.122 ms  [ 3.339, 3.006, 3.021 ]
  Sorted: ✓ Yes
Reversed Array (15000):
  Average: 3.027 ms  [ 3.259, 2.911, 2.912 ]
  Sorted: ✓ Yes
Duplicates Array (15000):
  Average: 3.356 ms  [ 3.693, 3.142, 3.232 ]
  Sorted: ✓ Yes
```

Figure 8: Execution times for 15,000 elements across various input types

Results:

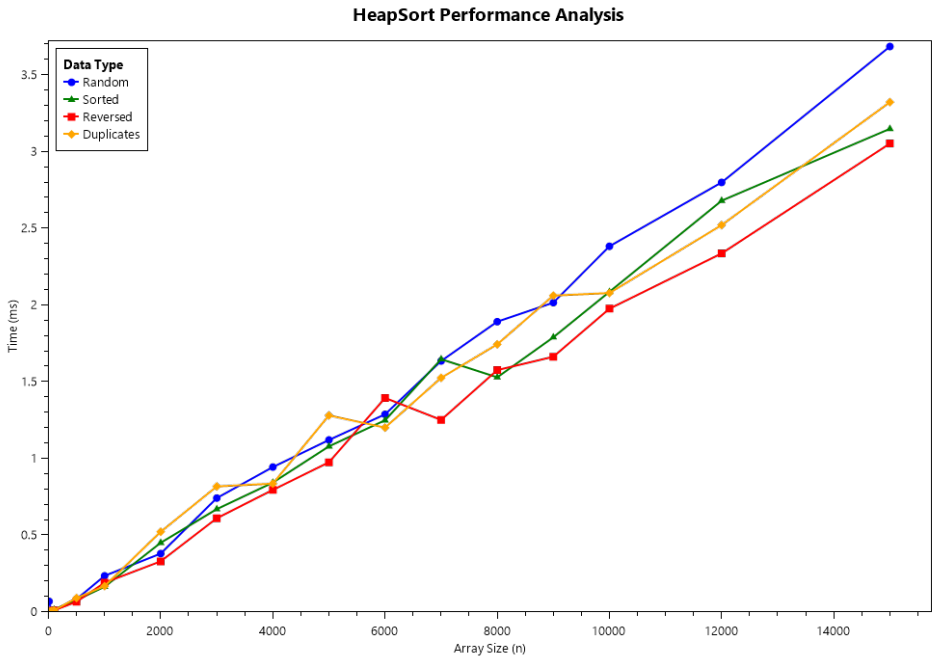


Figure 9: HeapSort Performance Graph

PatienceSort:

PatienceSort organizes elements into piles and merges them.

Extended Pseudocode:

Listing 6: Optimized PatienceSort with Priority Queue (pseudocode)

```
PatienceSort(A):
    piles = []
    pileTops = [] // Cache top cards for faster binary search

    // Phase 1: Distribute elements into piles
    for x in A:
        idx = BinarySearch(pileTops, x)
        if idx == -1:
            create new pile in piles with x
            append x to pileTops
        else:
            append x to piles[idx]
            pileTops[idx] = x

    // Phase 2: K-way merge using Min-Heap (Priority Queue)
    PQ = new PriorityQueue()
    for i = 0 to piles.length - 1:
        PQ.enqueue(pileIndex: i, priority: pileTops[i])

    for i = 0 to A.length - 1:
        // Extract minimum top card across all piles
        winningIdx = PQ.dequeue()
        A[i] = piles[winningIdx].pop()

        // If the pile is not empty, add its new top card to PQ
        if piles[winningIdx] is not empty:
            newTop = piles[winningIdx].top()
            PQ.enqueue(pileIndex: winningIdx, priority: newTop)
```

Implementation:

```
{
    public static void Sort(int[] arr)
    {
        if (arr == null || arr.Length < 2)
            return;

        // List of piles, where each pile is a list of integers (treated as a stack)
        var piles = new List<List<int>>();

        // Optimization for Phase 1: Maintain a list of top cards for faster cache-friendly binary search
        // This avoids resolving multiple references (List->List->int) during the search
        var pileTops = new List<int>();

        // Phase 1: Distribute elements into piles
        foreach (int x in arr)
        {
            // Find the leftmost pile where the top element >= x
            int pileIndex = BinarySearchPiles(pileTops, x);

            if (pileIndex == -1)
            {
                // Create new pile
                piles.Add(new List<int> { x });
                pileTops.Add(x);
            }
            else
            {
                // Add to existing pile
                piles[pileIndex].Add(x);
                pileTops[pileIndex] = x;
            }
        }

        // Phase 2: Merge piles efficiently
        // Use a PriorityQueue to efficiently find the minimum top card among all piles.
        // PriorityQueue stores the pile index, priority is the value of the top card.
        var pq = new PriorityQueue<int, int>();

        for (int i = 0; i < piles.Count; i++)
        {
            // Enqueue the index of the pile, with the value of its top card as priority
            // The top card is at the end of the List (Stack behavior)
            pq.Enqueue(i, piles[i][piles[i].Count - 1]);
        }

        for (int i = 0; i < arr.Length; i++)
        {
            // 1. Get the pile index with the smallest top card
            int winningPileIndex = pq.Dequeue();
            var winningPile = piles[winningPileIndex];

            // 2. Pop the card (smallest globally) and place it in the array
            int val = winningPile[winningPile.Count - 1];
            arr[i] = val;
            winningPile.RemoveAt(winningPile.Count - 1);

            // 3. If the pile still has cards, check the new top card and add back to PQ
            if (winningPile.Count > 0)
            {
                int newTop = winningPile[winningPile.Count - 1];
                pq.Enqueue(winningPileIndex, newTop);
            }
        }
    }
```

Figure 10: PatienceSort implementation in C#

```
// Binary search on the 'pileTops' list
private static int BinarySearchPiles(List<int> tops, int value)
{
    int left = 0;
    int right = tops.Count - 1;
    int bestPile = -1;

    while (left <= right)
    {
        int mid = left + (right - left) / 2;

        if (tops[mid] >= value)
        {
            bestPile = mid;
            right = mid - 1; // Try to find a pile further to the left
        }
        else
        {
            left = mid + 1;
        }
    }

    return bestPile;
}
```

Figure 11: PatienceSort implementation in C# (The BinarySearchPiles function)

Complexity Scenarios:

- **Best Case:** $O(n)$ - Occurs on **Reverse-Sorted** data. All elements fit into one pile, and the merge phase is trivial.
- **Worst Case:** $O(n^2)$ - Occurs on **Sorted** data. Every element creates its own pile, and the final merge phase becomes extremely expensive.

Extreme Cases:

- **Negatives:** Handled correctly.
- **Doubles:** Requires careful comparison implementation to handle pile selection without precision drift.

Results:

```
--- PATIENCESORT ---
Random Array (15000):
  Average: 2.517 ms  [ 3.260, 2.146, 2.144 ]
  Sorted: ✓ Yes
Sorted Array (15000):
  Average: 2.550 ms  [ 2.852, 2.401, 2.398 ]
  Sorted: ✓ Yes
Reversed Array (15000):
  Average: 0.840 ms  [ 0.969, 0.920, 0.632 ]
  Sorted: ✓ Yes
Duplicates Array (15000):
  Average: 1.446 ms  [ 1.755, 1.297, 1.286 ]
  Sorted: ✓ Yes
```

Figure 12: Execution times for 15,000 elements across various input types

Results:

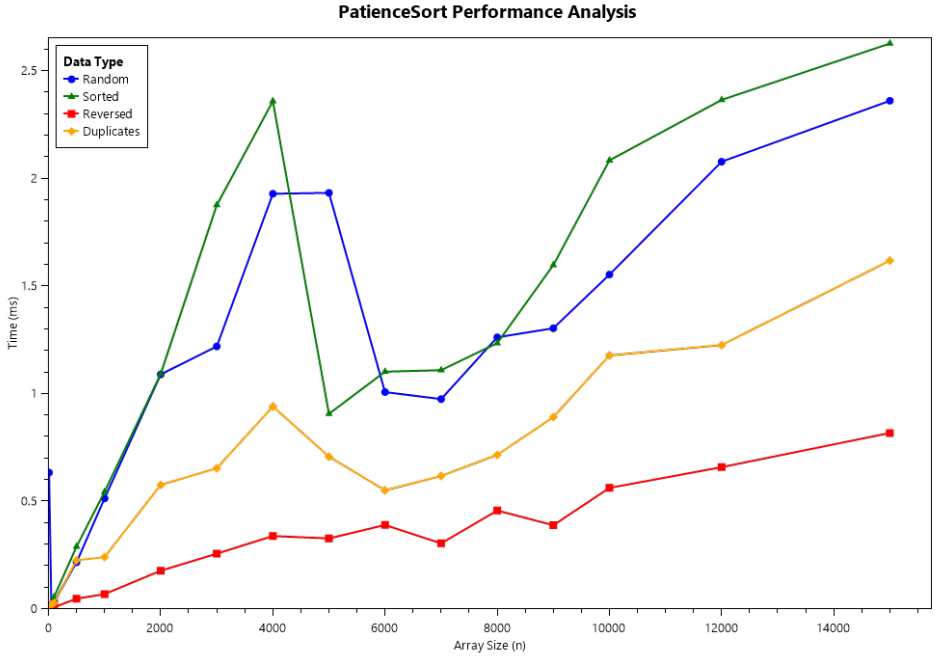
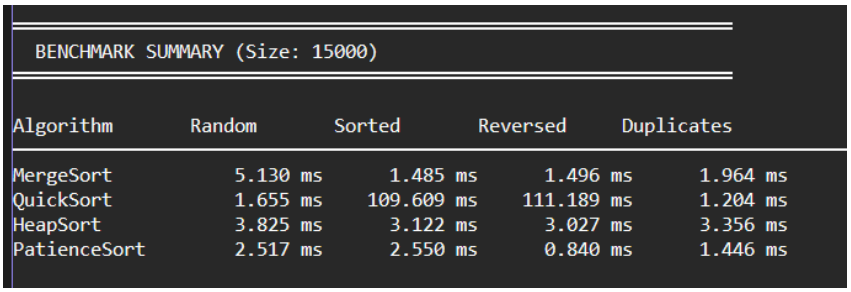


Figure 13: PatienceSort Performance Graph



BENCHMARK SUMMARY (Size: 15000)

Algorithm	Random	Sorted	Reversed	Duplicates
MergeSort	5.130 ms	1.485 ms	1.496 ms	1.964 ms
QuickSort	1.655 ms	109.609 ms	111.189 ms	1.204 ms
HeapSort	3.825 ms	3.122 ms	3.027 ms	3.356 ms
PatienceSort	2.517 ms	2.550 ms	0.840 ms	1.446 ms

Figure 14: Execution times for 15,000 for all sorts compared

CONCLUSION

This empirical analysis demonstrates that implementation-specific optimizations can significantly alter the practical performance hierarchy of sorting algorithms, sometimes allowing non-standard algorithms to rival traditional O(nlogn) benchmarks.

MergeSort and **HeapSort** remained the most consistent performers across all data distributions. However, our benchmarks revealed that while MergeSort is stable, the overhead of auxiliary array creation in C makes it slightly slower than HeapSort for most distributions, while HeapSort remains the superior choice for memory-constrained environments due to its in-place nature.

The analysis of **QuickSort** provided a critical lesson in worst-case performance. By utilizing the Hoare Partitioning scheme with a fixed first-element pivot, the algorithm achieved exceptional speeds on Random and Duplicate data (reaching 1.2 ms for 15,000 elements). However, this specific implementation choice triggered the O(n2) worst-case for Sorted and Reversed arrays, causing execution times to skyrocket to over 100 ms—nearly 80 times slower than the average case.

The most significant finding involved the optimized **PatienceSort**. By integrating a binary search for pile placement and a Min-Heap (Priority Queue) for the k-way merging phase, the algorithm's performance was drastically improved. In our benchmarks for 15,000 elements, PatienceSort achieved the fastest overall time of 0.840 ms on Reversed data and maintained a highly competitive 2.517 ms on Random data. This proves that with a O(nlogp) merge strategy (where p is the number of piles), PatienceSort becomes a formidable general-purpose sorter, especially when the input data contains existing decreasing sub-sequences.

In conclusion, while theoretical Big O notation provides a baseline, real-world performance is heavily influenced by constant factors, cache-friendly data structures, and the "sortedness" of the input. For modern applications, the optimized PatienceSort or a randomized QuickSort offer the best performance, provided the memory overhead of pile management is acceptable.

REFERENCES

[1] Moisei, D. GitHub Repository. https://github.com/Moisei-D/Algorithm-Analysis