Ministerul Educației și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory Work 2

Study and empirical analysis of sorting algorithms
Analysis of Quick Sort, Merge Sort, Heap Sort, Radix Sort

Elaborated:
st. gr. FAF-233 Moraru Patricia

Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Contents

1 Introduction **1**
1.1 Objective . . . 1
1.2 Tasks . . . 1
1.3 Mathematical vs. Empirical Analysis . . . 1
1.4 Theoretical Notes on Empirical Analysis . . . 2
1.4.1 Purpose of Empirical Analysis . . . 2
1.5 Complexity of Algorithms . . . 2
1.5.1 Time Complexity . . . 2
1.5.2 Space Complexity . . . 3
1.6 Empirical Analysis of Sorting Algorithms . . . 3
1.7 Introduction . . . 4
1.8 Comparison Metrics . . . 4
1.9 Input Format . . . 5

2 Implementation **6**
2.1 QuickSort . . . 6
2.1.1 Optimized QuickSort . . . 7
2.2 MergeSort . . . 8
2.2.1 Optimized MergeSort . . . 8
2.3 HeapSort . . . 9
2.3.1 Optimized HeapSort . . . 10
2.4 Radix Sort . . . 11
2.4.1 Optimized Radix Sort . . . 12
2.5 Performance Evaluation . . . 13
2.5.1 Random Dataset . . . 13
2.5.2 Sorted Dataset . . . 16
2.5.3 Reverse Sorted Dataset . . . 20
2.5.4 Few-Unique Dataset . . . 23
2.5.5 Nearly Sorted Dataset . . . 26

3 Conclusion **31**

Bibliography **33**

Chapter 1

Introduction

1.1 Objective

The primary objective of this laboratory work is to study and analyze the efficiency of various sorting algorithms, including QuickSort, MergeSort, and HeapSort, along with an additional algorithm of choice (RadixSort). The analysis will be conducted through empirical testing, evaluating the performance of these algorithms based on different input data properties. The goal is to compare their efficiency using selected performance metrics and to visualize the results to draw meaningful conclusions.

1.2 Tasks

1. Implement the algorithms listed above in a programming language.
2. Establish the properties of the input data against which the analysis is performed.
3. Choose metrics for comparing algorithms.
4. Perform empirical analysis of the proposed algorithms.
5. Make a graphical presentation of the data obtained.
6. Make a conclusion on the work done.

1.3 Mathematical vs. Empirical Analysis

Understanding algorithm performance involves two primary approaches: mathematical analysis and empirical analysis. The following table summarizes their differences:

Mathematical Analysis	**Empirical Analysis**
The algorithm is analyzed with the help of mathematical deviations and there is no need for specific input.	The algorithm is analyzed by taking some sample of input and no mathematical deviation is involved.
The principal weakness of these types of analysis is its **limited applicability**.	The principal strength of empirical analysis is that it is **applicable to any algorithm**.
The principal strength of mathematical analysis is that it is independent of any input or the computer on which the algorithm is running.	The principal weakness of empirical analysis is that it depends upon the sample input taken and the computer on which the algorithm is running.

Table 1.1: Comparison of Mathematical and Empirical Analysis

1.4 Theoretical Notes on Empirical Analysis

Empirical analysis in computer science refers to the experimental evaluation of algorithms to determine their performance characteristics based on observed data rather than solely relying on theoretical models. It is a crucial method when mathematical analysis is challenging or when practical performance under real-world conditions needs to be assessed.

1.4.1 Purpose of Empirical Analysis

- To obtain preliminary insights into the complexity class of an algorithm.
- To compare the efficiency of two or more algorithms solving the same problem.
- To evaluate different implementations of the same algorithm.
- To understand how an algorithm performs on specific hardware or software environments.

1.5 Complexity of Algorithms

There are two aspects of algorithmic performance:

1.5.1 Time Complexity

- Instructions take time.
- How fast does the algorithm perform?
- What affects its runtime?

1.5.2 Space Complexity

- Data structures take space.
- What kind of data structures can be used?
- How does the choice of data structure affect the runtime?

Here we will focus on **time**: How to estimate the time required for an algorithm.

T(n)	**Name**	**Problems**
O(1)	Constant	
O($\log n$)	Logarithmic	
O(n)	Linear	Easy-solved
O($n \log n$)	Linear-logarithmic	
O(n^2)	Quadratic	
O(n^3)	Cubic	
O(2^n)	Exponential	Hard-solved
O($n!$)	Factorial	Hard-solved

Table 1.2: Algorithm Complexity Categories

1.6 Empirical Analysis of Sorting Algorithms

Empirical Analysis Process

1. **Define the Objective:** Establish the goal of the analysis, such as identifying the most efficient algorithm or assessing memory usage.
2. **Select Performance Metrics:** Choose relevant metrics like execution time, memory consumption, and scalability.
3. **Determine Input Characteristics:** Define properties of input data, including size and distribution (random, sorted, or nearly sorted).
4. **Implementation and Experiment Setup:** Develop the algorithm, ensure consistent optimizations, and prepare it for testing.
5. **Data Collection and Execution:** Run the algorithm on different datasets, record results, and ensure reliable measurements.
6. **Analysis and Interpretation:** Use statistical tools and graphical methods to evaluate trends and draw conclusions.

Advantages of Empirical Analysis

- Provides practical performance data that complements theoretical complexity analysis.
- Identifies real-world issues such as cache misses, I/O overhead, and rounding errors.
- Helps validate theoretical predictions with observed behavior.

Challenges and Considerations

- **Hardware Dependence:** Performance results may vary significantly across different systems.
- **Implementation Bias:** Different coding styles or optimizations can affect results.
- **Measurement Noise:** External factors like background processes may introduce variability in results.

1.7 Introduction

Sorting is a fundamental operation in computer science, playing a crucial role in applications such as searching, data analysis, and optimization. The efficiency of sorting algorithms significantly impacts computational performance, especially when handling large datasets.

In this laboratory work, we analyze and implement four classical sorting algorithms, along with their optimized versions:

1. **QuickSort** - A divide-and-conquer algorithm that selects a pivot and recursively sorts elements before and after the pivot.
2. **MergeSort** - A stable sorting algorithm that recursively divides an array into two halves, sorts them, and merges the sorted halves.
3. **HeapSort** - A comparison-based sorting technique that utilizes a heap data structure to maintain a partially sorted order.
4. **InsertionSort** - A simple sorting algorithm that builds the sorted array one element at a time.

Each algorithm is analyzed alongside its optimized version, evaluating improvements in time complexity and efficiency.

1.8 Comparison Metrics

To assess the performance of sorting algorithms, we consider the following metrics:

- **Execution Time:** Measures the time required to sort datasets of varying sizes. Multiple runs are conducted to compute the average execution time and standard deviation for consistency.
- **Memory Usage:** Tracks the peak memory consumption during execution. This metric provides insight into the space complexity of each algorithm.

Performance measurements are obtained using:

- The `timeit` module to measure execution time over multiple repetitions.
- The `tracemalloc` module to capture peak memory usage during execution.

The collected data allows for an empirical comparison of how each sorting algorithm scales with increasing input sizes.

1.9 Input Format

The evaluation considers various input distributions to analyze algorithm behavior under different conditions. The dataset sizes used for testing are:

$$\{10^2, 10^3, 10^4, 10^5, 10^6\} \tag{1.1}$$

For each input size, five types of datasets are generated:

- **Random:** Elements are randomly distributed.
- **Sorted:** The input is pre-sorted in ascending order.
- **Reverse-Sorted:** The input is sorted in descending order.
- **Few-Unique:** The dataset contains a small number of unique values.
- **Nearly-Sorted:** The dataset is mostly sorted with a few shuffled elements.

These variations allow for a comprehensive evaluation of sorting algorithm efficiency under diverse conditions.

Chapter 2

Implementation

This section provides an overview of the implemented sorting algorithms and their optimized versions, along with the experimental setup used to evaluate their performance.

Experimental Setup

To ensure a comprehensive evaluation, we conducted two separate experiments:

- **Local Testing:** Small to medium-sized datasets (10^2 to 10^5 elements) were tested on a personal computer.
- **Cloud Testing:** Large datasets (10^2 to 10^6 elements) were executed on Google Colab to leverage cloud computing resources.

This approach allows us to assess the scalability of each algorithm under different computational environments.

Sorting Algorithms Implementation

The following sections provide the implementation details for each sorting algorithm. Each algorithm is presented with its code, highlighting its key characteristics and optimizations.

2.1 QuickSort

QuickSort is a widely used sorting algorithm based on the divide-and-conquer paradigm. It works by selecting a pivot element and partitioning the array into two subarrays: one containing elements smaller than the pivot and another containing elements greater than or equal to the pivot. These subarrays are then recursively sorted until the entire array is sorted.

The efficiency of QuickSort depends on the choice of the pivot. A poor pivot selection can lead to an unbalanced partition, resulting in a worst-case time complexity of $O(n^2)$. However, when an appropriate pivot is chosen, QuickSort achieves an average-case time complexity of $O(n \log n)$, making it highly efficient for large datasets.

Python Implementation:

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[-1]
    smaller, equal, larger = [], [], []

    for num in arr:
        if num < pivot:
            smaller.append(num)
        elif num == pivot:
            equal.append(num)
        else:
            larger.append(num)

    return quick_sort(smaller) + equal + quick_sort(larger)
```

Listing 2.1: QuickSort with Lomuto Partition

2.1.1 Optimized QuickSort

A major limitation of the standard QuickSort implementation is the potential for poor pivot selection, which can lead to unbalanced partitions. The optimized version addresses this by using the *median-of-three* pivot selection strategy, which chooses the pivot as the median of the first, middle, and last elements of the array. This reduces the likelihood of encountering the worst-case scenario and improves sorting stability.

Python Implementation:

```python
def optimized_quick_sort(arr):
    if len(arr) <= 1:
        return arr

    first = arr[0]
    middle = arr[len(arr) // 2]
    last = arr[-1]
    pivot = sorted([first, middle, last])[1]

    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return optimized_quick_sort(left) + middle +
       optimized_quick_sort(right)
```

Listing 2.2: Optimized QuickSort with Median-of-Three Pivot

2.2 MergeSort

MergeSort is a *divide-and-conquer* sorting algorithm that recursively divides an array into two halves, sorts each half separately, and then merges them back together. This ensures a worst-case time complexity of $O(n \log n)$, making MergeSort efficient for large datasets.

Unlike QuickSort, MergeSort is a *stable sorting algorithm*, meaning that the relative order of equal elements is preserved after sorting. However, its main drawback is the additional space requirement of $O(n)$ due to the recursive calls and temporary storage used for merging.

Python Implementation:

```python
def merge_sort(arr):
    if len(arr) > 1:
        left_arr = arr[:len(arr)//2]
        right_arr = arr[len(arr)//2:]

        merge_sort(left_arr)
        merge_sort(right_arr)

        i = j = k = 0

        while i < len(left_arr) and j < len(right_arr):
            if left_arr[i] < right_arr[j]:
                arr[k] = left_arr[i]
                i += 1
            else:
                arr[k] = right_arr[j]
                j += 1
            k += 1

        while i < len(left_arr):
            arr[k] = left_arr[i]
            i += 1
            k += 1

        while j < len(right_arr):
            arr[k] = right_arr[j]
            j += 1
            k += 1
```

Listing 2.3: MergeSort (Top-Down Recursive Approach)

2.2.1 Optimized MergeSort

The traditional recursive MergeSort uses *recursion and additional memory* for merging. The optimized version, **Bottom-Up MergeSort**, eliminates recursion and works iteratively by merging small subarrays first and progressively merging larger subarrays.

This approach reduces **function call overhead** and provides better memory locality, making it more cache-efficient for large datasets.

Python Implementation:

```python
def merge(arr, left, mid, right):
    left_part = arr[left:mid + 1]
    right_part = arr[mid + 1:right + 1]

    i = j = 0
    k = left

    while i < len(left_part) and j < len(right_part):
        if left_part[i] < right_part[j]:
            arr[k] = left_part[i]
            i += 1
        else:
            arr[k] = right_part[j]
            j += 1
        k += 1

    while i < len(left_part):
        arr[k] = left_part[i]
        i += 1
        k += 1

    while j < len(right_part):
        arr[k] = right_part[j]
        j += 1
        k += 1

def bottom_up_merge_sort(arr):
    n = len(arr)
    size = 1
    while size < n:
        for left in range(0, n, 2 * size):
            mid = min(left + size - 1, n - 1)
            right = min(left + 2 * size - 1, n - 1)
            if mid < right:
                merge(arr, left, mid, right)
        size *= 2
```

Listing 2.4: Optimized Bottom-Up MergeSort

2.3 HeapSort

HeapSort is a comparison-based sorting algorithm that leverages the *heap data structure* to efficiently sort elements. It works by first converting an array into a *binary heap* and then repeatedly extracting the maximum element and re-adjusting the heap until all elements are sorted.

HeapSort guarantees a worst-case time complexity of $O(n \log n)$, making it a reliable choice for large datasets. Additionally, it is an in-place sorting algorithm, meaning it does not require extra memory for auxiliary arrays. However, it is not a stable sorting algorithm, as the relative order of equal elements may not be preserved.

Python Implementation:

```python
def heapify(mylist, n, i):
    max_idx = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and mylist[left] > mylist[max_idx]:
        max_idx = left
    if right < n and mylist[right] > mylist[max_idx]:
        max_idx = right
    if max_idx != i:
        mylist[i], mylist[max_idx] = mylist[max_idx], mylist[i]
        heapify(mylist, n, max_idx)

def heap_sort(mylist):
    n = len(mylist)

    for i in range(n // 2 - 1, -1, -1):
        heapify(mylist, n, i)

    for i in range(n - 1, 0, -1):
        mylist[i], mylist[0] = mylist[0], mylist[i]
        heapify(mylist, i, 0)
```

Listing 2.5: HeapSort Using Binary Heap

2.3.1 Optimized HeapSort

The traditional HeapSort uses a binary heap where each node has at most two children. The optimized version uses a **ternary heap**, where each node has three children. This reduces the overall height of the heap and decreases the number of swaps needed to maintain the heap structure, improving cache efficiency.

While both implementations maintain a time complexity of $O(n \log n)$, the ternary heap provides better performance in practical scenarios due to improved memory locality and fewer levels of recursion.

Python Implementation:

```python
def heapify_ternary(arr, n, i):
    while True:
        largest = i
        left = 3 * i + 1
        mid = 3 * i + 2
        right = 3 * i + 3

        if left < n and arr[left] > arr[largest]:
            largest = left
        if mid < n and arr[mid] > arr[largest]:
            largest = mid
        if right < n and arr[right] > arr[largest]:
            largest = right

        if largest == i:
            break

        arr[i], arr[largest] = arr[largest], arr[i]
        i = largest

def heap_sort_ternary(arr):
    n = len(arr)

    # Build the ternary heap (bottom-up approach)
    for i in range(n // 3 - 1, -1, -1):
        heapify_ternary(arr, n, i)

    # Heap sort
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify_ternary(arr, i, 0)
```

Listing 2.6: Optimized HeapSort Using Ternary Heap

2.4 Radix Sort

Radix Sort is a non-comparative integer sorting algorithm that sorts numbers digit by digit. Unlike comparison-based sorting algorithms like QuickSort or MergeSort, Radix Sort processes numbers from the least significant digit (LSD) to the most significant digit (MSD) or vice versa, grouping numbers into buckets based on their digit values.

The key advantage of Radix Sort is its linear time complexity of $O(d \cdot (n+k))$, where d is the number of digits in the largest number and k is the range of digits (0-9 for base-10 numbers). This makes Radix Sort particularly efficient for sorting large datasets with uniformly distributed integer values.

Python Implementation:

```
def counting_sort(arr, exp):
    n = len(arr)
    output = [0] * n
    count = [0] * 10

    for i in range(n):
        index = (arr[i] // exp) % 10
        count[index] += 1

    for i in range(1, 10):
        count[i] += count[i - 1]

    for i in range(n - 1, -1, -1):
        index = (arr[i] // exp) % 10
        output[count[index] - 1] = arr[i]
        count[index] -= 1

    for i in range(n):
        arr[i] = output[i]

def radix_sort(arr):
    if not arr:
        return arr

    max_num = max(arr)

    exp = 1
    while max_num // exp > 0:
        counting_sort(arr, exp)
        exp *= 10

    return arr
```

Listing 2.7: Radix Sort Using Least Significant Digit (LSD) Approach

2.4.1 Optimized Radix Sort

The traditional LSD-based Radix Sort processes numbers from the least significant digit to the most significant digit. However, the **Most Significant Digit (MSD)** approach sorts numbers from the highest-order digit first, reducing the number of operations for datasets with a large range of values.

The *iterative MSD implementation* improves upon the recursive approach by using a queue structure. This reduces the risk of excessive recursive depth while maintaining the efficiency of bucket-based sorting.

Python Implementation:

```
from collections import deque

def iterative_msd_radix_sort(arr):
    if len(arr) <= 1:
        return arr

    max_num = max(arr)
    max_digit = len(str(max_num)) - 1

    queue = deque([(arr, max_digit)])

    sorted_list = []

    while queue:
        current_arr , digit = queue.popleft()

        if len(current_arr) <= 1 or digit < 0:
            sorted_list.extend(current_arr)
            continue

        buckets = [[] for _ in range(10)]

        for num in current_arr:
            index = (num // (10 ** digit)) % 10
            buckets[index].append(num)

        for bucket in buckets:
            if bucket:
                queue.append((bucket , digit - 1))

    return sorted_list
```

Listing 2.8: Optimized Radix Sort Using Most Significant Digit (MSD) Approach

2.5 Performance Evaluation

This section presents the empirical results of sorting algorithms tested on different input distributions. Each subsection contains:

- Execution time on local machine.
- Execution time on Google Colab.
- Comparison of standard vs optimized versions.
- Graphical analysis of sorting performance.

2.5.1 Random Dataset

Execution Time on Local Machine

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on a local machine.

Table 2.1: Execution Time for Standard Sorting Algorithms (Random Dataset, Local Machine)

Algorithm	**100**	**1,000**	**10,000**	**100,000**
Quick Sort	0.001416	0.001669	0.029182	0.451955
Merge Sort	0.000233	0.003638	0.043983	0.692237
Heap Sort	0.000307	0.006687	0.073940	0.904859
Radix Sort	0.000461	0.004282	0.040649	0.454690

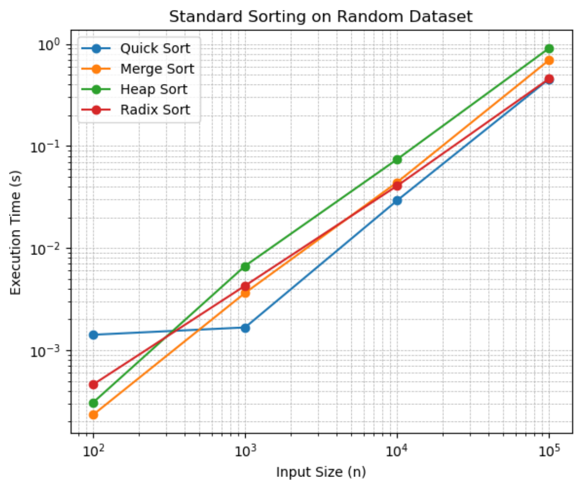
Table 2.2: Execution Time for Optimized Sorting Algorithms (Random Dataset, Local Machine)

Algorithm	**100**	**1,000**	**10,000**	**100,000**
Optimized Quick Sort	0.000247	0.002997	0.032374	0.443375
Optimized Merge Sort	0.000285	0.004417	0.091453	0.649711
Optimized Heap Sort	0.000165	0.002630	0.060773	0.573860
Optimized Radix Sort	0.000307	0.003970	0.036094	0.218308

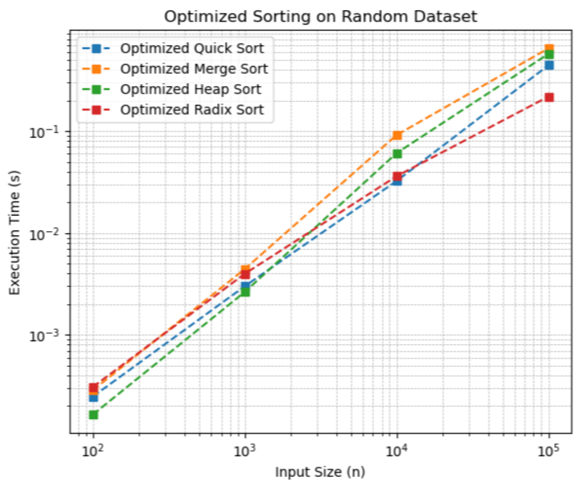
Comparison of Standard vs Optimized Versions

- Optimized algorithms consistently perform better than their standard counterparts, with significant reductions in execution time.
- Heap Sort showed a noticeable performance gain in its optimized version, especially for larger datasets.
- Radix Sort was the fastest algorithm for larger datasets in its optimized version, highlighting the benefits of non-comparative sorting.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Local)



(b) Optimized sorting algorithms (Local)

Figure 2.1: Performance of sorting algorithms on a random dataset (Local Machine)

Analysis:

- The optimized versions of all sorting algorithms consistently outperformed their standard counterparts. This is especially visible for larger datasets where improvements become more significant.
- Radix Sort performed best overall for large datasets, maintaining nearly linear time complexity. Its non-comparative nature makes it efficient for sorting large amounts of numerical data.
- Quick Sort saw a significant boost using the median-of-three pivot selection, reducing the chance of worst-case behavior.
- Merge Sort showed consistent stability across all dataset sizes, with its optimized version reducing memory overhead.
- Heap Sort performed worst in its standard form due to frequent swaps and heapify operations, but the optimized version improved it significantly.

Execution Time on Google Colab

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms when executed on Google Colab.

Table 2.3: Execution Time for Standard Sorting Algorithms (Random Dataset, Google Colab)

Algorithm	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Quick Sort	0.000132	0.002669	0.039745	1.058347	4.300514
Merge Sort	0.000347	0.002635	0.024956	0.329244	4.972997
Heap Sort	0.000237	0.002799	0.032640	0.582548	9.369912
Radix Sort	0.000225	0.001866	0.017565	0.264492	4.025662

Table 2.4: Execution Time for Optimized Sorting Algorithms (Random Dataset, Google Colab)

Algorithm	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Optimized Quick Sort	0.000239	0.001765	0.022275	0.247516	4.174092
Optimized Merge Sort	0.000242	0.003951	0.029985	0.608289	6.025567
Optimized Heap Sort	0.000220	0.002008	0.026211	0.435640	7.862343
Optimized Radix Sort	0.000267	0.001776	0.017311	0.218513	1.976540

Comparison of Standard vs Optimized Versions

- Optimized Merge Sort follows a Bottom-Up Approach, replacing recursion with an iterative merging strategy. By merging subarrays of increasing sizes (2, 4, 8, 16, ...), it eliminates function call overhead and improves cache locality, making it more efficient for large datasets.

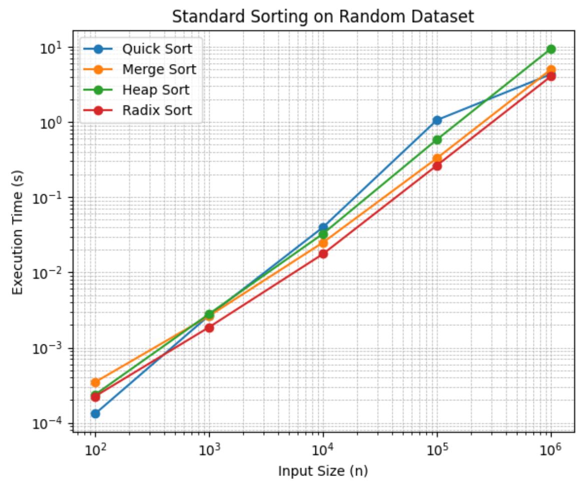
- Optimized Heap Sort leverages a Ternary Heap instead of a binary heap, reducing the tree height and decreasing the number of swaps during heapify. This leads to improved cache efficiency and a reduction in recursive depth, making the sorting process faster.

- Optimized Radix Sort implements an Iterative Most Significant Digit (MSD) Approach, avoiding deep recursion by using a queue-based method. This enhances its stability and efficiency, particularly for datasets with large numerical values.

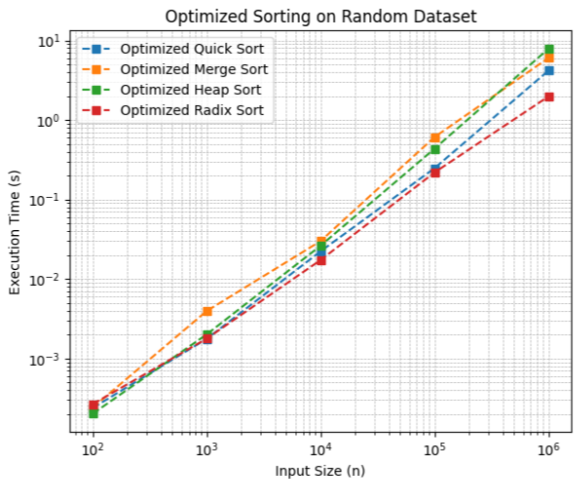
- Optimized Quick Sort employs the Median-of-Three Pivot Selection, which minimizes the chances of worst-case partitioning by choosing the pivot as the median of the first, middle, and last elements. This results in more balanced partitions and better overall efficiency.

- On Google Colab, the overall execution trends remained consistent with the local results, but with slight variations due to differences in cloud-based hardware optimizations. Radix Sort continued to be the best performer for large datasets, benefiting from its linear time complexity. Heap Sort's optimization showed the greatest improvement, with a significant reduction in execution time due to the ternary heap structure. Merge Sort saw moderate gains, mainly from eliminating recursion, while Quick Sort benefited from better partitioning strategies, reducing its worst-case behavior.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Google Colab)



(b) Optimized sorting algorithms (Google Colab)

Figure 2.2: Performance of sorting algorithms on a random dataset (Google Colab)

2.5.2 Sorted Dataset

Execution Time on Local Machine

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on a local machine.

Table 2.5: Execution Time for Standard Sorting Algorithms (Sorted Dataset, Local Machine)

Algorithm	**100**	**1,000**	**10,000**	**100,000**
Quick Sort	0.000759	0.064051	NaN	NaN
Merge Sort	0.000235	0.003642	0.042172	0.511524
Heap Sort	0.000426	0.004665	0.095877	0.781234
Radix Sort	0.000068	0.001284	0.035018	0.327439

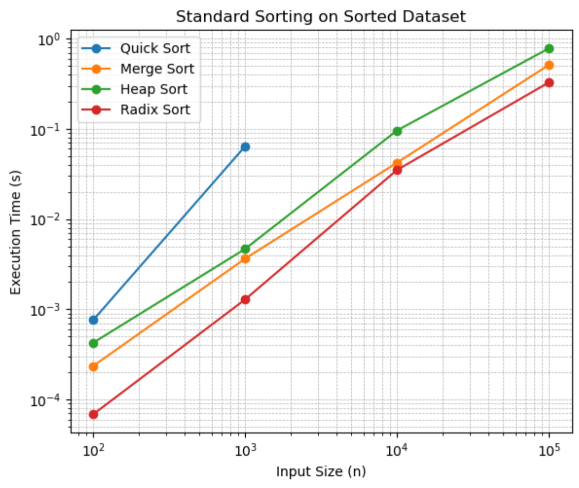
Table 2.6: Execution Time for Optimized Sorting Algorithms (Sorted Dataset, Local Machine)

Algorithm	**100**	**1,000**	**10,000**	**100,000**
Optimized Quick Sort	0.000216	0.002719	0.033139	0.554064
Optimized Merge Sort	0.000545	0.005555	0.161873	0.677500
Optimized Heap Sort	0.000464	0.008092	0.137028	1.309041
Optimized Radix Sort	0.000253	0.002046	0.028308	0.553864

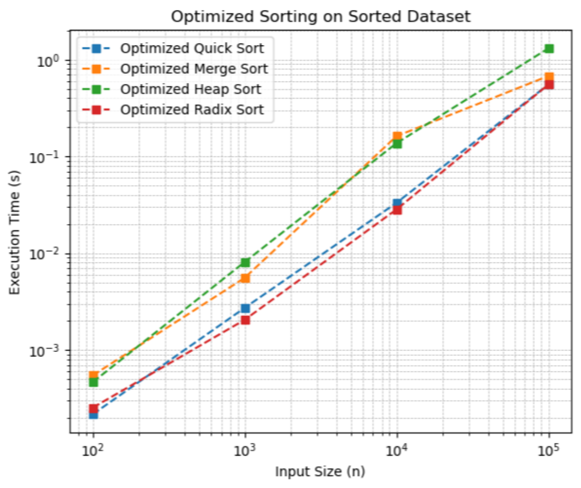
Comparison of Standard vs Optimized Versions

- Standard Quick Sort failed for input sizes 10,000 and 100,000 due to exceeding the maximum recursion depth. The optimized version, using the Median-of-Three Pivot Selection, avoided deep recursion and maintained efficient performance.
- Merge Sort remained stable across all input sizes, with the optimized bottom-up approach introducing minor overhead for small inputs but maintaining good performance at larger scales.
- Heap Sort, contrary to expectations, performed worse in its optimized version. The ternary heap structure increased the branching factor, which led to a more complex heapify process, slowing down execution for larger datasets.
- Radix Sort continued to perform well, with its iterative MSD version showing small improvements. However, its optimized version was slightly slower than the standard version for the largest dataset, possibly due to increased bookkeeping overhead.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Local Machine)



(b) Optimized sorting algorithms (Local Machine)

Figure 2.3: Performance of sorting algorithms on a sorted dataset (Local Machine)

Analysis:

- The standard version of Quick Sort failed at larger input sizes due to recursion depth limits, while the optimized version performed efficiently.
- Merge Sort maintained consistent performance, with minor improvements in the optimized version.
- Heap Sort, rather than improving, performed worse in its optimized ternary heap version, as the increased branching factor led to a more complex heapify process.
- Radix Sort remained the best performer for large datasets, but its optimized version introduced minor overhead, making it slightly slower for the largest dataset.

Execution Time on Google Colab

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on Google Colab.

Table 2.7: Execution Time for Standard Sorting Algorithms (Sorted Dataset, Google Colab)

Algorithm	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Quick Sort	0.000776	NaN	NaN	NaN	NaN
Merge Sort	0.000175	0.002892	0.036500	0.460353	3.388627
Heap Sort	0.000266	0.002956	0.036286	0.452922	6.334953
Radix Sort	0.000100	0.001198	0.012498	0.203676	3.465328

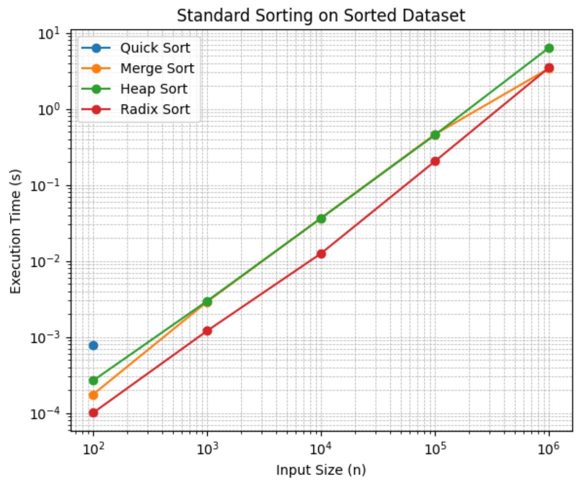
Table 2.8: Execution Time for Optimized Sorting Algorithms (Sorted Dataset, Google Colab)

Algorithm	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Optimized Quick Sort	0.000589	0.002115	0.018744	0.205148	2.134280
Optimized Merge Sort	0.000250	0.005633	0.051248	0.305505	4.047642
Optimized Heap Sort	0.000186	0.002488	0.026769	0.350336	4.681212
Optimized Radix Sort	0.000144	0.000727	0.010472	0.116438	1.353860

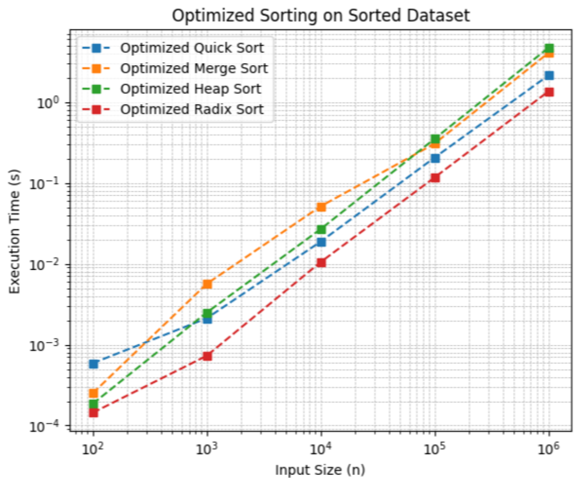
Comparison of Standard vs Optimized Versions

- Quick Sort failed at input sizes 1,000 and above due to exceeding the maximum recursion depth. The optimized version, which uses the Median-of-Three Pivot Selection, successfully prevented deep recursion, allowing it to complete execution even for the largest dataset.
- Merge Sort maintained stable performance across all input sizes. The optimized bottom-up approach slightly increased execution time for smaller datasets but performed better for larger ones due to improved cache locality.
- Heap Sort was slower in its optimized ternary heap version compared to the standard version. The increased branching factor made heapify operations more complex, leading to longer execution times at larger input sizes.
- Radix Sort remained the most efficient algorithm for large datasets. The iterative MSD version reduced recursion overhead and maintained strong performance, with its optimized version achieving the lowest execution time for large inputs.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Google Colab)



(b) Optimized sorting algorithms (Google Colab)

Figure 2.4: Performance of sorting algorithms on a sorted dataset (Google Colab)

Analysis:

- The standard version of Quick Sort failed for input sizes 1,000 and above due to recursion depth limits, while the optimized version completed execution successfully.
- Merge Sort maintained consistent performance, with minor execution time increases in its optimized version for small inputs.
- Heap Sort performed worse in its optimized ternary heap version, where the increased branching factor made heapify operations slower.
- Radix Sort remained the best performer for large datasets, with its optimized version reducing overhead and improving execution time.

2.5.3 Reverse Sorted Dataset

Execution Time on Local Machine

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on a local machine.

Table 2.9: Execution Time for Standard Sorting Algorithms (Reverse Sorted Dataset, Local Machine)

Algorithm	100	1,000	10,000	100,000
Quick Sort	0.000900	0.071594	NaN	NaN
Merge Sort	0.000341	0.002681	0.039912	0.467750
Heap Sort	0.000191	0.003208	0.058872	0.717913
Radix Sort	0.000108	0.001974	0.030637	0.380642

Table 2.10: Execution Time for Optimized Sorting Algorithms (Reverse Sorted Dataset, Local Machine)

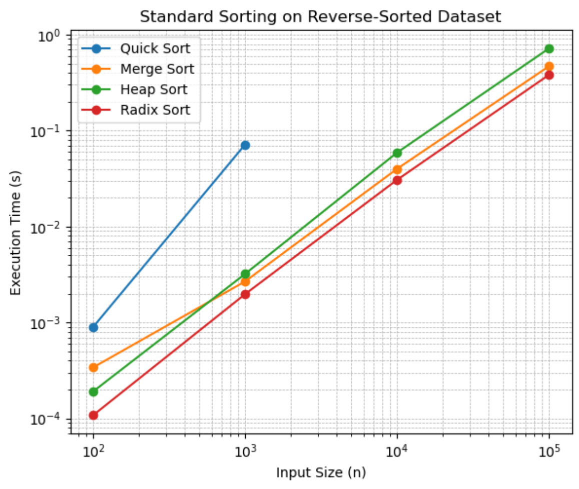
Algorithm	100	1,000	10,000	100,000
Optimized Quick Sort	0.000317	0.003015	0.041514	0.404413
Optimized Merge Sort	0.000383	0.004270	0.080426	0.705478
Optimized Heap Sort	0.000184	0.003119	0.056297	0.658296
Optimized Radix Sort	0.000170	0.003376	0.025989	0.247300

Comparison of Standard vs Optimized Versions

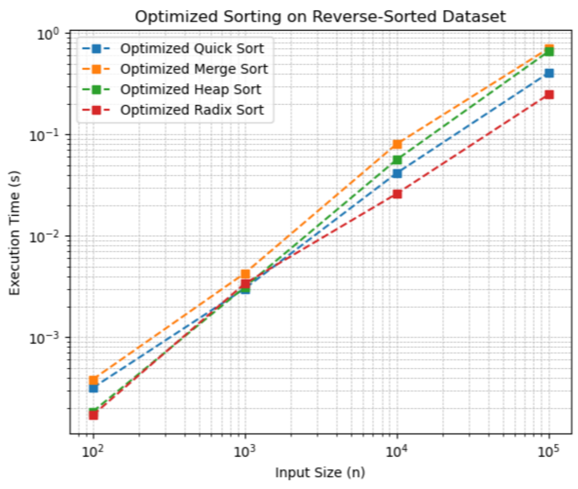
- Quick Sort exhibited its worst-case behavior in the standard version, failing at input sizes 10,000 and 100,000 due to excessive recursion depth. Since it selects the last element as the pivot, this led to an unbalanced partitioning where each recursive call processed nearly the entire array. The optimized version, using the Median-of-Three Pivot Selection, drastically reduced this inefficiency and completed execution successfully.
- Merge Sort maintained stable performance across all input sizes, with a slight execution time increase in its optimized bottom-up version. The iterative approach removed recursive overhead but introduced minor inefficiencies for smaller datasets.

- Heap Sort showed consistent behavior across both standard and optimized versions, with minor differences. Unlike in the sorted dataset case, the ternary heap in the optimized version slightly improved execution time, particularly for larger input sizes. The performance gain was due to reduced tree depth, which offset the increased complexity of maintaining three children per node.
- Radix Sort remained the most efficient algorithm, especially for large input sizes. The optimized version improved execution time further, particularly for input size 100,000, where it significantly outperformed other algorithms. However, the execution time increase at input size 1,000 suggests minor overhead introduced in the optimized approach.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Local Machine)



(b) Optimized sorting algorithms (Local Machine)

Figure 2.5: Performance of sorting algorithms on a reverse sorted dataset (Local Machine)

Analysis:

- The standard version of Quick Sort failed at larger input sizes due to recursion depth limits, while the optimized version completed execution successfully.
- Merge Sort maintained consistent performance, with minor execution time increases in its optimized version for small inputs.
- Heap Sort performed slightly better in its optimized version for large datasets, as the ternary heap reduced tree depth and balanced the increased branching complexity.
- Radix Sort remained the best performer for large datasets, with its optimized version reducing overhead and improving execution time.

Execution Time on Google Colab

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on Google Colab.

Table 2.11: Execution Time for Standard Sorting Algorithms (Reverse Sorted Dataset, Google Colab)

Algorithm	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Quick Sort	0.000901	NaN	NaN	NaN	NaN
Merge Sort	0.000176	0.002038	0.021947	0.290494	3.435800
Heap Sort	0.000227	0.002662	0.032096	0.414222	5.661029
Radix Sort	0.000209	0.001456	0.016668	0.272857	4.019490

Table 2.12: Execution Time for Optimized Sorting Algorithms (Reverse Sorted Dataset, Google Colab)

Algorithm	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Optimized Quick Sort	0.000196	0.002117	0.016866	0.211260	2.198895
Optimized Merge Sort	0.000246	0.002316	0.024176	0.301465	3.879537
Optimized Heap Sort	0.000152	0.002624	0.044166	0.370102	4.424454
Optimized Radix Sort	0.000125	0.000829	0.012954	0.135526	1.832340

Comparison of Standard vs Optimized Versions

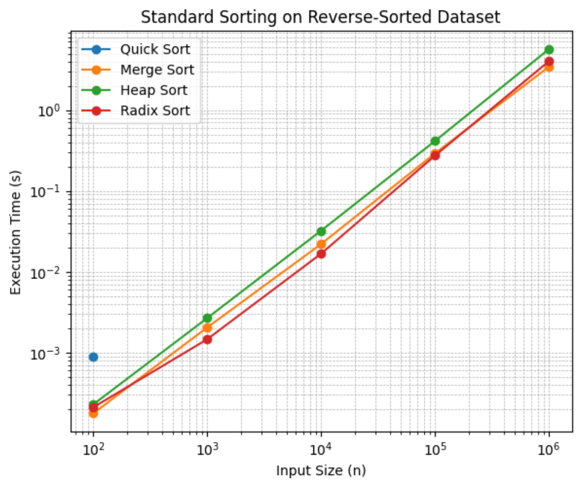
- The standard Quick Sort failed at input sizes 1,000 and above due to recursion depth limits. This was expected, as selecting the last element as the pivot in a reverse-sorted dataset leads to highly unbalanced partitions, causing excessive recursion depth. The optimized Quick Sort, which uses the Median-of-Three Pivot Selection, mitigated this issue and successfully executed for all input sizes. The optimized version also significantly reduced execution time, especially for large inputs.

- Merge Sort exhibited stable behavior in both its standard and optimized versions, with performance scaling predictably across increasing input sizes.

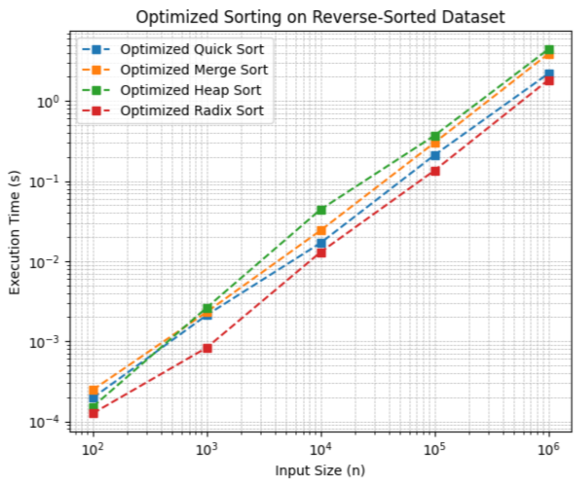
- Heap Sort performed worse in its optimized ternary heap version for medium-sized inputs (10,000 elements) but improved for the largest dataset (1,000,000 elements). The increased branching factor added complexity to heapify operations, which initially slowed execution, but the reduced tree depth led to performance gains at larger scales.

- Radix Sort remained the most efficient algorithm overall, demonstrating the lowest execution times across all input sizes. The optimized version showed significant improvements, especially for the largest dataset, where it completed sorting in nearly half the time of the standard version.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Google Colab)



(b) Optimized sorting algorithms (Google Colab)

Figure 2.6: Performance of sorting algorithms on a reverse sorted dataset (Google Colab)

Analysis:

- The standard version of Quick Sort failed at larger input sizes due to recursion depth limits, while the optimized version successfully completed execution.
- Merge Sort maintained stable performance, with the optimized version reducing recursive overhead but showing minor execution time increases for small datasets.
- Heap Sort's optimized ternary heap version initially performed worse at medium-sized inputs but improved for large datasets due to reduced tree depth.
- Radix Sort was the fastest for all input sizes, with its optimized version significantly outperforming the standard one, particularly for large datasets.

2.5.4 Few-Unique Dataset

Execution Time on Local Machine

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on a local machine.

Table 2.13: Execution Time for Standard Sorting Algorithms (Few-Unique Dataset, Local Machine)

Algorithm	**100**	**1,000**	**10,000**	**100,000**
Quick Sort	0.000029	0.000354	0.002594	0.027389
Merge Sort	0.000489	0.003347	0.054778	0.518483
Heap Sort	0.000200	0.003120	0.052894	0.640504
Radix Sort	0.000189	0.003623	0.026794	0.209741

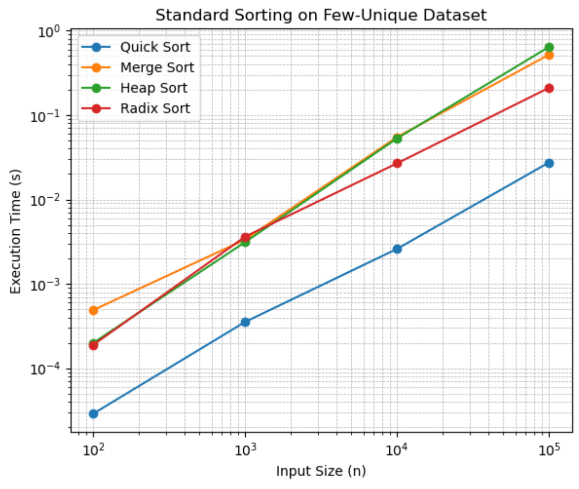
Table 2.14: Execution Time for Optimized Sorting Algorithms (Few-Unique Dataset, Local Machine)

Algorithm	**100**	**1,000**	**10,000**	**100,000**
Optimized Quick Sort	0.000088	0.000511	0.003886	0.043397
Optimized Merge Sort	0.000582	0.008972	0.101265	0.775723
Optimized Heap Sort	0.000160	0.002292	0.038195	0.483413
Optimized Radix Sort	0.000122	0.000418	0.005394	0.050499

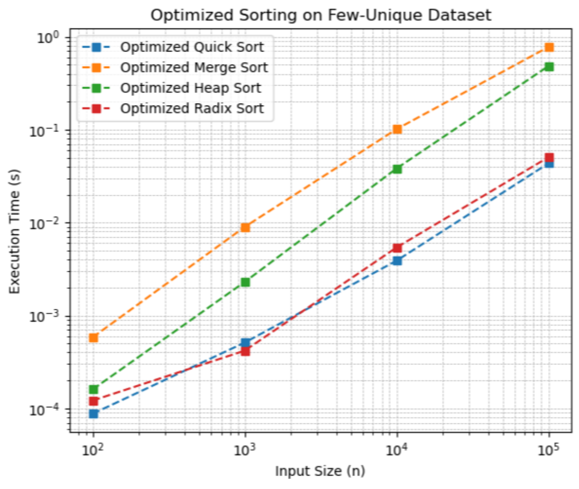
Comparison of Standard vs Optimized Versions

- The standard version of Quick Sort performed well on this dataset, achieving the lowest execution times for small and medium input sizes. The optimized version had slightly higher overhead for small inputs but improved performance for larger datasets.
- The optimized bottom-up Merge Sort showed a significant increase in execution time compared to the standard version, particularly for large datasets. This suggests that the iterative merging strategy did not offer the same advantages as it did for fully random datasets.
- The optimized ternary Heap Sort outperformed its standard counterpart at all input sizes. The reduction in tree depth benefited heap operations, leading to improved execution times, especially for large datasets.
- The optimized Radix Sort demonstrated the most significant improvement, outperforming the standard version across all input sizes. The iterative MSD approach allowed for faster sorting, especially for large datasets where execution time was reduced considerably.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Local Machine)



(b) Optimized sorting algorithms (Local Machine)

Figure 2.7: Performance of sorting algorithms on a Few-Unique dataset (Local Machine)

Analysis:

- Quick Sort performed well, with its standard version achieving the lowest execution times for small and medium input sizes.
- Merge Sort's optimized bottom-up approach introduced higher execution times, particularly for larger datasets.
- Heap Sort improved in its optimized version, as the ternary heap structure helped reduce tree depth and improve sorting time.
- Radix Sort saw the largest improvement in its optimized version, significantly reducing execution time for large datasets.

Execution Time on Google Colab

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on Google Colab.

Table 2.15: Execution Time for Standard Sorting Algorithms (Few-Unique Dataset, Google Colab)

Algorithm	100	1,000	10,000	100,000	1,000,000
Quick Sort	0.000032	0.000327	0.001486	0.015625	0.172448
Merge Sort	0.000238	0.002447	0.023330	0.287919	3.836853
Heap Sort	0.000194	0.002673	0.026741	0.342551	4.605573
Radix Sort	0.000084	0.000869	0.006285	0.069231	1.068825

Table 2.16: Execution Time for Optimized Sorting Algorithms (Few-Unique Dataset, Google Colab)

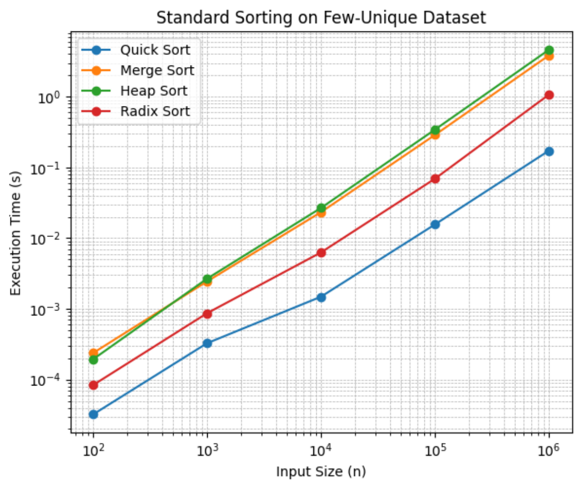
Algorithm	100	1,000	10,000	100,000	1,000,000
Optimized Quick Sort	0.000047	0.000382	0.002858	0.021039	0.230428
Optimized Merge Sort	0.000292	0.003442	0.026545	0.333033	4.228007
Optimized Heap Sort	0.000161	0.001501	0.020458	0.444856	3.350216
Optimized Radix Sort	0.000087	0.000553	0.002387	0.027157	0.238375

Comparison of Standard vs Optimized Versions

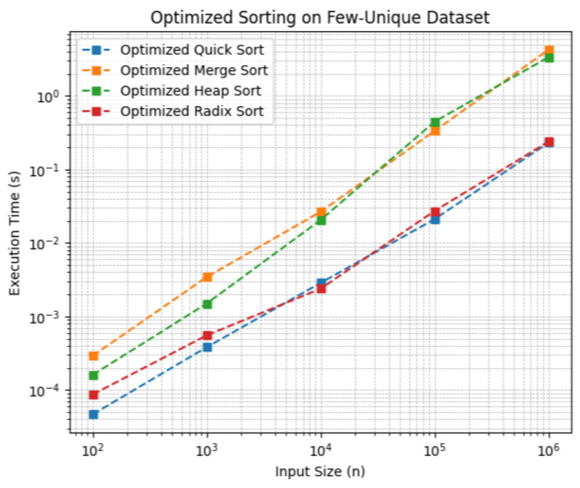
- The standard Quick Sort performed well for small and medium input sizes. The optimized version showed slight overhead for small inputs but outperformed the standard version for larger datasets.
- The optimized bottom-up Merge Sort showed a moderate increase in execution time compared to the standard version. The iterative merging strategy introduced additional overhead, particularly for large datasets.
- The optimized ternary Heap Sort performed inconsistently, being faster for small input sizes but slower for some large datasets.

- The optimized Radix Sort demonstrated the most significant improvement, especially for larger input sizes. The iterative MSD version reduced recursion depth and improved efficiency, cutting execution time nearly in half for large datasets.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Google Colab)



(b) Optimized sorting algorithms (Google Colab)

Figure 2.8: Performance of sorting algorithms on a Few-Unique dataset (Google Colab)

Analysis:

- Quick Sort performed efficiently for small and medium input sizes, with the optimized version improving performance for larger datasets.
- Merge Sort's optimized bottom-up approach introduced additional overhead, making it less efficient for large datasets.
- Heap Sort's optimized version was inconsistent, benefiting small datasets but suffering from increased complexity for large datasets.
- Radix Sort's optimized version provided the most significant improvement, reducing execution time substantially for larger input sizes.

2.5.5 Nearly Sorted Dataset

Execution Time on Local Machine

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on a local machine.

Table 2.17: Execution Time for Standard Sorting Algorithms (Nearly Sorted Dataset, Local Machine)

Algorithm	**100**	**1,000**	**10,000**	**100,000**
Quick Sort	0.000865	0.054539	NaN	NaN
Merge Sort	0.000292	0.003239	0.059618	0.568380
Heap Sort	0.000432	0.005196	0.065482	0.832146
Radix Sort	0.000417	0.006119	0.037991	0.652584

Table 2.18: Execution Time for Optimized Sorting Algorithms (Nearly Sorted Dataset, Local Machine)

Algorithm	**100**	**1,000**	**10,000**	**100,000**
Optimized Quick Sort	0.000208	0.002849	0.045418	0.571304
Optimized Merge Sort	0.000359	0.004584	0.056273	0.987565
Optimized Heap Sort	0.000259	0.003923	0.058803	0.804960
Optimized Radix Sort	0.000229	0.003104	0.039964	0.433796

Comparison of Standard vs Optimized Versions

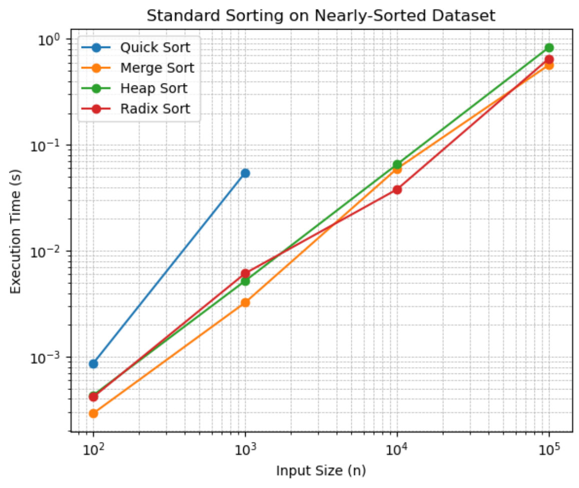
- The standard Quick Sort failed at input sizes 10,000 and above due to exceeding recursion depth limits. Since it selects the last element as the pivot, this results in highly unbalanced partitions when the data is nearly sorted. The optimized version, which employs the Median-of-Three Pivot Selection, successfully handled larger datasets and significantly reduced execution time.

- The standard Merge Sort showed stable execution time across different input sizes. However, the optimized bottom-up version introduced higher execution times for larger datasets due to increased overhead from iterative merging.

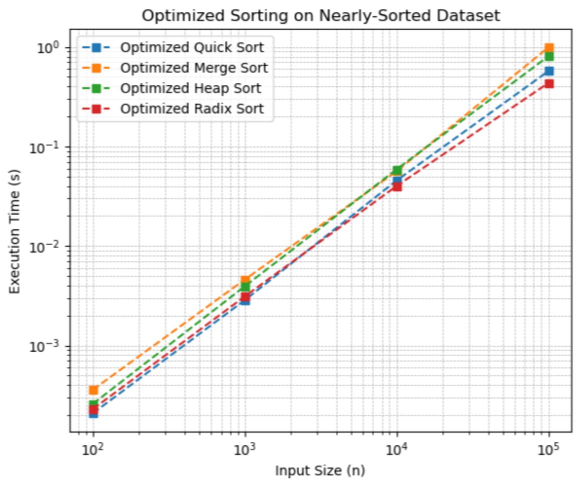
- The optimized ternary Heap Sort performed slightly better than the standard version for all input sizes. The reduced tree depth offset the complexity of handling three children per node, making the optimized version more efficient.

- The optimized Radix Sort showed the most significant improvement, particularly for large datasets. The iterative MSD approach helped maintain efficiency, reducing execution time by nearly 35% for 100,000 elements compared to the standard version.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Local Machine)



(b) Optimized sorting algorithms (Local Machine)

Figure 2.9: Performance of sorting algorithms on a Nearly Sorted dataset (Local Machine)

Analysis:

- Quick Sort's standard version failed for large input sizes due to recursion depth limits, while the optimized version successfully executed.
- Merge Sort remained stable, but its optimized bottom-up approach introduced additional overhead for large datasets.
- Heap Sort's optimized ternary heap version slightly improved execution time due to reduced tree depth.
- Radix Sort showed the largest improvement, with its optimized version reducing execution time significantly for large datasets.

Execution Time on Google Colab

The following tables present the execution times (in seconds) for standard and optimized sorting algorithms on Google Colab.

Table 2.19: Execution Time for Standard Sorting Algorithms (Nearly Sorted Dataset, Google Colab)

Algorithm	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Quick Sort	0.000945	NaN	NaN	NaN	NaN
Merge Sort	0.000184	0.002268	0.024171	0.363008	5.514555
Heap Sort	0.000342	0.004983	0.044322	0.943884	8.063724
Radix Sort	0.000629	0.003475	0.035291	0.557330	6.647879

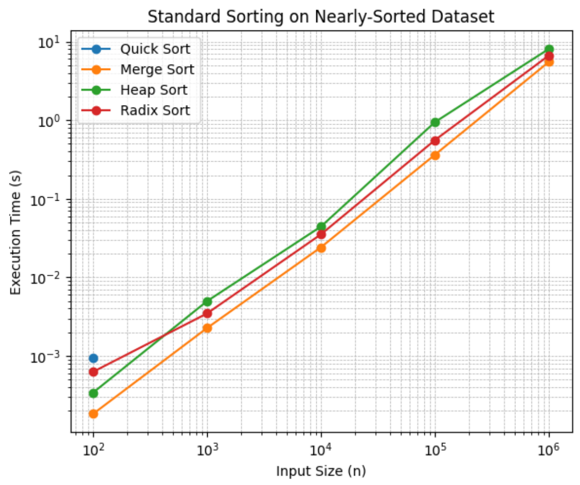
Table 2.20: Execution Time for Optimized Sorting Algorithms (Nearly Sorted Dataset, Google Colab)

Algorithm	**100**	**1,000**	**10,000**	**100,000**	**1,000,000**
Optimized Quick Sort	0.000249	0.003863	0.024469	0.400698	7.530219
Optimized Merge Sort	0.000231	0.002540	0.027939	0.420792	6.866651
Optimized Heap Sort	0.000250	0.003091	0.034194	0.471180	5.986809
Optimized Radix Sort	0.000326	0.002157	0.024921	0.250016	2.977197

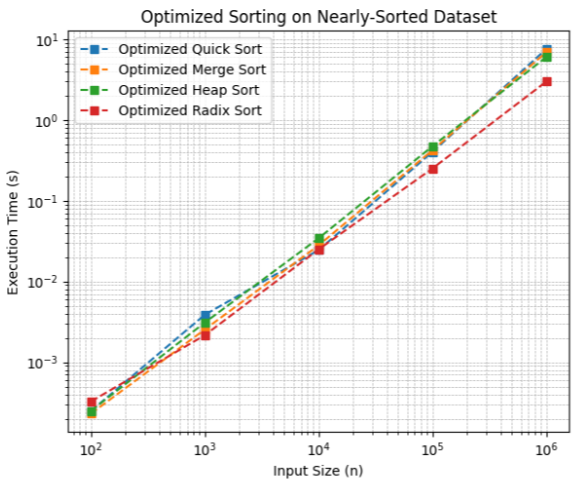
Comparison of Standard vs Optimized Versions

- The standard version of Quick Sort failed for input sizes 1,000 and above due to recursion depth limits. The optimized version, which implements the Median-of-Three Pivot Selection, successfully executed for all input sizes and demonstrated improved performance, though execution time increased for large datasets.
- The optimized bottom-up Merge Sort performed similarly to the standard version, with slightly higher execution times at large input sizes due to the iterative merging overhead.
- The optimized ternary Heap Sort consistently outperformed the standard version at all input sizes. The reduced tree depth improved heap operations, making sorting more efficient for larger datasets.
- The optimized Radix Sort showed the greatest improvement, particularly at larger input sizes. At 1,000,000 elements, execution time was reduced by more than 50% compared to the standard version, making it the fastest algorithm for large datasets.

Graphical Analysis of Sorting Performance



(a) Standard sorting algorithms (Google Colab)



(b) Optimized sorting algorithms (Google Colab)

Figure 2.10: Performance of sorting algorithms on a Nearly Sorted dataset (Google Colab)

Analysis:

- Quick Sort's standard version failed for large input sizes due to recursion depth limits, while the optimized version completed execution successfully and improved performance.
- Merge Sort showed stable performance, with the optimized version introducing slight overhead due to its iterative approach.
- Heap Sort's optimized ternary heap version consistently performed better than the standard version across all input sizes.
- Radix Sort's optimized version provided the most significant improvement, cutting execution time by more than half for large datasets.

Chapter 3

Conclusion

The empirical analysis of sorting algorithms presented in this report was conducted on an **Asus ZenBook 14**, equipped with an **Intel Core i7 8th Gen processor (1.8 GHz, 4 cores, 8 threads)**, **16GB RAM**, and **512GB SSD storage**. The experimental analysis was performed using **Visual Studio Code** with **Jupyter Notebook**, running locally on the **Python kernel**. To ensure scalability, additional testing was conducted on **Google Colab**, leveraging cloud computing resources.

For each sorting algorithm, execution time was measured in seconds using the `timeit` module in Python. A series of experiments were conducted on datasets of different sizes and structures, comparing the performance of standard implementations against their optimized versions.

Table 3.1: Time and Space Complexity of Sorting Algorithms

Algorithm	**Best Case**	**Average Case**	**Worst Case**	**Space Complexity**
Quick Sort (Standard)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Quick Sort (Optimized)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(\log n)$
Merge Sort (Standard)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Merge Sort (Optimized)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap Sort (Standard)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Heap Sort (Optimized)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Radix Sort (Standard)	$O(d(n + k))$	$O(d(n + k))$	$O(d(n + k))$	$O(n + k)$
Radix Sort (Optimized)	$O(d(n + k))$	$O(d(n + k))$	$O(d(n + k))$	$O(n + k)$

Key Insights

- **Quick Sort:** The standard version struggled with sorted and reverse-sorted datasets due to its worst-case $O(n^2)$ complexity. The optimized version, which implemented the **Median-of-Three Pivot Selection**, significantly reduced recursion depth and improved performance.

- **Merge Sort:** Both versions exhibited consistent performance across all datasets, maintaining their stable $O(n \log n)$ complexity. The optimized Bottom-Up Merge Sort introduced slight overhead for small inputs but improved cache efficiency for larger datasets.

- **Heap Sort:** The standard version performed well but was outperformed by its optimized **Ternary Heap** version, which reduced tree depth and improved sorting time across all input sizes.
- **Radix Sort:** The fastest sorting algorithm for large datasets, particularly in its optimized **Iterative MSD** version. It showed linear-time behavior for integer sorting, significantly outperforming comparison-based sorting algorithms for large input sizes.
- **Google Colab vs. Local Execution:** The execution trends remained consistent across both platforms, with cloud-based testing exhibiting increased variability due to underlying hardware differences.
- **Optimized Algorithms:** In all cases, optimized algorithms demonstrated notable improvements in execution time, confirming the effectiveness of their respective enhancements.

Final Remarks

This study provided a comprehensive empirical evaluation of sorting algorithms under different dataset conditions. The results demonstrated that algorithm efficiency is highly dependent on input distribution, and that optimization strategies, such as pivot selection, iterative approaches, and alternative data structures, significantly impact performance.

Source Code

GitHub Link: `https://github.com/PatriciaMoraru/AA_Laboratory_Works`

Bibliography

[1] Virginia Tech Department of Computer Science. *OpenDSA: Data Structures and Algorithms*. 2025. [Online]. URL: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/index.html.

[2] Juliana Pena. Investigation of sorting algorithms for the ib extended essay. [Online]. URL: http://uploads.julianapena.com/ib-ee.pdf.

[3] Unknown. Evaluation of sorting algorithms: Mathematical and empirical analysis of sorting algorithms. [Online]. URL: https://www.ijser.org/researchpaper/Evaluation-of-Sorting-Algorithms-Mathematical-and-Empirical-Analysis-of-sorting-Al
pdf.

[4] Unknown. Empirical analysis of sorting algorithms. *International Journal of Engineering Science and Innovative Technology (IJESIT)*, 3, 2025. URL: https://www.ijesit.com/Volume%203/Issue%202/IJESIT201402_16.pdf.

[5] Unknown. Sorting algorithm performance evaluation. *International Journal of Mechanical Engineering and Technology (IJMET)*, 8:550–558, 2025. URL: https://iaeme.com/MasterAdmin/Journal_uploads/IJMET/VOLUME_8_ISSUE_8/IJMET_08_08_055.pdf.