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

Elaborated:
st. gr. FAF-243 Poiata Calin

Verified:
asist. univ. Fiștic Cristofor

Chișinău – 2026

Table of Contents

ALGORITHM ANALYSIS

Objective

Study and analyze the empirical performance of sorting algorithms on randomized integer arrays.

Tasks

1. Implement QuickSort, MergeSort, HeapSort, and RadixSort in a programming language;
2. Establish properties of the input data for the analysis;
3. Choose a metric for comparing the algorithms;
4. Perform empirical analysis of the algorithms;
5. Present the results graphically;
6. Draw conclusions from the obtained data.

Theoretical Notes

Empirical analysis complements theoretical complexity by observing real execution times on concrete inputs. It is useful for comparing implementations, estimating practical constants, and understanding behavior on specific data distributions.

The standard steps are:

1. Define the analysis goal.
2. Select a metric (time or operation count).
3. Specify input properties (size, range, distribution).
4. Implement the algorithms.
5. Generate multiple test datasets.
6. Run experiments and record results.
7. Analyze and visualize the measurements.

Introduction

Sorting is a fundamental operation in computer science that arranges data into a specific order, most commonly ascending. The efficiency of sorting depends on algorithmic complexity, memory behavior, and data distribution. For this laboratory, four classic sorting techniques are compared: quick sort, merge sort, heap sort, and radix sort. The first three are comparison-based, while radix sort is non-comparison based and leverages digit processing for integer keys.

Comparison Metric

The comparison metric is wall-clock execution time for sorting arrays of increasing size.

Input Format

Each algorithm receives randomized integer arrays with sizes: $10, 100, 500, 1000, 5000, 10000, 50000, 100000$. The values are uniformly generated integers in the range $[0, 1000000]$.

IMPLEMENTATION

All algorithms are implemented in Python. The program prompts the user to select the algorithm, generates random arrays for each size, sorts them, and records the execution time. Results are printed using PrettyTable and plotted using matplotlib.

Quick Sort

Algorithm Description:

Quick sort is a divide-and-conquer algorithm that partitions an array around a pivot, then sorts the partitions recursively. Average time complexity is $O(n \log n)$ with good cache performance.

Pseudocode:

```
QuickSort(A, low, high):
    if low < high:
        p <- partition(A, low, high)
        QuickSort(A, low, p - 1)
        QuickSort(A, p + 1, high)
```

Implementation:

```python
def quick_sort(arr: List[int]) -> List[int]:
    if len(arr) < 2:
        return arr

    stack = [(0, len(arr) - 1)]
    while stack:
        left, right = stack.pop()
        if left >= right:
            continue

        pivot = arr[random.randint(left, right)]
        i, j = left, right
        while i <= j:
            while arr[i] < pivot:
                i += 1
            while arr[j] > pivot:
                j -= 1
            if i <= j:
                arr[i], arr[j] = arr[j], arr[i]
                i += 1
                j -= 1

        if left < j:
            stack.append((left, j))
        if i < right:
            stack.append((i, right))

    return arr
```

Results:

Size	Time (s)
10	0.000080
100	0.000423
500	0.002837
1000	0.001448
5000	0.008413
10000	0.016084
50000	0.084603
100000	0.175823

Figure 1 Results for Quick Sort

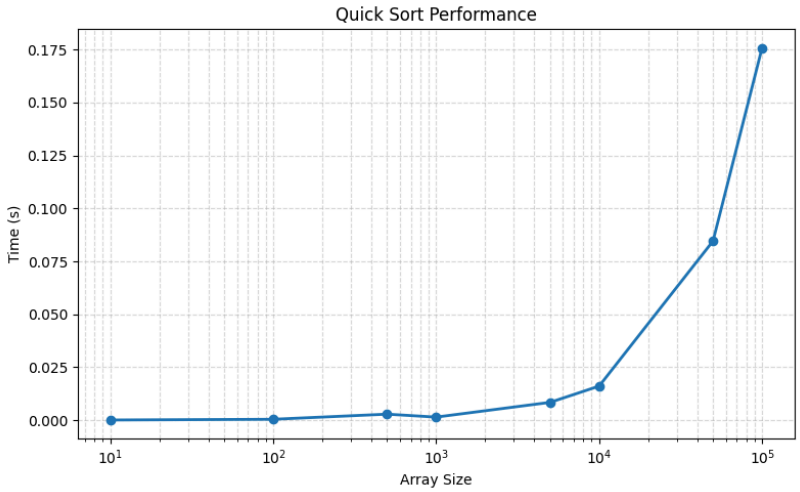

Figure 2 Quick Sort Performance Graph

Merge Sort

Algorithm Description:

Merge sort splits the array, sorts each half, and merges the results. It has stable behavior and guaranteed $O(n \log n)$ time complexity at the cost of additional memory.

Pseudocode:

```
MergeSort(A):
    if |A| <= 1: return A
    mid <- |A|/2
    L <- MergeSort(A[0:mid])
    R <- MergeSort(A[mid:])
    return Merge(L, R)
```

Implementation:

```
def merge_sort(arr: List[int]) -> List[int]:
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
```

```
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    merged: List[int] = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    if i < len(left):
        merged.extend(left[i:])
    if j < len(right):
        merged.extend(right[j:])

    return merged
```

Results:

Size	Time (s)
10	0.000023
100	0.000162
500	0.000732
1000	0.001629
5000	0.010580
10000	0.021984
50000	0.131800
100000	0.272817

Figure 3 Results for Merge Sort

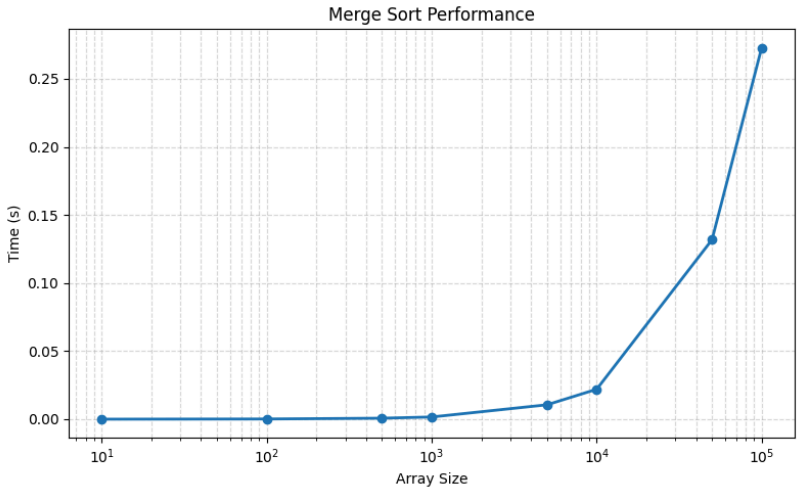

Figure 4 Merge Sort Performance Graph

Heap Sort

Algorithm Description:

Heap sort builds a max heap, then repeatedly extracts the maximum to build a sorted array. It guarantees $O(n \log n)$ time complexity with $O(1)$ extra space.

Pseudocode:

```
HeapSort(A):
    BuildMaxHeap(A)
    for i from n-1 downto 1:
        swap A[0], A[i]
        Heapify(A, 0, i)
```

Implementation:

```python
def heap_sort(arr: List[int]) -> List[int]:
    def sift_down(start: int, end: int) -> None:
        root = start
        while True:
            child = 2 * root + 1
            if child > end:
                return
            if child + 1 <= end and arr[child] < arr[child + 1]:
                child += 1
            if arr[root] < arr[child]:
                arr[root], arr[child] = arr[child], arr[root]
                root = child
            else:
                return

    n = len(arr)
    for start in range((n - 2) // 2, -1, -1):
        sift_down(start, n - 1)

    for end in range(n - 1, 0, -1):
        arr[end], arr[0] = arr[0], arr[end]
        sift_down(0, end - 1)

    return arr
```

Results:

Size	Time (s)
10	0.000056
100	0.000388
500	0.001610
1000	0.001282
5000	0.009898
10000	0.020526
50000	0.124197
100000	0.274203

Figure 5 Results for Heap Sort

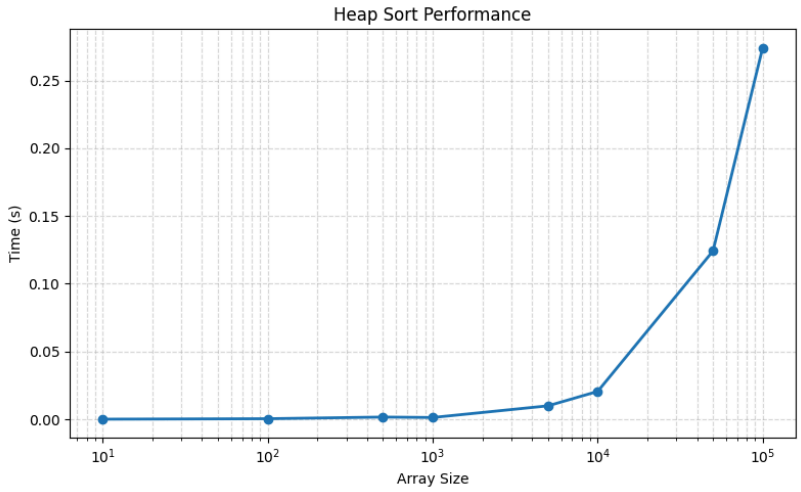

Figure 6 Heap Sort Performance Graph

Radix Sort

Algorithm Description:

Radix sort processes integer digits from least significant to most significant using counting sort as a stable subroutine. For fixed-size integers, the time complexity is $O(kc \cdot n)$.

Pseudocode:

```
RadixSort(A):
    exp <- 1
    while max(A)/exp > 0:
        CountingSortByDigit(A, exp)
        exp <- exp * 10
```

Implementation:

```
def radix_sort(arr: List[int]) -> List[int]:
    if not arr:
        return arr

    max_value = max(arr)
    exp = 1
    output = arr[:]
```

```
    while max_value // exp > 0:
        count = [0] * 10
        for value in output:
            count[(value // exp) % 10] += 1
        for i in range(1, 10):
            count[i] += count[i - 1]

        temp = [0] * len(output)
        for i in range(len(output) - 1, -1, -1):
            digit = (output[i] // exp) % 10
            count[digit] -= 1
            temp[count[digit]] = output[i]

        output = temp
        exp *= 10

    return output
```

Results:

Size	Time (s)
10	0.000139
100	0.000523
500	0.002756
1000	0.005577
5000	0.019421
10000	0.015010
50000	0.063715
100000	0.135337

Figure 7 Results for Radix Sort

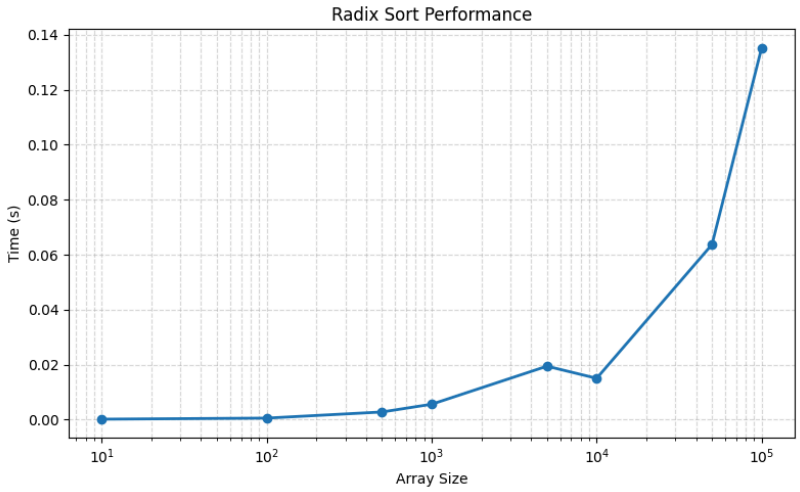

Figure 8 Radix Sort Performance Graph

CONCLUSION

The empirical results show the expected $O(n \log n)$ growth for the comparison-based algorithms and a lower slope for radix sort on fixed-size integers. For small sizes, all algorithms perform within fractions of a millisecond, with minor variability due to constant factors. For large inputs (50,000 and 100,000 elements), radix sort is the fastest, reaching 0.135337 seconds at 100,000 elements. Quick sort performs best among comparison-based methods on the larger datasets, while merge sort and heap sort show similar, slightly higher timings in this experiment.

Overall, radix sort is preferable when keys are non-negative integers with bounded digit length, while quick sort is a strong general-purpose choice for in-memory sorting due to its practical speed and in-place behavior. Merge sort offers predictable performance and stability, and heap sort provides consistent $O(n \log n)$ time with minimal extra space.