Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory Work 2:

Empirical Analysis of Sorting Algorithms

Elaborated:
st. gr. FAF-243

Soimu Ionut

Verified:

asist. univ. Fistic Cristofor

Chișinău – 2026

Contents

1 Algorithm Analysis

1.1 Objective

Study and empirically analyze four sorting algorithms: QuickSort, MergeSort, HeapSort, and ShellSort.

1.2 Tasks

1. Implement QuickSort, MergeSort, HeapSort, and ShellSort 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.

1.3 Theoretical Notes

Sorting is one of the most studied problems in computer science. Given a sequence of n elements, the goal is to rearrange them into non-decreasing order. Its importance extends beyond just ordering data — many other algorithms rely on sorted input as a preprocessing step, including binary search, merging operations, and various computational geometry problems.

Comparison-based sorting algorithms determine order by comparing pairs of elements. It has been proven that any comparison-based sort requires at least $\Omega(n \log n)$ comparisons in the worst case, which means that algorithms achieving $O(n \log n)$ (at least on average) are asymptotically optimal within this class.

The three main strategies used by efficient sorting algorithms are:

1. **Divide and conquer** — the array is split into smaller parts, each part is sorted recursively, and the results are combined (QuickSort, MergeSort).
2. **Selection via data structures** — a heap is used to repeatedly extract the maximum element and place it at the end of the array (HeapSort).
3. **Diminishing increment** — elements far apart are compared and swapped first, progressively reducing the gap until the array is fully sorted (ShellSort).

As with any empirical analysis, the measured execution time depends on the hardware, the programming language, and the specific implementation. The results should therefore be interpreted in terms of relative trends rather than absolute values.

1.4 Introduction

Sorting algorithms have been studied extensively since the early days of computing. As early as 1945, John von Neumann described the merge sort algorithm, and since then dozens of sorting methods have been proposed, each with different trade-offs in time complexity, space usage, and stability.

In practice, the choice of sorting algorithm depends on the context. QuickSort is often the fastest in the average case due to good cache behavior and low constant factors, even though its worst-case complexity is $O(n^2)$. MergeSort guarantees $O(n \log n)$ in all cases but needs additional memory for merging. HeapSort also guarantees $O(n \log n)$ without extra memory, but tends to be slower in practice because of poor cache locality. ShellSort is an interesting generalization of insertion sort that achieves better-than-quadratic performance through gap-based comparisons, and while it cannot match the theoretical optimality of the divide-and-conquer methods, its simplicity makes it worth studying.

In this laboratory, all four algorithms are implemented in Python and tested on randomly generated integer arrays of increasing size. The goal is to observe how each algorithm scales and to compare their practical performance on the same inputs.

1.5 Comparison Metric

The comparison metric for this laboratory is the execution time $T(n)$ of each algorithm, measured using Python's `perf_counter` for high-resolution timing. Each measurement is repeated three times and the shortest time is recorded to reduce the influence of system scheduling and background processes.

1.6 Input Format

Each algorithm receives a randomly generated array of integers. The array sizes range from small to large: 100, 500, 1000, 2000, 5000, 10000, 20000, 30000, 50000, 75000, and 100000. The values in each array are random integers between 0 and $10n$, ensuring a reasonable spread without excessive duplicates. All algorithms receive copies of the same pre-generated arrays for each size so that the comparison is fair. The random seed is fixed for reproducibility.

2 Implementation

All four algorithms are implemented in Python. The implementations prioritize readability over micro-optimization, since the purpose is to compare the algorithms' behavior empirically rather than to squeeze out language-level performance. Each algorithm receives a copy of the input array so that the original data stays unmodified between runs.

The measurement error margin is influenced by OS scheduling and interpreter overhead. For this reason, each test is repeated three times and only the best time is kept. The focus is on overall trends rather than individual data points, and all algorithms are compared using the same input sets.

2.1 Algorithm 1: QuickSort

Description: QuickSort works by selecting a pivot element and then partitioning the array into elements less than, equal to, and greater than the pivot. The three sub-arrays are handled recursively and concatenated. The average-case time complexity is $O(n \log n)$, but the worst case is $O(n^2)$ when the pivot consistently lands on the smallest or largest element. By choosing the middle element as the pivot, this worst case is largely avoided on random data.

Pseudocode:

```
QuickSort(arr):
    if length(arr) <= 1: return arr
    pivot <- arr[length(arr) / 2]
    left <- elements in arr less than pivot
    middle <- elements in arr equal to pivot
    right <- elements in arr greater than pivot
    return QuickSort(left) + middle + QuickSort(right)
```

Python Preview:

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    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 quicksort(left) + middle + quicksort(right)
```

Results:

Table 1: Quicksort Results

n	time (s)
100	0.000119
500	0.000750
1000	0.001689
2000	0.003408
5000	0.009671
10000	0.020397
20000	0.042282
30000	0.074507
50000	0.126202
75000	0.121324
100000	0.134361

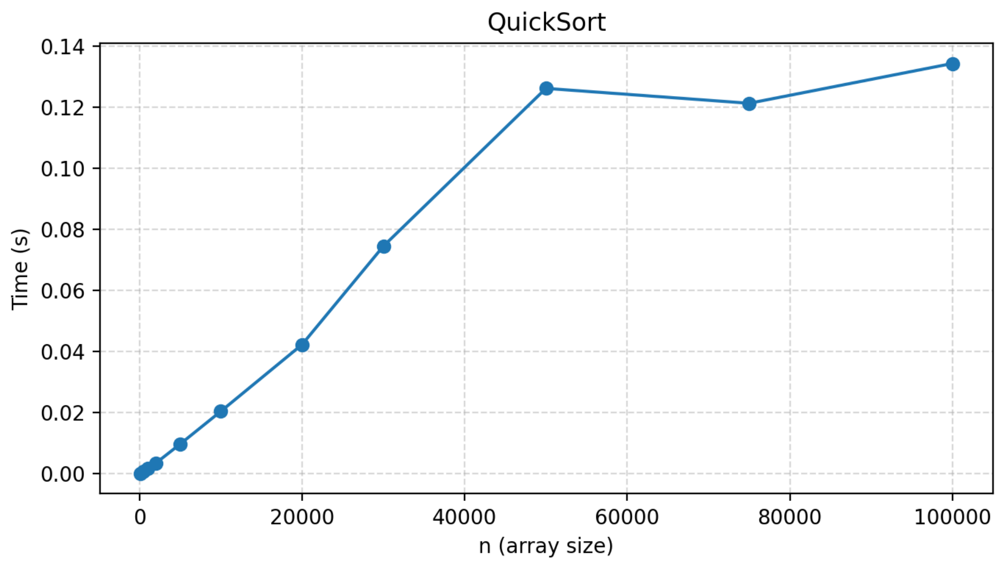

Figure 1: QuickSort Execution Time

The timing curve shows smooth growth throughout the input range. QuickSort benefits from the fact that list comprehensions in Python are quite efficient in practice, and the middle-element pivot keeps the recursion balanced on random data.

2.2 Algorithm 2: MergeSort

Description: MergeSort divides the array in half recursively until each sub-array has at most one element, then merges them back together in sorted order. It guarantees $O(n \log n)$ time in all cases — best, average, and worst. The trade-off is that merging requires $O(n)$ additional space.

Pseudocode:

```
MergeSort(arr):
    if length(arr) <= 1: return arr
    mid <- length(arr) / 2
    left <- MergeSort(arr[0..mid])
    right <- MergeSort(arr[mid..end])
    return Merge(left, right)

Merge(left, right):
    result <- empty list
    while left and right not empty:
        if left[0] <= right[0]:
            append left[0] to result
        else:
            append right[0] to result
    append remaining elements
    return result
```

Python Preview:

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Results:

Table 2: Mergesort Results

n	time (s)
100	0.000092
500	0.000489
1000	0.001058
2000	0.002316
5000	0.006521
10000	0.013822
20000	0.029111
30000	0.045841
50000	0.079870
75000	0.127668
100000	0.173802

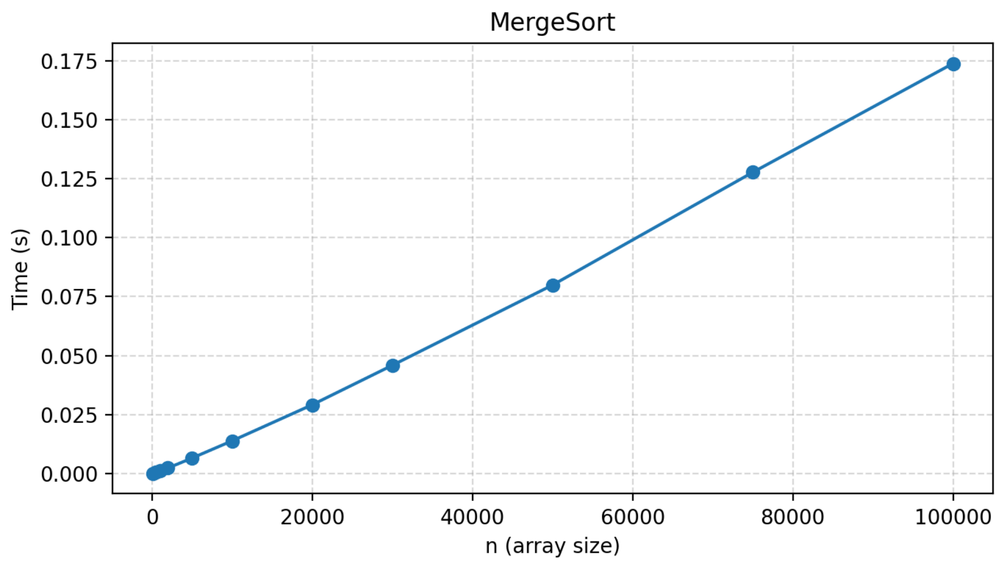

Figure 2: MergeSort Execution Time

MergeSort produces a very consistent curve with no significant spikes. This is expected since its complexity does not depend on the initial ordering — it always divides and merges the same way regardless of the data.

2.3 Algorithm 3: HeapSort

Description: HeapSort first organizes the array into a max-heap, then repeatedly extracts the largest element and places it at the end of the array. Both the heap construction and the extraction phase run in $O(n \log n)$ time, and the algorithm sorts in-place with only $O(1)$ extra space.

Pseudocode:

```
HeapSort(arr):
    n <- length(arr)
    for i from n/2-1 down to 0:
        Heapify(arr, n, i)
    for i from n-1 down to 1:
        swap arr[0] and arr[i]
        Heapify(arr, i, 0)

Heapify(arr, n, i):
    repeat:
        largest <- i
        left <- 2*i + 1
        right <- 2*i + 2
        if left < n and arr[left] > arr[largest]:
            largest <- left
        if right < n and arr[right] > arr[largest]:
            largest <- right
        if largest == i: break
        swap arr[i] and arr[largest]
        i <- largest
```

Python Preview:

```python
def heapify(arr, n, i):
    while True:
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2
        if left < n and arr[left] > arr[largest]:
            largest = left
        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 heapsort(arr):
    arr = arr[:]
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
    return arr
```

Results:

Table 3: Heapsort Results

n	time (s)
100	0.000071
500	0.000477
1000	0.001105
2000	0.002498
5000	0.007337
10000	0.019471
20000	0.036193
30000	0.056823
50000	0.101274
75000	0.158900
100000	0.220366

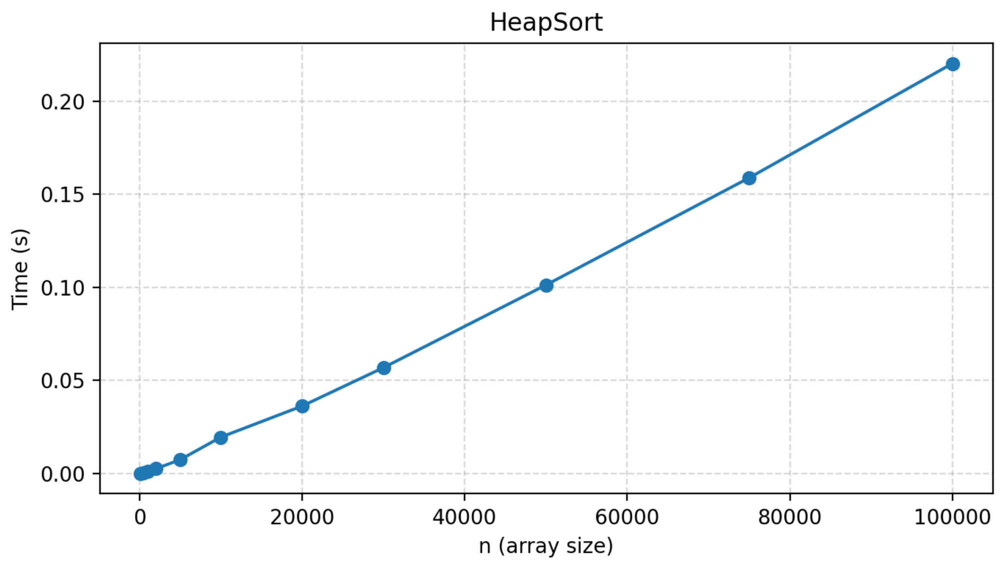

Figure 3: HeapSort Execution Time

HeapSort tends to be slower than QuickSort and MergeSort in practice, even though they share the same asymptotic complexity. The main reason is that the heap operations access memory in a scattered pattern, which leads to more cache misses compared to the sequential access that QuickSort and MergeSort benefit from.

2.4 Algorithm 4: ShellSort

Description: ShellSort is a generalization of insertion sort. Instead of comparing only adjacent elements, it compares elements separated by a gap that decreases over successive passes. When the gap reaches 1 the algorithm becomes a standard insertion sort, but by that point the array is already nearly sorted, so the final pass finishes quickly.

The complexity of ShellSort depends heavily on the gap sequence used. With the original Shell sequence ($n/2, n/4, \ldots, 1$), the worst case is $O(n^2)$, but on random data the empirical performance is closer to $O(n^{1.25})$. More advanced gap sequences exist (Knuth, Sedgewick, Ciura) that bring the worst case down to about $O(n^{4/3})$, but for simplicity we use the classic halving sequence here.

Pseudocode:

```
ShellSort(arr):
    n <- length(arr)
    gap <- n / 2
    while gap > 0:
        for i from gap to n-1:
            temp <- arr[i]
            j <- i
            while j >= gap and arr[j - gap] > temp:
                arr[j] <- arr[j - gap]
                j <- j - gap
            arr[j] <- temp
        gap <- gap / 2
```

Python Preview:

```python
def shell_sort(arr):
    arr = arr[:]
    n = len(arr)
    gap = n // 2
    while gap > 0:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and arr[j - gap] > temp:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
        gap //= 2
    return arr
```

Results:

Table 4: Shellsort Results

n	time (s)
100	0.000051
500	0.000411
1000	0.001145
2000	0.002768
5000	0.009639
10000	0.023249
20000	0.050208
30000	0.079152
50000	0.156590
75000	0.234624
100000	0.364882

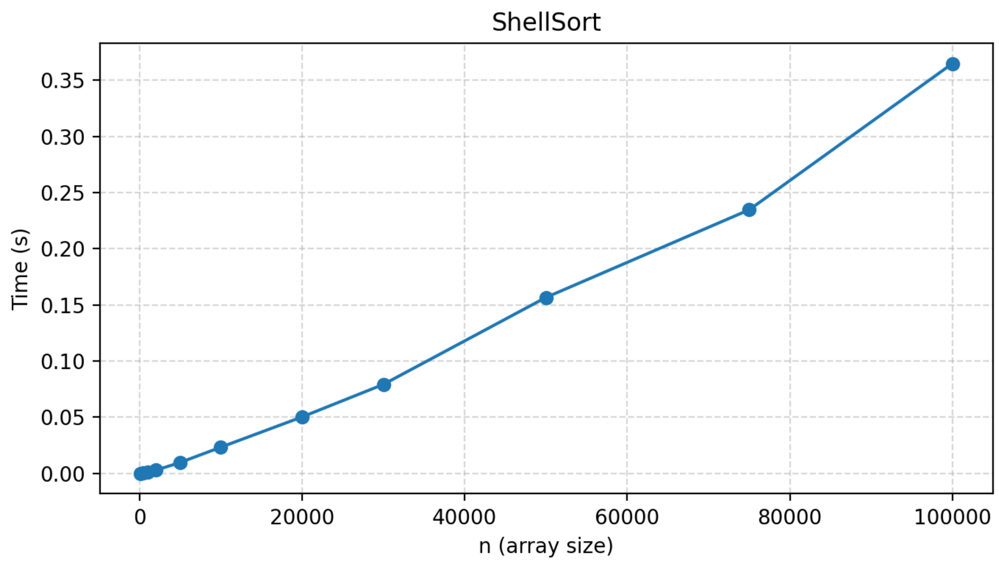

Figure 4: ShellSort Execution Time

ShellSort is the slowest of the four algorithms tested, especially for larger arrays. This reflects the fact that its complexity with Shell's original gap sequence is worse than $O(n \log n)$. Still, it is considerably faster than a naive insertion sort and its simplicity makes it a practical choice for moderately sized arrays or situations where recursion is not desirable.

2.5 Combined Comparison

The following plot shows all four algorithms on the same axes, allowing a direct comparison of their performance across the entire input range.

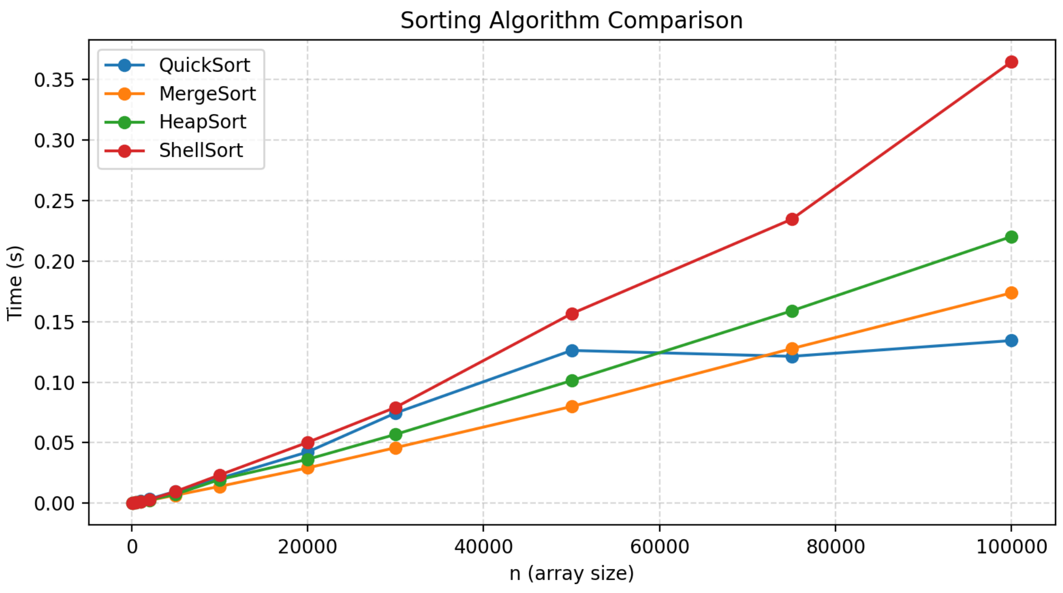

Figure 5: Comparison of All Sorting Algorithms

3 Conclusion

The empirical results confirm the theoretical expectations for the most part. QuickSort and MergeSort are the fastest algorithms in practice, with QuickSort typically having a slight edge thanks to its efficient partitioning and good cache behavior. MergeSort is the most consistent of all four since its performance does not depend on the input ordering, making it a strong choice when predictability matters.

HeapSort, despite its $O(n \log n)$ guarantee and in-place operation, turns out to be noticeably slower than both QuickSort and MergeSort on random data. The main culprit is poor cache locality — the heap operations jump around in memory rather than accessing elements sequentially, and this overhead adds up for large arrays.

ShellSort performs acceptably for small to medium arrays but falls behind the $O(n \log n)$ algorithms as the input grows. Its appeal lies in simplicity: it sorts in-place, uses no recursion, and is easy to implement. With better gap sequences its performance could be improved, but it still cannot match the theoretical optimality of the divide-and-conquer approaches.

Overall, for general-purpose sorting of large datasets, QuickSort with a reasonable pivot strategy or MergeSort are the best choices among these four. HeapSort is useful when memory is constrained, and ShellSort remains a good option for educational purposes and smaller datasets where its simplicity outweighs the performance difference.