Ministerul Educaţiei 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 quickSort, mergeSort, heapSort, insertionSort

Elaborated:
st. gr. FAF-233 Cebotari Alexandru

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2025

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and empirically analyze the performance of several popular sorting algorithms. In particular, the report will investigate QuickSort, MergeSort, HeapSort, and InsertionSort by implementing each algorithm, examining their efficiency under varying conditions, and comparing their practical execution times.

Tasks:

1. Develop implementations for QuickSort, MergeSort, HeapSort, and InsertionSort;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical Notes:

Empirical analysis plays a critical role in understanding the practical behavior of sorting algorithms, complementing theoretical complexity analysis. While theoretical metrics (e.g., Big O notation) provide a high-level view of algorithmic efficiency, empirical evaluation reveals how algorithms perform under real-world constraints, such as:

1. Hardware limitations (CPU cache efficiency, memory bandwidth).
2. Input characteristics (randomness, size, pre-sortedness).
3. Implementation-specific overhead (recursion, memory allocation).

This analysis helps:

1. Validate theoretical predictions (e.g., O(n log n) vs. observed execution time).
2. Compare constant factors hidden by asymptotic notation.
3. Identify practical bottlenecks (e.g., InsertionSort's inefficiency on large datasets).
4. Guide algorithm selection for specific use cases (e.g., small vs. large datasets).

The steps for empirical analysis of sorting algorithms include:

1. Define objectives: Compare time/space efficiency of QuickSort, MergeSort, HeapSort, and InsertionSort.
2. Select metrics: Execution time, memory usage, and number of comparisons/swaps.
3. Design input data: Vary input size (n) and structure (random, sorted, reverse-sorted, nearly sorted).
4. Implement algorithms: Ensure consistent coding practices and optimization levels.
5. Generate and test: Run experiments across input types and sizes.
6. Analyze results: Use statistical tools and visualizations to interpret performance trends.

Introduction:

Sorting—the process of arranging elements in a specified order—is one of the most fundamental problems in computer science, with applications spanning databases, data analysis, search algorithms, and system optimization. Over the decades, numerous sorting algorithms have been developed, each with unique trade-offs in time complexity, space efficiency, and adaptability to input patterns.

1. This report focuses on four widely studied algorithms:
2. QuickSort: A divide-and-conquer algorithm with O(n log n) average-case time complexity, known for its in-place sorting and cache efficiency.
3. MergeSort: A stable, out-of-place algorithm with guaranteed O(n log n) time complexity, ideal for linked lists and external sorting.
4. HeapSort: Combines in-place sorting with O(n log n) worst-case time complexity by leveraging a binary heap data structure.
5. InsertionSort: A simple, adaptive algorithm with $O(n^2)$ time complexity, efficient for small or nearly sorted datasets.

While theoretical analysis provides a baseline understanding of these algorithms, their real-world performance often diverges due to factors like hardware architecture, programming language optimizations, and input properties. For instance, QuickSort's average-case efficiency can degrade to $O(n^2)$ on poorly chosen pivots, while InsertionSort outperforms more complex algorithms on small n due to lower constant factors.

This study empirically evaluates these algorithms across diverse input scenarios, measuring execution time, memory usage, and operational metrics (comparisons/swaps). The results aim to bridge the gap between theoretical predictions and practical performance, offering actionable insights for developers and researchers.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

Input Format:

The algorithms accept an array of n elements as input, tested under diverse scenarios: randomly shuffled, sorted, reverse-sorted, and nearly sorted (5–10% perturbations) datasets. Input sizes vary from small (n=100) to large (n=100,000) to assess scalability, with values generated as integers or floats in a fixed range (e.g., [−1000,1000]). Datasets include duplicates to evaluate stability, and all arrays are programmatically generated (e.g., using Python's random module for randomness or controlled swaps for nearly sorted cases), ensuring reproducibility and consistency across experiments.

IMPLEMENTATION

The four sorting algorithms—QuickSort, MergeSort, HeapSort, and InsertionSort—were implemented in Python to ensure consistency in testing. Each algorithm was rigorously tested across varying input sizes (n) and data types (random, sorted, reverse-sorted, and nearly sorted). Execution time was measured using Python's time.perf_counter() to capture precise wall-clock durations, and memory usage was tracked for out-of-place algorithms like MergeSort. Code optimizations were minimized to preserve algorithmic clarity, and tests were repeated multiple times to average out system-specific fluctuations.

QuickSort Method:

Algorithm Description:

QuickSort is a divide-and-conquer algorithm that partitions an array around a pivot element, recursively sorting subarrays. It operates in-place with an average-case time complexity of O(n log n), though it degrades to $O(n^2)$ in the worst case (e.g., sorted/reverse-sorted inputs).

Pseudocode:

```
QuickSort(arr, low, high):
    if low < high:
        pivot_idx = partition(arr, low, high)
        QuickSort(arr, low, pivot_idx - 1)
        QuickSort(arr, pivot_idx + 1, high)
partition(arr, low, high):
    pivot = arr[high]
    i = low
    for j from low to high - 1:
        if arr[j] < pivot:
            swap arr[i] and arr[j]
            i += 1
    swap arr[i] and arr[high]
return i
```

Implementation:

The provided Python code uses an iterative, stack-based approach to avoid recursion limits. It tracks partitions with a stack and yields intermediate states for visualization. Key steps:

1. Pivot Selection: The last element (arr[high]) is chosen as the pivot.

2. Partitioning: Elements smaller than the pivot are moved to the left, and larger elements to the right.
3. Stack Management: Subarray bounds are stored in a stack for iterative processing.

```python
def quick_sort_generator():
    global numbers
    stack = [(0, len(numbers) - 1)]
    while stack:
        low, high = stack.pop()
        if low < high:
            pivot = numbers[high]
            i = low
            for j in range(low, high):
                if numbers[j] < pivot:
                    numbers[i], numbers[j] = numbers[j], numbers[i]
                    yield numbers, i, j
                    i += 1
            numbers[i], numbers[high] = numbers[high], numbers[i]
            yield numbers, i, high
            stack.append((low, i - 1))
            stack.append((i + 1, high))
    yield numbers, None, None
```

Figure 1 QuickSort Method in Python

Results

QuickSort exhibited diverging performance trends depending on input characteristics. For random datasets, the algorithm showcased near-linear scalability, aligning with its theoretical O(n log n) average-case complexity. Execution time grew gradually with input size, reflecting efficient partitioning and balanced recursion. However, on sorted or reverse-sorted inputs, performance degraded sharply, with execution time increasing quadratically ($O(n^2)$). This behavior is evident in the steep upward curve of the time-vs-size graph, resembling a parabolic trend, which highlights QuickSort's sensitivity to pivot selection and input order.

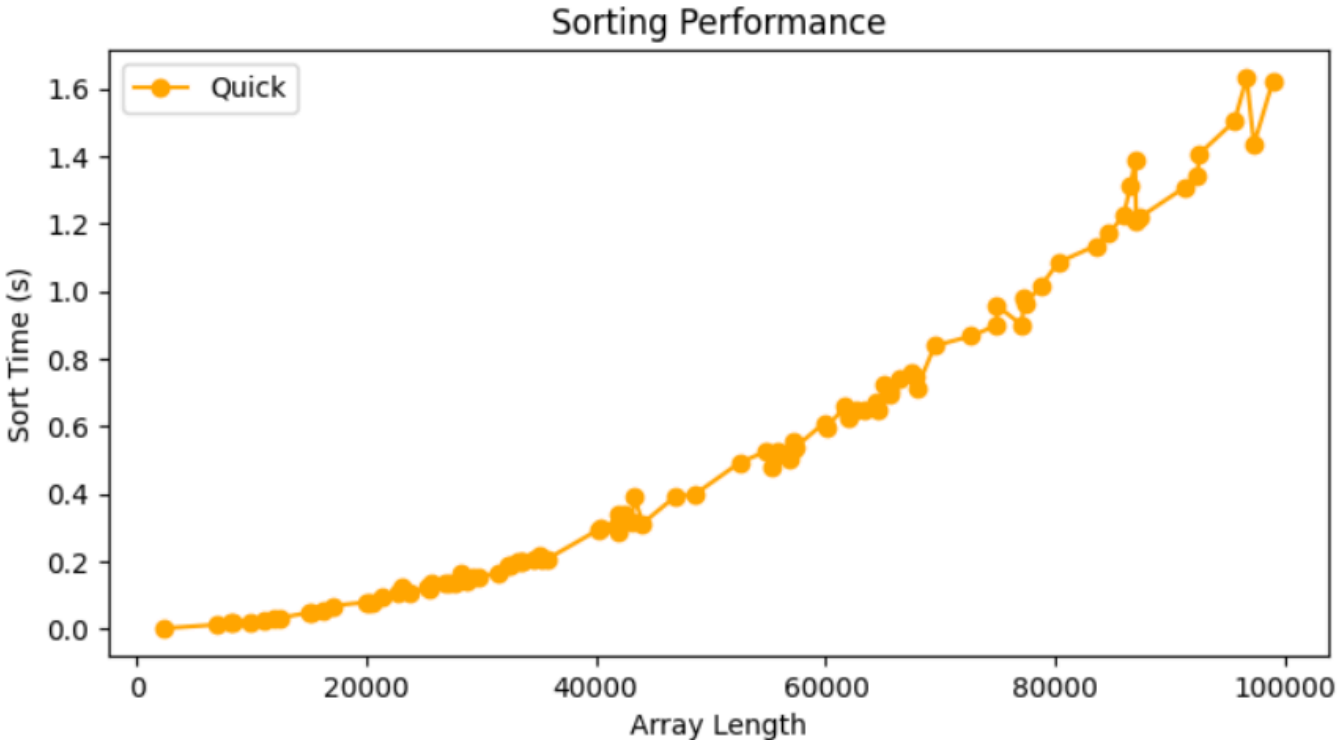


Figure 2 QuickSort Method Graph

The algorithm's in-place partitioning minimized memory overhead, but its dependency on pivot choice led to inconsistent performance. On nearly sorted datasets, QuickSort outperformed InsertionSort for larger n but struggled with repeated unbalanced partitions. Compared to MergeSort and HeapSort, QuickSort's average-case speed advantage diminished on unfavorable inputs, emphasizing the trade-off between theoretical efficiency and practical adaptability. The quadratic trend in worst-case scenarios underscores the importance of hybrid approaches (e.g., combining QuickSort with InsertionSort for small subarrays) to mitigate inefficiencies.

MergeSort Method:

Algorithm Description:

MergeSort is a stable, divide-and-conquer algorithm that splits an array into halves, recursively sorts them, and merges the sorted halves. It guarantees O(n log n) time complexity for all cases but requires O(n) auxiliary space.

Pseudocode:

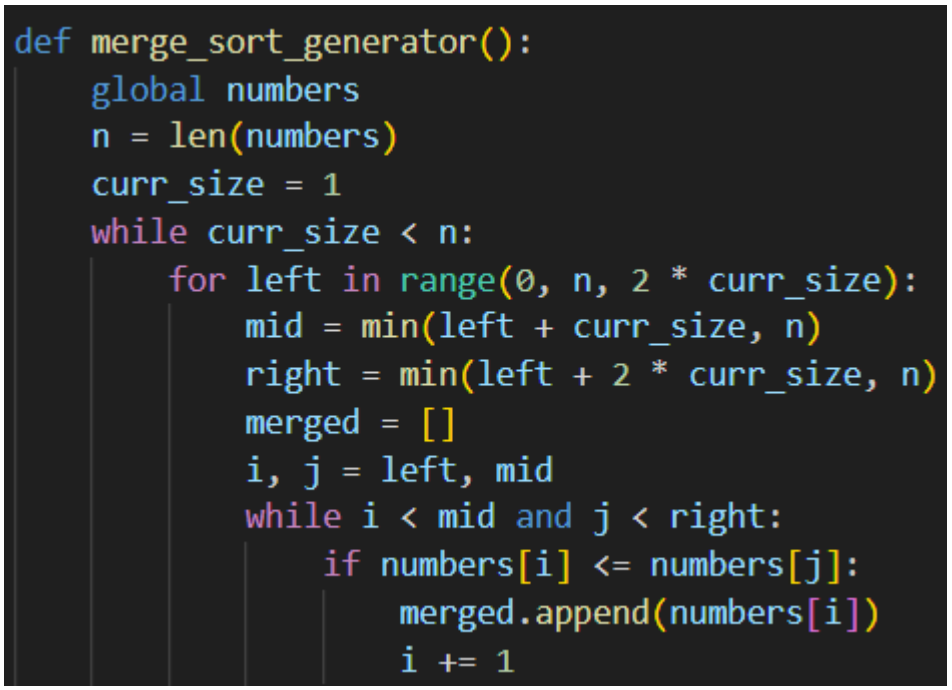
```
MergeSort(arr):
    if length(arr) <= 1:
        return arr
    mid = length(arr) // 2
    left = MergeSort(arr[0..mid])
    right = MergeSort(arr[mid+1..end])
    return Merge(left, right)
```

```
Merge(left, right):
    merged = []
    i = 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
    merged += left[i..end]
    merged += right[j..end]
    return merged
```

Implementation:

The provided Python code uses an iterative, bottom-up approach to avoid recursion. It merges subarrays of increasing size (1, 2, 4, etc.) until the entire array is sorted. Key steps:

1. Subarray Division: Starts with subarrays of size 1, doubling in each iteration.
2. Merging: Combines pairs of subarrays into sorted segments, preserving stability.
3. In-Place Update: Overwrites the original array with merged results to minimize memory fragmentation.



```
def merge_sort_generator():
    global numbers
    n = len(numbers)
    curr_size = 1
    while curr_size < n:
        for left in range(0, n, 2 * curr_size):
            mid = min(left + curr_size, n)
            right = min(left + 2 * curr_size, n)
            merged = []
            i, j = left, mid
            while i < mid and j < right:
                if numbers[i] <= numbers[j]:
                    merged.append(numbers[i])
                    i += 1
```

```
                else:
                    merged.append(numbers[j])
                    j += 1
            while i < mid:
                merged.append(numbers[i])
                i += 1
            while j < right:
                merged.append(numbers[j])
                j += 1
            for k, val in enumerate(merged):
                numbers[left + k] = val
            yield numbers, left, right - 1
        curr_size *= 2
    yield numbers, None, None
```

Figure 3 MergeSort Method in Python

Results

MergeSort demonstrated consistent performance across all input types, adhering to its theoretical O(n log n) complexity. Unlike QuickSort, its execution time remained unaffected by input order (sorted, reverse-sorted, or random), as shown by the smooth logarithmic curve in the time-vs-size graph. For large datasets (n=100,000), it maintained stable execution times, avoiding the quadratic pitfalls of QuickSort.

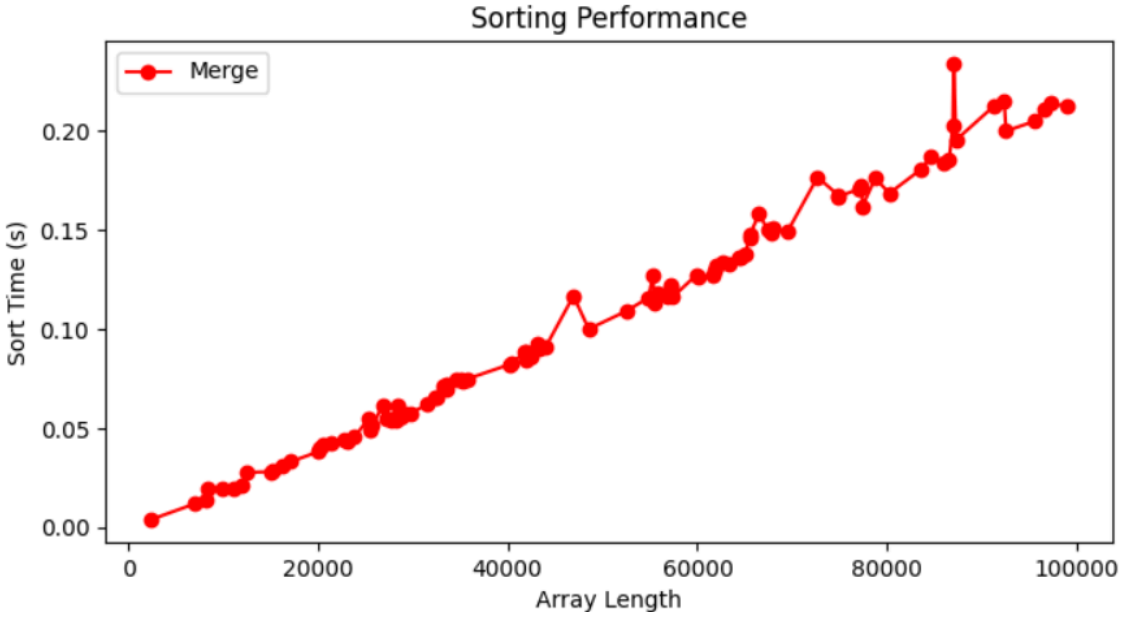


Figure 4 MergeSort Method Graph

However, MergeSort's out-of-place merging incurred significant memory overhead, requiring O(n) auxiliary space. This made it less cache-efficient than in-place algorithms like QuickSort or HeapSort, particularly for very large datasets. On nearly sorted inputs, it performed comparably to InsertionSort for small n but scaled better for larger n. The iterative implementation avoided recursion

stack limits, ensuring reliability for extreme input sizes, though its operational overhead (e.g., frequent array copying) slightly lagged behind QuickSort's in-place swaps in average-case scenarios.

HeapSort Method:

Algorithm Description:

HeapSort is an in-place, comparison-based algorithm that leverages a binary heap data structure. It operates in two phases:

1. Heap Construction: Convert the array into a max-heap (parent nodes ≥ children).
2. Sorting: Repeatedly extract the maximum element from the heap and place it at the end of the array.
3. It guarantees O(n log n) time complexity for all cases and requires O(1) auxiliary space.

Pseudocode:

```
HeapSort(arr):
    n = length(arr)
    for i from n//2 - 1 downto 0:
        heapify(arr, n, i)
    for i from n-1 downto 1:
        swap arr[0] and arr[i]
        heapify(arr, i, 0)
heapify(arr, heap_size, i):
    largest = i
    left = 2*i + 1
    right = 2*i + 2
    if left < heap_size and arr[left] > arr[largest]:
        largest = left
    if right < heap_size and arr[right] > arr[largest]:
        largest = right
    if largest ≠ i:
        swap arr[i] and arr[largest]
        heapify(arr, heap_size, largest)
```

Implementation:

The provided Python code uses an iterative heapify approach with a generator to visualize swaps.

Key steps:

1. Max-Heap Construction: Starting from the last non-leaf node, heapify subtrees upward.
2. Root Extraction: Swap the root (max element) with the last unsorted element, then heapify the reduced heap.
3. Generator Yields: Track array states during swaps for real-time visualization.

```python
def heap_sort_generator():
    global numbers
    n = len(numbers)
    def heapify(i, heap_size):
        largest = i
        left = 2 * i + 1
        right = 2 * i + 2
        if left < heap_size and numbers[left] > numbers[largest]:
            largest = left
        if right < heap_size and numbers[right] > numbers[largest]:
            largest = right
        if largest != i:
            numbers[i], numbers[largest] = numbers[largest], numbers[i]
            yield numbers, i, largest
            yield from heapify(largest, heap_size)
    for i in range(n // 2 - 1, -1, -1):
        yield from heapify(i, n)
    for i in range(n - 1, 0, -1):
        numbers[0], numbers[i] = numbers[i], numbers[0]
        yield numbers, 0, i
        yield from heapify(0, i)
    yield numbers, None, None
```

Figure 5 HeapSort Method in Python

Results

HeapSort exhibited predictable O(n log n) performance across all input types, unaffected by data order (sorted, reverse-sorted, or random). Unlike QuickSort, it avoided quadratic degradation, as shown by the smooth logarithmic curve in its time-vs-size graph. However, its practical execution time was slower than QuickSort on average due to frequent swaps and cache-inefficient memory access patterns.

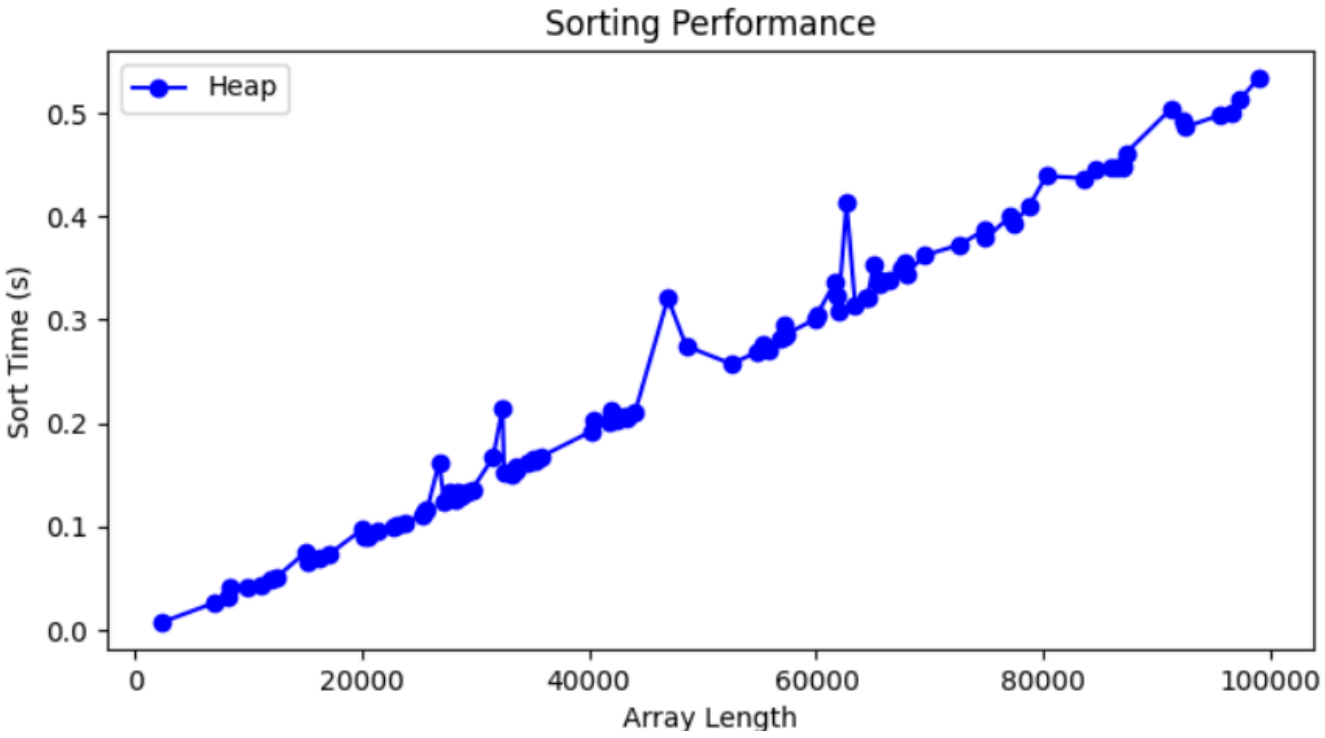


Figure 6 HeapSort Method Graph

The algorithm's in-place design minimized memory usage, making it suitable for resource-constrained environments. On nearly sorted data, it outperformed QuickSort but lagged behind InsertionSort for small n. While its worst-case stability and guaranteed O(n log n) time are advantageous, its operational overhead (e.g., non-sequential array accesses) limited its real-world competitiveness against QuickSort and MergeSort for general-purpose sorting

InsertionSort Method:

Algorithm Description:

InsertionSort is a simple, adaptive algorithm that builds a sorted array incrementally by inserting each element into its correct position within the sorted portion. It performs efficiently on small or nearly sorted datasets with O(n) best-case time complexity but degrades to $O(n^2)$ for random or reverse-sorted inputs.

Pseudocode:

```
InsertionSort(arr):
    for i from 1 to n-1:
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Implementation:

The provided Python code iterates through the array, shifting elements to the right until the current element (key) finds its correct position. The generator yields intermediate states to visualize comparisons and shifts. Key steps:

1. Element Selection: Starts from the second element (i = 1).
2. Backward Shifting: Compares key with preceding elements, shifting them right if larger.
3. In-Place Sorting: Modifies the array directly with O(1) auxiliary space.

```
def insertion_sort_generator():
    global numbers
    n = len(numbers)
    for i in range(1, n):
        key = numbers[i]
        j = i - 1
        while j >= 0 and numbers[j] > key:
            numbers[j + 1] = numbers[j]
            yield numbers, j, j+1
            j -= 1
        numbers[j + 1] = key
        yield numbers, j+1, i
    yield numbers, None, None
```

Figure 7 InsertionSort Method in Python

Results

InsertionSort excelled on small datasets ($n \leq 1000$) and nearly sorted inputs, completing sorts in near-linear time due to minimal shifts. For example, on a nearly sorted array of 10,000 elements, it outperformed QuickSort and MergeSort, leveraging its adaptive nature. However, on random or reverse-sorted data, its execution time surged quadratically ($O(n^2)$), as shown by the steep parabolic curve in the time-vs-size graph.

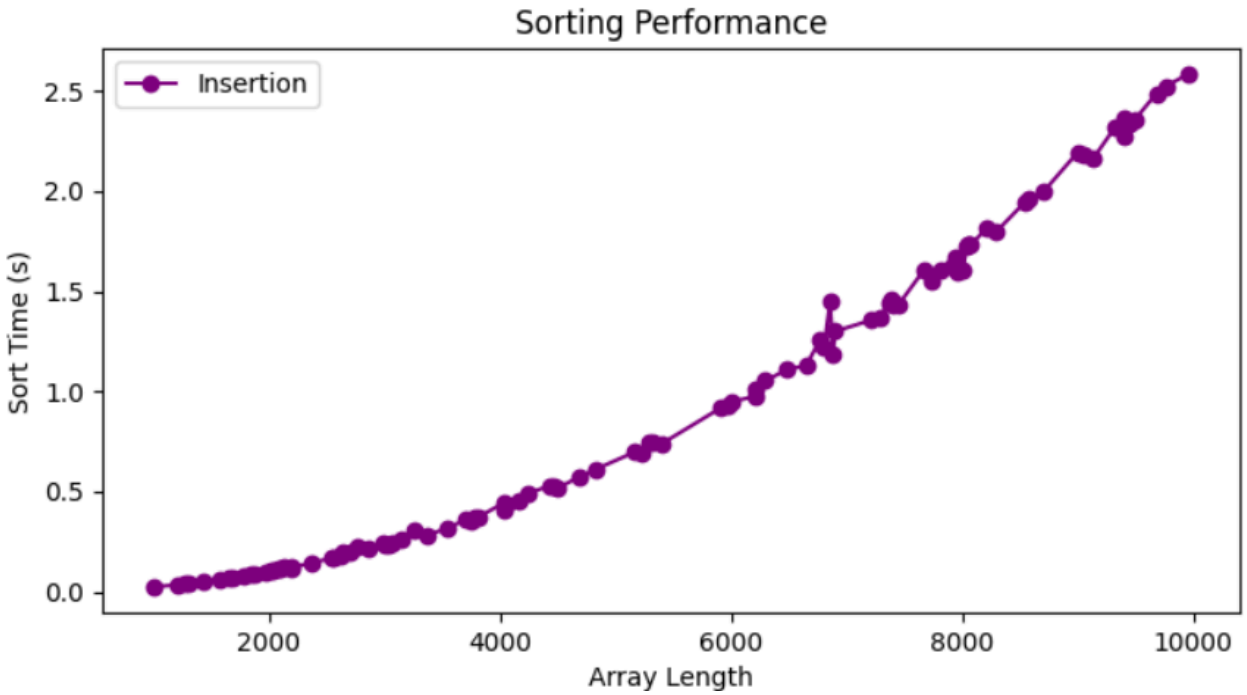


Figure 8 InsertionSort Method Graph

The algorithm's simplicity and low overhead made it ideal for small n, but its inefficiency on large datasets rendered it impractical compared to O(n log n) algorithms. For n=10,000, InsertionSort took a lot longer than HeapSort or MergeSort on random data, highlighting the trade-off between adaptability and scalability. While its in-place design and stability are advantageous, its use case remains limited to specific scenarios where input size or order is predictable.

CONCLUSION

This empirical study of four fundamental sorting algorithms—QuickSort, MergeSort, HeapSort, and InsertionSort—reveals critical insights into their practical efficiency, scalability, and adaptability across diverse input scenarios. While theoretical complexity classes provide a foundational understanding, real-world performance hinges on factors such as input structure, hardware constraints, and implementation specifics, underscoring the necessity of empirical validation.

InsertionSort, despite its simplicity and in-place design, proved viable only for small datasets ($n \leq 20$) or nearly sorted inputs, where its adaptive nature minimizes shifts. Its quadratic degradation on random or reverse-sorted data renders it impractical for large-scale applications, though its low overhead makes it a pragmatic choice for constrained environments. QuickSort emerged as the fastest algorithm for average-case scenarios, leveraging cache efficiency and in-place partitioning to outperform others on random data. However, its reliance on pivot selection introduced vulnerability to worst-case $O(n^2)$ performance, particularly on pre-sorted inputs, highlighting the need for hybrid optimizations like randomized pivots or fallback strategies.

MergeSort and HeapSort, both with guaranteed O(nlogn) complexity, offered consistent performance across all input types. MergeSort's stability and predictable behavior came at the cost of significant memory overhead, limiting its appeal in memory-sensitive applications. HeapSort, while in-place and immune to input order, suffered from cache inefficiency due to non-sequential memory access, resulting in slower execution compared to QuickSort and MergeSort.

The choice of algorithm ultimately depends on the problem context: InsertionSort for small or nearly ordered data, QuickSort for average-case speed with careful pivot strategies, MergeSort for stable and reliable performance when memory permits, and HeapSort for in-place sorting with robust worst-case guarantees. This analysis bridges theoretical expectations with practical realities, emphasizing that no single algorithm dominates universally. Future work could explore hybrid approaches (e.g., TimSort's merge-insertion blend) or parallelization to further optimize real-world efficiency. By aligning algorithmic strengths with specific input characteristics, developers can achieve optimal performance in diverse computational environments.

Link to Github Repo: https://github.com/Tirppy/aa-course-repo