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

Elaborated:
st. gr. FAF-233 Iamandii Ion

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2025

TABLE OF CONTENTS

ALGORITHM ANALYSIS 3
Objective 3
Analysis of quickSort, mergeSort, heapSort, insertionSort 3
Tasks: 3
1 Implement the algorithms listed above in a programming language 3
2 Establish the properties of the input data against which the analysis is performed 3
3 Choose metrics for comparing algorithms 3
4 Perform empirical analysis of the proposed algorithms 3
5 Make a graphical presentation of the data obtained 3
6 Make a conclusion on the work done. 3
Theoretical Notes: 3
Introduction: 4
Comparison Metric: 4
Input Format: 4
IMPLEMENTATION 4
Quick Sort Algorithm: 4
Merge Sort Algorithm: 7
Heap Sort Algorithm: 9
Insertion Sort Algorithm: 12
CONCLUSION 15

ALGORITHM ANALYSIS

Objective

Analysis of quickSort, mergeSort, heapSort, insertionSort

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.

Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction:

Sorting is a fundamental operation in computer science that involves arranging elements in a specific order, typically in ascending or descending sequence. It plays a crucial role in various applications, such as database indexing, searching, data compression, and even in areas like artificial intelligence and computer graphics. Efficient sorting is essential because it significantly improves the performance of other operations that rely on ordered data, such as binary search, merging datasets, and eliminating duplicates. The efficiency of sorting algorithms is measured in terms of their time complexity, space usage, and stability. Some algorithms work better for smaller datasets, while others are optimized for handling large amounts of data efficiently. Additionally, different sorting methods vary in their approach—some use comparison-based techniques like QuickSort and MergeSort, while others, like Counting Sort and Radix Sort, take advantage of the data's properties to sort without direct comparisons. Choosing the right sorting algorithm depends on the nature of the data, the required speed, memory constraints, and whether maintaining the relative order of equal elements (stability) is necessary.

Within this laboratory, we will be analyzing the 4 algorithms empirically.

Comparison Metric:

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

Input Format:

The input for sorting algorithms consists of a list of integers (or comparable elements) of varying sizes, such as 10, 100, 1,000, or more. The list can be in different initial orders—random, sorted, reverse-sorted, or partially sorted—to analyze algorithm performance under various conditions. Inputs may also have different value ranges (small vs. large) and may contain duplicate or unique elements. This structured input format allows for a thorough evaluation of sorting efficiency across different scenarios.

IMPLEMENTATION

All four algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Github repo: https://github.com/ion190/aa-labs/tree/main/lab2

Quick Sort Algorithm:

QuickSort is a divide-and-conquer sorting algorithm that works by selecting a pivot element, partitioning the array into elements smaller and larger than the pivot, and recursively sorting the subarrays. It is known for its average-case time complexity of $O(n \log n)$ and performs exceptionally well on large datasets. However, its worst-case complexity is $O(n^2)$ if an unbalanced partition occurs, such as when the pivot is always the smallest or largest element.

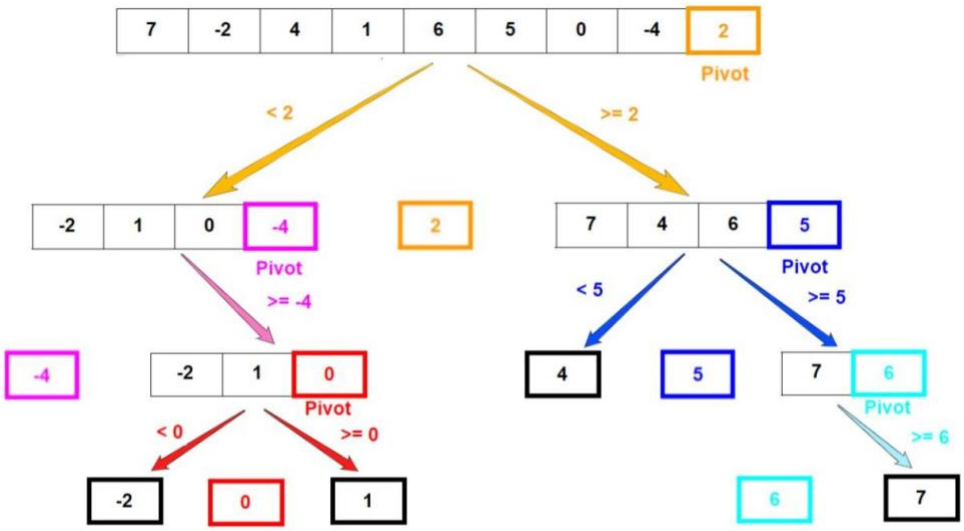

Figure 1. Quick sort example

Algorithm Description:

The quick sort algorithm follows the algorithm as shown in the next pseudocode:

```
QUICKSORT(A, low, high)
  if low < high
    pivotIndex ← PARTITION(A, low, high)
    QUICKSORT(A, low, pivotIndex - 1)
    QUICKSORT(A, pivotIndex + 1, high)


PARTITION(A, low, high)
  pivot ← A[high]
  i ← low - 1
  for j ← low to high - 1
    if A[j] ≤ pivot
      i ← i + 1
      swap A[i] with A[j]
  swap A[i + 1] with A[high]
  return i + 1
```

Implementation:

```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)
```

Figure 2 Quick sort algorithm in Python

Results:

After running the function for different sizes of array and saving the time for each, we obtained the following results:

Input Size	Time Taken
100	0.000126 seconds
500	0.000693 seconds
1000	0.001420 seconds
5000	0.008671 seconds
10000	0.018180 seconds

Figure 3 Results for various set of inputs

The graph showing the time needed to sort different sizes of array:

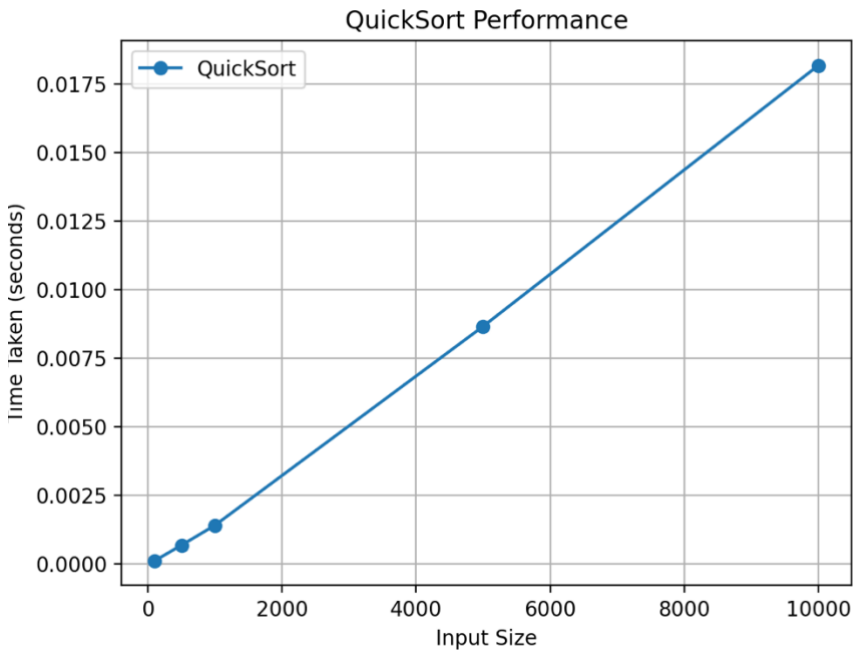

Figure 4 Graph of Quick Sort Function

QuickSort is one of the most efficient sorting algorithms for large datasets due to its O(n log n) average-case complexity. However, it is not stable (doesn't maintain relative order of equal elements) and can degrade to $O(n^2)$ if the pivot is chosen poorly. Implementing randomized or median-of-three pivot selection can mitigate this issue.

Merge Sort Algorithm:

MergeSort is another divide-and-conquer algorithm that splits the array into two halves, recursively sorts them, and then merges the sorted halves. It guarantees a time complexity of O(n log n) for all cases, making it one of the most predictable sorting algorithms.

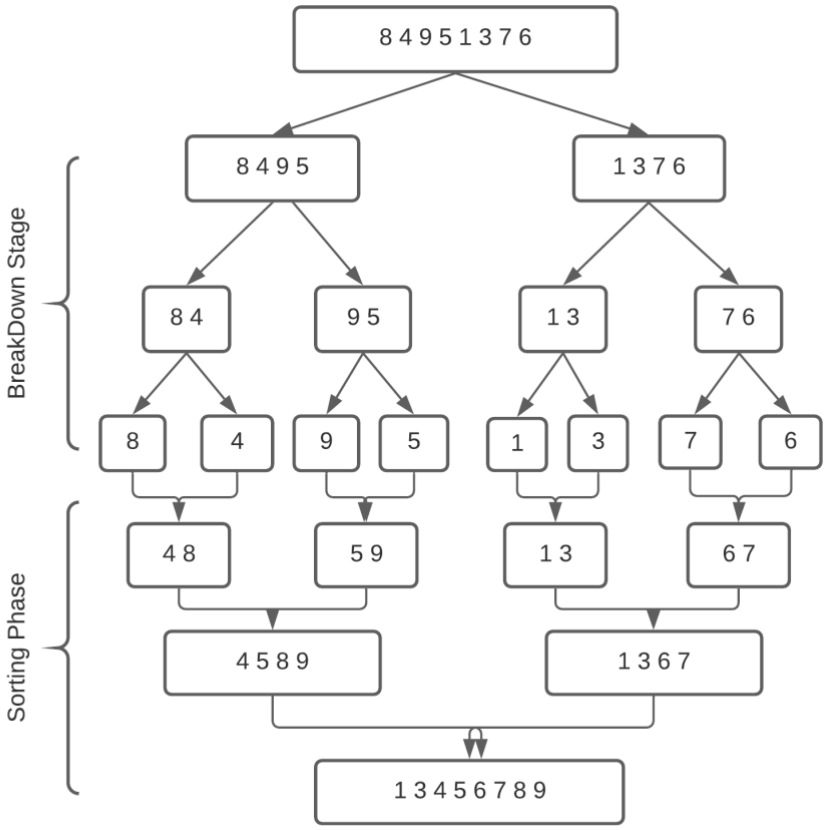

Figure 5. Merge sort example

Algorithm Description:

The merge sort algorithm follows the algorithm as shown in the next pseudocode:

```
MERGESORT(A, left, right)
  if left < right
    mid ← (left + right) / 2
    MERGESORT(A, left, mid)
    MERGESORT(A, mid + 1, right)
```

```
    MERGE(A, left, mid, right)


MERGE(A, left, mid, right)
  Create leftSubarray ← A[left:mid]
  Create rightSubarray ← A[mid+1:right]
  i ← 0, j ← 0, k ← left
  while i < size(leftSubarray) and j < size(rightSubarray)
    if leftSubarray[i] ≤ rightSubarray[j]
      A[k] ← leftSubarray[i]
      i ← i + 1
    else
      A[k] ← rightSubarray[j]
      j ← j + 1
    k ← k + 1
  Copy remaining elements
```

Implementation:

```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
```

Figure 6 Merge sort algorithm in Python

Results:

After running the function for different sizes of array and saving the time for each, we obtained the following results:

Input Size	Time Taken
100	0.000212 seconds
500	0.001172 seconds
1000	0.002383 seconds
5000	0.015644 seconds
10000	0.033390 seconds

Figure 7 Results for various set of inputs

The graph showing the time needed to sort different sizes of array:

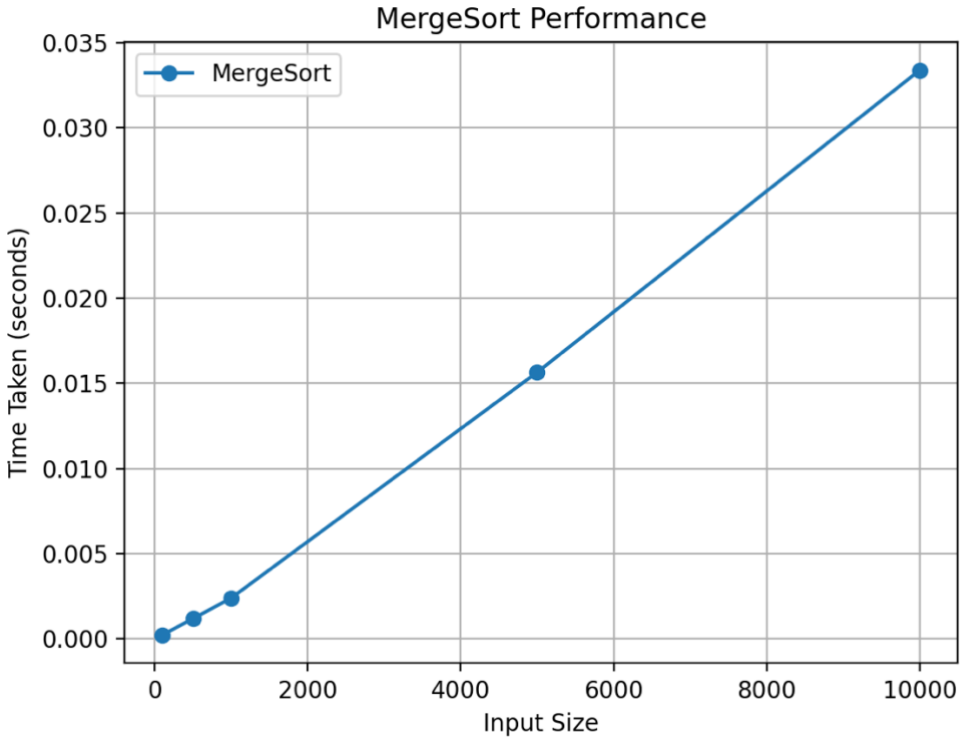

Figure 8 Graph of Merge Sort Function

MergeSort is a stable sorting algorithm that maintains relative order and is ideal for linked lists or datasets where stability matters. However, it requires O(n) additional space, making it less memory-efficient compared to in-place sorting algorithms like QuickSort.

Heap Sort Algorithm:

HeapSort is a comparison-based sorting algorithm that transforms an array into a binary heap (a complete binary tree satisfying the heap property). The largest (or smallest) element is extracted and placed at the end of the array, with heapification ensuring order.

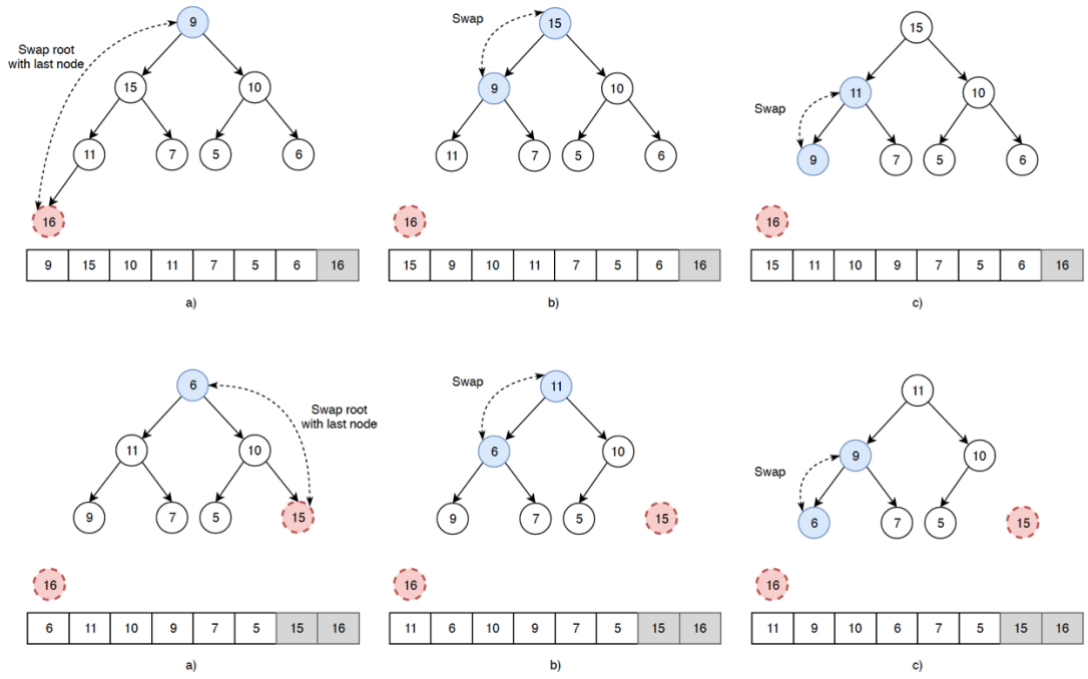

Figure 9. Heap sort example

Algorithm Description:

The heap sort algorithm follows the algorithm as shown in the next pseudocode:

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i ← length(A) down to 2
    swap A[1] with A[i]
    HEAPIFY(A, 1, i - 1)

BUILD-MAX-HEAP(A)
  for i ← floor(length(A)/2) down to 1
    HEAPIFY(A, i, length(A))

HEAPIFY(A, i, heapSize)
  left ← 2i, right ← 2i + 1
  largest ← i
  if left ≤ heapSize and A[left] > A[largest]
    largest ← left
  if right ≤ heapSize and A[right] > A[largest]
```

```
    largest ← right
  if largest ≠ i
    swap A[i] with A[largest]
    HEAPIFY(A, largest, heapSize)
```

Implementation:

```python
def heap_sort(arr):
    def heapify(arr, n, i):
        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:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)

    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[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
```

Figure 10 Heap sort algorithm in Python

Results:

After running the function for different sizes of array and saving the time for each, we obtained the following results:

Input Size	Time Taken
100	0.000174 seconds
500	0.001214 seconds
1000	0.002695 seconds
5000	0.018650 seconds
10000	0.044949 seconds

Figure 11 Results for various set of inputs

The graph showing the time needed to sort different sizes of array:

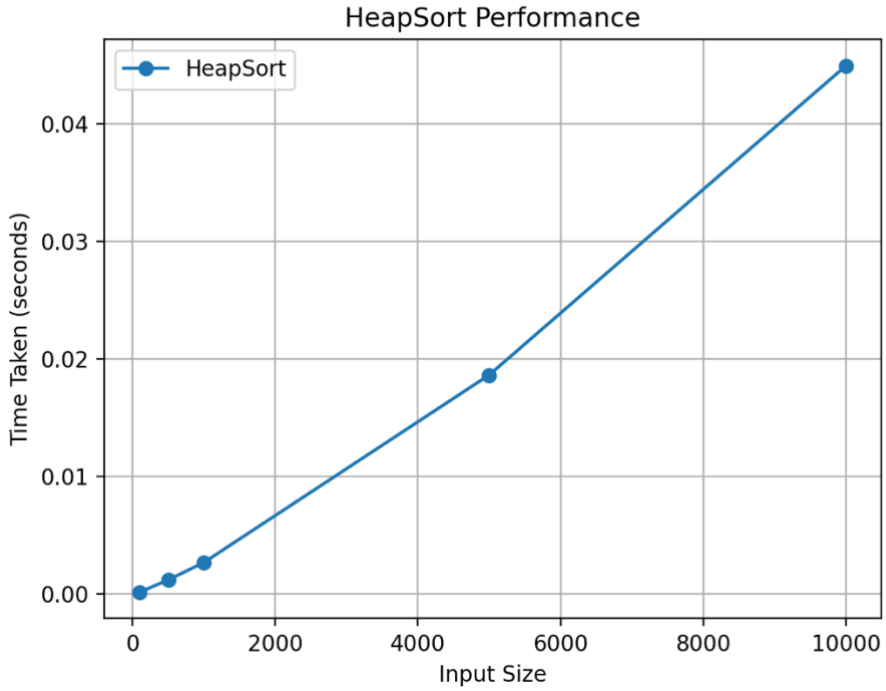

Figure 12 Graph of Heap Sort Function

HeapSort is an in-place sorting algorithm with a guaranteed O(n log n) time complexity. However, it is not stable and has a slightly higher constant factor than QuickSort, making it slower in practical applications. It is commonly used in priority queue implementations.

Insertion Sort Algorithm:

InsertionSort works similarly to sorting playing cards in hand. It picks an element and inserts it into its correct position relative to the already sorted part of the array. It has a worst-case time complexity of $O(n^2)$, making it inefficient for large datasets, but it performs well for small or nearly sorted arrays.

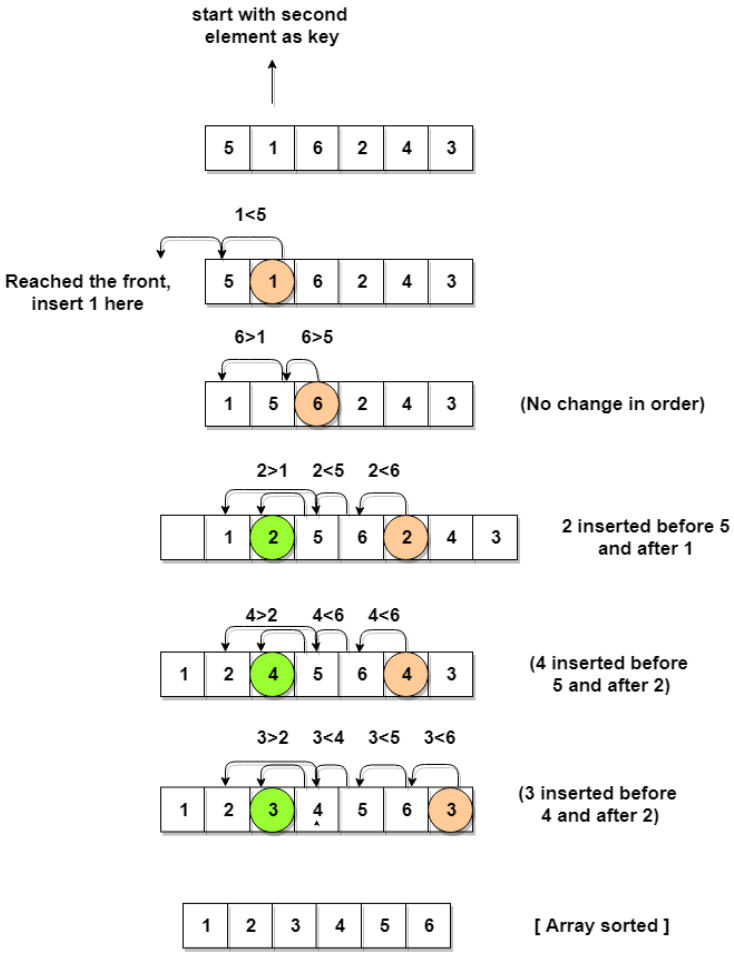

Figure 13. Insertion sort example

Algorithm Description:

The merge sort algorithm follows the algorithm as shown in the next pseudocode:

```
INSERTIONSORT(A)
  for i ← 1 to length(A) - 1
    key ← A[i]
    j ← i - 1
    while j ≥ 0 and A[j] > key
      A[j + 1] ← A[j]
      j ← j - 1
    A[j + 1] ← key
```

Implementation:

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
```

Figure 14 Insertion sort algorithm in Python

Results:

After running the function for different sizes of array and saving the time for each, we obtained the following results:

Input Size	Time Taken
100	0.000235 seconds
500	0.005446 seconds
1000	0.028156 seconds
5000	0.786999 seconds
10000	3.315042 seconds

Figure 15 Results for various set of inputs

The graph showing the time needed to sort different sizes of array:

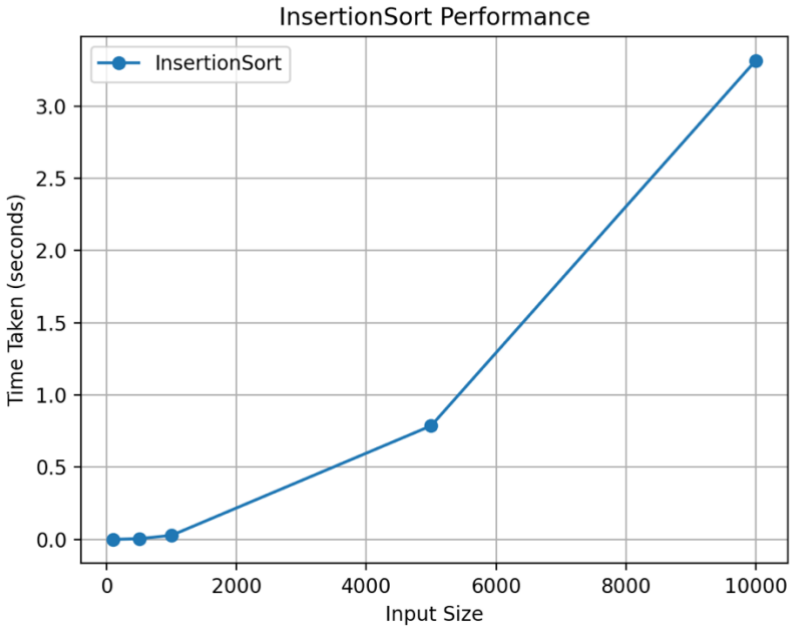

Figure 16 Graph of Insertion Sort Function

InsertionSort is stable, adaptive (performs well on nearly sorted data), and simple to implement, but it suffers from $O(n^2)$ worst-case time complexity. It is often used for small datasets or as a helper in hybrid sorting algorithms.

CONCLUSION

Through empirical analysis, within this paper, four sorting algorithms have been tested for their efficiency in both execution time and suitability for different input sizes, aiming to determine their optimal use cases and potential improvements.

QuickSort, being one of the fastest and most commonly used sorting algorithms, provides an average-case time complexity of $O(n \log n)$, making it ideal for large datasets. However, its worst-case complexity of $O(n^2)$ can be problematic if poor pivot selection occurs, though this can be mitigated with randomized pivot strategies.

MergeSort, while always guaranteeing $O(n \log n)$ complexity, requires additional memory due to its recursive nature, making it less efficient for in-place sorting. However, it is stable, meaning it preserves the order of equal elements, making it preferable in applications where stability is required.

HeapSort, also running in $O(n \log n)$, is slightly slower than QuickSort due to higher constant factors but does not require additional memory, making it useful in priority queue implementations and scenarios where worst-case performance guarantees are needed.

InsertionSort, with its $O(n^2)$ complexity, is inefficient for large datasets but performs exceptionally well on small or nearly sorted inputs, where it can run in $O(n)$ time due to its adaptive nature.

Based on this analysis, QuickSort proves to be the best overall choice for general-purpose sorting due to its balance between speed and memory efficiency. However, MergeSort is the better option when stability is required, HeapSort is useful when memory constraints are a concern, and InsertionSort is ideal for small or nearly sorted datasets.