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 Ploteanu Augustin

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, bubbleSort.

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 one of the most fundamental operations in computer science and is crucial for efficient data retrieval and manipulation. Many sorting algorithms have been developed, each with different characteristics regarding time complexity, space usage, and performance under various conditions.

In this laboratory, we focus on four commonly used sorting algorithms:

QuickSort – A divide-and-conquer sorting algorithm that is generally efficient but performs poorly with specific input patterns.

MergeSort – Another divide-and-conquer approach that provides stable sorting but requires additional memory for merging.

HeapSort – A comparison-based algorithm that uses a binary heap structure.

InsertionSort – A simple and stable sorting algorithm.

Sorting performance is influenced by various factors such as input ordering (already sorted, reversed, or random), algorithmic efficiency, and hardware execution. Some algorithms work well on nearly sorted datasets, while others degrade significantly.

In this laboratory, we will conduct an empirical analysis of the sorting algorithms, measuring their performance on multiple input types and comparing their execution time.

Comparison Metric:

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

Input Format:

To evaluate the performance of sorting algorithms under different conditions, ten datasets of size 100 will be used. These datasets will include predefined structured arrays, consisting of an already sorted (ascending order) array, a completely reversed (descending order) array, a partially sorted array with mixed order (containing both sorted and reversed segments), and an array where two halves are sorted differently. Additionally, five randomized datasets will be generated, each containing 100 random numbers independently produced. By incorporating both structured and randomized input datasets, this approach enables an assessment of sorting algorithm behavior under best-case, worst-case, and average-case conditions.

IMPLEMENTATION

All four algorithms will be implemented in python and 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 report with input will vary depending on memory of the device used.

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

Quick Sort:

Quick Sort is a divide-and-conquer sorting algorithm that selects a pivot element from the array and partitions the remaining elements into two subarrays: those less than or equal to the pivot and those greater than the pivot. The process is recursively applied to each subarray until the base case is reached (when the subarray has one or zero elements). QuickSort typically has an O(n log n) average-case time complexity but can degrade to $O(n^2)$ in the worst case if poor pivot selection occurs. It is an in-place sorting algorithm and does not require additional memory, making it highly efficient for large datasets.

Algorithm Description:

Quick Sort follows the algorithm as shown in the next pseudocode:

```
QUICKSORT(A, low, high)
    if low < high then
        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 do
        if A[j] ≤ pivot then
            i = i + 1
            swap A[i] with A[j]
    swap A[i + 1] with A[high]
    return i + 1
```

Implementation:

```python
def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    return i + 1

def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)
        quickSort(array, low, pi - 1)
        quickSort(array, pi + 1, high)
```

Figure 1 Quick sort in Python

Results:

After running the function for each of the described arrays:

Array #	QuickSort	MergeSort	HeapSort	InsertionSort
1	0.000286	0.000071	0.000102	0.000007
2	0.000198	0.000067	0.000082	0.000254
3	0.000191	0.000067	0.000092	0.000067
4	0.000170	0.000066	0.000093	0.000035
5	0.000109	0.000064	0.000088	0.000124
6	0.000038	0.000072	0.000091	0.000120
7	0.000043	0.000073	0.000090	0.000155
8	0.000072	0.000077	0.000093	0.000154
9	0.000040	0.000074	0.000123	0.000208
10	0.000040	0.000074	0.000138	0.000137
Average	0.000119	0.000070	0.000099	0.000126

Figure 2 Quick sort results

In Figure 3 is represented the table of results for the first set of inputs. The first line represents the times for quick sort. We may notice that it was slow for the predetermined arrays, but the fastest for the randomised ones.

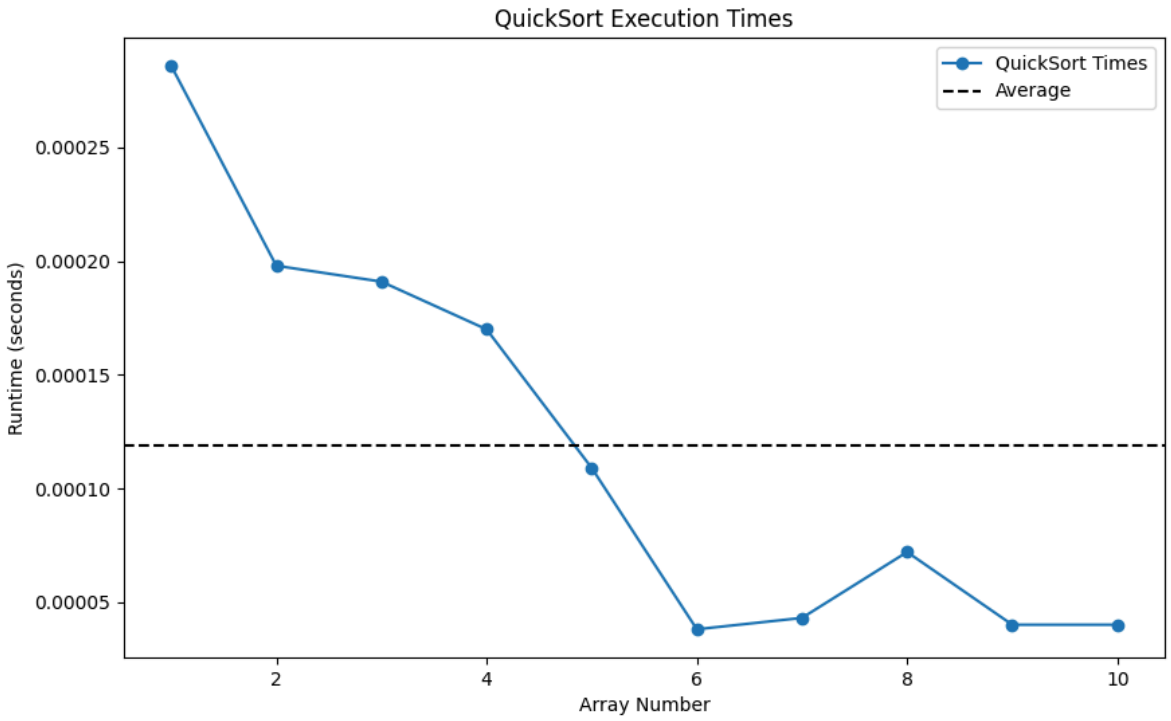

Figure 3 Graph of Ouick Sort

Not only that, but also in the graph in Figure 4 we notice how fast the algorithm is with randomised arrays.

Merge Sort:

MergeSort is a stable, divide-and-conquer sorting algorithm that recursively splits an array into two halves, sorts each half independently, and then merges them back together in sorted order. Unlike QuickSort, MergeSort always has an O(n log n) time complexity, making it efficient and predictable. However, it requires additional memory for merging, making it less space-efficient than in-place sorting methods like QuickSort or HeapSort. MergeSort is often preferred for linked lists or when stable sorting is necessary.

Algorithm Description:

Merge Sort follows the algorithm as shown in the next pseudocode:

```
MERGESORT(A, left, right)
    if left < right then
        mid = (left + right) / 2
        MERGESORT(A, left, mid)
        MERGESORT(A, mid + 1, right)
        MERGE(A, left, mid, right)

MERGE(A, left, mid, right)
    n1 = mid - left + 1
    n2 = right - mid
    create temporary arrays L[n1] and R[n2]

    copy A[left...mid] into L
```

```
copy A[mid+1...right] into R

i = 0, j = 0, k = left
while i < n1 and j < n2 do
    if L[i] ≤ R[j] then
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
    k = k + 1

copy remaining elements of L, if any
copy remaining elements of R, if any
```

Implementation:

```python
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = arr[l:m + 1]
    R = arr[m + 1:r + 1]
    i, j, k = 0, 0, l
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def mergeSort(arr, l, r):
    if l < r:
        m = l + (r - l) // 2
        mergeSort(arr, l, m)
        mergeSort(arr, m + 1, r)
        merge(arr, l, m, r)
```

Figure 4 Merge sort in Python

Results:

After the execution of the merge sort for the aforementioned arrays we obtain the following results:

Array #	QuickSort	MergeSort	HeapSort	InsertionSort
1	0.000286	0.000071	0.000102	0.000007
2	0.000198	0.000067	0.000082	0.000254
3	0.000191	0.000067	0.000092	0.000067
4	0.000170	0.000066	0.000093	0.000035
5	0.000109	0.000064	0.000088	0.000124
6	0.000038	0.000072	0.000091	0.000120
7	0.000043	0.000073	0.000090	0.000155
8	0.000072	0.000077	0.000093	0.000154
9	0.000040	0.000074	0.000123	0.000208
10	0.000040	0.000074	0.000138	0.000137
Average	0.000119	0.000070	0.000099	0.000126

Figure 5 Merge sort results

With the Merge Sort indicated in the second column, although being slower than quick sort for randomised arrays, it is the fastest when including the premade arrays.

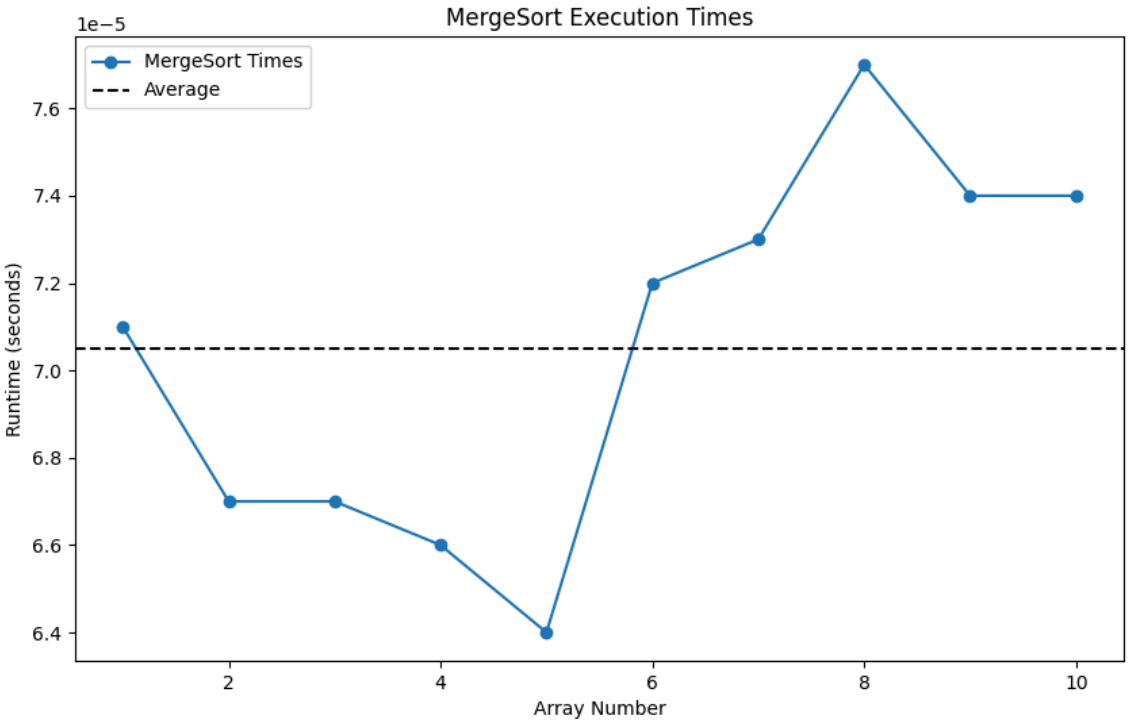

Figure 6 Merge sort graph

Heap Sort:

HeapSort is a comparison-based sorting algorithm that transforms an array into a binary heap data structure and repeatedly extracts the maximum (or minimum) element from the heap to build a sorted array. The process consists of two main steps: heap construction (building a max-heap in O(n) time) and heap extraction (removing the root and re-heapifying, which takes O(log n) per operation). The overall time complexity is O(n log n) in all cases, making it efficient for large datasets. Unlike QuickSort and MergeSort, HeapSort is not stable but is in-place, requiring no extra memory beyond the input array.

Algorithm Description:

Heap Sort follows the algorithm as shown in the next pseudocode:

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

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

HEAPIFY(A, i, heapSize)
    left = 2 * i
    right = 2 * i + 1
    largest = i

    if left ≤ heapSize and A[left] > A[largest] then
        largest = left
    if right ≤ heapSize and A[right] > A[largest] then
        largest = right
    if largest ≠ i then
        swap A[i] with A[largest]
        HEAPIFY(A, largest, heapSize)
```

Implementation:

```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[l] > arr[largest]:
        largest = l
    if r < n and arr[r] > arr[largest]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heapSort(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)
```

Figure 7 Heap sort in Python

Results:

After the execution of the function for the arrays we obtain the following results:

Array #	QuickSort	MergeSort	HeapSort	InsertionSort
1	0.000286	0.000071	0.000102	0.000007
2	0.000198	0.000067	0.000082	0.000254
3	0.000191	0.000067	0.000092	0.000067
4	0.000170	0.000066	0.000093	0.000035
5	0.000109	0.000064	0.000088	0.000124
6	0.000038	0.000072	0.000091	0.000120
7	0.000043	0.000073	0.000090	0.000155
8	0.000072	0.000077	0.000093	0.000154
9	0.000040	0.000074	0.000123	0.000208
10	0.000040	0.000074	0.000138	0.000137
Average	0.000119	0.000070	0.000099	0.000126

Figure 8 Heap sort results

With heap sort indicated in the third column, although being slower than quick sort for randomised arrays, still performing pretty well, the graph indicates a pretty solid T(n) time complexity.

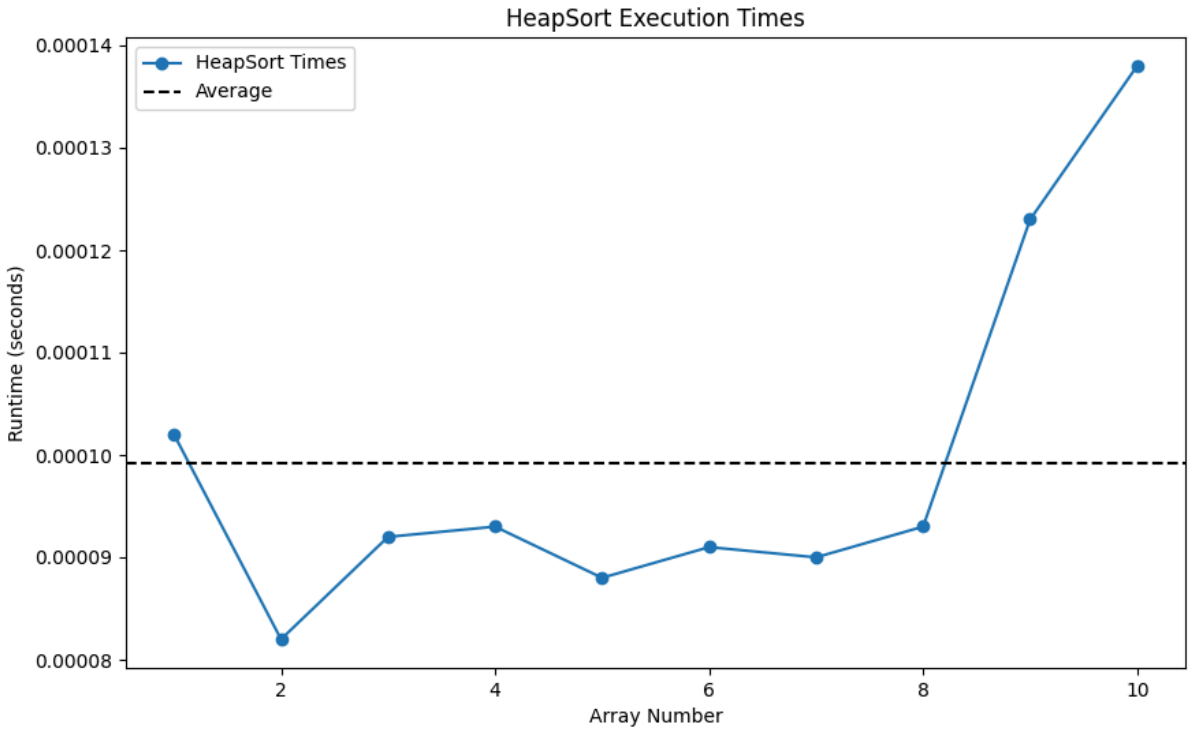

Figure 9 Heap sort graph

Insertion Sort:

Insertion Sort is a simple, stable, and in-place sorting algorithm that builds the sorted array one element at a time. It works by iterating through the input list and moving each element to its correct position within the already sorted portion of the array. At each step, the algorithm picks the next element, compares it with the elements in the sorted section, and shifts larger elements one position to the right until the correct position is found. This process continues until all elements are sorted.

Algorithm Description:

Insertion Sort can be described in pseudocode as follows:

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

Implementation:

The implementation of the function in Python is as follows:

```python
def insertionSort(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
```

Figure 10 Insertion sort in Python

Results:

Inserion sort times can be seen in the rightmost column, being the slowest algorithm from the ones implemented

Array #	QuickSort	MergeSort	HeapSort	InsertionSort
1	0.000286	0.000071	0.000102	0.000007
2	0.000198	0.000067	0.000082	0.000254
3	0.000191	0.000067	0.000092	0.000067
4	0.000170	0.000066	0.000093	0.000035
5	0.000109	0.000064	0.000088	0.000124
6	0.000038	0.000072	0.000091	0.000120
7	0.000043	0.000073	0.000090	0.000155
8	0.000072	0.000077	0.000093	0.000154
9	0.000040	0.000074	0.000123	0.000208
10	0.000040	0.000074	0.000138	0.000137
Average	0.000119	0.000070	0.000099	0.000126

Figure 11 Insertion sort results

And as shown in its performance graph,

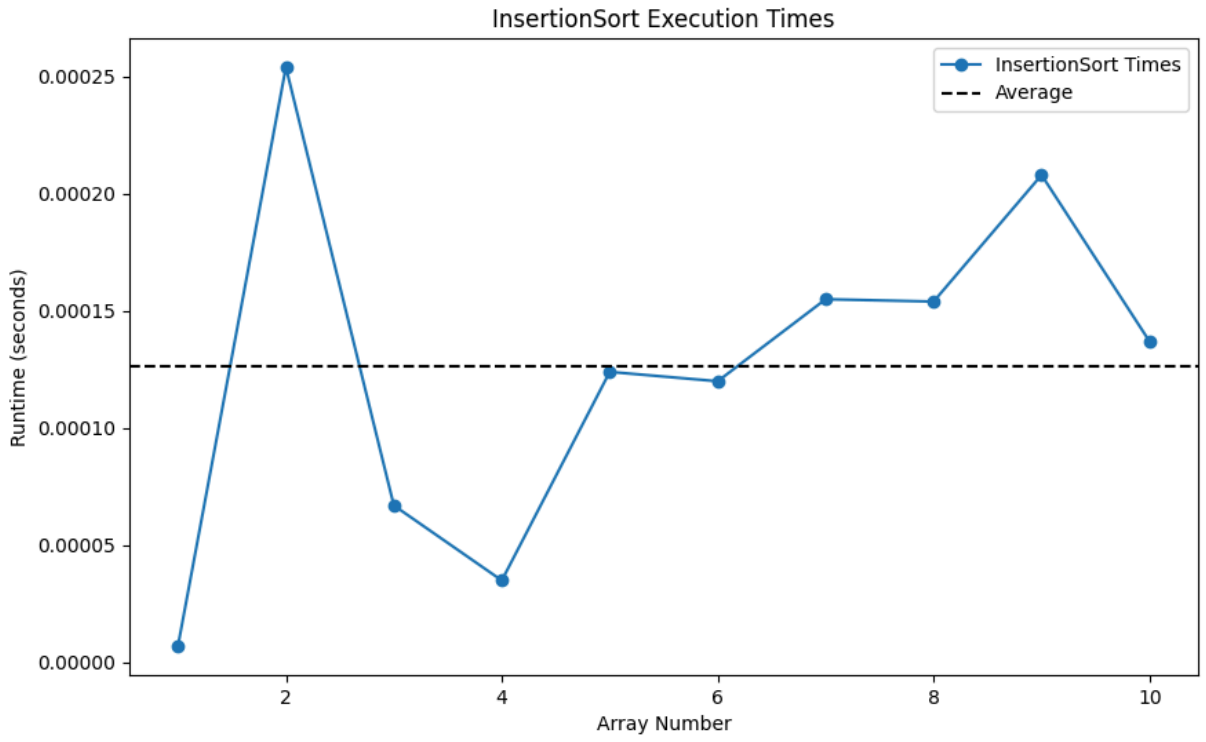

Figure 12 Insertion sort graph

Insertion sort is the slowest out of the implemented algorithms, but still performing pretty well, the graph indicates a pretty solid T(n) time complexity.

CONCLUSION

Through empirical analysis, this study has evaluated the efficiency of four sorting algorithms—QuickSort, MergeSort, HeapSort, and InsertionSort—by analyzing their execution time under different input conditions. The goal was to identify the scenarios in which each algorithm performs optimally and to determine their feasibility for various applications. The results provide insights into the strengths and limitations of each approach, as well as opportunities for further optimizations.

QuickSort, known for its divide-and-conquer approach, demonstrated strong performance with an O(n log n) average-case complexity, making it ideal for most general-purpose sorting tasks. However, its efficiency can degrade to $O(n^2)$ in the worst case, particularly with poor pivot selection, necessitating optimizations like randomized pivoting.

MergeSort, offering stable and predictable performance with a guaranteed O(n log n) complexity, proved to be highly reliable. However, its additional memory requirements for merging make it less space-efficient compared to QuickSort, which operates in-place.

HeapSort, with its O(n log n) complexity in all cases, provided consistent results and required no additional memory. However, its execution time was slightly higher than QuickSort due to the overhead of maintaining the heap structure, making it more suitable for applications where stability is not a concern, and memory usage is critical.

InsertionSort, being a simple and stable sorting algorithm, performed well on small datasets or nearly sorted inputs, achieving O(n) complexity in the best case. However, its $O(n^2)$ worst-case complexity makes it inefficient for large datasets, as each element may need to be compared and shifted multiple times. Despite its limitations, InsertionSort is useful in scenarios where minimal swaps are needed and is commonly used in hybrid sorting algorithms as a base case for small partitions.

In conclusion, QuickSort remains the preferred choice for general use, MergeSort is suitable for scenarios requiring stability, HeapSort offers predictable performance with minimal additional memory requirements, and InsertionSort is efficient for small or nearly sorted datasets but impractical for large unsorted inputs. Future optimizations, such as hybrid approaches combining these algorithms, could further enhance sorting efficiency based on input characteristics.