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-221 Chichioi Iuliana

Verified: Andrievschi-Bagrin Veronica

asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHMS ANALYSIS

Objective

Study and analyze different sorting algorithms suck as 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:

Sorting algorithms are fundamental in computer science, aiding in the organization and arrangement of data in various applications. QuickSort, MergeSort, HeapSort, and InsertionSort are commonly used sorting algorithms, each with its unique characteristics and efficiency.

QuickSort, based on the divide-and-conquer strategy, partitions an array into two sub-arrays around a pivot element. It is known for its average-case time complexity of O(n log n) and is widely used in practice.

MergeSort also utilizes the divide-and-conquer approach by recursively dividing the array into smaller sub-arrays until they are sorted, then merging them back together. It guarantees a time complexity of O(n log n) in all cases but requires additional space for merging.

HeapSort constructs a binary heap from the array elements and repeatedly extracts the maximum (for max heap) or minimum (for min heap) element to obtain a sorted array. It has a time complexity of O(n log n) and is not stable but has in-place sorting characteristics.

InsertionSort is a simple sorting algorithm that iterates through the array, repeatedly taking each element and inserting it into its correct position in the already sorted portion of the array. It has a time complexity of O(n^2) but performs well for small datasets or nearly sorted arrays

Introduction:

Sorting algorithms play a crucial role in various computational tasks, from data processing to algorithmic optimizations. The purpose of this analysis is to study and empirically evaluate the performance of four prominent sorting algorithms: QuickSort, MergeSort, HeapSort, and InsertionSort. By implementing these algorithms and analyzing their behavior under different input conditions, we aim to gain insights into their efficiency and suitability for different scenarios.

Comparison Metric:

For comparing the algorithms, we will primarily focus on their time complexity, measured in terms of execution time for sorting varying sizes of input data. Additionally, we will consider factors such as space complexity, stability, and adaptability to different input distributions.

Quick Sort:

Time Complexity: Quick Sort typically exhibits an average-case time complexity of O(n log n). However, in the worst-case scenario, it can degrade to O(n^2) when poorly chosen pivots cause unbalanced partitioning.
Space Complexity: Quick Sort is an in-place sorting algorithm, meaning it sorts the array in situ without requiring additional memory proportional to the input size.
Stability: Quick Sort is not stable since it may change the relative order of equal elements during partitioning and swapping.
Adaptability: Quick Sort performs well on a wide range of input distributions, including random, nearly sorted, and partially reversed arrays. It adapts efficiently to various input scenarios.

Merge Sort:

Time Complexity: Merge Sort guarantees a time complexity of O(n log n) in all cases, making it a consistent performer regardless of the input distribution.
Space Complexity: Merge Sort requires additional space proportional to the size of the input array for merging the sub-arrays, resulting in a space complexity of O(n).
Stability: Merge Sort is stable, meaning it preserves the relative order of equal elements during the merging process.
Adaptability: Merge Sort performs consistently well across different input distributions due to its divide-and-conquer approach, which ensures balanced splitting and merging of arrays.

Heap Sort:

Time Complexity: Heap Sort has a time complexity of O(n log n), making it efficient for large datasets. However, it is not optimal for nearly sorted arrays due to its inherent lack of adaptability.
Space Complexity: Heap Sort operates in situ and requires only a constant amount of additional memory, resulting in a space complexity of O(1).
Stability: Heap Sort is not stable since it involves swapping elements based on the heap property, which may change the relative order of equal elements.
Adaptability: Heap Sort performs consistently well on random and uniformly distributed inputs but may exhibit suboptimal performance on nearly sorted or partially reversed arrays.

Insertion Sort:

Time Complexity: Insertion Sort has a time complexity of O(n^2) in the worst-case scenario. However, it performs efficiently on small datasets and nearly sorted arrays, making it suitable for certain applications.
Space Complexity: Insertion Sort operates in situ and requires only a constant amount of additional memory, resulting in a space complexity of O(1).
Stability: Insertion Sort is stable, meaning it preserves the relative order of equal elements during the sorting process.
Adaptability: Insertion Sort performs well on partially sorted or nearly sorted arrays but may exhibit poor performance on large datasets or arrays with significant randomness.

Input Format:

The input data consists of arrays of integers with predefined sizes ranging from 100 to 100,000 elements. These sizes are specified in the input_sizes list:

```
# Define input sizes
input_sizes = [100, 500, 1000, 5000, 10000, 50000, 100000]
arr = [[random.randint( a: 0,  b: 1000000) for _ in range(length)] for length in input_sizes]
```

Figure 1 input_sizes

Arrays are generated for each size in input_sizes, with random integers in the range of 0 to 1,000,000. The generation process ensures diverse input scenarios for comprehensive analysis of the sorting algorithms' performance.This input format facilitates thorough evaluation of the algorithms across different dataset sizes, enabling insights into their efficiency and scalability.

IMPLEMENTATION

All 4 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 o memory of the device used.

Quick_Sort:

Divide and Conquer algorithm:

- Breaks down problem into multiple subproblems recursively until they become simple to solve
- Solutions are combined to solve original problem

Running time:

- O(n2) worst case
- O(n * log(n)) best and average case

Algorithm Description:

1. Choose pivot element (Usually last or random)
2. Stores elements less than pivot in left subarray
   Stores elements greater than pivot in right subarray
3. Call quicksort recursively on left subarray
   Call quicksort recursively on right subarray

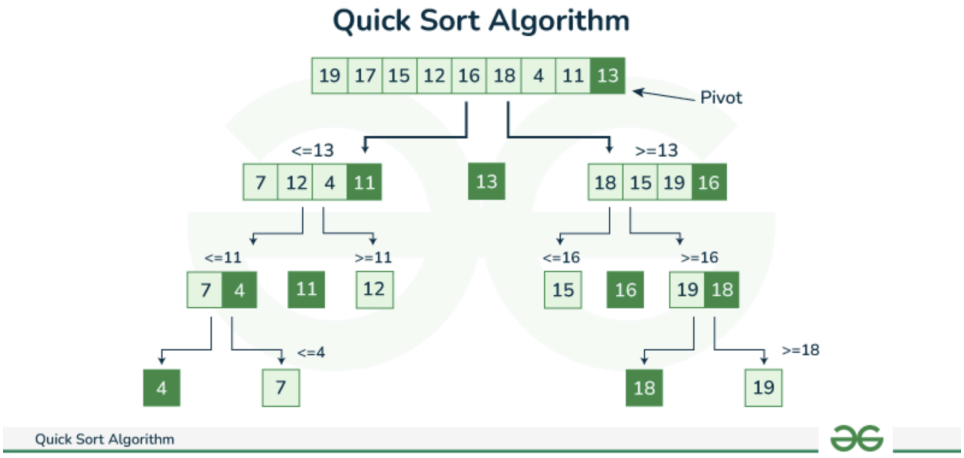

Figure 2 Quick_Sort Algorithm

Implementation:

```python
def quick_sort(arr, left, right):
    if left < right:
        # Choose pivot using median-of-three strategy
        mid = (left + right) // 2
        if arr[mid] < arr[left]:
            arr[mid], arr[left] = arr[left], arr[mid]
        if arr[right] < arr[left]:
            arr[right], arr[left] = arr[left], arr[right]
        if arr[right] < arr[mid]:
            arr[right], arr[mid] = arr[mid], arr[right]

        pivot = arr[mid]

        # Partitioning
        i = left - 1
        j = right + 1
        while True:
            i += 1
            while arr[i] < pivot:
                i += 1
            j -= 1
            while arr[j] > pivot:
                j -= 1
            if i >= j:
                break
            arr[i], arr[j] = arr[j], arr[i]

        # Recursively sort partitions
        quick_sort(arr, left, j)
        quick_sort(arr, j + 1, right)
```

Figure 3 Quick_Sort Implementation

Results:

Quick Sort:

	Input Size	Quick Sort
0	100	0.000049
1	500	0.000254
2	1000	0.000573
3	5000	0.003468
4	10000	0.007123
5	50000	0.040716
6	100000	0.083704

Figure 4 Quick_Sort Implementation

Graph Representation:

Figure 5 Quick_Sort Graph Representation

Merge_Sort:

Divide and Conquer algorithm:

- Breaks down problem into multiple subproblems recursively until they become simple to solve
- Solutions are combined to solve original problem

Running time:

- O(n * log(n)) running time
- Optimal running time for comparison based algorithms

Algorithm Description:

1. Split array in half
2. Call Merge Sort on each half to sort them recursively
3. Merge both sorted halves into one sorted array

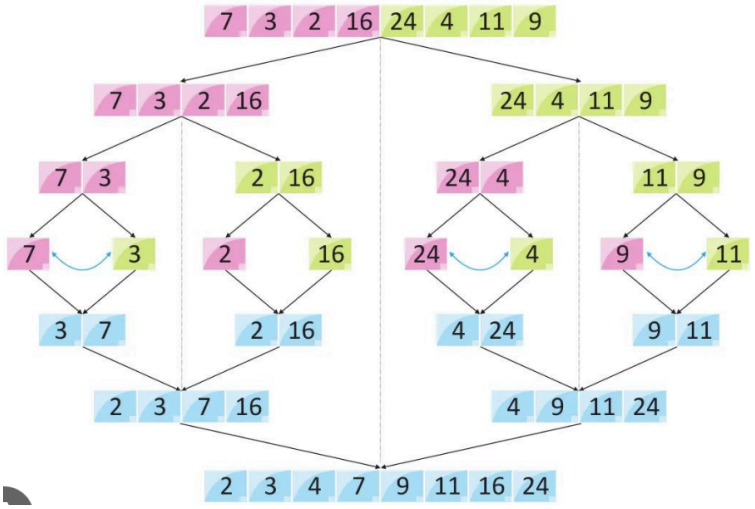

Figure 6 Merge_Sort Algorithm

Implementation:

```
def merge_sort(arr):
    if len(arr) > 1:
        left_arr = arr[:len(arr) // 2]  # from beginning to middle point
        right_arr = arr[len(arr) // 2:]  # from middle point to the end

        # recursion
        merge_sort(left_arr)
        merge_sort(right_arr)

        # merge
        i = 0  # leftmost element on left arr
        j = 0  # leftmost element on right arr
        k = 0  # index in the merged arr

        while i < len(left_arr) and j < len(right_arr):
            if left_arr[i] < right_arr[j]:
                arr[k] = left_arr[i]
                i += 1
            else:
                arr[k] = right_arr[j]
                j += 1
            k += 1
        while i < len(left_arr):
            arr[k] = left_arr[i]
            i += 1
            k += 1
        while j < len(right_arr):
            arr[k] = right_arr[j]
            j += 1
            k += 1
```

Figure 7 Merge_Sort Implementation

Results:

```
Merge Sort:
   Input Size  Merge Sort
0         100    0.000070
1         500    0.000382
2        1000    0.000800
3        5000    0.005188
4       10000    0.011283
5       50000    0.067516
6      100000    0.147693
```

Figure 8 Merge_Sort Implementation

Graph Representation:

Figure 9 Merge_Sort Graph Representation

Heap_Sort:

Divide and Conquer algorithm:

- Heap Sort employs a divide-and-conquer strategy to efficiently sort arrays. The algorithm divides the input array into a heap and a sorted region. It repeatedly extracts the maximum (or minimum) element from the heap and places it at the end of the sorted region. This process is iterated until the entire array is sorted.

Running time:

- Heap Sort has an optimal running time of O(n * log(n)), making it highly efficient for comparison-based algorithms. This complexity ensures fast sorting even for large datasets.

Optimal Running Time for Comparison-Based Algorithms:

- Heap Sort's time complexity of O(n * log(n)) is optimal for comparison-based algorithms, ensuring efficient sorting performance across various input distributions.

Algorithm Description:

1. **Build Max Heap:**
    a. Rearrange elements to form a max heap, ensuring each parent node is greater than or equal to its children.
    b. Iterate from the last non-leaf node to the root, heapifying each subtree.
2. **Heapify:**
    a. Swap the root (maximum element) with the last element.
    b. Restore heap property by heapifying the subtree rooted at the new root.
3. **Extract Maximum:**
    a. Extract the maximum element (root) and place it at the end of the sorted region.
    b. Reduce the heap size by one.
4. **Repeat:**
    a. Repeat heapifying and extracting until the heap is empty.

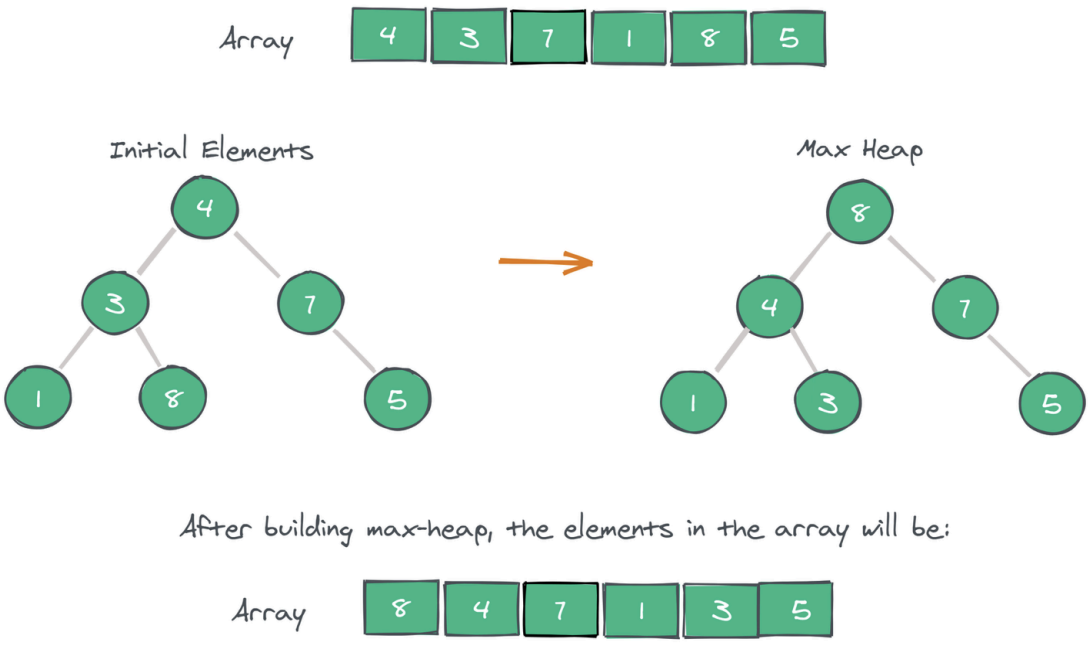

Figure 10 Heap_Sort Algorithm

Implementation:

```python
def heap_sort(arr):
    # Heapify subtree rooted at index i to satisfy heap property
    Chiuliana
    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)

    n = len(arr)

    # Build max-heap from input array
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract max element and maintain heap property
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

    return arr
```

Figure 11 Heap_Sort Implementation

Results:

Heap Sort:

	Input Size	Heap Sort
0	100	0.000075
1	500	0.000481
2	1000	0.001093
3	5000	0.007115
4	10000	0.015787
5	50000	0.095032
6	100000	0.207280

Figure 12 Heap_Sort Implementation

Graph Representation:

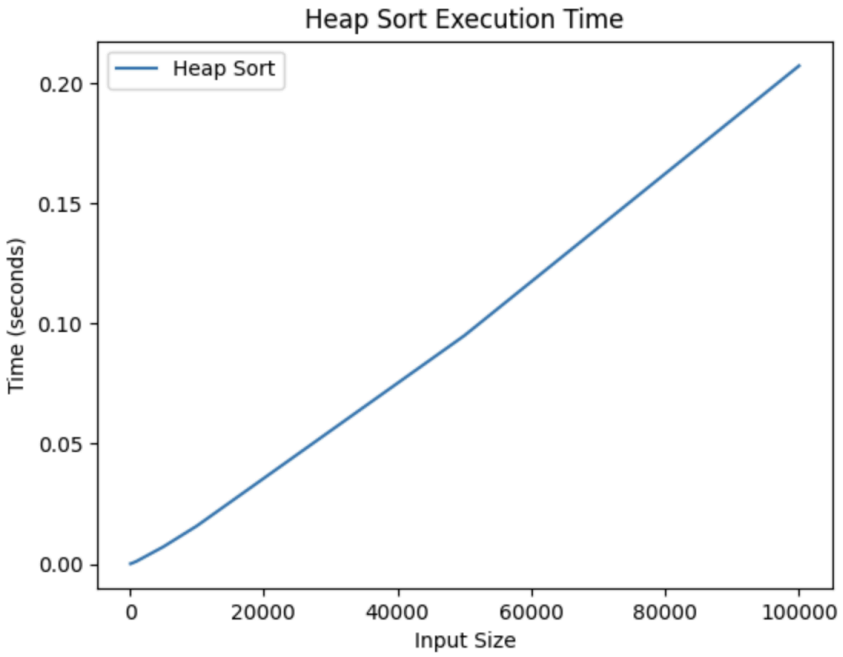

Figure 13 Heap_Sort Graph Representation

Insertion_Sort:

Insertion Sort is a simple comparison-based sorting algorithm that iteratively builds the final sorted array by gradually inserting each unsorted element into its correct position within the sorted region of the array.

Running time:

- Insertion Sort has a time complexity of O(n^2) in the worst-case scenario, where n is the number of elements in the array. This occurs when the array is in reverse order, and each element must be compared and shifted to the beginning of the array.
- However, in the best-case scenario, when the array is already sorted, Insertion Sort has a time complexity of O(n), as each element only needs to be compared to the preceding element to determine its correct position.
- Insertion Sort's average-case time complexity is also O(n^2), making it less efficient for large datasets compared to more advanced algorithms like Merge Sort and Quick Sort. Despite this, it performs well on small datasets and is efficient for nearly sorted arrays. Additionally, it operates in situ, requiring only constant additional memory.

Algorithm Description:

1. **Initialization:**
    a. Begin with the second element of the array. The first element is considered as a sorted region of size one.
2. **Insertion Process:**
    a. Iterate through the unsorted portion of the array.
    b. Compare the current element with the elements in the sorted region, moving from right to left.
    c. Shift elements in the sorted region to the right until finding the correct position for the current element.
    d. Insert the current element into its correct position within the sorted region.
3. **Repeat:**
    a. Repeat the insertion process for each remaining unsorted element until the entire array is sorted.
4. **Termination:**
    a. When all elements have been inserted into the correct positions, the array is fully sorted.

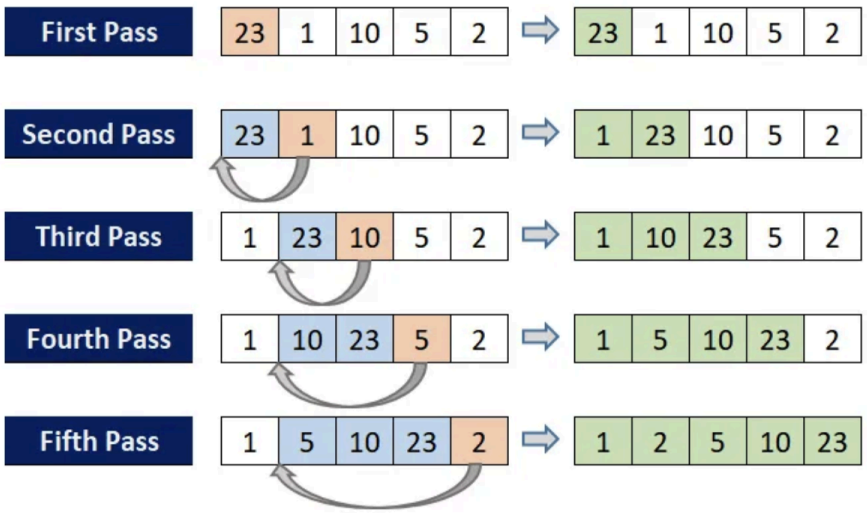

Figure 14 Insertion_Sort Algorithm

Implementation:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i
        while arr[j - 1] > arr[j] and j > 0:
            arr[j - 1], arr[j] = arr[j], arr[j - 1]  # swap
            j -= 1
```

Figure 15 Insertion_Sort Implementation

Results:

Insertion Sort:

	Input Size	Insertion Sort
0	100	0.000132
1	500	0.003747
2	1000	0.016787
3	5000	0.450338
4	10000	1.870892
5	50000	46.487500
6	100000	186.315786

Figure 16 Insertion_Sort Implementation

Graph Representation:

Figure 17 Insertion_Sort Graph Representation

CONCLUSION

When comparing all four sorting algorithms—Quick Sort, Merge Sort, Heap Sort, and Insertion Sort—based on their time complexity and empirical results, the following conclusions can be drawn regarding their efficiency:

Quick Sort:

- Time Complexity: O(n log n) on average, O(n^2) in the worst-case scenario.
- Efficiency: Quick Sort is highly efficient for large datasets with an average-case time complexity of O(n log n). However, its worst-case time complexity of O(n^2) can be a drawback for certain input distributions.
- Performance: Quick Sort often outperforms the other algorithms in practice due to its efficient partitioning strategy and low constant factors.

Merge Sort:

- Time Complexity: O(n log n) in all cases.
- Efficiency: Merge Sort consistently demonstrates efficient performance with a time complexity of O(n log n) in all cases. However, its space complexity of O(n) may limit its applicability for extremely large datasets.
- Performance: Merge Sort is stable and provides reliable performance across various input distributions, making it a popular choice for sorting large datasets.

Heap Sort:

- Time Complexity: O(n log n) in all cases.
- Efficiency: Heap Sort exhibits efficient performance with a time complexity of O(n log n). It operates in situ and requires only a constant amount of additional memory, making it space-efficient.
- Performance: While not as widely used as Quick Sort or Merge Sort, Heap Sort offers a good balance of efficiency and stability, making it suitable for certain applications.

Insertion Sort:

- Time Complexity: O(n^2) in the worst-case scenario, O(n) in the best-case scenario.
- Efficiency: Insertion Sort has a time complexity of O(n^2) in the worst-case scenario, making it less efficient for large datasets compared to the other algorithms. However, it can be efficient for small datasets or nearly sorted arrays.
- Performance: Insertion Sort's simplicity and low memory usage make it suitable for small datasets or situations where simplicity is prioritized over efficiency.

In summary, Quick Sort and Merge Sort are generally preferred for sorting large datasets due to their average-case time complexity of O(n log n). Heap Sort offers efficient performance and space efficiency, while Insertion Sort is suitable for small datasets or nearly sorted arrays. The choice of algorithm depends on the specific requirements of the application, including input size, stability, memory constraints, and ease of implementation.

Results:

	Input Size	Quick Sort	Heap Sort	Merge Sort	Insertion Sort
0	100	0.000049	0.000075	0.000070	0.000132
1	500	0.000254	0.000481	0.000382	0.003747
2	1000	0.000573	0.001093	0.000800	0.016787
3	5000	0.003468	0.007115	0.005188	0.450338
4	10000	0.007123	0.015787	0.011283	1.870892
5	50000	0.040716	0.095032	0.067516	46.487500
6	100000	0.083704	0.207280	0.147693	186.315786

Figure 18 Results of Sorting Algorithms

Graph Representation:

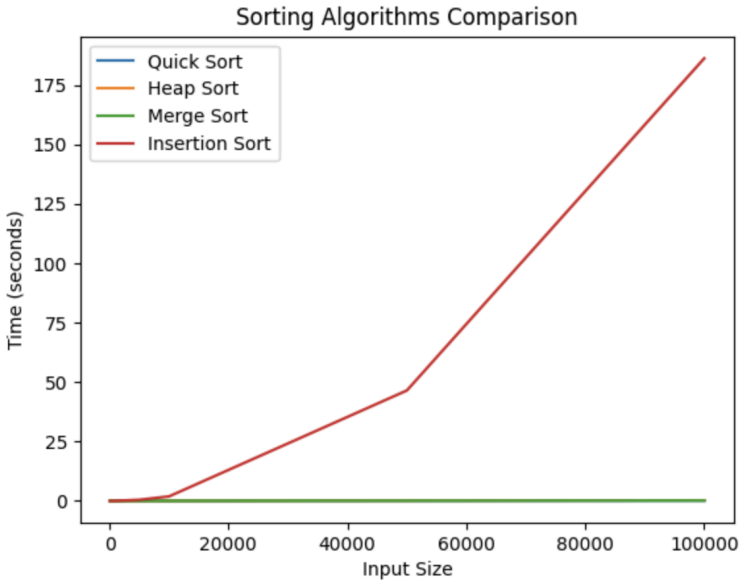

Figure 19 Sorting Algorithms Graph Representation

References

https://github.com/Chiuliana/Algorithms_Analysis_Laboratory_Works