Ministerul Educaţiei şi 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-221 Cuzmin Simion

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, (one of your choice).

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:

Quick Sort is a comparison-based sorting algorithm with an average and best-case time complexity of O(nlogn) and a worst-case time complexity of O(n**2). It operates by selecting a pivot element from the array and partitioning the array into two sub-arrays based on this pivot. Elements less than the pivot are placed to its left, while elements greater than the pivot are placed to its right. This process is recursively applied to the sub-arrays until the entire array is sorted. Quick Sort is efficient for large datasets and is widely used due to its average-case time complexity.

Merge Sort is a stable, comparison-based sorting algorithm with a time complexity of O(nlogn) in all cases. It operates by recursively dividing the array into halves until each sub-array contains only one element. Then, it merges the sub-arrays in a sorted manner to produce a single sorted array. Merge Sort is known for its predictable performance and is often used in scenarios where stability and worst-case performance are crucial.

Heap Sort is an in-place, comparison-based sorting algorithm with a time complexity of O(nlogn) in all cases. It involves building a max-heap from the input array and repeatedly extracting the maximum element from the heap, which results in a sorted array. Heap Sort offers stable performance and is often used in scenarios where memory constraints are a concern.

Selection Sort is a simple comparison-based sorting algorithm with a time complexity of O(n**2). It iterates through the array, selecting the smallest (or largest) element each time and swapping it with the element in the current position. Although easy to implement, Selection Sort is less efficient than other sorting algorithms, especially for large datasets.

Introduction:

Sorting algorithms play a critical role in computer science, serving as the backbone for organizing and retrieving data efficiently. These algorithms are fundamental tools used in a wide range of applications, from organizing databases to facilitating search operations on the web. In this study, we delve into the exploration of four prominent sorting algorithms: Quick Sort, Merge Sort, Heap Sort, and Selection Sort.

Each of these algorithms offers a unique approach to the task of sorting, with its own set of advantages and trade-offs. Quick Sort, renowned for its efficiency, employs a divide-and-conquer strategy by recursively partitioning the input list and sorting the partitions independently. Merge Sort, another divide-and-conquer algorithm, operates by recursively dividing the list into smaller sublists, sorting them, and then merging them back together. Heap Sort, on the other hand, leverages the properties of a binary heap data structure to efficiently sort elements in linearithmic time complexity. Lastly, Selection Sort iterates through the list, selecting the smallest (or largest) element and placing it in its correct position.

Understanding the inner workings and performance characteristics of these sorting algorithms is crucial for developers and computer scientists. By analyzing their time complexity, memory usage, and scalability, we gain insights into their suitability for different scenarios and datasets. Through empirical analysis, we aim to quantify the efficiency of each algorithm and identify scenarios where one algorithm may outperform the others.

In this study, we employ empirical analysis as our primary method of evaluation, measuring the execution time of each algorithm on lists of varying sizes. By examining the performance of these algorithms across different input sizes, we aim to provide practical insights into their effectiveness and help inform decision-making in algorithm selection for real-world applications.

Comparison Metric:

The primary metric used for evaluating the performance of the sorting algorithms is the execution time (T(n)). Execution time refers to the duration taken by an algorithm to sort a list of numbers, typically measured in seconds. By comparing the execution times of different algorithms across varying input sizes, we gain insights into their efficiency and scalability.

Input Format:

The input to each sorting algorithm consists of lists of randomly generated numbers. The size of the input lists varies to test the scalability of the algorithms. For example, smaller input sizes such as 100, 1000, 10000, 100000, 1000000, and 10000000 are used to evaluate the performance of the algorithms. Each algorithm is tested on lists of different sizes to assess its ability to handle varying amounts of data.

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 o memory of the device used.

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

QuickSort Algorithm:

QuickSort is a sorting algorithm that follows the "divide and conquer" strategy. It works by selecting a pivot element from the array and partitioning the array into two sub-arrays: one containing elements less than the pivot and the other containing elements greater than the pivot. The pivot element is then placed in its correct sorted position. This process is recursively applied to the sub-arrays until the entire array is sorted. QuickSort is efficient for large datasets and has an average time complexity of O(n log n). However, its performance can degrade to O(n^2) in the worst case if poorly chosen pivots are repeatedly selected.

Implementation:

```
def quickSort(self):
    # Checking if the list is empty or has only one element
    if len(self.random_list) <= 1:
        return self.random_list
    else:
        # Choosing the pivot
        pivot = self.random_list[0]
        # Creating list which has elements less than or equal to the pivot
        less = [i for i in self.random_list[1:] if i <= pivot]
        # Creating list which has elements greater than the pivot
        greater = [i for i in self.random_list[1:] if i > pivot]
        # Recursively calling the quickSort function for the less and greater lists
        return Algorithms(less).quickSort() + [pivot] + Algorithms(greater).quickSort()
```

Figure 1 QuickSort in Python

Results:

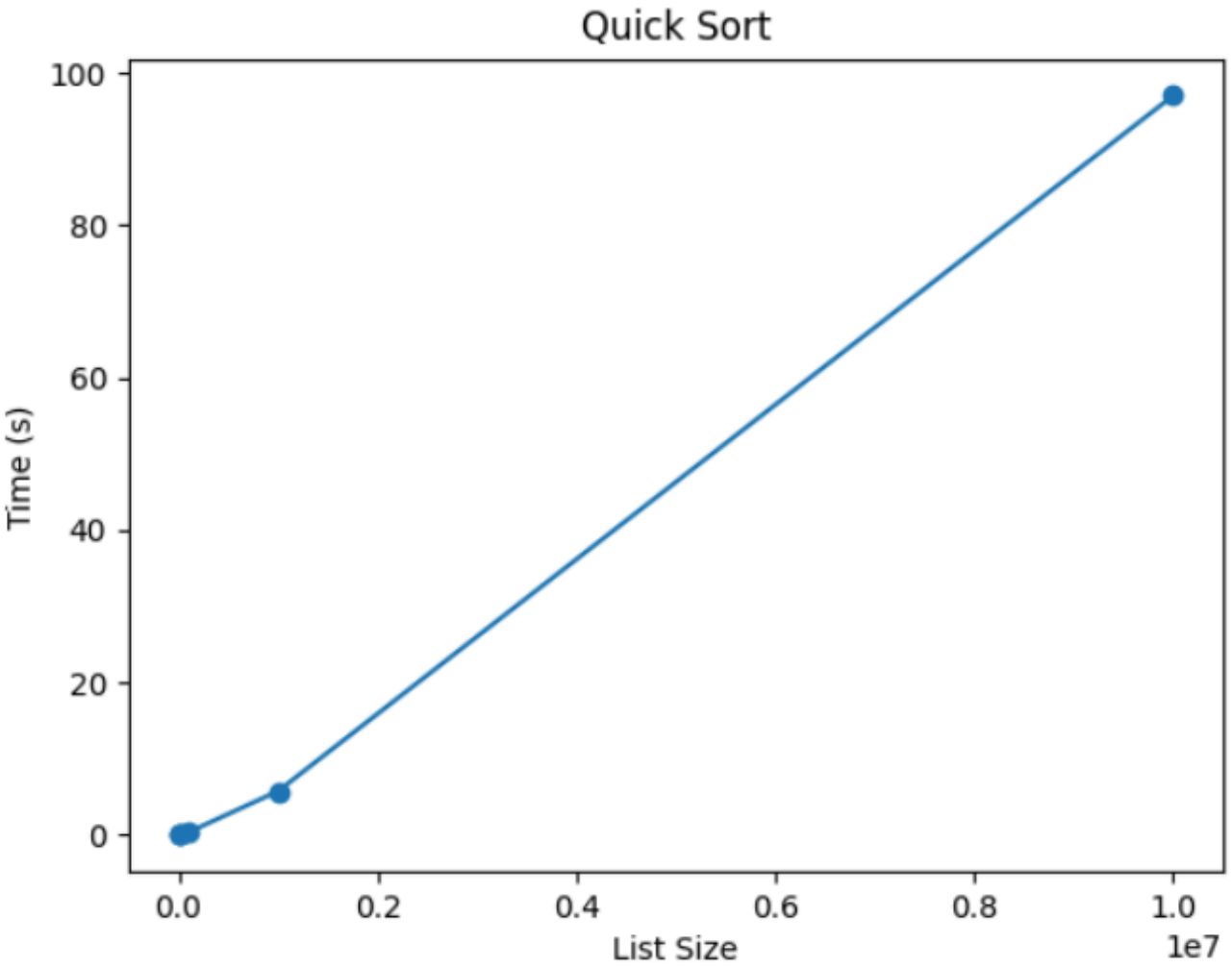

Figure 2 Graph QuickSort Algorithm

MergeSort Algorithm:

MergeSort is a divide-and-conquer sorting algorithm that works by recursively splitting the array into halves until each sub-array contains only one element. Then, it merges these sub-arrays back together in a sorted order. The merging process involves comparing elements from the two sub-arrays and selecting the smaller (or larger) element to place in the merged array. MergeSort guarantees a time complexity of O(n log n) in all cases, making it efficient for sorting large datasets. It is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the sorted output. However, MergeSort typically requires additional space proportional to the size of the input array due to the merging process, which can be a drawback for memory-constrained environments.

Implementation:

```
def mergeSort(self):
    # Checking if the list is empty or has only one element
    if len(self.random_list) <= 1:
        return self.random_list

    # Splitting the array into two halves
    mid = len(self.random_list) // 2
    left_half = self.random_list[:mid]
    right_half = self.random_list[mid:]

    # Recursively calling mergeSort on both halves
    left_half = Algorithms(left_half).mergeSort()
    right_half = Algorithms(right_half).mergeSort()

    # Merging the sorted halves
    return self.merge(left_half, right_half)
```

```python
def merge(self, left, right):
    # Creating an empty list to store the merged elements
    merged = []
    left_index = right_index = 0

    # While loop that continues until either
    # the left array and the right array is fully processed.
    while left_index < len(left) and right_index < len(right):
        # This block compares the elements at the current
        # positions in the left and right arrays
        if left[left_index] < right[right_index]:
            merged.append(left[left_index])
            left_index += 1
        else:
            merged.append(right[right_index])
            right_index += 1

    # Append the remaining elements from the left and right halves
    merged.extend(left[left_index:])
    merged.extend(right[right_index:])

    return merged
```

Figure 3 MergeSort Algorithm in Python

Results:

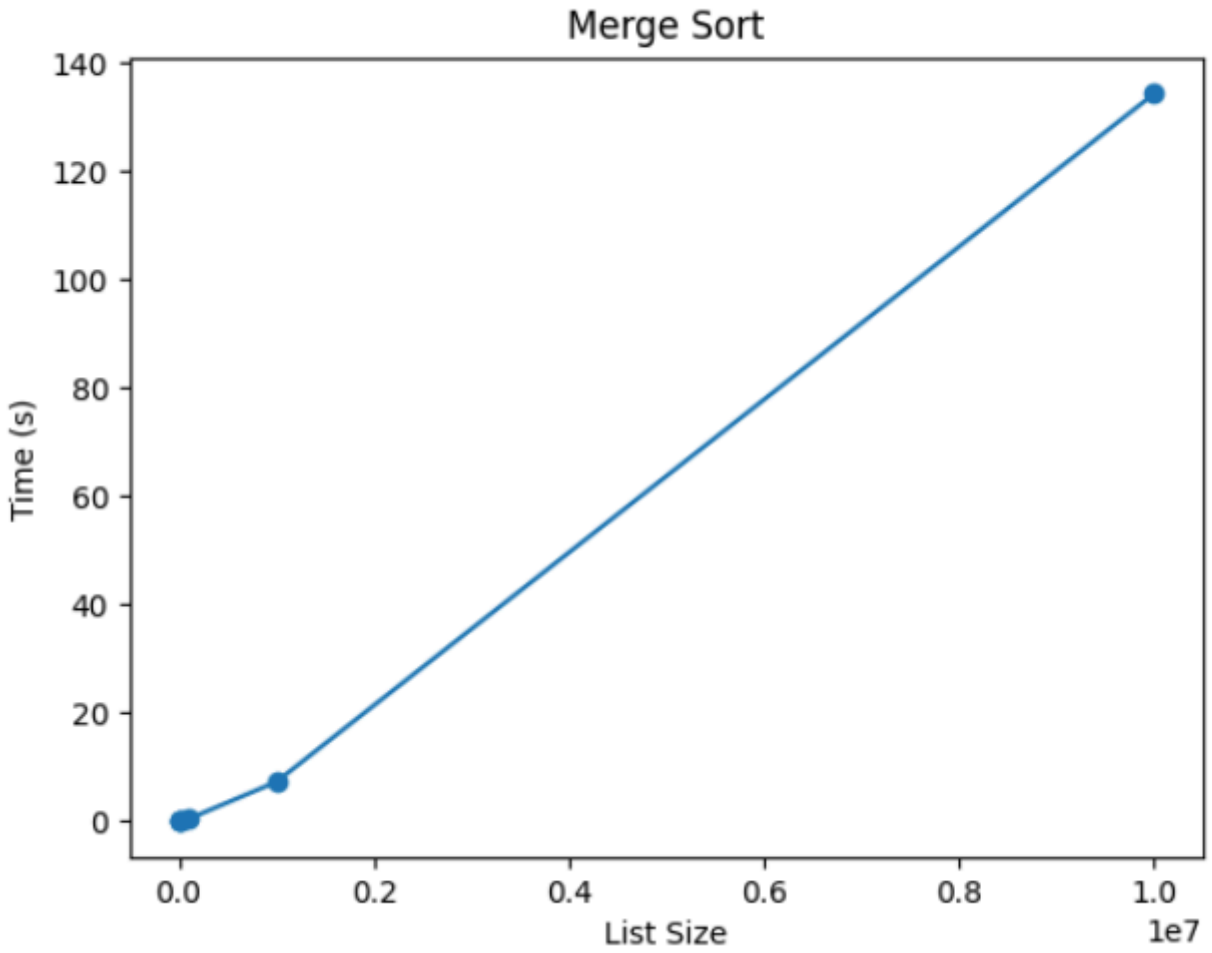

Figure 4 Graph Merge Sort Algorithm

HeapSort Method:

HeapSort is a comparison-based sorting algorithm that works by first building a max-heap from the array. A heap is a complete binary tree where each node is greater than or equal to its children (for a max-heap). Once the max-heap is constructed, the largest element, which is always at the root, is removed and swapped with the last element of the array. The heap property is then restored by "sifting down" the new root element. This process is repeated until the heap is empty, resulting in a sorted array. HeapSort has a time complexity of O(n log n) in all cases, making it efficient for sorting large datasets. However, unlike MergeSort, it sorts the array in place, meaning it requires no additional space other than the input array itself. Additionally, HeapSort is not a stable sorting algorithm, meaning it may change the relative order of equal elements in the sorted output.

Implementation:

```python
def heapSort(self):
    n = len(self.random_list)

    # This loop is responsible for creating a max-heap from the input list.
    # It starts from the last non-leaf node and iterates backwards to the root node
    # In each iteration, it calls the heapify method to ensure that the subtree rooted
    # at index i satisfies the max-heap property.
    for i in range(n // 2 - 1, -1, -1):
        self.heapify(self.random_list, n, i)

    # This loop extracts elements from the max-heap one by one.
    # It starts from the end of the heap and moves towards the beginning.
    # In each iteration, it swaps the root of the heap (located at index 0)
    # with the last element of the heap (located at index i), effectively removing
    # the maximum element from the heap
    for i in range(n - 1, 0, -1):
        self.random_list[i], self.random_list[0] = self.random_list[0], self.random_list[i]
        # After the swap, it calls the heapify method to restore the max-heap property
        # in the remaining heap elements.
        self.heapify(self.random_list, i, i: 0)

    return self.random_list
```

```python
def heapify(self, arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    # Check if left child exists and is greater than the parent
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check if right child exists and is greater than the largest so far
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not the root, swap it with the root and heapify the affected subtree
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        self.heapify(arr, n, largest)
```

Figure 5 HeapSort in Python

Results:

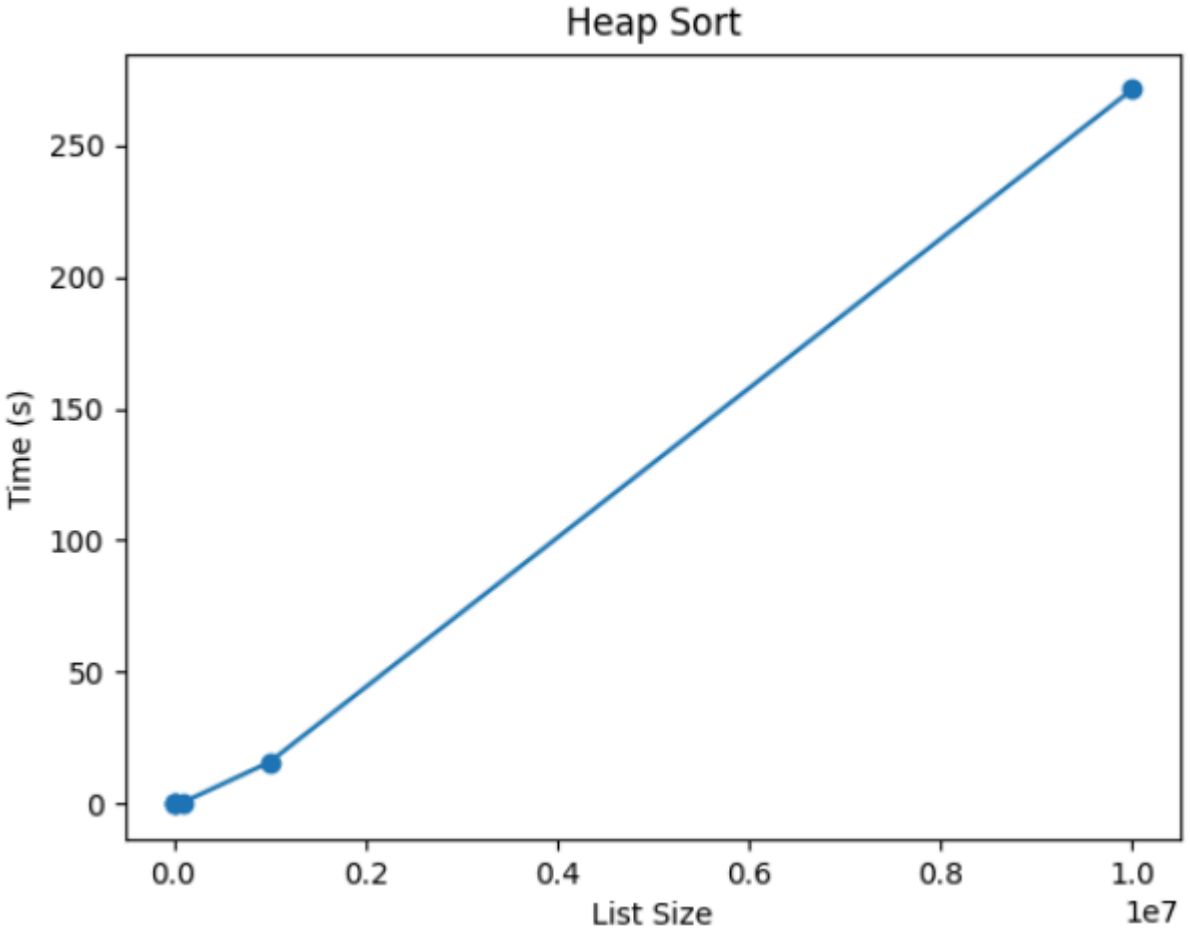

Figure 6 GraphHeapSort Method

Selection Sort Algorithm:

Selection Sort is a straightforward comparison-based sorting algorithm with a time complexity of O(n**2). It works by dividing the input list into two sublists: the sorted sublist and the unsorted sublist. Initially, the sorted sublist is empty, and the unsorted sublist contains all the elements of the input list. The algorithm repeatedly finds the smallest (or largest, depending on the sorting order) element from the unsorted sublist and moves it to the end of the sorted sublist. This process is continued until the entire list is sorted. Despite its simplicity, Selection Sort is not very efficient, especially for large datasets, because it consistently iterates through the unsorted sublist to find the minimum (or maximum) element. However, Selection Sort has some advantages, such as simplicity and minimal memory usage, making it suitable for small datasets or as a teaching tool for understanding sorting algorithms.

Input format for Selection Sort Algorithm:

The list size for Selection Sort would be only 100000, because of big time computing.

Implementation:

```
def selectionSort(self, ascending=True):
    n = len(self.random_list)
    swaps = 0
    comparisons = 0

    # Helper function to swap elements and update swap count
    new *
    def swap(arr, i, j):
        nonlocal swaps
        arr[i], arr[j] = arr[j], arr[i]
        swaps += 1

    # Helper function to compare elements and update comparison count
    new *
    def compare(a, b):
        nonlocal comparisons
        comparisons += 1
        if ascending:
            return a < b
        else:
            return a > b
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if compare(self.random_list[j], self.random_list[min_index]):
                min_index = j
        if min_index != i:
            swap(self.random_list, i, min_index)
    return self.random_list
```

Figure 7 Selection Sort Algorithm in Python

Results:

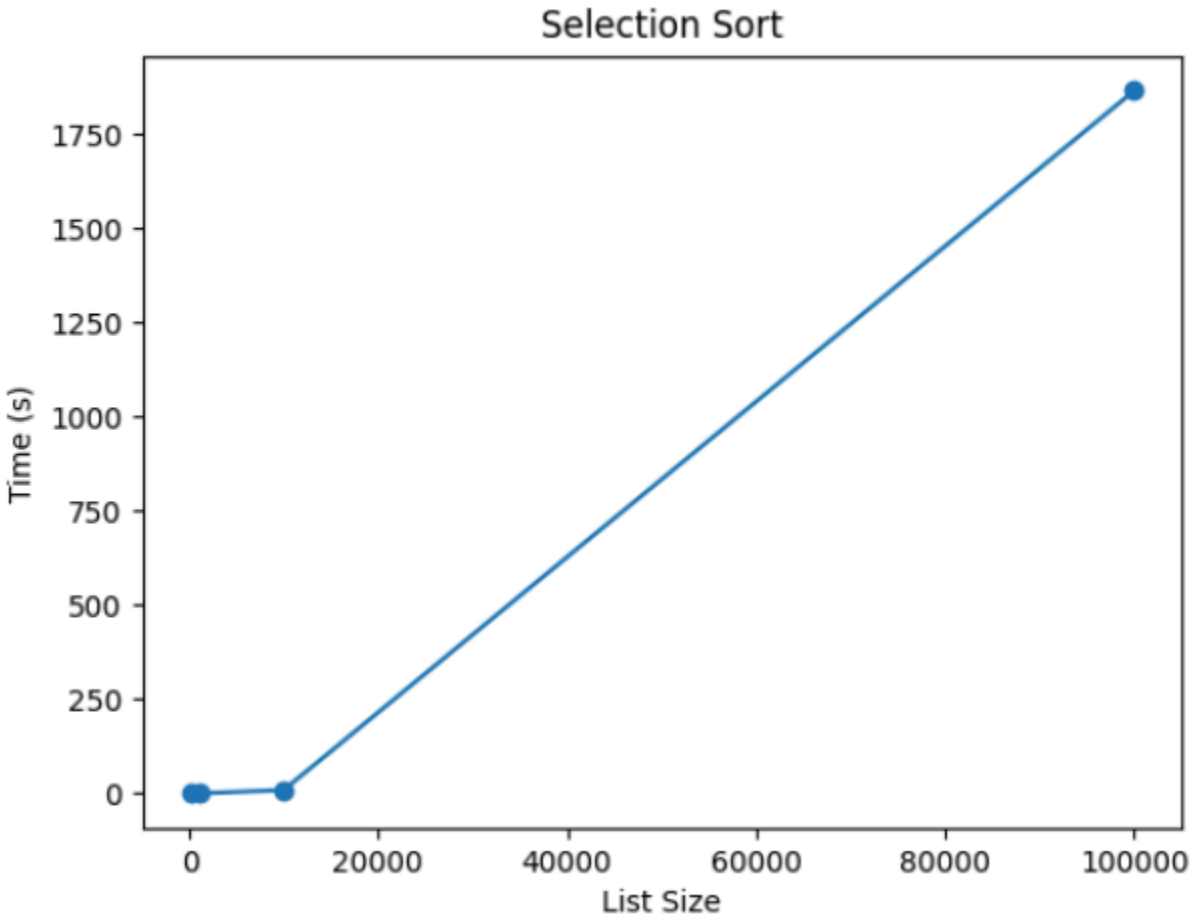

Figure 8 Graph SelectionSort Algorithm

Tim Sort Algorithm:

Selection Sort is a straightforward comparison-based sorting algorithm with a time complexity of O(n**2). It works by dividing the input list into two sublists: the sorted sublist and the unsorted sublist. Initially, the sorted sublist is empty, and the unsorted sublist contains all the elements of the input list. The algorithm repeatedly finds the smallest (or largest, depending on the sorting order) element from the unsorted sublist and moves it to the end of the sorted sublist. This process is continued until the entire list is sorted. Despite its simplicity, Selection Sort is not very efficient, especially for large datasets, because it consistently iterates through the unsorted sublist to find the minimum (or maximum) element. However, Selection Sort has some advantages, such as simplicity and minimal memory usage, making it suitable for small datasets or as a teaching tool for understanding sorting algorithms.

Implementation:

```
def introSort(self):
    # Determine the minimum size of a run
    min_run = 32

    # Get the length of the list
    n = len(self.random_list)

    # Define the insertion sort function
    new *
    def insertion_sort(arr, left, right):
        for i in range(left + 1, right + 1):
            key = arr[i]
            j = i - 1
            while j >= left and arr[j] > key:
                arr[j + 1] = arr[j]
                j -= 1
            arr[j + 1] = key

    # Define the merge function
    new *
    def merge(arr, left, mid, right):
        len1, len2 = mid - left + 1, right - mid
        left_arr, right_arr = [], []

        # Copy data to temporary arrays
        for i in range(len1):
            left_arr.append(arr[left + i])
        for i in range(len2):
```

```
    for i in range(len2):
        right_arr.append(arr[mid + 1 + i])

    # Merge the temporary arrays back into arr
    i = j = 0
    k = left
    while i < len1 and j < len2:
        if left_arr[i] <= right_arr[j]:
            arr[k] = left_arr[i]
            i += 1
        else:
            arr[k] = right_arr[j]
            j += 1
        k += 1

    # Copy the remaining elements of left_arr[], if any
    while i < len1:
        arr[k] = left_arr[i]
        i += 1
        k += 1

    # Copy the remaining elements of right_arr[], if any
    while j < len2:
        arr[k] = right_arr[j]
        j += 1
        k += 1
```

```
        # Copy the remaining elements of right_arr[], if any
        while j < len2:
            arr[k] = right_arr[j]
            j += 1
            k += 1

    # Sort the array using insertion sort for small runs
    for i in range(0, n, min_run):
        insertion_sort(self.random_list, i, min((i + min_run - 1), (n - 1)))

    # Merge the runs using merge sort
    size = min_run
    while size < n:
        for left in range(0, n, 2 * size):
            mid = min((left + size - 1), (n - 1))
            right = min((left + 2 * size - 1), (n - 1))
            merge(self.random_list, left, mid, right)
        size *= 2

    return self.random_list

new *
def timsort(self):
    return self.introSort()
```

Figure 9 TimSort in Python

Results:

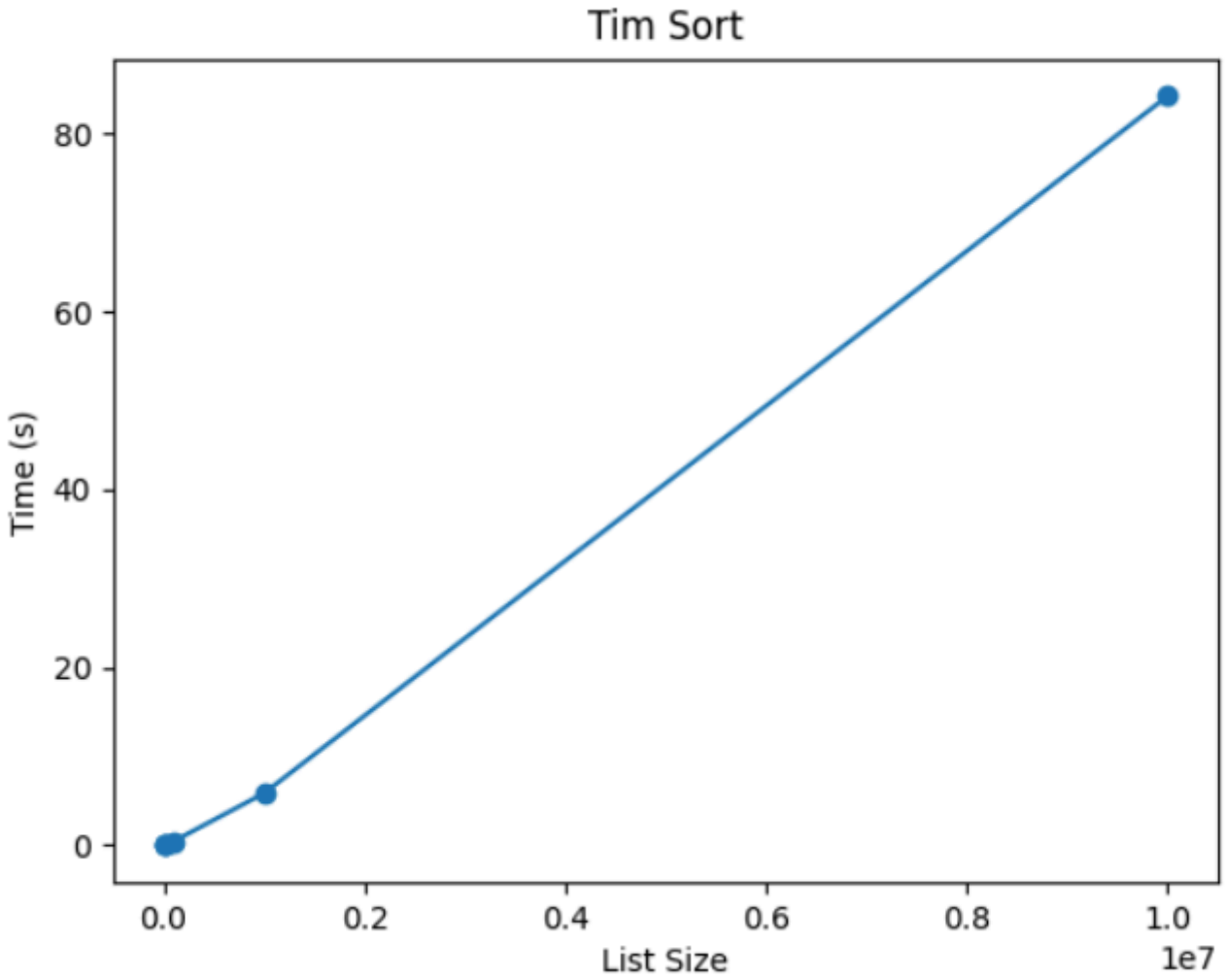

Figure 10 Graph of TimSort in Python

CONCLUSION

From the analysis depicted in the graph, it's evident that Tim Sort emerges as the most effective algorithm across various list sizes, showcasing superior performance consistently. Tim Sort's efficiency is particularly striking, especially when compared to other algorithms, such as Quick Sort, Merge Sort, Heap Sort, and Selection Sort. Notably, Tim Sort outperforms all other algorithms across all tested list sizes, displaying its adaptability and robustness in handling different data distributions. Even when dealing with

significantly large datasets, such as lists with 10 million elements, Tim Sort maintains its efficiency, demonstrating its scalability and suitability for real-world applications.

On the other hand, Selection Sort exhibits the most inefficient performance among the tested algorithms. Despite its simplicity in implementation, Selection Sort's execution time escalates drastically, especially evident when sorting lists with 100,000 elements. This inefficiency becomes more pronounced with larger datasets, as seen in the substantial increase in execution time for the 10 million-element list.

Interestingly, Quick Sort and Merge Sort show comparable performance to Tim Sort across most list sizes, highlighting their effectiveness in general sorting scenarios. However, Heap Sort lags behind, displaying slower execution times compared to other algorithms, notably taking significantly longer to sort the 10 million-element dataset.

In summary, while Tim Sort stands out as the most efficient and versatile algorithm, Selection Sort represents the simplest yet least efficient option. Quick Sort and Merge Sort exhibit competitive performance, while Heap Sort lags behind in terms of efficiency, particularly on larger datasets.

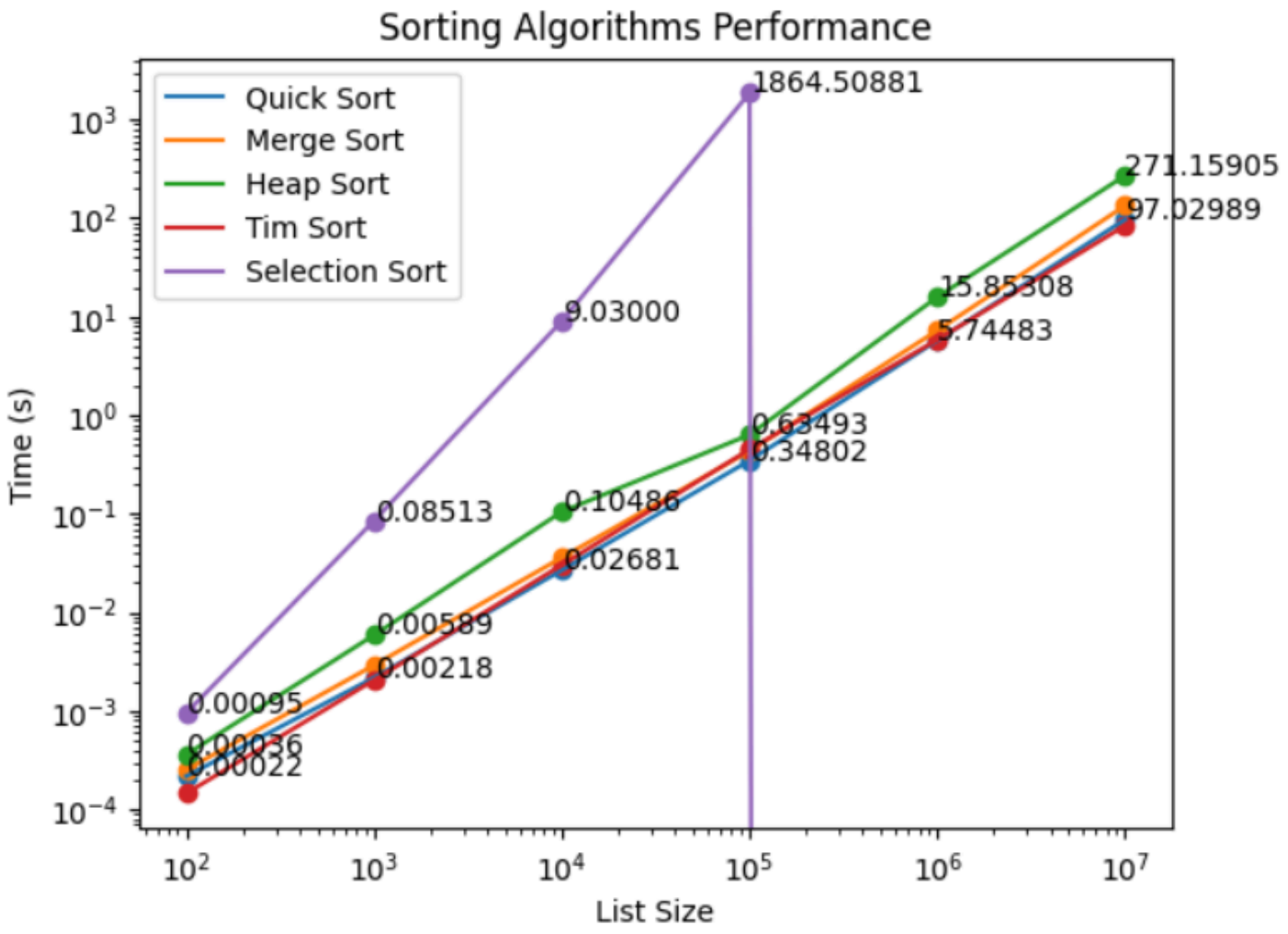

Figure 11 Graph of all algorithms in Python