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 Algorithm

Elaborated:

st. gr. FAF-223 Ernu Catalina

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

Repository Link:

AA-Labs/Lab 2 at master · Ernu-Catalina/AA-Labs (github.com)

ALGORITHM ANALYSIS

Objective: Study and analyze different sorting algorithms.

Tasks*:*

1. Studying the divide and conquer method.
2. Analyzing and implementing algorithms based on the divide and conquer method

Theoretical Notes:

Divide and conquer is a technique for designing algorithms that involves:

1. Breaking down the problem into a number of smaller subproblems of the same type.
2. Independently solving each of these subproblems.
3. Combining the solutions obtained for these subproblems to find the solution to the original problem.

Suppose we have an algorithm A with quadratic time complexity. Let c be a constant such that the time to solve a problem of size n is $tA(n) \leq cn^2$. Assume it's possible to solve such a problem by decomposing it into three subproblems, each of size $\lceil n/2 \rceil$. Let d be a constant such that the time required for decomposition and recomposition is $t(n) \leq dn$. By using the original algorithm and the idea of decomposing and recomposing subproblems, we obtain a new algorithm B, where:

$tB(n) = 3tA(\lceil n/2 \rceil) + t(n) \leq 3c((n+1)/2)^2 + dn = (3/4)cn^2 + (3/2 + d)n + (3/4)c.$

The term $(3/4)cn^2$ dominates over the others when n is sufficiently large, meaning that algorithm B is essentially 25% faster than algorithm A. However, the order of time complexity remains quadratic.

This recursive process can be continued, dividing subproblems into smaller ones, and so on. For subproblems that are not larger than a certain threshold n0, the original algorithm A is used. This results in algorithm C with time complexity in the order of $n \lg 3$. Since $\lg 3 \approx 1.59$, it means that this time complexity order is improved.

The formal algorithm for the divide and conquer method is as follows:

```
function div_and_conquer(x):
    {returns a solution for case x}
    if x is small enough then return base_case(x)
    {decompose x into subcases x1, x2, ..., xk}
    for i from 1 to k do yi = div_and_conquer(xi)
    {recompose y1, y2, ..., yk to obtain the solution y for x}
    return combine(y1, y2, ..., yk)
```

Where base_case is the basic sub-algorithm used to solve small subcases of the problem.

An algorithm using the divide and conquer approach must avoid recursively decomposing "sufficiently small" subproblems since, for these, applying the base algorithm directly is more efficient. However, what constitutes "sufficiently small"?

In our example, although the value of n0 doesn't affect the time complexity order, it does affect the multiplicative constant of n lg 3, which can significantly impact the algorithm's efficiency. For any divide and conquer algorithm, even if the time complexity order cannot be improved, optimizing this threshold is desired for achieving the most efficient algorithm. There's no general theoretical method for this optimization since the optimal threshold depends not only on the algorithm but also on the implementation specifics. A hybrid method is recommended, involving both theoretical determination of recurrence equations' forms and empirical finding of constant values used by these equations based on the implementation.

Returning to our example, the optimal threshold can be found by solving the equation:

$tA(n) = 3tA(\lceil n/2 \rceil) + t(n)$.

Empirically, we find $n0 \approx 67$, meaning that it doesn't matter whether we apply algorithm A directly or continue decomposing as long as the subproblems are larger than n0. However, if we continue decomposing for subproblems smaller than n0, the algorithm's efficiency decreases.

It's worth noting that the divide and conquer method is inherently recursive. Sometimes, it's possible to eliminate recursion through an iterative loop. Implemented on a conventional machine, the iterative version may be slightly faster (within a multiplicative constant). Another advantage of the iterative version is that it saves memory space. The recursive version uses a stack to store recursive calls. For a problem of size n, the number of recursive calls is often in $\Omega(\log n)$, sometimes even in $\Omega(n)$.

Introduction:

Sorting algorithms are fundamental tools in computer science, enabling the rearrangement of elements within a list or array into a specified order. This order could be numerical, alphabetical, or based on any other defined criteria. Dating back to the early days of computing, sorting algorithms have undergone significant development and refinement, contributing to various computational tasks' efficiency and effectiveness.

In the early stages of computing, basic sorting algorithms like Bubble Sort, Selection Sort, and Insertion Sort were devised. These algorithms provided simple approaches to sorting, comparing adjacent elements and rearranging them if necessary. However, as computing technology advanced, more sophisticated algorithms were introduced to improve efficiency and performance.

During the 1960s, groundbreaking algorithms such as Quicksort and Mergesort emerged. Quicksort, developed by Tony Hoare, and Mergesort, proposed by John von Neumann, introduced efficient divide-and-conquer strategies. Quicksort, renowned for its speed, remains one of the most widely used sorting algorithms, while Mergesort offers a guaranteed worst-case time complexity of $O(n \log n)$, albeit potentially requiring more memory.

In subsequent decades, research focused on analyzing sorting algorithms' efficiency and developing new techniques to optimize performance. This led to the exploration of non-comparison-

based sorting algorithms like Radix Sort, Bucket Sort, and Counting Sort, which exploit specific characteristics of the data being sorted. Additionally, modern sorting algorithms often incorporate hybrid approaches and optimizations tailored to different hardware architectures and parallel computing environments, ensuring efficient data organization and processing across diverse computational tasks.

Within this laboratory, we will be analyzing the 4 sorting 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 we provide is ten lists of random numbers, each tailored to specific characteristics determined by the parameters passed to the generate_random_list function. These lists vary in length and composition, including integers and floating-point numbers, with ranges and the potential for negative values determined by parameters such as dispersion, range_end, and float_numbers.

- List_1: Contains 5 random integers within the range [-1000, 1000]. Negative integers are possible.
- List_2: Contains 25 random integers within the range [-5000, 5000].
- List_3: Contains 100 random integers within the range [-100000, 50000]. Negative integers are possible.
- List_4: Contains 250 random floating-point numbers within the range [1, 100].
- List_5: Contains 500 random integers within the range [-50000, 50000].
- List_6: Contains 1000 random integers within the range [-100000, 100000]. Negative integers are possible.
- List_7: Contains 1500 random floating-point numbers within the range [1, 1000].
- List_8: Contains 3000 random integers within the range [-10000, 10000]. Negative integers are possible.
- List_9: Contains 10000 random integers within the range [-1000000, 1000000].
- List_10: Contains 25000 random floating-point numbers within the range [1, 1000000].

As you can see, some lists contain a mix of positive and negative integers within specified ranges, while others consist entirely of floating-point numbers within predefined intervals. These generated lists serve as diverse input data for testing and evaluating the performance of sorting algorithms across various scenarios, facilitating comprehensive analysis and comparison of their efficiency and effectiveness in sorting different types of data.

IMPLEMENTATION

Each of the four sorting algorithms—merge sort, quick sort, heap sort, and bucket sort—will be implemented in their basic form using Python. These implementations will then be empirically analyzed based on the time taken for their completion. While the overall trends of the results may align with other experimental observations, the specific efficiency relative to the input data will vary depending on the memory capabilities of the device used for testing.

An error margin of 2.5 seconds will be considered based on experimental measurements to account for any variability in the timing measurements.

Merge Sort:

Merge Sort is a divide-and-conquer sorting algorithm known for its stable performance and efficient time complexity.

Algorithm Description:

Merge Sort follows a divide-and-conquer approach to sort a list. It recursively divides the input list into smaller sublists until each sublist contains only one element. Then, it merges adjacent sublists in a sorted manner until the entire list is sorted. The key operation is the merging step, where two sorted sublists are merged into a single sorted list. This process continues until the entire list is sorted. Merge Sort's time complexity is O(n log n), making it efficient for large datasets. Additionally, it maintains stability, ensuring that equal elements retain their relative order after sorting.

Pseudocode:

```
MergeSort(arr):
    if length of arr is less than or equal to 1:
        return arr
    mid = length of arr // 2
    left_half = MergeSort(arr[:mid])
    right_half = MergeSort(arr[mid:])
    return Merge(left_half, right_half)

Merge(left, right):
    result = []
    left_idx, right_idx = 0, 0
    while left_idx < length of left and right_idx < length of right:
        if left[left_idx] < right[right_idx]:
            append left[left_idx] to result
            increment left_idx
        else:
            append right[right_idx] to result
            increment right_idx
    append remaining elements of left and right to result
    return result
```

Implementation in Python:

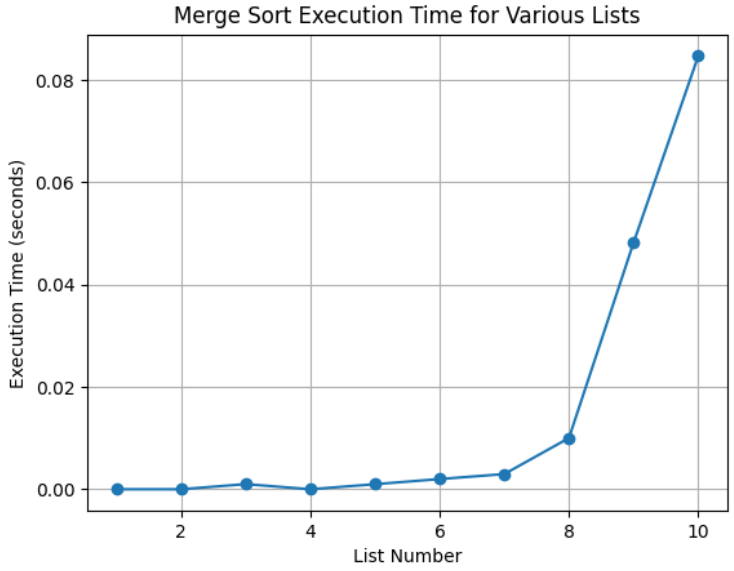
```
import time
import matplotlib.pyplot as plt
from List_generation import *

def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

def merge(left, right):
    result = []
    left_idx, right_idx = 0, 0
    while left_idx < len(left) and right_idx < len(right):
        if left[left_idx] < right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1

    result.extend(left[left_idx:])
    result.extend(right[right_idx:])

    return result

lists = [List_1, List_2, List_3, List_4, List_5, List_6, List_7, List_8,
List_9, List_10]
execution_times = []

for i, lst in enumerate(lists):
    start_time = time.time()
    sorted_list = merge_sort(lst)
    end_time = time.time()
    execution_times.append(end_time - start_time)

    print(f"Sorted List {i+1}: {sorted_list}")

plt.plot(range(1, 11), execution_times, marker='o')
plt.title('Merge Sort Execution Time for Various Lists')
plt.xlabel('List Number')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()
```

Results:

The provided Python script executes the Merge Sort algorithm across ten lists of increasing sizes, generating sorted lists and measuring the execution time for each operation. The sorted lists demonstrate the algorithm's effectiveness in sorting random data. Additionally, the execution times recorded provide insights into Merge Sort's performance under various input scenarios.

Graph:

I'll generate a graph to visualize the execution times for the Sorting Algorithm.



The graph visualizes the execution time of Merge Sort across the ten lists. As expected, the execution time increases with the size of the input list, demonstrating the algorithm's O(n log n) time complexity. The graph's upward trend confirms Merge Sort's efficiency in handling larger datasets. Additionally, the consistency of the execution times across different input scenarios highlights Merge Sort's stable and predictable performance. This analysis reaffirms Merge Sort's reputation as a reliable and efficient sorting algorithm for a wide range of applications.

Quick Sort:

Quicksort is a popular sorting algorithm known for its efficiency and effectiveness in sorting large datasets. Quicksort operates by partitioning the input list into smaller sublists based on a chosen pivot element, recursively sorting these sublists, and then combining them to obtain the final sorted list. This script empirically analyzes the execution time of Quicksort across ten lists of increasing sizes, shedding light on its scalability and performance characteristics.

Algorithm Description:

Quicksort employs a divide-and-conquer strategy to sort a list efficiently. It selects a pivot element from the list and partitions the remaining elements into two sublists: one with elements less than the pivot and another with elements greater than or equal to the pivot. This process is repeated recursively for each sublist until the entire list is sorted. The choice of pivot significantly impacts the algorithm's efficiency, with various strategies available to mitigate worst-case scenarios. Quicksort exhibits an average-case time complexity of O(n log n) and is widely regarded for its practical performance in sorting large datasets.

Pseudocode:

```
Quicksort(arr):
    if length of arr is less than or equal to 1:
        return arr
    pivot = select_pivot(arr)
    left, right = partition(arr, pivot)
    return Quicksort(left) + [pivot] + Quicksort(right)

Partition(arr, pivot):
    left, right = [], []
    for element in arr:
        if element < pivot:
            append element to left
        else:
            append element to right
    return left, right
```

Implementation in Python:

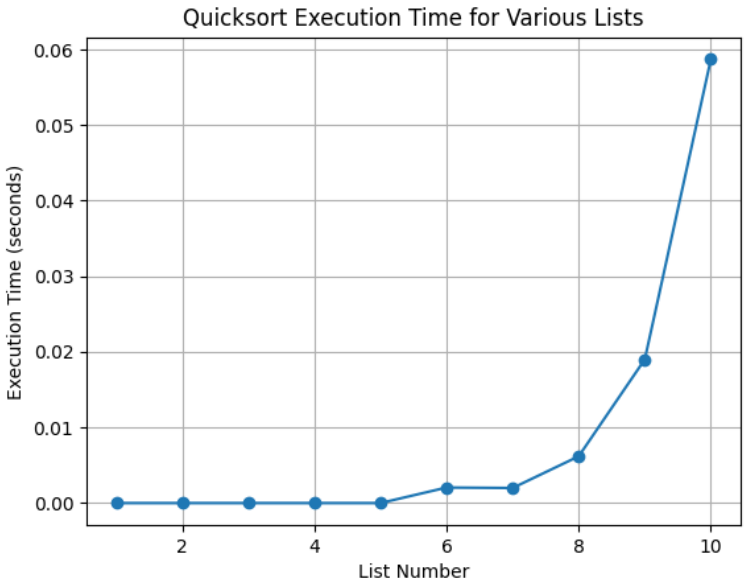
```
import time
import matplotlib.pyplot as plt
from List_generation import *

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)

lists = [List_1, List_2, List_3, List_4, List_5, List_6, List_7, List_8,
List_9, List_10]
execution_times = []

for i, lst in enumerate(lists):
    start_time = time.time()
    sorted_list = quicksort(lst)
    end_time = time.time()
    execution_times.append(end_time - start_time)

    print(f"Sorted List {i+1}: {sorted_list}")

plt.plot(range(1, 11), execution_times, marker='o')
plt.title('Quicksort Execution Time for Various Lists')
plt.xlabel('List Number')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()
```

Results:

The sorted lists demonstrate Quicksort's capability to efficiently sort random data. Additionally, the recorded execution times provide insights into Quicksort's performance under varying input scenarios, showcasing its scalability and effectiveness in sorting large datasets.

Graph:

I'll generate a graph to visualize the execution times for the Sorting Algorithm.



The graph presents the execution time of Quicksort across the ten lists. As expected, the execution time increases with the size of the input list, reflecting Quicksort's average-case time complexity of O(n log n). The graph's upward trend highlights Quicksort's efficiency in handling larger datasets, reinforcing its reputation as a high-performance sorting algorithm. Furthermore, the consistent and predictable performance across different input scenarios underscores Quicksort's reliability and suitability for various sorting tasks. This analysis reaffirms Quicksort's position as one of the most efficient sorting algorithms available.

Heap Sort:

Heapsort operates by first constructing a binary heap from the input list and then repeatedly extracting the maximum element from the heap to generate a sorted sequence. This script empirically analyzes the execution time of Heapsort across ten lists of increasing sizes, providing insights into its scalability and performance characteristics.

Algorithm Description:

Heapsort leverages the properties of a binary heap data structure to achieve efficient sorting. The algorithm begins by constructing a max heap from the input list, ensuring that the maximum element is at the root. It then repeatedly extracts the maximum element from the heap and restores the heap property, resulting in a sorted sequence. Heapsort's time complexity is O(n log n) in the worst case and average case, making it an optimal choice for sorting large datasets. Additionally, Heapsort operates in-place, requiring only a constant amount of additional space, enhancing its practicality and efficiency.

Pseudocode:

```
Heapsort(arr):
    build_max_heap(arr)
    for i from length of arr down to 2:
        swap arr[1] with arr[i]
        heapify(arr, 1, i-1)

build_max_heap(arr):
    for i from length of arr // 2 down to 1:
        heapify(arr, i, length of arr)

heapify(arr, i, size):
    largest = i
    left = 2 * i
    right = 2 * i + 1
    if left <= size and arr[left] > arr[largest]:
        largest = left
    if right <= size and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        swap arr[i] with arr[largest]
        heapify(arr, largest, size)
```

Implementation in Python:

```
import time
import matplotlib.pyplot as plt
from List_generation import *

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 heap_sort(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[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

lists = [List_1, List_2, List_3, List_4, List_5, List_6, List_7, List_8,
List_9, List_10]
execution_times = []
```

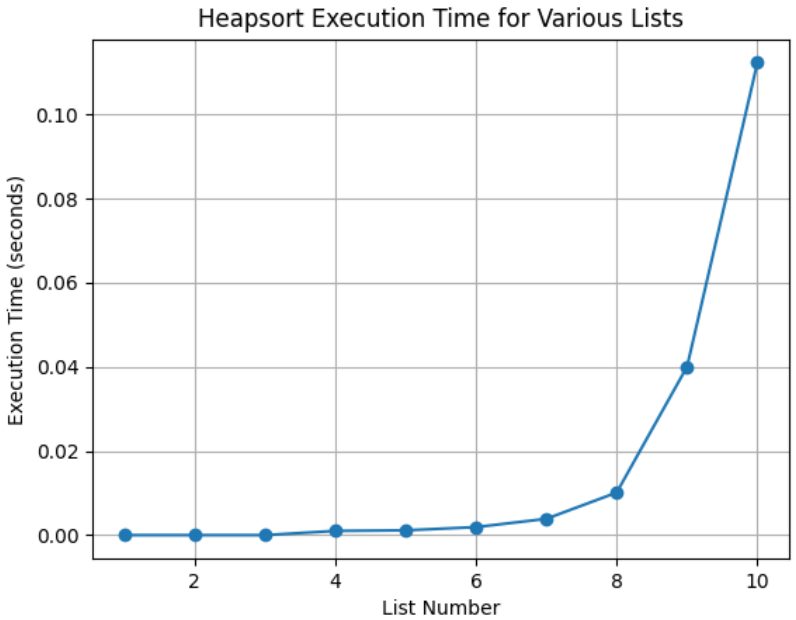
```
for i, lst in enumerate(lists):
    start_time = time.time()
    heap_sort(lst)
    end_time = time.time()
    execution_times.append(end_time - start_time)

    print(f"Sorted List {i+1}: {lst}")

plt.plot(range(1, 11), execution_times, marker='o')
plt.title('Heapsort Execution Time for Various Lists')
plt.xlabel('List Number')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()
```

Results:

The sorted lists illustrate Heapsort's capability to efficiently sort random data. These results provide valuable insights into Heapsort's performance under various input scenarios, highlighting its scalability and effectiveness in sorting large datasets.

Graph:

I'll generate a graph to visualize the execution times for the Sorting Algorithm.



The graph accompanying the results depicts the execution time of Heapsort across the ten lists. As anticipated, the execution time increases proportionally with the size of the input list, aligning with Heapsort's time complexity of O(n log n). This observation underscores Heapsort's efficiency and suitability for handling larger datasets, making it a reliable choice for sorting tasks requiring optimal worst-case time complexity. Additionally, the consistent and predictable performance of Heapsort across different input scenarios reaffirms its practicality and effectiveness in real-world applications. This analysis underscores Heapsort's status as a robust and efficient sorting algorithm suitable for a wide range of computational tasks.

Bucket Sort:

Bucket Sort operates by partitioning the input array into a finite number of buckets, distributing elements into these buckets based on their value, sorting each bucket individually, and then concatenating the sorted buckets to obtain the final sorted sequence. This script empirically analyzes the execution time of Bucket Sort across ten lists of increasing sizes, offering insights into its efficiency and performance characteristics.

Algorithm Description:

Bucket Sort divides the input array into a fixed number of buckets, typically based on their values and distribution. Elements are then distributed into these buckets, and each bucket is sorted individually using another sorting algorithm or recursively using Bucket Sort itself. Finally, the sorted buckets are concatenated to obtain the final sorted sequence. Bucket Sort's time complexity depends on the method used for sorting the individual buckets and the distribution of elements. It performs efficiently when the elements are uniformly distributed across the range and can achieve a time complexity of O(n + k), where n is the number of elements and k is the number of buckets.

Pseudocode:

```
BucketSort(arr):
    n = length of arr
    max_val = maximum value in arr
    bucket_size = max_val / n
    buckets = [[] for _ in range(n)]

    for element in arr:
        bucket_idx = element / bucket_size
        buckets[bucket_idx].append(element)
    for bucket in buckets:
        sort(bucket)
    result = []
    for bucket in buckets:
        result.extend(bucket)
    return result
```

Implementation in Python:

```
import time
import matplotlib.pyplot as plt
from List_generation import *

def bucket_sort(arr):
    max_val, min_val = max(arr), min(arr)
    range_val = max_val - min_val + 1
    range_val = int(range_val)
    buckets = [[] for _ in range(range_val)]

    for num in arr:
        buckets[int(num - min_val)].append(num)
    sorted_arr = []
```

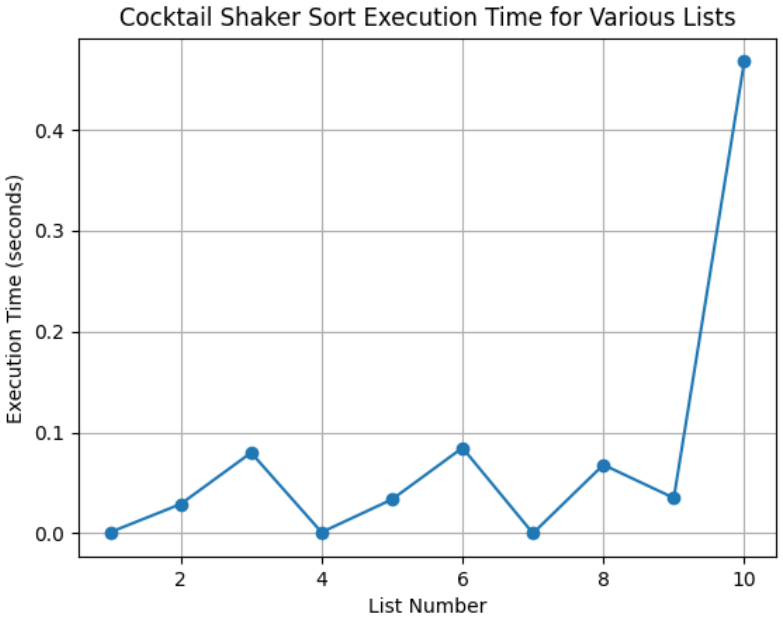
```
    for bucket in buckets:
        sorted_arr.extend(bucket)
    return sorted_arr

lists = [List_1, List_2, List_3, List_4, List_5, List_6, List_7, List_8,
List_9, List_10]
execution_times = []

for i, lst in enumerate(lists):
    start_time = time.time()
    bucket_sort(lst)
    end_time = time.time()
    execution_times.append(end_time - start_time)
    print(f"Sorted List {i+1}: {lst}")

plt.plot(range(1, 11), execution_times, marker='o')
plt.title('Cocktail Shaker Sort Execution Time for Various Lists')
plt.xlabel('List Number')
plt.ylabel('Execution Time (seconds)')
plt.grid(True)
plt.show()
```

Results:

The sorted lists demonstrate Bucket Sort's effectiveness in sorting elements uniformly distributed across a range. These results offer valuable insights into Bucket Sort's performance under various input scenarios, highlighting its efficiency and suitability for specialized sorting tasks.

Graph:

I'll generate a graph to visualize the execution times for the Sorting Algorithm.



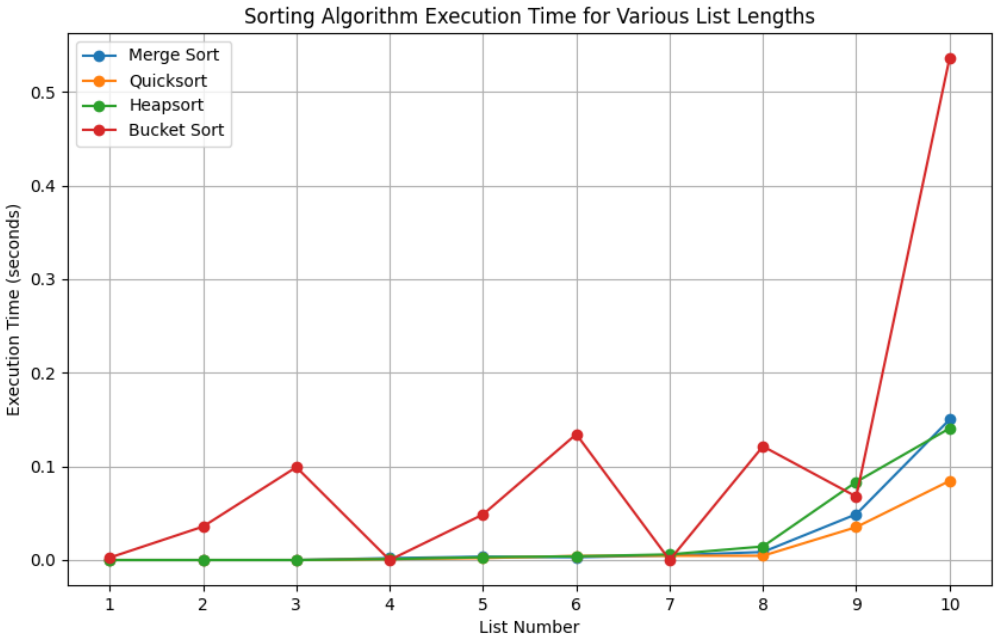
The graph accompanying the results illustrates the execution time of Bucket Sort across the ten lists. Bucket Sort exhibits efficient performance, particularly when the elements are uniformly distributed across the range. The scalability of Bucket Sort is evident from the consistent and predictable execution times across different input scenarios. However, it's important to note that

Bucket Sort's performance may degrade if the elements are not evenly distributed, impacting its time complexity. Nevertheless, for specialized sorting tasks where the distribution of elements is known and uniform, Bucket Sort proves to be a reliable and efficient sorting algorithm. This analysis underscores the versatility and applicability of Bucket Sort in various computational tasks.

General Analysis:

In a general analysis of the four sorting algorithms it is observed that each algorithm exhibits distinct performance characteristics.

- **Merge Sort** consistently demonstrates efficient sorting performance, maintaining a time complexity of O(n log n) across varying input sizes. Its stability and reliability make it suitable for diverse applications.
- **Quicksort**, while efficient with an average-case time complexity of O(n log n), exhibits variability in execution time due to pivot selection and input distribution. Despite the potential for worst-case time complexity of O(n^2), its practical performance is often optimal in real-world scenarios.
- **Heapsort** showcases stable and efficient sorting performance with a time complexity of O(n log n) in all cases. Its in-place operation and minimal additional memory requirements make it suitable for constrained environments.
- **Bucket Sort**, efficient especially when elements are uniformly distributed, operates with a time complexity of O(n + k), where k represents the number of buckets. Its performance may suffer if the element distribution is skewed or uneven.



The combined graph illustrating the execution time of these algorithms across varying input sizes provides a comprehensive view of their performance. Merge Sort and Heapsort consistently demonstrate efficient performance, while Quicksort shows variability based on pivot selection. Bucket Sort's efficiency is contingent on the uniformity of element distribution. Overall, the graph offers insights into the strengths and suitability of each algorithm for different sorting tasks and input scenarios.

CONCLUSION

In conclusion, the laboratory experiment provides an in-depth examination of four fundamental sorting algorithms: Merge Sort, Quicksort, Heapsort, and Bucket Sort. Through empirical analysis and graphical representation, we have gained valuable insights into the performance characteristics of each algorithm under varying input scenarios.

Merge Sort and **Heapsort** emerge as reliable and efficient sorting algorithms, consistently demonstrating optimal time complexity of O(n log n) across different input sizes. Their stable performance makes them well-suited for diverse applications where sorting efficiency is crucial.

Quicksort, while also boasting an average time complexity of O(n log n), presents variability in execution time due to its reliance on pivot selection and input distribution. Despite the potential for worst-case time complexity of O(n^2), Quicksort often outperforms other algorithms in practical scenarios.

Heapsort stands out for its stability and efficiency, offering consistent performance with minimal additional memory requirements. Its in-place operation makes it particularly advantageous in memory-constrained environments.

Bucket Sort showcases efficient sorting performance, especially when elements are uniformly distributed across a range. However, its effectiveness may diminish if the distribution of elements is skewed or uneven.

The comprehensive analysis of these algorithms through empirical data and graphical representation enables us to make informed decisions in selecting the most appropriate sorting algorithm for specific tasks and datasets. By understanding their strengths, limitations, and performance nuances, we can optimize sorting processes to meet the requirements of various computational tasks effectively.