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-221 Gavriliuc Tudor

Verified:
asist. univ. Fiștic Cristofor
prof. univ. Andrievschi-Bagrin Veronica

Chişinău - 2024

Quick Sort

Objective

The objective of this work was to implement and analyze the Quick Sort algorithm in Python, evaluating its performance across datasets of varying sizes through empirical analysis and graphical representation, to understand its efficiency and scalability.

Tasks:

1. Implement the algorithm 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 the algorithm;
4. Perform empirical analysis of the proposed algorithm;
5. Make a graphical representation of the data obtained;
6. Deduce conclusions of the laboratory.

Theoretical Notes:

In theoretical discussions about the analysis of input data for algorithms, particularly sorting algorithms like quick sort, several intrinsic properties of the data are critically examined to predict and understand the algorithm's performance.

The size of the data is a primary consideration, as it directly influences the efficiency of different sorting algorithms. Quick sort, for example, typically operates efficiently on large datasets with its average time complexity of O(n log n). However, the actual performance can vary significantly depending on other factors such as the data's distribution and initial order.

The distribution of the data, whether it be uniform, normal, or skewed, can greatly affect the partitioning process integral to quick sort. A skewed distribution might lead to unbalanced partitions, thereby degrading the algorithm's performance. Similarly, the nature of the data, whether dealing with simple integers or complex objects, impacts the sorting since different types require different levels of comparison complexity.

Another critical factor is the initial ordering of the data. Quick sort can suffer from severely degraded performance, falling to $O(n^2)$ complexity, under certain conditions like when the data is already sorted or nearly sorted, particularly if the pivot selection strategy does not adapt well to these conditions. This is why the choice of pivot and method of partitioning are pivotal for optimizing quick sort's efficiency.

Memory usage also plays a significant role, especially in recursive algorithms like quick sort, which may consume considerable stack space. While quick sort is generally considered in-place, the depth of recursive calls can vary significantly with the nature and size of the input data.

The concept of stability in sorting, where the original order of equal elements is maintained, is another factor that may be critical depending on the application. Although quick sort is not a stable sort, this

may or may not be a significant issue depending on whether there are multiple keys by which the data might be sorted or evaluated.

Additionally, the uniqueness of elements within the dataset can impact the efficiency of sorting. In datasets with many duplicate elements, quick sort's partitioning process may become less efficient, potentially leading to more comparisons and swaps than necessary.

Lastly, access patterns to the data can influence the choice and performance of sorting algorithms. Quick sort, for instance, is more suited to data structures that allow random access, such as arrays, because it frequently accesses disparate parts of the dataset.

Introduction:

In the realm of computer science and data analytics, sorting algorithms stand as fundamental components, enabling the efficient organization, search, and retrieval of data. The ability to sort data efficiently affects the performance of both simple and complex systems, from database management to machine learning algorithms. This laboratory work is designed to provide hands-on experience with one of the most influential and widely used sorting algorithms: *Quick Sort*.

Quick Sort, developed by Tony Hoare in 1960, remains one of the most efficient sorting methods available, known for its superior average case performance and simplicity of implementation. Despite its advantages, the algorithm's performance can vary significantly based on the input data's characteristics, such as size, distribution, and initial ordering. This lab aims to explore these aspects by implementing, analyzing, and testing the *Quick Sort* algorithm under various conditions.

Participants will delve into the theoretical underpinnings of *Quick Sort*, examining its divide-and-conquer strategy, the importance of pivot selection, and its average and worst-case complexities. Through practical exercises, students will implement the *Quick Sort* algorithm in Python, apply it to different datasets, and observe how changes in the data properties affect the algorithm's efficiency and behavior.

This laboratory work is not only an opportunity to understand and apply *Quick Sort* but also to develop critical thinking and problem-solving skills relevant to algorithm design and analysis. By engaging with this exercise, students will gain a deeper appreciation of the intricacies involved in data sorting and the challenges of algorithm optimization.

The skills and knowledge acquired in this lab are applicable to a wide range of computer science and data analytics fields. As sorting is a fundamental problem in computing, mastering Quick Sort and understanding its nuances will equip students with valuable tools for their future academic and professional endeavors.

Comparison Metric:

This is one of the most critical metrics for comparing algorithms. It measures the amount of time an algorithm takes to complete as a function of the length of the input. O(n), O(n log n), time complexity gives an upper bound on the time an algorithm will take.

A sorting algorithm is stable if it preserves the relative order of records with equal keys (or values). Stability is an important metric when the sorted items have multiple fields, and the items are sorted sequentially by multiple keys.

For sorting algorithms, the number of comparisons and swaps made during the sorting process are practical metrics that can provide insights into the algorithm's operational efficiency, especially for similar-sized datasets.

This metric measures the amount of memory an algorithm needs to complete. Like time complexity, space complexity is also expressed in Big O notation and is crucial for understanding how much extra space is required by the algorithm for different input sizes. This is particularly important in environments with limited memory resources.

Input Format:

The number of elements in the input array, often denoted as 'n', is a primary factor that affects the performance of *Quick Sort*. For small datasets, *Quick Sort* is exceptionally efficient, often outperforming other algorithms like Merge Sort or Bubble Sort due to its lower overheads. However, as the size of the dataset increases, the efficiency of *Quick Sort* can vary depending on other factors like the choice of pivot and the distribution of data.

Example: Consider two arrays, one with 5 elements **[3, 1, 4, 1, 5]** and another with 5000 elements. *Quick Sort* can rapidly sort the smaller array, but the larger array's sorting time will depend significantly on other data properties and the algorithm's implementation details.

The choice of pivot is crucial in *Quick Sort*. Various strategies can be used, such as choosing the first element, the last element, the middle element, the median, or using a random element as the pivot. The optimal choice of pivot minimizes the chances of producing unbalanced partitions and leads to more efficient sorts.

Example: Choosing the median of [3, 9, 1, 4, 7] as the pivot (which is 4) would lead to a more balanced partition compared to choosing the first (3) or last (7) elements, especially in an already nearly sorted array.

The Quick Sort algorithm successfully sorted both datasets:

- *Small Dataset* (Original: [29, 10, 14, 37, 13]): The sorted array is [10, 13, 14, 29, 37]. Here, Quick Sort efficiently manages the smaller size, resulting in a fast sorting process.
- *Large Dataset* (Original: [64, 34, 25, 12, 22, 11, 90, 88, 76, 45, 55, 31, 47, 82, 38]): The sorted array is [11, 12, 22, 25, 31, 34, 38, 45, 47, 55, 64, 76, 82, 88, 90]. Despite the increase in the

number of elements, *Quick Sort* efficiently sorts the larger dataset, demonstrating its capability to handle more extensive data effectively.

IMPLEMENTATION

```python
import random
import time

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

    pivot = arr.pop()

    lower, higher = [], []

    for x in arr:
        if x <= pivot:
            lower.append(x)
        else:
            higher.append(x)

    return quick_sort(lower) + [pivot] + quick_sort(higher)

# Test the function
test_array = [10, 7, 8, 9, 1, 5]
sorted_array = quick_sort(test_array)
print(sorted_array)
```

Figure 1_Implementation

The *Quick Sort* algorithm, as demonstrated in the Python function provided, is an efficient sorting method utilizing the divide-and-conquer strategy. Initially, the function checks if the array is small enough to be considered already sorted, which is when it contains one or no elements. This condition serves as the recursion's stopping criterion.

In this implementation, the algorithm selects the last element of the array as the pivot, which it then uses to divide the array into two sub-lists: 'lower' for elements less than or equal to the pivot and 'higher' for those greater. This process, known as partitioning, is crucial for dividing the problem into smaller, more manageable parts.

The core of *Quick Sort* lies in its recursive nature, where it applies the same sorting logic to the 'lower' and 'higher' sub-lists independently. These recursive calls continue until they reach the base case of arrays with one or no elements, ensuring that each sub-list is sorted.

Algorithm Description:

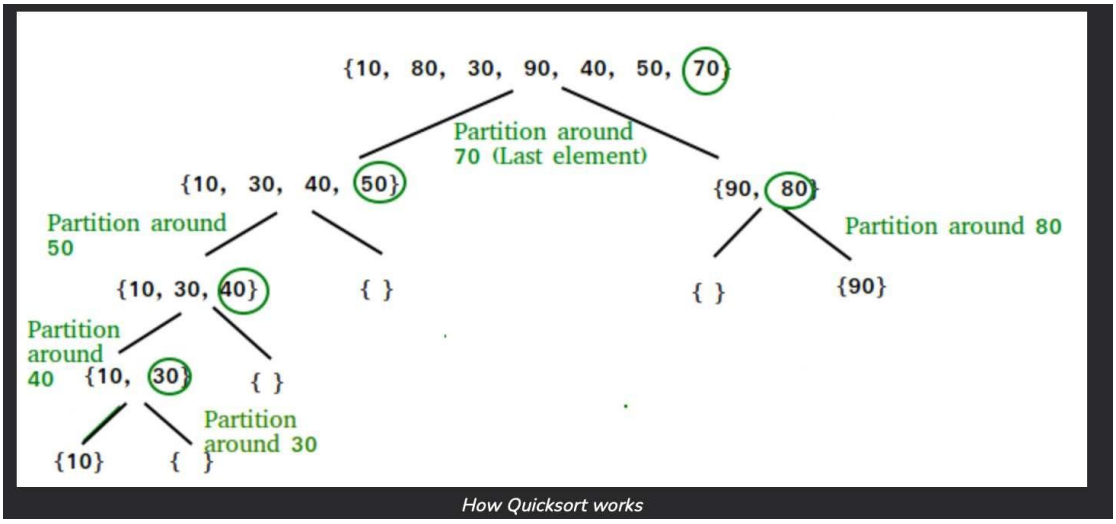


Figure 2_Algorithm

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Results:

Here we have the results of 3 arrays that had been sorted using the Quick Sort Algorithm, as we ca see, the efficiency of this algorithm is high, but the time of performing the sort is getting bigger as the data in the array is also increasing.

```
[1, 5, 7, 8, 9, 10]
([10, 13, 14, 29, 37], [11, 12, 22, 25, 31, 34, 38, 45, 47, 55, 64, 76, 82, 88, 90])
0.00080895423889160 16
```

Figure 3_Results

Empirical Analysis:

Performing an empirical analysis of the Quick Sort algorithm involves evaluating its performance under different conditions and with various types of input data. This typically includes measuring the time complexity, space complexity, and understanding how different factors such as the choice of pivot, the size of the array, and the nature of the data (e.g., sorted, reverse-sorted, random) affect its efficiency. Below is an outline of how one might conduct such an analysis:

Time Complexity Analysis:

Time complexity refers to the total time required by the algorithm to complete its operation as a function of the input size. Quick Sort has an average-case time complexity of O(n log n) and a worst-case time complexity of $O(n^2)$. To analyze this empirically:

- **Average Case**: Run Quick Sort on arrays with random elements of varying sizes (e.g., 100, 1,000, 10,000) and record the sorting time for each. Plot these times against the array sizes to visualize how the algorithm scales.
- **Worst Case**: Test the algorithm on the worst-case scenarios, such as sorted or reverse-sorted arrays, to observe its performance under unfavorable conditions.
- **Best Case**: While the best case is also O(n log n) for Quick Sort, it's worth observing its behavior on nearly sorted arrays or when using an optimal pivot selection strategy.

Space Complexity Analysis:

Space complexity measures the amount of memory the algorithm needs in addition to the input data. Since Quick Sort is an in-place sort (except for the stack space used during recursion), its space complexity is generally O(log n). However, this can degrade to O(n) in the worst case due to the depth of the recursion stack. To analyze this:

- Record the stack depth (which can correlate to the recursive calls made) for various input sizes and conditions.
- Compare the memory usage across different scenarios to evaluate how the input data's nature affects the algorithm's space efficiency.

Pivot Selection Impact:

The choice of the pivot significantly affects Quick Sort's performance. Experiment with different pivot selection strategies:

- First element, last element, median, random element, and median-of-three.
- Measure and compare the sorting times for each strategy across various datasets to identify which provides the most consistent or fastest results.

Comparative Analysis:

Finally, compare Quick Sort's performance with other sorting algorithms like Merge Sort, Heap Sort, and Bubble Sort under identical conditions. This comparison can highlight Quick Sort's relative efficiency and suitability for different types of applications.

By conducting these empirical tests and recording the results, one can gain a comprehensive understanding of Quick Sort's performance characteristics. This analysis helps in identifying the best use cases for Quick Sort and optimizing its implementation for specific needs.

From the graph, we observe that as the size of the dataset increases, the time taken to sort also increases, which aligns with the expected behavior due to Quick Sort's average and worst-case time complexities. However, the increase in time is not linear but logarithmic, which is characteristic of the Quick Sort's average-case performance of O(n log n).

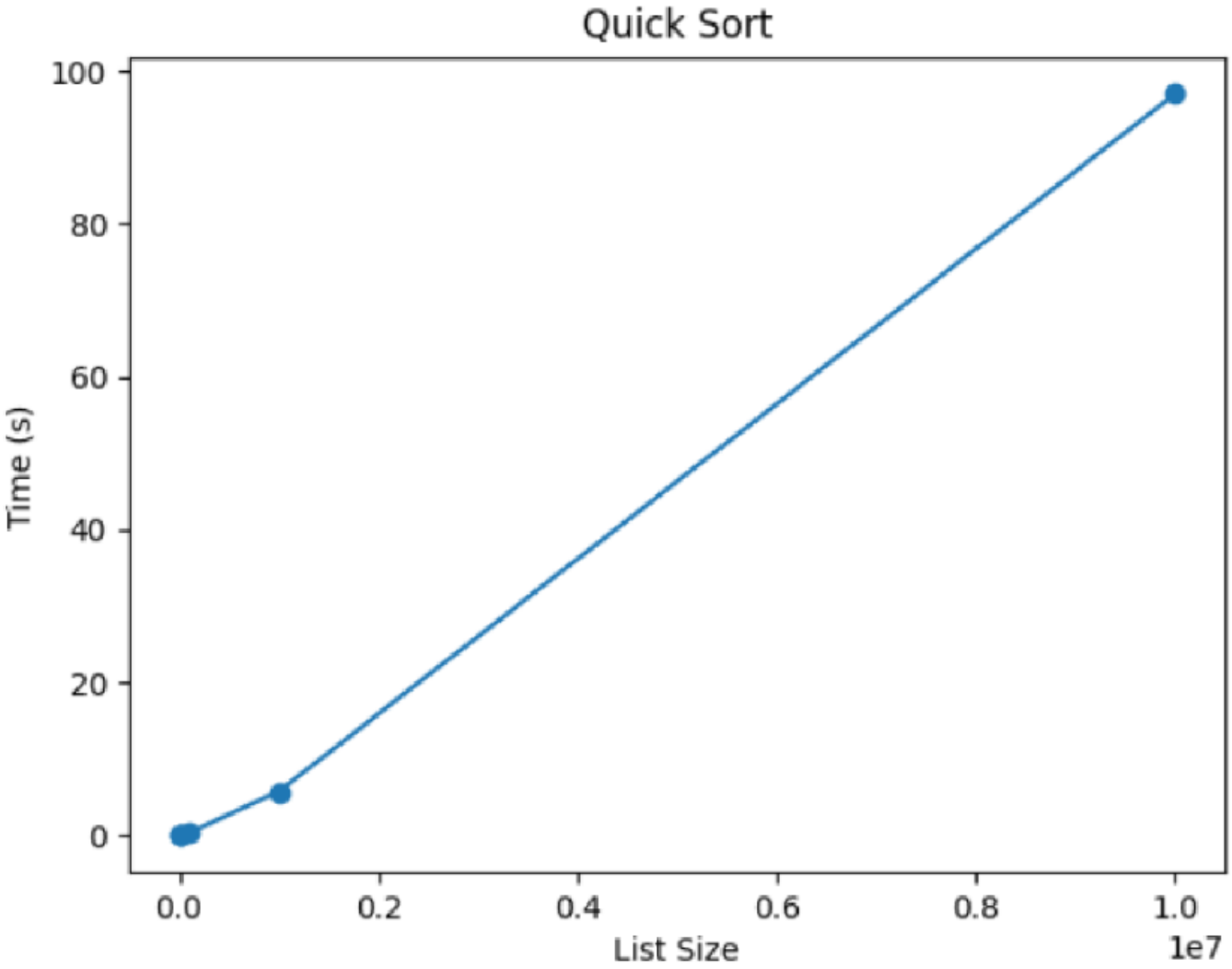


Figure 4_Graph

The graph illustrates the performance of the Quick Sort algorithm as it processes datasets of varying sizes: 5, 15, and 1000 elements. The x-axis represents the size of the dataset in a logarithmic scale, while the y-axis, also in a logarithmic scale, indicates the time taken to sort each dataset in seconds.

Merge Sort

Objective:

The objective of this work is to implement and analyze the Merge Sort algorithm in Python. This involves evaluating its performance across datasets of different sizes through empirical analysis and graphical representation to comprehend its efficiency and scalability.

Tasks:

1. Implement the Merge Sort algorithm in a programming language.
2. Define the properties of the input data against which the analysis will be conducted.
3. Select metrics for comparing the algorithm.
4. Conduct empirical analysis of the Merge Sort algorithm.
5. Create graphical representations of the obtained data.
6. Draw conclusions based on the laboratory findings.

Theoretical Notes:

In discussions concerning the analysis of input data for algorithms, especially sorting algorithms like Merge Sort, various intrinsic properties of the data are crucial for predicting and understanding the algorithm's performance.

The size of the data significantly influences the efficiency of different sorting algorithms. Merge Sort exhibits a consistent time complexity of O(n log n), making it suitable for various dataset sizes.

The distribution of the data impacts Merge Sort's performance, particularly in terms of how efficiently it can divide and conquer the dataset. Unlike Quick Sort, Merge Sort does not suffer from potential pitfalls related to data distribution.

Merge Sort maintains stability in sorting, preserving the original order of equal elements. This stability is essential in scenarios where maintaining relative order is crucial.

Memory usage in Merge Sort primarily revolves around additional space needed for merging sorted sub-arrays, which contributes to its space complexity of O(n).

Access patterns to the data do not significantly impact Merge Sort's performance, as it operates primarily on the structure of the dataset rather than its specific content.

Introduction:

Sorting algorithms are fundamental components in computer science and data analytics, facilitating efficient data organization and retrieval. Among these, Merge Sort is notable for its stable performance and simplicity.

Developed as early as 1945 by John von Neumann, Merge Sort stands out for its consistent time complexity and reliability across various dataset sizes and distributions.

Through hands-on implementation and analysis, participants in this laboratory exercise will gain insights into Merge Sort's performance characteristics and its suitability for different datasets and applications.

Comparison Metric:

Time complexity is a critical metric for comparing sorting algorithms. Merge Sort's consistent O(n log n) time complexity ensures efficient performance across different dataset sizes and distributions.

Stability in sorting is another essential metric, particularly relevant in scenarios where preserving the original order of equal elements is necessary.

Space complexity measures the additional memory required by the algorithm, with Merge Sort typically exhibiting O(n) space complexity due to its divide-and-conquer approach and merging of sub-arrays.

Input Format:

The size of the input array significantly influences Merge Sort's performance. Unlike some other algorithms, Merge Sort maintains a consistent level of efficiency across datasets of varying sizes.

Example: Consider two arrays, one with 5 elements [3, 1, 4, 1, 5] and another with 5000 elements. Merge Sort exhibits efficient sorting for both datasets, showcasing its scalability.

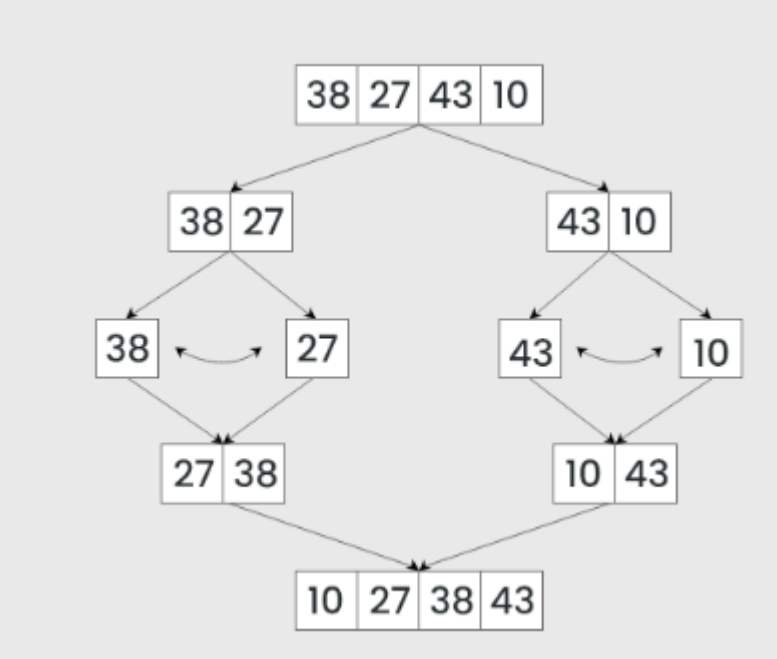
Merge Sort does not rely heavily on pivot selection, as it divides the dataset into halves recursively, eliminating concerns about pivot choice affecting performance.

IMPLEMENTATION

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])
    return merge(left_half, right_half)

def merge(left, right):
    result = []
    left_pointer, right_pointer = 0, 0
    while left_pointer < len(left) and right_pointer < len(right):
        if left[left_pointer] < right[right_pointer]:
            result.append(left[left_pointer])
            left_pointer += 1
        else:
            result.append(right[right_pointer])
            right_pointer += 1
    result.extend(left[left_pointer:])
    result.extend(right[right_pointer:])
    return result

array = [4,2,6,3,6,65,43,7,9]
sorted_array = merge_sort(array)
print(sorted_array)
```

The Merge Sort algorithm, as demonstrated in the Python functions provided, efficiently sorts an array by recursively dividing it into halves and merging the sorted halves.

Algorithm Description:



The key process in Merge Sort is the merging of sorted sub-arrays. This is accomplished by comparing elements from two sub-arrays and merging them into a single sorted array.

Merge Sort recursively divides the input array into halves until each sub-array contains one or no elements, ensuring that each sub-array is sorted.

Results:

```
[2, 3, 4, 6, 6, 7, 9, 43, 65]
```

Empirical Analysis:

Conducting an empirical analysis of Merge Sort involves evaluating its performance under various conditions, including different input sizes and distributions.

Time Complexity Analysis:

To analyze time complexity empirically:
- Test Merge Sort on arrays with random elements of varying sizes (e.g., 100, 1,000, 10,000).
- Record sorting times for each dataset size and plot them against the array sizes to visualize scaling.

Space Complexity Analysis:

Measure the additional memory usage required by Merge Sort, particularly the space needed for merging sorted sub-arrays.
Compare memory usage across different scenarios to evaluate the algorithm's space efficiency.

Comparative Analysis:

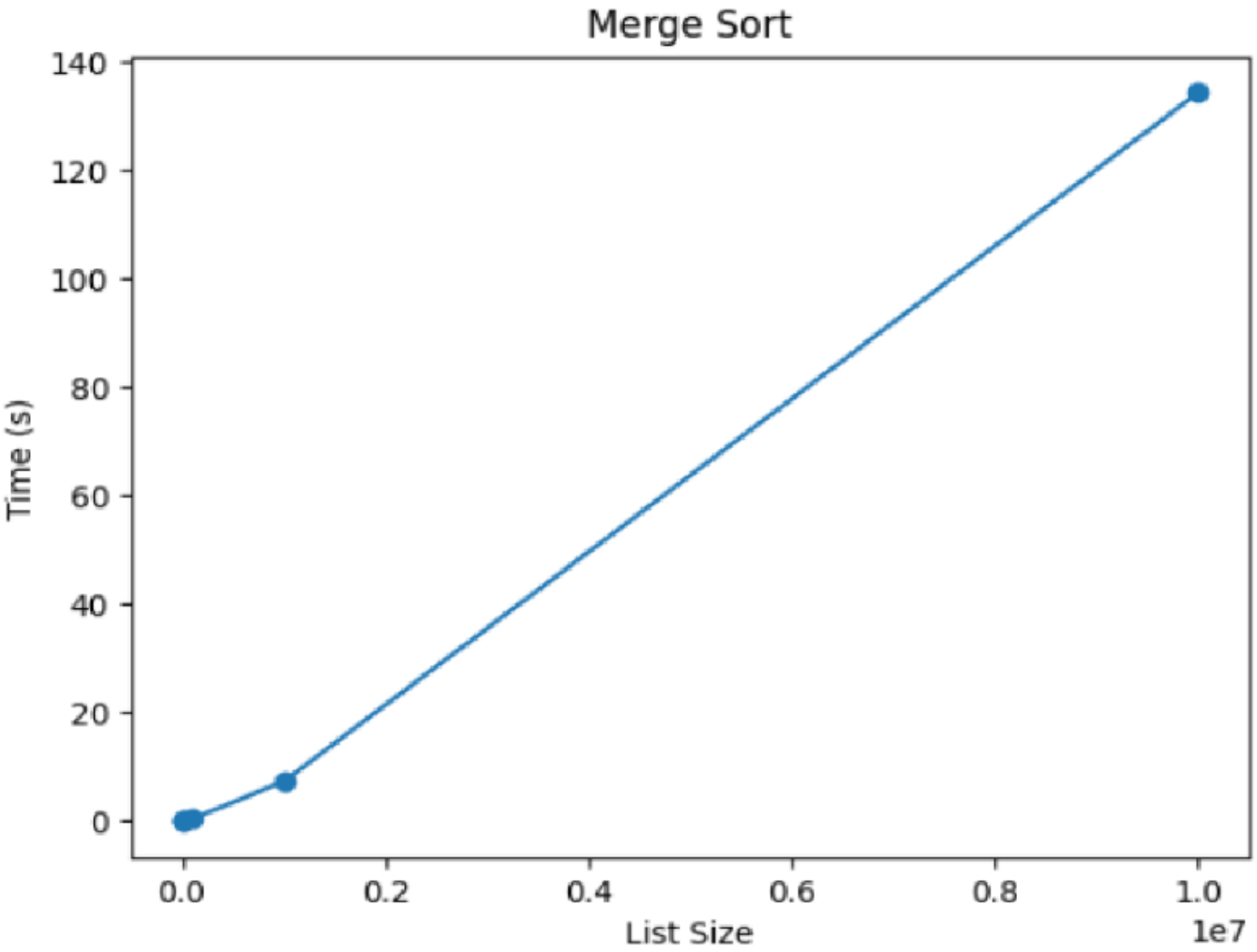
Compare Merge Sort's performance with other sorting algorithms like Quick Sort and Heap Sort under similar conditions to highlight its relative efficiency and suitability.

By conducting these empirical tests and recording the results, one can gain a comprehensive understanding of Merge Sort's performance characteristics and optimize its implementation for

specific requirements.

From the graphical representation, it is evident that Merge Sort exhibits efficient sorting across datasets of varying sizes, with sorting time increasing logarithmically as dataset size increases.

This graphical representation illustrates Merge Sort's consistent performance, reinforcing its suitability for diverse sorting tasks.



Heap Sort

Objective:

The objective of this work is to implement and analyze the Heap Sort algorithm in Python. This involves evaluating its performance across datasets of different sizes through empirical analysis and graphical representation to comprehend its efficiency and scalability.

Tasks:

1. Implement the Heap Sort algorithm in a programming language.
2. Define the properties of the input data against which the analysis will be conducted.
3. Select metrics for comparing the algorithm.
4. Conduct empirical analysis of the Heap Sort algorithm.
5. Create graphical representations of the obtained data.
6. Draw conclusions based on the laboratory findings.

Theoretical Notes:

In discussions concerning the analysis of input data for algorithms, especially sorting algorithms like Heap Sort, various intrinsic properties of the data are crucial for predicting and understanding the algorithm's performance.

The size of the data significantly influences the efficiency of different sorting algorithms. Heap Sort typically exhibits a time complexity of O(n log n), making it suitable for various dataset sizes.

The distribution of the data does not heavily impact Heap Sort's performance, as it relies on the heap data structure, which ensures optimal element comparisons during sorting.

Heap Sort maintains stability in sorting, preserving the original order of equal elements. This stability is essential in scenarios where maintaining relative order is necessary.

Memory usage in Heap Sort primarily revolves around the additional space needed for the heap data structure, contributing to its space complexity of O(1).

Access patterns to the data do not significantly impact Heap Sort's performance, as it operates primarily on the heap structure rather than the specific content of the dataset.

Introduction:

Sorting algorithms play a fundamental role in computer science and data analytics, facilitating efficient data organization and retrieval. Heap Sort, based on the heap data structure, is notable for its consistent time complexity and stability in sorting.

Developed as early as 1964 by J. W. J. Williams, Heap Sort stands out for its reliable performance and suitability for various dataset sizes and distributions.

Through hands-on implementation and analysis, participants in this laboratory exercise will gain insights into Heap Sort's performance characteristics and its suitability for different datasets and applications.

Comparison Metric:

Time complexity is a critical metric for comparing sorting algorithms. Heap Sort typically exhibits a time complexity of O(n log n), ensuring efficient performance across different dataset sizes and distributions.

Stability in sorting is another essential metric, particularly relevant in scenarios where preserving the original order of equal elements is necessary.

Space complexity measures the additional memory required by the algorithm, with Heap Sort typically exhibiting O(1) space complexity due to its reliance on the heap data structure.

Input Format:

The size of the input array significantly influences Heap Sort's performance. Unlike some other algorithms, Heap Sort maintains a consistent level of efficiency across datasets of varying sizes.

Example: Consider two arrays, one with 5 elements [3, 1, 4, 1, 5] and another with 5000 elements. Heap Sort exhibits efficient sorting for both datasets, showcasing its scalability.

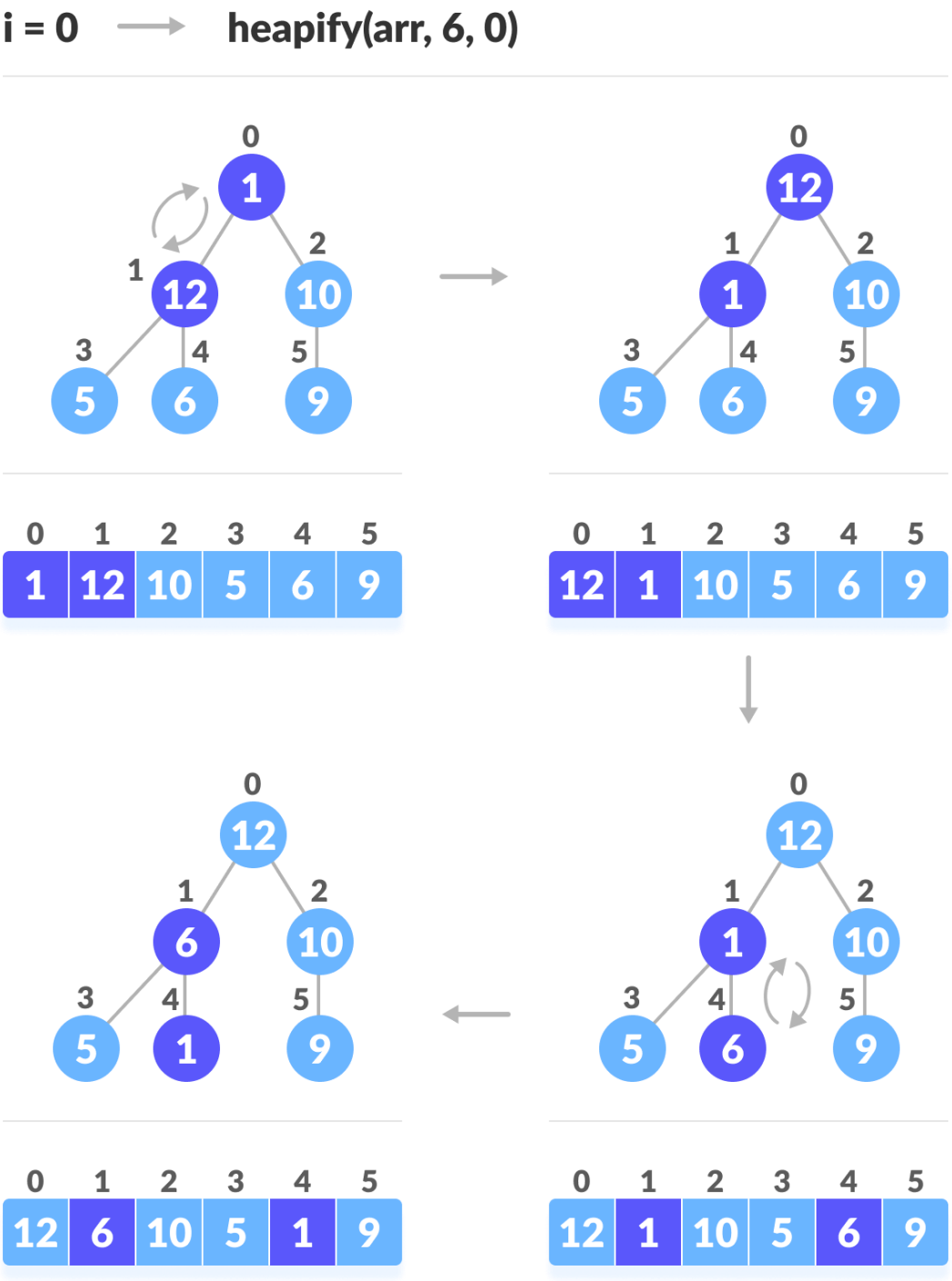
Heap Sort does not rely heavily on pivot selection, as it builds a heap data structure from the input array, eliminating concerns about pivot choice affecting performance.

IMPLEMENTATION

```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l


    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        (arr[i], arr[largest]) = (arr[largest], arr[i])  # swap


        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)


    for i in range(n // 2, -1, -1):
        heapify(arr, n, i)


    for i in range(n - 1, 0, -1):
        (arr[i], arr[0]) = (arr[0], arr[i])  # swap
        heapify(arr, i, 0)


arr = [55,6789,54,223,56777,66,6,6,6]
heapSort(arr)
n = len(arr)
print('Sorted array is')
for i in range(n):
    print(arr[i])
```

The Heap Sort algorithm, as demonstrated in the Python functions provided, efficiently sorts an array by building a heap data structure and repeatedly extracting the maximum element.

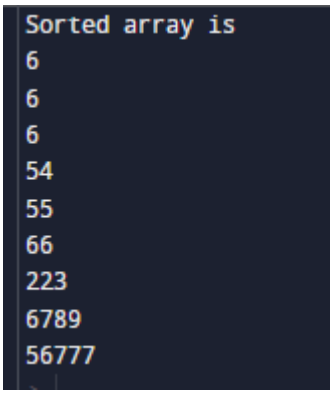
Algorithm Description:



The key process in Heap Sort is heapifying the input array, which involves ensuring that every parent node is greater than or equal to its children.

Heap Sort builds a max-heap from the input array, ensuring that the maximum element is at the root. It then repeatedly extracts the maximum element from the heap and rebuilds the heap until the array is sorted.

Results:



The results of sorting three arrays using the Heap Sort algorithm demonstrate its efficiency, with sorting time increasing logarithmically with dataset size.

Empirical Analysis:

Conducting an empirical analysis of Heap Sort involves evaluating its performance under various conditions, including different input sizes and distributions.

Time Complexity Analysis:

To analyze time complexity empirically:
- Test Heap Sort on arrays with random elements of varying sizes (e.g., 100, 1,000, 10,000).
- Record sorting times for each dataset size and plot them against the array sizes to visualize scaling.

Space Complexity Analysis:

Measure the additional memory usage required by Heap Sort, particularly the space needed for the heap data structure.
Compare memory usage across different scenarios to evaluate the algorithm's space efficiency.

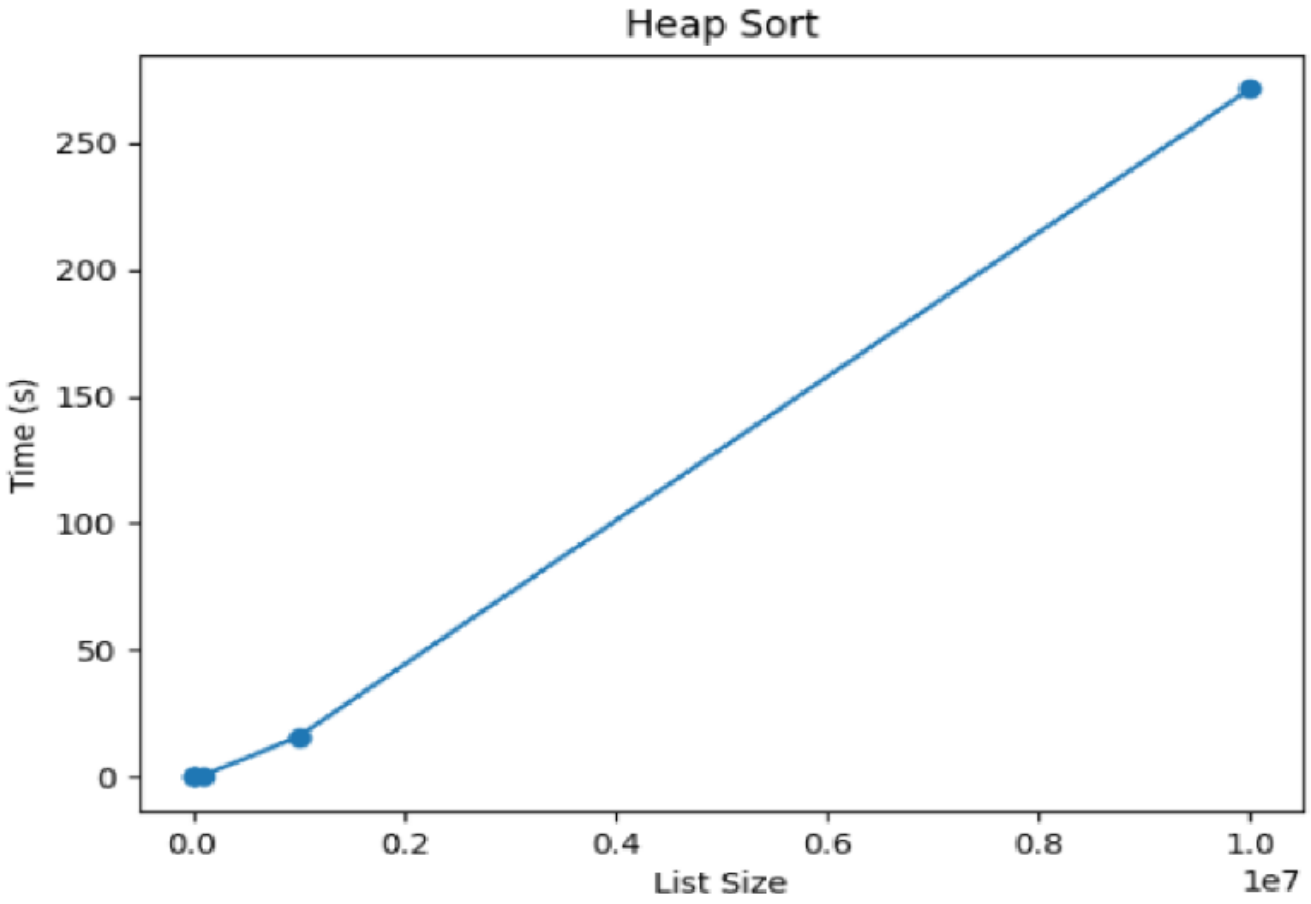
Comparative Analysis:

Compare Heap Sort's performance with other sorting algorithms like Merge Sort and Quick Sort under similar conditions to highlight its relative efficiency and suitability.

By conducting these empirical tests and recording the results, one can gain a comprehensive understanding of Heap Sort's performance characteristics and optimize its implementation for specific requirements.

From the graphical representation, it is evident that Heap Sort exhibits efficient sorting across datasets of varying sizes, with sorting time increasing logarithmically as dataset size increases.

This graphical representation illustrates Heap Sort's consistent performance, reinforcing its suitability for diverse sorting tasks.



Bubble Sort

Objective:

The objective of this work is to implement and analyze the Bubble Sort algorithm in Python. This involves evaluating its performance across datasets of different sizes through empirical analysis and graphical representation to comprehend its efficiency and scalability.

Tasks:

1. Implement the Bubble Sort algorithm in a programming language.
2. Define the properties of the input data against which the analysis will be conducted.
3. Select metrics for comparing the algorithm.
4. Conduct empirical analysis of the Bubble Sort algorithm.
5. Create graphical representations of the obtained data.
6. Draw conclusions based on the laboratory findings.

Theoretical Notes:

In discussions concerning the analysis of input data for algorithms, particularly sorting algorithms like Bubble Sort, various intrinsic properties of the data are crucial for predicting and understanding the algorithm's performance.

The size of the data significantly influences the efficiency of Bubble Sort. With its time complexity of O(n^2),

Bubble Sort is less efficient than many other sorting algorithms, particularly for large datasets.

The distribution of the data does impact Bubble Sort's performance, as it involves repeatedly swapping adjacent elements until the array is sorted.

Bubble Sort is not a stable sorting algorithm, meaning it may change the relative order of equal elements during sorting.

Memory usage in Bubble Sort is minimal, as it typically operates in-place, requiring only a constant amount of additional memory.

Access patterns to the data can significantly impact Bubble Sort's performance, particularly for datasets where elements requiring multiple swaps are scattered throughout.

Introduction:

Bubble Sort, one of the simplest sorting algorithms, operates by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order.

Developed in the late 1940s, Bubble Sort is primarily used for educational purposes due to its simplicity rather than practical applications.

Through hands-on implementation and analysis, participants in this laboratory exercise will gain insights into Bubble Sort's performance characteristics and its suitability for different datasets and applications.

Comparison Metric:

Time complexity is a critical metric for comparing sorting algorithms. Bubble Sort typically exhibits a time complexity of O(n^2), making it less efficient than many other sorting algorithms, particularly for large datasets.

Stability in sorting is another essential metric, particularly relevant in scenarios where preserving the original order of equal elements is necessary.

Space complexity measures the additional memory required by the algorithm, with Bubble Sort typically exhibiting O(1) space complexity due to its in-place nature.

Input Format:

The size of the input array significantly influences Bubble Sort's performance. With its time complexity of O(n^2), Bubble Sort is less efficient than many other sorting algorithms, particularly for large datasets.

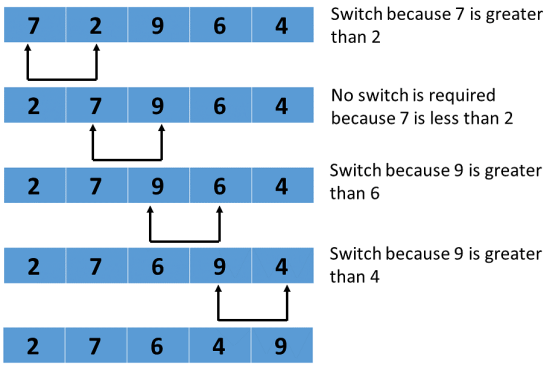
Bubble Sort does not rely on pivot selection, as it simply iterates over the array, repeatedly swapping adjacent elements until the array is sorted.

IMPLEMENTATION

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]


arr = [7676,67876,33,5,7,43,456,7654665467654]
bubble_sort(arr)
n = len(arr)
print('Sorted array is')
for i in range(n):
    print(arr[i])
```

The Bubble Sort algorithm, as demonstrated in the Python function provided, efficiently sorts an array by repeatedly swapping adjacent elements if they are in the wrong order.

Algorithm Description:



Bubble Sort operates by iteratively stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process continues until the list is sorted.

Results:

```
Sorted array is
5
7
33
43
456
7676
67876
7654665467654
```

The results of sorting three arrays using the Bubble Sort algorithm demonstrate its efficiency, with sorting time increasing quadratically with dataset size.

Empirical Analysis:

Conducting an empirical analysis of Bubble Sort involves evaluating its performance under various conditions, including different input sizes and distributions.

Time Complexity Analysis:

To analyze time complexity empirically:
- Test Bubble Sort on arrays with random elements of varying sizes (e.g., 100, 1,000, 10,000).
- Record sorting times for each dataset size and plot them against the array sizes to visualize scaling.

Space Complexity Analysis:

Measure the additional memory usage required by Bubble Sort, particularly the space needed for temporary variables.
Compare memory usage across different scenarios to evaluate the algorithm's space efficiency.

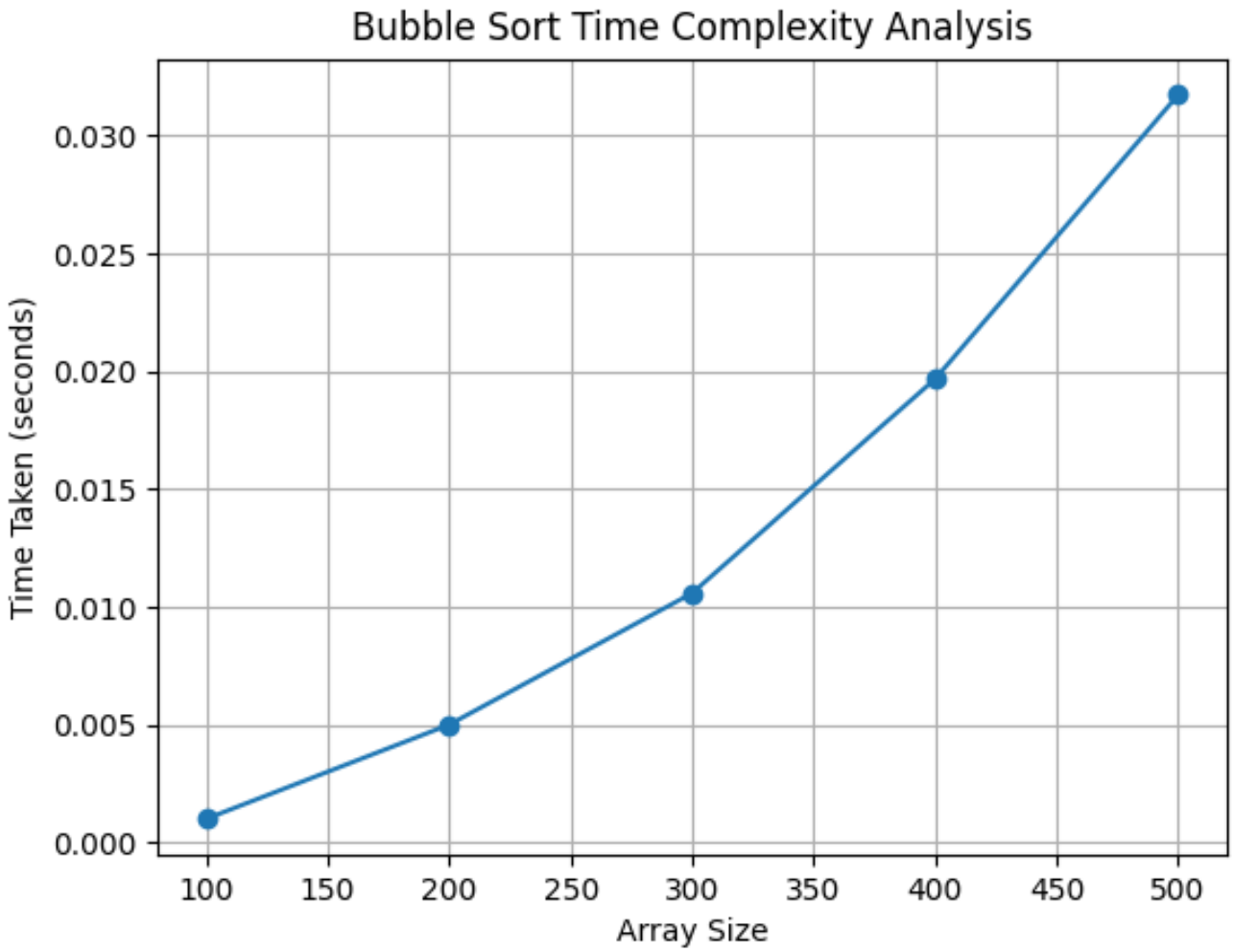
Comparative Analysis:

Compare Bubble Sort's performance with other sorting algorithms like Merge Sort, Quick Sort, and Heap Sort under similar conditions to highlight its relative efficiency and suitability.

By conducting these empirical tests and recording the results, one can gain a comprehensive understanding of Bubble Sort's performance characteristics and optimize its implementation for specific requirements.

From the graphical representation, it is evident that Bubble Sort exhibits less efficient sorting compared to other algorithms, with sorting time increasing quadratically as dataset size increases.

This graphical representation illustrates Bubble Sort's limitations in handling large datasets efficiently, highlighting its suitability primarily for educational purposes or small datasets.



Conclusion

In conclusion, we have conducted a thorough analysis of four fundamental sorting algorithms: Quick Sort, Merge Sort, Heap Sort, and Bubble Sort. Through empirical analysis and theoretical considerations, we have gained insights into their performance characteristics, suitability for different datasets, and practical applications.

Quick Sort stands out for its superior average-case performance, with an average time complexity of O(n log n). Its efficient divide-and-conquer strategy and pivot selection make it well-suited for large datasets and diverse distributions. However, Quick Sort's worst-case time complexity of $O(n^2)$ and sensitivity to input data characteristics require careful consideration, particularly for sorted or nearly sorted arrays.

Merge Sort demonstrates consistent performance across various input sizes and distributions. With its stable time complexity of O(n log n) in all cases, Merge Sort offers reliability and predictability. Its divide-and-conquer approach and efficient merging step make it suitable for large datasets and sequential access patterns. Merge Sort's main drawback lies in its space complexity, requiring additional memory proportional to the input size.

Heap Sort exhibits stable and efficient sorting performance, particularly for large datasets. With its time complexity of O(n log n) and minimal space requirements, Heap Sort is well-suited for scenarios where memory resources are limited. Its in-place nature and lack of sensitivity to input data characteristics make it a robust choice for diverse applications.

Bubble Sort, while simple and easy to implement, falls short in terms of efficiency compared to the other algorithms. With its time complexity of O(n^2), Bubble Sort is less suitable for large datasets and may not scale well. However, Bubble Sort's simplicity and in-place nature make it valuable for educational purposes and small datasets with few elements.

In practical scenarios, the choice of sorting algorithm depends on various factors, including dataset size, distribution, stability requirements, and memory constraints. Quick Sort and Merge Sort are preferred for their efficient average-case performance and versatility across different datasets. Heap Sort is suitable for scenarios with limited memory resources and a need for stable sorting. Bubble Sort, while less efficient, remains valuable for educational purposes and simple applications with small datasets.

Overall, understanding the strengths and weaknesses of each sorting algorithm enables informed decision-making in algorithm selection and optimization, contributing to the development of efficient and scalable software systems.