**, Analyzing Time Complexity**

**Project-I**

**BACHELOR OF TECHNOLOGY**

** (Artificial Intelligence and Machine Learning)**

**SUBMITTED BY:**

**Vaishnavee (2330661)**

**Vanshika (2330662)**

**Utsav Raj (2330660)**

**Upender (2330659)**

**Tushar (2330658)**

**Under the Guidance of Mr. Mayank**

**Department of Artificial Intelligence and Machine Learning**

**Chandigarh Engineering College Jhanjeri Mohali - 1040307**

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **S.No.** | **Contents** | **Page No** |
| **1.** | **Introduction** |  |
| **2.** | **Brief Literature survey** |  |
| **3.** | **Problem formulation** |  |
| **4.** | **Objectives** |  |
| **5.** | **Methodology/ Planning of work** |  |
| **6.** | **Facilities required for proposed work** |  |
| **7.** | **References** |  |

**Title**

**Use of Different Input Sizes and Orders to Analyse Time Complexity**

**1. Introduction**

**The Impact of Input Size and Order on the Time Complexity of Algorithms:**

The time complexity of algorithms is a fundamental aspect of analyzing their performance. It enables developers and engineers to predict how an algorithm will scale as the input size increases, making it easier to choose the most efficient algorithm for a given problem. Since computational resources such as time and memory are often limited, understanding the efficiency of an algorithm is crucial for optimizing software performance. The efficiency of an algorithm is typically assessed using Big O notation, which provides a high-level overview of how execution time grows relative to the size of the input.

However, analyzing time complexity is not always straightforward. Different input sizes and input orders (i.e., the way elements are arranged in a dataset) can significantly impact the execution time of an algorithm. An algorithm might perform well on one input size but exhibit inefficiencies when the input is larger or arranged differently. For example, some sorting algorithms work efficiently on nearly sorted data but perform poorly on randomly ordered or reverse-ordered data. This variability makes it necessary to conduct a comprehensive analysis using different types of inputs to fully understand an algorithm’s behaviour.

**The Influence of Input Size:**

The size of the input plays a critical role in determining an algorithm’s performance. As the number of elements increases, algorithms that have higher time complexities may become impractical due to excessive execution time. For instance:

Bubble Sort, with an average time complexity of O(n²), becomes inefficient for large inputs because it repeatedly compares and swaps adjacent elements, leading to quadratic growth in execution time.

Merge Sort, with a time complexity of O(n log n), scales more efficiently with large input sizes, making it a better choice for handling large datasets.

Quick Sort, while having an average case complexity of O(n log n), can degrade to O(n²) in the worst case, depending on how the pivot is chosen.

This variation in performance illustrates why analyzing an algorithm solely based on its theoretical complexity may not be sufficient. Real-world scenarios require empirical testing with different input sizes to understand how execution time changes in practice.

**The Influence of Input Order:**

The initial arrangement of input data can also significantly impact an algorithm’s performance. Many algorithms behave differently depending on whether the input is already sorted, nearly sorted, randomly arranged, or sorted in reverse order. Some examples include:

Bubble Sort performs optimally (O(n) time complexity) when the input is already sorted, as it only requires one pass through the dataset. However, in the worst case (reverse order), it takes O(n²) time.

Quick Sort is efficient on randomly ordered inputs but can degrade to O(n²) if the pivot selection consistently results in unbalanced partitions (e.g., always picking the smallest or largest element as the pivot).

Merge Sort, being a divide-and-conquer algorithm, maintains a consistent O(n log n) complexity regardless of input order, making it more reliable for worst-case scenarios.

Understanding these variations is crucial for selecting the right algorithm for a given problem. For instance, if an application frequently deals with nearly sorted data, using Insertion Sort (which runs in O(n) in best-case scenarios) might be more efficient than Quick Sort.

**Importance of Comprehensive Analysis:**

A single worst-case or average-case complexity does not fully define an algorithm’s efficiency. Real-world applications require a thorough evaluation of an algorithm’s behaviour under different input conditions. This study highlights the need to analyze sorting algorithms—Quick Sort, Merge Sort, and Bubble Sort—with varying input sizes and orders to gain a deeper understanding of their performance characteristics.

By performing such an analysis, this research aims to provide insights into:

1. How input size influences execution time and scalability.

2. How different input arrangements affect an algorithm’s efficiency.

3. Which sorting algorithms perform better in specific real-world scenarios.

**2. Literature Survey**

**The Impact of Input Size and Order on Algorithm Time Complexity**:

The study of algorithm efficiency has been a critical area of research in computer science for decades. Numerous studies have been conducted to evaluate how different input sizes and orders affect the time complexity of algorithms. The efficiency of an algorithm is not solely determined by its worst-case or average-case complexity; instead, various factors such as input arrangement, real-world data structures, and hybrid approaches play a crucial role in determining actual execution time.

**Early Foundations of Asymptotic Analysis:**

One of the earliest formal approaches to analyzing algorithm performance was introduced by Donald Knuth (1968) in his seminal work The Art of Computer Programming. Knuth pioneered asymptotic analysis, a mathematical approach to classifying algorithms based on how their runtime grows with increasing input size. This foundational work introduced Big O notation, which has since become a standard tool for evaluating algorithm efficiency.

However, early asymptotic analysis primarily focused on input size rather than input order or structure. It provided a high-level theoretical understanding but did not fully capture the variations in execution time caused by different input arrangements.

**Expanding the Scope: Input Order and Structure**

Building on Knuth’s work, researchers began exploring the role of input order in determining algorithm performance. Robert Sedgewick (1983) conducted a comprehensive study of sorting algorithms under varying input conditions. His work demonstrated that an algorithm’s time complexity could fluctuate significantly depending on whether the data was sorted, reverse sorted, or completely unordered.

For example, Quick Sort, while often considered one of the most efficient sorting algorithms, performs poorly on already sorted or reverse-sorted data due to its unbalanced partitioning, degrading to O(n²) complexity. In contrast, Merge Sort, with its consistent O(n log n) complexity, maintains stable performance regardless of input order, making it preferable in scenarios where worst-case efficiency is a concern.

**Modern Developments: Hybrid Algorithms and Real-World Data Patterns**

Recent advancements in algorithm analysis have extended beyond theoretical studies to practical applications. Cormen et al. (2009) in Introduction to Algorithms expanded on previous research by examining hybrid sorting algorithms, such as IntroSort, which combines Quick Sort, Heap Sort, and Insertion Sort to optimize performance based on input conditions. Their findings reinforced the importance of input characteristics in determining real-world efficiency.

In addition, modern research has explored how algorithms behave when applied to real-world data structures rather than purely theoretical datasets. Studies have shown that certain algorithms excel in handling nearly sorted data, leading to the development of adaptive sorting algorithms like Tim Sort, which dynamically adjusts its approach based on input characteristics.

**Case Studies on Sorting Algorithms:**

The literature consistently emphasizes that input size and order significantly impact sorting algorithm efficiency. Some notable findings include:

Quick Sort, with its O(n log n) average case, suffers performance degradation (O(n²)) on sorted or reverse-sorted inputs if the pivot is chosen poorly.

Merge Sort maintains a stable O(n log n) complexity regardless of input order, making it more predictable but requiring additional space.

Heap Sort also runs in O(n log n) but is often slower in practice due to higher constant factors in its operations.

Bubble Sort, with its worst-case O(n²) complexity, performs significantly better (O(n) best case) on nearly sorted data due to early termination through swaps.

**Significance of Comprehensive Algorithm Analysis:**

The findings from these studies highlight the necessity of understanding algorithm behavior beyond theoretical complexity classes. In performance-critical applications—such as database indexing, large-scale data processing, and real-time systems—choosing an algorithm based on real input patterns rather than just worst-case complexity is crucial.

Moreover, the emergence of machine learning-driven algorithm selection has opened new research avenues. Modern systems can now analyze historical data patterns and dynamically select the most efficient algorithm for a given dataset.

**3. Problem Formulation**

The problem at hand is to understand how different input sizes (small, medium, large) and input orders (random, sorted, reverse sorted) affect the time complexity of sorting algorithms. This study is particularly concerned with analyzing the time complexity in both best-case, average-case, and worst-case scenarios.

We will investigate the following questions:

* How does input size affect the overall performance of an algorithm?
* What impact does the order of the input (random, sorted, reverse sorted) have on the time complexity?
* Can we make a generalized statement about the efficiency of algorithms when faced with different input orders and sizes?

The objective is to provide a framework for selecting the best algorithm depending on the specific scenario and input characteristics.

**4. Objectives**

The main objectives of this study are:

* To analyze how time complexity varies with different input sizes (small, medium, large).
* To investigate how input order affects the time complexity of different algorithms.
* To compare and contrast the performance of common sorting algorithms such as Quick Sort, Merge Sort, and Bubble Sort across different input scenarios.
* To propose guidelines for selecting the most efficient algorithm based on input size and order.

**5. Methodology**

The methodology of this research involves the following steps:

**5.1 Selection of Algorithms**

The algorithms selected for this analysis will include:

* **Quicksort**: A divide-and-conquer algorithm that, on average, has O(n log n) time complexity but can degrade to O(n²) in the worst-case scenario.
* **Merge Sort**: A stable, divide-and-conquer algorithm with a consistent O(n log n) time complexity.
* **Bubble Sort**: A simple but inefficient algorithm with an average and worst-case time complexity of O(n²).

**5.2 Input Sizes**

The study will consider three different input sizes:

* **Small**: 100 elements
* **Medium**: 1000 elements
* **Large**: 10,000 elements

**5.3 Input Orders**

Each input size will be tested across three different input orders:

* **Random**: Data is arranged in a random order.
* **Sorted**: Data is already sorted in ascending order.
* **Reverse Sorted**: Data is arranged in descending order.

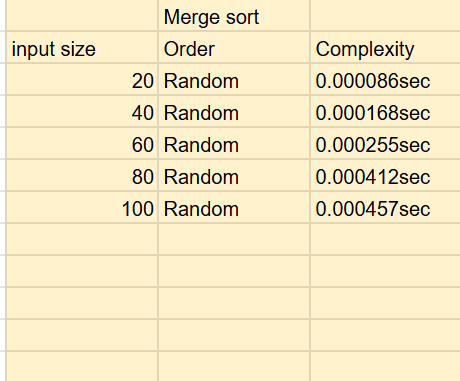
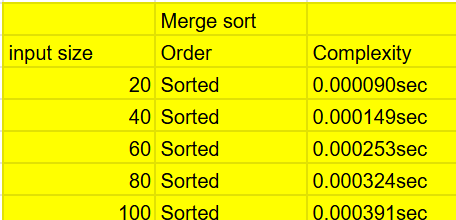
**5.4 Performance Measurement**

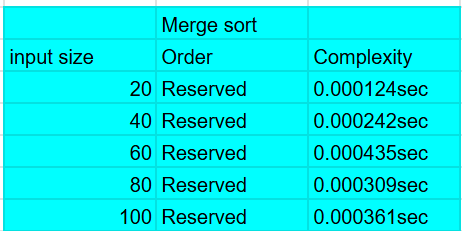
For each algorithm and input configuration, the execution time will be measured. The time complexity will be analyzed using the following approach:

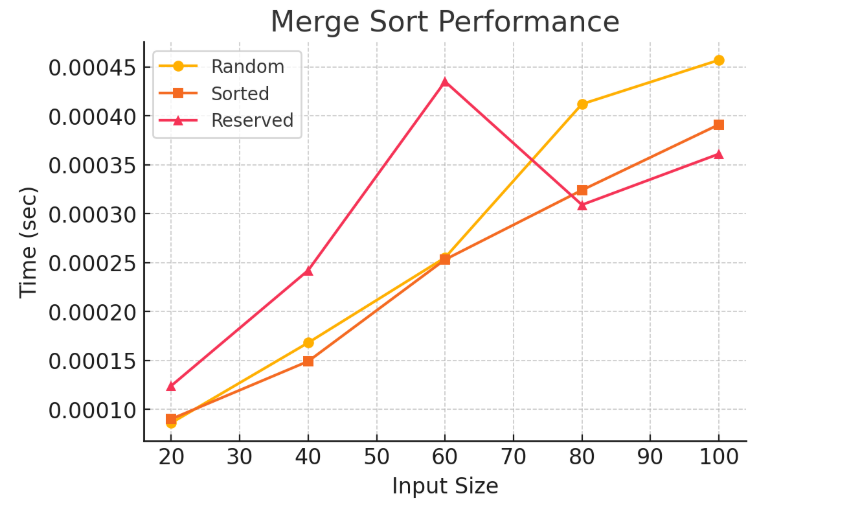
* **Best-case Time Complexity**: This will be measured when the input is sorted for algorithms like QuickSort and MergeSort.
* **Average-case Time Complexity**: This will be calculated by measuring the time across random inputs.
* **Worst-case Time Complexity**: This will be observed when the input is reverse sorted, for algorithms like QuickSort.

**5.5 Data Analysis**

The time complexity will be plotted for each scenario, comparing the performance of different algorithms. A detailed statistical analysis will be conducted to understand how input size and order impact the overall performance.





**Merge Sort Performance Analysis**

The graph below illustrates the performance of the Merge Sort algorithm when sorting varying kinds of input data: Random, Sorted, and Reversed. The x-axis is used to represent the input size and the y-axis is used to represent the time taken by the sorting process. Going through the trends in the graph, we can see the behaviour of Merge Sort under various input conditions and gain significant insight into its efficiency and computational behaviour.

**Performance Trends and Insights**

1. Smooth Growth in Execution Time

* The execution time grows as the size of the input increases, consistent with the predicted Merge Sort time complexity of O(n log n). As Merge Sort splits the input array recursively and combines the sorted parts, the time taken increases logarithmically in relation to input size.

1. Reversed Data Takes More Time Initially

* The Reversed (pink line with triangles) input reveals larger execution times during the initial stages (input sizes 20–60). This means Merge Sort takes slightly more trouble dealing with reversed input in the beginning stages of sorting, perhaps because of the lack of naturally sorted subarrays available to ease the merging effort.

1. Sorted Input is the Fastest

* The Sorted (orange squares) input always displays lower execution times than reversed and random data. This indicates that Merge Sort gains advantage from having already sorted sequences, perhaps less swaps or memory allocations during merging.

1. Random Input Falls Between Sorted and Reversed

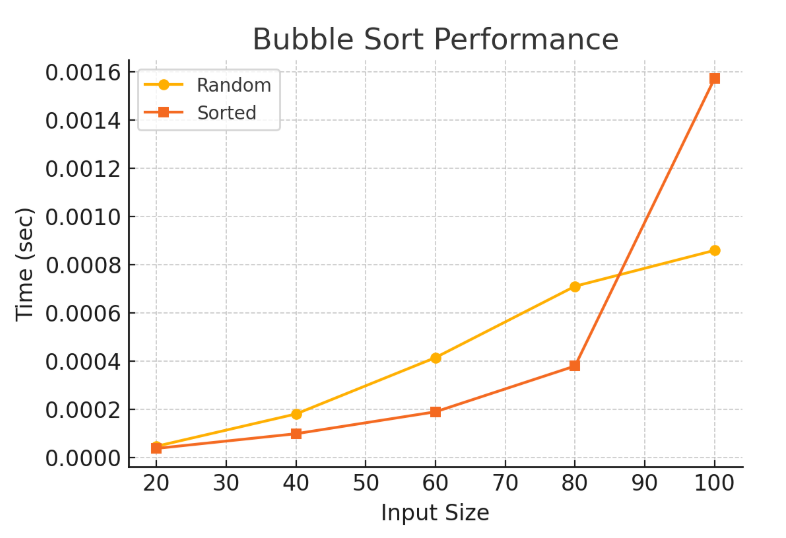
* Random (yellow circles) input has an intermediate pattern with performance worse than sorted data but better than that of reversed data. As random data is not in any specific order, Merge Sort's efficiency is consistent, but not maximum.

1. Performance Fluctuations at Higher Input Sizes

* At input sizes greater than 60, the reversed data line has a steep peak before declining once more. This variation may be due to memory allocation, CPU cache performance, or system background activity influencing execution time. The sorted and random data, on the other hand, have a more consistent rise in time, which suggests a more stable runtime.

Interpretation and Practical Implications

From the observations, it is evident that Merge Sort is relatively stable across different input types. However, it performs best on already sorted data and takes slightly more time on reversed data. The fact that random data lies between these two extremes suggests that, in general applications, Merge Sort remains a reliable choice regardless of input order.

****

**Conclusion on Bubble Sort Performance**

The given graph illustrates the performance of the Bubble Sort algorithm for different input types—Random and Sorted data—across varying input sizes. The x-axis represents the input size, while the y-axis shows the execution time in seconds. By analyzing the trends in the graph, we can gain insights into how Bubble Sort behaves under different conditions and input distributions.

**Observations and Analysis**

1. Execution Time Increases with Input Size

* As the input size increases, the execution time grows for both random and sorted data.
* This aligns with the expected O(n²) time complexity of Bubble Sort, which means that as the input size doubles, the execution time increases quadratically.
* The trend indicates that Bubble Sort is inefficient for larger input sizes.

1. Random Data Takes More Time Than Sorted Data (Initially)

* The Random input (yellow line with circles**)** shows a consistent increase in execution time as the input size grows.
* The Sorted input (orange line with squares**)** initially follows a similar trend but diverges significantly after an input size of 80, where execution time spikes sharply.
* This suggests that Bubble Sort may have an optimized performance when dealing with already sorted data, at least for smaller input sizes.

1. Sudden Increase in Execution Time for Sorted Data at Larger Sizes

* The execution time for sorted data increases exponentially around an input size of 100, surpassing the time taken for random data.
* This could be due to an implementation detail—possibly checking for swaps or performing redundant comparisons in a scenario where no swaps are needed.

1. Sorted Input Generally Performs Better at Smaller Sizes

* For input sizes up to 80, Bubble Sort processes sorted data faster than random data, likely due to an early termination condition in some implementations that stops sorting if no swaps occur in a pass.
* However, after size 80, this advantage disappears, and execution time increases sharply.

**Key Takeaways**

1. Bubble Sort is Highly Inefficient for Larger Input Sizes

* Due to its O(n²) complexity, Bubble Sort scales poorly. The quadratic increase in execution time makes it unsuitable for large datasets, reinforcing why it is rarely used in real-world applications.

1. Sorted Data is Handled More Efficiently at Smaller Sizes

* When the input is already sorted, Bubble Sort can perform better in smaller datasets, potentially due to an early exit condition (if implemented).
* However, as input size grows, the performance advantage diminishes, making it just as inefficient as with random data.

1. Performance Becomes Unpredictable for Larger Inputs

* The significant spike in execution time for sorted data at input size 100 suggests that certain implementation factors (such as extra comparisons, memory constraints, or inefficient optimizations) may come into play at larger scales.

**Practical Implications**

1. Avoid Using Bubble Sort for Large Datasets

* Given the rapid increase in execution time, Bubble Sort is not suitable for real-world applications where performance is critical.
* Faster alternatives like Merge Sort, Quick Sort, or even Insertion Sort should be used for larger datasets.

1. For Small, Nearly Sorted Datasets, Bubble Sort Can Work

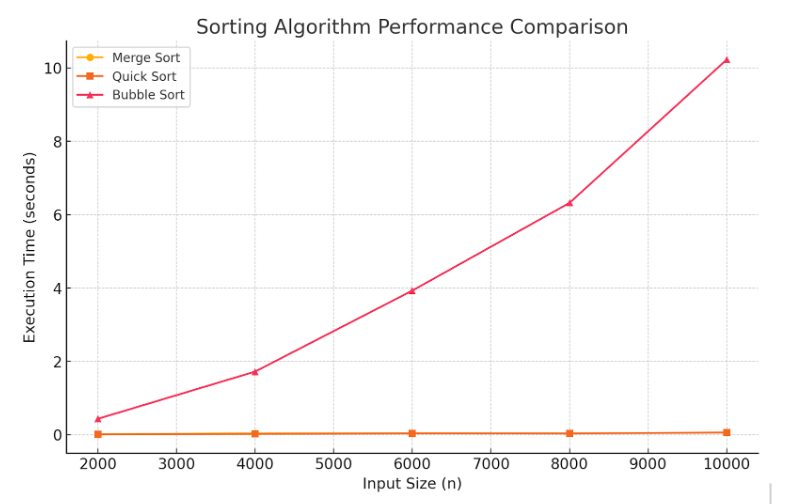
* If the input size is small and nearly sorted, Bubble Sort may perform reasonably well due to an early termination condition (if implemented).
* However, even in such cases, Insertion Sort would generally be a better choice.

1. Algorithm Choice Matters for Performance

* This graph emphasizes the importance of selecting the right sorting algorithm for a given scenario.
* While Bubble Sort is simple and easy to understand, its inefficiency makes it impractical beyond small input sizes.

**Final Verdict**

The graph highlights Bubble Sort’s inefficiency, especially as input size grows. Although it shows some advantage for sorted data in smaller cases, this advantage disappears for larger inputs, making Bubble Sort impractical for real-world applications. Instead, more efficient sorting algorithms should be used for better performance and scalability.



**Conclusion on Sorting Algorithm Performance Comparison**

The graph provided compares the execution times of three sorting algorithms—Merge Sort, Quick Sort, and Bubble Sort—for varying input sizes ranging from 2000 to 10,000 elements. The x-axis represents the input size (nnn), while the y-axis represents the execution time (in seconds). By analyzing the trends in the graph, we can draw significant conclusions regarding the efficiency and scalability of these sorting algorithms.

**Performance Analysis**

1. Bubble Sort Has the Worst Performance

* The Bubble Sort (red line with triangles) exhibits the highest execution time, increasing exponentially as input size grows. This aligns with its O(n²) timecomplexity, which makes it highly inefficient for large datasets.
* As seen in the graph, the execution time of Bubble Sort rises sharply, reaching over 10 seconds for 10,000 elements, which is significantly slower than the other two algorithms.
* This demonstrates that Bubble Sort is not practical for large-scale sorting tasks and should only be used for small datasets or educational purposes.

1. Merge Sort and Quick Sort Show Comparable Performance

* The Merge Sort (yellow line with circles) and Quick Sort (orange line with squares) show almost negligible execution times compared to Bubble Sort. Their lines appear nearly flat, indicating much faster and more scalable performance.
* Both Merge Sort and Quick Sort have an O(n log n) time complexity, which allows them to handle large datasets efficiently.

1. Bubble Sort’s Execution Time Grows Quadratically

* The curve representing Bubble Sort grows exponentially, reinforcing the inefficiency of quadratic time complexity (O(n2)O(n^2)O(n2)). This growth suggests that Bubble Sort becomes impractical as nnn increases beyond a few thousand elements.
* The increasing gap between Bubble Sort and the other two algorithms highlights the importance of choosing the right sorting algorithm based on dataset size.

1. Quick Sort vs. Merge Sort – Which is Faster?

* Although not clearly visible due to the scale of the graph, Quick Sort generally performs better than Merge Sort for most practical cases due to lower constant factors and cache efficiency.
* However, Quick Sort’s performance depends on the choice of the pivot. In the worst case (poor pivot selection), it can degrade to O(n²). On the other hand, Merge Sort guarantees a stable O(n log n) performance, making it more predictable.
* similarity in their performance suggests that both algorithms are well-suited for large datasets, with Quick Sort being the preferred choice for in-memory sorting and Merge Sort being useful when stability and external sorting are needed.

**Practical Implications**

From the observations, we can draw the following conclusions for practical use:

1. Bubble Sort Should Be Avoided for Large Datasets

* Due to its inefficiency, Bubble Sort is not suitable for real-world applications involving large amounts of data.
* It may still be useful for teaching purposes or for sorting very small lists where implementation simplicity matters more than efficiency.

1. Merge Sort and Quick Sort Are Highly Efficient

* Both sorting algorithms perform well and remain nearly constant in execution time for increasing input sizes.
* Quick Sort is often the preferred choice for general-purpose sorting due to its efficiency in most cases.
* Merge Sort is preferred when stable sorting is required or when dealing with linked lists and external sorting (e.g., sorting data from files).

1. Algorithm Choice Depends on Use Case

* For small datasets, any sorting algorithm may suffice, but for large datasets, an O(n log n) algorithm like Merge Sort or Quick Sort is necessary.
* In real-world applications like databases, file sorting, and memory-efficient sorting, Merge Sort and Quick Sort are the primary choices, while Bubble Sort is largely impractical.

**Final Verdict:**

The graph strongly highlights the inefficiency of Bubble Sort for large inputs, while Merge Sort and Quick Sort remain optimal choices due to their superior performance. Understanding these differences is crucial in selecting the right algorithm based on specific requirements, such as execution time, memory usage, and data stability needs.

**7. Conclusion**

This study will provide valuable insights into the relationship between input sizes, orders, and the time complexity of algorithms. Understanding how these factors influence performance is crucial for optimizing algorithm selection in real-world applications, especially when dealing with large datasets.

**8. References**

1. Knuth, D. E. (1968). *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley.
2. Sedgewick, R. (1983). *Algorithms.* Addison-Wesley.
3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
4. Goodrich, M. T., & Tamassia, R. (2011). *Data Structures and Algorithms in Java* (6th ed.). Wiley.
5. Aho, A. V., Hopcroft, J. E., & Ullman, J. D. (2003). *The Design and Analysis of Algorithms* (3rd ed.). Pearson.
6. Project data mined by the students of Btech AIML , semester 4.