****

**A PROJECT REPORT**

**On**

**MAXIMUM NUMBER OF NON-OVERLAPPING SUBSTRINGS**

SUBMITTED TO

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

In partial fulfillment of the award of the course of

**CSA0697-Analyzing the time complexity and performance of different sorting algorithms**

**By**

THARUN.R (192210662)

**SUPERVISOR**

Dr. GNANA SOUNDARI



**SAVEETHA SCHOOL OF ENGINEERING, SIMATS CHENNAI- 602105**

**SEPTEMBER-2024**

**BONAFIDE CERTIFICATE**

Certified that this project report titled **“Analysing the time complexity of difference**

**Sorting algorithm** is the bonafide work **THARUN.R [192210662]** , who carried out the project work under my supervision as a batch. Certified further, that to the best of my knowledge, the work reported here in does not form any other project report.

Project Supervisor Head of the Department

Date: Date:

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.NO** | **CONTENT** | **PAGE.NO** |
| 1. | ABSTRACT |  |
| 2. | OBJECTIVE |  |
| 3. | INTRODUCTION |  |
| 4. | EXISTING TECHNIQUES |  |
| 5. | PROPOSED FEATURES |  |
| 6. | METHODOLOGY |  |
| 7. | MATERIALS AND METHODS |  |
| 8. | FLOWCHART |  |
| 9. | APPLICATIONS |  |
| 10. | SAMPLE CODE |  |
| 11. | SAMPLE OUTPUT |  |
| 12. | RESULTS AND DISCUSSIONS |  |
| 13. | FUTURE ENHANCEMENT |  |
| 14. | CONCLUSION |  |
| 15. | REFERENCES |  |

**ABSTRACT**

The project explores and evaluates the time complexity and performance of various sorting algorithms. The algorithms analyzed include Bubble Sort, Merge Sort, Quick Sort, and others. The study involves a comparative analysis of the algorithms based on input size, time complexity, and computational performance. The goal is to determine the most efficient sorting method under different conditions, providing insights into their practical applications.

**OBJECTIVE**

The primary objective of this project is to:

Analyze the time complexity of different sorting algorithms.

Compare the performance of each sorting algorithm on different data sizes and types (best, average, and worst cases).

Evaluate the trade-offs between time and space complexity.

Identify the most efficient sorting technique for different scenarios and data types.

**INTRODUCTION**

Sorting algorithms play a critical role in various computational tasks, including database indexing, searching, and information retrieval. Analyzing their performance provides insights into optimizing software applications. This project focuses on understanding the time complexity, execution performance, and efficiency of several widely used sorting algorithms, namely:

Bubble Sort

Insertion Sort

Merge Sort

Quick Sort

Heap Sort The study aims to measure and compare their performance in terms of time and space complexity using various datasets.

**EXISTING TECHNIQUES**

**Sorting Algorithms:**

Bubble Sort: A simple comparison-based algorithm with a time complexity of *O*(*n*2)) ) in the worst case.

Insertion Sort: Efficient for small datasets, with a time complexity of ON2 but linear performance

O(n)

O(n) for nearly sorted data.

Merge Sort: A divide-and-conquer algorithm with a guaranteed O(nlogn) time complexity, but higher space complexity.

Quick Sort: Another divide-and-conquer algorithm, generally faster with average-case time complexity

O(n)

O(nlogn), but worst-case

O(n)

O(n 2)

Heap Sort: A comparison-based algorithm that builds a heap and sorts in

O(n)

O(nlogn) with constant space complexity.

**PROPOSED FEATURES**

This project proposes:

Comparative analysis of sorting algorithms on various datasets.

Visualization of performance metrics (time and space complexity).

Discussion of the best algorithm for specific use cases (e.g., large datasets, nearly sorted data).

Implementation of different algorithms using standard programming practices.

A flowchart to describe the overall methodology.

**METHODOLOGY**

Literature Review: Study of time and space complexities of different sorting algorithms from academic sources.

Algorithm Implementation: Implementing each algorithm (Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort) in a programming language (e.g., C, C++, or Python).

Testing and Analysis: Conducting tests on various datasets (random, sorted, reverse-sorted) and measuring their performance using metrics such as execution time and memory usage.

Data Visualization: Plotting graphs comparing the performance of each algorithm.

**MATERIALS AND METHODS**

Materials:

Programming Languages: C/C++/Python

Datasets: Randomly generated datasets, nearly sorted datasets, and reverse sorted datasets.

Tools: GCC compiler or Python interpreter, plotting libraries like Matplotlib (Python) or Gnuplot (C/C++).

Methods:

Implementing algorithms in a modular fashion.

Measuring the time complexity using large datasets.

Using the clock() function (C/C++) or time module (Python) to record execution times.

**FLOW DIAGRAM:**

**+------------------------+**

**| Start |**

**+------------------------+**

**|**

**v**

**+------------------------+**

**| Input Dataset Size |**

**+------------------------+**

**|**

**v**

**+------------------------+**

**| Apply Sorting Algorithms|**

**+------------------------+**

**|**

**v**

**+------------------------+**

**| Measure Performance |**

**+------------------------+**

**|**

**v**

**+------------------------+**

**| Compare Time Complexity |**

**+------------------------+**

**|**

**v**

**+------------------------+**

**| Display Results |**

**+------------------------+**

**|**

**v**

**+------------------------+**

**| End |**

**+------------------------+**

**APPLICATIONS**

Database Management: Efficient sorting techniques can enhance search operations and data organization in databases.

File Sorting: Sorting algorithms are used in file systems to manage and access large data files efficiently.

Algorithm Design: Helps in developing efficient algorithms by selecting appropriate sorting techniques based on input characteristics.

Here’s an example of the Quick Sort algorithm in Python:

def quick\_sort(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 quick\_sort(left) + middle + quick\_sort(right)

# Sample input

arr = [3, 6, 8, 10, 1, 2, 1]

print("Original Array:", arr)

sorted\_arr = quick\_sort(arr)

print("Sorted Array:", sorted\_arr)

**RESULTS AND DISCUSSIONS**

The results of the analysis show that algorithms like Merge Sort and Quick Sort perform consistently well with large datasets due to their average-case time complexity of

O(nlogn). However, algorithms like Bubble Sort and Insertion Sort perform poorly with larger inputs due to their O(nlogn).

O(nlogn). )

O(n 2)

time complexity. Heap Sort offers a reliable choice when space efficiency is a concern. The analysis demonstrates that the choice of sorting algorithm depends heavily on the input data's size and initial state.

**Key Findings:**

Quick Sort is generally the fastest, except in the worst-case scenario.

Merge Sort is reliable with predictable performance but requires extra memory.

Heap Sort balances performance and space efficiency well.

**FUTURE ENHANCEMENT**

In the future, the following improvements can be considered:

Parallel Sorting: Implementing parallel sorting techniques to speed up execution on multi-core systems.

Hybrid Sorting: Combining multiple algorithms to optimize performance based on dataset characteristics.

Advanced Data Structures: Analyzing the performance of sorting algorithms on complex data structures (e.g., linked lists, binary trees).

**CONCLUSION**

This project provides a comprehensive analysis of the time complexity and performance of various sorting algorithms. The results suggest that the choice of sorting technique depends on the dataset size, data characteristics, and performance needs. Quick Sort and Merge Sort are generally the most efficient for larger datasets, while simpler algorithms like Insertion Sort may be preferred for smaller or nearly sorted data.

**REFERENCES**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.

Knuth, D. E. (1997). The Art of Computer Programming: Sorting and Searching. Addison-Wesley.

Sedgewick, R. (1988). Algorithms. Addison-Wesley.