**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**CHENNAI-602105**

Create sorted array

**A CAPSTONE PROJECT REPORT**

*Submitted in the partial fulfillment for the award of the degree of*

# BACHELOR OF ENGINEERING

**IN COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE AND DATA SCIENCE**

**Submitted by**

**Y sasi kumar (192210474) Under the Supervision of**

**Dr. kanimozhi**

# DECLARATION

I, Y sasi kumar student of **Bachelor of Engineering in Computer Science Engineering and Artificial Intelligence and Data Science** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **"Title"** is the outcome of my own bonafide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

(Y sasi kumar 192210474)

Date:22/09/2024

Place:Saveetha School of Engineering, Thandalam.

# CERTIFICATE

This is to certify that the project entitled **“Title”** submitted by

Y sasi kumar has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B.E Computer science engineering and B.Tech Artificial Intelligence in Data science.

Faculty-in-charge

Dr. kanimozhi

**ABSTRACT**

**The problem of finding the median of two sorted arrays is a classic algorithmic challenge in computer science. Given two sorted arrays, the goal is to determine the median value efficiently, leveraging the properties of sorted sequences. This problem is particularly interesting because it can be solved in logarithmic time complexity, O(log(min(n, m))), where n and m are the lengths of the two arrays.**

**This abstract presents the significance of the problem, outlines the optimal approach, and discusses potential applications. The median, defined as the middle value in a sorted list, is crucial in various statistical analyses and data processing tasks. By combining the two arrays and identifying the median without explicitly merging them, we can achieve significant performance improvements, particularly with large datasets.**

**We explore a binary search strategy to partition the arrays in such a way that the left partitions contain the smaller elements and the right partitions contain the larger elements. This ensures that the median can be computed efficiently. The proposed method is not only theoretically sound but also has practical implications in real-time data analytics, where performance is paramount.**

**This work emphasizes the importance of understanding algorithmic efficiency and the intricacies of handling data structures, paving the way for further research and development in algorithm optimization and statistical methods.**

**The output of the algorithm is an array where each element represents the number of smaller elements to its right in the original array.**

**Theoretical Framework:**

**Time Complexity: O(n log n)**

**Space Complexity: O(n)**

**Algorithmic Paradigms: Divide-and-Conquer, Binary Indexed**

**Tree, Augmented Data Structures**

**KeywordsArray**

**➤ Merge sort**

**➤ Algorithm**

**➤ Time complexity**

**Space complexity**

**➤ Data structures**

**➤ Programing**

**INTRODUCTION**

**Sorting algorithms are fundamental to computer science and are used in a wide**

**range of applications.**

**Data Organization**

**Sorting algorithms are crucial for organizing data in a meaningful way, making it**

**easier to search, access, and process information.**

**Efficiency**

**Understanding sorting algorithms helps us optimize code and improve the**

**efficiency of our programs.**

**Problem-Solving**

**They provide a framework for solving various computational problems, including**

**searching, merging, and finding the minimum or maximum elements in a dataset.**

**Real-World Applications**

**Sorting algorithms are applied in countless real-world applications, such as**

**database management, search engines, and data analysis.**

**Sorting Algorithms Overview**

**There are numerous sorting algorithms, each with its own strengths and weaknesses.**

**Bubble Sort**

**Compares adjacent elements and swaps them if they are in the wrong order.**

**Simple but inefficient for large datasets.**

**Insertion Sort**

**Builds a sorted array by inserting elements one at a time into their correct**

**positions. Efficient for small datasets.**

**Merge Sort**

**Divides the array into halves, sorts each half recursively, and then merges the**

**sorted halves. Generally efficient for large datasets.**

**CODING**

**CODING**

**#include <stdio.h>**

**void bubbleSort(int arr[], int n) {**

**for (int i = 0; i < n - 1; i++) {**

**for (int j = 0; j < n - i - 1; j++) {**

**if (arr[j] > arr[j + 1]) {**

**int temp = arr[j];**

**arr[j] = arr[j + 1];**

**arr[j + 1] = temp;**

**}**

**}**

**}**

**}**

**int main() {**

**int arr[] = {64, 34, 25, 12, 22, 11, 90};**

**int n = sizeof(arr) / sizeof(arr[0]);**

**bubbleSort(arr, n);**

**printf("Sorted array: \n");**

**for (int i = 0; i < n; i++) {**

**printf("%d ", arr[i]);**

**}**

**printf("\n");**

**return 0;**

**}**

**OUTPUT OUTPUT**

**Sorted array:**

**11 12 22 25 34 64 90**

**Complexity Analysis**

**Complexity Analysis**

**helps understand the performance of algorithms,**

**particularly for large datasets.**

**Time Complexity**

**Measures the time taken by an algorithm to complete its task, typically**

**expressed in terms of the input size.**

**Space Complexity**

**Measures the amount of memory used by an algorithm during execution**

**Best Case - O(n)**

**- Occurs when the input array is already sorted in ascending order.**

**- Each insertion takes constant time.**

**WorstCase:** **O(n log n)**

**- Occurs when the input array is sorted in descending order.**

**- Each insertion requires binary search (O(log n)) and shifting elements (O(n)).**

**AverageCase:** **O(n log n)**

**- Assumes random input data.**

**- Binary search reduces the number of comparisons, but insertions still require shifting elements.**

**Overall Complexity - Time Complexity: O(n log n)**

**- Space Complexity: O(n)**

**CONCLUSION**

The Bubble Sort algorithm is a simple and intuitive sorting algorithm, but it is not the most

efficient choice for large datasets due to its quadratic time complexity in the worst and

average cases.

Ease of Implementation

It is straightforward to understand and implement, making it a good starting point for

learning about sorting algorithms.

Limited Efficiency

Its performance degrades significantly with larger datasets, making it impractical for many

real-world applications.

Educational Value

It provides valuable insights into fundamental sorting concepts, such as comparisons and

swaps, which are crucial for understanding more advanced algorithms.