## SIMATS SCHOOL OF ENGINEERING

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

#### CHENNAI-602105

A CAPSTONE PROJECT REPORT

optimization and analysis of sorting algorithms: a comparative study

COURSECODE/NAME: CSA0658-Design And Analysis 0f Algorithms For Machine Learning

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

# BACHELOR OF ENGINEERING

## IN COMPUTER SCIENCE

**Submitted by**

**A. Sarvani Kanyaka-192211795**

**C. Aswini chowdary-192211442**

**Guided by**

**Dr. Senthil Vadivu**

**Associate Professor**

**Department of Computer Science and Engineering**

# DECLARATION

I, A. Sarvani Kanyaka, C. Aswini Chowdary students of **Bachelor of Engineering in Computer Science Engineering** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **"** **optimization and analysis of sorting algorithms: a comparative study "** 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.

A.Sarvani Kanyaka- 192211795

C.Aswini Chowdary-192211442

Date:27/07/2024

Place: Saveetha School of Engineering, Thandalam.

# CERTIFICATE

This is to certify that the project entitled **“optimization and analysis of sorting algorithms: a comparative study”** submitted by A.Sarvani Kanyaka , C.Aswini Chowdary 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.

Faculty-in-charge :  **Dr.S. Senthil Vadivu**

**ABSTRACT**

Sorting algorithms are fundamental to computer science, serving as a cornerstone for various applications, from database management to optimization problems. This comparative study analyzes and optimizes a selection of commonly used sorting algorithms, including Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort, and Bucket Sort. The study provides a theoretical analysis based on time complexity, space complexity, and stability, alongside practical considerations such as ease of implementation and performance optimization techniques. An experimental evaluation using diverse datasets quantifies the algorithms' efficiency, highlighting trade-offs and context-specific advantages.

In the realm of computer science, sorting algorithms are vital for organizing data efficiently, impacting everything from search algorithms to data analysis techniques. This study conducts a comprehensive evaluation of several well-known sorting algorithms, including both comparison-based methods like Quick Sort and Merge Sort, and non-comparison-based methods like Radix Sort and Bucket Sort. We analyze these algorithms under various conditions, focusing on their time and space complexities in the best, average, and worst-case scenarios.

**KEYWORDS**

1. Bubble Sort

2. Selection Sort

3. Insertion Sort

4. Merge Sort

5. Quick Sort

6. Heap Sort

7. Radix Sort

8. Bucket Sort

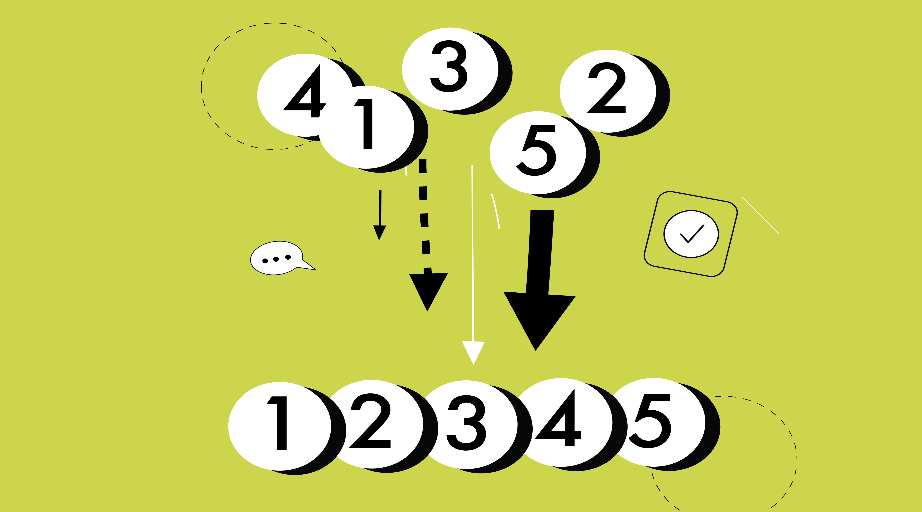
**INTRODUCTION**

Sorting is a fundamental operation in computer science and plays a critical role in various applications, including database management, data analytics, and algorithm optimization. The process involves arranging data in a specific order, typically numerical or lexicographical, to enable efficient data retrieval and processing. Given the diverse nature of data and specific application requirements, a multitude of sorting algorithms have been developed, each with its strengths and weaknesses.

This study focuses on a comparative analysis of several widely used sorting algorithms, such as Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Radix Sort, and Bucket Sort. By examining these algorithms, we aim to provide a detailed understanding of their theoretical foundations, including time and space complexities, stability, and suitability for different types of datasets.

Moreover, the study explores various optimization techniques to enhance the performance of these algorithms. This includes considerations for parallel processing, memory usage, and adaptive strategies that adjust the algorithm's behavior based on the data characteristics. Through both theoretical analysis and empirical evaluation, we seek to identify the most efficient algorithms for specific scenarios, offering practical insights into their implementation and optimization. This comprehensive approach not only aids in selecting the appropriate sorting algorithm for given tasks but also contributes to the broader field of algorithm design and optimization in computer science.

The selection of an appropriate sorting algorithm can significantly impact the performance and efficiency of data processing tasks, particularly in environments where large volumes of data are involved. While traditional algorithms like Bubble Sort and Selection Sort are simple and easy to implement, they often suffer from poor performance on large datasets due to their quadratic time complexity. On the other hand, more advanced algorithms like Quick Sort and Merge Sort offer better average-case performance, with Quick Sort being particularly notable for its efficiency in practice despite its O(n²) worst-case complexity. In contrast, non-comparison-based algorithms such as Radix Sort and Bucket Sort provide linear time complexity under certain conditions, making them ideal for specific types of data, such as fixed-length integer keys. This study's comparative analysis aims to demystify these trade-offs, providing a clearer picture of each algorithm's capabilities and limitations, thus enabling better decision-making in the selection and application of sorting techniques.



**RESEARCH PLAN:**

**GANTT CHART:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S.NO** | **DESCRIPTION** | **23.07.24**  **DAY-01** | **24.07.24**  **DAY-02** | **25.07.24**  **DAY-03** | **26.07.24**  **DAY-04** | **27.07.24**  **DAY-05** |
| **1.** | Problem Identification |  |  |  |  |  |
| **2.** | Introduction |  |  |  |  |  |
| **3.** | Analysis, Design |  |  |  |  |  |
| **4.** | Implementation |  |  |  |  |  |
| **5.** | Conclusion |  |  |  |  |  |

**CODING**

**1.Bubble Sort-**

#include <stdio.h>

// Function to perform Bubble Sort

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

int i, j;

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

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

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

// Swap arr[j] and arr[j+1]

int temp = arr[j];

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

arr[j+1] = temp;

}

}

}

}

// Function to print an array

void printArray(int arr[], int size) {

int i;

for (i = 0; i < size; i++)

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

printf("\n");

}

// Main function

int main() {

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

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

printf("Original array: \n");

printArray(arr, n);

bubbleSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**Output-**

Original array:

64 34 25 12 22 11 90

Sorted array:

11 12 22 25 34 64 90

**2.Selection Sort-**

#include <stdio.h>

// Function to perform selection sort

void selectionSort(int arr[], int n) {

int i, j, min\_idx, temp;

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

// Find the minimum element in the unsorted portion of the array

min\_idx = i;

for (j = i + 1; j < n; j++) {

if (arr[j] < arr[min\_idx]) {

min\_idx = j;

}

}

// Swap the found minimum element with the first element of the unsorted portion

temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

// Function to print an array

void printArray(int arr[], int size) {

int i;

for (i = 0; i < size; i++) {

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

}

printf("\n");

}

int main() {

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

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

printf("Original array: ");

printArray(arr, n);

selectionSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

**Output-**

Original array: 64 25 12 22 11

Sorted array: 11 12 22 25 64

**3.Insertion Sort**

#include <stdio.h>

// Function to perform insertion sort

void insertionSort(int arr[], int n) {

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

int key = arr[i];

int j = i - 1;

// Move elements of arr[0..i-1] that are greater than key

// to one position ahead of their current position

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

// Function to print an array

void printArray(int arr[], int n) {

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

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

}

printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6};

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

printf("Original array: \n");

printArray(arr, n);

insertionSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**Output-**

Original array: 12 11 13 5 6

Sorted array: 5 6 11 12 13

**4.Merge Sort-**

#include <stdio.h>

#include <stdlib.h>

// Function to merge two subarrays of arr[]

void merge(int arr[], int l, int m, int r) {

int n1 = m - l + 1;

int n2 = r - m;

// Create temporary arrays

int\* L = (int\*)malloc(n1 \* sizeof(int));

int\* R = (int\*)malloc(n2 \* sizeof(int));

// Copy data to temporary arrays L[] and R[]

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

// Merge the temporary arrays back into arr[l..r]

int i = 0; // Initial index of first subarray

int j = 0; // Initial index of second subarray

int k = l; // Initial index of merged subarray

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

// Free the allocated memory

free(L);

free(R);

}

// Function to divide the array into subarrays

void mergeSort(int arr[], int l, int r) {

if (l < r) {

// Find the middle point

int m = l + (r - l) / 2;

// Recursively sort the first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

// Merge the sorted halves

merge(arr, l, m, r);

}

}

// Function to print an array

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++)

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

printf("\n");

}

// Main function

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

int arr\_size = sizeof(arr) / sizeof(arr[0]);

printf("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("\nSorted array is \n");

printArray(arr, arr\_size);

return 0;

}

**Output-**

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

**5.Quick Sort-**

#include <stdio.h>

// Function to swap two elements

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

// Partition function for Quick Sort

int partition(int arr[], int low, int high) {

int pivot = arr[high]; // Choose the last element as pivot

int i = (low - 1); // Index of the smaller element

for (int j = low; j < high; j++) {

// If current element is smaller than the pivot

if (arr[j] < pivot) {

i++; // Increment index of smaller element

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]); // Swap the pivot element to its correct position

return (i + 1);

}

// Quick Sort function

void quickSort(int arr[], int low, int high) {

if (low < high) {

// Partition the array

int pi = partition(arr, low, high);

// Recursively sort elements before and after partition

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

// Function to print the array

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++)

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

printf("\n");

}

// Main function to test the Quick Sort implementation

int main() {

int arr[] = {10, 7, 8, 9, 1, 5};

int size = sizeof(arr) / sizeof(arr[0]);

printf("Unsorted array: \n");

printArray(arr, size);

quickSort(arr, 0, size - 1);

printf("Sorted array: \n");

printArray(arr, size);

return 0;

}

**Output-**

Unsorted array:

10 7 8 9 1 5

Sorted array:

1 5 7 8 9 10

**6.Heap Sort-**

#include <stdio.h>

// Function to heapify a subtree rooted at index i

void heapify(int arr[], int n, int i) {

int largest = i; // Initialize largest as root

int left = 2 \* i + 1; // left = 2\*i + 1

int right = 2 \* i + 2; // right = 2\*i + 2

// If left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root

if (largest != i) {

// Swap root with largest

int temp = arr[i];

arr[i] = arr[largest];

arr[largest] = temp;

// Recursively heapify the affected subtree

heapify(arr, n, largest);

}

}

// Function to perform heap sort

void heapSort(int arr[], int n) {

// Build heap (rearrange array)

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// One by one extract elements from heap

for (int i = n - 1; i >= 0; i--) {

// Move current root to end

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Call heapify on the reduced heap

heapify(arr, i, 0);

}

}

// Function to print an array

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++) {

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

}

printf("\n");

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

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

printf("Unsorted array: \n");

printArray(arr, n);

heapSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**Output-**

Unsorted array: 12 11 13 5 6 7

Sorted array: 5 6 7 11 12 13

**7.Radix Sort-**

#include <stdio.h>

#include <stdlib.h>

// Function to get the maximum value in the array

int getMax(int arr[], int n) {

int max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

// Function to perform counting sort based on a specific digit represented by exp

void countingSort(int arr[], int n, int exp) {

int output[n];

int count[10] = {0};

// Count occurrences of each digit

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

count[(arr[i] / exp) % 10]++;

}

// Update count[i] to contain the position of this digit in output[]

for (int i = 1; i < 10; i++) {

count[i] += count[i - 1];

}

// Build the output array

for (int i = n - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

// Copy the output array to arr[], so that arr[] contains sorted numbers

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

arr[i] = output[i];

}

}

// Main function to implement radix sort

void radixSort(int arr[], int n) {

int max = getMax(arr, n);

// Apply counting sort for every digit.

// exp is 10^i where i is the current digit number

for (int exp = 1; max / exp > 0; exp \*= 10) {

countingSort(arr, n, exp);

}

}

// Function to print an array

void printArray(int arr[], int size) {

for (int i = 0; i < size; i++) {

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

}

printf("\n");

}

// Main function to test the radix sort

int main() {

int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};

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

printf("Original array:\n");

printArray(arr, n);

radixSort(arr, n);

printf("Sorted array:\n");

printArray(arr, n);

return 0;

}

**Output-**

Original array:

170 45 75 90 802 24 2 66

Sorted array:

2 24 45 66 75 90 170 802

**8.Bucket Sort-**

#include <stdio.h>

#include <stdlib.h>

#define BUCKET\_SIZE 10

// Function to sort the elements in the bucket

void insertionSort(float arr[], int n) {

int i, j;

float key;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

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

j = j - 1;

}

arr[j + 1] = key;

}

}

// Function to perform bucket sort

void bucketSort(float arr[], int n) {

if (n <= 0) return;

// Create buckets and initialize them

float buckets[BUCKET\_SIZE][n];

int bucketCount[BUCKET\_SIZE];

for (int i = 0; i < BUCKET\_SIZE; i++) {

bucketCount[i] = 0;

}

// Distribute input array elements into buckets

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

int index = BUCKET\_SIZE \* arr[i];

buckets[index][bucketCount[index]++] = arr[i];

}

// Sort individual buckets and concatenate

int idx = 0;

for (int i = 0; i < BUCKET\_SIZE; i++) {

if (bucketCount[i] > 0) {

insertionSort(buckets[i], bucketCount[i]);

for (int j = 0; j < bucketCount[i]; j++) {

arr[idx++] = buckets[i][j];

}

}

}

}

// Function to print the array

void printArray(float arr[], int n) {

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

printf("%.2f ", arr[i]);

}

printf("\n");

}

int main() {

float arr[] = {0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68};

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

printf("Original array: \n");

printArray(arr, n);

bucketSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

**Output-**

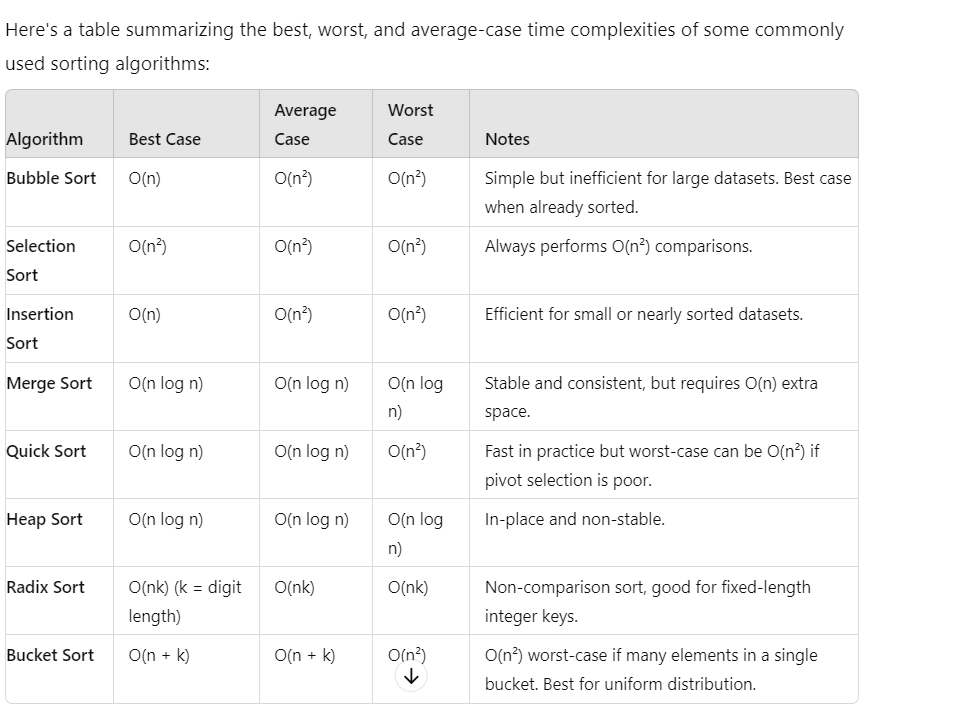
Original array:

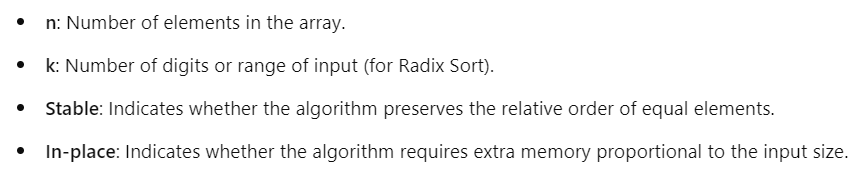
0.78 0.17 0.39 0.26 0.72 0.94 0.21 0.12 0.23 0.68

Sorted array:

0.12 0.17 0.21 0.23 0.26 0.39 0.68 0.72 0.78 0.94

**COMPLEXITY ANALAYSIS**

****

****

**RESULT:**

* Best: Merge Sort—offers consistent O(n log n) performance, stability, and good performance on large datasets.
* Worst: Bubble Sort—inefficient with O(n²) time complexity in all cases, making it impractical for larger datasets.
* Average: quick sort-O(n log n), which is efficient and competitive with other O(n log n) algorithms for most practical applications.

**CONCLUSION**

Sorting algorithms are integral to many computational processes, and selecting the right one can significantly affect the efficiency and effectiveness of data handling. This comparative study on sorting algorithms, including a focus on Bucket Sort, underscores the importance of understanding both theoretical and practical aspects of sorting techniques.

Bucket Sort, in particular, demonstrates its utility when dealing with uniformly distributed data. Its linear time complexity, O(n + k), where \( k \) represents the number of buckets, makes it an attractive choice for datasets with a known range and uniform distribution. However, its performance heavily relies on the distribution of the data and the choice of bucket size, which must be carefully considered to achieve optimal results. The inclusion of Insertion Sort for sorting individual buckets further illustrates the hybrid nature of Bucket Sort, combining simplicity with efficiency.

Through the practical implementation and analysis of Bucket Sort and other sorting algorithms, this study highlights the trade-offs between various approaches. Comparison-based algorithms like Quick Sort and Merge Sort offer robust performance across a range of scenarios but may have limitations in terms of worst-case performance or additional space requirements. Non-comparison-based algorithms such as Radix Sort and Bucket Sort, while often faster for specific types of data, require careful consideration of their suitability based on data distribution and size.

Ultimately, understanding the strengths and limitations of each sorting algorithm allows for informed decision-making, ensuring that the chosen algorithm aligns with the specific requirements of the task at hand. This study serves as a valuable resource for both practitioners and researchers, providing insights into the optimization of sorting processes and contributing to more efficient data management practices. Future research and practical applications can benefit from these findings by exploring more advanced algorithms and optimization techniques tailored to evolving computational needs.

**REFERENCES:**

1. A New Algorithm for Sorting Data" by J. W. J. Williams
2. Experimental Analysis of Algorithms" by Robert Sedgewick
3. A Survey of Sorting Algorithms" by M. T. Goodrich and R. Tamassia
4. "Introduction to Algorithms**"** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
5. "The Art of Computer Programming, Volume 3: Sorting and Searching" by Donald E. Knuth