|  |  |  |  |
| --- | --- | --- | --- |
| **Academic Year** | **2025 - 26** | **Experiment No.** | **1** |
| **Course & Semester** | **S.E. – Sem. III** | **Subject Name** | **Analysis of Algorithm** |
| **Experiment Type** | **Software Performance** | **Subject Code** | **25PCC12CS05** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Name of Student:** | Atharva Dharmendra Jagtap | **Roll No.:** | 10937 |
| **Date of Performance:** |  | **Date of Submission:** |  |
| **LO Mapping** | 25PCC12CS05: Analyze the time and space complexity of algorithms | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Indicator** | **Poor** | **Average** | **Good** | | Timeline Maintains submission deadline (3) | Submission not done (0) | One or More than One week late (1-2) | Maintains deadline (3) | | Completion and Organization (3) | N/A | Document is just acceptable (1-2) | Completed whole document and neatly organized (3) | | Program Performance (2) | Could not perform at all (0) | Implemented few parts (1) | Full implementation (2) | | Knowledge In depth knowledge of the Experiment (2) | Unable to answer questions (0) | Unable to answer few questions (1) | Able to answer all questions (2) | |
| **Assessment Marks:**   |  |  | | --- | --- | | Timeline |  | | Completion and Organization |  | | Program Performance |  | | Knowledge |  | |
| Total: (Out of 10) |
| Teacher’s Sign: Student Sign: |

**Experiment No. 1**

**AIM:** To Implement and analyze time and space complexity of Modified bubble, Insertion and Selection sort to display exam result of students based on their total marks scored

**THEORY:** Sorting is one of the most fundamental operations in computer science and plays a key role in organizing data efficiently for better readability, accessibility, and analysis. In this experiment, we explore three classic sorting algorithms—Modified Bubble Sort, Insertion Sort, and Selection Sort—to arrange student records in descending order based on their total marks.

Each sorting algorithm has its own strategy, time complexity, and space usage characteristics. By understanding and analyzing these, we can evaluate which algorithm performs best under different circumstances, such as small vs. large datasets.

1. Modified Bubble Sort:

Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The modified version optimizes the process by adding a flag to stop the algorithm early if the array becomes sorted before all passes are complete.

* Best Case Time Complexity: O(n) — when the array is already sorted.
* Average and Worst Case Time Complexity: O(n²)
* Space Complexity: O(1) — in-place sorting.

2. Insertion Sort:

Insertion Sort builds the sorted array one element at a time by comparing each new element to those already sorted and inserting it into the correct position. It's very efficient for small or nearly sorted datasets.

* Best Case Time Complexity: O(n)
* Average and Worst Case Time Complexity: O(n²)
* Space Complexity: O(1) — in-place sorting.

3. Selection Sort:

Selection Sort divides the input list into a sorted and unsorted region. It repeatedly selects the minimum (or maximum, for descending order) element from the unsorted region and swaps it with the leftmost unsorted element.

* Best, Average, and Worst Case Time Complexity: O(n²) — due to the nested loop structure.
* Space Complexity: O(1)

Relevance to Student Exam Results:

In the context of this experiment, each student record contains attributes such as name, roll number, and total marks. The objective is to sort the list of students in descending order based on their marks. This mimics real-world scenarios like generating merit lists, cut-off rankings, or competitive results.

By comparing the three sorting algorithms in terms of time and space complexity, especially on varying sizes of datasets, we can make informed decisions about which algorithm is more suitable for a given application.

**ALGORITHM:**

Step 1: Start

Step 2: Declare a structure

* Define a structure Student with two fields:
  + name (string)
  + marks (integer)

Step 3: Input number of students

* Read the total number of students n.

Step 4: Input student data

* For each student from 1 to n:
  + Input the student's name.
  + Input the student's total marks.
  + Store this data in an array students[].

Step 5: Copy data

* Copy the students[] array into a temporary array tempArr[] to preserve original data.

Step 6: Display sorting menu

* Display the following sorting options:
  1. Modified Bubble Sort
  2. Insertion Sort
  3. Selection Sort

Step 7: Read user’s choice

* Read the user's choice and perform the corresponding sorting algorithm on tempArr[]:
  + If choice = 1, use Modified Bubble Sort to sort in descending order.
  + If choice = 2, use Insertion Sort to sort in descending order.
  + If choice = 3, use Selection Sort to sort in descending order.
  + Else, display an invalid choice message and terminate the program.

Step 8: Display sorted results

* Print the sorted list of students with their rank, name, and marks.

Step 9: End

**CODE:**

#include <stdio.h>

struct Student {

    char name[50];

    int marks;

};

int main() {

    int n, choice;

    printf("Enter the number of students: ");

    scanf("%d", &n);

    struct Student students[n], tempArr[n];

    for (int i = 0; i < n; i++) {

        printf("Enter name of student %d: ", i + 1);

        scanf("%49s", students[i].name);

        printf("Enter marks of student %d: ", i + 1);

        scanf("%d", &students[i].marks);

    }

    for (int i = 0; i < n; i++) {

        tempArr[i] = students[i];

    }

    printf("\nChoose sorting method:\n");

    printf("1. Modified Bubble Sort\n");

    printf("2. Insertion Sort\n");

    printf("3. Selection Sort\n");

    printf("Enter your choice: ");

    scanf("%d", &choice);

    switch (choice) {

        case 1:

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

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

                    if (tempArr[j].marks < tempArr[j + 1].marks) {

                        struct Student temp = tempArr[j];

                        tempArr[j] = tempArr[j + 1];

                        tempArr[j + 1] = temp;

                    }

                }

            }

            break;

        case 2:

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

                struct Student key = tempArr[i];

                int j = i - 1;

                while (j >= 0 && tempArr[j].marks < key.marks) {

                    tempArr[j + 1] = tempArr[j];

                    j--;

                }

                tempArr[j + 1] = key;

            }

            break;

        case 3:

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

                int maxIndex = i;

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

                    if (tempArr[j].marks > tempArr[maxIndex].marks) {

                        maxIndex = j;

                    }

                }

                if (maxIndex != i) {

                    struct Student temp = tempArr[i];

                    tempArr[i] = tempArr[maxIndex];

                    tempArr[maxIndex] = temp;

                }

            }

            break;

        default:

            printf("Invalid choice. Exiting.\n");

            return 1;

    }

    printf("\nSorted Student Results (Descending Order):\n");

    printf("-----------------------------------------------------------\n");

    printf("|\tRank\t|\t  Name  \t|\tMarks\t|\n");

    printf("-----------------------------------------------------------\n");

    for (int i = 0; i < n; i++) {

        printf("|\t  %d  \t|\t %s   \t|\t%d  \t|\n", i + 1, tempArr[i].name, tempArr[i].marks);

    }

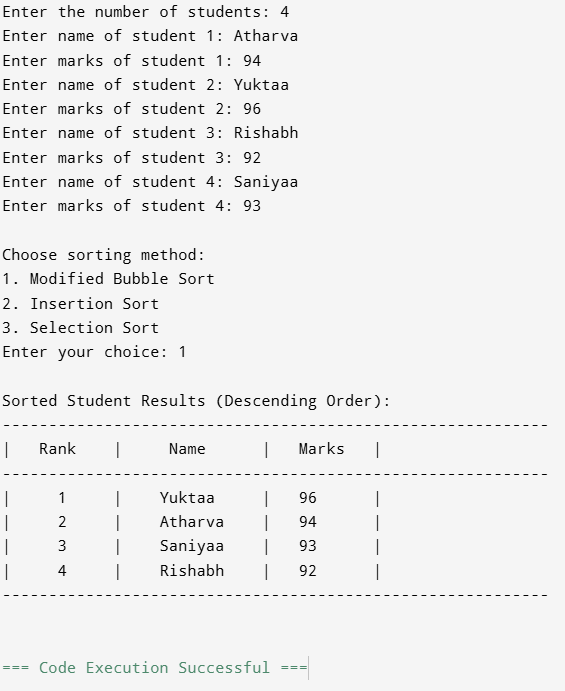
    printf("-----------------------------------------------------------\n");

    return 0;

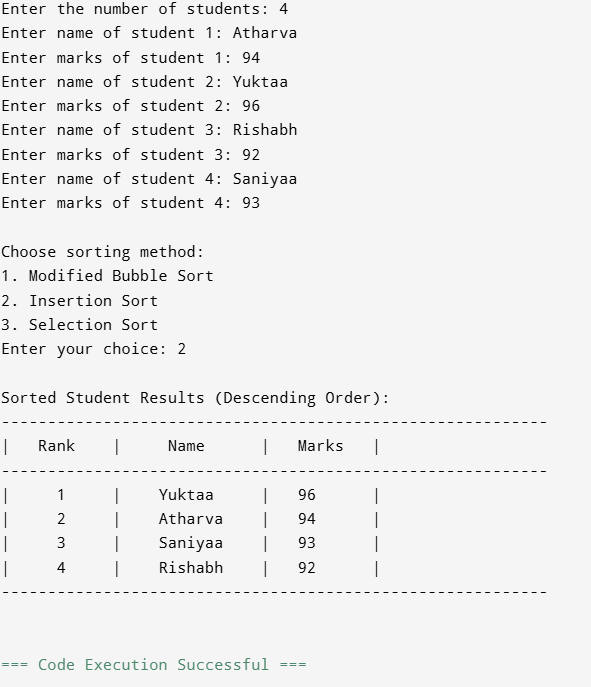
}

**OUTPUT:**

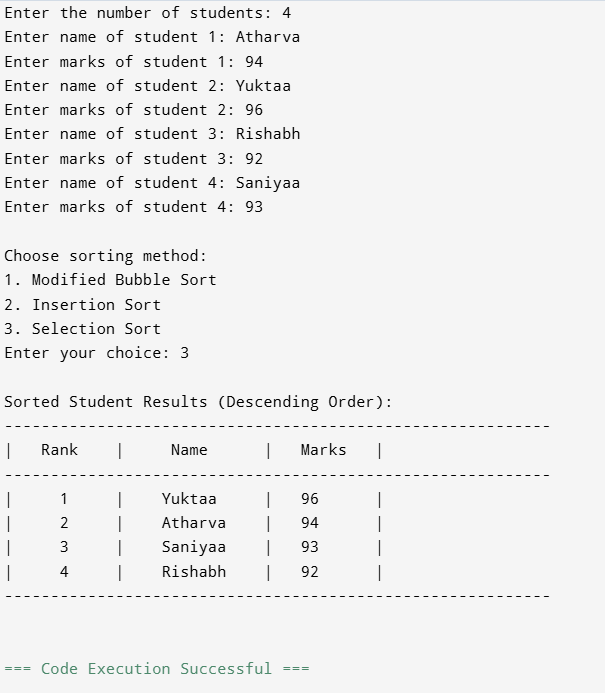
* + 1. Modified Bubble Sort

****

* + 1. Insertion Sort



* + 1. Selection Sort



**POST LAB QUESTIONS**

**1.** Between Insertion, Selection, and Modified Bubble Sort, which algorithm performs best for nearly sorted student data? Why?

**Ans.** Insertion Sort performs best for nearly sorted student data. Following are reasons why it is better than other sorting algorithms:

1. Insertion Sort has O(n) complexity for nearly sorted data, as it requires minimal shifting.
2. Selection Sort remains O(n²) regardless of order.
3. Modified Bubble Sort improves slightly but still averages O(n²), making it less efficient than Insertion Sort.

**2.** What trade-offs might arise if you switched from these basic sorting algorithms to more advanced ones like Merge Sort or Quick Sort?

**Ans.** Switching to advanced algorithms like Merge Sort or Quick Sort introduces these trade-offs:

1. Performance
   * Better average/worst-case time: O(n log n) vs. O(n²).
   * Less efficient for small or nearly sorted datasets compared to Insertion Sort.
2. Space Complexity
   * Merge Sort requires O(n) extra space; Quick Sort works in-place but risks stack usage for recursion.
3. Implementation Complexity
   * More complex to implement and debug than basic sorts.
4. Stability
   * Merge Sort is stable; Quick Sort isn’t unless modified.
5. Cache Performance
   * Quick Sort often has better cache locality, while Merge Sort may incur overhead due to extra memory usage.

**3.** Imagine a school wants to publish student rankings on its notice board and website. These rankings must be updated weekly based on newly added exam scores.

Question:

Which of the implemented sorting algorithms—Modified Bubble, Insertion, or Selection Sort—would be the most efficient for updating the rankings each week if only a small number of students’ scores change frequently? Justify your answer based on algorithm behavior and complexity.

**Ans.** Insertion Sort would be the most efficient for updating the rankings each week if only a small number of students’ scores change frequently because of:

1. Adaptiveness: Runs in O(n) when the list is almost sorted; only elements that changed get shifted locally.
2. Low overhead: Single pass with local shifts; minimal swaps vs. Bubble’s repeated passes.
3. Selection Sort: Always O(n²) regardless of order-wastes work.
4. Modified Bubble: Best case O(n) with early-exit, but still requires multiple passes to confirm order; slower than targeted insertions.

**CONCLUSION:**

In summary, among Modified Bubble Sort, Selection Sort, and Insertion Sort, **Insertion Sort** is the most efficient for handling nearly sorted or gradually changing data, such as weekly student ranking updates. It excels in such cases by quickly managing already ordered sections of the list. **Modified Bubble Sort** improves over the basic version with early-exit detection but still performs unnecessary comparisons. **Selection Sort**, while straightforward, is the least efficient as it doesn’t take advantage of existing order. Thus, **Insertion Sort** remains the optimal choice for frequent, small updates.