OBE IMPLEMENTATION:COURSE REFERENCE

BY :

T . NAGASRI (AP23110010715 )

M . REETHU PRIYA ( AP23110011128 )

RAMARAJU RUDRARAJU ( AP23110011097 )

V . SRIDHAR VALLEPU ( AP23110011127 )

A report for the CS204:Design and Analysis of Algorithm project



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SRM UNIVERSITY AP::AMARAVATI

INDEX

Introduction

Project Modules:

Architecture Diagram

Module Description

Programming Details naming conventions to be used:

Field/table details

Algorithm Details

(i)Sorting

(ii)Searching

(ii) Storing the details in a text file

Source Code

Comparison of Sorting Algorithms

Comparison of Searching Algorithms

ScreenShots

Conclusion

Introduction

Our University (herewith considered as SRM-AP) is going to implement OBE(Outcome Based Education) in their university and you assigned in the project to develop an application with any programming Language you are well versed and you were supposed to do searching and sorting using learned algorithms,comparing your sorting algorithm with any one of existing algorithm,displaying the time complexity of both algorithms and explaining advantages and disadvantages of the algorithm.

Project Modules:

Various Modules available in the project are

1.Blooms Level setting

2.Program Level Objective Setting

3.University

4.Schools

5.Department

6.Programs

7.Courses

8.Course objective setting

9.Course Outcome Setting

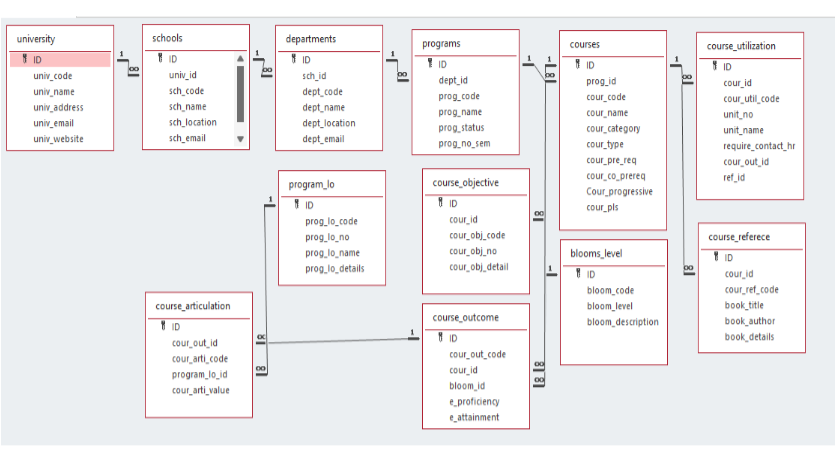
10.Course Articulation matrix Setting

11.Course Utilization Setting

12.Course Reference Setting.



ARCHITECTURAL DIAGRAM :



Module Description

Module Name: COURSE REFERENCE

Module Description:

This module is used to create, Update, Retrieve, Delete details of the module and storing the details in the text file , you have to provide option for searching and sorting of fields mentioned below according to algorithms given for you

Programming Details naming conventions to be used:

● File name: course reference\_name

● Function/method name

1. ○ Create: course reference\_create
2. ○ Update: course reference\_update
3. ○ Retrieve: course reference\_retrive
4. ○ Delete: course reference\_delete
5. ○ Sorting: course reference\_name
6. ○ Searching: course reference\_name
7. ○ Storing: course reference \_storing
8. ○ Comparison (both searching and Sorting ):

■ For Searching-

Course reference\_Compare\_Search\_linear search

Course reference\_Compare\_Search\_binary search

■ For Sorting-

Course reference\_Compare\_sorting\_bubble sort

Course reference\_Compare\_sorting\_selection sort

○ Time Complexity(both searching and Sorting):

■ For Searching- Course reference \_complexity\_Search

■ For Sorting- Course reference \_compexity\_sorting

○ Algorithm Details(both searching and Sorting):

■ For Searching-

Course reference\_algo\_linear search

Course reference\_algo\_binary search

■ For Sorting-

Course reference\_algo\_bubble sort

Course reference\_algo\_selection sort

● File name - File name to be used is course reference\_setting.txt

Field/table details:

Field Name Data type

Id integer

Course reference \_code String

Course reference \_name String

Course reference \_address String

Course reference \_email String

Course reference \_website String

Algorithm Details:

(i)Sorting

Bubble Sort and Selection Sort are both simple comparison-based sorting algorithms. They are often used for educational purposes to illustrate basic sorting concepts. Here's a comparison between the two:

1. Algorithm Overview

Bubble Sort: This algorithm repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Selection Sort: This algorithm divides the list into a sorted and an unsorted region. It repeatedly selects the smallest (or largest) element from the unsorted region and moves it to the end of the sorted region.

2. Time Complexity

Bubble Sort:

Best Case: (O(n)) (when the array is already sorted)

Average Case: (O(n^2))

Worst Case: (O(n^2))

Selection Sort:

Best Case: (O(n^2))

Average Case: (O(n^2))

Worst Case: (O(n^2))

3. Space Complexity

Both Bubble Sort and Selection Sort have a space complexity of (O(1)) since they are in-place sorting algorithms and do not require additional storage proportional to the input size.

4. Stability

Bubble Sort: It is a stable sort, meaning that it maintains the relative order of equal elements.

Selection Sort: It is not stable by default, as it can change the relative order of equal elements during the selection process.

5. Implementation Complexity

Both algorithms are easy to implement, but Bubble Sort is generally considered simpler due to its straightforward swapping mechanism.

6. Performance on Small Data Sets

Both algorithms perform poorly on large datasets, but they can be acceptable for small datasets (typically (n < 10)) due to their simplicity.

7. Number of Swaps

Bubble Sort: May perform many swaps, especially if the list is in reverse order, as it swaps adjacent elements.

Selection Sort: Performs fewer swaps overall, as it only swaps once per iteration to place the selected element in its correct position.

8. Use Cases

Both sorting algorithms are rarely used in practice for large datasets due to their inefficiency. However, they are often taught as introductory algorithms for understanding sorting concepts.

Summary

Bubble Sort is better suited for educational purposes and small datasets due to its simplicity and stability.

Selection Sort is generally faster in terms of the number of swaps, but it is not stable and has a consistently poor performance on larger datasets.

(ii)Searching

Linear Search and Binary Search are two fundamental searching algorithms used to find an element in a list or array. They differ significantly in terms of efficiency, implementation, and use cases. Below is a detailed comparison between the two:

Linear Search Binary Search

1 ALOGORITHM OVERVIEW

This element in the list sequentially until the desired element is found or the end of the list is reached. Divides the sorted list in half repeatedly to find the desired element, comparing it with the middle element. It works on both sorted and unsorted lists . It basically requires a sorted list or array.

2 Time Complexity

LINEAR SEARCH

Best Case: (O(1))

Average Case: (O(n))

Worst Case: (O(n))

BINARY SEARCH

Best Case: (O(1))

Average Case: (O(\log n))

Worst Case: (O(\log n))

3 Space Complexity

(O(1))

(O(1)) or (O(\log n))

4 Stability

Not applicable FOR BOTH

5 Implementation Complexity

Simple to implement, straightforward logic. More complex than linear search due to the need for sorting and recursive or iterative logic.

Use Cases

Suitable for small or unsorted datasets; when the dataset is not large or performance is not a concern. Efficient for large sorted datasets where quick search times are required.

Performance

Slower for large datasets due to linear traversal. Much faster for large datasets due to logarithmic reduction in search space.

Best Practices

Use when simplicity is more important than performance, or when dealing with small or unsorted lists. Use when dealing with large, sorted datasets for optimal performance.

Summary of Key Differences:

Efficiency: Binary Search is significantly more efficient than Linear Search for large datasets, but it requires the data to be sorted.

Data Structure Requirements: Linear Search can work with any list (sorted or unsorted), while Binary Search only works with sorted lists.

Complexity: Linear Search is simpler and easier to implement, while Binary Search requires more complex logic due to its divide-and-conquer approach.

Conclusion

In summary, Linear Search is a straightforward algorithm suitable for small or unsorted datasets, while Binary Search is a much more efficient algorithm for searching in large, sorted datasets. The choice between the two depends on the specific requirements of the application, including the size of the dataset and whether it is sorted. (ii) Storing the details in a text file

● Storing the details in the text file once details are entered.

● Delete the detail from the text file once details are deleted.

● Update the text file once details are updated.

SOURCE CODE

#include <stdio.h>

#include <string.h>

#define MAX\_COURSES 100

typedef struct {

int ID;

char cour\_id[20];

char course\_ref\_code[20];

char book\_title[100];

char book\_author[100];

char book\_details[200];

} CourseReference;

int linearSearch(CourseReference courses[], int size, int targetID);

int binarySearch(CourseReference courses[], int size, int targetID);

void bubbleSort(CourseReference courses[], int size);

void selectionSort(CourseReference courses[], int size);

void printCourses(CourseReference courses[], int size);

int main() {

CourseReference courses[MAX\_COURSES] = {

{1, "CSE101", "REF001", "Introduction to C", "Author A", "Details about C programming."},

{3, "CSE103", "REF003", "Algorithms", "Author C", "Details about algorithms."},

{2, "CSE102", "REF002", "Data Structures", "Author B", "Details about data structures."},

{5, "CSE105", "REF005", "Database Systems", "Author E", "Details about databases."},

{4, "CSE104", "REF004", "Operating Systems", "Author D", "Details about operating systems."}

};

int size = 5; // Number of courses

int targetID = 3; // ID to search for

int linearIndex = linearSearch(courses, size, targetID);

if (linearIndex != -1) {

printf("Linear Search: Course found at index %d\n", linearIndex);

} else {

printf("Linear Search: Course not found.\n");

}

bubbleSort(courses, size);

printf("\nCourses after Bubble Sort:\n");

printCourses(courses, size);

int binaryIndex = binarySearch(courses, size, targetID);

if (binaryIndex != -1) {

printf("Binary Search: Course found at index %d\n", binaryIndex);

} else {

printf("Binary Search: Course not found.\n");

}

selectionSort(courses, size);

printf("\nCourses after Selection Sort:\n");

printCourses(courses, size);

return 0;

}

int linearSearch(CourseReference courses[], int size, int targetID) {

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

if (courses[i].ID == targetID) {

return i; // Target found at index i

}

}

return -1; // Target not found

}

int binarySearch(CourseReference courses[], int size, int targetID) {

int low = 0;

int high = size - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (courses[mid].ID == targetID) {

return mid; // Target found at index mid

} else if (courses[mid].ID < targetID) {

low = mid + 1; // Search in the right half

} else {

high = mid - 1; // Search in the left half

}

}

return -1; // Target not found

}

void bubbleSort(CourseReference courses[], int size) {

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

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

if (courses[j].ID > courses[j + 1].ID) {

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

CourseReference temp = courses[j];

courses[j] = courses[j + 1];

courses[j + 1] = temp;

}

}

}

}

void selectionSort(CourseReference courses[], int size) {

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

int minIndex = i;

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

if (courses[j].ID < courses[minIndex].ID) {

minIndex = j;

}

}

CourseReference temp = courses[minIndex];

courses[minIndex] = courses[i];

courses[i] = temp;

}

}

void printCourses(CourseReference courses[], int size) {

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

printf("ID: %d, Course ID: %s, Reference Code: %s, Title: %s, Author: %s, Details: %s\n",

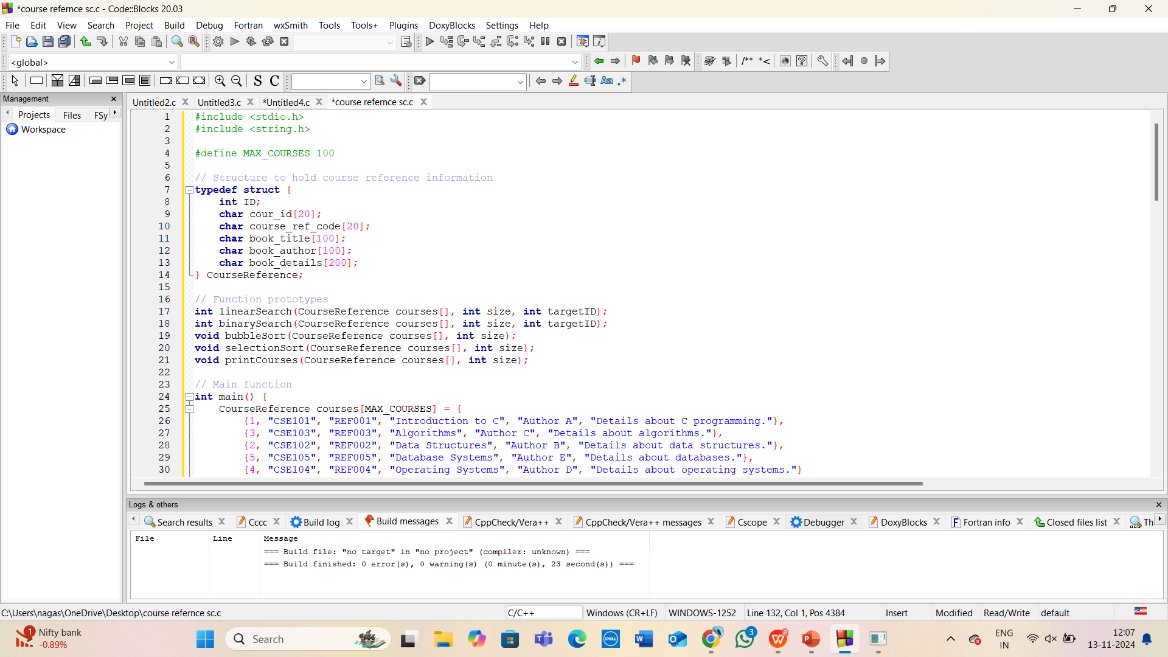
courses[i].ID, courses[i].cour\_id, courses[i].course\_ref\_code,

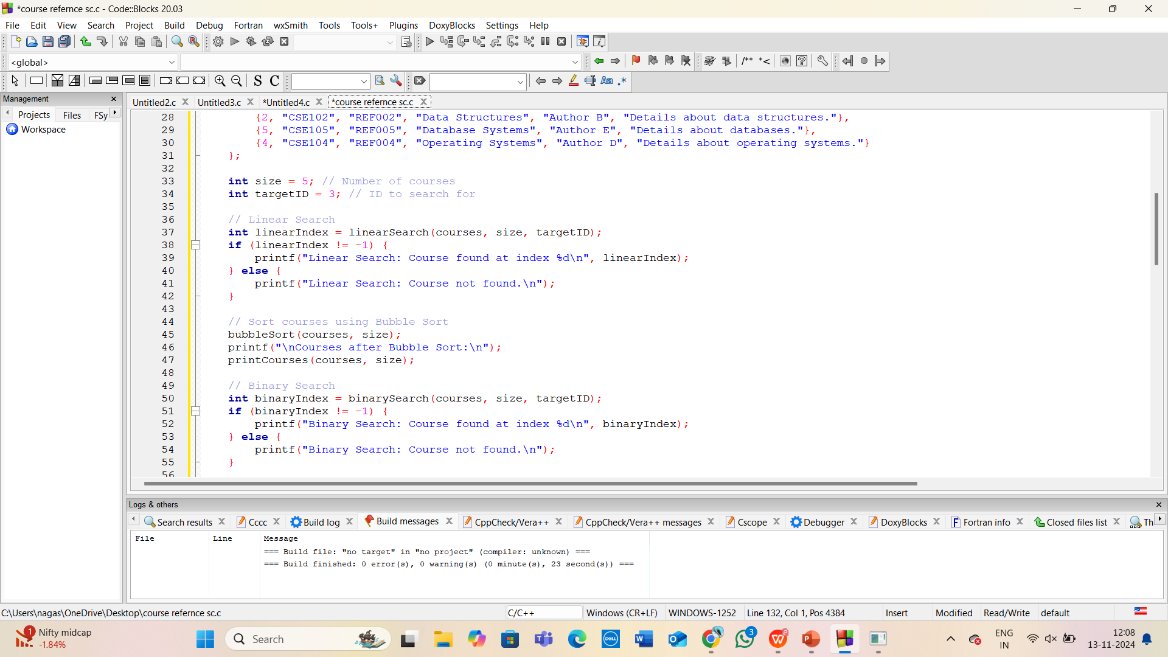
courses[i].book\_title, courses[i].book\_author, courses[i].book\_details);

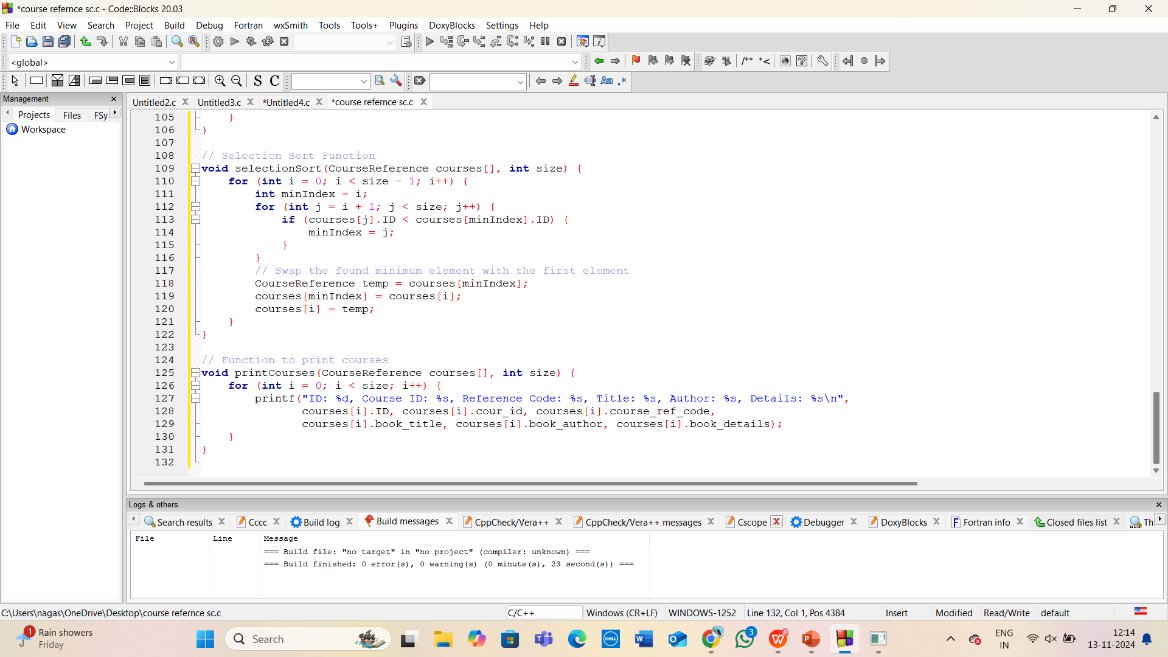
}

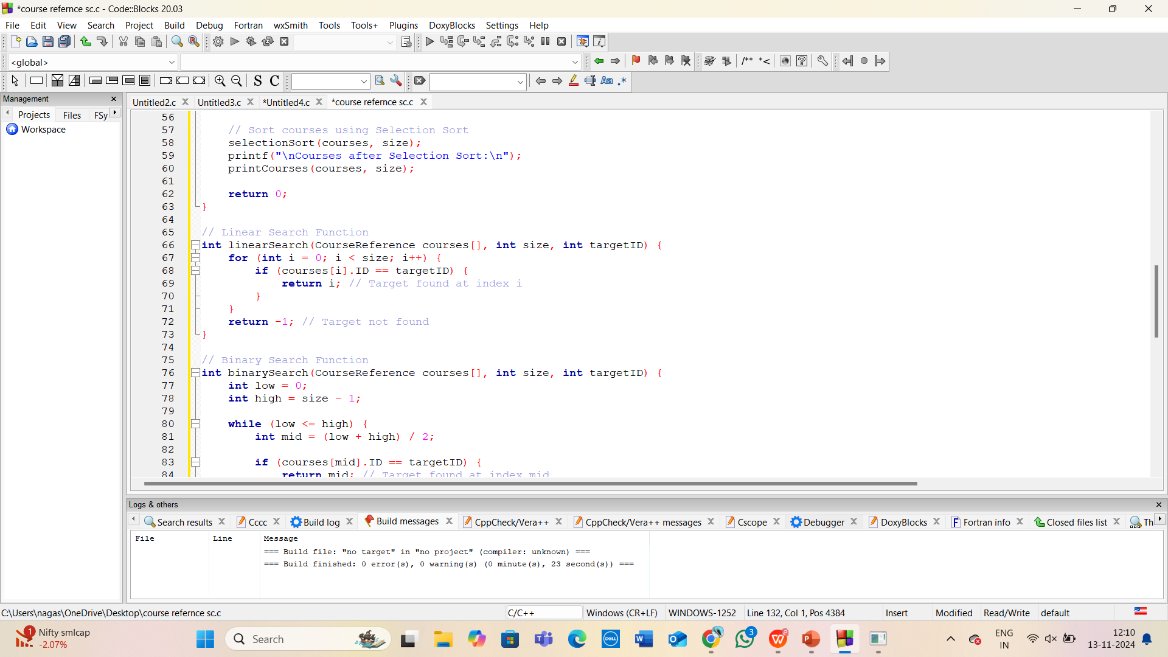
}

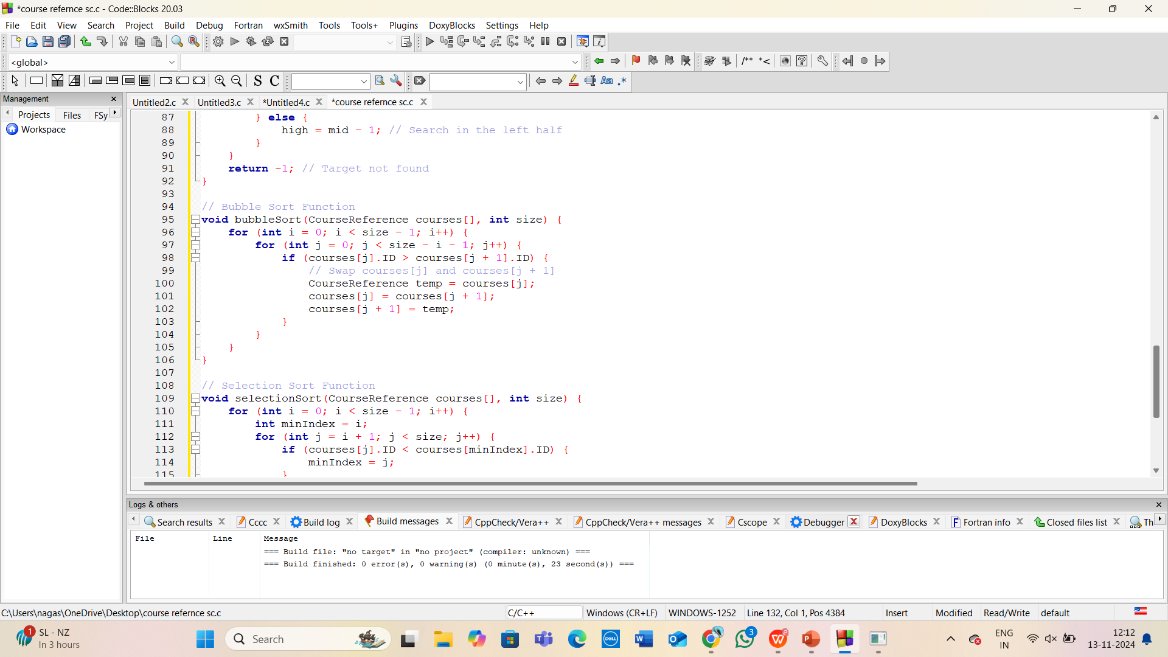
SCREENSHOTS



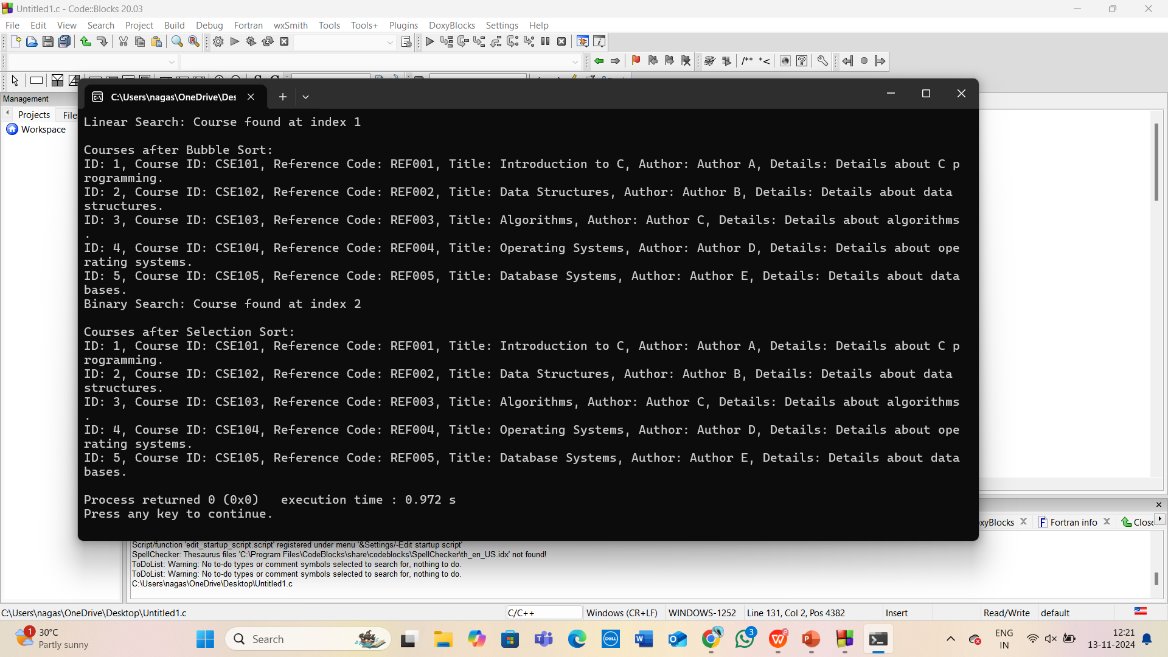








OUTPUT :



Conclusive difference between bubble and selection sort

#include <stdio.h>

#include <string.h>

#define MAX\_ENTRIES 5

typedef struct {

int ID;

int course\_id;

int course\_ref\_code;

char book\_title[100];

char book\_author[100];

char book\_details[200];

} CourseReference;

void bubbleSort(CourseReference data[], int n) {

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

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

if (data[j].course\_ref\_code > data[j + 1].course\_ref\_code) {

CourseReference temp = data[j];

data[j] = data[j + 1];

data[j + 1] = temp;

}

}

}

}

void selectionSort(CourseReference data[], int n) {

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

int minIndex = i;

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

if (data[j].course\_ref\_code < data[minIndex].course\_ref\_code) {

minIndex = j;

}

}

CourseReference temp = data[minIndex];

data[minIndex] = data[i];

data[i] = temp;

}

}

void printData(CourseReference data[], int n) {

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

printf("ID: %d, Course ID: %d, Course Ref Code: %d, Title: %s, Author: %s, Details: %s\n",

data[i].ID, data[i].course\_id, data[i].course\_ref\_code,

data[i].book\_title, data[i].book\_author, data[i].book\_details);

}

}

int main() {

CourseReference data[MAX\_ENTRIES] = {

{1, 101, 1200, "Data Structures", "Mark Weiss", "C Programming Reference"},

{2, 102, 1350, "Algorithms", "Thomas Cormen", "Comprehensive Algorithm Guide"},

{3, 103, 1100, "Operating Systems", "Abraham Silberschatz", "Modern OS Concepts"},

{4, 104, 1250, "Computer Networks", "Andrew Tanenbaum", "Fundamentals of Networking"},

{5, 105, 1400, "Database Systems", "Ramez Elmasri", "Database Management Systems"}

};

int n = MAX\_ENTRIES;

printf("Original Data:\n");

printData(data, n);

CourseReference bubbleSortedData[MAX\_ENTRIES];

memcpy(bubbleSortedData, data, sizeof(data)); // Copy original data

bubbleSort(bubbleSortedData, n);

printf("\nData after Bubble Sort:\n");

printData(bubbleSortedData, n);

CourseReference selectionSortedData[MAX\_ENTRIES];

memcpy(selectionSortedData, data, sizeof(data)); // Copy original data

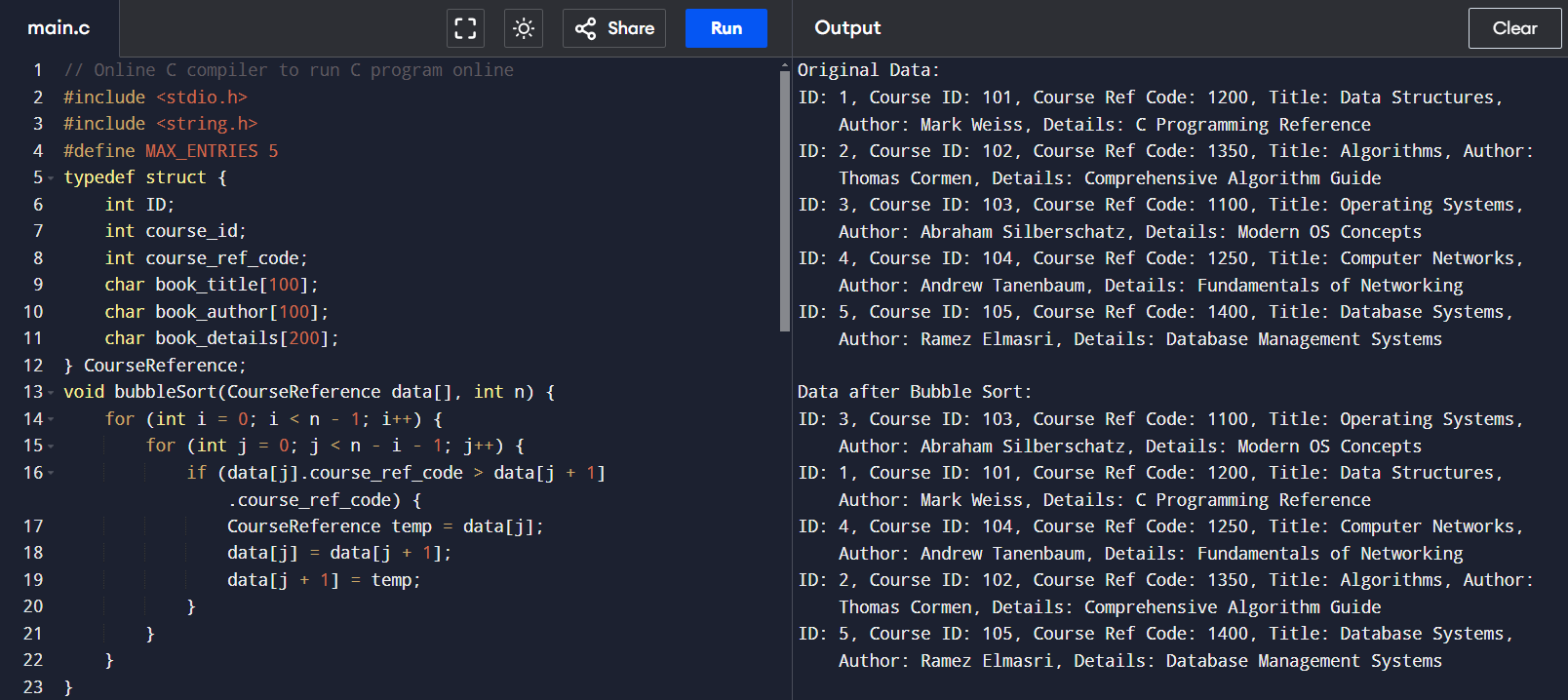
selectionSort(selectionSortedData, n);

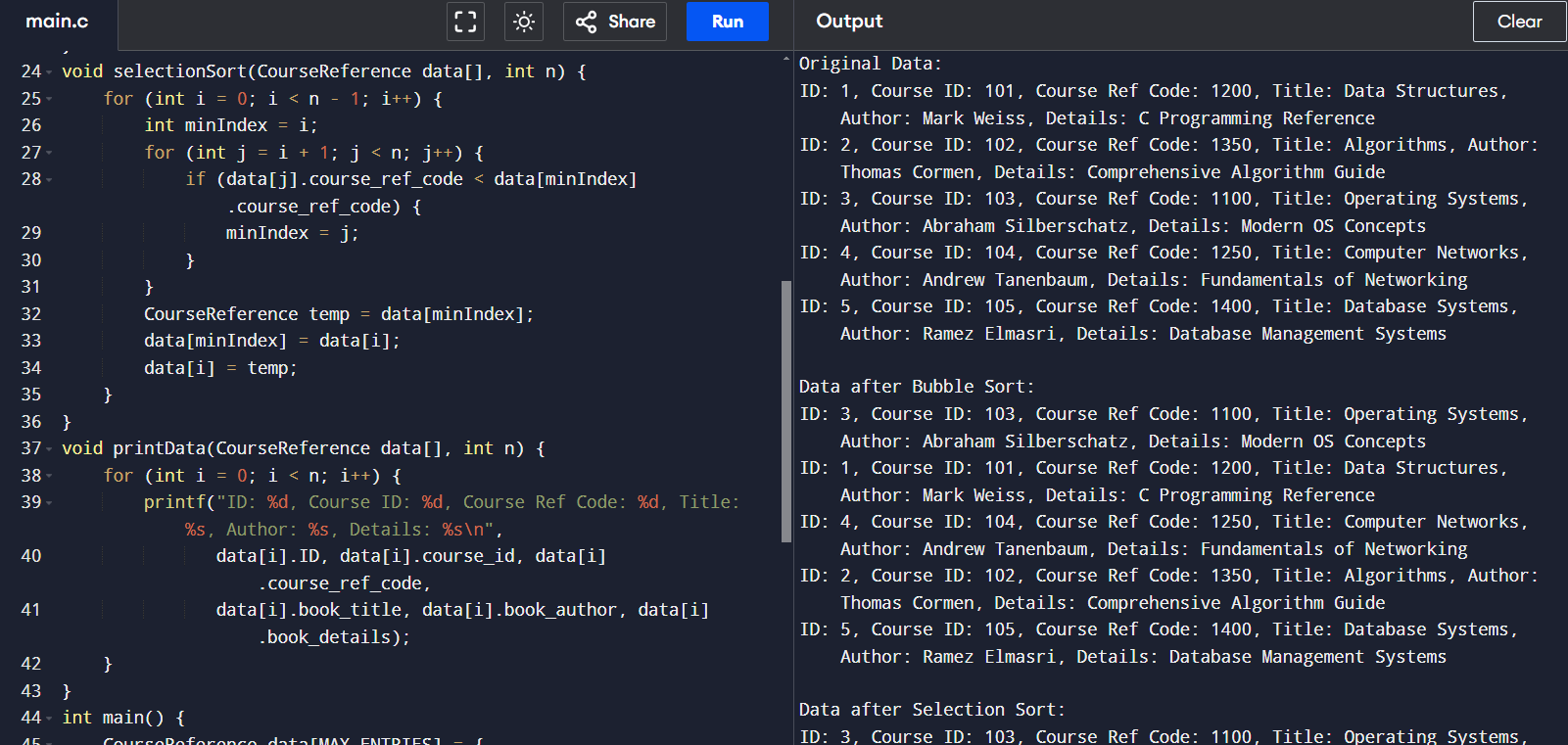
printf("\nData after Selection Sort:\n");

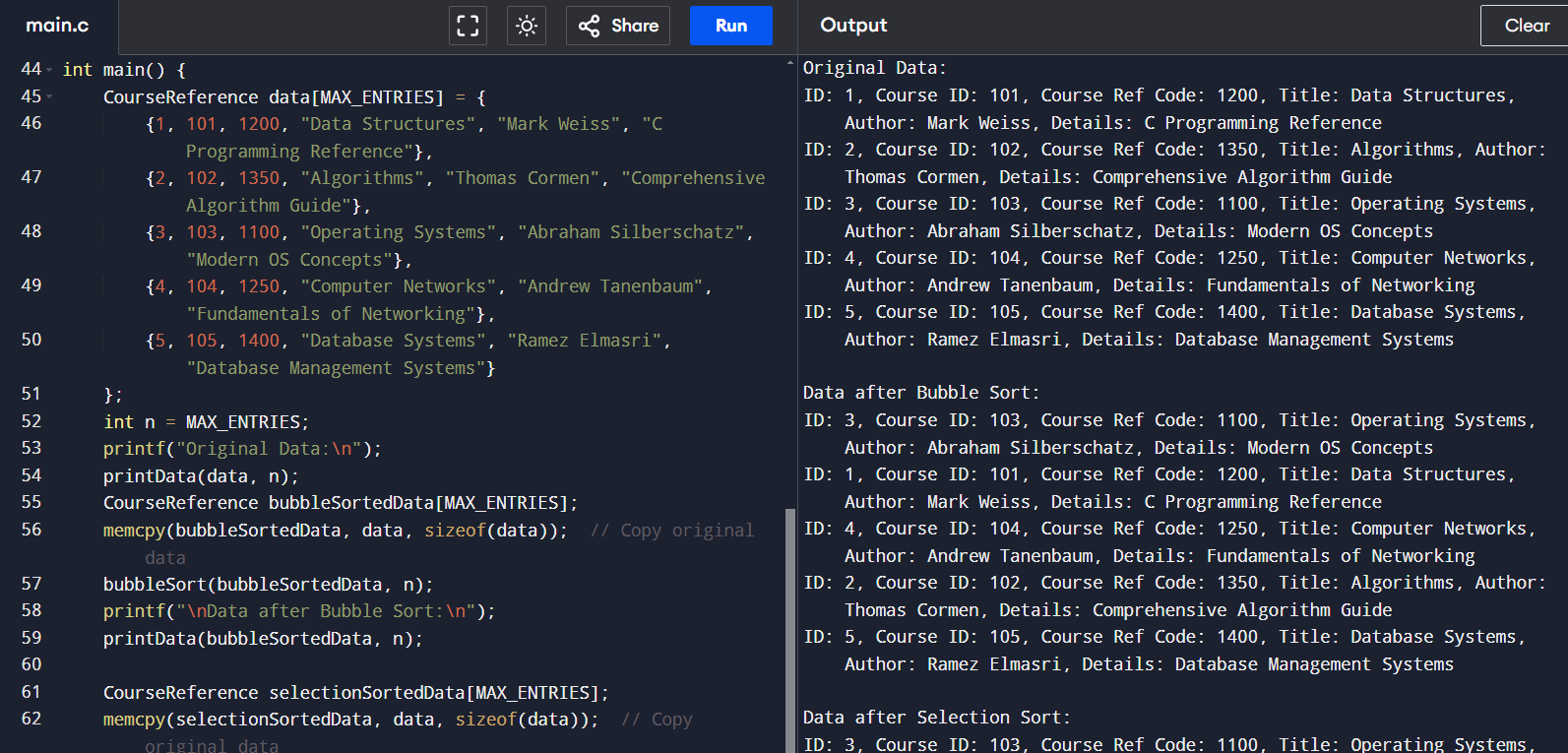
printData(selectionSortedData, n);

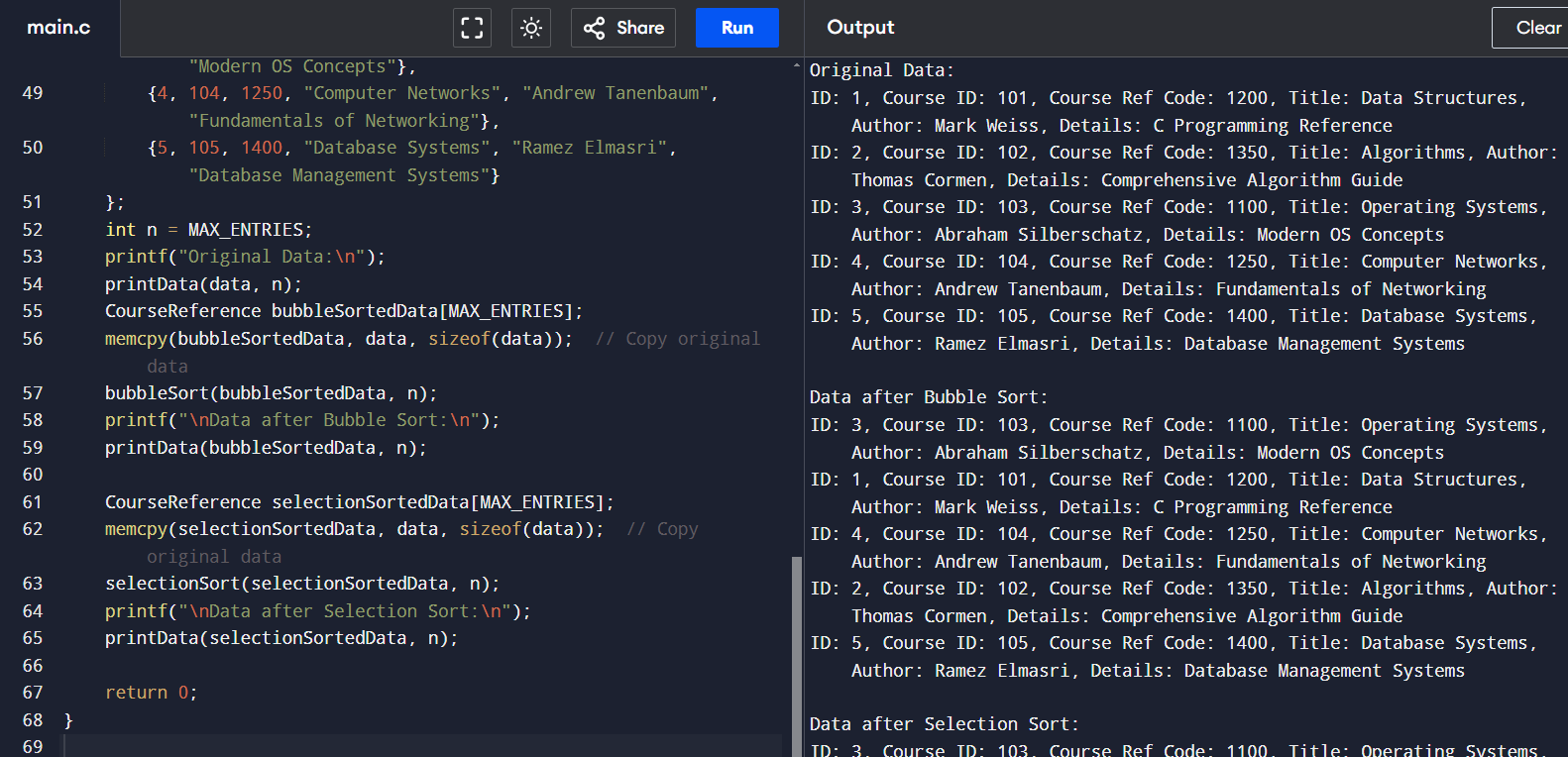
return 0;

}









CONCLUSIVE DIFFERENCE BETWEEN LINEAR AND BINARY SEARCH

#include <stdio.h>

#include <string.h>

#define MAX\_ENTRIES 100

typedef struct {

int ID;

int course\_id;

int course\_ref\_code;

char book\_title[100];

char book\_author[100];

char book\_details[200];

} CourseReference;

int linearSearch(CourseReference data[], int n, int target\_code);

int binarySearch(CourseReference data[], int n, int target\_code);

void sortByCourseRefCode(CourseReference data[], int n);

int main() {

CourseReference data[MAX\_ENTRIES] = {

{1, 101, 1200, "Data Structures", "Mark Weiss", "C Programming Reference"},

{2, 102, 1350, "Algorithms", "Thomas Cormen", "Comprehensive Algorithm Guide"},

{3, 103, 1100, "Operating Systems", "Abraham Silberschatz", "Modern OS Concepts"},

{4, 104, 1250, "Computer Networks", "Andrew Tanenbaum", "Fundamentals of Networking"},

{5, 105, 1400, "Database Systems", "Ramez Elmasri", "Database Management Systems"}

};

int n = 5;

int target\_code;

printf("Enter course reference code to search: ");

scanf("%d", &target\_code);

int index = linearSearch(data, n, target\_code);

if (index != -1) {

printf("Linear Search: Entry found at index %d\n", index);

printf("ID: %d, Course ID: %d, Course Ref Code: %d, Title: %s, Author: %s, Details: %s\n",

data[index].ID, data[index].course\_id, data[index].course\_ref\_code,

data[index].book\_title, data[index].book\_author, data[index].book\_details);

} else {

printf("Linear Search: Entry not found.\n");

}

sortByCourseRefCode(data, n);

index = binarySearch(data, n, target\_code);

if (index != -1) {

printf("Binary Search: Entry found at index %d\n", index);

printf("ID: %d, Course ID: %d, Course Ref Code: %d, Title: %s, Author: %s, Details: %s\n",

data[index].ID, data[index].course\_id, data[index].course\_ref\_code,

data[index].book\_title, data[index].book\_author, data[index].book\_details);

} else {

printf("Binary Search: Entry not found.\n");

}

return 0;

}

int linearSearch(CourseReference data[], int n, int target\_code) {

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

if (data[i].course\_ref\_code == target\_code) {

return i;

}

}

return -1;

}

int binarySearch(CourseReference data[], int n, int target\_code) {

int left = 0, right = n - 1;

while (left <= right) {

int mid = left + (right - left) / 2;

if (data[mid].course\_ref\_code == target\_code) {

return mid; // Return the index where match is found

} else if (data[mid].course\_ref\_code < target\_code) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return -1;

}

void sortByCourseRefCode(CourseReference data[], int n) {

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

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

if (data[j].course\_ref\_code > data[j + 1].course\_ref\_code) {

CourseReference temp = data[j];

data[j] = data[j + 1];

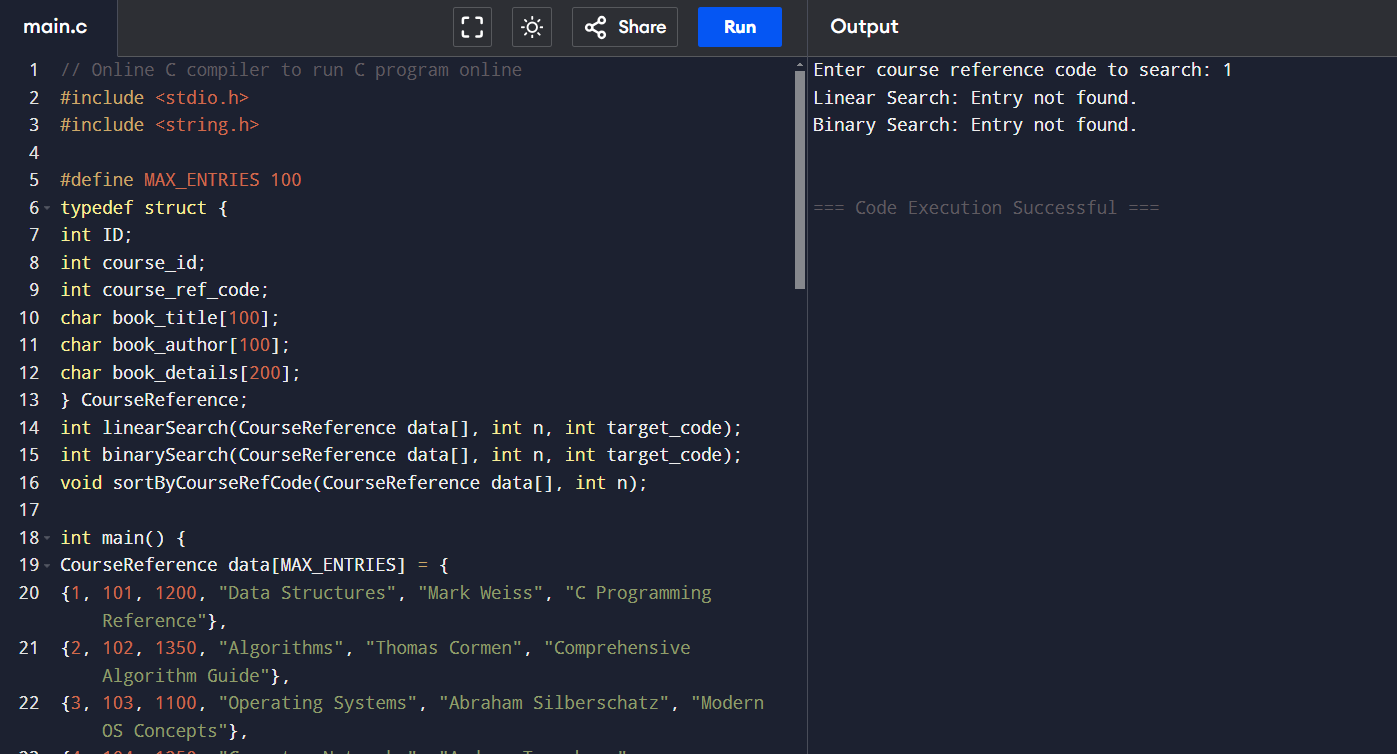
data[j + 1] = temp;

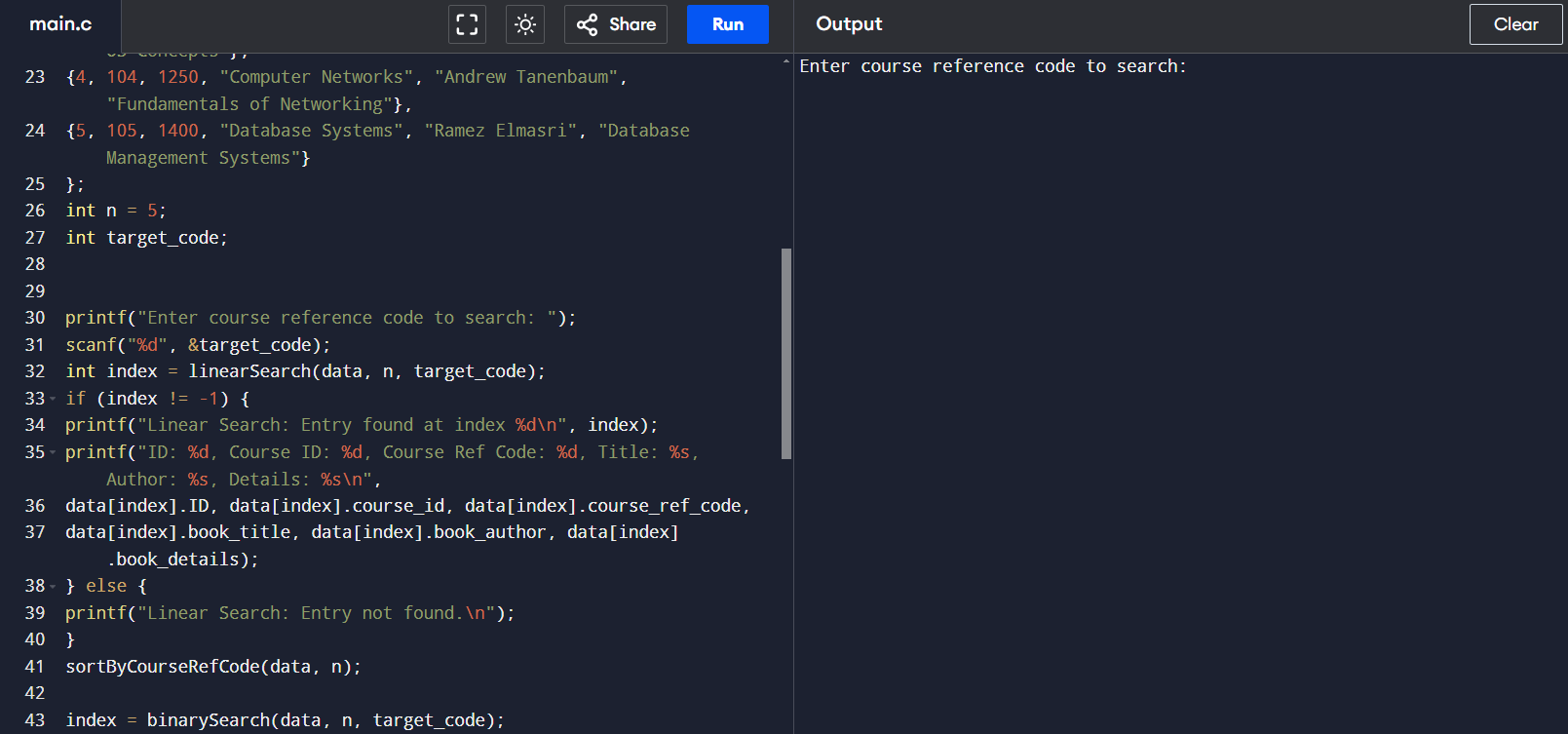
}

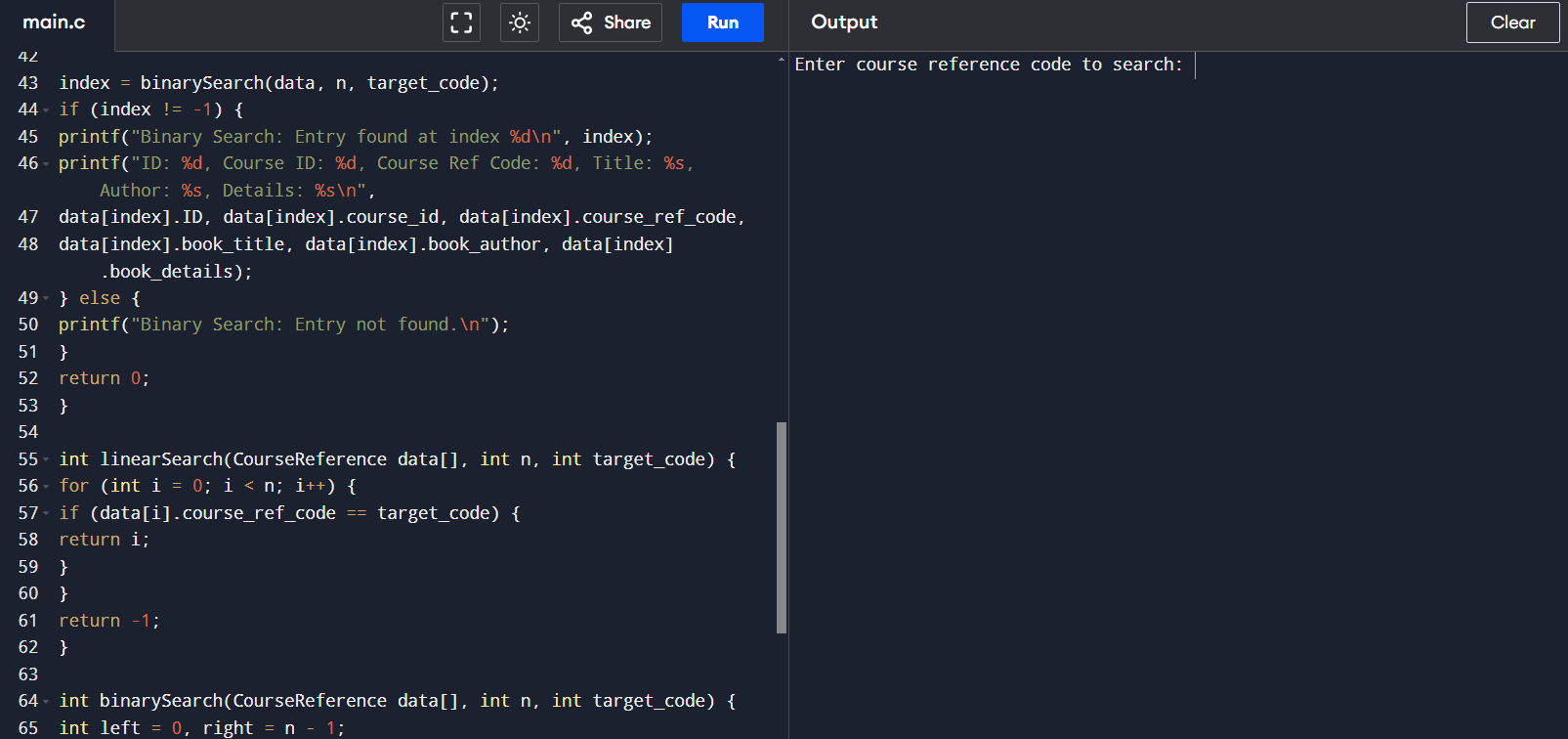
}

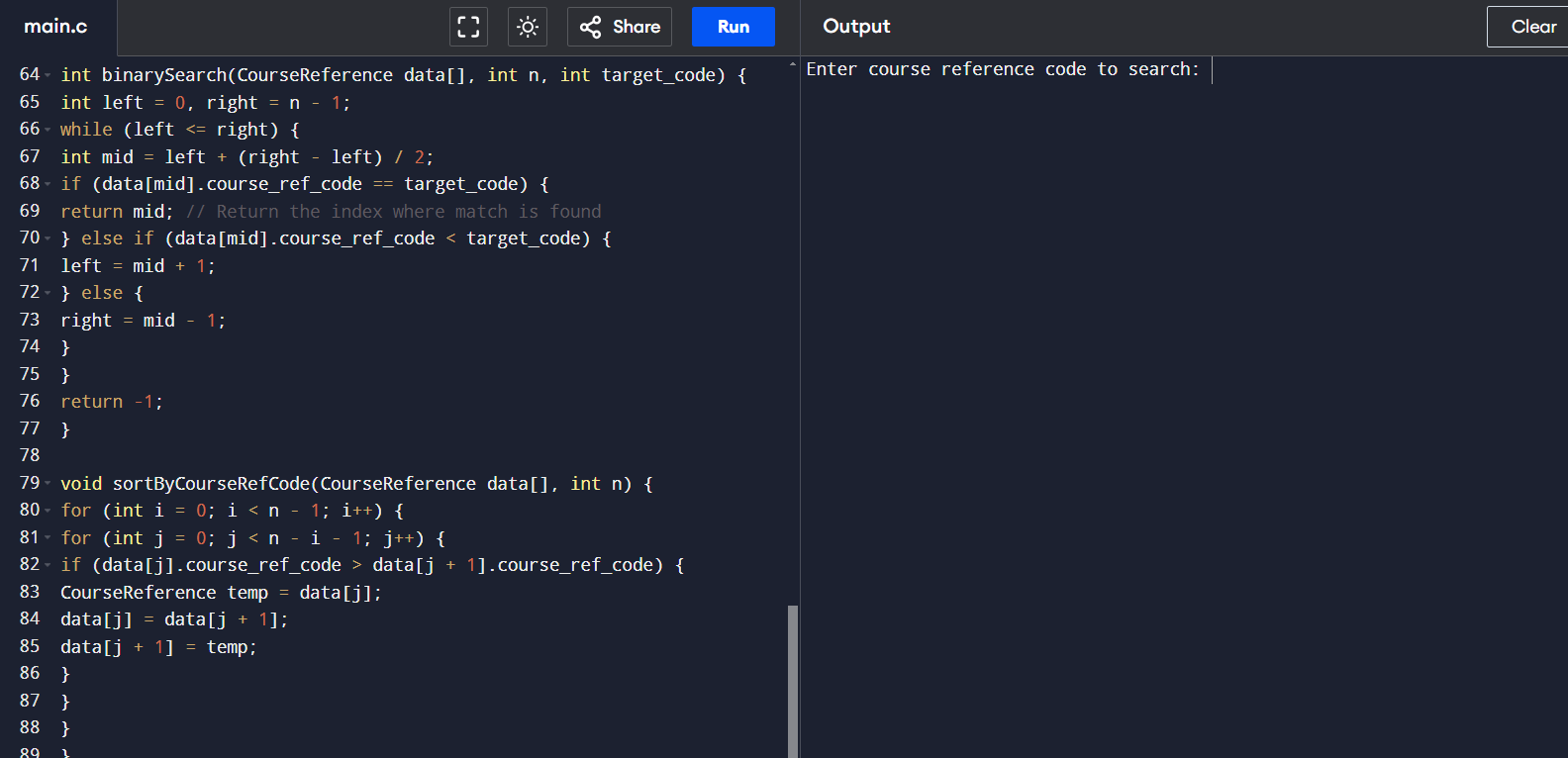
}

}









CONCLUSION :

In conclusion, **Bubble Sort**, **Selection Sort**, **Linear Search**, and **Binary Search** each offer unique advantages and limitations, making them suitable for different scenarios.

Bubble Sort and Selection Sort are straightforward, comparison-based algorithms with O(n^2) time complexity, best used for educational purposes or small datasets. Bubble Sort benefits from stability and early stopping on nearly sorted lists, while Selection Sort minimizes swaps, making it suitable when write operations need to be reduced.

In the realm of searching, Linear Search and Binary Search serve distinct purposes. Linear Search is versatile and works on unsorted lists, though it becomes inefficient with large datasets due to its O(n) complexity. In contrast, Binary Search, with its O(logn) efficiency, is ideal for quickly locating items in sorted lists, making it the preferred choice for performance-critical applications involving large, ordered datasets.

Each algorithm shines in specific contexts, emphasizing the importance of choosing the right approach based on data size, ordering, and performance needs.