**Structure to save Student info And find Topper:**

#include <stdio.h>

// Structure to store student information

struct Student {

char name[50];

float marks[3];

};

// Function to find the topper among the students

void findTopper(struct Student students[], int n) {

float max\_marks = 0;

int topper\_index = -1;

// Loop through all students

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

// Calculate total marks for each student

float total\_marks = 0;

for (int j = 0; j < 3; j++) {

total\_marks += students[i].marks[j];

}

// Update topper information if total marks are higher

if (total\_marks > max\_marks) {

max\_marks = total\_marks;

topper\_index = i;

}

}

// Display information of the topper

printf("Topper: %s\n", students[topper\_index].name);

printf("Marks in subject 1: %.2f\n", students[topper\_index].marks[0]);

printf("Marks in subject 2: %.2f\n", students[topper\_index].marks[1]);

printf("Marks in subject 3: %.2f\n", students[topper\_index].marks[2]);

}

int main() {

int n;

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

scanf("%d", &n);

// Array of structures to store information of multiple students

struct Student students[n];

// Input information for each student

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

printf("Enter details for student %d:\n", i + 1);

printf("Name: ");

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

printf("Marks in subject 1: ");

scanf("%f", &students[i].marks[0]);

printf("Marks in subject 2: ");

scanf("%f", &students[i].marks[1]);

printf("Marks in subject 3: ");

scanf("%f", &students[i].marks[2]);

}

// Find and display the topper among the students

findTopper(students, n);

return 0;

}

**ArmStrong Number :**

#include <stdio.h>

int isArmstrong(int number) {

int originalNumber, remainder, result = 0, n = 0;

originalNumber = number;

while (originalNumber != 0) {

originalNumber /= 10;

++n;

}

originalNumber = number;

while (originalNumber != 0) {

remainder = originalNumber % 10;

result += pow(remainder, n);

originalNumber /= 10;

}

if (result == number)

return 1;

else

return 0;

}

int main() {

printf("Armstrong numbers between 1 to 10000:\n");

for (int i = 1; i <= 10000; ++i) {

if (isArmstrong(i))

printf("%d\n", i);

}

return 0;

}

**Sum of Row of Matrix After dynamically allocated :**

#include <stdio.h>

#include <stdlib.h>

int main() {

int rows, cols;

// Input the number of rows and columns for the matrix

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

scanf("%d", &rows);

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

scanf("%d", &cols);

// Dynamically allocate memory for the matrix

int \*\*matrix = (int \*\*)malloc(rows \* sizeof(int \*));

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

matrix[i] = (int \*)malloc(cols \* sizeof(int));

}

// Fill the matrix with elements dynamically between 1 to 100

int count = 1;

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

for (int j = 0; j < cols; j++) {

matrix[i][j] = count++;

if (count > 100) {

// If count exceeds 100, reset it to 1

count = 1;

}

}

}

// Calculate and display the sum of each row of the matrix

printf("Sum of each row:\n");

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

int row\_sum = 0;

for (int j = 0; j < cols; j++) {

row\_sum += matrix[i][j];

}

printf("Row %d: %d\n", i + 1, row\_sum);

}

// Free dynamically allocated memory

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

free(matrix[i]);

}

free(matrix);

return 0;

}

Lab 2

**Ticket Counter using Queue:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_SIZE 100

// Structure to represent a queue

struct Queue {

int items[MAX\_SIZE];

int front;

int rear;

};

// Function to create an empty queue

struct Queue\* createQueue() {

struct Queue\* queue = (struct Queue\*)malloc(sizeof(struct Queue));

queue->front = -1;

queue->rear = -1;

return queue;

}

// Function to check if the queue is full

int isOverflow(struct Queue\* queue) {

return (queue->rear == MAX\_SIZE - 1);

}

// Function to check if the queue is empty

int isUnderflow(struct Queue\* queue) {

return (queue->front == -1 || queue->front > queue->rear);

}

// Function to add an element to the queue (enqueue)

void enqueue(struct Queue\* queue, int value) {

if (isOverflow(queue)) {

printf("Queue Overflow! Cannot add more members.\n");

return;

}

if (queue->front == -1)

queue->front = 0;

queue->rear++;

queue->items[queue->rear] = value;

printf("Person joined the queue. Position in queue: %d\n", queue->rear - queue->front + 1);

}

// Function to remove an element from the queue (dequeue)

void dequeue(struct Queue\* queue) {

if (isUnderflow(queue)) {

printf("Queue Underflow! No one in the queue.\n");

return;

}

printf("Person left the queue. Position in queue: %d\n", queue->rear - queue->front + 1);

queue->front++;

}

int main() {

struct Queue\* queue = createQueue();

int choice, person;

while (1) {

printf("\nMenu:\n");

printf("1. Join the queue\n");

printf("2. Leave the queue\n");

printf("3. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter person's ID to join the queue: ");

scanf("%d", &person);

enqueue(queue, person);

break;

case 2:

dequeue(queue);

break;

case 3:

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

free(queue);

exit(0);

default:

printf("Invalid choice! Please enter a valid option.\n");

}

}

return 0;

}

**Reverse String Using Stack :**

#include <stdio.h>

#include <string.h>

#define MAX 100

typedef struct {

char items[MAX];

int top;

} Stack;

void push(Stack \*s, char item) {

if (s->top == MAX - 1) {

printf("Stack is full\n");

} else {

s->items[++(s->top)] = item;

}

}

char pop(Stack \*s) {

if (s->top == -1) {

printf("Stack is empty\n");

return '\0';

} else {

return s->items[(s->top)--];

}

}

int isEmpty(Stack \*s) {

return s->top == -1;

}

int isFull(Stack \*s) {

return s->top == MAX - 1;

}

int main() {

Stack s;

s.top = -1;

char str[MAX];

int i;

printf("Input a string: ");

scanf("%s", str);

for (i = 0; i < strlen(str); i++) {

push(&s, str[i]);

}

for (i = 0; i < strlen(str); i++) {

str[i] = pop(&s);

}

printf("Reversed string is: %s\n", str);

return 0;

}

#include <stdio.h>

#include <string.h>

#define MAX 100

int top = -1;

char stack[MAX];

// Function to push an element onto the stack

void push(char item) {

if (isFull()) {

printf("Stack is full\n");

} else {

stack[++top] = item;

}

}

// Function to pop an element from the stack

char pop() {

if (isEmpty()) {

printf("Stack is empty\n");

return '\0'; // Return null character if stack is empty

} else {

return stack[top--];

}

}

// Function to check if the stack is empty

int isEmpty() {

return top == -1;

}

// Function to check if the stack is full

int isFull() {

return top == MAX - 1;

}

int main() {

char str[MAX], item;

int i;

printf("Input a string: ");

scanf("%s", str);

// Push all characters of the string into the stack

for (i = 0; i < strlen(str); i++) {

push(str[i]);

}

// Pop all characters from the stack and store them back in the string

for (i = 0; i < strlen(str); i++) {

str[i] = pop();

}

printf("Reversed string is: %s\n", str);

return 0;

}

Lab3  
  
**Insertion And Selection Sort:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define MAX 100

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

int steps = 0;

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

int key = arr[i];

int j = i - 1;

steps++;

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

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

j--;

steps++;

}

arr[j + 1] = key;

}

return steps;

}

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

int steps = 0;

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

int min\_idx = i;

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

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

min\_idx = j;

}

steps++;

}

int temp = arr[min\_idx];

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

arr[i] = temp;

steps++;

}

return steps;

}

int main() {

int arr[MAX];

srand(time(0));

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

arr[i] = rand() % 1000;

}

int arr\_copy[MAX];

memcpy(arr\_copy, arr, MAX \* sizeof(int));

printf("Insertion Sort Steps: %d\n", insertionSort(arr, MAX));

printf("Selection Sort Steps: %d\n", selectionSort(arr\_copy, MAX));

return 0;

}

Lab 4

**Merge Sort:**

#include <stdio.h>

int merge(int arr[], int left, int mid, int right) {

int i, j, k;

int swaps = 0;

int n1 = mid - left + 1;

int n2 = right - mid;

// Create temporary arrays

int L[n1], R[n2];

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

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

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

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

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

// Merge the temporary arrays back into arr[left..right]

i = 0;

j = 0;

k = left;

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

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

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

swaps += n1 - i; // Count number of swaps

}

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++;

}

return swaps;

}

int mergeSort(int arr[], int left, int right) {

int swaps = 0;

if (left < right) {

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

// Recursively sort first and second halves

swaps += mergeSort(arr, left, mid);

swaps += mergeSort(arr, mid + 1, right);

// Merge the sorted halves

swaps += merge(arr, left, mid, right);

}

return swaps;

}

int main() {

int arr[7];

printf("Enter 7 elements: ");

for (int i = 0; i < 7; i++)

scanf("%d", &arr[i]);

int swaps = mergeSort(arr, 0, 6);

printf("Sorted array: ");

for (int i = 0; i < 7; i++)

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

printf("\n");

printf("Number of swaps required: %d\n", swaps);

return 0;

}

**Quick Sort:**

#include <stdio.h>

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

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

int pivot = arr[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++) {

if (arr[j] < pivot) {

i++;

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

}

}

swap(&arr[i + 1], &arr[high]);

return (i + 1);

}

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

if (low < high) {

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

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

int main() {

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

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

quickSort(arr, 0, n - 1);

printf("Sorted array: \n");

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

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

}

printf("\n");

return 0;

}

Lab 5

**Min Heap:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_HEAP\_SIZE 100

struct MinHeap {

int \*heap;

int size;

int capacity;

};

struct MinHeap\* createMinHeap(int capacity) {

struct MinHeap\* minHeap = (struct MinHeap\*) malloc(sizeof(struct MinHeap));

minHeap->capacity = capacity;

minHeap->size = 0;

minHeap->heap = (int\*) malloc(capacity \* sizeof(int));

return minHeap;

}

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

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void minHeapify(struct MinHeap\* minHeap, int index) {

int smallest = index;

int left = 2 \* index + 1;

int right = 2 \* index + 2;

if (left < minHeap->size && minHeap->heap[left] < minHeap->heap[smallest])

smallest = left;

if (right < minHeap->size && minHeap->heap[right] < minHeap->heap[smallest])

smallest = right;

if (smallest != index) {

swap(&minHeap->heap[index], &minHeap->heap[smallest]);

minHeapify(minHeap, smallest);

}

}

void insert(struct MinHeap\* minHeap, int key) {

if (minHeap->size == minHeap->capacity) {

printf("Heap Overflow\n");

return;

}

int i = minHeap->size;

minHeap->heap[i] = key;

minHeap->size++;

while (i != 0 && minHeap->heap[(i-1)/2] > minHeap->heap[i]) {

swap(&minHeap->heap[i], &minHeap->heap[(i-1)/2]);

i = (i-1)/2;

}

}

int extractMin(struct MinHeap\* minHeap) {

if (minHeap->size <= 0)

return -1; // Heap Underflow

if (minHeap->size == 1) {

minHeap->size--;

return minHeap->heap[0];

}

int root = minHeap->heap[0];

minHeap->heap[0] = minHeap->heap[minHeap->size - 1];

minHeap->size--;

minHeapify(minHeap, 0);

return root;

}

void printHeap(struct MinHeap\* minHeap) {

printf("Min Heap: ");

for (int i = 0; i < minHeap->size; i++)

printf("%d ", minHeap->heap[i]);

printf("\n");

}

int main() {

struct MinHeap\* minHeap = createMinHeap(MAX\_HEAP\_SIZE);

int choice, key;

while (1) {

printf("\nMenu:\n");

printf("1. Insert\n");

printf("2. Extract Min\n");

printf("3. Print Heap\n");

printf("4. Exit\n");

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

printf("Enter key to insert: ");

scanf("%d", &key);

insert(minHeap, key);

break;

case 2:

printf("Extracted Min: %d\n", extractMin(minHeap));

break;

case 3:

printHeap(minHeap);

break;

case 4:

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

free(minHeap->heap);

free(minHeap);

exit(0);

default:

printf("Invalid choice! Please enter a valid option.\n");

}

}

return 0;

}

**Counting Sort:**

#include <stdio.h>

#include <stdlib.h>

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

int i, min = arr[0], max = arr[0];

// Find the minimum and maximum values in the array

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

if (arr[i] < min) {

min = arr[i];

}

if (arr[i] > max) {

max = arr[i];

}

}

int range = max - min + 1;

// Create and initialize the counting array

int \*count = (int \*)calloc(range, sizeof(int));

if (count == NULL) {

printf("Memory allocation failed\n");

return;

}

// Count the occurrences of each element in the input array

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

count[arr[i] - min]++;

}

// Modify the count array to store the cumulative count of elements

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

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

}

// Create an output array to store the sorted elements

int \*output = (int \*)malloc(n \* sizeof(int));

if (output == NULL) {

printf("Memory allocation failed\n");

free(count);

return;

}

// Build the output array by placing elements in their correct positions

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

output[count[arr[i] - min] - 1] = arr[i];

count[arr[i] - min]--;

}

// Copy the sorted elements from the output array back to the input array

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

arr[i] = output[i];

}

// Free the dynamically allocated memory

free(count);

free(output);

}

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

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

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

}

printf("\n");

}

int main() {

int n;

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

scanf("%d", &n);

int arr[n];

printf("Enter the elements:\n");

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

scanf("%d", &arr[i]);

}

printf("Original array: ");

printArray(arr, n);

countingSort(arr, n);

printf("Sorted array: ");

printArray(arr, n);

return 0;

}

Lab 6

**Fractional Knapsack:**

#include <stdio.h>

// Structure to represent an item

typedef struct {

int weight;

int profit;

double ratio;

} Item;

// Function to sort items by their profit-to-weight ratio in descending order

void sortItems(Item items[], int n) {

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

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

if (items[i].ratio < items[j].ratio) {

Item temp = items[i];

items[i] = items[j];

items[j] = temp;

}

}

}

}

// Function to solve the fractional knapsack problem

double fractionalKnapsack(int capacity, Item items[], int n) {

sortItems(items, n);

int currentWeight = 0;

double totalProfit = 0.0;

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

if (currentWeight + items[i].weight <= capacity) {

currentWeight += items[i].weight;

totalProfit += items[i].profit;

} else {

int remainingWeight = capacity - currentWeight;

totalProfit += items[i].ratio \* remainingWeight;

break;

}

}

return totalProfit;

}

int main() {

int n;

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

scanf("%d", &n);

Item items[n];

printf("Enter the weight and profit of each item:\n");

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

printf("Item %d - Weight: ", i + 1);

scanf("%d", &items[i].weight);

printf("Item %d - Profit: ", i + 1);

scanf("%d", &items[i].profit);

items[i].ratio = (double) items[i].profit / items[i].weight;

}

int capacity;

printf("Enter the capacity of the knapsack: ");

scanf("%d", &capacity);

double maxProfit = fractionalKnapsack(capacity, items, n);

printf("Maximum profit: %.2f\n", maxProfit);

return 0;

}

**Min Num of coins Problem:**

#include <stdio.h>

void findMinCoins(int coins[], int n, int V) {

int result[n]; // To store the result (number of coins of each denomination)

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

result[i] = 0; // Initialize the result array with 0

}

// Traverse through all the denominations

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

// Find the maximum number of coins of coins[i] that can fit into V

while (V >= coins[i]) {

V -= coins[i];

result[i]++;

}

}

printf("Minimum coins required:\n");

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

if (result[i] != 0) {

printf("%d coin(s) of %d\n", result[i], coins[i]);

}

}

}

int main() {

int coins[] = {25, 10, 5, 1}; // US coin denominations

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

int V;

printf("Enter the total value V: ");

scanf("%d", &V);

findMinCoins(coins, n, V);

return 0;

}

Lab7

**Huffman Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_TREE\_HT 100

// A Huffman tree node

struct MinHeapNode {

char data;

unsigned freq;

struct MinHeapNode \*left, \*right;

};

// A Min Heap: Collection of min-heap (or Huffman tree) nodes

struct MinHeap {

unsigned size;

unsigned capacity;

struct MinHeapNode\*\* array;

};

// A utility function to allocate a new min heap node with given character and frequency

struct MinHeapNode\* newNode(char data, unsigned freq) {

struct MinHeapNode\* temp = (struct MinHeapNode\*)malloc(sizeof(struct MinHeapNode));

temp->left = temp->right = NULL;

temp->data = data;

temp->freq = freq;

return temp;

}

// A utility function to create a min heap of given capacity

struct MinHeap\* createMinHeap(unsigned capacity) {

struct MinHeap\* minHeap = (struct MinHeap\*)malloc(sizeof(struct MinHeap));

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode\*)malloc(minHeap->capacity \* sizeof(struct MinHeapNode));

return minHeap;

}

// A utility function to swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b) {

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

// The standard minHeapify function.

void minHeapify(struct MinHeap\* minHeap, int idx) {

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// A utility function to check if size of heap is 1 or not

int isSizeOne(struct MinHeap\* minHeap) {

return (minHeap->size == 1);

}

// A standard function to extract minimum value node from heap

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap) {

struct MinHeapNode\* temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

// A utility function to insert a new node to Min Heap

void insertMinHeap(struct MinHeap\* minHeap, struct MinHeapNode\* minHeapNode) {

++minHeap->size;

int i = minHeap->size - 1;

while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

// A standard function to build min heap

void buildMinHeap(struct MinHeap\* minHeap) {

int n = minHeap->size - 1;

int i;

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

minHeapify(minHeap, i);

}

// A utility function to print an array of size n

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

int i;

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

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

printf("\n");

}

// Utility function to check if this node is leaf

int isLeaf(struct MinHeapNode\* root) {

return !(root->left) && !(root->right);

}

// Creates a min heap of capacity equal to size and inserts all character of data[]

// in min heap. Initially size of min heap is equal to capacity

struct MinHeap\* createAndBuildMinHeap(char data[], int freq[], int size) {

struct MinHeap\* minHeap = createMinHeap(size);

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

minHeap->array[i] = newNode(data[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

// The main function that builds Huffman tree

struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size) {

struct MinHeapNode \*left, \*right, \*top;

struct MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);

// Iterate while size of heap doesn't become 1

while (!isSizeOne(minHeap)) {

left = extractMin(minHeap);

right = extractMin(minHeap);

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

printMinHeap(minHeap); // Display the min-heap content at each step

}

return extractMin(minHeap);

}

// Print the contents of the min-heap

void printMinHeap(struct MinHeap\* minHeap) {

printf("Min-Priority Queue: ");

for (int i = 0; i < minHeap->size; ++i)

printf("(%c, %d) ", minHeap->array[i]->data, minHeap->array[i]->freq);

printf("\n");

}

// Prints Huffman codes from the root of Huffman Tree.

// It uses arr[] to store codes

void printCodes(struct MinHeapNode\* root, int arr[], int top) {

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

if (isLeaf(root)) {

printf("%c: ", root->data);

printArr(arr, top);

}

}

// The main function that builds a Huffman Tree and print codes by traversing the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size) {

struct MinHeapNode\* root = buildHuffmanTree(data, freq, size);

int arr[MAX\_TREE\_HT], top = 0;

printCodes(root, arr, top);

}

// Driver program to test above functions

int main() {

char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

int freq[] = { 5, 9, 12, 13, 16, 45 };

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

HuffmanCodes(arr, freq, size);

return 0;

}

Lab8  
**LCS:**

#include <stdio.h>

#include <string.h>

#define MAX\_LENGTH 100

// Function to find the length of LCS

void LCS\_LENGTH(char X[], char Y[], int m, int n) {

int C[MAX\_LENGTH][MAX\_LENGTH]; // 2D array to store lengths of LCS

// Initialize the first row and first column with zeros

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

C[i][0] = 0;

}

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

C[0][j] = 0;

}

// Fill the 2D array C with LCS lengths

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

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

if (X[i - 1] == Y[j - 1]) {

C[i][j] = C[i - 1][j - 1] + 1;

} else {

if (C[i - 1][j] >= C[i][j - 1]) {

C[i][j] = C[i - 1][j];

} else {

C[i][j] = C[i][j - 1];

}

}

}

}

// Print the 2D array C

printf("2D array C containing lengths of LCS:\n");

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

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

printf("%d ", C[i][j]);

}

printf("\n");

}

// The length of LCS will be in C[m][n]

printf("Length of LCS: %d\n", C[m][n]);

}

int main() {

char X[MAX\_LENGTH], Y[MAX\_LENGTH];

// Input sequences from user

printf("Enter sequence X: ");

scanf("%s", X);

printf("Enter sequence Y: ");

scanf("%s", Y);

int m = strlen(X);

int n = strlen(Y);

// Call the LCS\_LENGTH function

LCS\_LENGTH(X, Y, m, n);

return 0;

}