DAY-4 LAB PROGRAMS

31.#include <stdio.h>

// Recursive function to find the sum of digits of a number

int sumOfDigits(int n) {

if (n == 0)

return 0;

return (n % 10) + sumOfDigits(n / 10);

}

int main() {

int number = 1234; // Example number

int sum = sumOfDigits(number);

printf("Sum of digits of %d is %d\n", number, sum);

return 0;

}

32.#include <stdio.h>

int linearSearch(int arr[], int size, int target) {

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

if (arr[i] == target) {

return i; // Return the index of the target element

}

}

return -1; // Target not found

}

int main() {

int arr[] = {2, 4, 0, 1, 9};

int target = 1;

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

int result = linearSearch(arr, size, target);

if (result != -1) {

printf("Element found at index %d\n", result);

} else {

printf("Element not found\n");

}

return 0;

}

33.#include <stdio.h>

#include <stdbool.h>

#define N 4 // Change this value to the size of the board

void printSolution(int board[N][N]) {

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

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

printf("%s ", board[i][j] ? "Q" : ".");

}

printf("\n");

}

}

bool isSafe(int board[N][N], int row, int col) {

int i, j;

// Check this row on left side

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

if (board[row][i]) {

return false;

}

}

// Check upper diagonal on left side

for (i = row, j = col; i>= 0 && j >= 0; i--, j--) {

if (board[i][j]) {

return false;

}

}

// Check lower diagonal on left side

for (i = row, j = col; j >= 0 &&i< N; i++, j--) {

if (board[i][j]) {

return false;

}

}

return true;

}

bool solveNQueensUtil(int board[N][N], int col) {

// If all queens are placed, return true

if (col >= N) {

return true;

}

// Try placing this queen in all rows one by one

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

// Check if the queen can be placed on board[i][col]

if (isSafe(board, i, col)) {

// Place this queen in board[i][col]

board[i][col] = 1;

// Recur to place rest of the queens

if (solveNQueensUtil(board, col + 1)) {

return true;

}

// If placing queen in board[i][col] doesn't lead to a solution,

// then remove queen from board[i][col]

board[i][col] = 0; // Backtrack

}

}

// If the queen cannot be placed in any row in this column col, then return false

return false;

}

void solveNQueens() {

int board[N][N] = {0};

if (solveNQueensUtil(board, 0) == false) {

printf("Solution does not exist\n");

return;

}

printSolution(board);

}

int main() {

solveNQueens();

return 0;

}

34.#include <stdio.h>

#define MAX\_SIZE 100 // Define maximum size of the list

void insertElement(int arr[], int \*size, int element, int position) {

if (\*size >= MAX\_SIZE) {

printf("Error: List is full.\n");

return;

}

if (position < 0 || position > \*size) {

printf("Error: Invalid position.\n");

return;

}

// Shift elements to the right to make space for the new element

for (int i = \*size; i> position; i--) {

arr[i] = arr[i - 1];

}

// Insert the new element

arr[position] = element;

// Update the size of the list

(\*size)++;

}

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

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

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

}

printf("\n");

}

int main() {

int arr[MAX\_SIZE] = {1, 2, 4, 5}; // Initial list

int size = 4; // Current size of the list

int element = 3; // Element to insert

int position = 2; // Position to insert the element

printf("Original list: ");

printList(arr, size);

insertElement(arr, &size, element, position);

printf("List after insertion: ");

printList(arr, size);

return 0;

}

35.#include <stdio.h>

#define MAX\_SIZE 100

void printSubset(int subset[], int size) {

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

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

}

printf("\n");

}

void findSubsets(int arr[], int subset[], int start, int end, int index, int target) {

int sum = 0;

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

sum += subset[i];

}

if (sum == target) {

printSubset(subset, index);

return;

}

for (int i = start; i<= end; i++) {

// Include arr[i] in the subset

subset[index] = arr[i];

// Recur with the remaining elements

findSubsets(arr, subset, i + 1, end, index + 1, target);

}

}

void subsetSum(int arr[], int n, int target) {

int subset[MAX\_SIZE];

findSubsets(arr, subset, 0, n - 1, 0, target);

}

int main() {

int arr[] = {3, 34, 4, 12, 5, 2}; // Input array

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

int target = 9; // Target sum

printf("Subsets with sum %d are:\n", target);

subsetSum(arr, n, target);

return 0;

}

36.#include <stdio.h>

#include <stdbool.h>

#define V 4 // Number of vertices in the graph

// Function to check if the current color assignment is safe for vertex v

bool isSafe(int graph[V][V], int color[], int v, int c) {

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

if (graph[v][i] && color[i] == c) {

return false;

}

}

return true;

}

// Utility function to solve the graph coloring problem using backtracking

bool graphColoringUtil(int graph[V][V], int m, int color[], int v) {

// Base case: If all vertices are assigned a color then return true

if (v == V) {

return true;

}

// Consider this vertex v and try different colors

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

// Check if assignment of color c to v is safe

if (isSafe(graph, color, v, c)) {

color[v] = c; // Assign color c to vertex v

// Recur to assign colors to the rest of the vertices

if (graphColoringUtil(graph, m, color, v + 1)) {

return true;

}

// If assigning color c doesn't lead to a solution, remove it

color[v] = 0;

}

}

// If no color can be assigned to this vertex, return false

return false;

}

// Function to solve the graph coloring problem

void graphColoring(int graph[V][V], int m) {

int color[V];

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

color[i] = 0; // Initialize all vertices as uncolored

}

if (graphColoringUtil(graph, m, color, 0)) {

printf("Solution exists: Following are the assigned colors:\n");

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

printf("Vertex %d: Color %d\n", i, color[i]);

}

} else {

printf("Solution does not exist\n");

}

}

int main() {

// Adjacency matrix representation of the graph

int graph[V][V] = {

{0, 1, 1, 1},

{1, 0, 1, 0},

{1, 1, 0, 1},

{1, 0, 1, 0}

};

int m = 3; // Number of colors

graphColoring(graph, m);

return 0;

}

37.#include <stdio.h>

#include <stdlib.h>

int main() {

int \*array;

int size, i;

// Input the size of the array

printf("Enter the size of the array: ");

scanf("%d", &size);

// Allocate memory for the array

array = (int \*)malloc(size \* sizeof(int));

if (array == NULL) {

printf("Memory allocation failed\n");

return 1; // Exit the program with an error code

}

// Initialize the array with values

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

array[i] = i \* 10; // Example: populate with multiples of 10

}

// Print the array

printf("Array elements:\n");

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

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

}

printf("\n");

// Free the allocated memory

free(array);

return 0; // Successful execution

}

38.#include <stdio.h>

// Recursive function to find and print factors

void printFactors(int n, int i) {

if (i> n / 2) {

if (n % i == 0) {

printf("%d ", i);

}

return;

}

// Recursive call with the next integer

printFactors(n, i + 1);

// Check if the current value of i is a factor

if (n % i == 0) {

printf("%d ", i);

}

}

int main() {

int n;

// Input the number

printf("Enter a number: ");

scanf("%d", &n);

// Handle the case where n is less than or equal to 0

if (n <= 0) {

printf("Please enter a positive integer.\n");

return 1;

}

// Print the factors of the number

printf("Factors of %d are:\n", n);

// Start the recursive function with i = 1

printFactors(n, 1);

return 0;

}

39.#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define N 4 // Number of agents/tasks

// Function to find the minimum cost of an assignment problem using branch and bound

void findMinCost(int cost[N][N], int visited[], int agent, int currCost, int\* minCost) {

// Base case: all agents are assigned

if (agent == N) {

if (currCost< \*minCost) {

\*minCost = currCost;

}

return;

}

// Recur for each task

for (int task = 0; task < N; task++) {

// Check if the task is not already assigned

if (!visited[task]) {

visited[task] = 1; // Mark the task as assigned

findMinCost(cost, visited, agent + 1, currCost + cost[agent][task], minCost);

visited[task] = 0; // Backtrack

}

}

}

int main() {

// Example cost matrix

int cost[N][N] = {

{10, 2, 8, 7},

{6, 9, 7, 5},

{5, 8, 6, 9},

{7, 6, 9, 10}

};

int visited[N] = {0}; // To keep track of assigned tasks

int minCost = INT\_MAX;

findMinCost(cost, visited, 0, 0, &minCost);

printf("Minimum assignment cost: %d\n", minCost);

return 0;

}

40.#include <stdio.h>

#include <stdbool.h>

#define V 5

bool isSafe(int v, bool graph[V][V], int path[], int pos) {

if (graph[path[pos - 1]][v] == 0)

return false;

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

if (path[i] == v)

return false;

return true;

}

bool hamCycleUtil(bool graph[V][V], int path[], int pos) {

if (pos == V)

return graph[path[pos - 1]][path[0]] == 1;

for (int v = 1; v < V; v++) {

if (isSafe(v, graph, path, pos)) {

path[pos] = v;

if (hamCycleUtil(graph, path, pos + 1))

return true;

path[pos] = -1;

}

}

return false;

}

void printSolution(int path[]) {

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

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

printf("%d\n", path[0]);

}

int main() {

bool graph[V][V] = {{0, 1, 0, 1, 0},

{1, 0, 1, 1, 1},

{0, 1, 0, 0, 1},

{1, 1, 0, 0, 1},

{0, 1, 1, 1, 0}};

int path[V];

for (int i = 0; i< V; i++) path[i] = -1;

path[0] = 0;

if (hamCycleUtil(graph, path, 1))

printSolution(path);

else

printf("No solution exists\n");

return 0;

}