**Source code:**

// Dijkstra's Algorithm in C

#include <stdio.h>

#define INFINITY 9999

#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {

int cost[MAX][MAX], distance[MAX], pred[MAX];

int visited[MAX], count, mindistance, nextnode, i, j;

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

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

if (Graph[i][j] == 0)

cost[i][j] = INFINITY;

else

cost[i][j] = Graph[i][j];

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

distance[i] = cost[start][i];

pred[i] = start;

visited[i] = 0;

}

distance[start] = 0;

visited[start] = 1;

count = 1;

while (count < n - 1) {

mindistance = INFINITY;

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

if (distance[i] < mindistance && !visited[i]) {

mindistance = distance[i];

nextnode = i;

}

visited[nextnode] = 1;

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

if (!visited[i])

if (mindistance + cost[nextnode][i] < distance[i]) {

distance[i] = mindistance + cost[nextnode][i];

pred[i] = nextnode;

}

count++;

}

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

if (i != start) {

printf("\nDistance from source to %d: %d", i, distance[i]);

}

}

int main() {

int Graph[MAX][MAX], i, j, n, u;

n = 7;

Graph[0][0] = 0;

Graph[0][1] = 0;

Graph[0][2] = 1;

Graph[0][3] = 2;

Graph[0][4] = 0;

Graph[0][5] = 0;

Graph[0][6] = 0;

Graph[1][0] = 0;

Graph[1][1] = 0;

Graph[1][2] = 2;

Graph[1][3] = 0;

Graph[1][4] = 0;

Graph[1][5] = 3;

Graph[1][6] = 0;

Graph[2][0] = 1;

Graph[2][1] = 2;

Graph[2][2] = 0;

Graph[2][3] = 1;

Graph[2][4] = 3;

Graph[2][5] = 0;

Graph[2][6] = 0;

Graph[3][0] = 2;

Graph[3][1] = 0;

Graph[3][2] = 1;

Graph[3][3] = 0;

Graph[3][4] = 0;

Graph[3][5] = 0;

Graph[3][6] = 1;

Graph[4][0] = 0;

Graph[4][1] = 0;

Graph[4][2] = 3;

Graph[4][3] = 0;

Graph[4][4] = 0;

Graph[4][5] = 2;

Graph[4][6] = 0;

Graph[5][0] = 0;

Graph[5][1] = 3;

Graph[5][2] = 0;

Graph[5][3] = 0;

Graph[5][4] = 2;

Graph[5][5] = 0;

Graph[5][6] = 1;

Graph[6][0] = 0;

Graph[6][1] = 0;

Graph[6][2] = 0;

Graph[6][3] = 1;

Graph[6][4] = 0;

Graph[6][5] = 1;

Graph[6][6] = 0;

u = 0;

Dijkstra(Graph, n, u);

return 0;

}

**Output:**

Distance from source to 1: 3

Distance from source to 2: 1

Distance from source to 3: 2

Distance from source to 4: 4

Distance from source to 5: 4

Distance from source to 6: 3

**Source code:**

// Floyd-Warshall Algorithm in C

#include <stdio.h>

#define nV 4

#define INF 999

void printMatrix(int matrix[][nV]);

void floydWarshall(int graph[][nV]) {

int matrix[nV][nV], i, j, k;

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

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

matrix[i][j] = graph[i][j];

for (k = 0; k < nV; k++) {

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

for (j = 0; j < nV; j++) {

if (matrix[i][k] + matrix[k][j] < matrix[i][j])

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

} } }

printMatrix(matrix);

}

void printMatrix(int matrix[][nV]) {

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

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

if (matrix[i][j] == INF)

printf("%4s", "INF");

else

printf("%4d", matrix[i][j]);}

printf("\n");

}}

int main() {

int graph[nV][nV] = {{0, 3, INF, 5},

{2, 0, INF, 4},

{INF, 1, 0, INF},

{INF, INF, 2, 0}};

floydWarshall(graph);

**Output**

0 3 7 5

2 0 6 4

3 1 0 5

5 3 2 0

**Source code:**

**// Hamiltonian cycle**

#include <stdio.h>

#define NODE 5

int graph[NODE][NODE] = {

{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[NODE];

void displayCycle() {

printf("Cycle Found: ");

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

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

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

int isValid(int v, int k) {

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

return 0;

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

if (path[i] == v)

return 0;

return 1;}

int cycleFound(int k) {

if (k == NODE) {

if (graph[path[k - 1]][path[0]] == 1)

return 1;

else

return 0;

}

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

if (isValid(v, k)) {

path[k] = v;

if (cycleFound(k + 1) == 1)

return 1;

path[k] = -1;

}

}

return 0;

}

int hamiltonianCycle() {

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

path[i] = -1;

path[0] = 0;

if (cycleFound(1) == 0) {

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

return 0; }

displayCycle();

return 1;}

int main() {

hamiltonianCycle();

return 0;

}

**Output :**

Cycle Found: 0 1 2 4 3 0

**Source code:**

//job assignment

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define N 4

int minCost = INT\_MAX;

int minAssignment[N];

void assignJobs(int costMatrix[N][N], int assigned[N], int cost, int level) {

if (level == N) {

if (cost < minCost) {

minCost = cost;

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

minAssignment[i] = assigned[i];

}

}

return;

}

if (cost >= minCost)

return;

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

if (!assigned[i]) {

assigned[i] = 1;

assignJobs(costMatrix, assigned, cost + costMatrix[i][level], level + 1);

assigned[i] = 0;

}

}

}

int main() {

int costMatrix[N][N] = {{9, 2, 7, 8},

{6, 4, 3, 7},

{5, 8, 1, 8},

{7, 6, 9, 4}};

int assigned[N] = {0};

assignJobs(costMatrix, assigned, 0, 0);

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

printf("Correct job assignment to workers:\n");

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

printf("Worker %d assigned to Job %d\n", i+1, minAssignment[i]+1);

}

return 0;

}

**Output :**

Minimum cost of job assignment: 13

Correct job assignment to workers:

Worker 1 assigned to Job 2

Worker 2 assigned to Job 1

Worker 3 assigned to Job 3

Worker 4 assigned to Job 4

**Source code:**

//Knapsack problem

#include <stdio.h>

int cost[4+1][5+1];

int knapsack(int n, int W, int wm[], int vm[]) {

int w, i;

for(w=0; w<=W; w++) {

cost[0][w] = 0;

}

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

cost[i][0] = 0;

}

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

for(w=1; w<=W; w++) {

if(wm[i] > w) {

cost[i][w] = cost[i-1][w];

}

else {

if (vm[i]+cost[i-1][w-wm[i]] > cost[i-1][w]) {

cost[i][w] = vm[i] + cost[i-1][w-wm[i]];

}

else {

cost[i][w] = cost[i-1][w];

}

}

}

}

return cost[n][W];

}

void items\_in\_optimal(int n, int W, int wm[]) {

int i = n;

int j = W;

while (i > 0 && j > 0) {

if(cost[i][j] != cost[i-1][j]) {

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

j = j-wm[i];

i = i-1;

}

else {

i = i-1;

}

}

}

int main() {

int wm[] = {0, 3, 2, 4, 1};

int vm[] = {0, 8, 3, 9, 6};

knapsack(4, 5, wm, vm);

items\_in\_optimal(4, 5, wm);

return 0;

}

**Output :**

**4**

**3**

**Source code:**

//Kruskal

#include <stdio.h>

#include <stdlib.h>

#define MAX\_EDGES 1000

typedef struct Edge {

int src, dest, weight;

} Edge;

typedef struct Graph {

int V, E;

Edge edges[MAX\_EDGES];

} Graph;

typedef struct Subset {

int parent, rank;

} Subset;

Graph\* createGraph(int V, int E) {

Graph\* graph = (Graph\*) malloc(sizeof(Graph));

graph->V = V;

graph->E = E;

return graph;

}

int find(Subset subsets[], int i) {

if (subsets[i].parent != i) {

subsets[i].parent = find(subsets, subsets[i].parent);

}

return subsets[i].parent;

}

void Union(Subset subsets[], int x, int y) {

int xroot = find(subsets, x);

int yroot = find(subsets, y);

if (subsets[xroot].rank < subsets[yroot].rank) {

subsets[xroot].parent = yroot;

} else if (subsets[xroot].rank > subsets[yroot].rank) {

subsets[yroot].parent = xroot;

} else {

subsets[yroot].parent = xroot;

subsets[xroot].rank++;

}

}

int compare(const void\* a, const void\* b) {

Edge\* a\_edge = (Edge\*) a;

Edge\* b\_edge = (Edge\*) b;

return a\_edge->weight - b\_edge->weight;

}

void kruskalMST(Graph\* graph) {

Edge mst[graph->V];

int e = 0, i = 0;

qsort(graph->edges, graph->E, sizeof(Edge), compare);

Subset\* subsets = (Subset\*) malloc(graph->V \* sizeof(Subset));

for (int v = 0; v < graph->V; ++v) {

subsets[v].parent = v;

subsets[v].rank = 0;

}

while (e < graph->V - 1 && i < graph->E) {

Edge next\_edge = graph->edges[i++];

int x = find(subsets, next\_edge.src);

int y = find(subsets, next\_edge.dest);

if (x != y) {

mst[e++] = next\_edge;

Union(subsets, x, y);

}

}

printf("Minimum Spanning Tree:\n");

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

printf("(%d, %d) -> %d\n", mst[i].src, mst[i].dest, mst[i].weight);

}

}

int main() {

int V, E;

printf("Enter number of vertices and edges: ");

scanf("%d %d", &V, &E);

Graph\* graph = createGraph(V, E);

printf("Enter edges and their weights:\n");

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

scanf("%d %d %d", &graph->edges[i].src, &graph->edges[i].dest, &graph->edges[i].weight);

}

kruskalMST(graph);

return 0;

}

**Output :**

Enter number of vertices and edges: 2 2

Enter edges and their weights:

1 2 3

2 2 4

Minimum Spanning Tree:

(1, 2) -> 3

**Source code:**

// NQueens problem

#define N 4

#include <stdbool.h>

#include <stdio.h>

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

{

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

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

if(board[i][j])

printf("Q ");

else

printf(". ");

}

printf("\n");

}

}

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

{

int i, j;

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

if (board[row][i])

return false;

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

if (board[i][j])

return false;

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

if (board[i][j])

return false;

return true;

}

bool solveNQUtil(int board[N][N], int col)

{

if (col >= N)

return true;

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

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

board[i][col] = 1;

if (solveNQUtil(board, col + 1))

return true;

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

}

}

return false;

}

bool solveNQ()

{

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

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

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

printf("Solution does not exist");

return false;

}

printSolution(board);

return true;

}

int main()

{

solveNQ();

return 0;

}

**OUTPUT :**

. . Q .

Q . . .

. . . Q

. Q . .

**Source code:**

// Prim's Algorithm in C

#include<stdio.h>

#include<stdbool.h>

#include<string.h>

#define INF 9999999

#define V 5

int G[V][V] = {

{0, 9, 75, 0, 0},

{9, 0, 95, 19, 42},

{75, 95, 0, 51, 66},

{0, 19, 51, 0, 31},

{0, 42, 66, 31, 0}};

int main() {

int no\_edge; // number of edge

int selected[V];

memset(selected, false, sizeof(selected));

no\_edge = 0;

selected[0] = true;

int x; // row number

int y; // col number

printf("Edge : Weight\n");

while (no\_edge < V - 1) {

int min = INF;

x = 0;

y = 0;

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

if (selected[i]) {

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

if (!selected[j] && G[i][j]) { // not in selected and there is an edge

if (min > G[i][j]) {

min = G[i][j];

x = i;

y = j;

}

}

}

}

}

printf("%d - %d : %d\n", x, y, G[x][y]);

selected[y] = true;

no\_edge++;

}

return 0;

}

**OUTPUT :**

Edge : Weight

0 - 1 : 9

1 - 3 : 19

3 - 4 : 31

3 - 2 : 51

**Source code:**

// sum of subset problem

#include <stdio.h>

#include <stdlib.h>

static int total\_nodes;

void printValues(int A[], int size){

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

printf("%\*d", 5, A[i]);

}

printf("\n");

}

void subset\_sum(int s[], int t[], int s\_size, int t\_size, int sum, int ite, int const target\_sum){

total\_nodes++;

if (target\_sum == sum) {

printValues(t, t\_size);

subset\_sum(s, t, s\_size, t\_size - 1, sum - s[ite], ite + 1, target\_sum);

return;

}

else {

for (int i = ite; i < s\_size; i++) {

t[t\_size] = s[i];

subset\_sum(s, t, s\_size, t\_size + 1, sum + s[i], i + 1, target\_sum);

}

}

}

void generateSubsets(int s[], int size, int target\_sum){

int\* tuplet\_vector = (int\*)malloc(size \* sizeof(int));

subset\_sum(s, tuplet\_vector, size, 0, 0, 0, target\_sum);

free(tuplet\_vector);

}

int main(){

int set[] = { 5, 10, 12 , 13, 15 , 18};

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

printf("The set is ");

printValues(set , size);

printf("subsets are:\n");

generateSubsets(set, size, 30);

printf("Total Nodes generated %d\n", total\_nodes);

return 0;

}

**Output:**

**The set is 5 10 12 13 15 18**

**subsets are:**

**5 10 15**

**5 12 13**

**12 18**

**12**

**Total Nodes generated 65**

**Source code:**

// transitive closure

#include<stdio.h>

#include<math.h>

int max(int, int);

void warshal(int p[10][10], int n) {

int i, j, k;

for (k = 1; k <= n; k++)

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

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

p[i][j] = max(p[i][j], p[i][k] && p[k][j]);

}

int max(int a, int b) {

;

if (a > b)

return (a);

else

return (b);

}

void main() {

int p[10][10] = { 0 }, n, e, u, v, i, j;

printf("\n Enter the number of vertices:");

scanf("%d", &n);

printf("\n Enter the number of edges:");

scanf("%d", &e);

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

printf("\n Enter the end vertices of edge %d:", i);

scanf("%d%d", &u, &v);

p[u][v] = 1;

}

printf("\n Matrix of input data: \n");

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

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

printf("%d\t", p[i][j]);

printf("\n");

}

warshal(p, n);

printf("\n Transitive closure: \n");

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

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

printf("%d\t", p[i][j]);

printf("\n");

}

}

**Output::**

**Enter the number of vertices:4**

**Enter the number of edges:4**

**Enter the end vertices of edge 1:2 3**

**Enter the end vertices of edge 2: 3 43 4**

**Enter the end vertices of edge 3: 4 54 5**

**Enter the end vertices of edge 4:5 6**

**Matrix of input data:**

**0 0 0 0**

**0 0 1 0**

**0 0 0 1**

**0 0 0 0**

**Transitive closure:**

**0 0 0 0**

**0 0 1 1**

**0 0 0 1**

**0 0 0 0**

**Source code:**

// Travelling salesman problem

#include <stdio.h>

int tsp\_g[10][10] = {

{12, 30, 33, 10, 45},

{56, 22, 9, 15, 18},

{29, 13, 8, 5, 12},

{33, 28, 16, 10, 3},

{1, 4, 30, 24, 20}

};

int visited[10], n, cost = 0;

void travellingsalesman(int c){

int k, adj\_vertex = 999;

int min = 999;

visited[c] = 1;

printf("%d ", c + 1);

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

if((tsp\_g[c][k] != 0) && (visited[k] == 0)) {

if(tsp\_g[c][k] < min) {

min = tsp\_g[c][k];

}

adj\_vertex = k;

}

}

if(min != 999) {

cost = cost + min;

}

if(adj\_vertex == 999) {

adj\_vertex = 0;

printf("%d", adj\_vertex + 1);

cost = cost + tsp\_g[c][adj\_vertex];

return;

}

travellingsalesman(adj\_vertex);

}

int main(){

int i, j;

n = 5;

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

visited[i] = 0;

}

printf("Shortest Path: ");

travellingsalesman(0);

printf("\nMinimum Cost: ");

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

return 0;

}

**Output:**

**Shortest Path: 1 5 4 3 2 1**

**Minimum Cost: 99**

**Source code:**

// Towers of Hanoi

#include <stdio.h>

void towers\_of\_hanoi(int n, char source, char target, char auxiliary) {

if (n == 1) {

printf("Move disk 1 from %c to %c\n", source, target);

return;

}

towers\_of\_hanoi(n - 1, source, auxiliary, target);

printf("Move disk %d from %c to %c\n", n, source, target);

towers\_of\_hanoi(n - 1, auxiliary, target, source);

}

int main() {

int num\_disks = 3;

towers\_of\_hanoi(num\_disks, 'A', 'C', 'B');

return 0;

}

**Output:**

**Move disk 1 from A to C**

**Move disk 2 from A to B**

**Move disk 1 from C to B**

**Move disk 3 from A to C**

**Move disk 1 from B to A**

**Move disk 2 from B to C**

**Move disk 1 from A to C**

**Source code:**

//factorial using non recursive algorithm

#include<stdio.h>

long int fact(int n);

int main()

{

long int number, result;

printf("Enter any positive integer: ");

scanf("%ld", &number);

result = fact(number);

printf("%ld != %ld",number,result);

return(0);

}

long int fact(int n)

{

long int i, f=1;

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

{

f = f\*i;

}

return f;

}

**Output:**

**Enter any positive integer: 4**

**4 != 24**

**Source code:**

// Bubble sort

#include <stdio.h>

void bubble\_sort(int arr[], int n) {

int i, j, temp;

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

int swapped = 0;

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

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

temp = arr[j];

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

arr[j+1] = temp;

swapped = 1;

}

}

if (!swapped) {

break;} }}

int main() {

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

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

bubble\_sort(arr, n);

printf("Sorted array is: ");

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

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

}

printf("\n");

return 0;

}

**Output:**

**Sorted array is: 11 12 22 25 34 64 90**

**Source code:**

// factorial using recursive algorithm

#include<stdio.h>

long int multiplyNumbers(int n);

int main() {

int n;

printf("Enter a positive integer: ");

scanf("%d",&n);

printf("Factorial of %d = %ld", n, multiplyNumbers(n));

return 0;

}

long int multiplyNumbers(int n) {

if (n>=1)

return n\*multiplyNumbers(n-1);

else

return 1;

}

**Output :**

**Enter a positive integer: 5**

**Factorial of 5 = 120**