**EXP 4:-**

#include <stdio.h>

#include <stdlib.h>

typedef struct Node {

char data;

unsigned freq;

struct Node \*left, \*right;

} Node;

typedef struct MinHeap {

unsigned size;

unsigned capacity;

Node\*\* array;

} MinHeap;

Node\* newNode(char data, unsigned freq) {

Node\* node = (Node\*)malloc(sizeof(Node));

node->left = node->right = NULL;

node->data = data;

node->freq = freq;

return node;

}

MinHeap\* createMinHeap(unsigned capacity) {

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

minHeap->size = 0;

minHeap->capacity = capacity;

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

return minHeap;

}

void swapMinHeapNode(Node\*\* a, Node\*\* b) {

Node\* t = \*a;

\*a = \*b;

\*b = t;

}

void minHeapify(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);

}

}

void buildMinHeap(MinHeap\* minHeap) {

int n = minHeap->size - 1;

int i;

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

minHeapify(minHeap, i);

}

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

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

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

printf("\n");

}

void printCodes(Node\* 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 (!root->left && !root->right) {

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

printArr(arr, top);

}

}

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

Node \*left, \*right, \*top;

MinHeap\* minHeap = createMinHeap(size);

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

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

minHeap->size = size;

buildMinHeap(minHeap);

while (minHeap->size != 1) {

left = extractMin(minHeap);

right = extractMin(minHeap);

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

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

return extractMin(minHeap);

}

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

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

int arr[100], top = 0;

printCodes(root, arr, top);

}

int main() {

int size;

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

scanf("%d", &size);

char\* data = (char\*)malloc(size \* sizeof(char));

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

printf("Enter elements and their frequencies:\n");

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

printf("Element %d: ", i + 1);

scanf(" %c", &data[i]);

printf("Frequency %d: ", i + 1);

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

}

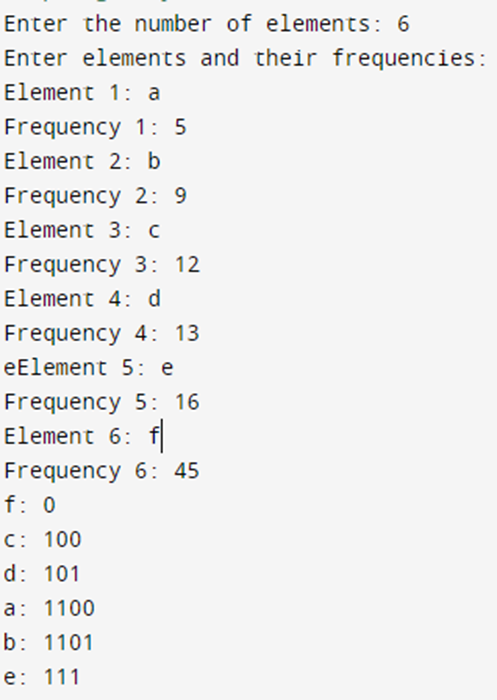
HuffmanCodes(data, freq, size);

free(data);

free(freq);

return 0;

}



**EXP 5:-**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

void printSolution(int \*\*dist, int V) {

printf("Shortest distances between every pair of vertices:\n");

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

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

printf("%d\t", dist[i][j] == INT\_MAX ? -1 : dist[i][j]);

}

printf("\n");

}

}

void floydWarshall(int \*\*graph, int V) {

int \*\*dist = (int \*\*)malloc(V \* sizeof(int \*));

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

dist[i] = (int \*)malloc(V \* sizeof(int));

memcpy(dist[i], graph[i], V \* sizeof(int));

}

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

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

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

if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX &&

dist[i][k] + dist[k][j] < dist[i][j]) {

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

}

}

}

}

printSolution(dist, V);

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

free(dist[i]);

}

free(dist);

}

int main() {

int V;

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

scanf("%d", &V);

int \*\*graph = (int \*\*)malloc(V \* sizeof(int \*));

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

graph[i] = (int \*)malloc(V \* sizeof(int));

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

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

if (graph[i][j] == -1) graph[i][j] = INT\_MAX;

}

}

floydWarshall(graph, V);

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

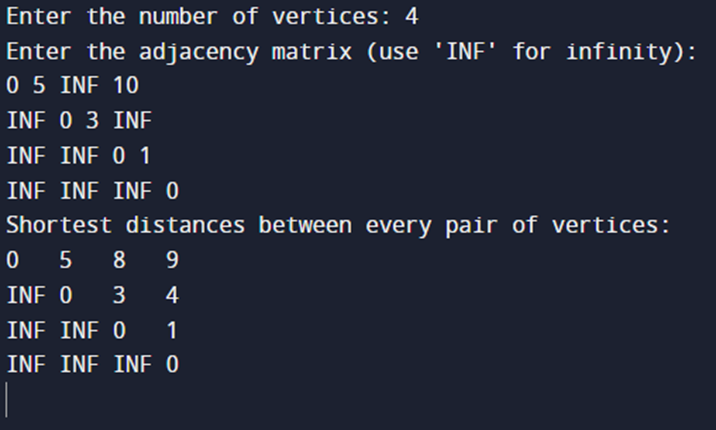
free(graph[i]);

}

free(graph);

return 0;

}



**EXP 6:-**

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

int min(int x, int y) {

return (x < y) ? x : y;

}

int tsp(int \*\*graph, int mask, int pos, int n, int \*\*dp) {

if (mask == (1 << n) - 1)

return graph[pos][0];

if (dp[mask][pos] != -1)

return dp[mask][pos];

int ans = INT\_MAX;

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

if ((mask & (1 << city)) == 0) {

int newAns = graph[pos][city] + tsp(graph, mask | (1 << city), city, n, dp);

ans = min(ans, newAns);

}

}

return dp[mask][pos] = ans;

}

int main() {

int n;

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

scanf("%d", &n);

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

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

graph[i] = (int \*)malloc(n \* sizeof(int));

}

int \*\*dp = (int \*\*)malloc((1 << n) \* sizeof(int \*));

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

dp[i] = (int \*)malloc(n \* sizeof(int));

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

dp[i][j] = -1;

}

}

printf("Enter the adjacency matrix for the graph:\n");

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

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

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

}

}

int minCost = tsp(graph, 1, 0, n, dp);

printf("The minimum cost to visit all cities: %d\n", minCost);

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

free(graph[i]);

}

free(graph);

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

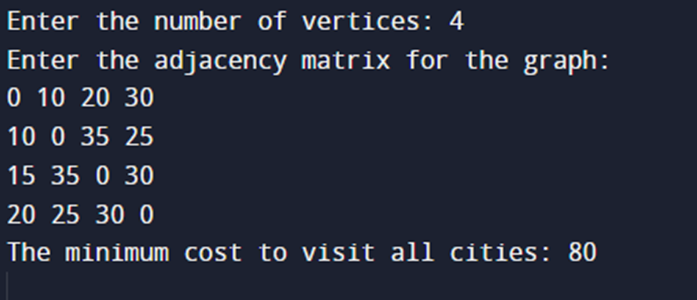
free(dp[i]);

}

free(dp);

return 0;

}



**EXP 7:-**

#include <stdio.h>

#include <stdbool.h>

#define N 4

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

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

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

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

printf("\n");

}

}

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

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

if (board[row][i]) return false;

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

if (board[i][j]) return false;

for (int 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;

}

int main() {

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

if (!solveNQUtil(board, 0)) {

printf("Solution does not exist");

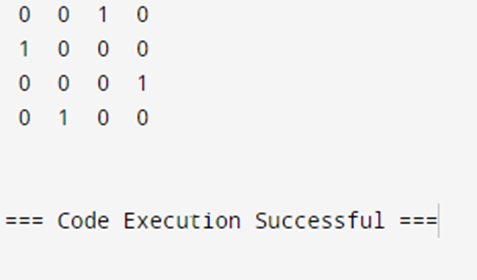
return 0;

}

printSolution(board);

return 0;

}



**EXP 8:-**

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <limits.h>

#define N 4

typedef struct {

int puzzle[N][N];

int x, y;

int cost;

} PuzzleState;

typedef struct {

PuzzleState state;

int level;

} SearchNode;

PuzzleState createPuzzleState(int puzzle[N][N], int x, int y, int cost) {

PuzzleState newState;

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

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

newState.puzzle[i][j] = puzzle[i][j];

newState.x = x;

newState.y = y;

newState.cost = cost;

return newState;

}

bool isGoalState(PuzzleState state) {

int count = 1;

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

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

if (state.puzzle[i][j] != count && !(i == N - 1 && j == N - 1))

return false;

else

count++;

return true;

}

void printPuzzleState(PuzzleState state) {

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

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

printf("%2d ", state.puzzle[i][j]);

printf("\n");

}

printf("\n");

}

void swap(int\* x, int\* y) {

int temp = \*x;

\*x = \*y;

\*y = temp;

}

PuzzleState move(PuzzleState state, int newX, int newY) {

swap(&state.puzzle[state.x][state.y], &state.puzzle[newX][newY]);

state.x = newX;

state.y = newY;

return state;

}

bool isValidMove(int x, int y) {

return (x >= 0 && x < N && y >= 0 && y < N);

}

int calculateHeuristicCost(PuzzleState state) {

int cost = 0;

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

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

if (state.puzzle[i][j] != 0) {

int goalX = (state.puzzle[i][j] - 1) / N;

int goalY = (state.puzzle[i][j] - 1) % N;

cost += abs(i - goalX) + abs(j - goalY);

}

return cost;

}

void solvePuzzle(int puzzle[N][N], int x, int y) {

PuzzleState initialState = createPuzzleState(puzzle, x, y, 0);

SearchNode priorityQueue[1000];

int front = -1, rear = -1;

priorityQueue[++rear] = (SearchNode){initialState, 0};

while (front != rear) {

SearchNode currentNode = priorityQueue[++front];

PuzzleState currentState = currentNode.state;

if (isGoalState(currentState)) {

printf("Solution Path:\n");

printf("Initial State:\n");

printPuzzleState(initialState);

printf("Final State:\n");

printPuzzleState(currentState);

printf("Number of steps required: %d\n", currentNode.level);

return;

}

int dx[] = {-1, 1, 0, 0};

int dy[] = {0, 0, -1, 1};

for (int i = 0; i < 4; i++) {

int newX = currentState.x + dx[i];

int newY = currentState.y + dy[i];

if (isValidMove(newX, newY)) {

PuzzleState nextState = move(currentState, newX, newY);

nextState.cost = currentState.cost + 1 + calculateHeuristicCost(nextState);

priorityQueue[++rear] = (SearchNode){nextState, currentNode.level + 1};

}

}

}

printf("No solution found!\n");

}

int main() {

int puzzle[N][N];

printf("Enter the initial state of the puzzle (0 represents the empty cell):\n");

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

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

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

int emptyCellX, emptyCellY;

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

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

if (puzzle[i][j] == 0) {

emptyCellX = i;

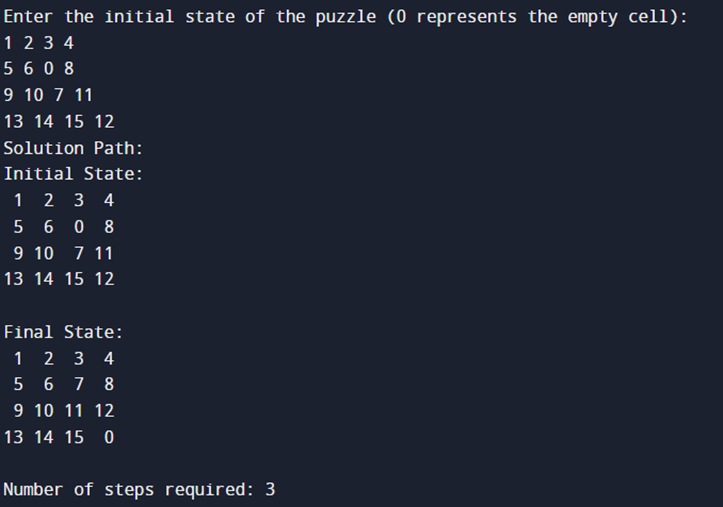
emptyCellY = j;

}

solvePuzzle(puzzle, emptyCellX, emptyCellY);

return 0;

}



**Longest Common Subsequence:-**

#include <stdio.h>

#include <string.h>

int max(int a, int b) {

return (a > b) ? a : b;

}

int lcs(char\* X, char\* Y, int m, int n) {

int L[m + 1][n + 1];

int i, j, index;

// Initialize the LCS table

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

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

if (i == 0 || j == 0)

L[i][j] = 0;

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

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

else

L[i][j] = max(L[i - 1][j], L[i][j - 1]);

}

}

// Backtrack to find the LCS

index = L[m][n];

char LCS[index + 1];

LCS[index] = '\0';

i = m, j = n;

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

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

LCS[index - 1] = X[i - 1];

i--;

j--;

index--;

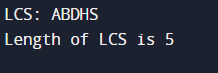
} else if (L[i - 1][j] > L[i][j - 1])

i--;

else

j--;

}

printf("LCS: %s\n", LCS);

return L[m][n];

}

int main() {

char X[] = "ABSDHS";

char Y[] = "ABDHSP";

int m = strlen(X);

int n = strlen(Y);

printf("Length of LCS is %d\n", lcs(X, Y, m, n));

return 0;

}