15Puzzle

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 4
#define N2 16

typedef struct {
    int tiles[N][N];
    int x, y;
    int cost;
    int level;
} State;

void printState(int tiles[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%2d ", tiles[i][j]);
        }
        printf("\n");
    }
}
int calculateHeuristic(int tiles[N][N]) {
    int heuristic = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int value = tiles[i][j];
            if (value != 0) {
                int targetX = (value - 1) / N;
                int targetY = (value - 1) % N;
                int dx = abs(i - targetX);
                int dy = abs(j - targetY);
                heuristic += dx + dy;
            }
        }
    }
    return heuristic;
}

void solve(int initial[N][N], int x, int y) {
    State *root = (State *)malloc(sizeof(State));
    memcpy(root->tiles, initial, sizeof(root->tiles));
    root->x = x;
    root->y = y;
    root->level = 0;
    root->cost = calculateHeuristic(initial);

    printf("Initial state:\n");
    printState(root->tiles);
    printf("Initial heuristic cost: %d\n", root->cost);

    free(root);
}
```

```
int main() {
    int initial[N][N] = {
        {1, 2, 3, 4},
        {5, 6, 0, 8},
        {9, 10, 7, 11},
        {13, 14, 15, 12}
    };
    solve(initial, 1, 2);
    return 0;
}
```

CoinDynamic

```
#include <stdio.h>

// Function to find the number of ways to make change for a given
amount using specified coins
int countWays(int coins[], int n, int amount) {
    int dp[amount + 1];
    int i, j;

    // Initialize the dp array with 0
    for (i = 0; i <= amount; i++) {
        dp[i] = 0;
    }

    // Base case: There is one way to make the amount 0
    dp[0] = 1;

    // Update the dp array for each coin
    for (i = 0; i < n; i++) {
        for (j = coins[i]; j <= amount; j++) {
            dp[j] += dp[j - coins[i]];
        }
    }

    return dp[amount];
}

int main() {
    int n, amount, i;

    printf("Enter the number of different coins: ");
    scanf("%d", &n);

    int coins[n];

    printf("Enter the denominations of the coins: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &coins[i]);
    }

    printf("Enter the amount to make change for: ");
    scanf("%d", &amount);

    printf("Number of ways to make change for %d using the given
coins: %d\n", amount, countWays(coins, n, amount));

    return 0;
}
```

CoinGreedy

```
#include <stdio.h>

void coinChange(int coins[], int n, int amount) {
    int result[100] = {0};  // To store the result (number of coins
of each type used)
    int i;

    for (i = n - 1; i >= 0; i--) {
        while (amount >= coins[i]) {
            amount -= coins[i];
            result[i]++;
        }
    }

    printf("Coin count:\n");
    for (i = n - 1; i >= 0; i--) {
        if (result[i] != 0) {
            printf("%d coin(s) of %d\n", result[i], coins[i]);
        }
    }
}

int main() {
    int coins[100]; // Array to store the denominations
    int n, amount, i;

    printf("Enter the number of different coin denominations: ");
    scanf("%d", &n);

    printf("Enter the coin denominations values:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &coins[i]);
    }

    printf("Enter the amount to make change for: ");
    scanf("%d", &amount);

    coinChange(coins, n, amount);
    return 0;
}
```

Dijikstras

```
#include <stdio.h>
#include <limits.h>

#define V 9

int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == 0 && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;
}


void printSolution(int dist[]) {
    printf("Vertex \t Distance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d \t %d\n", i, dist[i]);
}

void dijkstra(int graph[V][V], int src) {
    int dist[V];
    int sptSet[V];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = 0;

    dist[src] = 0;

    for (int count = 0; count < V – 1; count++) {

        sptSet[u] = 1;

        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX &&
dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    printSolution(dist);
}

int main() {
    // Example graph in form of adjacency matrix
    int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                       {4, 0, 8, 0, 0, 0, 0, 11, 0},
                       {0, 8, 0, 7, 0, 4, 0, 0, 2},
                       {0, 0, 7, 0, 9, 14, 0, 0, 0},
                       {0, 0, 0, 9, 0, 10, 0, 0, 0},
                       {0, 0, 4, 14, 10, 0, 2, 0, 0},
                       {0, 0, 0, 0, 0, 2, 0, 1, 6},
                       {8, 11, 0, 0, 0, 0, 1, 0, 7},
                       {0, 0, 2, 0, 0, 0, 6, 7, 0}};
    dijkstra(graph, 0);
    return 0;
}
```

KMP

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void computeLPSArray(char* pat, int M, int* lps) {
    int len = 0;  // length of the previous longest prefix suffix
    int i = 1;
    lps[0] = 0;  // lps[0] is always 0

    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        } else { // (pat[i] != pat[len])
            if (len != 0) {
                len = lps[len - 1];
            } else { // if (len == 0)
                lps[i] = 0;
                i++;
            }
        }
    }
}

void KMPSearch(char* pat, char* txt) {
    int M = strlen(pat);
    int N = strlen(txt);

    int lps[M];
    computeLPSArray(pat, M, lps);

    int i = 0;  // index for txt[]
    int j = 0;  // index for pat[]
    while (i < N) {
        if (pat[j] == txt[i]) {
            j++;
            i++;
        }

        if (j == M) {
            printf("Pattern found at index %d\n", i - j);
            j = lps[j - 1];
        }

        else if (i < N && pat[j] != txt[i]) {
            // Do not match lps[0..lps[j-1]] characters,
            // they will match anyway
            if (j != 0)
                j = lps[j - 1];
            else
                i = i + 1;
        }
    }
}

int main() {
    char txt[1024];
    char pat[256];

    printf("Enter the text: ");
    fgets(txt, sizeof(txt), stdin);
    txt[strcspn(txt, "\n")] = 0;

    printf("Enter the pattern: ");
    fgets(pat, sizeof(pat), stdin);
    pat[strcspn(pat, "\n")] = 0;

    KMPSearch(pat, txt);
    return 0;
}
```

0-1 Knapsack problem

```
#include <stdio.h>

// A utility function that returns
// maximum of two integers
int max(int a, int b) { return (a > b) ? a : b; }

// Returns the maximum value that can be
// put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
    // Base Case
    if (n == 0 || W == 0)
        return 0;
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);

    else
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1], wt,
val, n - 1),
            knapSack(W, wt, val, n - 1));
}
int main()
{
    int profit[] = { 60, 100, 120 };
    int weight[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(profit) / sizeof(profit[0]);
    printf("%d", knapSack(W, weight, profit, n));
    return 0;
}
```

```c
// C code to implement Kruskal's algorithm

#include <stdio.h>
#include <stdlib.h>

// Comparator function to use in sorting
int comparator(const void* p1, const void* p2)
{
        const int(*x)[3] = p1;
        const int(*y)[3] = p2;

        return (*x)[2] - (*y)[2];
}

// Initialization of parent[] and rank[] arrays
void makeSet(int parent[], int rank[], int n)
{
        for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 0;
        }
}

// Function to find the parent of a node
int findParent(int parent[], int component)
{
        if (parent[component] == component)
                return component;

        return parent[component]
                = findParent(parent, parent[component]);
}

// Function to unite two sets
void unionSet(int u, int v, int parent[], int rank[], int n)
{
        // Finding the parents
        u = findParent(parent, u);
        v = findParent(parent, v);

        if (rank[u] < rank[v]) {
                parent[u] = v;
        }
        else if (rank[u] > rank[v]) {
                parent[v] = u;
        }
        else {
                parent[v] = u;

                // Since the rank increases if
                // the ranks of two sets are same
                rank[u]++;
        }
}
```

```
// Function to find the MST
void kruskalAlgo(int n, int edge[n][3])
{
        // First we sort the edge array in ascending order
        // so that we can access minimum distances/cost
        qsort(edge, n, sizeof(edge[0]), comparator);

        int parent[n];
        int rank[n];

        // Function to initialize parent[] and rank[]
        makeSet(parent, rank, n);

        // To store the minimun cost
        int minCost = 0;

        printf(
                "Following are the edges in the constructed MST\n");
        for (int i = 0; i < n; i++) {
                int v1 = findParent(parent, edge[i][0]);
                int v2 = findParent(parent, edge[i][1]);
                int wt = edge[i][2];

                // If the parents are different that
                // means they are in different sets so
                // union them
                if (v1 != v2) {
                        unionSet(v1, v2, parent, rank, n);
                        minCost += wt;
                        printf("%d -- %d == %d\n", edge[i][0],
                                edge[i][1], wt);
                }
        }

        printf("Minimum Cost Spanning Tree: %d\n", minCost);
}

// Driver code
int main()
{
        int edge[5][3] = { { 0, 1, 10 },
                                        { 0, 2, 6 },
                                        { 0, 3, 5 },
                                        { 1, 3, 15 },
                                        { 2, 3, 4 } };

        kruskalAlgo(5, edge);

        return 0;
}
```

```
}
```

LCS

```
#include <stdio.h>
#include <string.h>

#define MAX 1000  // Maximum size of the string

int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to find the length of the longest common subsequence.
int lcs(char *X, char *Y, int m, int n) {
    int dp[m+1][n+1];
    int i, j;

    // Building the table in bottom-up manner
    for (i = 0; i <= m; i++) {
        for (j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                dp[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }

    return dp[m][n];  // The last entry in dp[][] contains the
length of the LCS.
}

int main() {
    char X[MAX], Y[MAX];
    int m, n;

    printf("Enter first string: ");
    fgets(X, MAX, stdin);  // Read string including spaces
    X[strcspn(X, "\n")] = 0;  // Remove newline character

    printf("Enter second string: ");
    fgets(Y, MAX, stdin);  // Read string including spaces
    Y[strcspn(Y, "\n")] = 0;  // Remove newline character

    m = strlen(X);
    n = strlen(Y);

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

    return 0;
}
```

MatrixChainDynamic

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// Function to find the minimum number of multiplications needed
// to multiply the chain of matrices
int MatrixChainOrder(int p[], int n) {
    int m[n][n];
    int i, j, k, L, q;

    // Initialize single matrix multiplication cost to zero
    for (i = 1; i < n; i++)
        m[i][i] = 0;

    // Calculate minimum multiplication costs for increasing chain
lengths
    for (L = 2; L < n; L++) {
        for (i = 1; i < n - L + 1; i++) {
            j = i + L - 1;
            m[i][j] = INT_MAX;
            for (k = i; k <= j - 1; k++) {
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j];
                if (q < m[i][j])
                    m[i][j] = q;
            }
        }
    }

    return m[1][n - 1];
}

// Driver code with user input
int main() {
    int n, i;
    printf("Enter the number of matrices: ");
    scanf("%d", &n);
    int arr[n+1];

    printf("Enter dimensions of matrices: \n");
    for (i = 0; i <= n; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Minimum number of multiplications is %d\n",
MatrixChainOrder(arr, n + 1));

    return 0;
}
```

N Queen Problem using backtracking

```
#define N 4
#include <stdbool.h>
#include <stdio.h>

{
        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;
}
```

Prims

```
#include <stdio.h>
#include <limits.h>

#define V 5
int minKey(int key[], int mstSet[]) {
    int min = INT_MAX, min_index = -1;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == 0 && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}
void printMST(int parent[], int graph[V][V]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < V; i++)
        printf("%d - %d \t%d \n", parent[i], i, graph[i]
[parent[i]]);
}

void primMST(int graph[V][V]) {
    int parent[V];
    int key[V];
    int mstSet[V];

    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = 0;

    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < V - 1; count++) {
        int u = minKey(key, mstSet);
        mstSet[u] = 1;
        for (int v = 0; v < V; v++)
            if (graph[u][v] && mstSet[v] == 0 && graph[u][v] <
key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    printMST(parent, graph);
}

int main() {
    int graph[V][V] = {{0, 2, 0, 6, 0},
                       {2, 0, 3, 8, 5},
                       {0, 3, 0, 0, 7},
                       {6, 8, 0, 0, 9},
                       {0, 5, 7, 9, 0}};

    primMST(graph);

    return 0;
}
```

Strassens

```c
#include <stdio.h>
#include <stdlib.h>

int **allocate_matrix(int n) {
    int **matrix = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++) {
        matrix[i] = (int *)malloc(n * sizeof(int));
    }
    return matrix;
}

void free_matrix(int **matrix, int n) {
    for (int i = 0; i < n; i++) {
        free(matrix[i]);
    }
    free(matrix);
}

void add_matrix(int **a, int **b, int **result, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = a[i][j] + b[i][j];
        }
    }
}

void subtract_matrix(int **a, int **b, int **result, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            result[i][j] = a[i][j] - b[i][j];
        }
    }
}

void strassen(int **a, int **b, int **c, int n) {
    if (n == 1) {
        c[0][0] = a[0][0] * b[0][0];
        return;
    }

    int new_size = n / 2;
    int **a11 = allocate_matrix(new_size);
    int **a12 = allocate_matrix(new_size);
    int **a21 = allocate_matrix(new_size);
    int **a22 = allocate_matrix(new_size);
    int **b11 = allocate_matrix(new_size);
    int **b12 = allocate_matrix(new_size);
    int **b21 = allocate_matrix(new_size);
    int **b22 = allocate_matrix(new_size);

    int **c11 = allocate_matrix(new_size);
    int **c12 = allocate_matrix(new_size);
    int **c21 = allocate_matrix(new_size);
    int **c22 = allocate_matrix(new_size);

    int **p1 = allocate_matrix(new_size);
    int **p2 = allocate_matrix(new_size);
    int **p3 = allocate_matrix(new_size);
    int **p4 = allocate_matrix(new_size);
    int **p5 = allocate_matrix(new_size);
    int **p6 = allocate_matrix(new_size);
    int **p7 = allocate_matrix(new_size);

    int **tempA = allocate_matrix(new_size);
    int **tempB = allocate_matrix(new_size);

    // Dividing matrices into 4 sub-matrices
    for (int i = 0; i < new_size; i++) {
        for (int j = 0; j < new_size; j++) {
            a11[i][j] = a[i][j];
            a12[i][j] = a[i][j + new_size];
            a21[i][j] = a[i + new_size][j];
            a22[i][j] = a[i + new_size][j + new_size];

            b11[i][j] = b[i][j];
            b12[i][j] = b[i][j + new_size];
            b21[i][j] = b[i + new_size][j];
            b22[i][j] = b[i + new_size][j + new_size];
        }
    }

    // p1 = (a11 + a22) * (b11 + b22)
    add_matrix(a11, a22, tempA, new_size);
    add_matrix(b11, b22, tempB, new_size);
    strassen(tempA, tempB, p1, new_size);

    // p2 = (a21 + a22) * b11
    add_matrix(a21, a22, tempA, new_size);
    strassen(tempA, b11, p2, new_size);

    // p3 = a11 * (b12 - b22)
    subtract_matrix(b12, b22, tempB, new_size);
    strassen(a11, tempB, p3, new_size);

    // p4 = a22 * (b21 - b11)
    subtract_matrix(b21, b11, tempB, new_size);
    strassen(a22, tempB, p4, new_size);

    // p5 = (a11 + a12) * b22
    add_matrix(a11, a12, tempA, new_size);
    strassen(tempA, b22, p5, new_size);

    // p6 = (a21 - a11) * (b11 + b12)
    subtract_matrix(a21, a11, tempA, new_size);
    add_matrix(b11, b12, tempB, new_size);
    strassen(tempA, tempB, p6, new_size);
```

```
    // p7 = (a12 - a22) * (b21 + b22)
    subtract_matrix(a12, a22, tempA, new_size);
    add_matrix(b21, b22, tempB, new_size);
    strassen(tempA, tempB, p7, new_size);

    // Calculating c11, c12, c21, c22
    add_matrix(subtract_matrix(add_matrix(p1, p4, tempA, new_size),
p5, tempB, new_size), p7, c11, new_size);
    add_matrix(p3, p5, c12, new_size);
    add_matrix(p2, p4, c21, new_size);
    add_matrix(subtract_matrix(add_matrix(p1, p3, tempA, new_size),
p2, tempB, new_size), p6, c22, new_size);

    // Grouping the results into the output matrix c
    for (int i = 0; i < new_size; i++) {
        for (int j = 0; j < new_size; j++) {
            c[i][j] = c11[i][j];
            c[i][j + new_size] = c12[i][j];
            c[i + new_size][j] = c21[i][j];
            c[i + new_size][j + new_size] = c22[i][j];
        }
    }

    // Freeing all dynamically allocated memory
    free_matrix(a11, new_size);
    free_matrix(a12, new_size);
    free_matrix(a21, new_size);
    free_matrix(a22, new_size);
    free_matrix(b11, new_size);
    free_matrix(b12, new_size);
    free_matrix(b21, new_size);
    free_matrix(b22, new_size);
    free_matrix(c11, new_size);
    free_matrix(c12, new_size);
    free_matrix(c21, new_size);
    free_matrix(c22, new_size);
    free_matrix(p1, new_size);
    free_matrix(p2, new_size);
    free_matrix(p3, new_size);
    free_matrix(p4, new_size);
    free_matrix(p5, new_size);
    free_matrix(p6, new_size);
    free_matrix(p7, new_size);
    free_matrix(tempA, new_size);
    free_matrix(tempB, new_size);
}

int main() {
    int n = 4; // Size of the matrix (must be a power of 2)
    int **a = allocate_matrix(n);
    int **b = allocate_matrix(n);
    int **c = allocate_matrix(n);

    // Initialize matrices a and b with some values
    // For example: Fill with random numbers or fixed pattern
    // Here we use a simple increasing pattern for demonstration
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            a[i][j] = i * n + j;
            b[i][j] = j * n + i;
        }
    }

    strassen(a, b, c, n);

    // Output the resulting matrix
    printf("Resulting Matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%d ", c[i][j]);
        }
        printf("\n");
    }

    // Free all matrices
    free_matrix(a, n);
    free_matrix(b, n);
    free_matrix(c, n);

    return 0;
}
```

SumOFSubsets

```
#include <stdio.h>
#include <stdlib.h>

// Function prototypes
void findSubsets(int idx, int n, int target, int currentSum, int*
elements, int* subset, int subsetSize);
void printSubset(int* subset, int subsetSize);

// Main function to drive the program
int main() {
    int n, target;
    printf("Enter the number of elements in the set: ");
    scanf("%d", &n);

    int* elements = malloc(n * sizeof(int));
    printf("Enter the elements of the set: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &elements[i]);
    }

    printf("Enter the target sum: ");
    scanf("%d", &target);

    int* subset = malloc(n * sizeof(int));
    printf("Subsets that sum to %d are:\n", target);
    findSubsets(0, n, target, 0, elements, subset, 0);

    free(elements);
    free(subset);

    return 0;
}

// Recursive function to find subsets that sum to the target
void findSubsets(int idx, int n, int target, int currentSum, int*
elements, int* subset, int subsetSize) {
    if (currentSum == target) {
        printSubset(subset, subsetSize);
        return;
    }

    // Stop the recursion if the current sum exceeds the target or
if we've considered all elements
    if (currentSum > target || idx == n) {
        return;
    }

    // Include the current element and recurse
    subset[subsetSize] = elements[idx];
    findSubsets(idx + 1, n, target, currentSum + elements[idx],
elements, subset, subsetSize + 1);

    // Exclude the current element and recurse
    findSubsets(idx + 1, n, target, currentSum, elements, subset, 
subsetSize);
}

// Function to print the found subset
void printSubset(int* subset, int subsetSize) {
    printf("{");
    for (int i = 0; i < subsetSize; i++) {
        printf("%d ", subset[i]);
    }
    printf("}\n");
}
```

TSP

```
#include <stdio.h>
#include <limits.h>

#define N 10  // Maximum number of cities
#define VISITED_ALL ((1<<N) - 1)

int dist[N][N];  // Distance matrix
int dp[N][1<<N]; // DP table

// Recursive function to solve the TSP using dynamic programming and
bitmasking
int tsp(int mask, int pos, int n) {
    if (mask == VISITED_ALL) {
        return dist[pos][0];  // Return to the starting city
    }
    if (dp[pos][mask] != -1) {
       return dp[pos][mask];
    }

    // Initialize with a large number
    int ans = INT_MAX;
    // Try to go to an unvisited city
    for (int city = 0; city < n; city++) {
        if ((mask & (1 << city)) == 0) {
            int newAns = dist[pos][city] + tsp(mask | (1 << city),
city, n);
            ans = ans < newAns ? ans : newAns;
        }
    }

    return dp[pos][mask] = ans;
}

int main() {
    int n;
    printf("Enter the number of cities: ");
    scanf("%d", &n);

    printf("Enter the distance matrix:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            scanf("%d", &dist[i][j]);
        }
    }

    // Initialize dp array
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < (1<<N); j++) {
            dp[i][j] = -1;
        }
    }
    printf("The minimum travelling cost is: %d\n", tsp(1, 0, n));

    return 0;
}
```

```c
#include <stdio.h>

typedef struct {
    int id;       // Job ID
    int deadline; // Deadline of the job
    int profit;   // Profit of the job
} Job;

void scheduleJobs(Job jobs[], int n) {

    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (jobs[j].deadline > jobs[j +
1].deadline) {
                // Swap jobs
                Job temp = jobs[j];
                jobs[j] = jobs[j + 1];
                jobs[j + 1] = temp;
            }
        }
    }

    int slots[n];
    for (int i = 0; i < n; i++) {
        slots[i] = -1; // -1 indicates slot is
empty
    }

    int totalProfit = 0;
    for (int i = 0; i < n; i++) {
        for (int j = (jobs[i].deadline - 1); j >=
0; j--) {
            if (slots[j] == -1) {
                slots[j] = jobs[i].id;
                totalProfit += jobs[i].profit;
                break;
            }
        }
    }
    printf("Scheduled Jobs: ");
    for (int i = 0; i < n; i++) {
        if (slots[i] != -1) {
            printf("%d ", slots[i]);
        }
    }
    printf("\nTotal Profit: %d\n", totalProfit);
}

int main() {

    Job jobs[] = {
        {1, 4, 20},
        {2, 1, 10},
        {3, 1, 40},
        {4, 1, 30}
    };
    int n = sizeof(jobs) / sizeof(jobs[0]);

    scheduleJobs(jobs, n);

    return 0;
}
```