Given two strings, find the minimum number of edits required to convert one string to another

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAXLEN 1000 // maximum length of the strings

int min(int a, int b, int c) {

   int min = a;

   if (b < min) {

      min = b;

   }

   if (c < min) {

      min = c;

   }

   return min;

}

int edit_distance(char *s1, char *s2) {

   int len1 = strlen(s1);

   int len2 = strlen(s2);

   int dist[len1 + 1][len2 + 1];

   for (int i = 0; i <= len1; i++) {

      dist[i][0] = i;

   }

   for (int j = 0; j <= len2; j++) {

      dist[0][j] = j;

   }

   for (int i = 1; i <= len1; i++) {

      for (int j = 1; j <= len2; j++) {

         int cost = (s1[i - 1] == s2[j - 1]) ? 0 : 1;

         dist[i][j] = min(dist[i - 1][j] + 1, dist[i][j - 1] + 1, dist[i - 1][j - 1] + cost);

      }
```

```
    }
    return dist[len1][len2];
}

int main() {
    char s1[MAXLEN], s2[MAXLEN];
    printf("Enter the first string: ");
    scanf("%s", s1);
    printf("Enter the second string: ");
    scanf("%s", s2);
    int dist = edit_distance(s1, s2);
    printf("The minimum number of edits required to convert %s to %s is %d.\n", s1, s2, dist);
    return 0;
}
```

Find a subset of a given set S = {sl, s2,....., sn} of n positive integers whose sum is equal to a given positive integer d. For example, if S= {1, 2, 5, 6, 8} and d = 9 there are two solutions {1, 2, 6} and {1,8}. A suitable message is to be displayed if the given problem instance doesn't have a solution.

```
#include <stdio.h>
int main() {
    int S[] = {1, 2, 5, 6, 8}; // given set
    int n = sizeof(S) / sizeof(S[0]); // size of set
    int d = 9; // given sum
    int subset[n]; // to store the subset
    int count = 0; // number of elements in subset
    int sum = 0; // sum of elements in subset
    int i, j;

    // iterate over all possible subsets
    for (i = 0; i < (1 << n); i++) {
        sum = 0;
        count = 0;
        for (j = 0; j < n; j++) {
            if (i & (1 << j)) {
                sum += S[j];
                subset[count++] = S[j];
            }
        }
        // if subset sum equals given sum, print the subset and exit
        if (sum == d) {
            printf("Subset found: ");
            for (j = 0; j < count; j++) {
                printf("%d ", subset[j]);
            }
```

```
        printf("\n");
        return 0;
      }
    }
    // if no subset found, display a message
    printf("No subset found.\n");
    return 0;
}
```

Implement N-Queens problem using backtracking

```
#include <stdio.h>
#include <stdbool.h>
#define N 8 // change N for different board sizes
void printSolution(int board[N][N]) {
   int i, j;
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         printf("%d ", board[i][j]);
      }
      printf("\n");
   }
}
bool isSafe(int board[N][N], int row, int col) {
   int i, j;
   // check row
   for (i = 0; i < col; i++) {
      if (board[row][i]) {
         return false;
      }
   }
   // check upper diagonal
   for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
      if (board[i][j]) {
         return false;
      }
```

```
    }
    // check lower diagonal
    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j]) {
            return false;
        }
    }
    return true;
}
bool solveNQueens(int board[N][N], int col) {
    int i;
    // base case: all queens have been placed
    if (col == N) {
        printSolution(board);
        return true;
    }
    // try placing queen in each row of current column
    bool result = false;
    for (i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            // recursively solve for next column
            result = solveNQueens(board, col + 1) || result;
            board[i][col] = 0; // backtrack if no solution found
        }
    }
```

```
    return result;
}

int main() {
    int board[N][N] = {0};
    if (!solveNQueens(board, 0)) {
        printf("No solution found.\n");
    }
    return 0;
}
```

Solve the longest common subsequence problem using dynamic programming .

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


#define MAX(x, y) ((x) > (y) ? (x) : (y))


// Function to find the length of the longest common subsequence of two strings

int LCS_length(char *X, char *Y, int m, int n)

{

   // Create a table to store the lengths of longest common subsequences

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


   // Fill the table in bottom-up manner

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

      for (int 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]);

         }

      }
```

```
    }

    // Return the length of the longest common subsequence
    return L[m][n];
}

int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

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

    int length = LCS_length(X, Y, m, n);

    printf("Length of Longest Common Subsequence: %d\n", length);

    return 0;
}
```

Find the length of the longest subsequence in a given array of integers such that all elements of the subsequence are sorted in strictly ascending order

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

#define MAX(x, y) ((x) > (y) ? (x) : (y))

// Function to find the length of the longest increasing subsequence in an array
int LIS(int arr[], int n)
{
   int lis[n];
   int max_len = 0;

   // Initialize the LIS of each element as 1
   for (int i = 0; i < n; i++) {
      lis[i] = 1;
   }

   // Compute the LIS of each element
   for (int i = 1; i < n; i++) {
      for (int j = 0; j < i; j++) {
         if (arr[i] > arr[j]) {
```

```
            lis[i] = MAX(lis[i], lis[j] + 1);
         }
      }
   }

   // Find the maximum LIS
   for (int i = 0; i < n; i++) {
      if (lis[i] > max_len) {
         max_len = lis[i];
      }
   }

   return max_len;
}

int main()
{
   int arr[] = {10, 22, 9, 33, 21, 50, 41, 60};
   int n = sizeof(arr) / sizeof(arr[0]);
   int length = LIS(arr, n);
   printf("Length of Longest Increasing Subsequence: %d\n", length);
   return 0;
}
```

Implement graph coloring problem using backtracking

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

#define MAX_VERTICES 100

// Function to check if it is safe to color the vertex v with color c
bool isSafe(int v, int graph[][MAX_VERTICES], int color[], int c, int V) {
    // Check if any adjacent vertex of v has the same color
    for (int i = 0; i < V; i++) {
        if (graph[v][i] && color[i] == c) {
            return false;
        }
    }
    return true;
}

// Function to solve the graph coloring problem using backtracking
bool graphColoring(int graph[][MAX_VERTICES], int color[], int m, int
v, int V) {
    // If all vertices are assigned a color, return true
    if (v == V) {
        return true;
    }
```

```
    // Try all possible colors for vertex v
    for (int c = 1; c <= m; c++) {
        // Check if it is safe to color vertex v with color c
        if (isSafe(v, graph, color, c, V)) {
            // Assign color c to vertex v
            color[v] = c;

            // Recursively color the remaining vertices
            if (graphColoring(graph, color, m, v + 1, V)) {
                return true;
            }

            // If coloring with color c doesn't lead to a solution, backtrack
            color[v] = 0;
        }
    }

    // If no color can be assigned to vertex v, return false
    return false;
}

int main() {
    int graph[MAX_VERTICES][MAX_VERTICES] = {
```

```
        {0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0}
    };
    int V = 4; // Number of vertices

    int color[MAX_VERTICES] = {0}; // Initialize all colors to 0
    int m = 3; // Number of colors

    if (graphColoring(graph, color, m, 0, V)) {
        printf("Graph is colorable with %d colors. The colors of the
vertices are:\n", m);
        for (int i = 0; i < V; i++) {
            printf("Vertex %d: Color %d\n", i, color[i]);
        }
    } else {
        printf("Graph is not colorable with %d colors.\n", m);
    }

    return 0;
}
```

Apply dynamic programming methodology to find all pairs shortest path of a directed graph using Floyd's algorithm

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

#define MAX_VERTICES 100

// Function to find all pairs shortest path using Floyd's algorithm
void allPairsShortestPath(int graph[][MAX_VERTICES], int V) {
    // Initialize the distance matrix
    int dist[MAX_VERTICES][MAX_VERTICES];
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
        }
    }

    // Perform the dynamic programming algorithm
    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][j] >
dist[i][k] + dist[k][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
```

```
            }
        }
    }

    // Print the distance matrix
    printf("All pairs shortest path:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INT_MAX) {
                printf("INF ");
            } else {
                printf("%d ", dist[i][j]);
            }
        }
        printf("\n");
    }
}

int main() {
    int graph[MAX_VERTICES][MAX_VERTICES] = {
        {0, 5, INT_MAX, 10},
        {INT_MAX, 0, 3, INT_MAX},
        {INT_MAX, INT_MAX, 0, 1},
        {INT_MAX, INT_MAX, INT_MAX, 0}
    };
    int V = 4; // Number of vertices
```

```
    allPairsShortestPath(graph, V);

    return 0;
}
```

Implement matrix chain multiplication and find the optimal sequence of parentheses.

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

#define MAX_DIMENSIONS 100

// Function to print the optimal sequence of parentheses
void printOptimalParentheses(int s[][MAX_DIMENSIONS], int i, int j) {
    if (i == j) {
        printf("A%d", i);
    } else {
        printf("(");
        printOptimalParentheses(s, i, s[i][j]);
        printOptimalParentheses(s, s[i][j] + 1, j);
        printf(")");
    }
}

// Function to perform matrix chain multiplication and find the
optimal sequence of parentheses
void matrixChainMultiplication(int dimensions[], int n) {
    // Initialize the cost and split matrices
```

```
    int cost[MAX_DIMENSIONS][MAX_DIMENSIONS];
    int split[MAX_DIMENSIONS][MAX_DIMENSIONS];
    for (int i = 0; i < n; i++) {
        cost[i][i] = 0;
    }

    // Perform the dynamic programming algorithm
    for (int l = 2; l <= n; l++) {
        for (int i = 0; i <= n - l; i++) {
            int j = i + l - 1;
            cost[i][j] = INT_MAX;
            for (int k = i; k < j; k++) {
                int q = cost[i][k] + cost[k+1][j] + dimensions[i] *
dimensions[k+1] * dimensions[j+1];
                if (q < cost[i][j]) {
                    cost[i][j] = q;
                    split[i][j] = k;
                }
            }
        }
    }

    // Print the minimum cost and the optimal sequence of
parentheses
```

```
    printf("Minimum cost: %d\n", cost[0][n-1]);
    printf("Optimal sequence of parentheses: ");
    printOptimalParentheses(split, 0, n-1);
    printf("\n");
}

int main() {
    int dimensions[] = {5, 10, 3, 12, 5, 50, 6}; // Dimensions of the
matrices
    int n = 6; // Number of matrices

    matrixChainMultiplication(dimensions, n);

    return 0;
}
```

Apply dynamic programming methodology to implement 0/1 Knapsack problem

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

#define MAX_ITEMS 100
#define MAX_CAPACITY 1000

// Function to solve the 0/1 Knapsack problem using dynamic
programming
int knapsack(int values[], int weights[], int n, int capacity) {
    int dp[MAX_ITEMS + 1][MAX_CAPACITY + 1];
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= capacity; w++) {
            if (i == 0 || w == 0) {
                dp[i][w] = 0;
            } else if (weights[i-1] <= w) {
                dp[i][w] = (values[i-1] + dp[i-1][w-weights[i-1]]) > dp[i-1][w]
? (values[i-1] + dp[i-1][w-weights[i-1]]) : dp[i-1][w];
            } else {
                dp[i][w] = dp[i-1][w];
            }
        }
```

```c
    }
    return dp[n][capacity];
}

int main() {
    int values[MAX_ITEMS] = {60, 100, 120, 140}; // Value of each item
    int weights[MAX_ITEMS] = {10, 20, 30, 40}; // Weight of each item
    int n = 4; // Number of items
    int capacity = 50; // Capacity of the knapsack

    int max_value = knapsack(values, weights, n, capacity);

    printf("Maximum value: %d\n", max_value);

    return 0;
}
```

Implement minimum spanning tree using Prim's algorithm and analyse its time complexity.

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

#define MAX_VERTICES 100

int minKey(int key[], int mstSet[], int V) {
    int min = INT_MAX, min_index;
    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[MAX_VERTICES][MAX_VERTICES], int 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[MAX_VERTICES][MAX_VERTICES], int V) {
    int parent[MAX_VERTICES];
    int key[MAX_VERTICES];
    int mstSet[MAX_VERTICES];

    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, V);
        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, V);
}

int main() {
    int graph[MAX_VERTICES][MAX_VERTICES] = {
        {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, 5);

    return 0;
}
```