**Lecture Notes on Edit Distance and Minimum Cost**

**1. Introduction to Edit Distance**

Edit Distance, also known as **Levenshtein Distance**, is a dynamic programming problem that measures the minimum number of operations required to convert one string into another. The operations allowed are:

* Insertion
* Deletion
* Substitution

It has wide applications in areas such as:

* Spell checking
* DNA sequencing
* Natural language processing
* Plagiarism detection

**2. Problem Definition: Edit Distance**

Given two strings str1 and str2, you need to find the minimum number of operations (insert, delete, or replace) required to convert str1 into str2.

**Step-by-Step Procedure (Dynamic Programming)**

1. **Create a DP table** of size (m+1) x (n+1) where m is the length of str1 and n is the length of str2.
2. **Base Case**:
   * If one string is empty, the minimum number of operations is the length of the other string (inserts or deletes).
3. **Recursive Relation**:
   * If the characters of the two strings match, no operation is required.
   * If they don't match, consider the minimum cost from:
     + Insertion
     + Deletion
     + Substitution
4. **Final Result**: The value at dp[m][n] contains the minimum edit distance.

**Example:**

Let's take an example to convert kitten to sitting.

str1 = "**kitten**"

str2 = "**sitting**"

| **Step** | **Operation** | **Result** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| 1 | Replace 'k' with 's' | "sitten" |

|  |  |  |
| --- | --- | --- |
| 2 | Replace 'e' with 'i' | "sittin" |

|  |  |  |
| --- | --- | --- |
| 3 | Insert 'g' at the end | "sitting" |

**The total number of operations = 3.  
  
Code:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to calculate the minimum edit distance

int editDistance(string str1, string str2) {

    int m = str1.length();

    int n = str2.length();

    // Create a DP table to store the minimum edit distances

    vector<vector<int>> dp(m + 1, vector<int>(n + 1));

    // Step 1: Initialize the base cases

    for (int i = 0; i <= m; i++) {

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

            if (i == 0) {

                // If first string is empty, insert all characters of second string

                dp[i][j] = j;

            } else if (j == 0) {

                // If second string is empty, remove all characters of first string

                dp[i][j] = i;

            } else {

                // Step 2: Calculate the minimum cost of edit operations

                if (str1[i - 1] == str2[j - 1]) {

                    // Characters are the same, no need to edit

                    dp[i][j] = dp[i - 1][j - 1];

                } else {

                    // Find the minimum cost between insert, delete, or replace

                    dp[i][j] = 1 + min({dp[i - 1][j],    // Delete

                                        dp[i][j - 1],    // Insert

                                        dp[i - 1][j - 1] // Replace

                                       });

                }

            }

        }

    }

    // Return the final result stored in dp[m][n]

    return dp[m][n];

}

int main() {

    string str1 = "kitten";

    string str2 = "sitting";

    // Call the function and display the result

    cout << "The minimum edit distance is: " << editDistance(str1, str2) << endl;

    return 0;

}

### **Step-by-Step Code Explanation**

1. **Base Case Initialization**: The DP table is initialized for cases where one string is empty. For example, if str1 is empty, the number of insertions required is the length of str2, which is why dp[i][0] = i.
2. **Recursive Relation**: For each pair of characters from str1 and str2, if the characters are the same, no operation is required (dp[i][j] = dp[i-1][j-1]). If the characters differ, the cost is the minimum of insertion, deletion, or substitution, incremented by 1.
3. **Final Result**: The final edit distance is stored in dp[m][n] after the table is filled.

**MIN COST:**

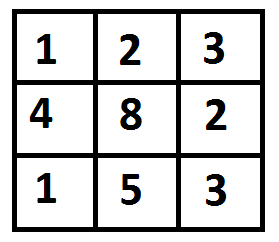
#### This problem is closely related to the Edit Distance but focuses on finding the minimum cost to reach a cell in a grid from a starting point. You can only move right, down, or diagonally, and each cell has a cost associated with it. **Problem Definition: Minimum Cost Path**

Given a cost matrix cost[][] and a position (M, N) in cost[][], write a function that returns cost of minimum cost path to reach (M, N) from (0, 0). Each cell of the matrix represents a cost to traverse through that cell. The total cost of a path to reach (M, N) is the sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1), and (i+1, j+1) can be traversed.

**Note:**You may assume that all costs are positive integers.

**Example:**

**Input:**



*The path with minimum cost is highlighted in the following figure.*

*The path is (0, 0) –> (0, 1) –> (1, 2) –> (2, 2). The cost of the path is 8 (1 + 2 + 2 + 3).*

**Output:**

A green line in a square with numbers

Description automatically generated

**Step-by-Step Procedure**

1. **Initialize a DP table** of the same size as the cost matrix.
2. **Base Case**:
   * The first cell's cost is the starting point.
   * The first row can only be reached by moving right.
   * The first column can only be reached by moving down.
3. **Recursive Case**: For each cell, the cost is the minimum of three possible moves:
   * From the left (moving right).
   * From above (moving down).
   * From diagonally above-left.
4. **Final Solution**: The final minimum cost is found at the bottom-right corner of the DP table.

**Code:**

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

// Function to find the minimum cost path

int minCost(vector<vector<int>> cost) {

    int m = cost.size();

    int n = cost[0].size();

    // Create a DP table to store the minimum costs

    vector<vector<int>> dp(m, vector<int>(n));

    // Initialize the first cell

    dp[0][0] = cost[0][0];

    // Initialize the first row

    for (int j = 1; j < n; j++) {

        dp[0][j] = dp[0][j - 1] + cost[0][j];

    }

    // Initialize the first column

    for (int i = 1; i < m; i++) {

        dp[i][0] = dp[i - 1][0] + cost[i][0];

    }

    // Fill the rest of the DP table

    for (int i = 1; i < m; i++) {

        for (int j = 1; j < n; j++) {

            dp[i][j] = cost[i][j] + min({dp[i - 1][j],    // From top

                                         dp[i][j - 1],    // From left

                                         dp[i - 1][j - 1] // From diagonal

                                        });

        }

    }

    // Return the minimum cost to reach the bottom-right corner

    return dp[m - 1][n - 1];

}

int main() {

    vector<vector<int>> cost = {{1, 2, 3},

                                {4, 8, 2},

                                {1, 5, 3}};

    // Call the function and display the result

    cout << "The minimum cost path is: " << minCost(cost) << endl;

    return 0;

}