# Documentation for Edit Distance Problem and Solution

## Problem Statement

The Edit Distance problem, also known as Levenshtein Distance, is a classic algorithmic problem that involves finding the minimum number of operations required to convert one string into another. The operations allowed are:

1. Insert a character
2. Delete a character
3. Replace a character

Given two strings, word1 and word2, the objective is to determine the minimum number of these operations needed to transform word1 into word2.

## Example 1

- **Input:** word1 = "horse", word2 = "ros"
- **Output:** 3
- **Explanation:**
    - horse -> rorse (replace 'h' with 'r')
    - rorse -> rose (remove 'r')
    - rose -> ros (remove 'e')

## Example 2

- **Input:** word1 = "intention", word2 = "execution"
- **Output:** 5

- **Explanation:**
    - intention -> inention (remove 't')
    - inention -> enention (replace 'i' with 'e')
    - enention -> exention (replace 'n' with 'x')
    - exention -> exection (replace 'n' with 'c')
    - exection -> execution (insert 'u')

## Constraints

- 0 <= word1.length, word2.length <= 500
- word1 and word2 consist of lowercase English letters.

## Solution Explanation

The solution to this problem uses dynamic programming (DP) to build a table that helps in calculating the minimum edit distance efficiently.

## Steps to Solve

1. **Initialize Variables:**

- Determine the lengths of the two words, m and n.
- Create a DP table dp of size (m+1) x (n+1) initialized to zeros.

2. **Base Case Initialization:**

- Fill the first row and the first column of the DP table:
- dp[i][0] = i for all i from 0 to m (representing converting a string to an empty string by deleting characters).
- dp[0][j] = j for all j from 0 to n (representing converting an empty string to a string by inserting characters).

3. **Fill the DP Table:**

- For each cell dp[i][j], determine the minimum number of operations required:
- If word1[i-1] == word2[j-1], no operation is needed (dp[i][j] = dp[i-1][j-1]).
- Otherwise, consider the three possible operations (insert, delete, replace) and take the minimum value:
- dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + 1)

4. **Return the Result:**

- The value at dp[m][n] will be the minimum edit distance between word1 and word2.

## Explanation of the Code

- **Initialization:** The lengths of word1 and word2 are stored in m and n. A 2D list dp of size (m+1) x (n+1) is created to store the minimum edit distances for substrings of word1 and word2.

- **Base Case Setup:** The first row and column of the DP table are initialized to represent converting to and from empty strings.

- **DP Table Filling:** The nested loops iterate through each character of word1 and word2, filling the DP table based on the minimum operations calculated.

- **Result Return:** The value at dp[m][n] represents the minimum number of operations required to convert word1 to word2.

This solution has a time complexity of O(m * n) and a space complexity of O(m * n), making it efficient for the given problem constraints.