72. Edit Distance

Hard String Dynamic Programming

Leetcode Link

Problem Description

The task is to find the minimum number of operations required to convert one string (`word1`) into another (`word2`). The only operations allowed are:

1. Inserting a character.
2. Deleting a character.
3. Replacing one character with another.

These operations can be applied in any order and any number of times, and the goal is to achieve this transformation with the least number of them.

Intuition

To solve this problem, we use a technique in computer science known as dynamic programming. The core idea is to break down the big problem into smaller subproblems and solve each of them just once, storing their solutions - often in a table - so that the next time the same subproblem occurs, instead of recomputing its solution, one simply looks it up in the table. This is particularly effective for problems where the same subproblems recur many times.

For our specific case, we construct a matrix `f` where each cell `f[i][j]` represents the minimum number of operations to convert the first `i` characters of `word1` into the first `j` characters of `word2`. The first row (`f[0][j]`) is initialized with the sequence `0, 1, 2, ..., n` because if `word1` is empty, the only option is to insert characters into it, and the number of operations equals the number of characters in `word2`. Similarly, the first column (`f[i][0]`) is `0, 1, 2, ..., m` because if `word2` is empty, the only option is to delete characters from `word1`.

The intuition for the recursive step is as follows:

- If the current characters in `word1` and `word2` are equal (`word1[i - 1] == word2[j - 1]`), no operation is needed, and the number of operations will be the same as it was for `i - 1` and `j - 1`.
- If they are not equal, we need to consider three possible operations:
  - Inserting (`f[i][j - 1] + 1`): We have matched up to `j - 1` of `word2`, and then by adding the `j`-th character of `word2`, we will match `j` characters. The number of operations is one more than it took to match `j - 1` characters.
  - Deleting (`f[i - 1][j] + 1`): If we remove the `i`-th character from `word1`, we fall back to the subproblem of matching `i - 1` characters of `word1` with `j` characters of `word2`, and again, this is one more operation than that subproblem.
  - Replacing (`f[i - 1][j - 1] + 1`): Here, we change the `i`-th character of `word1` to match the `j`-th character of `word2`. So, the number of operations is one more than the operations needed for `i - 1` and `j - 1`.

We take the minimum of these three options at each step, and the last cell `f[m][n]` will give us the minimum number of operations required to transform `word1` into `word2`.

Solution Approach

The approach to this problem is a classic example of Dynamic Programming (DP), which uses a 2D table to store solutions to subproblems. This memory storage is critical because many subproblems are solved multiple times, and storing their solutions significantly reduces computation time. This technique is known as memoization.

To implement this:

1. We initialize a 2D array `f` with `m+1` rows and `n+1` columns, where `m` is the length of `word1` and `n` is the length of `word2`. Each element `f[i][j]` represents the minimum number of operations needed to convert the first `i` characters of `word1` to the first `j` characters of `word2`.

2. We fill in the base cases:

   - The first row represents converting an empty `word1` into the first `j` characters of `word2`, which obviously requires `j` insertions. So, we set `f[0][j] = j` for all `j`.
   - The first column represents converting the first `i` characters of `word1` into an empty `word2`, which requires `i` deletions. Hence, `f[i][0] = i` for all `i`.

3. We iterate over the array starting from `f[1][1]` to fill in the remaining cells. At each cell `f[i][j]`, we decide the best option (minimum operations) based on whether the characters at positions `i-1` in `word1` and `j-1` in `word2` are the same.

4. The choice at each step is between:

   - Keeping the character if it's the same (`f[i-1][j-1]`) or
   - Performing one operation (delete, insert, or replace) to make the strings match up to that point.

5. We apply the following state transition equation:

```
f[i][j] = {
    f[i-1][j-1]                                 if word1[i-1] == word2[j-1]
    min(f[i-1][j], f[i][j-1], f[i-1][j-1]) + 1  otherwise
}
```

   - `f[i-1][j] + 1` represents deleting the `i`-th character from `word1`.
   - `f[i][j-1] + 1` means inserting the `j`-th character into `word1`.
   - `f[i-1][j-1] + 1` indicates replacing the `i`-th character of `word1` with the `j`-th character of `word2`.

6. After filling the DP table, the value at `f[m][n]` gives us the minimum number of operations required to convert `word1` to `word2`.

This algorithm's runtime complexity is `O(m * n)` because we have to fill a table with `m * n` cells, and the work for each cell is constant. The space complexity is also `O(m * n)` for the DP table.

Example Walkthrough

Let's consider a small example where we want to convert `word1 = "intention"` to `word2 = "execution"`. We will walk through the Dynamic Programming approach to illustrate the solution:

1. Initialize a 2D array `f` with 10 rows (since `word1` has 9 characters plus 1 for the empty prefix) and 10 columns (since `word2` has 9 characters plus 1 for the empty prefix).

2. Fill in the base cases:

   - The first row `f[0][j]` from `f[0][0]` to `f[0][9]` will be `0, 1, 2, ..., 9` as it takes `j` insertions to convert an empty `word1` into `word2[0..j-1]`.
   - The first column `f[i][0]` from `f[0][0]` to `f[9][0]` will be `0, 1, 2, ..., 9` as it takes `i` deletions to convert `word1[0..i-1]` into an empty `word2`.

3. Now, we iterate over the remaining cells starting from `f[1][1]`. We compare characters of `word1` and `word2` starting from first (`i-1` for `word1` and `j-1` for `word2`) and fill `f[i][j]` considering three cases:

   - If `word1[i-1] == word2[j-1]`, we copy the value from `f[i-1][j-1]` to `f[i][j]` (since no operation is needed).
   - Otherwise, we find the minimum of:
     - `f[i-1][j]` + 1 (delete case),
     - `f[i][j-1]` + 1 (insert case),
     - `f[i-1][j-1]` + 1 (replace case).

For the first non-base cell `f[1][1]`, since `word1[0]` is 'i' and `word2[0]` is 'e', they're not the same, so we compute:

```
f[1][1] = min(f[0][1], f[1][0], f[0][0]) + 1
        = min(1, 1, 0) + 1
        = 1
```

4. We continue this process for each cell. For instance:

```
f[3][2] (to convert "int" to "ex"):
- word1[2] is "t" and word2[1] is "x", so they are different.
- f[3][2] = min(f[2][2], f[3][1], f[2][1]) + 1
          = min(2, 3, 2) + 1
          = 3
```

5. After we have populated the entire array, we look at the last cell `f[9][9]` to find the minimum number of operations required to convert `word1` into `word2`. In this example, let's say the last cell value is 5 (as your specific DP table may vary during actual execution).

Thus, the answer is that it requires a minimum of 5 operations to transform "intention" into "execution" using the allowed operations.

Python Solution

```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        # Get the lengths of both words
        len_word1, len_word2 = len(word1), len(word2)

        # Initialize a table to store the edit distances
        # The table size will be (len_word1+1) x (len_word2+1)
        dp_table = [[0] * (len_word2 + 1) for _ in range(len_word1 + 1)]

        # Set up the initial state where converting an empty string to word2
        # requires adding all letters of word2
        for j in range(1, len_word2 + 1):
            dp_table[0][j] = j

        # Set up the state where converting word1 to an empty string
        # requires removing all letters of word1
        for i in range(1, len_word1 + 1):
            dp_table[i][0] = i

            for j in range(1, len_word2 + 1):
                # If the current characters match, take the previous best without these characters
                if word1[i - 1] == word2[j - 1]:
                    dp_table[i][j] = dp_table[i - 1][j - 1]
                else:
                    # If the characters don't match, consider all possible operations
                    # 1. Add a character (dp_table[i][j - 1])
                    # 2. Remove a character (dp_table[i - 1][j])
                    # 3. Replace a character (dp_table[i - 1][j - 1])
                    # Take the minimum of these possibilities and add 1 to represent the cost of the operation
                    dp_table[i][j] = min(
                        dp_table[i - 1][j],      # Deletion
                        dp_table[i][j - 1],      # Insertion
                        dp_table[i - 1][j - 1]   # Substitution
                    ) + 1

        # The answer is in the bottom-right cell of the table
        # It represents the minimum edit distance between the two full words
        return dp_table[len_word1][len_word2]

```

Java Solution

```java
class Solution {
    public int minDistance(String word1, String word2) {
        // Lengths of the input strings
        int lenWord1 = word1.length();
        int lenWord2 = word2.length();

        // Create a 2D array to store the subproblem results
        int[][] dpTable = new int[lenWord1 + 1][lenWord2 + 1];

        // Initialize the first column, representing insertions needed to transform an empty string into word2
        for (int indexWord2 = 1; indexWord2 <= lenWord2; ++indexWord2) {
            dpTable[0][indexWord2] = indexWord2;
        }

        // Fill out the dpTable for all subproblems
        for (int indexWord1 = 1; indexWord1 <= lenWord1; ++indexWord1) {
            // First row represents deletions needed to transform word1 into an empty string
            dpTable[indexWord1][0] = indexWord1;

            for (int indexWord2 = 1; indexWord2 <= lenWord2; ++indexWord2) {
                // If the characters are the same, take the value from the diagonal (no operation needed)
                if (word1.charAt(indexWord1 - 1) == word2.charAt(indexWord2 - 1)) {
                    dpTable[indexWord1][indexWord2] = dpTable[indexWord1 - 1][indexWord2 - 1];
                } else {
                    // If the characters are different, take the minimum operations from left (insert), top (delete), or diagonal (re
                    int insertOps = dpTable[indexWord1][indexWord2 - 1];
                    int deleteOps = dpTable[indexWord1 - 1][indexWord2];
                    int replaceOps = dpTable[indexWord1 - 1][indexWord2 - 1];

                    dpTable[indexWord1][indexWord2] = Math.min(insertOps, Math.min(deleteOps, replaceOps)) + 1;
                }
            }
        }

        // The bottom-right cell gives the final result
        return dpTable[lenWord1][lenWord2];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>

class Solution {
public:
    int minDistance(string word1, string word2) {
        int lengthWord1 = word1.size(), lengthWord2 = word2.size();

        // Create a DP table with dimensions (lengthWord1+1) x (lengthWord2+1)
        std::vector<std::vector<int>> dpTable(lengthWord1 + 1, std::vector<int>(lengthWord2 + 1));

        // Initialize the first column of the DP table which represents
        // the number of operations required to convert an empty string to word2
        for (int j = 0; j <= lengthWord2; ++j) {
            dpTable[0][j] = j;
        }

        // Fill out the DP table
        for (int i = 1; i <= lengthWord1; ++i) {
            // The first row of the DP table represents the number of operations
            // required to convert word1 to an empty string
            dpTable[i][0] = i;

            for (int j = 1; j <= lengthWord2; ++j) {
                // If characters at current position in both words are equal,
                // take the value from the previous top-left diagonal cell,
                // as no operation is required
                if (word1[i - 1] == word2[j - 1]) {
                    dpTable[i][j] = dpTable[i - 1][j - 1];
                } else {
                    // Otherwise, use the minimum value from the cell to the left (insert),
                    // above (delete) or top-left diagonal (replace), plus one for the current operation
                    dpTable[i][j] = std::min({
                        dpTable[i - 1][j],      // Deletion
                        dpTable[i][j - 1],      // Insertion
                        dpTable[i - 1][j - 1]   // Replacement
                    }) + 1;
                }
            }
        }

        // The bottom-right cell of the DP table contains the final answer
        return dpTable[lengthWord1][lengthWord2];
    }
};

```

Typescript Solution

```typescript
function minDistance(word1: string, word2: string): number {
    const lenWord1 = word1.length;
    const lenWord2 = word2.length;
    // Create a 2D array to hold the minimum edit distances.
    const dp: number[][] = Array.from(Array(lenWord1 + 1), () => new Array(lenWord2 + 1).fill(0));

    // Initialize the first row of the matrix.
    for (let col = 1; col <= lenWord2; ++col) {
        dp[0][col] = col;
    }
    // Initialize the first column of the matrix.
    for (let row = 1; row <= lenWord1; ++row) {
        dp[row][0] = row;
        for (let col = 1; col <= lenWord2; ++col) {
            // Check if the current characters are the same.
            if (word1[row - 1] === word2[col - 1]) {
                dp[row][col] = dp[row - 1][col - 1];
            } else {
                // If not, find the minimum cost among deletion, insertion, and replacement.
                dp[row][col] = Math.min(
                    dp[row - 1][col], // Deletion (from word1 to word2).
                    dp[row][col - 1], // Insertion (from word1 to word2).
                    dp[row - 1][col - 1] // Replacement (from word1 to word2).
                ) + 1;
            }
        }
    }
    // The bottom-right cell contains the final minimum edit distance.
    return dp[lenWord1][lenWord2];
}

```

Time and Space Complexity

The provided code snippet is an implementation of the dynamic programming approach to solve the problem of finding the minimum number of operations required to convert one word into another, where operations can be insertion, deletion, or substitution of a single character.

Time Complexity: The time complexity of this algorithm is `O(m * n)` where `m` is the length of `word1` and `n` is the length of `word2`. This time complexity arises because the algorithm iterates through all characters of `word1` using the variable `i` and all characters of `word2` using the variable `j`. For each pair of characters `(i, j)`, a constant amount of work is done to compute `f[i][j]`. Since the two for-loops are nested, each of the `m * n` pairs is considered exactly once, leading to the overall time complexity of `O(m * n)`.

Space Complexity: The space complexity of the algorithm is also `O(m * n)` due to the utilization of a two-dimensional array `f` that has `(m + 1) * (n + 1)` elements. Each element in `f` represents the minimum number of operations required to convert the first `i` characters of `word1` to the first `j` characters of `word2`. Since the array `f` has a size proportional to the product of `m` and `n`, the space complexity is `O(m * n)`.