583. Delete Operation for Two Strings

Dynamic Programming

Problem Description

String

The problem entails figuring out the least number of steps necessary to equalize two strings, word1 and word2. A single step consists of deleting one character from either one of the strings. This procedure is repeated until both strings are identical. The goal is to identify the minimum number of such deletions to make the two strings match.

Intuition

Medium

strings. The LCS is the longest sequence of characters that appear in both strings in the same order, possibly with other characters in between. Once we find the LCS, we know that the characters not part of the LCS need to be deleted to make the strings the same. To find the LCS, we can use dynamic programming. The main idea is to construct a 2D array dp where each element dp[i][j]

The intuition behind solving this task lies in the concept of finding the longest common subsequence (LCS) between the two

represents the length of the LCS between word1 up to index i and word2 up to index j. If the characters at word1[i] and word2[j] match, it means we can extend the LCS by one more character, so we take the LCS up to word1[i-1] and word2[j-1]. If they do not match, we take the longer of the LCSs without the current character of either word1 or word2. The edges of the array are initialized to the index values, representing the need to delete all previous characters in a string to match an empty string.

Once the dp array is fully populated, the value at dp[m][n] (where m and n are the lengths of word1 and word2, respectively) will

give us the length of the LCS. The minimum number of deletions required is then the sum of the lengths of the two input strings

The solution adopts a dynamic programming approach to efficiently compute the number of necessary deletion steps. Dynamic

minus twice the length of the LCS, which accounts for every character not in the LCS needing to be deleted from both strings. Solution Approach

programming is a method for solving complex problems by breaking them down into simpler subproblems and storing the results of those subproblems to avoid redundant computations.

needed.

dp = [

Initialize the 2D array dp with dimensions $(m+1) \times (n+1)$, where m is the length of word1 and n is the length of word2. This array will hold the lengths of LCS for different substrings of word1 and word2. We add 1 to include the empty string cases as well.

Fill the first row and first column of dp with increasing indices. This takes into account that converting any string to an empty

Here is the breakdown of the implementation:

- string requires a number of deletions equal to the length of the string. Loop through the matrix starting at dp[1][1] and decide for each pair (i, j):
- ∘ If word1[i 1] is equal to word2[j 1], then dp[i][j] is set to dp[i 1][j 1] as the characters match and do not need to be deleted (we extend the LCS). If the characters do not match, we look at two scenarios: deleting the last character from word1 or word2. The value of dp[i][j] is set to the

minimum of the two possibilities plus one (for the deletion step): dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j - 1]).

full problem by adding one character at a time and determining what the best decision is at each step.

After completely filling in the dp table, the value at dp[m] [n] is the minimum number of deletion steps necessary for the two words to become the same. This is because dp[m][n] represents the size of the LCS, and by subtracting this from the total

length of the two strings, we get the number of characters not part of the LCS, which is exactly the number of deletions

Return dp [m] [n] as the solution. The algorithm uses a 2D array for <u>dynamic programming</u>, which is a common data structure for storing intermediary results in

dynamic programming tasks. The pattern used is to build up the solution from the smallest subproblems (empty strings) to the

Let's walk through the solution approach with a small example where word1 = "sea" and word2 = "eat". The goal is to find the

Example Walkthrough

 $[1, 0, 0, 0], // "s" \rightarrow ""$

[2, 0, 0, 0], // "se" -> ""

[3, 0, 0, 0] // "sea" -> ""

minimum number of deletions to make word1 equal to word2.

dimensions (4×4) to include the empty string cases.

Loop through the dp array and fill it following the rules:

Fill the first row and column with increasing indices, reflecting the deletion count to match an empty string. dp = [[0, 1, 2, 3], // "" -> ""

1. Initialize the 2D array dp. Given that m = 3 (the length of "sea") and n = 3 (the length of "eat"), the dp array will have

 If word1[i - 1] equals word2[j - 1], set dp[i][j] to dp[i - 1][j - 1]. If they do not match, set dp[i][j] to the minimum of dp[i − 1][j] and dp[i][j − 1] plus one.

become the same with a minimum of 2 deletion steps (m + n - 2 * dp[3][3] = 3 + 3 - 2 * 1 = 4).

The answer is 4, which means that 4 deletions are necessary for word1 and word2 to become the same.

The last cell is '1' since 'a' at the end of both words matches, so we carry over the length of LCS without 'a', which was 0 (since "se" \rightarrow "e" required no LCS extensions), and add 1 for the common 'a'. After filling in the dp table, the last cell dp[3][3] now contains the length of the LCS. This represents that the strings can

to calculate the minimal number of deletions required to equalize the provided strings.

Initialize the DP table with dimensions (length_word1+1) x (length_word2+1)

dp_table[i][0] = i # Distance of converting word1 to an empty string

If the characters are the same, no operation is required

dp_table = [[0] * (length_word2 + 1) for _ in range(length_word1 + 1)]

Base cases: fill out the first row and column of the DP table

dp_table[i][j] = dp_table[i - 1][j - 1]

for i in range(1, length_word1 + 1):

for j in range(1, length_word2 + 1):

for j in range(1, length_word2 + 1):

Take the previous state's value

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

// Function to find the minimum distance between two strings.

int length1 = word1.size(); // Length of the first string

int length2 = word2.size(); // Length of the second string

// 2D vector to memorize distances (Dynamic Programming table)

vector<vector<int>> distanceMatrix(length1 + 1, vector<int>(length2 + 1));

dp[i][j] = 1 + Math.min(dp[i - 1][j], // Deletion

dp[i][j-1]); // Insertion

// The edit distance between 'word1' and 'word2' is in the bottom-right corner of the DP table

// Initialize the first column (all the distances from an empty string to prefixes of word1)

// Initialize the first row (all the distances from an empty string to prefixes of word2)

} else {

return dp[lenWord1][lenWord2];

int minDistance(string word1, string word2) {

for (int i = 1; i <= length1; ++i) {</pre>

for (int j = 1; j <= length2; ++j) {</pre>

distanceMatrix[0][j] = j;

// Minimum number of operations required

Get the lengths of both words

for i in range(1, length_word1 + 1):

for j in range(1, length_word2 + 1):

for i in range(1, length_word1 + 1):

for j in range(1, length_word2 + 1):

Take the previous state's value

Fill out the DP table

def minDistance(self, word1: str, word2: str) -> int:

length_word1, length_word2 = len(word1), len(word2)

class Solution:

distanceMatrix[i][0] = i;

if word1[i - 1] == word2[j - 1]:

Following this logic, the array is updated as shown below (explanations in comments):

Python

In conclusion, using dynamic programming, the algorithm iteratively finds the longest common subsequence and uses its length

class Solution: def minDistance(self, word1: str, word2: str) -> int: # Get the lengths of both words length_word1, length_word2 = len(word1), len(word2)

dp_table[0][j] = j # Distance of converting an empty string to word2 # Fill out the DP table for i in range(1, length_word1 + 1):

Solution Implementation

```
# If the characters are different, consider the minimum of
                # (deletion from word1, insertion into word1)
                # Add 1 representing the one operation needed
                else:
                    dp_table[i][j] = 1 + min(dp_table[i - 1][j], # Delete character from word1
                                               dp_table[i][j - 1]) # Insert character into word1
       # The answer is in the cell (length_word1, length_word2)
        return dp_table[-1][-1]
Java
class Solution {
    public int minDistance(String word1, String word2) {
        int lenWord1 = word1.length();
        int lenWord2 = word2.length();
       // 'dp' table where 'dp[i][j]' will be the edit distance of first 'i' characters of 'word1' and first 'j' characters of '
       int[][] dp = new int[lenWord1 + 1][lenWord2 + 1];
       // Initialize the first column, which represents the edits required to convert 'word1' substrings into an empty 'word2'
        for (int i = 1; i <= lenWord1; ++i) {</pre>
            dp[i][0] = i;
       // Initialize the first row, which represents the edits required to convert an empty 'word1' into substrings of 'word2'
        for (int j = 1; j <= lenWord2; ++j) {</pre>
            dp[0][j] = j;
       // Fill in the rest of the DP table
        for (int i = 1; i <= lenWord1; ++i) {</pre>
            for (int j = 1; j <= lenWord2; ++j) {</pre>
                // If the current characters of 'word1' and 'word2' match, no additional edit is required, take the diagonal valu
                if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
```

// If the characters don't match, consider the minimum of the cell to the left or above plus one for the edit

C++

public:

class Solution {

```
// Compute distances using the bottom-up approach
        for (int i = 1; i <= length1; ++i) {</pre>
            for (int j = 1; j <= length2; ++j) {</pre>
                // If characters at current positions in both strings are the same
                if (word1[i - 1] == word2[j - 1]) {
                    // No operation is needed, the distance is the same as for the previous characters
                    distanceMatrix[i][j] = distanceMatrix[i - 1][j - 1];
                } else {
                    // Characters are different, consider the minimum of the operations:
                    // (Replace) Take the previous distance where both strings have one less character
                    // (Add/Delete) Take the min of either deleting a character from word1 or adding a character to word2
                    distanceMatrix[i][j] = 1 + min(distanceMatrix[i - 1][j], distanceMatrix[i][j - 1]);
        // The distance from word1 to word2 is stored in the last cell
        return distanceMatrix[length1][length2];
};
TypeScript
function minDistance(word1: string, word2: string): number {
    // Lengths of both words
   const word1Length = word1.length;
    const word2Length = word2.length;
    // Initialize a 2D array for dynamic programming
   const dpMatrix = Array.from({ length: word1Length + 1 }, () => Array(word2Length + 1).fill(0));
    // Build up the dpMatrix with the lengths of longest common subsequence for substrings
    for (let i = 1; i <= word1Length; i++) {</pre>
        for (let j = 1; j <= word2Length; j++) {</pre>
            if (word1[i - 1] === word2[j - 1]) {
                // Characters match, take the diagonal value and add 1
                dpMatrix[i][j] = dpMatrix[i - 1][j - 1] + 1;
            } else {
                // Characters do not match, take the max value from left or top cell
                dpMatrix[i][j] = Math.max(dpMatrix[i - 1][j], dpMatrix[i][j - 1]);
    // Length of the longest common subsequence
    const longestCommonSubsequence = dpMatrix[word1Length][word2Length];
```

if word1[i - 1] == word2[j - 1]: dp_table[i][j] = dp_table[i - 1][j - 1] # If the characters are different, consider the minimum of # (deletion from word1, insertion into word1) # Add 1 representing the one operation needed else: dp_table[i][j] = 1 + min(dp_table[i - 1][j], # Delete character from word1 dp_table[i][j - 1]) # Insert character into word1 # The answer is in the cell (length_word1, length_word2) return dp_table[-1][-1]

// Subtract the length of the longest common subsequence from the total length of both strings

return word1Length - longestCommonSubsequence + word2Length - longestCommonSubsequence;

Initialize the DP table with dimensions (length_word1+1) x (length_word2+1)

dp_table[i][0] = i # Distance of converting word1 to an empty string

dp_table[0][j] = j # Distance of converting an empty string to word2

If the characters are the same, no operation is required

dp_table = [[0] * (length_word2 + 1) for _ in range(length_word1 + 1)]

Base cases: fill out the first row and column of the DP table

Time Complexity The time complexity of the code can be analyzed based on the nested for loops:

Time and Space Complexity

• The inner loop runs n times for each iteration of the outer loop, where n is the length of word2.

• The outer loop runs m times, where m is the length of word1.

Because each cell in the dp array is computed once, the total number of computations is proportional to the product of m and n.

space complexity of the code is also 0 (m * n).

Therefore, the time complexity of the code is 0(m * n).

where the operations can be insertions, deletions, or substitutions of characters.

Space Complexity The space complexity is determined by the size of the dp array used in the dynamic programming approach: • The dp array is a 2D array with dimensions $(m + 1) \times (n + 1)$.

Therefore, the amount of space used is proportional to the product of (m + 1) and (n + 1). Though the +1 is constant and does

not affect the asymptotic complexity, it accounts for the extra row and column needed for the base cases. Consequently, the

The given Python code implements the solution to find the minimum number of operations needed to convert word1 to word2,