# Delete Operation for Two Strings

Given two words *word1* and *word2*, find the minimum number of steps required to make *word1* and *word2* the same, where in each step you can delete one character in either string.

## Example 1:

```
Input: "sea", "eat"
Output: 2
Explanation: You need one step to make "sea" to "ea" and another step to make "eat" t
o "ea".
```

## Note:

1. The length of given words won't exceed 500.
2. Characters in given words can only be lower-case letters.

## Solution 1

To make them identical, just find the longest common subsequence. The rest of the characters have to be deleted from the both the strings, which does not belong to longest common subsequence.

```
public int minDistance(String word1, String word2) {
    int dp[][] = new int[word1.length()+1][word2.length()+1];
    for(int i = 0; i <= word1.length(); i++) {
        for(int j = 0; j <= word2.length(); j++) {
            if(i == 0 || j == 0) dp[i][j] = 0;
            else dp[i][j] = (word1.charAt(i-1) == word2.charAt(j-1)) ? dp[i-1][j-1]
+ 1
                    : Math.max(dp[i-1][j], dp[i][j-1]);
        }
    }
    int val =  dp[word1.length()][word2.length()];
    return word1.length() - val + word2.length() - val;
}
```

written by jaqenhgar original link here

## Solution 2

```
public class Solution {
    public int minDistance(String word1, String word2) {
        int len1 = word1.length(), len2 = word2.length();
        if (len1 == 0) return len2;
        if (len2 == 0) return len1;

        // dp[i][j] stands for distance of first i chars of word1 and first j chars
of word2
        int[][] dp = new int[len1 + 1][len2 + 1];
        for (int i = 0; i <= len1; i++)
            dp[i][0] = i;
        for (int j = 0; j <= len2; j++)
            dp[0][j] = j;

        for (int i = 1; i <= len1; i++) {
            for (int j = 1; j <= len2; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1))
                    dp[i][j] = dp[i - 1][j - 1];
                else
                    dp[i][j] = Math.min(Math.min(dp[i - 1][j - 1] + 2, dp[i - 1][j]
+ 1), dp[i][j - 1] + 1);
            }
        }

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

written by shawngao original link here

## Solution 3

Since the only operation allowed is deletion, this problem actually becomes finding the longest common subsequence.

```
public int minDistance(String word1, String word2) {
  int longest = findLongestCommonSubSequence(word1, word2);
  return word1.length() - longest + word2.length() - longest;
}

private int findLongestCommonSubSequence(String word1, String word2) {
  int[][] matrix = new int[word1.length() + 1][word2.length() + 1];
  int re = 0;
  for (int i = 1; i <= word1.length(); i++) {
    for (int j = 1; j <= word2.length(); j++) {
      if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
        matrix[i][j] = matrix[i - 1][j - 1] + 1;
      } else {
 matrix[i][j] = Math.max(matrix[i - 1][j], matrix[i][j - 1]);
      }
      re = Math.max(matrix[i][j], re);
      }
    }
  return re;
}
```

The way to calculate longest common subsequence (LCS) is the following. Assume we have two strings "seat", "ocean" and '#' denotes to empty position.

1. We first create a matrix of (n + 1) * (m+1) where n is the length of word1 and m is the length of word2. The reason why we need to plus 1 is so we can handle some edge case like when the first character is the same.
   # | # | s | e | a | t
   # | 0 | 0 | 0 | 0 | 0
   o | 0 | 0 | 0 | 0 | 0
   c | 0 | 0 | 0 | 0 | 0
   e | 0 | 0 | 0 | 0 | 0
   a | 0 | 0 | 0 | 0 | 0
   n | 0 | 0 | 0 | 0 | 0
2. We start comparing word1 and word2. In this case, we first calculate the first row. Since o is not equal to any character in "seat", the length of longest common subsequence will be zeron. Same case for character c.
   # | # | s | e | a | t
   # | 0 | 0 | 0 | 0 | 0
   **o | 0 | 0 | 0 | 0 | 0**
   **c | 0 | 0 | 0 | 0 | 0**
   e | 0 | 0 | 0 | 0 | 0
   a | 0 | 0 | 0 | 0 | 0
   n | 0 | 0 | 0 | 0 | 0
3. We now calculate row 'e'. e doesn't equal to s, so the length of LCS is 0.
   # | # | s | e | a | t

# | 0 | 0 | 0 | 0 | 0
o | 0 | 0 | 0 | 0 | 0
c | 0 | 0 | 0 | 0 | 0
e | 0 | **0** | 0 | 0 | 0
a | 0 | 0 | 0 | 0 | 0
n | 0 | 0 | 0 | 0 | 0
=> Now, we compare 'e' with 'se', since the character is the same, we now the length of LCS between "se" and "oce" is the length of LCS between "s" and "oc" plus one. That's why we have this code:

```
if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
 matrix[i][j] = matrix[i - 1][j - 1] + 1;}
```

Now, the matrix becomes the following:
# | # | s | e | a | t
# | 0 | 0 | 0 | 0 | 0
o | 0 | 0 | 0 | 0 | 0
c | 0 | **0** | 0 | 0 | 0
e | 0 | 0 | **1** | 0 | 0
a | 0 | 0 | 0 | 0 | 0
n | 0 | 0 | 0 | 0 | 0
Next, we compare c with a, since 'e' doesn't equal to 'a', we then know that, the length of LCS between 'oce' and 'sea' is the larger one of LCS('coe', 'se') and LCS('oc', 'sea').That's why we have the else statement written as below:

```
else {
  matrix[i][j] = Math.max(matrix[i - 1][j], matrix[i][j - 1]);
}
```

There are plenty of videos explaning LCS online, I didn't include the part about how to actually find the subsequence here since we only care about the length of the LCS. Hope this will help.

written by wmcalyj original link here