https://leetcode.com/problems/edit-distance/description/

Given two strings word1 and word2, return *the minimum number of operations required to convert word1 to word2*.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

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.

Attempt 1: 2023-07-12

(1) 基于L115文献的逆向递归进化到DP的思路

word1, word2 scan from left to right

在L115中文文献中，从递归的角度讲，"顶"就是递归最开始在主体方法中被呼叫的状态，本题中就是i == 0 和 j == 0 时，"底"在本题中就是当 递归到达i == s1.length() 和 n == s2.length() 时，也就是递归实际方法中的base condition，递归就是先从"顶"即i == 0 和 j == 0逐层到达"底"即i == s1.length() 和 n ==

s2.length()，然后在到达"底"后再通过返回语句逐层从"底"返回到"顶"，而DP能够省略掉递归中"从顶到底"的过程，而"直接由底向顶"，这也意味着从二维数组DP状态表的角度讲，从右下角逆推到左上角的过程，也就是i == s1.length()(底) --> i == 0(顶)，n == s2.length()(底) --> n == 0(顶)的过程

第一步：实现一个基本递归(逆向版本)：
在递归的过程就是由顶到底再回到顶

递归中由顶到底的过程：
我们的递归始于i == 0和j == 0时，i == 0(顶) --> i == s1.length()(底)，j == 0(顶) --> j == s2.length()(底)，然后在到底的时候触碰到base condition开启return返回过程

递归中再由底回到顶的过程：
在从顶到底并触碰到base condition开启return之后，逐层返回，i == s1.length()(底) --> i == 0(顶)，j == s2.length()(底) --> j == 0(顶)，此时最终状态实际上在顶，也就是i == 0和j == 0时取得，和二维DP中最终状态在左上角[0, 0]处获得形成一致

```java
class Solution {
    public int minDistance(String word1, String word2) {
        // 从顶i = 0和j = 0开始递归
        return helper(word1, 0, word2, 0);
    }
    private int helper(String s1, int i, String s2, int j) {
        // 在底i == s1.length()和j == s2.length()触底开启逐层返回到顶过程
        // Base condition 1:
        // When s1 scan index as i reach the end, the deviation
        // between s2 scan index as j and the length of s2 is the
        // minimum number of operations required to match s1 and s2
        if(i == s1.length()) {
            return s2.length() - j;
        }
        // Base condition 2:
        // When s2 scan index as j reach the end, the deviation
        // between s1 scan index as i and the length of s1 is the
        // minimum number of operations required to match s1 and s2
        if(j == s2.length()) {
            return s1.length() - i;
        }
        // Divide
```

```
        int result = 0;
        // If current position pair match
        if(s1.charAt(i) == s2.charAt(j)) {
            // no +1 as current position match, not require change and move on to next
            result = helper(s1, i + 1, s2, j + 1);
        // If current position pair not match
        } else {
            // Insert
            // Try to modify s1 to match s2, if insert new character
            // at the head of s1
            // e.g s1 = "bcd", i = 0, s2 = "abc", j = 0, insert 'a' at the
            // head of s1 then s1 = "abcd", but we don't really add 'a', the
            // dummy 'a' is used to flatten the first different pair of
            // character's deviation as 'b' in s1 and 'a' in s2, mock it as
            // 'a' in s1 and 'a' in s2.
            // Now we don't have to change index as i in s1, keep i = 0,
            // still point to the original first character as 'b' in original
            // s1 = "bcd", which also the second character in new s1 = "abcd",
            // but for index as j in s2, we have to shift to next position
            // because its current position as j = 0 character as 'a' already
            // "perished" after utilization as counterpart with dummy 'a' on s1.
            // In current example, i = 0 in s1 keep pointing to 'b' in updated
            // s1 = "abcd", j = 0 update to j = 1 in s2 will skip same character
            // as 'a' it original point to in s2 and point to next character as
            // 'b' in s2 = "abc"
            // s1 = "bcd"              s1 = "abcd"
            //       ^  --> i = 0  ==>       ^ --> i = 0
            // s2 = "abc"              s2 = "abc"
            //       ^  --> j = 0            ^ --> j + 1 = 1
            int insert_step = helper(s1, i, s2, j + 1);
            // Delete
            // Removes the first character, shifting s1 character to left.
            // Since we do not actually delete the character, incrementing i
            // simulates skipping this character
            int delete_step = helper(s1, i + 1, s2, j);
            // Replace
            // We replace the cur char with the char we need from s2, then
            // increment i and j to look at the next char
            int replace_step = helper(s1, i + 1, s2, j + 1);
            // +1 => each operation take one more step
```

```
            result = Math.min(Math.min(insert_step, delete_step), replace_step) + 1;
        }
        return result;
    }
}
```

第二步：递归配合Memoization(逆向版本)：

```
class Solution {
    public int minDistance(String word1, String word2) {
        Integer[][] memo = new Integer[word1.length() + 1][word2.length() + 1];
        // 从顶i = 0和j = 0开始递归
        return helper(word1, 0, word2, 0, memo);
    }

    private int helper(String s1, int i, String s2, int j, Integer[][] memo) {
        if(memo[i][j] != null) {
            return memo[i][j];
        }
        // 在底i == s1.length()和j == s2.length()触底开启逐层返回到顶过程
        // Base condition 1:
        // When s1 scan index as i reach the end, the deviation
        // between s2 scan index as j and the length of s2 is the
        // minimum number of operations required to match s1 and s2
        if(i == s1.length()) {
            return s2.length() - j;
        }
        // Base condition 2:
        // When s2 scan index as j reach the end, the deviation
        // between s1 scan index as i and the length of s1 is the
        // minimum number of operations required to match s1 and s2
        if(j == s2.length()) {
            return s1.length() - i;
        }
        // Divide
        int result = 0;
        // If current position pair match
```

```
        if(s1.charAt(i) == s2.charAt(j)) {
            // no +1 as current position match, not require change and move on to next
            result = helper(s1, i + 1, s2, j + 1, memo);
        // If current position pair not match
        } else {
            // Insert
            // Try to modify s1 to match s2, if insert new character
            // at the head of s1
            // e.g s1 = "bcd", i = 0, s2 = "abc", j = 0, insert 'a' at the
            // head of s1 then s1 = "abcd", but we don't really add 'a', the
            // dummy 'a' is used to flatten the first different pair of
            // character's deviation as 'b' in s1 and 'a' in s2, mock it as
            // 'a' in s1 and 'a' in s2.
            // Now we don't have to change index as i in s1, keep i = 0,
            // still point to the original first character as 'b' in original
            // s1 = "bcd", which also the second character in new s1 = "abcd",
            // but for index as j in s2, we have to shift to next position
            // because its current position as j = 0 character as 'a' already
            // "perished" after utilization as counterpart with dummy 'a' on s1.
            // In current example, i = 0 in s1 keep pointing to 'b' in updated
            // s1 = "abcd", j = 0 update to j = 1 in s2 will skip same character
            // as 'a' it original point to in s2 and point to next character as
            // 'b' in s2 = "abc"
            // s1 = "bcd"              s1 = "abcd"
            //       ^  --> i = 0  ==>       ^ --> i = 0
            // s2 = "abc"              s2 = "abc"
            //       ^  --> j = 0            ^ --> j + 1 = 1
            int insert_step = helper(s1, i, s2, j + 1, memo);
            // Delete
            // Removes the first character, shifting s1 character to left.
            // Since we do not actually delete the character, incrementing i
            // simulates skipping this character
            int delete_step = helper(s1, i + 1, s2, j, memo);
            // Replace
            // We replace the cur char with the char we need from s2, then
            // increment i and j to look at the next char
            int replace_step = helper(s1, i + 1, s2, j + 1, memo);
            // +1 => each operation take one more step
            result = Math.min(Math.min(insert_step, delete_step), replace_step) + 1;
        }
```

```
            return memo[i][j] = result;
        }
    }
```

第三步：基于递归的2D DP(逆向版本)：

DP能够省略掉递归中"从顶到底"的过程，而"直接由底向顶"，这也意味着从二维数组DP状态表的角度讲，从右下角逆推到左上角的过程，也就是i == s1.length()(底) --> i == 0(顶)，j == s2.length()(底) --> j == 0(顶)的过程

这里我们用一个二维数组 dp[i][j] 对应于从 s[i，s1.length()) 所代表的的字符串需要多少步变成 s2[j，s2.length())。

当 i == s1.length()，意味着s1是空串，此时dp[s1.length()][j]，取值随 j 变化，即 s2.length() - j

当 j == s2.length()，意味着s2是空串，此时dp[i][s2.length()]，取值随 i 变化，即 s1.length() - i

然后状态转移的话和解法一分析的一样。如果求dp[i][j]。

- s1[i] == s2[j]，当前两个字符相等，需要多少步s1能变成s2取决于同时跳过当前字符时两个字符串的关系，之前需要多少步现在仍需要多少步

dp[i][j] = dp[i+1][j+1]

- s1[i] != s2[j]，有三种情况，1.去掉一个字符，2.加入一个字符，3.替换一个字符，取三种方案中所需步骤最少的方案

dp[i][j] = Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1])

insert -> 对应dp[i][j + 1]

delete -> 对应dp[i + 1][j]

replace -> 对应dp[i + 1][j + 1]

代码就可以写了。

```
class Solution {
    /**
        在底i == s1.length()和j == s2.length()触底开启逐层返回到顶i = 0和j = 0过程
        观察基础状态在i和j到达底(即原字符串长度时)获得，尤其当i和j同时到达底时，即
        dp[s1.length()][s2.length()] = 0
        dp[s1.length()][j] = s2.length() - j = {3,2,1,0}
        dp[i][s2.length()] = s1.length() - i = {5,4,3,2,1,0}
        if(i == s1.length()) {
            return s2.length() - j;
        }
```

```
        if(j == s2.length()) {
            return s1.length() - i;
        }
        然后是递推关系也符合递归（s1 and s2 scan from left to right）中的逻辑，即
        if(word1.charAt(i) != word2.charAt(j)) {
            dp[i][j] = Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1]) + 1;
        } else {
            dp[i][j] = dp[i + 1][j + 1];
        }

        e.g s1 = "horse", s2 = "ros"
            0 1 2 3
         s2 r o s '' -> j
       s1
      0 h   3 3 4 5
      1 o   3 2 3 4
      2 r   2 2 2 3
      3 s   3 2 1 2
      4 e   3 2 1 1
      5 ''  3 2 1 0
     -> i
    */
    public int minDistance(String word1, String word2) {
        int w1_len = word1.length();
        int w2_len = word2.length();
        int[][] dp = new int[w1_len + 1][w2_len + 1];
        // No need set up below since it covered by two base conditions
        // dp[w1_len][w2_len] = 0
        // 对应递归底时的基础状态1：i到底（即word1的长度）
        // dp[word1.length()][j] = word2.length() - j
        for(int j = 0; j <= w2_len; j++) {
            dp[w1_len][j] = w2_len - j;
        }
        // 对应递归底时的基础状态2：j到底（即word2的长度）
        // dp[i][word2.length()] = word1.length() - i
        for(int i = 0; i <= w1_len; i++) {
            dp[i][w2_len] = w1_len - i;
        }
        // 倒着进行，word1每次增加一个字母（row维度）
```

```
        for(int i = w1_len - 1; i >= 0; i--) {
            // 倒着进行，word2每次增加一个字母（column维度）
            for(int j = w2_len - 1; j >= 0; j--) {
                // 当两个字母不相等，当前状态取决于上一层insert，delete，replace三种情况的
                // 最小结果，然后加一步
                // 对应递归中的关系：result = Math.min(Math.min(insert_step,
delete_step), replace_step) + 1;
                // 同原始递归解法中一致，依然以改变word1（row维度i）去匹配word2（column维度
j）
                // insert -> 对应dp[i][j + 1]
                // delete -> 对应dp[i + 1][j]
                // replace -> 对应dp[i + 1][j + 1]
                /**
                    e.g
                    s1 = "horse", s2 = "ros"
                          0 1 2 3
                       s2 r o s '' -> j
                    s1
                    0 h   ?     5
                    1 o         4
                    2 r         3
                    3 s         2
                    4 e       1 1
                    5 ''  3 2 1 0
                    -> i
                    dp[4][2] = Math.min(Math.min(dp[5][2], dp[4][3]), dp[5][3])
                    即需要多少步把s1 = "e"变成s2 = "s"呢？
                    ----------------------------------------------------------------
--
                    insert -> 和递归中一样采用头插法，在s1前面虚拟插入字符's'，此时s1变
成"se"，
                    s2依然是"s"，遵循递归中的思路，扫描s1的坐标i不用改变，保持原位，即i = 4
不变，
                    但s2的坐标j需要向下一个位置移动，因为此时s1中虚拟插入的字符's'已经和s2当
前
                    j = 2所指示的's'匹配并抵消了，要判断s1是否和s2一致需要看s1中i = 4所指代
的字
                    符和s2中j + 1 = 2 + 1 = 3所指代的字符是否一致，不过我们看到s2并没有下一
个字符，
                    此时直接到达s2到头的边界条件（对应递归中的底之一，j == s2.length()），那
么还
                    需要多少步实现s1和s2一致呢？dp[4][3] = 1的结果和递归中以下返回值保持了一
致性，
```

```
                if(j == s2.length()) return s1.length() - i; --> 5 - 4 = 1
                可以理解为要让dp[4][2]在s1中insert一个字符（头插法）的情况下去匹配没有改变的
                s2还需要dp[4][3]所代表的步骤，即还需要1步，s1 = "se"还需要delete 'e'这一步
                就可以变成"s"，此时匹配了s2
                s1 = "e"                s1 = "se"
                     ^  --> i = 4  ==>        ^ --> i = 4
                s2 = "s"                s2 = "s"
                     ^  --> j = 2             ^ --> j + 1 = 3

                反映在2D DP数组中dp[4][2]可以是dp[4][3] + 1
                ------------------------------------------------------------------
                delete -> 删除一个字符，在s1中删除一个字符，此时s1变成""空字符串，此时直接
                到达s1到头的边界条件（对应递归中的底之一，i == s1.length()），s2依然是
                "s"，遵循递归中的思路，扫描s2的j不用改变，扫描s1的i需要向下一个位置移动，因为
                当前i位置所代表的字符已经被删除，需要查看s1中的下一个字符与没有改变的s2的当前
                字符的匹配关系，而由于前述s1在去除仅有的一个字符后到达边界，那么还需要多少步
                实现s1和s2一致呢？dp[5][2] = 1的结果和递归中以下返回值保持了一致性，
                if(i == s1.length()) return s2.length() - j; --> 3 - 2 = 1
                可以理解为要让dp[4][2]在s1中delete一个字符的情况下去匹配没有改变的s2还需要
                dp[5][2]所代表的步骤，即还需要1步，s1 = ""还需要insert 's'这一步就可以变
                成"s"，此时匹配了s2
                s1 = "e"                s1 = ""
                     ^ --> i = 4  ==>         ^ --> i + 1 = 5
                s2 = "s"                s2 = "s"
                       --> j = 2              ^  --> j = 2
                反映在2D DP数组中dp[4][2]可以是dp[5][2] + 1
                ------------------------------------------------------------------
                replace -> 替换一个字符，在s1中替换一个字符，此时s1从"e"变成"s"，直接就匹配
                了s2，根据递归中的关系，在替换一个字符的时候s1和s2当前坐标所指示的字符完成了
                匹配，两个坐标都需要向下一个位置移动，以继续匹配后续的字符，不过由于s1和s2在
```

```
                    i = 4和j = 2的情况下同时后移一个位置同时到头，同时达到边界条件（对应递归中底
                    的两种情况：i == s1.length()和j == s2.length()），照说此时s1和s2已经完成了匹配，
                    那么还需要多少步实现s1和s2一致呢？理论上不再需要步骤了，应该为0步，我们来看看
                    dp[5][3]的结果是否符合猜想，dp[5][3]在初始化中就同时被包含在以下2个初始化中：
                    for(int j = 0; j <= w2_len; j++) {dp[w1_len][j] = w2_len - j;}
                    for(int i = 0; i <= w1_len; i++) {dp[i][w2_len] = w1_len - i;}
                    无论从哪个角度都是dp[5][3] = 0，本质含义就是当s1和s2都是空串的时候不需要步骤
                    来完成匹配了，这也符合递归中"底"（同时满足两个base condition）的表述
                    s1 = "e"                  s1 = ""
                          ^ --> i = 4  ==>          ^ --> i + 1 = 5
                    s2 = "s"                  s2 = ""
                            --> j = 2               ^ --> j + 1 = 3

                    反映在2D DP数组中dp[4][2]可以是dp[5][3] + 1
                    ------------------------------------------------------------------------
                 */
                if(word1.charAt(i) != word2.charAt(j)) {
                    dp[i][j] = Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1]) + 1;
                // 当两个字母相等，两个字符串都不需要做任何变动，当前状态和上一层状态一致，
                // 直接跳过当前层，不需要加一步
                // 对应递归中的关系：result = helper(s1, i + 1, s2, j + 1)
                } else {
                    dp[i][j] = dp[i + 1][j + 1];
                }
            }
        }
        return dp[0][0];
    }
}
```

第四步：基于2D DP的空间优化1D DP(逆向版本)：

优化为2 rows (相对于L115真正展现2 rows array替代2D array的本质)

```java
class Solution {
    public int minDistance(String word1, String word2) {
        int w1_len = word1.length();
        int w2_len = word2.length();
        // 原2D DP数组中的定义：word1是row维度，word2是column维度
        //int[][] dp = new int[w1_len + 1][w2_len + 1];
        // No need set up below since it covered by two base conditions
        // dp[word1.length()][word2.length()] = 0
        // 对应递归底时的基础状态1：i到底（即word1的长度）
        // dp[word1.length()][j] = word2.length() - j
        //for(int j = 0; j <= w2_len; j++) {
        //    dp[w1_len][j] = w2_len - j;
        //}
        // 对应递归底时的基础状态2：j到底（即word2的长度）
        // dp[i][word2.length()] = word1.length() - i
        //for(int i = 0; i <= w1_len; i++) {
        //    dp[i][w2_len] = w1_len - i;
        //}
        // -> 现在只保留了column维度，因为本质上是row的维度上"上一行只依赖于下一行"，在原2D数组中上一行是dp[i]，下一行是dp[i + 1]，现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i + 1][j]平行替换为dpPrev[j]，dp[i + 1][j + 1]平行替换为dpPrev[j + 1]
        int[] dp = new int[w2_len + 1];
        int[] dpPrev = new int[w2_len + 1];
        // -> 去掉row维度后初始化状态进化为只需要设定剩下column维度
        for(int i = 0; i <= w2_len; i++) {
            // 注意：这题和L115中随便初始化dp或者dpPrev效果一致不同，这题的初始化让人更透彻的
            // 理解如何用2 rows array来模拟2D array的效果
            // 如下使用s1 = "horse", s2 = "ros"演示逆向模式（以dp[0][0]为结束）下的情况：
            // 因为我们是用dpPrev和dp两个1D array迭代来替代原先的2D array，在实际替代的初始化中，
            // 第一个被替换的就是原先2D array中的最后一行，但是不像L115中随便用dp或者dpPrev
            // 替代那样效果相当，不出问题。实际上dpPrev必须是最优先被赋值的，因为得用dpPrev推算出
            // dp，但在L115中巧合的是dpPrev即使和dp初始化为同样的值也不影响，因为在L115逆向模式
            // 中倒数第一行和倒数第二行的最后一个数字都是一样的（都为1），而且L115中计算dp时用到
            // 的公式为dp[i] = dpPrev[i + 1] + dpPrev[i]，并不需要dp本行的元素参与，但是本题中
            // 倒数第一行和倒数第二行的最后一个数字并不是一样的（一个为0，一个为1），而且本题计算
```

```
            // dp时用到的公式为dp[j] = Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1]) + 1,
            // 我们发现dp[j]用到了本行元素dp[j + 1]。举个例子，假设倒数第一行在初始化中用dpPrev
            // 代表，倒数第二行在初始化中用dp代表，假设倒数第二行的倒数第二个元素是dp[j]，那么在
            // 计算该元素的时候不仅会用到倒数第一行的元素dpPrev[j + 1]和dpPrev[j]，也会用到倒数
            // 第二行的倒数第一个元素dp[j + 1]，所以实际上dpPrev和dp必须是严格分开定义的，不可混淆。
            // 总体来说，在本题的逆向模式中dpPrev代表原2D DP array的倒数第一行，dp则是倒数第二行，
            // 并且使用dpPrev来推算，但在推算dp，也即倒数第二行所有其他元素前，其最后一个元素也得
            // 附上数值，原因如前述
            /**
                    e.g s1 = "horse", s2 = "ros"
                       0 1 2 3
                    s2 r o s '' -> j
                s1
                0 h   3 3 4 5
                1 o   3 2 3 4
                2 r   2 2 2 3
                3 s   3 2 1 2
                4 e   3 2 1[1] -> the last element '1' in second last row equal initial dp array
                5 ''  3 2 1 0  -> the last row equal initial dpPrev array (not dp array)
                -> i
             */
            //dp[i] = w2_len - i; --> wrong way, since last row initialize must only allocate to dpPrev
            dpPrev[i] = w2_len - i;
        }
        // 倒着进行，word1每次增加一个字母（row维度）
        // -> 外层循环依旧为row维度，而且dpPrev/dp在row维度的反复替换也在外层循环发生，为了维持row维度的替换，外层循环必须使用row维度
        for(int i = w1_len - 1; i >= 0; i--) {
            // -> 根据上述细节叙述，因为dp推算公式中含有dp同行元素，推算dp其他元素前必须初始化dp的最后一个元素(因为逆向模式中是从右往左推导)
            dp[w2_len] = w1_len - i;
            // 倒着进行，word2每次增加一个字母（column维度）
            for(int j = w2_len - 1; j >= 0; j--) {
```

```
                if(word1.charAt(i) != word2.charAt(j)) {
                    //dp[i][j] = Math.min(Math.min(dp[i + 1][j], dp[i][j + 1]), dp[i + 1][j + 1]) + 1;
                    // -> 现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i + 1][j]平行替换为dpPrev[j]，dp[i][j + 1]平行替换为dp[j + 1]，dp[i + 1][j + 1]平行替换为dpPrev[j + 1]
                    dp[j] = Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1]) + 1;
                // 当两个字母相等，两个字符串都不需要做任何变动，当前状态和上一层状态一致，
                // 直接跳过当前层，不需要加一步
                // 对应递归中的关系：result = helper(s1, i + 1, s2, j + 1)
                } else {
                    //dp[i][j] = dp[i + 1][j + 1];
                    dp[j] = dpPrev[j + 1];
                }
            }
            dpPrev = dp.clone();
        }
        return dpPrev[0];
    }
}
```

进一步优化为1 row (不是真正的1 row方案，内层循环不需要反转，因为只是用2个变量替代了2 rows中的1 row)

```
class Solution {
    public int minDistance(String word1, String word2) {
        int w1_len = word1.length();
        int w2_len = word2.length();
        //int[] dp = new int[w2_len + 1]; -> remove dp array
        int[] dpPrev = new int[w2_len + 1];
        for(int i = 0; i <= w2_len; i++) {
            dpPrev[i] = w2_len - i;
        }
        int prev = 0;
        for(int i = w1_len - 1; i >= 0; i--) {
            // Special handling to store current row(dpPrev)'s last
            // element in each round as a single variable 'prev',
            // since it will be used in formula below to replace
```

```
            // 'dpPrev[j + 1]' further
            // dp[j] = Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1]) + 1;
            prev = dpPrev[w2_len];
            // Then update current row(dpPrev)'s last element to next round
            // row(dp)'s last element value(w1_len - i) which suppose
            // belongs to dp, but we don't create dp to represent
            // next round row now
            dpPrev[w2_len] = w1_len - i;
            // No need to reverse the scanning order like L115, but why ?
            // Because its NOT the REAL 1D array solution, its just use two
            // variables as 'prev' to represent the necessary last element in
            // 'dpPrev' array and 'temp' to update 'prev' each round, not
            // fully dismiss 'dp' array, just use two variables to MOCK another
            // 'dp' array
            // for(int j = 0; j <= w2_len - 1; j++) { --> Wrong Way !!!
            for(int j = w2_len - 1; j >= 0; j--) {
                int temp = dpPrev[j];
                if(word1.charAt(i) != word2.charAt(j)) {
                    //dp[j] = Math.min(Math.min(dpPrev[j], dp[j + 1]), dpPrev[j + 1])
+ 1; -> 'prev' replace dpPrev[j + 1]
                    dpPrev[j] = Math.min(Math.min(dpPrev[j], dpPrev[j + 1]), prev) + 1;
                } else {
                    //dp[j] = dpPrev[j + 1]; -> 'prev' replace dpPrev[j + 1]
                    dpPrev[j] = prev;
                }
                prev = temp;
            }
            //dpPrev = dp.clone();
        }
        return dpPrev[0];
    }
}
```

Refer to

https://leetcode.com/problems/edit-distance/solutions/25846/c-o-n-space-dp/

```
class Solution {
public:
```

```
    int minDistance(string word1, string word2) {
        int m = word1.size(), n = word2.size(), pre;
        vector<int> cur(n + 1, 0);
        for (int j = 1; j <= n; j++) {
            cur[j] = j;
        }
        for (int i = 1; i <= m; i++) {
            pre = cur[0];
            cur[0] = i;
            for (int j = 1; j <= n; j++) {
                int temp = cur[j];
                if (word1[i - 1] == word2[j - 1]) {
                    cur[j] = pre;
                } else {
                    cur[j] = min(pre, min(cur[j - 1], cur[j])) + 1;
                }
                pre = temp;
            }
        }
        return cur[n];
    }
};
```

(2) 基于L115文献的正向递归进化到DP的思路

word1, word2 scan from right to left

第一步：实现一个基本递归(正向版本)：

在LeetCode解答中，从递归的角度讲，"顶"就是递归最开始在主体方法中被呼叫的状态，本题中就是i == s1.length() 和 j == s2.length() 时，"底"在本题中就是当 递归到达i == 0 和 j == 0 时，也就是递归实际方法中的base condition，递归就是先从"顶"即i == s1.length() 和 j == s2.length()逐层到达"底"即i == 0 和 j == 0，然后在到达"底"后再通过返回语句逐层从"底"返回到"顶"，而DP能够省略掉递归中"从顶到底"的过程，而"直接由底向顶"，这也意味着从二维数组DP状态表的角度讲，从左上角正推到右下角的过程，也就是i == 0(底) --> i == s1.length()(顶)，j == 0(底) --> j == s2.length()(顶)的过程

递归中由顶到底的过程：

我们的递归始于i == s1.length()和j == s2.length()时，i == s1.length()(顶) --> i == 0(底)，j == s2.length()(顶) --> j == 0(底)，然后在到底的时候触碰到base condition开启return返回过程

递归中再由底回到顶的过程：

在从顶到底并触碰到base condition开启return之后，逐层返回，i == 0(底) --> i == s1.length()(顶)，j == 0(底) --> j == s2.length()(顶)，此时最终状态实际上在顶，也就是i == s1.length()和j == s2.length()时取得，和二维DP中最终状态在右下角[s1.length(), s2.length()]处获得形成一致

```java
class Solution {
    public int minDistance(String word1, String word2) {
        return helper(word1, word1.length(), word2, word2.length());
    }

    public int helper(String s1, int i, String s2, int j) {
        // Imaging s1 is empty string "" now, how many steps for s2 to become
        // empty string "" also ? At least delete all its characters requires
        // j steps of 'delete' operation, hence return j
        if(i == 0) {
            return j;
        }
        // Imaging s2 is empty string "" now, how many steps for s1 to become
        // empty string "" also ? At least delete all its characters requires
        // i steps of 'delete' operation, hence return i
        if(j == 0) {
            return i;
        }
        int result = 0;
        // If current pair of character in both string equals, no need any step,
        // directly move on
        if(s1.charAt(i - 1) == s2.charAt(j - 1)) {
            result = helper(s1, i - 1, s2, j - 1);
        // If not equal characters, move on with 3 choices but add 1 more step required
        } else {
            int insert_step = helper(s1, i, s2, j - 1);
            int delete_step = helper(s1, i - 1, s2, j);
            int replace_step = helper(s1, i - 1, s2, j - 1);
            result = Math.min(Math.min(insert_step, delete_step), replace_step) + 1;
        }
```

```
        return result;
    }
}
```

第二步：递归配合Memoization(正向版本)：

```
class Solution {
    public int minDistance(String word1, String word2) {
        Integer[][] memo = new Integer[word1.length() + 1][word2.length() + 1];
        return helper(word1, word1.length(), word2, word2.length(), memo);
    }

    public int helper(String s1, int i, String s2, int j, Integer[][] memo) {
        if(memo[i][j] != null) {
            return memo[i][j];
        }
        // Imaging s1 is empty string "" now, how many steps for s2 to become
        // empty string "" also ? At least delete all its characters requires
        // j steps of 'delete' operation, hence return j
        if(i == 0) {
            return j;
        }
        // Imaging s2 is empty string "" now, how many steps for s1 to become
        // empty string "" also ? At least delete all its characters requires
        // i steps of 'delete' operation, hence return i
        if(j == 0) {
            return i;
        }
        int result = 0;
        // If current pair of character in both string equals, no need any step,
        // directly move on
        if(s1.charAt(i - 1) == s2.charAt(j - 1)) {
            result = helper(s1, i - 1, s2, j - 1, memo);
        // If not equal characters, move on with 3 choices but add 1 more step required
        } else {
            int insert_step = helper(s1, i, s2, j - 1, memo);
            int delete_step = helper(s1, i - 1, s2, j, memo);
```

```
            int replace_step = helper(s1, i - 1, s2, j - 1, memo);
            result = Math.min(Math.min(insert_step, delete_step), replace_step) + 1;
        }
        return memo[i][j] = result;
    }
}
```

第三步：基于递归的2D DP(正向版本)：

```
class Solution {
    /**
      s1.charAt(i - 1) == s2.charAt(j - 1)
      -> dp[i][j] = dp[i - 1][j - 1];

      s1.charAt(i - 1) != s2.charAt(j - 1)
      -> dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);

      e.g s1 = "horse", s2 = "ros"

            0 1 2 3
         s2 '' r o s -> j
       s1
      0 ''   0 1 2 3
      1 h    1 1 2 3
      2 o    2 2 1 2
      3 r    3 2 2 2
      4 s    4 3 3 2
      5 e    5 4 4 3
     -> i
     */
    public int minDistance(String word1, String word2) {
        int w1_len = word1.length();
        int w2_len = word2.length();
        int[][] dp = new int[w1_len + 1][w2_len + 1];
        for(int i = 0; i <= w1_len; i++) {
            dp[i][0] = i;
        }
```

```
        for(int j = 0; j <= w2_len; j++) {
            dp[0][j] = j;
        }
        for(int i = 1; i <= w1_len; i++) {
            for(int j = 1; j <= w2_len; 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], dp[i][j - 1]), dp[i - 1][j - 1]) + 1;
                }
            }
        }
        return dp[w1_len][w2_len];
    }
}
```

第四步：基于2D DP的空间优化1D DP(正向版本，基于第三步)：

优化为2 rows (相对于L115真正展现2 rows array替代2D array的本质)

```
class Solution {
    /**
      s1.charAt(i - 1) == s2.charAt(j - 1)
      -> dp[i][j] = dp[i - 1][j - 1];

      s1.charAt(i - 1) != s2.charAt(j - 1)
      -> dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);

      e.g s1 = "horse", s2 = "ros"

            0 1 2 3
         s2 '' r o s -> j
       s1
      0 ''   0 1 2 3 -> the first row equal initial dpPrev array (not dp array)
      1 h   [1]1 2 3 -> the first element '1' in second row equal initial dp array
      2 o    2 2 1 2
```

```
       3 r    3 2 2 2
       4 s    4 3 3 2
       5 e    5 4 4 3
      -> i
      */
    public int minDistance(String word1, String word2) {
        int w1_len = word1.length();
        int w2_len = word2.length();
        //int[][] dp = new int[w1_len + 1][w2_len + 1];
        //for(int i = 0; i <= w1_len; i++) {
        //    dp[i][0] = i;
        //}
        //for(int j = 0; j <= w2_len; j++) {
        //    dp[0][j] = j;
        //}
        int[] dp = new int[w2_len + 1];
        int[] dpPrev = new int[w2_len + 1];
        for(int i = 0; i <= w2_len; i++) {
            dpPrev[i] = i;
        }
        // 正着进行，word1每次增加一个字母（row维度）
        // -> 外层循环依旧为row维度，而且dpPrev/dp在row维度的反复替换也在外层循环发生，为了维持row维度的替换，外层循环必须使用row维度
        for(int i = 1; i <= w1_len; i++) {
            // -> 根据上述细节叙述，因为dp推算公式中含有dp同行元素，推算dp其他元素前必须初始化dp的第一个元素(因为正向模式中是从左往右推导)
            dp[0] = i;
            // 正着进行，word2每次增加一个字母（column维度）
            for(int j = 1; j <= w2_len; j++) {
                if(word1.charAt(i - 1) == word2.charAt(j - 1)) {
                    //dp[i][j] = dp[i - 1][j - 1];
                    // -> 现在由于去掉了row维度，dp[i][j]平行替换为dp[j]，dp[i - 1][j]平行替换为dpPrev[j]，dp[i][j - 1]平行替换为dp[j - 1]，dp[i - 1][j - 1]平行替换为dpPrev[j - 1]
                    dp[j] = dpPrev[j - 1];
                // 当两个字母相等，两个字符串都不需要做任何变动，当前状态和上一层状态一致，
                // 直接跳过当前层，不需要加一步
                // 对应递归中的关系：result = helper(s1, i - 1, s2, j - 1)
                } else {
                    //dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]) + 1;
```

```
                    dp[j] = Math.min(Math.min(dpPrev[j], dp[j - 1]), dpPrev[j - 1]) + 1;
                }
            }
            dpPrev = dp.clone();
        }
        return dpPrev[w2_len];
    }
}
```

进一步优化为1 row (不是真正的1 row方案，内层循环不需要反转，因为只是用2个变量替代了2 rows中的1 row)

```
class Solution {
    /**
      s1.charAt(i - 1) == s2.charAt(j - 1)
      -> dp[i][j] = dp[i - 1][j - 1];

      s1.charAt(i - 1) != s2.charAt(j - 1)
      -> dp[i][j] = Math.min(Math.min(dp[i - 1][j], dp[i][j - 1]), dp[i - 1][j - 1]);


      e.g s1 = "horse", s2 = "ros"

            0 1 2 3
         s2 '' r o s -> j
       s1
      0 ''   0 1 2 3 -> the first row equal initial dpPrev array (not dp array)
      1 h   [1]1 2 3 -> the first element '1' in second row equal initial dp array
      2 o    2 2 1 2
      3 r    3 2 2 2
      4 s    4 3 3 2
      5 e    5 4 4 3
     -> i
     */
    public int minDistance(String word1, String word2) {
        int w1_len = word1.length();
```

```java
        int w2_len = word2.length();
        //int[] dp = new int[w2_len + 1]; -> remove dp array
        int[] dpPrev = new int[w2_len + 1];
        for(int i = 0; i <= w2_len; i++) {
            dpPrev[i] = i;
        }
        int prev = 0;
        for(int i = 1; i <= w1_len; i++) {
            // Special handling to store current row(dpPrev)'s first
            // element in each round as a single variable 'prev',
            // since it will be used in formula below to replace
            // 'dpPrev[j - 1]' further
            // dp[j] = Math.min(Math.min(dpPrev[j], dp[j - 1]), dpPrev[j - 1]) + 1;
            prev = dpPrev[0];
            // Then update current row(dpPrev)'s first element to next round
            // row(dp)'s first element value(i) which suppose
            // belongs to dp, but we don't create dp to represent
            // next round row now
            dpPrev[0] = i;
            // No need to reverse the scanning order like L115, but why ?
            // Because its NOT the REAL 1D array solution, its just use two
            // variables as 'prev' to represent the necessary last element in
            // 'dpPrev' array and 'temp' to update 'prev' each round, not
            // fully dismiss 'dp' array, just use two variables to MOCK another
            // 'dp' array
            // for(int j = w2_len; j >= 0; j--) { --> Wrong Way !!!
            for(int j = 1; j <= w2_len; j++) {
                int temp = dpPrev[j];
                if(word1.charAt(i - 1) != word2.charAt(j - 1)) {
                    //dp[j] = Math.min(Math.min(dpPrev[j], dp[j - 1]), dpPrev[j - 1])
+ 1; -> 'prev' replace dpPrev[j - 1]
                    dpPrev[j] = Math.min(Math.min(dpPrev[j], dpPrev[j - 1]), prev) + 1;
                } else {
                    //dp[j] = dpPrev[j - 1]; -> 'prev' replace dpPrev[j - 1]
                    dpPrev[j] = prev;
                }
                prev = temp;
            }
            //dpPrev = dp.clone();
        }
```

```
        return dpPrev[w2_len];
    }
}
```

Refer to

https://leetcode.com/problems/edit-distance/solutions/25846/c-o-n-space-dp/

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        int m = word1.size(), n = word2.size(), pre;
        vector<int> cur(n + 1, 0);
        for (int j = 1; j <= n; j++) {
            cur[j] = j;
        }
        for (int i = 1; i <= m; i++) {
            pre = cur[0];
            cur[0] = i;
            for (int j = 1; j <= n; j++) {
                int temp = cur[j];
                if (word1[i - 1] == word2[j - 1]) {
                    cur[j] = pre;
                } else {
                    cur[j] = min(pre, min(cur[j - 1], cur[j])) + 1;
                }
                pre = temp;
            }
        }
        return cur[n];
    }
};
```

Refer to

https://leetcode.com/problems/edit-distance/solutions/25895/step-by-step-explanation-of-how-to-optimize-the-solution-from-simple-recursion-to-dp/comments/562196

I was having trouble with this and figuring out the recurrences for the edit operations. i and j basically keep track of the current characters that are getting compared, each operation shifts them differently. The important thing to note is that **we are simulating the edit operations by moving i and j around**, not actually changing the input strings.

EXAMPLE

c1 = sample, c2 = example

i = 0 (s), j = 0 (e)

Replace is simplest for me to understand, we replace the cur char with the char we need from word2. We then increment i and j to look at the next char.

Replace -> match(c1,c2, i+1, j+1)

c1 = *e*ample, c2 = example

i = 1 (a), j = 1 (x)

Delete removes the first character, shifting word 1 character to left. Since we do not actually delete the char, incrementing i simulates skipping this char.

Delete -> match(c1, c2, i+1, j)

c1 = sample, c2 = example

i = 1 (a) , j = 0 (e)

Insert is the opposite of delete, we insert the char we need, shifting word 1 to the right. Since we do not actually add a char, leave i alone. It's the similar as doing:

```
c1 = "e" + c1;
match(c1,c2,i+1,j+1) //Since we added "e", i+1 would point to "s"
```

Insert -> match(c1, c2, i, j+1)

c1 = *e*sample, c2 = example

i = 0 (s), j = 1 (x)

Refer to

L115.Distinct Subsequences