**最长公共子序列**

1. **公共子序列相关定义**
2. **子序列定义**

对于两序列 X = <x\_1, x\_2, …, x\_m> 与 Z = <z\_1, z\_2, …, z\_m> ，如果存在 X 的元素构成的按下标严格递增序列<x\_i1, x\_i2, …, x\_im> 使得x\_i\_j = z\_j, j = 1,2,…,k,那么称 Z 是 X 的**子序列**。Z含有的元素个数，称为**子序列的长度**。

**注意：** 子序列和字符串子串不同，简单来看，子序列**可以是**一个序列中**不连续**的元素组成的序列，而子串是字符串中**连续**的一个子字符串。

1. **公共子序列定义**

设X和Y是两个序列，如果Z既是X的子序列，也是Y的子序列，则称Z是X与Y的**公共子序列**。

而X和Y的公共子序列中长度最长的子序列，是X与Y的**最长公共子序列**。

**举例：**

对于数组X = {1, 3, 7, 1, 7, 5}和数组Y = {1, 9, 2, 5, 1}，他们的子序列可以是{1}, {5}, {1, 1}, {1, 5}，而其中最长的子序列是{1, 1}和{1, 5}，他们的长度为2.

1. **最长公共子序列问题(Longest Common Subsequence, LCS)**

给定序列：

X = <x\_1, x\_2, …, x\_m>, Y = < y\_1, y\_2, …, y\_n>

求X和Y的最长公共子序列。

1. **问题算法**
2. **蛮力算法**

可以想出最简单的蛮力算法。当m≤n时，可以列出序列X的所有子序列，再分别判断每个子序列是否为Y的子序列，如果是，则记录该公共子序列，最终从记录的公共子序列中找出最长序列。

算法分析：

生成序列X所有的子序列，需要对m个位置分别确定“是”或“否”包含在某子序列中，则一共有2^m种可能性，即2^m个子序列。检查序列X的某个子序列时，需要遍历序列Y，需要O(n)的时间，则该算法的时间复杂度为O(n\*2^m).

1. **经典动态规划**

**注意到问题可以分解成子问题**：

在子问题中，X和Y都从第1个元素开始，而X只考虑到位置i(1≤i≤m），Y只考虑到位置j(1≤j≤n），且可通过x\_i和y\_j的关系逐层递推到i=m且j=n时，也就是原问题的情况。

从**子问题递推**的关键在于x\_i和y\_j的关系，可分为以下几类：

子问题的序列分别表示为X\_i = <x\_1, x\_2, …, x\_i>, Y\_j = < y\_1, y\_2, …, y\_j>，使序列Z\_k = <z\_1, z\_2, …, z\_k>为序列X\_i与序列Y\_j的最长公共子序列，则

1）若x\_i == y\_j时，则必然有x\_i == y\_j == z\_k，且Z\_k-1为X\_i-1和Y\_j-1的最长公共子序列。（观察易得，如果不是，则Z\_k-1加上新元素z\_k得到的序列Z\_k也不会是X\_i和Y-j的最长公共子序列，从而产生矛盾）

2）若x\_i≠y\_j，可以分为下面两种情况：

a) z\_k ≠ x\_i，则从Z\_k和X\_i同时去掉最后一个元素并不会影响前提条件，则有Z\_k-1是X\_i-1与Y\_j的最长公共子序列；

b) z\_k ≠ y\_j，同理，有Z\_k-1是X\_i与Y\_j-1的最长公共子序列。

由此可以**得到问题的递推公式**：

使L(i,j)为序列X\_i与序列Y\_j的最长公共子序列的长度，则

**| L(i,j) = L(i-1,j-1) + 1 (x\_i==y\_j)**

**| L(i,j) = max(L(i,j-1), L(i-1,j)) (x\_i≠y\_j)**

在分析子问题之后，需要**确定子问题的边界条件**。容易看出，使得i=0或j=0时，X\_i与Y\_j中至少有一个序列为空，也就是说，此时最长公共子序列也为空，长度为0

**| L(0,j) = L(i,0) = 0 (1≤i≤m, 1≤j≤n)**

最后，我们**对解进行追踪（设定标记函数）**，由此，我们可以轻易地递推得出原问题的答案。

对算法分析，我们在将某个元素加入最长公共子序列时都有x\_i==y\_j，也就是说，只有此情况下的元素才会加入最终答案；对于x\_i≠y\_j的情况，我们会转化为查看(i,j-1)和(i-1,j)的情况，以此类推，故在追踪解的时候，需要给出判断方向。

【对例子进行列表分析】

我们将这种得到位置(i,j)处解的状态的函数称为标记函数，用B(i,j)表示。

【伪代码】

从伪代码可以看出，主要操作都在一个两层for循环中，不难得出，该算法的时间复杂度为O(mn).

1. **【优化】一维DP**

在编程实现中，我们需要划分内存存放各个子问题的信息，对于m\*n个子问题，我们经典动态规划算法的空间复杂度为O(mn)，并不算优秀。

我们观察问题的递推公式，可以发现，我们每次递推利用到的数据都是(i-1,j-1)、(i,j-1)、(i-1,j)中的数据，我们可以仅用一维数组保存每次数据的内容，也就达成了一维空间复杂度。

【列表展示原理】

由于(i-1, j-1)处的数据会被覆盖，则可以用一个新变量保存或者直接复制一份旧数组。

【优化后的伪代码】

【参考资料

https://leetcode.cn/problems/longest-common-subsequence/solution/by-baboon-m-kg01/

https://leetcode.cn/problems/longest-common-subsequence/solution/er-wei-shu-zu-dphe-yi-wei-shu-zu-dpde-sh-w96r/】

1. 【\*另解】哈希表+二分查找

https://leetcode.cn/problems/longest-common-subsequence/solution/liang-chong-fang-fa-zi-dian-er-fen-cha-z-pst8/

1. \*代码实现
2. 相关题目

连线

https://leetcode.cn/problems/delete-operation-for-two-strings/

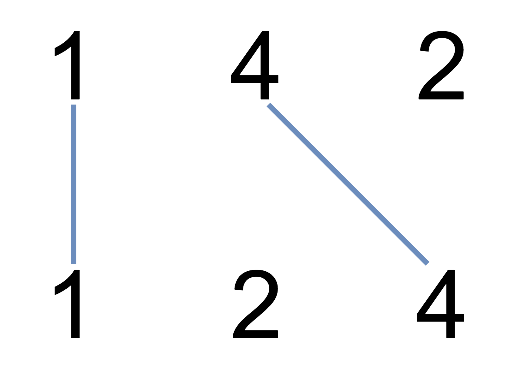
https://leetcode.cn/problems/minimum-ascii-delete-sum-for-two-strings/

**【例题】** 力扣1035. 不相交的线

在两条独立的水平线上按给定的顺序写下 nums1 和 nums2 中的整数。

现在，可以绘制一些连接两个数字 nums1[i] 和 nums2[j] 的直线，这些直线需要同时满足满足：nums1[i] == nums2[j] 且绘制的直线不与任何其他连线（非水平线）相交。

请注意，连线即使在端点也不能相交：每个数字只能属于一条连线。

以这种方法绘制线条，并返回可以绘制的最大连线数。

**输入：**nums1 = [1,4,2], nums2 = [1,2,4]

**输出：**2

**解释：**可以画出两条不交叉的线，如上图所示。

但无法画出第三条不相交的直线，因为从 nums1[1]=4 到 nums2[2]=4 的直线将与从 nums1[2]=2 到 nums2[1]=2 的直线相交。

**输入：**nums1 = [2,5,1,2,5], nums2 = [10,5,2,1,5,2]

**输出：**3

**输入：**nums1 = [1,3,7,1,7,5], nums2 = [1,9,2,5,1]

**输出：**2

**【例题】**最长公共子序列

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。

两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

**输入：**text1 = "abcde", text2 = "ace"

**输出：**3

**解释：**最长公共子序列是 "ace" ，它的长度为 3 。

**输入：**text1 = "abc", text2 = "abc"

**输出：**3

**解释：**最长公共子序列是 "abc" ，它的长度为 3 。

**输入：**text1 = "abc", text2 = "def"

**输出：**0

**解释：**两个字符串没有公共子序列，返回 0 。

class Solution:

    def maxUncrossedLines(self, text1: str, text2: str) -> int:

        m,n = len(text1),len(text2)

        dp = [[0]\*(n+1) for \_ in range(m+1)]

        for i in range(m):

            for j in range(n):

                if text1[i] == text2[j]:

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

                else:

                    dp[i+1][j+1] = max(dp[i][j+1],dp[i+1][j])

        return dp[m][n]