# 题目

给定三个字符串 s1、s2、s3，请你帮忙验证 s3 是否是由 s1 和 s2 交错 组成的。

两个字符串 s 和 t 交错 的定义与过程如下，其中每个字符串都会被分割成若干 非空 子字符串：

s = s1 + s2 + ... + sn

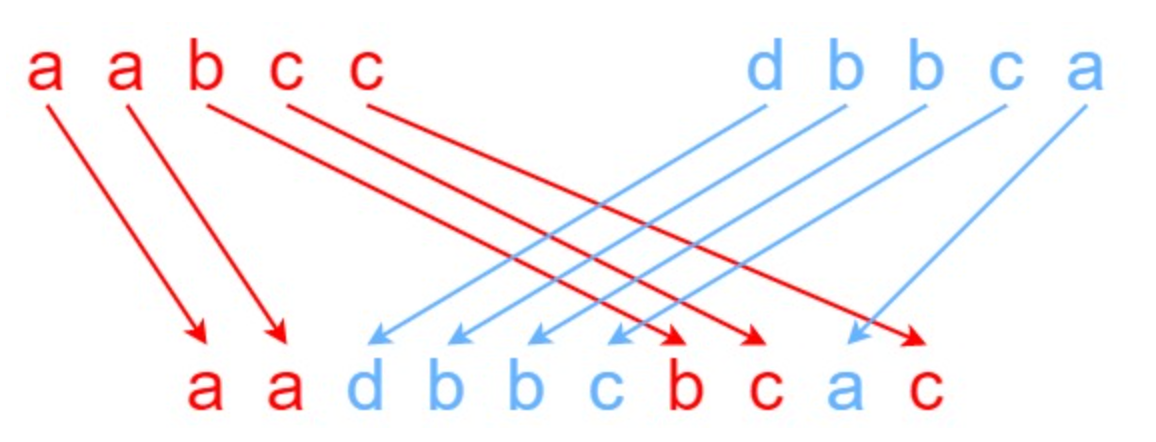
t = t1 + t2 + ... + tm

|n - m| <= 1

交错 是 s1 + t1 + s2 + t2 + s3 + t3 + ... 或者 t1 + s1 + t2 + s2 + t3 + s3 + ...

注意：a + b 意味着字符串 a 和 b 连接。

示例 1：



输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"

输出：true

示例 2：

输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"

输出：false

示例 3：

输入：s1 = "", s2 = "", s3 = ""

输出：true

提示：

0 <= s1.length, s2.length <= 100

0 <= s3.length <= 200

s1、s2、和 s3 都由小写英文字母组成

进阶：您能否仅使用 O(s2.length) 额外的内存空间来解决它?

# 分析

要验证s3是否由s1和s2交错组成，核心思路是动态规划（DP）。通过构建DP表（或优化为滚动数组），逐步判断s3的前i+j个字符是否能由s1的前i个字符和s2的前j个字符交错形成。

解题思路

1、前置验证：

若s1长度 + s2长度 ≠ s3长度，直接返回false（长度不匹配，不可能交错组成）。

2、DP状态定义：

设dp[i][j]表示s1的前i个字符（s1[0..i-1]）和s2的前j个字符（s2[0..j-1]）能否交错组成s3的前i+j个字符（s3[0..i+j-1]）。

3、边界条件：

- dp[0][0] = true：空s1和空s2可组成空s3。

- dp[i][0]（仅用s1组成s3）：需满足s1[0..i-1] == s3[0..i-1]，即dp[i][0] = dp[i-1][0] && (s1[i-1] == s3[i-1])。

- dp[0][j]（仅用s2组成s3）：需满足s2[0..j-1] == s3[0..j-1]，即dp[0][j] = dp[0][j-1] && (s2[j-1] == s3[j-1])。

4、状态转移方程：

dp[i][j]`的取值取决于两种可能的交错方式（需满足字符匹配）：

- 最后一个字符来自s1：dp[i-1][j] && (s1[i-1] == s3[i+j-1])（s1的第i个字符匹配s3的第i+j个字符，且s1前i-1个和s2前j个已能组成s3前i+j-1个）。

- 最后一个字符来自s2：dp[i][j-1] && (s2[j-1] == s3[i+j-1])（同理，s2的第j个字符匹配s3的第i+j个字符）。

因此，dp[i][j] = (dp[i-1][j] && s1[i-1] == s3[i+j-1]) || (dp[i][j-1] && s2[j-1] == s3[i+j-1])。

5、空间优化（进阶要求）：

观察到dp[i][j]仅依赖dp[i-1][j]（上一行）和dp[i][j-1]（同一行前一列），可将二维DP表优化为一维数组（大小为s2.length + 1），用滚动更新减少空间开销。

代码：

class Solution {

public:

bool isInterleave(string s1, string s2, string s3) {

int m = s1.size();

int n = s2.size();

int len3 = s3.size();

// 前置验证：长度不匹配，直接返回false

if (m + n != len3) {

return false;

}

// 空间优化：用一维数组dp[j]表示当前i下，s2前j个字符的匹配状态

vector<bool> dp(n + 1, false);

// 1. 初始化边界条件：dp[0][0] = true（i=0,j=0）

dp[0] = true;

// 2. 初始化i=0的情况（仅用s2组成s3前j个字符）

for (int j = 1; j <= n; ++j) {

dp[j] = dp[j - 1] && (s2[j - 1] == s3[j - 1]);

}

// 3. 填充DP数组（i从1到m，j从0到n）

for (int i = 1; i <= m; ++i) {

// 先更新j=0的情况（仅用s1组成s3前i个字符）

dp[0] = dp[0] && (s1[i - 1] == s3[i - 1]);

// 更新j从1到n的情况

for (int j = 1; j <= n; ++j) {

// 状态转移：来自s1的上一行（dp[j]未更新前是i-1时的值） 或 来自s2的前一列（dp[j-1]是i时的值）

dp[j] = (dp[j] && s1[i - 1] == s3[i + j - 1])

|| (dp[j - 1] && s2[j - 1] == s3[i + j - 1]);

}

}

// 最终结果：s1全串和s2全串能否组成s3全串

return dp[n];

}

};

代码解释

1、前置验证：快速排除长度不匹配的无效情况，避免后续计算。

2、一维DP初始化：

- dp[0] = true对应dp[0][0]，是空字符串的匹配状态。

- 初始化j从1到n：判断仅用s2的前j个字符能否组成s3的前j个字符。

3、滚动更新一维DP：

- 外层循环i（s1的前i个字符）：先更新dp[0]（仅用s1的前i个字符匹配`s3`的前i个字符）。

- 内层循环j（s2的前j个字符）：dp[j]未更新前存储的是i-1行的值（对应“来自s1的上一行”），dp[j-1]是已更新的i行值（对应“来自s2的前一列”），按状态转移方程更新。

4、结果提取：dp[n]表示s1全串（i=m）和s2全串（j=n）能否组成s3全串，直接返回即可。

复杂度分析

- 时间复杂度：O(m\*n)，其中m是s1长度，n是s2长度。需遍历m行n列的状态，每个状态计算为O(1)。

- 空间复杂度：O(n)（满足进阶要求），仅使用大小为n+1的一维数组，替代了(m+1)\*(n+1)的二维数组。