# 题目

给你一个字符串s ，找出其中最长的回文子序列，并返回该序列的长度。

子序列定义为：不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

示例 1：

输入：s = "bbbab"

输出：4

解释：一个可能的最长回文子序列为 "bbbb" 。

示例 2：

输入：s = "cbbd"

输出：2

解释：一个可能的最长回文子序列为 "bb" 。

提示：

1 <= s.length <= 1000

s 仅由小写英文字母组成

# 分析

## 方法一：动态规划

要解决找出字符串中最长回文子序列的问题，我们可以利用动态规划（DP）技术。回文子序列的特点是正着读和反着读都相同，且子序列不需要连续（只需保持字符顺序）。动态规划通过分解问题，利用子问题的解来构建更大问题的解，从而高效求解。

解题思路：

1、问题分析：

- 对于字符串s，最长回文子序列（LPS）的长度取决于子串的LPS长度。例如，对于子串s[i..j]（从索引i到j）：

若s[i] == s[j]，则LPS长度为子串s[i+1..j-1]的LPS长度加2（加上s[i]和s[j]）。

若s[i] != s[j]，则LPS长度为子串s[i+1..j]和s[i..j-1]的LPS长度的最大值（取两者中较长的）。

2、DP数组定义：

- 设dp[i][j]表示子串s[i..j]的最长回文子序列的长度。

3、状态转移方程：

- 若i == j（单个字符）：dp[i][j] = 1（单个字符本身是回文）。

- 若i < j：

若s[i] == s[j]：dp[i][j] = dp[i+1][j-1] + 2。

若s[i] != s[j]：dp[i][j] = max(dp[i+1][j], dp[i][j-1])。

4、填充顺序：

- 由于dp[i][j]依赖于更短的子串（i+1到j、i到j-1、i+1到j-1），因此需要按子串长度从短到长填充DP数组。先处理长度为1的子串，再处理长度为2的，直到处理长度为n（字符串总长度）的子串。

5、结果提取：

- 整个字符串s[0..n-1]的最长回文子序列长度为dp[0][n-1]。

代码：

class Solution {

public:

int longestPalindromeSubseq(string s) {

int n = s.size();

// dp[i][j] 表示 s[i..j] 的最长回文子序列长度

vector<vector<int>> dp(n, vector<int>(n, 0));

// 单个字符的最长回文子序列长度为1

for (int i = 0; i < n; ++i) {

dp[i][i] = 1;

}

// 按子串长度从2到n遍历（len=1已处理）

for (int len = 2; len <= n; ++len) {

// 遍历所有起始索引i，计算对应的结束索引j = i + len - 1

for (int i = 0; i + len - 1 < n; ++i) {

int j = i + len - 1;

if (s[i] == s[j]) {

// 若两端字符相同，长度为内部子串长度+2

// 若子串长度为2（i和j相邻），则内部子串长度为0

dp[i][j] = (len == 2) ? 2 : dp[i+1][j-1] + 2;

} else {

// 若两端字符不同，取左右子串的最大值

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

}

}

}

// 整个字符串的最长回文子序列长度

return dp[0][n-1];

}

};

解释：

1、DP数组初始化：

- 对角线dp[i][i]均初始化为1，因为单个字符的最长回文子序列就是其本身。

2、填充DP数组：

- 按子串长度len从2到n遍历，确保计算dp[i][j]时，依赖的更短子串的dp值已被计算。

- 对于每个子串s[i..j]：

若s[i] == s[j]，则长度为内部子串s[i+1..j-1]的长度加2（特殊处理len=2的情况，此时内部子串为空，长度直接为2）。

若s[i] != s[j]，则取去掉左侧字符（s[i+1..j]）或右侧字符（s[i..j-1]）后的最长回文子序列长度的最大值。

3、时间复杂度：O(n²)，其中n是字符串长度。需要填充n×n的DP数组，每个位置的计算是O(1)。

4、空间复杂度：O(n²)，用于存储dp数组。