# 题目

给你两个字符串 s 和 t ，统计并返回在 s 的 子序列 中 t 出现的个数。

测试用例保证结果在 32 位有符号整数范围内。

示例 1：

输入：s = "rabbbit", t = "rabbit"

输出：3

解释：

如下所示, 有 3 种可以从 s 中得到 "rabbit" 的方案。

rabbbit

rabbbit

rabbbit

示例 2：

输入：s = "babgbag", t = "bag"

输出：5

解释：

如下所示, 有 5 种可以从 s 中得到 "bag" 的方案。

babgbag

babgbag

babgbag

babgbag

babgbag

提示：

1 <= s.length, t.length <= 1000

s 和 t 由英文字母组成

# 分析

要解决“统计字符串s的子序列中t出现个数”的问题，核心思路是动态规划（DP）。通过构建二维DP表，存储s的前i个字符与t的前j个字符匹配时的子序列个数，逐步推导最终结果。

解题思路

1、子序列匹配的核心观察：

- 子序列允许跳过字符，但顺序必须保持一致。例如s="rabbbit"中，t="rabbit"的匹配依赖于s中字符按r→a→b→b→i→t的顺序出现，且可跳过多余的b。

- 对于s的第i个字符（s[i-1]）和t的第j个字符（t[j-1]），存在两种情况：

1）若s[i-1] == t[j-1]：可选择用s[i-1]匹配t[j-1]（此时子序列个数等于s前i-1个字符匹配t前j-1个字符的个数），也可选择跳过s[i-1]（此时子序列个数等于s前i-1个字符匹配t前j个字符的个数），总个数为两者之和。

2）若s[i-1] != t[j-1]：只能跳过s[i-1]，子序列个数等于s前i-1个字符匹配t前j个字符的个数。

2、DP表定义：

- 设dp[i][j]表示s的前i个字符（s[0..i-1]）中，t的前j个字符（t[0..j-1]）作为子序列出现的个数。

3、边界条件：

- 当j=0（t为空字符串）：空字符串是任何字符串的子序列，故dp[i][0] = 1（对所有i≥0）。

- 当i=0且j>0（s为空但t非空）：无法匹配，故dp[0][j] = 0（对所有j>0）。

4、状态转移方程：

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

- 若s[i-1] != t[j-1]：dp[i][j] = dp[i-1][j]

代码：

class Solution {

public:

int numDistinct(string s, string t) {

int m = s.size();

int n = t.size();

// 若t的长度大于s，直接返回0（无法匹配）

if (n > m) return 0;

// DP表：dp[i][j]表示s前i个字符中t前j个字符的子序列个数

vector<vector<unsigned int>> dp(m + 1, vector<unsigned int>(n + 1, 0));

// 边界条件1：t为空字符串时，所有s的子序列都包含空字符串，故dp[i][0] = 1

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

dp[i][0] = 1;

}

// 填充DP表（i从1到m，j从1到n）

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

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

// 情况1：当前字符匹配，可选择用s[i-1]匹配t[j-1]，或跳过s[i-1]

if (s[i-1] == t[j-1]) {

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

}

// 情况2：当前字符不匹配，只能跳过s[i-1]

else {

dp[i][j] = dp[i-1][j];

}

}

}

// 返回s全串匹配t全串的子序列个数

return dp[m][n];

}

};

代码解释

1、边界处理：若t的长度大于s，直接返回0（因子序列长度不能超过原字符串，无法匹配）。

2、DP表初始化：

- 用unsigned int存储dp值，避免因结果过大导致的整数溢出（题目保证结果在32位有符号整数范围内，unsigned int可覆盖）。

- dp[i][0] = 1：空字符串是任何字符串的子序列，这是推导的基础。

3、DP表填充：

- 外层循环遍历s的每个字符（i从1到m），内层循环遍历t的每个字符（j从1到n）。

- 按字符是否匹配执行对应的状态转移，确保每个子问题的解基于更小的子问题。

4、结果提取：dp[m][n]即s全串中t全串作为子序列的个数，直接返回即可。

复杂度分析

- 时间复杂度：O(m\*n)，其中m是s的长度，n是t的长度。需填充(m+1)\*(n+1)的DP表，每个单元格的计算为O(1)。

- 空间复杂度：O(m\*n)，用于存储二维DP表。可优化为O(n)（滚动数组，因dp[i][j]仅依赖dp[i-1][...]），但基础实现更直观，且在m,n≤1000时空间开销可接受。