# 题目

给定字符串s和t，判断s是否为t的子序列。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）。

你可以认为s和t中仅包含英文小写字母。字符串t可能会很长（长度 ~= 500,000），而s是个短字符串（长度<=100）。

字符串的一个子序列是原始字符串删除一些（也可以不删除）字符而不改变剩余字符相对位置形成的新字符串。（例如，"ace"是"abcde"的一个子序列，而"aec"不是）。

**示例1:**

s = "abc", t = "ahbgdc"

返回true.

**示例2:**

s = "axc", t = "ahbgdc"

返回false.

**后续挑战 :**

如果有大量输入的 S，称作S1, S2, ... , Sk 其中 k >= 10亿，你需要依次检查它们是否为 T 的子序列。在这种情况下，你会怎样改变代码？

# 分析

## 方法一：双指针

**思路：**

可以知道的是，当扫描s中的第k个字符时，假如它在t字符串中的第i位和第j位都出现过(i < j)，那么我们从左到右扫描到第i位时，就认为已经找到了s中第k个字符。因为i后面有更多的备选字符可以用来找s中的剩余字符。也就是说，我们在t中找字符时，是严格不回溯的。这个问题可以使用双指针解决。

初始化指针i，j为0，分别指向s和j的第0个字符，在t中找到s[i]字符后，i++试图找下一个字符。

若最后i到达s末尾，则说明找到了该字符串。

**代码：**

class Solution {

public:

bool isSubsequence(string s, string t) {

int i = 0, j = 0;

int m = s.size(), n = t.size();

while(i < m && j < n){

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

i++;

j++;

}

else j++;

}

return i == m;

}

};

**复杂度分析：**

时间复杂度：O(n+m)，其中n为s的长度，m为t的长度。每次无论是匹配成功还是失败，都有至少一个指针发生右移，两指针能够位移的总距离为n+m。

空间复杂度：O(1)。

## 方法二：动态规划

**思路：**

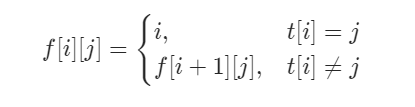
本题与[最长公共子序列（动态规划）](https://leetcode-cn.com/problems/longest-common-subsequence/solution/zui-chang-gong-gong-zi-xu-lie-dong-tai-g-fhyd/)相似，可以参考。

考虑前面的双指针的做法，我们注意到我们有大量的时间用于在t中找到下一个匹配字符。

这样我们可以预处理出对于t的每一个位置，从该位置开始往后每一个字符第一次出现的位置。

我们可以使用动态规划的方法实现预处理，令f[i][j]表示字符串t中从位置i开始往后字符j第一次出现的位置。在进行状态转移时，如果t中位置i的字符就是j，那么f[i][j]=i，否则j出现在位置i+1开始往后，即f[i][j]=f[i+1][j]，因此我们要倒过来进行动态规划，从后往前枚举i。

这样我们可以写出状态转移方程：



假定下标从0开始，那么f[i][j]中有0≤i≤m−1，对于边界状态f[m−1][..]，我们置f[m][..]为m，让f[m−1][..]正常进行转移。这样如果f[i][j]=m，则表示从位置i开始往后不存在字符j。

这样，我们可以利用f 数组，每次O(1)地跳转到下一个位置，直到位置变为m或s中的每一个字符都匹配成功。

**代码：**

class Solution {

public:

bool isSubsequence(string s, string t) {

int n = s.size(), m = t.size();

vector<vector<int> > f(m + 1, vector<int>(26, 0));

for (int i = 0; i < 26; i++) {

f[m][i] = m;

}

for (int i = m - 1; i >= 0; i--) {

for (int j = 0; j < 26; j++) {

if (t[i] == j + 'a')

f[i][j] = i;

else

f[i][j] = f[i + 1][j];

}

}

int add = 0;

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

if (f[add][s[i] - 'a'] == m) {

return false;

}

add = f[add][s[i] - 'a'] + 1;

}

return true;

}

};

**复杂度分析：**

时间复杂度：O(m×∣Σ∣+n)，其中n为s的长度，m为t的长度，Σ为字符集，在本题中字符串只包含小写字母，∣Σ∣=26。预处理时间复杂度O(m)，判断子序列时间复杂度O(n)。

如果是计算k个平均长度为n的字符串是否为t的子序列，则时间复杂度为O(m×∣Σ∣+k×n)。

空间复杂度：O(m×∣Σ∣)，为动态规划数组的开销。