题意分析

给定只包含字母的两个字符串 A,B，求 A,B 两个字符串的最长公共子序列，要求构成子序列的子串长度都必须大于等于 3。

比如"abcdefghijklmn"和"ababceghjklmn"，其最长满足题意要求的子序列为"abcjklmn"，其由公共子串"abc"和"jklmn"组成。

这里我们要注意子串和子序列的区别：

子串：连续的元素
子序列：不连续的元素
比如"abcdefghijklmn"和"ababceghjklmn"的最长公共子串就只是"jklmn"了。

算法分析

首先我们来复习一道经典的题目：

给定只包含字母的两个字符串 A,B，求 A,B 两个字符串的最长公共子序列。

比如 A="**abcde**"和 B="**abdfg**"的最长公共子序列为"**abd**"

对于最长公共子序列，我们知道解法为

```
dp[0][0..j] = 0      // 边界
dp[0..i][0] = 0      // 边界
For i = 1 .. n
    For j = 1 .. m
        If a[i] == b[j] Then
            dp[i][j] = dp[i - 1][j - 1] + 1
        Else
            dp[i][j] = Max(dp[i - 1][j], dp[i][j - 1])
        End If
    End For
End For
```

而这一道题目是在最长公共子序列上加入了一个条件：构成最长公共子序列的每一个子串长度必须大于等于 3.

一个简单的想法：我们求出最长公共子序列，然后将其中长度小于 3 的部分去掉。

显然，这是不对的。

举个例子："**aaabaa**"和"**acaaaca**"的最长子序列为"**aaaaa**"。其对应关系为：

```
a  aaba  a
acaa  aca
```

因为在"**acaaaca**"中第一个字母 **a** 长度为 1，所以我们需要去掉它，对应的我们也去掉了"**aaabaa**"中第一个字母 **a**。

```
  aaba  a
 caa  aca
```

此时构成"**aaabaa**"和"**acaaaca**"公共子序列的 3 个子串为"**aa**","**a**"和"**a**"，长度都小于了 3，所以全部删去，则得到了新

的公共子序列长度为 0。

这显然不正确，因为实际有符合题意要求的公共子序列：

```
   aaa  baa
ac  aaa  ca
```

其中包含有长度为 3 的公共子序列。

对最大公共子序列的结果进行再次处理这个方法不可行，那么我们只能从计算公共子序列的算法着手。

首先我想我们可以做一个预处理，用 f[i][j]表示以字符串 A 的第 i 个字母作为结尾的前缀和以字符串 B 的第 j 个字母作为结尾的前缀的公共后缀的长度。这样看上去似乎很绕，不如举个例子：

A="abcd"和 B="acbc"。f[3][4]的就表示 A[1..3]和 B[1..4]的公共后缀的长度，其中 A[1..3]="abc"，B[1..4]="acbc"，其公共后缀为"bc"，所以 f[3][4]=2.

预处理的伪代码为：

```
For i = 1 .. n
    For j = 1 .. m
        If a[i] == b[j] Then
            f[i][j] = f[i - 1][j - 1] + 1
        Else
            f[i][j] = 0
        End If
    End For
End For
```

有了这个预处理的数组，我们可以在原来最大公共子序列上做这样一个改进：

```
dp[0][0..j] = 0     // 边界
dp[0..i][0] = 0     // 边界
For i = 1 .. n
    For j = 1 .. m
        dp[i][j] = Max(dp[i - 1][j], dp[i][j - 1])
        if f[i][j] >= 3 Then // 改进
            dp[i][j] = dp[i - f[i][j]][j - f[i][j]] + f[i][j]
        End If
    End For
End For
```

这个改进的意义为：当我们出现一个长度大于 3 的子串时，我们就直接将这个子串合并入我们的子序列。

加入这个改进后，我们通过了样例的数据，这样看上去似乎就应该没什么问题了。

然而事实并不是这样，在**这道题目中还隐藏着陷阱**：

比如"**abcdef**"和"**abcxcdef**"

根据我们算法，上面这个例子算出的结果为 4，然而其实际的结果应该为 6，即"**abc**"和"**def**"两个公共子串构成的子

序列。

那么出错的原因在哪？就在字符串"**cdef**"上。

我们计算结果出 4 是因为将"**cdef**"看做了一个整体，而将"**abcdef**"分割成了"**ab**"和"**cdef**"。

在 DP 的过程中 f[6][7] = 4，我们使用了 dp[6][7] = dp[2][3] + 4，而 dp[2][3] = 0，所以 dp[6][7] = 4。

ab cdef
abcxcdef
而实际上的最优解是将 f[6][7]看作 3，dp[6][7] = dp[3][4] + 3，其中 dp[3][4] = 3，得到了 dp[6][7] = 6。

abc def
abcxcdef
也就是说，如果我们将 f[i][j]>3 的子串进行分割，有可能得到更优的情况。因此我们需要进一步的改进：

```
dp[0][0..j] = 0       // 边界
dp[0..i][0] = 0       // 边界
For i = 1 .. n
    For j = 1 .. m
        dp[i][j] = 0
        dp[i][j] = Max(dp[i - 1][j], dp[i][j - 1])
        If f[i][j] >= 3 Then //     改进
            For k = 3 .. f[i][j] // 枚举分割长度
                dp[i][j] = Max(dp[i][j], dp[i - k][j - k] + k)
            End For
        End If
    End For
End For
```

但是这样的改进使得整个算法的时间复杂度变为了 O(n^3)，当 n=2100 时，有可能会超时。

让我们考虑一下如何进一步改进这个算法。以上算法复杂度高的地方在于对于每一个(i, j)，我们为了计算 dp[i][j]都需要枚举分割长度 k：

```
For k = 3 .. f[i][j]       // 枚举分割长度
    dp[i][j] = Max(dp[i][j], dp[i - k][j - k] + k)
End For
```

这一步实际上我们计算了 max{dp[i-k][j-k]+k}, k=3..f[i][j]。我们不妨把它记作 dp1[i][j]，即：

dp1[i][j] = max{dp[i-k][j-k]+k} = max{dp[i-3][j-3]+3, dp[i-4][j-4]+4, dp[i-5][j-5]+5, ... }

同时

dp1[i-1][j-1] = max{dp[i-1-3][j-1-3]+3, dp[i-1-4][j-1-4] + 4, dp[i-1-5][j-1-5]+5 ... }
= max{dp[i-4][j-4]+3, dp[i-5][j-5]+4, dp[i-6][j-6]+5, ... }

我们可以发现，dp1[i][j]的展开式中除了 dp[i-3][j-3]+3 这一项，是与 dp1[i-1][j-1]中的每一项一一对应的，并且刚好大

1。所以实际上 dp[i-1][j-1]计算时枚举过分割长度，我们并不需要再次计算：

dp1[i][j] = max{dp1[i-1][j-1] + 1, dp[i-3][j-3]+3}

可以看出 dp1 只枚举两个地方：

1. (i - f[i][j], j - f[i][j])
2. (i - 3, j - 3)

所以得出这样性质以后直接用在 dp 上就好了，**dp[i][j] = max(dp[i - 3][j - 3] + 3, dp[i - f[i][j]][j - f[i][j]]+f[i][j])**。

最后得到我们新的伪代码如下:

```
dp[0][0..j] = 0        // 边界
dp[0..i][0] = 0        // 边界
For i = 1 .. n
    For j = 1 .. m
        dp[i][j] = 0
        dp[i][j] = Max(dp[i-1][j], dp[i][j-1])
        If f[i][j] >= 3 Then //  改进
            dp[i][j] = Max(dp[i][j],dp[i-3][j-3]+3,dp[i - f[i][j]][j - f[i][j]] + f[i][j])
        End If
    End For
End For
```

结果分析

这个题目是在经典的动态规划题目《最长公共子序列》上做了一点修改。虽然只增加了一个条件，不过难度增大很多。能想出一个复杂度是 O(n^2)的正确算法不是很容易，需要仔细分析清楚各种情况。一不小心就会掉进各种陷阱里。

很多选手都能够想到经典最长子序列的改进算法而获得 80 分。

剩下的测试点则对应了算法分析中提到的陷阱，所**以能否找出这种特殊的例子也是解决这道题的关键**。

很多 O(N^2)的程序不能通过"babad"和"babacabad"这组数据。

O(n^3)算法

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
const int N=2200;
int len1,len2;
int dp[N][N];
int f[N][N];
char s1[N],s2[N];
int main()
{
    scanf("%s%s",s1+1,s2+1);
    len1=strlen(s1+1);
    len2=strlen(s2+1);
    f[0][0]=0;
    for (int i=1; i<=len1; i++)
    {
        for (int j=1; j<=len2; j++)
        {
            if(s1[i]==s2[j])
                f[i][j]=f[i-1][j-1]+1;
            else
                f[i][j]=0;
        }
    }
    for (int i=0; i<=len1; i++)
    {
        dp[i][0]=0;
        dp[0][i]=0;
    }
    for (int i=1; i<=len1; i++)
    {
        for (int j=1; j<=len2; j++)
        {
            dp[i][j]=0;
            dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
            if(f[i][j]>=3)
            {
                for (int k=3; k<=f[i][j]; k++)
                    dp[i][j]=max(dp[i][j],dp[i-k][j-k]+k);
            }

        }
    }
    printf("%d\n",dp[len1][len2]);
    return 0;
}
```

O(n^2)算法:

```
#include <iostream>
#include <stdio.h>
#include <string.h>
#include <algorithm>
using namespace std;
const int N=2200;
int len1,len2;
int dp[N][N];
int f[N][N];
char s1[N],s2[N];
int main()
{
	scanf("%s%s",s1+1,s2+1);
	len1=strlen(s1+1);
	len2=strlen(s2+1);
	f[0][0]=0;
	for (int i=1; i<=len1; i++)
	{
		for (int j=1; j<=len2; j++)
		{
			if(s1[i]==s2[j])
				f[i][j]=f[i-1][j-1]+1;
			else
				f[i][j]=0;
		}
	}
	for (int i=0; i<=len1; i++)
	{
		dp[i][0]=0;
		dp[0][i]=0;
	}
	for (int i=1; i<=len1; i++)
	{
		for (int j=1; j<=len2; j++)
		{
			dp[i][j]=0;
			dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
			if(f[i][j]>=3)
			{
				dp[i][j]=max(dp[i][j],dp[i-3][j-3]+3);
				dp[i][j]=max(dp[i][j],dp[i-f[i][j]][j-f[i][j]]+f[i][j]);
			}
		}
	}
	printf("%d\n",dp[len1][len2]);
	return 0;
}
```