**[最长公共子序列和字符串相似度](http://www.cnblogs.com/DiaoCow/archive/2010/04/16/1713867.html)**

经典的DP问题。

1.假设求X = {x1 , x2 , x3, ... , xm}和Y = {y1 , y2 , y3, ... ,yn}的最长公共子序列MaxLen(Xm , Yn)，我们可以这样思考：

若xm == yn ,那么最长公共序列为MaxLen(Xm-1 , Yn-1) + 1；

若xm != yn ,那么最长公共序列为MaxLen(Xm-1 ,Yn)和MaxLen(Xm ,Yn-1)中的最大值；

因此状态转移方程是：c[i][j] = c[i-1][j-1] + 1 (Xi == Yj) , Max(c[i][j-1] , c[i-1][j])  (Xi != Yj)

从递归式中我们可以看出，对于c[i][j]我们总是通过c[i-1][j-1]或者c[i][j-1]和c[i-1][j]求出，即下图

**(i-1,j-1)**   **(i-1,j)**

**(i ,  j-1)**   **求(i,j)**

因此可以从前往后递推求解

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

#include <stdio.h>  
#include <string.h>  
  
#define N 257  
#define Max(a , b) ((a) > (b) ? (a) : (b))  
  
char x[N], y[N];   
int c[N][N];  
  
int ComSub(int m , int n)  
{   
 int i , j;   
   
 for(i = 0 ; i <= m ; i++) c[i][0] = 0;   
 for(j = 0 ; j <= n ; j++) c[0][j] = 0;   
  
 for(i = 1 ; i <= m ; i++)   
 {   
 for(j = 1 ; j <= n ; j++)   
 {   
 if(x[i] == y[j])   
 c[i][j] = c[i - 1][j - 1] + 1;   
 else   
 c[i][j] = Max(c[i - 1][j] , c[i][j - 1]);   
 }   
 }   
 return c[m][n];  
}  
  
int main(void)  
{   
 int n , m;   
  
 while(scanf("%s%s", x + 1 , y + 1) != EOF)   
 {   
 m = strlen(x + 1);   
 n = strlen(y + 1);   
 printf("%d\n", ComSub(m , n));   
 }   
 return 0;  
}

上面程序的空间复杂度和时间复杂度都是O(mn)。由于受之前0/1背包启发(观察递推求解过程），我们可以用类似的方法把空间复杂度降低为O(Min(m,n))

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

#include <stdio.h>  
#include <string.h>  
  
#define N 257  
#define Max(a , b) ((a) > (b) ? (a) : (b))  
  
char x[N], y[N];   
int c[2][N];  
  
int ComSub(int m , int n)  
{   
 int i , j;   
   
 memset(c , 0 , sizeof(c));  
 for(i = 1 ; i <= m ; i++)   
 {   
 memcpy(c[0] , c[1] , sizeof(c[0])); //循环滚动赋值  
 for(j = 1 ; j <= n ; j++)   
 {   
 if(x[i] == y[j])   
 c[1][j] = c[0][j - 1] + 1;   
 else   
 c[1][j] = Max(c[0][j] , c[1][j - 1]);   
 }   
 }   
 return c[1][n];  
}  
  
int main(void)  
{   
 int n , m;   
  
 while(scanf("%s%s", x + 1 , y + 1) != EOF)   
 {   
 m = strlen(x + 1);   
 n = strlen(y + 1);   
 printf("%d\n", ComSub(m , n));   
 }   
 return 0;  
}

还可以在优化，避免copy:令p = 0 , 在外循环设置p =p ^ 1,然后c[p][j]与c[1-p][j]交替使用

2.计算字符串的相似度和求最长公共子序列的思考方法非常相近。

假设求X = {x1 , x2 , x3, ... , xm}和Y = {y1 , y2 , y3, ... ,yn}的相似度D(Xm , Yn)，我们可以这样思考：

若xm == yn , 那么只要求D(Xm-1, Yn-1)就可以了；

若xm != yn ,那么我们总可以通过一步操作（以X为例，可以在xm后添加yn，删除xm或者修改xm为yn)来达到第一步的匹配，然后求剩下字符串的相似度即可，即Min(D(Xm, Yn-1) , D(Xm-1, Yn) , D(Xm-1, Yn-1)) + 1；

因此状态转移方程是：c[i][j] = c[i-1][j-1]  (Xi == Yj) , Min(c[i][j-1] , c[i-1][j] , c[i-1][j-1])  (Xi != Yj)

先用备忘录法写一个（向前递归与向后递归本质是没有区别的）：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

#include <stdio.h>  
#include <string.h>  
#define N 256  
#define Min(a , b) ((a) < (b) ? (a) : (b))  
  
char x[N], y[N];  
int c[N][N];  
  
int CalStrDistance(int i , int m , int j , int n)  
{   
 int t1 , t2 , t3 , t;   
  
 if(c[i][j] > 0) return c[i][j]; //避免重复计算   
 if(i == m) return n - j;   
 if(j == n) return m - i;   
   
 if(x[i] == y[j])   
 {   
 return c[i][j] = CalStrDistance(i + 1 , m , j + 1 , n);   
 }   
 else   
 {   
 t1 = CalStrDistance(i + 1 , m , j + 1 , n);   
 t2 = CalStrDistance(i , m , j + 1 , n);   
 t3 = CalStrDistance(i + 1 , m , j , n);   
 t = Min(t1, t2);   
 return c[i][j] = Min(t, t3) + 1;   
 }  
}  
  
int main(void)  
{   
 int n , m , k;   
 float f;   
   
 while(scanf("%s%s", x, y) != EOF)   
 {   
 m = strlen(x);   
 n = strlen(y);   
 memset(c ,0 , sizeof(c));   
 k = CalStrDistance(0, m, 0, n) + 1;   
 f = (float)1 / k;   
 printf("%.2f\n", f);   
 }   
 return 0;  
}

动态规划自底向上求解：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

#include <stdio.h>  
#include <string.h>  
  
#define N 256  
#define Min(a , b) ((a) < (b) ? (a) : (b))  
  
char x[N], y[N];   
int c[N][N];  
  
int CalStrDistance(int m , int n)  
{   
 int i , j , k;  
  
 for(i = 0 ; i <= m ; i++) c[i][0] = i;  
 for(j = 0 ; j <= n ; j++) c[0][j] = j;  
   
 for(i = 1 ; i <= m ; i++)  
 {  
 for(j = 1 ; j <= n ; j++)  
 {  
 if(x[i] == y[j])  
 {  
 c[i][j] = c[i-1][j-1];  
 }  
 else  
 {  
 k = Min(c[i][j-1] ,c[i-1][j-1]);  
 c[i][j] = Min(k ,c[i-1][j]) + 1;  
 }  
 }  
 }  
 return c[m][n];  
}  
  
  
int main(void)  
{   
 int n , m , k;   
 float f;  
  
 while(scanf("%s%s", x + 1, y + 1) != EOF)   
 {   
 m = strlen(x + 1);   
 n = strlen(y + 1);   
  
 k = CalStrDistance(m, n) + 1;  
 f = (float)1 / k;  
 printf("%.2f\n", f);   
 }   
 return 0;  
}

对于动态规划的方法，我们可以对空间复杂度继续优化O(N)，在此不再赘述。