392.判断子序列

题目链接：https://leetcode-cn.com/problems/is-subsequence/

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）。

示例 1：

输入：s = "abc", t = "ahbgdc"

输出：true

示例 2：

输入：s = "axc", t = "ahbgdc"

输出：false

> 由于子序列是不连续的那么dp数组的定义如下:
> 1、dp[i][j]表示字符串S下标为0....i-1是否为字符串t下标为0...j-1的子序列
> 2、递推公式
> if(s[i-1]==t[j-1])dp[i][j]=dp[i-1][j-1]||dp[i][j-1];
> else dp[i][j]=dp[i][j-1];

3、初始化 最难部分就是

> dp[0][0....t.size()]=true,其余部分都初始化为false
> 4、遍历顺序
> 先字符串s后字符串t

```
class Solution {
public:
    bool isSubsequence(string s, string t) {
   vector<vector<bool>> dp(s.size()+1,vector<bool>(t.size()+1,false));
        if(s.size()==0) return true;
        if(t.size()==0) return false;
        //对于dp[0][i]应该都必须初始化为0
        for(int i=0;i<=t.size();i++) dp[0][i]=true;
        for(int i=1;i<=s.size();i++){
```

```
            for(int j=1;j<=t.size();j++){
                if(s[i-1]==t[j-1])dp[i][j]=dp[i-1][j-1]||dp[i][j-1];
                else dp[i][j]=dp[i][j-1];
            }
        }
        return dp[s.size()][t.size()];
    }
};
```

115.不同的子序列

给定一个字符串 s 和一个字符串 t，计算在 s 的子序列中 t 出现的个数。

字符串的一个 子序列 是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，"ACE" 是 "ABCDE" 的一个子序列，而 "AEC" 不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

输入： s = "rabbbit", t = "rabbit"
输出： 3
解释：
如下图所示，有 3 种可以从 s 中得到 "rabbit" 的方案。
(上箭头符号 ^ 表示选取的字母)
rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^

> 由于本题目中子序列是不连续的所以dp数组的定义如下:
> 1dp[i][j]表示字符串s下标为0..i-1在字符串t下标为0...j-1出现的个数

2递推公式 难点

> 这一类问题,基本是要分析两种情况
> s[i-1]与t[j-1]**相等情况**和s[i-1]与t[j-1]**不相等情况**。

s[i-1]与t[j-1]相等情况的可以有两部分组成:
一部分使用s[i-1]来匹配，那么个数为dp[i-1][j-1];
另一部分不使用s[i-1]匹配，那么个数就是dp[i-1][j];
s[i-1]与t[j-1]不相等情况那么就是dp[i-1][j]
3初始化
dp[0][0]=1;
dp[0][0.....t.size()=0;
dp[0.....s.size()][0]=1;
4遍历顺序:先字符串s后字符串t

```
class Solution {
public:
    int numDistinct(string s, string t) {
 vector<vector<int>>dp(s.size()+1,vector<int>(t.size()+1,0));
 for(int i=0;i<=s.size();i++) dp[i][0]=1;
 for(int i=1;i<=s.size();i++){
 for(int j=1;j<=t.size();j++){
 if(t[i-1]==s[j-1]) dp[i][j]=dp[i-1][j-1]+dp[i-1][j];
 else dp[i][j]=dp[i-1][j];
  }
  }
  return dp[s.size()][t.size()];
    }
};
```

583. 两个字符串的删除操作

题目链接：https://leetcode-cn.com/problems/delete-operation-for-two-strings/

给定两个单词 word1 和 word2，找到使得 word1 和 word2 相同所需的最小步数，每步可以删除任意一个字符串中的一个字符。

示例：

输入: "sea", "eat"
输出: 2 解释: 第一步将"sea"变为"ea"，第二步将"eat"变为"ea"

1、dp数组的定义及其下标为:
dp[i][j]表示word1下标为0....i-1和word2下标为0...j-1两个相同所需要删除的最少字符数

2、递推公式
if(word1[i-1]==word2[j-1]) dp[i][j]=dp[i-1][j-1];
else dp[i][j]=min(dp[i-1][j],dp[i][j-1])+1;
3 初始化
dp[0][j]=j;
dp[i][0]=i;

4遍历 从word1为外层遍历,word2为内层遍历

```cpp
class Solution {
public:
    int minDistance(string word1, string word2) {
 vector<vector<int>>dp(word1.size()+1,vector<int>(word2.size()+1,0));
 //初始化,dp数组的下标表示第几个字符即数量，word1的下标表示位置
 for(int i=0;i<=word1.size();i++) dp[i][0]=i;
 for(int j=0;j<=word2.size();j++) dp[0][j]=j;
 //遍历
 for(int i=1;i<=word1.size();i++){
  for(int j=1;j<=word2.size();j++){
  if(word1[i-1]==word2[j-1]) dp[i][j]=dp[i-1][j-1];
  else dp[i][j]=min(dp[i][j-1],dp[i-1][j])+1;
  }
  }
  return dp[word1.size()][word2.size()];
     }
};
```

子序列问题的最终问题就是编辑距离：

72. 编辑距离

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

- 插入一个字符
- 删除一个字符
- 替换一个字符

示例 1：输入：word1 = "horse", word2 = "ros"
输出：3
解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')

dp数组的下标及其含义:

> dp[i][j]:表示word1单词的下标为0...i-1部分以及word2单词的下标为0...j-1部分相等时所需的最少操作数

递推公式

> if(word1[i-1]==word2[j-1]) dp[i][j]=dp[i-1][j-1];
> else 此时两个单词不相等部分可以用增加、删除以及替换来弥补
> 由于增加和删除是一对逆运算，删除word1的字符相当于增加word2的字符即只要考虑一个即可。
> 替换的部分就要另算一个部分
> dp[i][j]=min(dp[i-1][j]+1,min(dp[i][j-1]+1,dp[i-1][j-1]+1));

初始化

> dp[0][j]=j;dp[i][0]=i;

遍历顺序

从上到下，从左到右

```
class Solution {
public:
    int minDistance(string word1, string word2) {
 vector<vector<int>>dp(word1.size()+1,vector<int>(word2.size()+1,0));
 //初始化,dp数组的下标表示第几个字符即数量，word1的下标表示位置
 for(int i=0;i<=word1.size();i++) dp[i][0]=i;
 for(int j=0;j<=word2.size();j++) dp[0][j]=j;
 //遍历
 for(int i=1;i<=word1.size();i++){
  for(int j=1;j<=word2.size();j++){
  if(word1[i-1]==word2[j-1])dp[i][j]=dp[i-1][j-1];
  else dp[i][j]=min(dp[i-1][j]+1,min(dp[i][j-1]+1,dp[i-1][j-1]+1));
  }
  }
  return dp[word1.size()][word2.size()];
    }
};
```

647. 回文子串

题目链接：https://leetcode-cn.com/problems/palindromic-substrings/

给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。

示例 1：

输入："abc"
输出：3
解释：三个回文子串: "a", "b", "c"

示例 2：

输入："aaa"
输出：6
解释：6个回文子串: "a", "a", "a", "aa", "aa", "aaa"

提示：输入的字符串长度不会超过 1000。

dp数组下标及其含义

> dp[i][j]:表示区间范围[i,j](左闭右闭)的子串是否是回文子串，dp[i][j]数据类型为bool

递推公式

> dp[i][j]是建立于更小区间的结果。且i<j
> if(nums[i+1]==nums[j-1]) dp[i][j]=dp[i+1][j-1];

初始化

> 把数组中的值均初始化为false
> 由于dp结果来自于左下角的值,所以计算的空间为矩阵的右上三角
> 对角线初始化为true

```
for(int i=0;i<nums.size();i++){
 dp[i][i]=true;
 if(i+1<nums.size()&&nums[i+1]==nums[i]) dp[i][j]=true;
}
```

遍历顺序

> **从下到上，从左到右**

```
class Solution {
public:
```

```
    int countSubstrings(string s) {
int result=0;
vector<vector<bool>> dp(s.size(),vector<bool>(s.size(),false));
//初始化
for(int i=0;i<s.size();i++){
dp[i][i]=true;
result++;
 if(i+1<s.size()&&s[i+1]==s[i]){
 dp[i][i+1]=true;
 result++;
 }
 }
 //遍历
 for(int i=s.size()-3;i>=0;i--){
 for(int j=i+2;j<s.size();j++){
 if(s[i]==s[j]) dp[i][j]=dp[i+1][j-1];
 if(dp[i][j])result++;
 }
 }
 return result;
    }
};
```

双指针法：

首先**确定回文串**，就是找中心然后向两边扩散看是不是对称的就可以了。

在遍历中心点的时候，要注意中心点有两种情况。一种就是一个元素点作为中心点，两个元素也可以作为中心点.

为什么不考虑三个或者三个以上中心点呢?

因为三个或者三个以上中心点都可以从**一个元素点**或者**两个元素点**为中心推导出来

```
class Solution {
public:
    int countSubstrings(string s) {
 int result=0;
 for(int i=0;i<s.size();i++){
 result+=extend(s,i,i,s.size());
 result+=extend(s,i,i+1,s.size());
 }
 return result;
    }
```

```
    //计算以left和right元素作为中心，判断是否是回文数列
    int extend(string& s,int left,int right,int size){
    int result=0;
    while(left>=0&&right<size&&s[left--]==s[right++])
    result++;
    return result;
    }
};
```

516.最长回文子序列

题目链接：https://leetcode-cn.com/problems/longest-palindromic-subsequence/

给定一个字符串 s ，找到其中最长的回文子序列，并返回该序列的长度。可以假设 s 的最大长度为 1000 。

示例 1:
输入: "bbbab"
输出: 4
一个可能的最长回文子序列为 "bbbb"。

示例 2:
输入:"cbbd"
输出: 2
一个可能的最长回文子序列为 "bb"。

提示：

- 1 <= s.length <= 1000
- s 只包含小写英文字母

回文子串是连续的，回文子序列可不是连续的

方法1：不能这么做由于子序列不是连续的，不能这么做

```
class Solution {
public:
    int longestPalindromeSubseq(string s) {
    int maxn=0;
    for(int i=0;i<s.size();i++){
    maxn=max(maxn,extend(s,i,i,s.size()));
    maxn=max(maxn,extend(s,i,i+1,s.size()));
```

```
    }
    return maxn;
        }
     int extend(string& s,int left,int right,int size){
     while(left>=0&&right<size&&s[left]==s[right]){
     left--;
     right++;
     }
     return right-left+1;
     }
};
```

dp数组及其下标含义

> dp[i][j]:在区间范围[i,j]内最长回文子序列长度

递归公式

> if(s[i]==s[j]) dp[i][j]=dp[i+1][j-1]+2;
> else dp[i][j]=max(dp[i+1][j],dp[i][j-1]);

初始化

> for(int i=0;i<s.size();i++)
> dp[i][i]=1;

遍历：从下到上，从左到右

```
class Solution {
public:
    int longestPalindromeSubseq(string s) {
     vector<vector<int>> dp(s.size()+1,vector<int>(s.size()+1,0));
     //初始化
     for(int i=0;i<s.size();i++){
     dp[i][i]=1;
     }
     //遍历
      for(int i=s.size()-2;i>=0;i--){
      for(int j=i+1;j<s.size();j++){
      if(s[i]==s[j])dp[i][j]=dp[i+1][j-1]+2;
      else dp[i][j]=max(dp[i][j-1],dp[i+1][j]);
      }
      }
      return dp[0][s.size()-1];
        }
};
```