让动态规划的题来的更猛烈些吧

今日目标：

1：完成子序列相关面试题

子序列系列-续

1143. 最长公共子序列

字节，京东最近面试题，1143. 最长公共子序列

> 最长公共子序列（Longest Common Subsequence，简称 LCS）的解法是典型的二维动态规划，大部分比较困难的字符串问题都和这个问题一个套路，
>
> **对于两个字符串求子序列的问题，都是用两个指针i和j分别在两个字符串上移动，大概率是动态规划思路。**

算法分析：

1、确定状态参数和选择：

> 原问题：
>
> ```
> 输入：text1 = "abcde", text2 = "ace"
> 输出：3  "ace"
> ```
>
> 子问题：
>
> ```
> 输入：text1 = "abc", text2 = "ac"
> 输出：2  "ac"
> ```

状态参数有两个：`text1` 中 `[1,i]` 个字符，`text2` 中 `[1,j]` 个字符数

选择就很简单，从前到后遍历字符串，增加 `text1,text2` 中的字符，遇到相同的字符就选择作为LCS中的一个字符，遇到不同的就不选。

2、定义dp数组的含义

`dp[i][j]` 的含义是：对于 `text1[1..i]` 和 `text2[1..j]`，它们的 LCS 长度是 `dp[i][j]`。

> 比如：
>
> ```
> text1="ace",text2="babcde"
> ```

		0	1	2	3	4	5	6
	str2 / str1	“	b	a	b	c	d	e
0	“	0	0	0	0	0	0	0
1	a	0	0	1	1	1	1	1
2	c	0	0	1	1	2	2	2
3	e	0	0	1	1	2	2	3

`d[2][4]`的含义就是：对于`"ac"`和`"babc"`，它们的 LCS 长度是 2。

这里要注意的是`i,j`并非指的是字符的下标。

而我们最终想得到的答案应该是`dp[3][6]`。

3、确定初始条件：

`dp[0][....]=0`： `text1`是空串，则LCS肯定是0

`dp[...][0]=0`： `text2`是空串，则LCS也肯定是0

4、状态转移逻辑

状态转移，其实就是做选择，这里做选择就是从前到后遍历字符串，遇到相同的字符就选择作为`lcs`中的字符，遇到不同的字符，要么`text1`增加一个字符,要么`text2`增加一个字符，要么同时增加一个字符。

根据动态规划的思想，我们要思考应该如何从子问题的结果推导到大问题，即**通过前面求解的值如何推导出现在所求的值。**

	1	**2**	**3**	**4**	**5**	**6**
t1[i]	b	**a**	b	**c**	d	**e**
t2[j]	**a**	**c**	z	**e**		
lcs	a	c	e			

这里有四种情况：

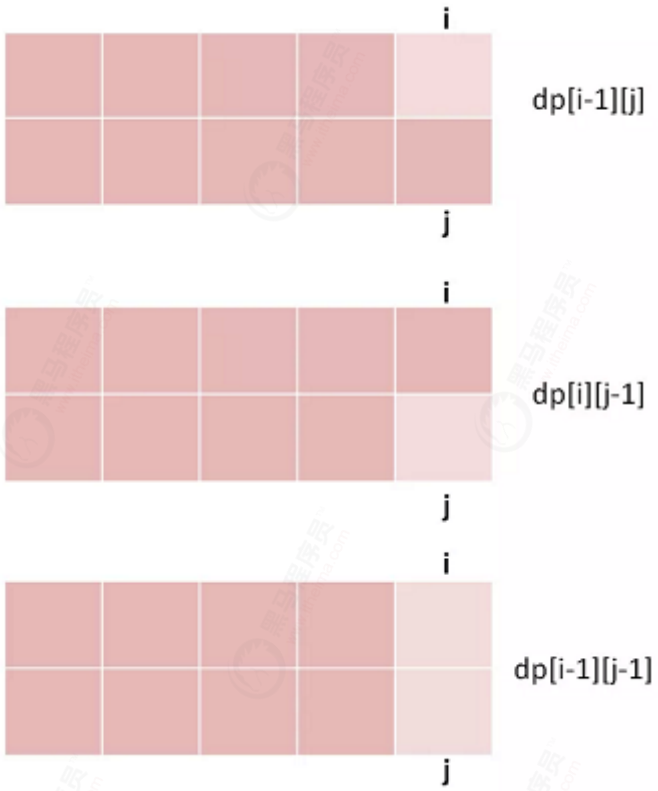
1. t1[i] == t2[j]：该字符一定是lcs中的一个字符，`dp[i][j]=dp[i-1][j-1] +1`
2. t1[i] != t2[j]：又分三种小情况，`dp[i][j]`的值取这三种情况中的最大值，因为最终要返回的是最长的公共子序列

   `dp[i][j]`可由以下三个子问题解推导出来：

- `t1` 增加一个字符，`dp[i][j]=dp[i][j-1]`
- `t2` 增加一个字符，`dp[i][j]=dp[i-1][j]`
- `t1,t2` 均增加一个字符，注意大前提是增加完后 `t1[i] != t2[j]`，因此 `dp[i][j]=dp[i-1][j-1]`

  之：`dp[i][j]` 最终的取值应该是：`d[i][j] =MAX(dp[i-1][j],dp[i-1][j-1],dp[i][j-1])`

但是，其实最终的取值只需要满足：`dp[i][j]=MAX(dp[i-1][j],dp[i][j-1])`，原因如下图



这样一看，显然 `dp[i-1][j-1]` 对应的 `lcs` 长度不可能比前两种情况大，所以没有必要参与比较。我们只需要找谁能让 lcs 最长，就听谁的。

状态转移的逻辑总结如下：

```
if (text1[i] == text2[j]) {
    // 这边找到一个 lcs 的元素，继续往前找
    dp[i][j] = dp[i-1][j-1] + 1;
}else {
    //谁能让 lcs 最长，就听谁的
    //dp[i][j] = Math.max(dp[i-1][j],dp[i-1][j-1],dp[i][j-1]);
    dp[i][j] = Math.max(dp[i-1][j],dp[i][j-1]);
}
```

5、代码实现

```
class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        char[] t1 = text1.toCharArray();
        char[] t2 = text2.toCharArray();
```

```
        //定义dp
        int n = t1.length;
        int m = t2.length;
        int[][] dp = new int[n+1][m+1];
        //初始化状态 dp[0][...]=0,dp[...][0] = 0
        for (int i=0;i<=n;i++) {
            dp[i][0]=0;
        }
        for (int i=0;i<=m;i++) {
            dp[0][i]=0;
        }
        //状态转移
        for (int i=1;i<=n;i++) {
            for (int j=1;j<=m;j++) {
                if (t1[i-1] == t2[j-1]) {
                     // 这边找到一个 lcs 的元素，继续往前找
                    dp[i][j] = dp[i-1][j-1] + 1;
                }else {
                    //谁能让 lcs 最长，就听谁的
                    //dp[i][j] = Math.max(dp[i-1][j],dp[i-1][j-1],dp[i][j-1]);
                    dp[i][j] = Math.max(dp[i-1][j],dp[i][j-1]);
                }
            }
        }
        return dp[n][m];
    }
}
```

> 可自行在dp table中 分析状态转移的过程

> **进阶**：和最长公共子序列相似的几道题目
>
> 583. 两个字符串的删除操作
>
> 712. 两个字符串的最小ASCII删除和

72. 编辑距离

字节，网易，华为，百度，腾讯最近面试题，72. 编辑距离

算法分析：

1、确定状态参数和选择：

> 原问题：

```
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'）
```

子问题：

```
输入：word1 = "hor", word2 = "ro"
输出：2
解释：
hor -> ho（删除 'r'）
ho -> ro（将 'h' 替换为 'r'）
```

状态参数也有两个：`word1`中`[1,i]`的字符，`word2`中`[1,j]`的字符

做决策/选择的过程就是：两个指针`i`和`j`分别在`word1`和`word2`上从左到右遍历（或者从右到左）

> 从左到右遍历是从子问题到大问题；从右到左是从原问题到子问题。

2、定义dp数组的含义

`dp[i][j]`的含义是：将`word1[1...i]`转换成`word2[1...j]`所使用的最少操作步数。

> 比如：

```
输入：word1 = "rad", word2 = "apple"
输出：5
需要注意的是：将`word1`转成成`word2`，和将`word2`转换成`word1`所使用的最少操作步数是一样的。
```

	""	a	p	p	l	e
""	0	1	2	3	4	5
r	1	1	2	3	4	5
a	2	1	2	3	4	5
d	3	2	2	3	4	5

`dp[2][4]`的含义就是将`word1="ra"`转换成`word2="appl"`使用的最少操作步骤为4；

这里要注意的是`i,j`并非指的是字符的下标。

当然对于原问题，我们最终返回`dp[3][5]`即可。

3、确定初始条件：

如果`word1=""`并且`word2=""`，不需要任何操作，即`dp[0][0]=0`

如果`word1=""`但是`word2`不为空串，则最少的操作步数就是`word2`串的长度，即`dp[0][j]=j`

如果`word1`不为空串，但是`word2=""`，则最少的操作步数就是`word1`串的长度，即`dp[i][0]=i`

4、思考状态转移逻辑：

由题设条件，我们能采取三种操作：

- 插入一个字符；
- 删除一个字符；
- 替换一个字符。

题目给定了两个单词，设为 A 和 B，这样我们就能够六种操作方法。

但我们可以发现，如果我们有单词 A 和单词 B：

- 对单词 A 删除一个字符和对单词 B 插入一个字符是等价的。例如当单词 A 为 doge，单词 B 为 dog 时，我们既可以删除单词 A 的最后一个字符 e，得到相同的 dog，也可以在单词 B 末尾添加一个字符 e，得到相同的 doge；
- 同理，对单词 A 插入一个字符和对单词 B 删除一个字符也是等价的；
- 对单词 A 替换一个字符和对单词 B 替换一个字符是等价的。例如当单词 A 为 bat，单词 B 为 cat 时，我们修改单词 A 的第一个字母 b -> c，和修改单词 B 的第一个字母 c -> b 是等价的。

这样以来，本质不同的操作实际上只有三种：

1. 在单词 A 中插入一个字符；
2. 对单词 A 删除一个字符；
3. 修改单词 A 的一个字符。

基于动态规划的思想，我们要考虑如何通过子问题的解来推导大问题的解。

dp[i][j]的含义是：将word1[1...i]转换成word2[1...j]所使用的最少操作步数。

动态规划思想中：根据子问题的解，推导出大问题的解，即如何利用 dp[i][j] 前面的解来推导出 dp[i][j];

case1: 如果 word1[i] == word2[j],则不需要任何操作，接着去比较word1[1 ... i-1]和word2[1 ... j-1]

dp[i][j]的值继承自 dp[i-1][j-1]

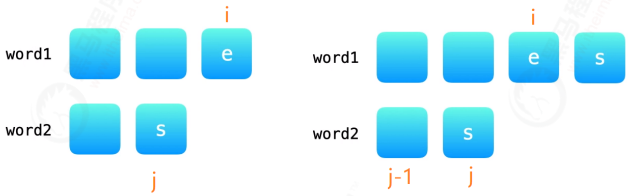
dp[i][j] = dp[i-1][j-1]

case2：如果word1[i] != word2[j]，据题意，我们有如下三种操作，最终我们要的是能让整体步骤最少的

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

1：执行插入操作，然后比较word1[1 ... i]和word2[1 ... j-1]的结果



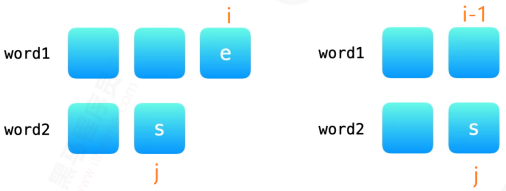
在i后面插入word2[j]这个字符，这样word1和word2中的该字符就可以匹配跳过了，
接下来比较word1[1 ... i]和word2[1 ... j-1]即可

子问题dp[i][j-1]的值：word1[1 ... i]和word2[1 ... j-1] 编辑的最小距离

那么有了dp[i][j-1]的值后，对于word2的第 j 个字符，我们只需在word1的末尾添加一个相同字符，这个距离就算编辑完成了。

那么dp[i][j]的最小值可以为 dp[i][j-1] +1

2：执行删除操作，然后比较word1[1 ... i-1] 和 word2[1 ... j]的结果



删除word1[i]这个字符，然后去接着去比较word1[1 ... i-1] 和word2[1 ... j]即可

子问题dp[i-1][j]的值：word1[1 ... i-1] 和 word2[1 ... j]编辑的最小距离

有了dp[i-1][j]的值之后，对于word1的第i个字符，我们只要删除该字符就算编辑完成了。

那么dp[i][j]的最小值可以是：dp[i-1][j] +1

3：执行替换操作，然后接着比较 word1[1 ... i-1] 和 word2[1 ... j-1]

用word2[j] 替换 word1[i]，这样匹配之后只需接着去比较
word1[1 ... i-1] 和 word2[1 ... j-1]

子问题 dp[i-1][j-1]的值：word1[1 ... i-1] 和 word2[1 ... j-1]编辑的最小距离

有了dp[i-1][j-1]的值之后，对于word1中的第i个字符，word2中的第j个字符，我们可以采用替换操作，将其中一个替换成另一个，这样也就编辑完成了

那么 dp[i][j] 的最小值可以为：dp[i-1][j-1] + 1

最终：在这种情况下，dp[i][j]有三种可能的取值，而我们要取的是三种可能中的最小值
即：dp[i][j] = min (dp[i][j-1] +1,dp[i-1][j] + 1,dp[i-1][j-1] +1)

> 当我们获得 `dp[i][j-1]`，`dp[i-1][j]` 和 dp`[i-1][j-1]` 的值之后就可以计算出 `D[i][j]`。
>
> - `dp[i][j-1]` 为 A 的前 i 个字符和 B 的前 j - 1 个字符编辑距离的子问题。即对于 B 的第 j 个字符，我们在 A 的末尾添加了一个相同的字符，那么 `dp[i][j]` 最小可以为 `dp[i][j-1] + 1`；
> - `dp[i-1][j]` 为 A 的前 i - 1 个字符和 B 的前 j 个字符编辑距离的子问题。即对于 A 的第 i 个字符，我们在 B 的末尾添加了一个相同的字符，那么 `dp[i][j]` 最小可以为 `dp[i-`

1][j] + 1;

- dp[i-1][j-1]为 A 前 i - 1 个字符和 B 的前 j - 1 个字符编辑距离的子问题。即对于 B 的第 j 个字符，我们修改 A 的第 i 个字符使它们相同，那么 dp[i][j]最小可以为 dp[i-1][j-1] + 1。特别地，如果 A 的第 i 个字符和 B 的第 j 个字符原本就相同，那么我们实际上不需要进行修改操作。在这种情况下，dp[i][j]最小可以为 dp[i-1][j-1]。

翻译过来的转移逻辑是：

```
if (word1[i] == word2[j]) {
    //不用操作，继承之前的结果
    dp[i][j] = dp[i-1][j-1];
}else {
    //找三种操作中能让编辑距离最小的
    dp[i][j] = min(dp[i-1][j] +1,
                   dp[i][j-1] +1,
                   dp[i-1][j-1] +1);
}
```

5、代码实现

```
class Solution {
    //dp table
    public int minDistance(String word1, String word2) {
        char[] w1 = word1.toCharArray();
        char[] w2 = word2.toCharArray();
        int n = w1.length;
        int m = w2.length;
        //定义dp数组
        int[][] dp = new int[n+1][m+1];
        //初始条件 dp[0][j] = j,dp[i][0] = i;
        for (int i=0;i<=n;i++) {
            dp[i][0] = i;
        }
        for (int j=0;j<=m;j++) {
            dp[0][j] = j;
        }
        //状态转移
        for (int i=1;i<=n;i++) {
            for (int j=1;j<=m;j++) {
                if (w1[i-1] == w2[j-1]) {
                    //唯一的一种操作,跳过(i,j),dp[i][j]继承自dp[i-1][j-1]
                    dp[i][j] = dp[i-1][j-1];
                }else {
                    /*
                        删除操作：dp[i][j] = dp[i-1][j] +1
                        插入操作：dp[i][j] = dp[i][j-1] +1;
                        替换操作：dp[i][j] = dp[i-1][j-1] +1;
                    */
                    dp[i][j] = min(dp[i-1][j] +1,
                                   dp[i][j-1] +1,
                                   dp[i-1][j-1] +1);
                }
            }
```

```
        }
        return dp[n][m];
    }

    public int min(int x,int y,int z) {
        return Math.min(Math.min(x,y),z);
    }
}
```

5. 最长回文子串

字节，华为，腾讯最近高频面试题，5. 最长回文子串

算法分析：

1、确定状态参数和选择

> 原问题：
>
> ```
> 输入："babad"
> 输出："bab"
> ```
>
> 子问题：
>
> ```
> 输入："aba"
> 输出："aba"
> ```
>
> 这里因为对于输入的字符串`s="cbbd"`它的任意一个字串都是它的子问题，比如找`cbb`，`bbd`它们的最长回文子串。

这道题也是一道经典的区间dp问题。因此区间的起始位置和结束位置就是在原问题和子问题之间发生变化的变量，故：

状态参数有两个：字符串的起始位置`i`和结束位置`j`，这里的`i`和`j`代表的就是下标了。

选择也很简单：改变区间的大小，

2、定义dp数组的含义

`dp[i][j]`的含义是：对于字串`s[i...j]`是否是回文串，为`true`表明`s[i...j]`是回文串，否则不是。

> 要注意的是：
>
> 1：这里`dp`数组中存储的值不能直接作为题目返回的结果，以前我们说大部分情况下，dp数组中存储的就是我们想要的返回结果，在这里要做一个变形，最终返回的最长回文子串是什么呢？
>
> `i`和`j`就是字串的左右边界，我们要返回的是所有`dp[i][j]==true`的选项中，最长的那个字串。

		0	1	2	3
		c	b	b	d
0	c	T	F	F	F
1	b	F	T	T	F
2	b	F	F	T	F
3	d	F	F	F	T

`dp[1][2]`的含义就是：子串`bb`是回文子串，`dp[0][2]`表明子串`cbb`不是回文子串

当然在此处的`i`和`j`代表的就是下标了。

而我们最终要返回的是：所有`dp[i][j]==true`的子串中，长度最长的子串。

3、确定初始状态

因为`i`和`j`代表了子串的左右边界，满足`i<=j`，所以初始状态为：当`i==j`，只有一个字符，肯定是回文串，即当`i==j`时：`dp[i][j]=true`

另外当`i>j`时不符合我们字串的定义，所以`dp[i][j]=false`。

4、状态转移逻辑

状态转移，**需要数学归纳法思维，说白了就是如何从已知的结果推出未知的部分，如何根据子问题的解推导原问题的解**，这样定义容易归纳，容易发现状态转移关系。

在这里原问题是求：`dp[i][j]`，子问题是`dp[i+1][j-1]`甚至更小的子问题。

也就是说如果我们想求`dp[i][j]`，假设已经求出来子问题`dp[i+1][j-1]`的结果（`s[i+1..j-1]`是回文字串，或者不是回文字串）或者更小的子问题的结果，你是否能想办法算出`dp[i][j]`的值（`s[i..j]`是否是回文字串）

为了讨论一个字串是否是回文串，我们从回文串的定义出发：

1. 如果一个字符串的头尾两个字符都不相等，那么这个字符串一定不是回文串；
2. 如果一个字符串的头尾两个字符相等，才有必要继续判断下去。

- 如果里面的子串是回文，整体就是回文串；
- 如果里面的子串不是回文串，整体就不是回文串。

翻译出来的逻辑就是

```
if (s[i] != s[j]) {
    //字符串的头尾两个字符都不相等，那么一定不是回文串
    dp[i][j] = false;
}else {
    //字符串的头尾两个字符相等,它是否是回文串取决于它的字串s[i+1...j-1]
    dp[i][j] = dp[i+1][j-1];
}
```

需要注意：

当 `s[i]=s[j]` 时我们说 `s[i...j]` 是否是回文串取决于它的子串 `s[i+1...j-1]` 是否是回文串，这样当这个字串区间不断缩小时，会遇到的边界情况就是：`[i+1,j-1]` 并不能构成区间，即这个区间的长度小于2，即：

`(j-1) - (i+1)<2` 推导得到：`j-i<3`，等价于子串 `s[i...j]` 的长度等于2或者等于3。

> 举个例子：现在子串 `s[i...j]` 为 `aba` 或者为 `aa`，满足 `s[i]==s[j]`，此时 `s[i+1...j-1]` 为 `"b"` 或者是空串 `""`

显然：

- 如果子串 s[i + 1... j - 1] 只有 1 个字符，即去掉两头，剩下中间部分只有 1 个字符，显然是回文；
- 如果子串 s[i + 1... j - 1] 为空串，那么子串 s[i, j] 一定是回文子串。

因此状态转移的逻辑修改如下：

```
if (s[i] != s[j]) {
    //字符串的头尾两个字符都不相等，那么一定不是回文串
    dp[i][j] = false;
}else {
    //字符串的头尾两个字符相等,它是否是回文串取决于它的字串s[i+1...j-1]
    if (j-i<3) {
        dp[i][j] = true;//s[i+1...j-1]为边界情况，只有一个字符或者是空串。
    }else {
        dp[i][j] = dp[i+1][j-1];
    }
}
```

5、代码实现

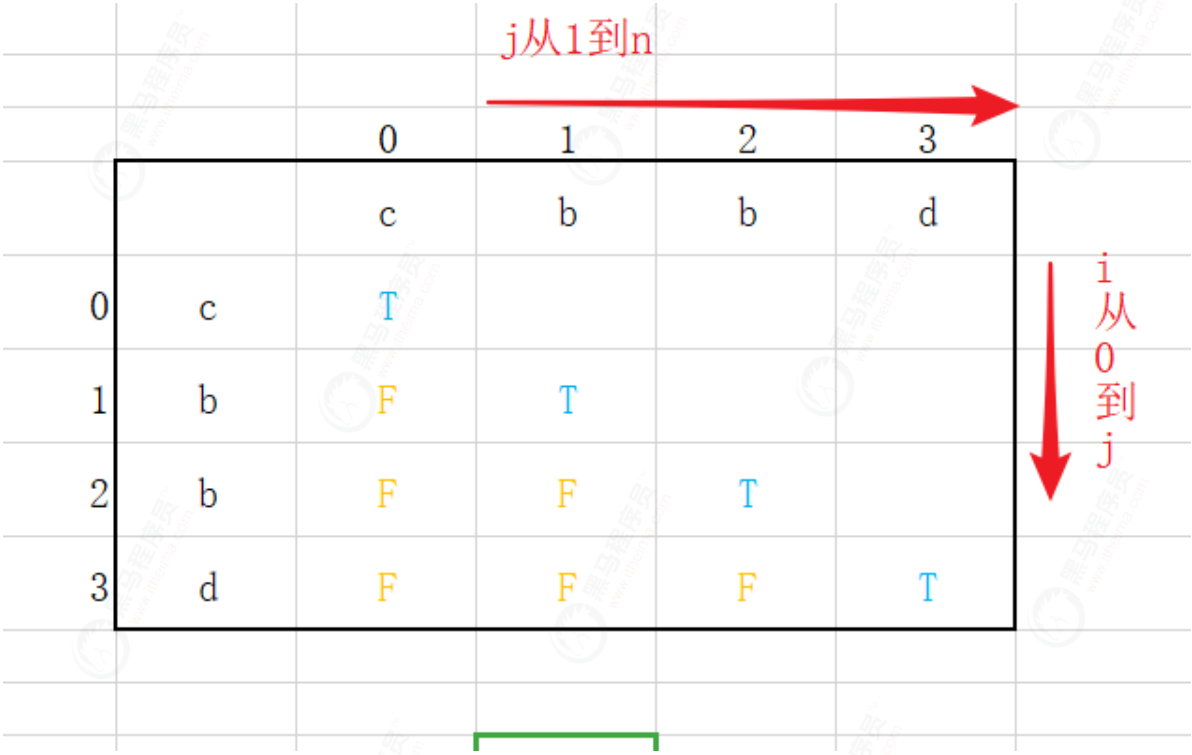
```
class Solution {
    public String longestPalindrome(String s) {
        //特殊判断
        if (s==null || s.length() < 2) {
            return s;
        }
        char[] strs = s.toCharArray();
        int n = strs.length;

        /*
            定义dp数组:dp[i][j]代表字串s[i ... j]是否是一个回文串
            最终返回的在所有dp[i][j]为true的组合中,(j-i+1)值最大的那个
        */
        boolean[][] dp = new boolean[n][n];
```

```
15
16          //确定初始化状态 dp[i][i] = true,当i>j时dp[i][j] = false;
17          for (int i=0;i<n;i++) {
18              dp[i][i] = true;
19          }
20          //定义最大回文字串的开始位置和长度
21          int begin=0;
22          int maxLen = 1;
23
24          //状态转移
25          for (int j=1;j<n;j++) {
26              for (int i=0;i<j;i++) {
27
28                  //如果s[i] != s[j]那么 s[i ... j]肯定不是回文串
29                  if (strs[i] != strs[j]){
30                      dp[i][j] = false;
31                  }else {
32                      /*
33                          如果s[i] == s[j],此时还不能直接下结论s[i ... j]是回文串，还得看它的子串s[i+1 ... j-1]是不是回文串，
34                          如果s[i+1 ... j-1]是那么s[i ... j]肯定也是
35                      */
36                      if (j-i<3) {
37                          dp[i][j] = true;
38                      }else {
39                          dp[i][j] = dp[i+1][j-1];
40                      }
41                      //dp[i][j] = dp[i+1][j-1];
42                  }
43
44                  // 只要 dp[i][j] == true ，就表示子串 s[i..j] 是回文，此时记录最大回文长度和起始位置
45                  if (dp[i][j] && (j-i+1) > maxLen) {
46                      maxLen = j-i+1;
47                      begin = i;
48                  }
49              }
50          }
51
52
53          return s.substring(begin,begin+maxLen);
54      }
55  }
```

需要注意：

1、根据状态转移方程，`dp[i][j]`是由`dp[i+1][j-1]`转移而来。为了保证每次计算`dp[i][j]`时`dp[i+1][j-1]`的值都被计算出来了，我们采用的遍历方式需要注意，如下图



> **进阶：**
>
> 1、该题还有一个更优的解法：中心扩散法，也是一个典型的双指针类算法思想。
>
> 2、还有一个Manacher 算法，不用掌握。

516. 最长回文子序列

字节，微软，快手最近面试题，516. 最长回文子序列

算法分析：

1、确定状态参数和选择：

> 原问题：
>
> ```
> 输入:"dcbbd"
> 输出:4
> 解释：一个可能的最长回文子序列为 "dbbd"。
> ```
>
> 子问题：
>
> ```
> 输入:"cbb"
> 输出:2
> 解释：一个可能的最长回文子序列为 "bb"。
> ```
>
> 这里因为对于输入的字符串`s="dcbbd"`它的任意一个字串都是它的子问题，比如找`dcb`，`cbbd`它们的最长回文子序列。

这道题是一道经典的区间dp问题。因此区间的起始位置和结束位置就是在原问题和子问题之间发生变化的变量，故：

状态参数有两个：字符串的起始位置`i`和结束位置`j`，这里的`i`和`j`代表的就是下标了。

选择也很简单：改变区间的大小，

> 改变区间共三种情况：
>
> 两边都扩大，`i`和`j`分别向两边扩张
>
> `i`向左边扩张
>
> `j`向右边扩张

2、定义dp数组的含义：

在子串s[i..j]中，最长回文子序列的长度为`dp[i][j]`。

> 要注意的是：
>
> 1、`s[i...j]中最长回文子序列跟回文串还不一样`，回文串中，`dp[i][j]`代表的是`s[i...j]`是否是回文串，而最长回文子序列`dp[i][j]`代表的是在子串`s[i...j]`中的回文子序列，这个子序列有可能就是子串`s[i...j]`，也可能是`s[i...j]`中的一部分或某几部分，因此`dp[i][j]`没法单独代表字串`s[i...j]`是否是回文子序列。
>
> 举个例子：

```
输入："bbbab"
输出：4
解释：最长的回文子序列是："bbbb"
```

			j				
			0	1	2	3	4
			b	b	b	a	b
	0	b	1	2	3	3	4
	1	b	0	1	2	2	3
i	2	b	0	0	1	1	3
	3	a	0	0	0	1	1
	4	b	0	0	0	0	1

> `dp[0][3]`的含义就是：`bbba`的最长回文子序列长度为：3
>
> 当然在此处的`i`和`j`代表的就是下标了。
>
> 而我们最终要返回的是：`dp[0][4]`，也就是`dp[0][n-1]`，n是输入字符串的长度。

3、确定初始状态：

因为`i`和`j`代表了子串的左右边界，满足`i<=j`，所以初始状态为：当`i==j`，只有一个字符，最长回文子序列肯定是1，即当`i==j`时：`dp[i][j]=1`

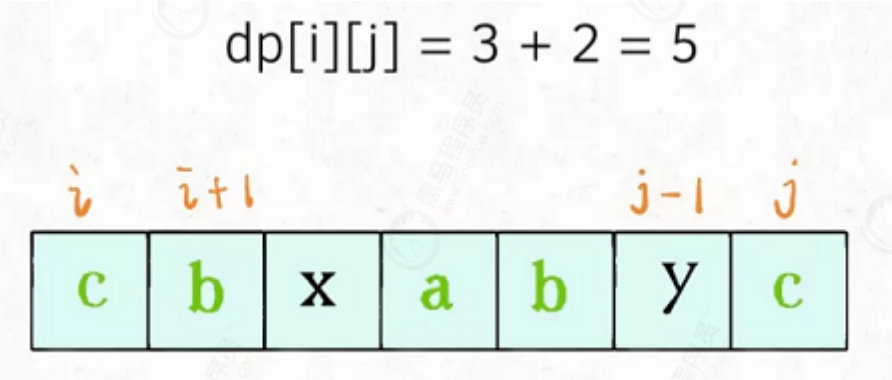
另外当`i>j`时不符合我们字串的定义，所以`dp[i][j]=0`。

4、状态转移逻辑

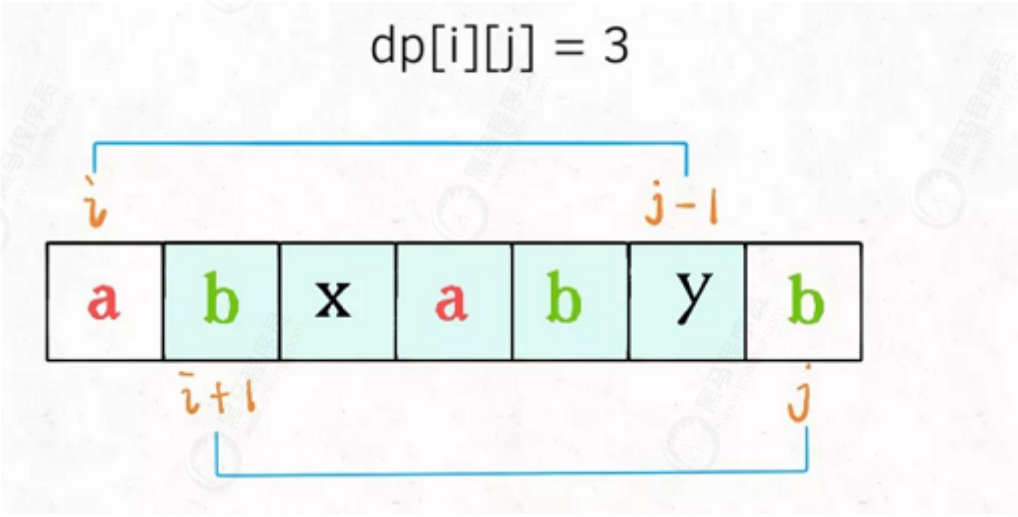
> 状态转移，**需要数学归纳法思维，说白了就是如何从已知的结果推出未知的部分，如何根据子问题的解推导原问题的解**，这样定义容易归纳，容易发现状态转移关系。
>
> 在这里原问题是求：`dp[i][j]`，子问题是`dp[i+1][j-1]`甚至更小的子问题。
>
> 也就是说如果我们想求`dp[i][j]`，假设已经求出来子问题`dp[i+1][j-1]`的结果（`s[i+1..j-1]`中最长回文子序列的长度）或者更小的子问题的结果，你是否能想办法算出`dp[i][j]`的值（`s[i..j]`中，最长回文子序列的长度）

假设已知`dp[i+1][j-1]`或更小的子问题的值，求`dp[i][j]`，这里根据`s[i]`和`s[j]`两个字符是否相同分成了以下三种情况：

如果：`s[i]==s[j]`，那么这两个字符加上`dp[i+1][j-1]`的值就是`dp[i][j]`的值。即这两个字符加上`s[ i+1 ... j-1]`中的最长回文子序列就是`s[i ... j]`中的最长回文子序列



如果：`s[i] != s[j]`，那么说明它俩**不可能同时**出现在`s[i...j]`的最长回文子序列中，那么只需把它俩**分别**加入`s[i+1..j-1]`中，看看哪个子串产生的回文子序列更长即可：



> 翻译过来的转移逻辑：

```
if (s[i] == s[j])
// 它俩一定在最长回文子序列中
dp[i][j] = dp[i + 1][j - 1] + 2;
else
//看 s[i+1..j] 和 s[i..j-1] 谁的回文子序列更长？
dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
```

5、代码实现

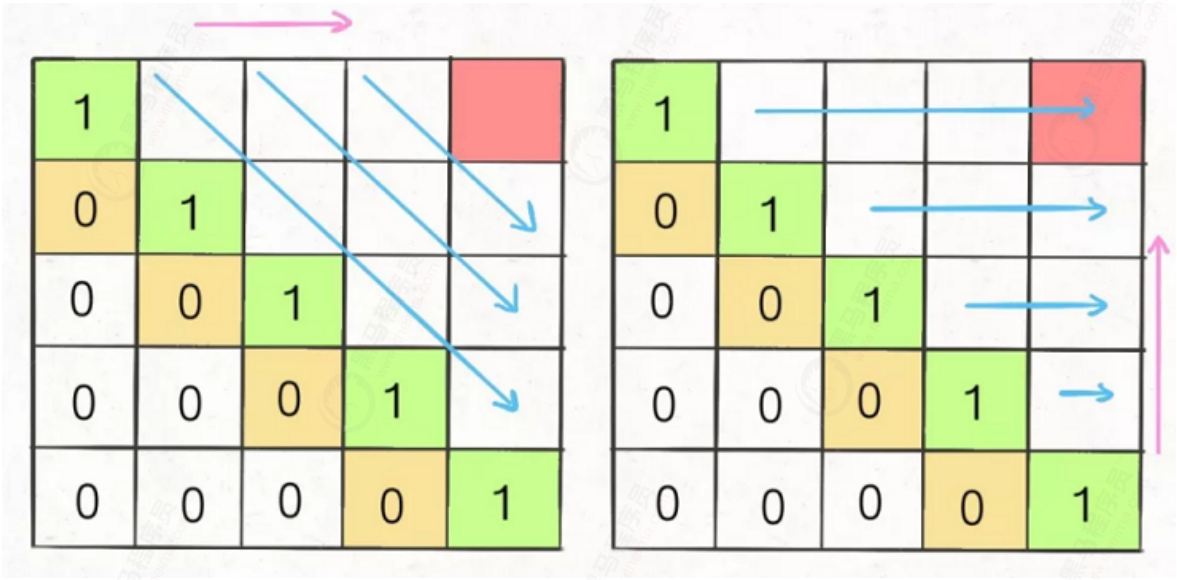
```
class Solution {
    public int longestPalindromeSubseq(String s) {
        char[] cr = s.toCharArray();
        //获取字符串长度
        int n = s.length();
        //定义dp数组
        int[][] dp = new int[n][n];
        //初始化条件,当i==j时,dp[i][j] =1;当i>j的所有选项dp[i][j]=0;
        for (int i=0;i<n;i++) {
            dp[i][i] = 1;
        }
        //状态转移,这里dp[i][j]的值每次是由其左边，下边，左下边三个位置计算出来的，为了保证这一点，只能斜着遍历或者反着遍历，我们选择反着遍历
        for (int i=n-1;i>=0;i--) {
            for (int j=i+1;j<n;j++) {
                if (cr[i] == cr[j]) {
                    dp[i][j] = dp[i+1][j-1] +2;
                }else {
                    dp[i][j] = Math.max(dp[i+1][j],dp[i][j-1]);
                }
            }
        }

        return dp[0][n-1];
    }
}
```

需要注意两点：

1：初始条件里面，对于那些 `i > j` 的位置，根本不存在什么子序列，应该初始化为 0。

2：根据状态转移方程，想求 `dp[i][j]` 需要知道 `dp[i+1][j-1]`，`dp[i+1][j]`，`dp[i][j-1]` 这三个位置的值，**为了保证每次计算 `dp[i][j]`，左、下、左下三个方向的位置已经被计算出来，只能斜着遍历或者反着遍历**：



6、总结

找到初始化条件和状态转移逻辑之后，**一定要观察 DP table**，看看怎么遍历才能保证通过已计算出来的结果解决新的问题。