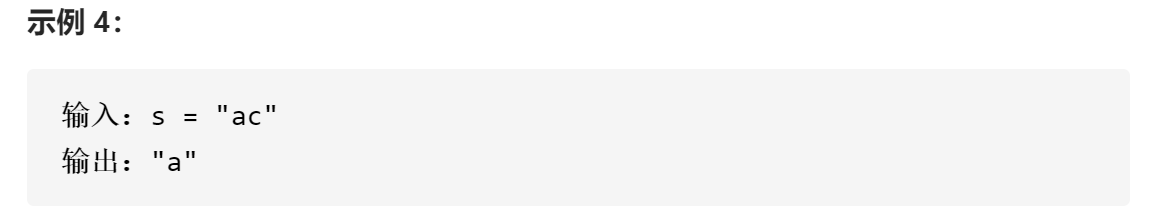
# #5.最长回文子串





## 题目分析:我的分析

**一维动态数组：dp[i]表示以s[i]为结尾的“前缀子串”中以s[i]结尾的的最长回文子串**

**状态转移方程：dp[i]=d[i-1]+2当且仅当s[i]=s[i-dp[i-1]+1];也就是说加入该字符后，子串中有相应的字符与之匹配。**

### 不成熟的动态规划

这个动态规划不是很成熟！

**当不满足s[i]=s[i-dp[i-1]+1]时，需要从前向后遍历来确立dp[i]的值，非常的麻烦！**

### 为什么不成熟？

虽然这样分析也把大问题分解为了一个个小问题，但是这个小问题与大问题之间的联系不是那么的紧密，也就是说动态的转移方程并没有覆盖所有的情况！导致在覆盖不到的情况下不得不去遍历（相当于暴力求解了）；

**可能一开始的思维就有局限性，由于线性的思考方式，总想着从左向右遍历，然后从左向右遍历的时候如何将这个问题分解为一个个小问题，但是这样线性的思考方式导致了小问题与大问题之间的联系不是很紧密！**

### 代码实现

class Solution {  
public:  
 string longestPalindrome(string s) {  
//动态规划  
vector<int>dp(s.size(),1);//初始化  
  
//遍历+动态规划  
for(int i=1;i<s.size();i++){  
 if(dp[i-1]!=1){  
 if(i-dp[i-1]-1>=0&&s[i-dp[i-1]-1]==s[i]){  
 dp[i]=dp[i-1]+2;  
 continue;  
 }  
 }  
  
 for(int j=0;j<i;j++){  
 if(s[j]==s[i]){  
 int temp1=j,temp2=i,count=0;  
 while(temp1<temp2){  
 if(s[temp1]==s[temp2]){  
 temp1++;  
 temp2--;  
 count+=2;  
 }  
 else {  
 count=0;  
 break;  
 }  
 }  
 if(count){  
 if(temp1==temp2){  
 count+=1;  
 dp[i]=count;  
 break;  
 }else{  
 dp[i]=count;  
 break;  
 }  
 }  
 }  
 }  
}  
  
//结果  
string res;  
int tag=-1,max=-1;  
for(int i=0;i<s.size();i++){  
 if(dp[i]>max){  
 max=dp[i];  
 tag=i;  
 }  
}  
  
for(int i=tag-dp[tag]+1;i<=tag;i++){  
 res.push\_back(s[i]);  
}  
  
return res;  
 }  
};

## 高阶动态规划

**从最后所得结果的最长的回文串的角度来分析：**

**假设二维动态数组dp[i] [j]表示下标范围从 i 到 j 的子串是否是是回文串；**

**状态转移方程：dp[i] [j]=dp[i-1] [j+1] ^s[i-1]==s[j+1];**

1. **二维动态数组的初始化：dp[i] [i]=true;&&dp[i] [i+1]=true （s[i]==s[i+1]）;**
2. **特殊情况处理dp[i] [j]=false(i>j);**

### 高阶代码

class Solution {  
public:  
 string longestPalindrome(string s) {  
 int n = s.size();  
 vector<vector<int>> dp(n, vector<int>(n));//动态数组的设置  
 string ans;  
 for (int l = 0; l < n; ++l) {  
 for (int i = 0; i + l < n; ++i) {  
 int j = i + l;  
 if (l == 0) {//动态数组的初始化dp[i][i]  
 dp[i][j] = 1;  
 } else if (l == 1) {//动态数组的特殊情况处理dp[i][i+1]=dp[i][i](s[i]==s[j]);  
 dp[i][j] = (s[i] == s[j]);  
 } else {  
 dp[i][j] = (s[i] == s[j] && dp[i + 1][j - 1]);  
 }  
 if (dp[i][j] && l + 1 > ans.size()) {  
 ans = s.substr(i, l + 1);//见注释  
   
 }  
 }  
 }  
 return ans;  
 }  
};  
  
作者：LeetCode-Solution  
链接：https://leetcode-cn.com/problems/longest-palindromic-substring/solution/zui-chang-hui-wen-zi-chuan-by-leetcode-solution/  
来源：力扣（LeetCode）  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

#### substr的简介：

