5. 最长回文子串

难度：中等

给定一个字符串 `s`，找到 `s` 中最长的回文子串。你可以假设 `s` 的最大长度为 1000。

示例 1：

```
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。
```

示例 2：

```
输入: "cbbd"
输出: "bb"
```

思路1：暴力法

从头开始遍历，按如下方式寻找以当前字母α_i为开头的最长回文子串：

- 从字符串尾开始，从后往前遍历字符串中的字母α_j，$i < j$，初始化最大字符串长度为0
- 判断α_i到α_j所构成的子串是否构成回文字符串，如果不是，继续遍历
- 如果是，跳出当前从后往前的遍历。
- 比较当前子串长度与最大字符串长度来更新最大字符串长度，并记录所对应的字符串

当当前最大字符串长度大于等于当前遍历的字母到结尾的长度时，跳出循环，并返回最大字符串长度所对应的字符串

时间复杂度：$O(n^3)$，空间复杂度：$O(1)$

结果：**超时**

代码（两个for循环）：

```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        def isPalindrome(s: str) -> bool:
            left = 0
            right = len(s)-1
            while left < right:
                if s[left] != s[right]:
                    return False
                left += 1
                right -= 1
            return True

        max_len = 0
        for i in range(len(s)):
            if max_len >= len(s) - i:
                break
            for j in reversed(range(len(s))):
                if isPalindrome(s[i:j + 1]):
                    if max_len < j - i + 1:
```

```
                    max_len = j - i + 1
                    res = s[i:j + 1]
                break
        return res
```

思路2：中心扩展算法

中心扩展算法其实跟上面暴力法思路是一样的，它也是先从前往后遍历字符串中的字母，不同的是，它并不是以当前字母为开头来寻找最长的回文子串，而是向两边扩展来寻找最长的回文子串。这样寻找的优点在于，它扩展得到的子串就是回文子串而不需要在进行判断。不过这里的扩展有两种情况：

1. 以当前遍历的字母α_i为中心向两边扩展，也就是不停地判断α_{i-j}和α_{i+j}是否相等（$j=1,2,3...$），如果不相等，则停止扩展，反之相等的话则继续扩展，直到找到以当前字母为中心的最长回文子串。这种情况对应的是回文子串长度是奇数情况。
2. 以当前遍历的字母α_i和α_{i+1}为中心向两边扩展，及不停地判断α_{i-j}和α_{i+1+j}是否相等（$i=0,1,2,3...$），如果不相等，则停止扩展，反之相等的话则继续扩展，直到找到以字母α_n和α_{n+1}为中心的最长回文子串。这种情况对应的是回文子串长度是偶数情况。

至于上面还有一个情况是以α_{i-1}和α_i为中心扩展，由于我们是从前往后遍历字符串，对于α_{i-1}和α_i为中心扩展的情况对应的就是在遍历α_{i-1}时上面的第二种情况，所以只需要考虑上面两种扩展情况即可。

从前往后遍历过程中，当发现当前最长回文子串的长度的一半（也就是臂长，下面会讲到）大于等于当前遍历字母到结尾的长度时，停止遍历，输出前面得到的最长回文子串。

时间复杂度：$O(n^2)$，空间复杂度：$O(1)$

代码

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        """
        遍历字符串，从当前字符串往两边扩展，找到最长的字符串并记录
        方法：定义一个扩展函数，从当前遍历到的字母处开始，1种是以当前字母为中心，向两边扩展，
        另外一种是以当前字母后后一个字母整体为中心向两边扩展，如果发现扩展的两端字母不等
        则停止。两种扩展方法分别对应奇数长度和偶数长度回文字符串。
          函数返回当前字母两种扩展最长的回文子串长度和对应的回文子串
        时间复杂度：O(n^2) 空间复杂度： O(1)
        """
        def expand(idx: int, s: str):
            len1 = 1
            left, right = idx - 1, idx + 1
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
                len1 += 2
            len2 = 0
            left, right = idx, idx + 1
            while left >= 0 and right < len(s) and s[left] == s[right]:
                left -= 1
                right += 1
                len2 += 2
            if len1 >= len2:
                return [len1, s[idx - len1 // 2:idx + len1 // 2 + 1]]
            else:
                return [len2, s[idx - len2 // 2 + 1:idx + len2 // 2 + 1]]

        max_len = 0
```

```
        for i in range(len(s)):
            if max_len // 2 >= len(s) - i:
                break
            l, ch = expand(i, s)
            if max_len < l:
                max_len = l
                res = ch
        return res
```

leetcode官方题解上代码更为简洁，也是同样思路，只不过在扩展时并没有存储得到的最长回文子串的长度，而是记录扩展得到的最左和最右两个字母的index

```
class Solution:
    def expandAroundCenter(self, s, left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return left + 1, right - 1

    def longestPalindrome(self, s: str) -> str:
        start, end = 0, 0
        for i in range(len(s)):
            left1, right1 = self.expandAroundCenter(s, i, i)
            left2, right2 = self.expandAroundCenter(s, i, i + 1)
            if right1 - left1 > end - start:
                start, end = left1, right1
            if right2 - left2 > end - start:
                start, end = left2, right2
        return s[start: end + 1]
```

思路3：动态规划

leetcode上面的官方题解

我们用$P(i,j)$ 表示字符串 s 的第 i 到 j 个字母组成的串是否为回文串：

$$P(i,j)=\begin{cases}\text{true}, & \text{如果子串} s[i]\dots s[j] \text{是回文串}\\ \text{false}, & \text{其它情况}\end{cases}$$

那么我们可以写出状态转移方程：

$$P(i,j) \ :- P(i+1,j-1) \wedge (s[i] == s[j])$$

明显发现，上面的状态转移方程只适用于j-i>=3这种情况，那么我们还要考虑小于3的情况：

$$\begin{aligned} P(i,i) &= True \\ P(i,i+1) &:- (s[i] == s[i+1]) P(i,i) = True \\ P(i,i+1) &:- (s[i] == s[i+1]) \end{aligned}$$

时间复杂度：$O(n^2)$，空间复杂度：$O(n^2)$

结果：**超时**

代码

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        len_s = len(s)
        dp = [[False] * len_s for _ in range(len_s)]
        ans = ""
        #子序列从长度为0开始遍历
        for len_subseq in range(len_s):
            #子序列首字母i
            for i in range(len_s):
                #子序列尾字母j
                j = i+len_subseq
                if j >= len_s:
                    break
                if len_subseq == 0:
                    dp[i][j] = True
                elif len_subseq == 1:
                    dp[i][j] = (s[i] == s[j])
                else:
                    dp[i][j] = (dp[i+1][j-1]) & (s[i] == s[j])
                if dp[i][j] & (len_subseq+1>len(ans)):
                    ans = s[i:j+1]
        return ans
```

动态规划+分类讨论

这里我们用dp存储以每个字母结尾的最长回文子串的长度，也就是说dp[i]表示以s[i]结尾的最长回文子串的长度。

如果：$i-dp[i-1]-1>=0$且$s[i]==s[i-dp[i-1]-1]$

- $dp[i]=dp[i-1]+2$
- 因为已知以s_{i-1}为结尾时的最长回文子串dp[i-1]，则$s[i-dp[i-1]-1]$对应的是这个回文子串前一个字母

否则：

- 从$i-dp[i-1]$字母处从左往右遍历，看是否能形成回文子串

之所以从$i-dp[i-1]$处开始遍历，是因为不可能有更长的回文子串，以下给出证明。

令这么一段字符串$x_0,x_1,\dots,x_k,x_{k+1}\dots,x_{j-1},x_j,\dots,x_{i-1},x_i,\dots,x_n$现在我们遍历到了第i+1个元素处，即$x_i$，且对于以$x_{i-1}$结尾的最长回文子串为$x_j,\dots,x_{i-1}$

我们的推断是，当$x_{j-1}\neq x_i$时，以x_i结尾的最长回文子串必为$x_j,\dots,x_i$中以x_i结尾的最长回文子串，接下来我们用反证法

假设在x_j之前还存在一个x_k（$k<j-1$），使得$x_k,\dots,x_i$构成回文子串，那么有$x_k=x_i,x_{k+1}=x_{i-1},\dots$，那么我们自然也就得到$x_{k+1},\dots,x_{i-1}$也能构成回文子串，矛盾

得证

时间复杂度：$O(n^2)$，空间复杂度：$O(n)$

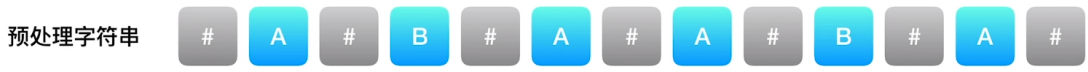
代码

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        dp = [1]
        max_len = 1
        end_index = 0
        for i in range(1, len(s)):
            if (i - dp[i - 1] - 1 >= 0) & (s[i] == s[i - dp[i - 1] - 1]):
                dp.append(dp[i - 1] + 2)
            else:
                #从i-dp[i-1]个字母处从左往右遍历
                j = i - dp[i - 1]
                while j<=i:
                    if self.isPalindrome(s[j:i+1]):
                        dp.append(i-j+1)
                        break
                    j += 1
            if dp[i] > max_len:
                max_len = dp[i]
                end_index = i
        return s[end_index - max_len + 1: end_index + 1]
    #判断是否为回文子串
    def isPalindrome(self, s: str) -> bool:
        i = 0
        j = len(s) - 1
        while i < j:
            if s[i] != s[j]:
                return False
            i += 1
            j -= 1
        return True
```

思路4：Manacher 算法

预处理：扩展字符串，使其最长回文子串的长度始终为奇数

- 为了将奇、偶数回文串的性质统一表示，将原始字符串进行预处理，用不在输入字符串中的字符隔开



下标 i	0	1	2	3	4	5	6	7	8	9	10	11	12
以 i 为中心的最长回文子串的长度	1	3	1	7	1	3	13	3	1	7	1	3	1
对应原始字符串中的最长回文子串的长度	0	1	0	3	0	1	6	1	0	3	0	1	0

算法分析

动态规划与中心扩展相结合：

定义一个数组p, p[i]表示以字符串中第i个字符向左右两边扩展的最大回文子串长度

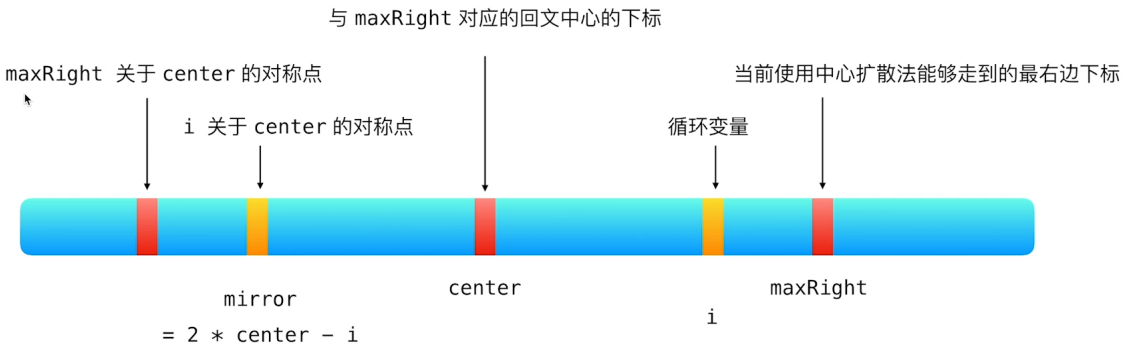
定义两个变量：

- maxRight：记录前面遍历的回文子串能够触及到的最右边字母的下标

- center：上面maxRight对应的回文子串的中心字母下标

下面我们开始对字符串从前往后进行遍历，并不断更新p，更新时候有以下情况：

1. i >= maxRight，只能使用中心扩展方法
2. i < maxRight，需要进行分类讨论：
    - 当p[mirror] < maxRight-i 时，有p[i] = p[mirror] < maxRight-i。
    - 当p[mirror] == maxRight-i 时，有p[i] >= maxRight-i，所以需要从maxRight处开始进行中心扩展方法。
    - 当p[mirror] > maxRight-i 时，有p[i] = maxRight-i。



代码

```
class Solution:
    def expand(self, s, left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return (right - left - 2) // 2

    def longestPalindrome(self, s: str) -> str:
        end, start = -1, 0
        s = '#' + '#'.join(list(s)) + '#'
        arm_len = []
        right = -1
        j = -1
        for i in range(len(s)):
            if right >= i:
                i_sym = 2 * j - i
                min_arm_len = min(arm_len[i_sym], right - i)
                cur_arm_len = self.expand(s, i - min_arm_len, i + min_arm_len)
            else:
                cur_arm_len = self.expand(s, i, i)
            arm_len.append(cur_arm_len)
            if i + cur_arm_len > right:
                j = i
                right = i + cur_arm_len
            if 2 * cur_arm_len + 1 > end - start:
                start = i - cur_arm_len
                end = i + cur_arm_len
        return s[start+1:end+1:2]
```

参考文献

- leetcode中文官网：https://leetcode-cn.com/