回文子串与回文子序列

一、回文子串

子字符串 是字符串中 **连续** 的非空字符序列。
或者说是从前端或者后端删除若干个或 0 个形成的新字符串。

LeetCode:5.最长回文子串

> 给你一个字符串 s，找到 s 中最长的 **回文子串**。
>
> 示例 1：
> 输入：s = "babad"
> 输出："bab"
> 解释："aba" 同样是符合题意的答案。
>
> 示例 2：
> 输入：s = "cbbd"
> 输出："bb"
>
> 提示：
> 1 <= s.length <= 1000
> s 仅由数字和英文字母组成

1.区间 dp

用布尔二维数组 dp 来表示某个区间是否是回文串。
`dp[i][j]==True` 表示从 `s[i]`到 `s[j]`是一个回文串，那么当 `i==j` 时 `dp[i][j]=True`;当 `i>j` 时 `dp[i][j]=False`;
当 `i<j` 时，一旦 `s[i]!=s[j]`则直接为 False；否则还要对区间长度是否为 2 判断一下，`dp[i][j] = dp[i+1][j-1] & (s[i]==s[j])`。
由于 dp 的顺序是从小子串到大子串，所以应当对区间长度 L 进行遍历，同时初始化长度为 1 时都为 True，并维护 `maxl` 和 `begin` 来分别记录最长长度和起始位置。

```
def longestPalindrome(s: str) -> str:
    maxl,begin,n=1,0,len(s)
    dp=[[False]*n for _ in range(n)]

    #单个字符是回文的
    for i in range(n):
        dp[i][i]=True

    #枚举子串可能的所有长度
    for L in range(2,n+1):
        for i in range(n):
            j=L+i-1
            if j>=n:
                break

            if s[i]!=s[j]:
```

```
                dp[i][j]=False
            else:
                if L==2:
                    dp[i][j]=True
                else:
                    dp[i][j]=dp[i+1][j-1]

            #更新 maxl 和 begin
            if dp[i][j] and L>maxl:
                maxl=L
                begin=i

    return s[begin:begin+maxl]
```

或者用遍历右边界，再反向遍历左边界的写法：

```
def longestPalindrome(self, s: str) -> str:
    #dp[i][j] 表示从 i 到 j 是回文的
    #dp[i][j]=dp[i+1][j-1] and (s[i]==s[j])
    n=len(s)
    maxl=1;begin=0
    dp=[[False]*n for _ in range(n)]

    for j in range(n):
        for i in range(j,-1,-1):
            L=j-i+1
            if L==1:
                dp[i][j]=True
            elif L==2 and s[i]==s[j]:
                dp[i][j]=True
            elif L>=3 and s[i]==s[j] and dp[i+1][j-1]:
                dp[i][j]=True

            if dp[i][j] and L>maxl:
                maxl=L
                begin=i
    return s[begin:begin+maxl]
```

这里查看题解发现更简洁的写法：初始化都设置为 True，然后利用 `dp[i][j]=(dp[i+1][j-1] and s[i]==s[j])`来不断更新

```
def longestPalindrome(self, s: str) -> str:
    #dp[i][j] 表示从 i 到 j 是回文的
    #dp[i][j]=dp[i+1][j-1] and (s[i]==s[j])
    n=len(s)
    maxl=1;begin=0
    dp=[[True]*n for _ in range(n)]

    for j in range(n):
        for i in range(j-1,-1,-1):
            L=j-i+1
            # if L==1:
            #     dp[i][j]=True
            # elif L==2 and s[i]==s[j]:
            #     dp[i][j]=True
```

```python
            # elif L>=3 and s[i]==s[j] and dp[i+1][j-1]:
            #     dp[i][j]=True
            dp[i][j]=(dp[i+1][j-1] and s[i]==s[j])

            if dp[i][j] and L>maxl:
                maxl=L
                begin=i
    return s[begin:begin+maxl]
```

事实上，如果只需要预处理出某个字符串的各个区间是否是回文串（以方便后续以 O(1)的复杂度判断），只需要几行代码：

```python
def process(s):
    n=len(s)
    dp=[[True]*n for _ in range(n)]
    for j in range(n):
        for i in range(j-1,-1,-1):
            dp[i][j]=(dp[i+1][j-1] and s[i]==s[j])
```

这样，如果想知道 `s[i:j+1]`是否是回文串，只需要判断 `dp[i][j]==True` 即可。

2.Manacher 算法

利用了 **对称性优化**，实际上是对每个位置的初始化进行优化：

> 回文对称性观察：
> 假设我们在位置 i 处扩展得到了回文串，并且这个回文串的半径是 P[i]，也就是说位置 i 的回文串在左右各有 P[i] 个字符。
> 由于回文的对称性，我们可以知道回文串的对称位置也有相同长度的回文串。
> 利用对称性推测： 如果我们知道当前回文串的中心是 C，右边界是 R，并且 i 在这个回文串的右边界内（即 i < R），那么我们可以利用 i 的对称位置来推测 P[i]。
> P[mirr]：mirr 是 i 相对于中心 C 的对称位置。可以通过公式 mirr = 2 * C - i 计算得到。
> 因为 i 和 mirr 的位置对称，它们的回文串的半径应该相同。也就是说，如果 i 的回文半径 P[i] 没有超出右边界 R，那么我们可以初始化 P[i] = P[mirr]。这样就避免了重新计算 i 位置的回文串，而是直接利用对称的 mirr 位置。

```python
def ManacherPalindrome(s: str) -> str:
    t='#'+'#'.join(s)+'#'
    n=len(t)
    P=[0]*n
    C,R=0,0
    for i in range(n):
        # 找对称位置并判断是否可以初始化为对称位置的半径
        mirr=2*C-i
        if i<R:
            P[i]=min(P[mirr],R-i)

        # 拓展该位置的半径
        while i+P[i]+1<n and i-P[i]-1>=0 and t[i+P[i]+1]==t[i-P[i]-1]:
            P[i]+=1

        # 拓展后的回文串如果超出了右边界，更新
        if i+P[i]>R:
```

```
            C,R=i,i+P[i]

    max_len,center_idx=max((n,i) for i, n in enumerate(P))
    begin=(center_idx-max_len)//2
    return s[begin:begin+max_len]
```

LeetCode:131.分割回文串

> 给你一个字符串 s，请你将 s 分割成一些 子串，使每个子串都是 回文串 。返回 s 所有可能的分割方案。
>
> 示例 1：
> 输入：s = "aab"
> 输出：[["a","a","b"],["aa","b"]]
>
> 示例 2：
> 输入：s = "a"
> 输出：[["a"]]
>
> 提示：
> 1 <= s.length <= 16
> s 仅由小写英文字母组成

该题只需要使用模板处理出回文子序列的布尔 dp 数组，然后就是 dfs 来找到所有回文串了。
此题是 **回文子串** 和 **dfs 的回溯** 结合的题目。（但是难度居然只是中等？）

```
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        n=len(s)
        dp=[[True]*n for _ in range(n)]

        for j in range(n):
            for i in range(j-1,-1,-1):
                dp[i][j]=(dp[i+1][j-1] and s[i]==s[j])

        ans=[]
        #从 0 位置进入，让 j 向右遍历，每找到一个回文串就从 j+1 的位置进一步 dfs
        def dfs(i,curr):
            if i==n:
                ans.append(curr[:])
                return
            for j in range(i,n):
                if dp[i][j]:
                    curr.append(s[i:j+1])
                    dfs(j+1,curr)
                    curr.pop()

        dfs(0,[])

        return ans
```

二、回文子序列

子序列 是不改变剩余字符顺序的情况下，删除某些字符或者不删除任何字符形成的一个序列。

LeetCode:516.最长回文子序列

> 给你一个字符串 s ，找出其中最长的回文子序列，并返回该序列的长度。
>
> 示例 1：
> 输入：s = "bbbab"
> 输出：4
> 解释：一个可能的最长回文子序列为 "bbbb" 。
>
> 示例 2：
> 输入：s = "cbbd"
> 输出：2
> 解释：一个可能的最长回文子序列为 "bb" 。
>
> 提示：
> 1 <= s.length <= 1000
> s 仅由小写英文字母组成

1.区间 dp

用数值二维数组 dp 来表示某个区间内最长的回文子序列的长度。
`dp[i][j]`表示从 `s[i]`到 `s[j]`区间范围内的最长回文子序列长度，那么当 `i>j` 时 `dp[i][j]=0`； 当 `i==j` 时 `dp[i][j]=1`；
当 `i<j` 时，考虑区间两端点：如果 `s[i]==s[j]`，那么就全部考虑，`dp[i][j]=2+dp[i+1][j-1]`；反之则抛弃一个端点，`dp[i][j]=max(dp[i+1][j],dp[i][j-1])`
这里仍然采用 **遍历区间长度 L 和左端点 i** 和 **遍历右端点 j 和反向左端点 i** 两种方法。

```
def longestPalindromeSubseq(s: str) -> int:
    n=len(s)
    dp=[[0]*n for _ in range(n)]
    for i in range(n):
        dp[i][i]=1

    for L in range(2,n+1):
        for i in range(n):
            j=i+L-1
            if j>=n:
                break

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

    return dp[0][-1]
```

这是正向 j 反向 i 的方法。

```
def longestPalindromeSubseq(s: str) -> int:
    n=len(s)
    dp=[[0]*n for _ in range(n)]
    for j in range(n):
        for i in range(j,-1,-1):
            if i==j:
                dp[i][j]=1
            elif i>j:
                dp[i][j]=0
            elif i<j:
                if s[i]==s[j]:
                    dp[i][j]=2+dp[i+1][j-1]
                else:
                    dp[i][j]=max(dp[i+1][j],dp[i][j-1])
    return dp[0][-1]
```

2.递归

同样，我们取的结果是 `dp[0][-1]`，整体思路是从小区间到大区间，那么也可以采用递归的方式：

```
from functools import lru_cache
def longestPalindromeSubseq(s: str) -> int:
    @lru_cache(maxsize=None)
    def dfs(i,j):
        if i>j:
            return 0
        if i==j:
            return 1
        if s[i]==s[j]:
            return 2+dfs(i+1,j-1)
        return max(dfs(i+1,j), dfs(i,j-1))
    return dfs(0,len(s)-1)
```

LeetCode:3472.至多 K 次操作后的最长回文子序列

> 给你一个字符串 s 和一个整数 k。
> 在一次操作中，你可以将任意位置的字符替换为字母表中相邻的字符（字母表是循环的，因此 'z' 的下一个字母是 'a'）。例如，将 'a' 替换为下一个字母结果是 'b'，将 'a' 替换为上一个字母结果是 'z'；同样，将 'z' 替换为下一个字母结果是 'a'，替换为上一个字母结果是 'y'。
> 返回在进行 **最多 k 次操作** 后，s 的 **最长回文子序列** 的长度。
> **子序列** 是一个 **非空** 字符串，可以通过 **删除原字符串中的某些字符（或不删除任何字符）并保持剩余字符的相对顺序** 得到。
> **回文** 是正着读和反着读都相同的字符串。
>
> 示例 1：
> 输入: s = "abced", k = 2
> 输出: 3
> 解释:
> 将 s[1] 替换为下一个字母，得到 "acced"。

将 s[4] 替换为上一个字母，得到 "accec"。
子序列 "ccc" 形成一个长度为 3 的回文，这是最长的回文子序列。

示例 2：
输入: s = "aaazzz", k = 4
输出: 6
解释:
将 s[0] 替换为上一个字母，得到 "zaazzz"。
将 s[4] 替换为下一个字母，得到 "zaazaz"。
将 s[3] 替换为下一个字母，得到 "zaaaaz"。
整个字符串形成一个长度为 6 的回文。

提示:
1 <= s.length <= 200
1 <= k <= 200
s 仅由小写英文字母组成。

1.区间 dp

这里多了一个变量 K,在 dp 中新添一维作为 k 的 dp.

```
def longestPalindromicSubsequence(self, s: str, K: int) -> int:
    def min_steps(c1, c2):
        """
        计算将字符 c1 变为 c2 所需的最小步数，考虑字母表是循环的。
        """
        diff = abs(ord(c1) - ord(c2))
        return min(diff, 26 - diff)

    def longestPalindromeSubseq(s):
        n = len(s)

        # dp[k][i][j]表示在最多操作 k 步的前提下，s[i..j]的最长回文子序列长度
        # 注意第一位的长度是 K+1，因为要考虑操作数恰好是 K 的情况
        dp = [[[0] * n for _ in range(n)] for _ in range(K+1)]

        # 填充 dp 表
        for k in range(K+1):
            for j in range(n):
                for i in range(j,-1,-1):
                    if i==j:
                        dp[k][i][j]=1
                    else:
                        if s[i]==s[j]:
                            dp[k][i][j]=2+dp[k][i+1][j-1]
                        else:
                            cost=min_steps(s[i],s[j])
                            if cost<=k:
                                dp[k][i][j]=2+dp[k-cost][i+1][j-1]

dp[k][i][j]=max(dp[k][i][j],dp[k][i+1][j],dp[k][i][j-1])
```

```
        return dp[K][0][-1]
    return longestPalindromeSubseq(s)
```

2.递归

```
from functools import lru_cache
def longestPalindromicSubsequence(self, s: str, K: int) -> int:
    def min_steps(c1, c2):
        """
        计算将字符 c1 变为 c2 所需的最小步数，考虑字母表是循环的。
        """
        diff = abs(ord(c1) - ord(c2))
        return min(diff, 26 - diff)

    @lru_cache(maxsize=None)
    def dfs(k,i,j):
        if i>j:
            return 0
        if i==j:
            return 1
        if s[i]==s[j]:
            return 2+dfs(k,i+1,j-1)
        cost=min_steps(s[i],s[j])
        if cost<=k:
            return max(2+dfs(k-cost,i+1,j-1), dfs(k,i+1,j), dfs(k,i,j-1))
        return max(dfs(k,i+1,j), dfs(k,i,j-1))

    n=len(s)
    ans=dfs(K,0,n-1)
    dfs.cache_clear() # 防止超内存
    return ans
```