南京信息工程大学

课程报告

题　　目　回文串的算法设计与分析

学生姓名　叶成宇/孙艺纱

学　　号　201833050027/201833050097

学　　院　应用技术学院

专　　业　计算机科学与技术

指导教师　庞亚伟

二〇二一　年 6 月 15 日

目 录

回文串的算法设计与分析

叶成宇/孙艺纱

南京信息工程大学 应用技术学院，江苏 南京 210044

摘要：本文主要内容是基于算法设计与分析课程要求，通过动态规划，回溯算法等算法设计方法对回文串的匹配算法以及回文串的分割算法进行求解。

关键词：动态规划，回溯，回文串。

Algorithm Design and Analysis of Palindrome

ChengyuYe/YishaSun

Nanjing University of Information and Science Technology Applied Technology College
Nanjing Jiangsu 210044

Abstract：The main content of this paper is to solve the matching algorithm of palindrome string through dynamic programming, backtracking algorithm and other algorithm design methods based on algorithm design and analysis course requirements.

Key Words：Dynamic programming, backtracking, palindrome.

1 绪论

1.1 课题背景

字符串算法的一个广阔的应用领域就是基因序列，比如有了回文串的识别算法，我们就可以在存储基因序列的时候进行压缩，节省时间和空间。

另一方面，基因序列中自然的存在许多回文序列，这些序列有很多功能，高中生物选修三里就提到了。比如人类 Y 染色体上就有一段近似回文序列，这个序列和睾丸有关。因为回文序列太重要了，所以搜索回文算法的应用，得到的结果里大部分都是和基因序列有关的。

本文通过暴力解法、动态规划、回溯算法等方法对回文串的匹配与分割进行求解。

1.2 回文串的定义

正读和反读都相同的字符序列为“回文”，如“abba”、“abccba”是“回文”，“abcde”和“ababab”则不是“回文”。字符串的最长回文子串，是指一个字符串中包含的最长的回文子串。例如“1212134”的最长回文子串是“12121”。

2 回文串的匹配

2.1 问题

求解字符串的最长回文子串，是指一个字符串中包含的最长的回文子串。例如“1212134”的最长回文子串是“12121”。

2.2 暴力解法

2.2.1 解析

暴力求解是最容易想到的，要截取字符串的所有子串，然后再判断这些子串中哪些是回文的，最后返回回文子串中最长的即可。

2.2.2 设计

（1）枚举 s 的所有子串；

（2）然后逐个判断每个子串的回文性质；

（3）同时记录最长子串；

细节：记录最长子串需要截取，截取有一定性能消耗。替代方式：记录最长回文子串的起始位置 start 和最长回文子串的长度 maxLen，到遍历完成以后，再做截取。

下面给出判断回文串的函数：

```
private boolean isPalindrome(String s, int start, int end) {
    while (start < end) {
        if (s.charAt(start++) != s.charAt(end--))
            return false;
    }
    return true;
}
```

下面给出暴力求解回文串主体部分代码：

```
public String longestPalindrome(String s) {
    if (s.length() < 2)
        return s;
    int start = 0;
    int maxLen = 0;
```

```
    for (int i = 0; i < s.length() - 1; i++) {
        for (int j = i; j < s.length(); j++) {
            //截取所有子串，然后在逐个判断是否是回文的
            if (isPalindrome(s, i, j)) {
                if (maxLen < j - i + 1) {
                    start = i;
                    maxLen = j - i + 1;
                }
            }
        }
    }
    return s.substring(start, start + maxLen);
}
```

2.2.3 分析

暴力解法复杂度分析：

时间复杂度：$O(N^3)$

这里 N 是字符串的长度，枚举字符串的左边界、右边界是 $O(N^2)$ ，然后继续验证子串是否是回文子串，这一步操作是 O(N)

空间复杂度：O(1)

只使用到常数个临时变量，与字符串长度无关。

2.3 **中心扩展法**

2.3.1 解析

中心扩展算法的核心是：遍历整个字符串，将每个字符和字符的间隔作为“回文子串”的中心，向两侧扩展，直到左右两侧扩展字符不相同为止，即得到一个回文子串。

如图 2-1 所示，举例 str = acdbbdaa。每个位置向两边扩散都会出现一个窗口大小（len）。如果 len>maxLen(用来表示最长回文串的长度）。则更新 maxLen 的值。

因为我们最后要返回的是具体子串，而不是长度，因此，还需要记录一下 maxLen 时的起始位置（maxStart），即此时还要 maxStart=len。

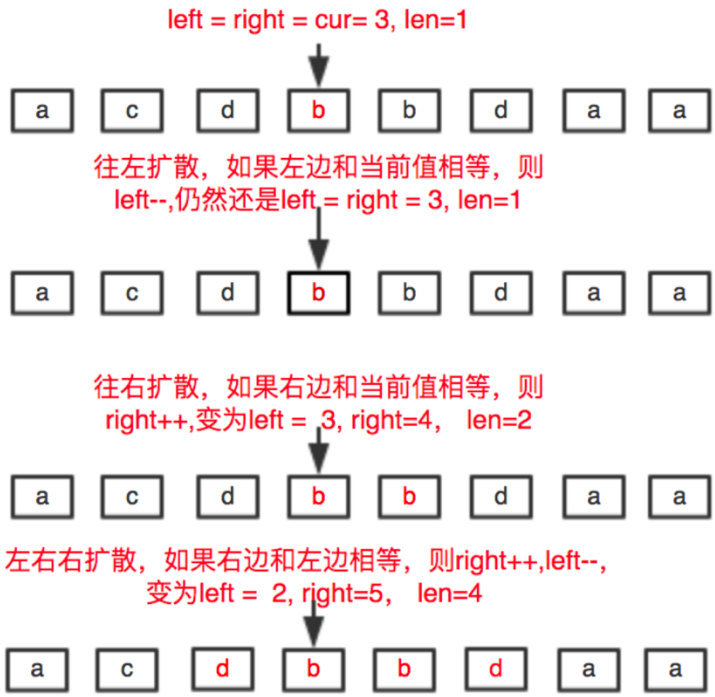

图 2-1 中心扩展法

2.3.2 设计

我们仔细观察一下状态转移方程：

$$P(i,i)=true$$
$$P(i,i+1)=(Si==Si+1)$$
$$P(i,j)=P(i+1,j-1)\wedge(Si==Sj)$$

找出其中的状态转移链：

$$P(i,j)\leftarrow P(i+1,j-1)\leftarrow P(i+2,j-2)\leftarrow\ldots\leftarrow某一边界情况$$

可以发现，所有的状态在转移的时候的可能性都是唯一的。也就是说，我们可以从每一种边界情况开始扩展，也可以得出所有的状态对应的答案。

边界情况即为子串长度为 1 或 2 的情况。我们枚举每一种边界情况，并从对应的子串开始不断地向两边扩展。如果两边的字母相同，我们就可以继续扩展，例如从 P(i+1,j−1)扩展到 P(i,j)；如果两边的字母不同，我们就可以停止扩展，因为在这之后的子串都不能是回文串了。

我们可以发现，边界情况对应的子串实际上就是我们扩展出的回文串的回文中心。中心扩展法的本质即为：我们枚举所有的回文中心并尝试扩展，直到无法扩展为止，此时的回文串长度即为此回文中心下的最长回文串长度。我们对所有的长度求出最大值，即可得到最终的答案。

下面给出中心扩展法主体部分代码：

```
class Solution {
    public String longestPalindrome(String s) {
        if (s == null || s.length() < 1) {
            return "";
        }
```

```
        int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);
            int len2 = expandAroundCenter(s, i, i + 1);
            int len = Math.max(len1, len2);
            if (len > end - start) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return s.substring(start, end + 1);
    }
    public int expandAroundCenter(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            --left;
            ++right;
        }
        return right - left - 1;
    }
}
```

2.3.3 分析

中心扩展法复杂度分析：

时间复杂度：O(N^2)

其中 N 是字符串的长度。长度为 1 和 2 的回文中心分别有 N 和 N-1 个，每个回文中心最多会向外扩展 O(N)次。

空间复杂度：O(1)

2.4 动态规划

2.4.1 解析

动态规划算法的核心是：对于一个子串而言，如果它是回文串，并且长度大于 2，那么将它首尾的两个字母去除之后，它仍然是个回文串。例如对于字符串“ababa”，如果我们已经知道“bab”是回文串那么“ababa”一定是回文串，这是因为它的首尾两个字母都是 “a”。

如图 2-2 所示，回文串是天然存在状态转移性质的，因为在去掉了两头的字符以后，他依然是一个回文串。我们也就知道了，当一个子串的两头不想等，那么他就不是回文串。如果一个子串的两头想等，我们需要看去掉了两头以后的剩下部分的子串是否是回文串，也就是说字符串是否是回文，由中间子串是否是回文决定。显然中间的 C 和 D 不同，所以该字符串不是回文串。

图 2-2 判断回文串

2.4.2 设计

根据这样的思路，我们就可以用动态规划的方法解决本题。我们用 P(i,j)表示字符串 s 的第 i 到 j 个字母组成的串（下文表示成 s[i:j]）是否为回文串：

P(i,j)= true,如果子串 Si…Sj 是回文串

false，其它情况

这里的「其它情况」包含两种可能性：

（1）s[i,j] 本身不是一个回文串；

（2）i>j，此时 s[i,j] 本身不合法。

那么我们就可以写出动态规划的状态转移方程：

$$P(i,j) = P(i+1,j-1) \wedge (Si == Sj)$$

也就是说，只有 s[i+1:j−1]是回文串，并且 s 的第 i 和 j 个字母相同时，s[i:j]才会是回文串。

上文的所有讨论是建立在子串长度大于 2 的前提之上的，我们还需要考虑动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1 的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。因此我们就可以写出动态规划的边界条件：

$$P(i,i)=true$$
$$P(i,i+1)=(Si==Si+1)$$

根据这个思路，我们就可以完成动态规划了，最终的答案即为所有 P(i,j)=true 中 j−i+1（即子串长度）的最大值。

下面给出动态规划算法的主体部分代码：

```
public class Solution {
    public String longestPalindrome(String s) {
        int len = s.length();
        if (len < 2) {
            return s;
        }
        int maxLen = 1;
        int begin = 0;
        // dp[i][j] 表示 s[i..j] 是否是回文串
        boolean[][] dp = new boolean[len][len];
        // 初始化：所有长度为 1 的子串都是回文串
        for (int i = 0; i < len; i++) {
            dp[i][i] = true;
        }
        char[] charArray = s.toCharArray();
        // 递推开始
        // 先枚举子串长度
        for (int L = 2; L <= len; L++) {
            // 枚举左边界，左边界的上限设置可以宽松一些
            for (int i = 0; i < len; i++) {
                // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得
                int j = L + i - 1;
                // 如果右边界越界，就可以退出当前循环
                if (j >= len) {
```

```
                    break;
                }
                if (charArray[i] != charArray[j]) {
                    dp[i][j] = false;
                } else {
                    if (j - i < 3) {
                        dp[i][j] = true;
                    } else {
                        dp[i][j] = dp[i + 1][j - 1];
                    }
                }
                // 只要 dp[i][L] == true 成立，就表示子串 s[i..L] 是回文，此时记录回文长度和起始位置
                if (dp[i][j] && j - i + 1 > maxLen) {
                    maxLen = j - i + 1;
                    begin = i;
                }
            }
        }
        return s.substring(begin, begin + maxLen);
    }
}
```

2.4.3 分析

动态规划算法的复杂度分析：

时间复杂度：$O(N^2)$

其中 n 是字符串的长度。动态规划的状态总数为 $O(N^2)$，对于每个状态，我们需要转移的时间为 O(1)。

空间复杂度：$O(N^2)$

即存储动态规划状态需要的空间。

3 回文串的分割

3.1 问题

给定一个字符串 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。

3.2 回溯算法解法

3.2.1 解析

因为我们要分割出所有的字符串，所以显然我们要采用回溯算法求解此道问题。他可以考察所有的可能。随后基于当前选择展开出的分支，展开一颗包含所有解的二叉树，随后在这颗二叉树上采用深度优先遍历，找出所有的解，遍历到的解加入到解集数组中，待深度优先遍历结束后，返回解集即可。

基于上一次选择，而来到的节点可能有不同的选择，用 for 循环枚举出来，结合约束条件剪枝，被剪掉的是选项，是不可能产生合法解的搜索分支，避免了不必要的搜索。每一次迭代没被剪枝的话，会作出一个选择，基于这个选择，继续递归下去。上面这个递归结束时，不管由于什么原因，是找到合法解而返回，还是走完 for 循环而自然结束。都意味着，基于这

个选择的所有可能性都考察完了，基于这个选择所展开的树都搜索完了。无需再考察这个选择，撤销这个选择，回到选择前的状态，去作另一个选择，即切到另一个迭代，这样我们即可找到字符串中所有被切割的子回文串。

3.2.2 设计

如图 3-1 所示。我们通过一棵二叉树简单的分析了字符串 aab 的情况。每产生一个结点表示剩余没有扫描到的字符串，产生的分支是截取了剩余字符串的前缀。

如果前缀字符串是回文串，则可以产生分支和结点。

如果前缀字符串不是回文串，则不产生分支和结点。

当我们的某个叶子结点是空字符串的时候，从叶子结点到跟结点的路径就是我们要的一个回文串，也就是应当保存到结果集中的一个数据。所以显然我们只需用一个深度优先遍历把所有的结果记录下来。

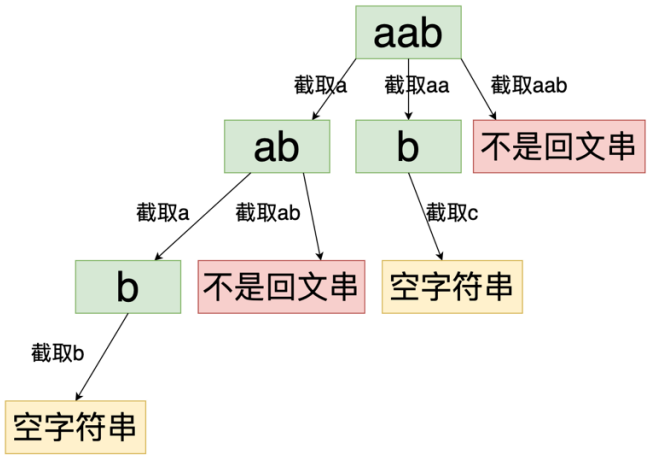

图 3-1

因为是对回文字符串进行搜索，所以我们这里定义回文字符串的判断函数如下：

```
def pending_s(s):
    l, r = 0, len(s) - 1
    while l < r:
        if s[l] != s[r]:
            return False
        l += 1
        r -= 1
    return True
```

接下来定义回溯函数和剪枝操作如下，这里的 index 作为遍历到的索引的位置，也作为判断的条件。

```
def back_track(s, index):
    if index == len(s):
        result.append(path[:])
        return
    for i in range(index, len(s)):
        if pending_s(s[index : i + 1]):
            path.append(s[index: i + 1])
            back_track(s, i + 1)
```

```
            path.pop( )
```

3.2.3 分析

回溯算法复杂度分析：

时间复杂度：$O(N\cdot 2^N)$

这里 N 为输入字符串的长度，每一个位置可拆分，也可不拆分，尝试是否可以拆分的时间复杂度为 $O(2^N)$，判断每一个子串是否是回文子串，时间复杂度为 O(N)；

空间复杂度：$O(N)$或 $O(2^N\cdot N)$

如果不计算保存结果的空间，空间复杂度为 O(N)，递归调用栈的高度为 N。

如果计算保存答案需要空间 2^N*N，这里 2^N 为保守估计，实际情况不会这么多。空间复杂度为 $O(2^N*N)$。

3.3 回溯算法+动态规划预处理解法

3.3.1 解析

当我们仅仅使用回溯法和剪枝操作进行字符串处理的时候，我们是使用了双指针法对字符串进行判断是否为回文串。但是很显然这种方法会产生重复的计算

如当 s = aaba 时，对于前 2 个字符 aa，我们有 2 种分割方法[aa]和[a,a]，当我们每一次搜索到字符串的第 i=2 个字符 b 时，都需要对于每个 s[i..j]使用双指针判断其是否为回文串，这就产生了重复计算。

因此，我们可以将字符串 s 的每个子串 s[i..j]是否为回文串预处理出来，使用动态规划即可。

3.3.2 设计

由于需要求出字符串 s 的所有分割方案，因此我们考虑使用搜索+ 溯的方法枚举所有可能的分割方法并进行判断。

假设我们当前搜索到字符串的第 i 个字符，且 [0..i−1] 位置的所有字符已经被分割成若干个回文串，并且分割结果被放入了答案数组 ans 中，那么我们就需要枚举下一个回文串的右边界 j，使得 s[i..j] 是一个回文串。

因此，我们可以从 i 开始，从小到大依次枚举 j。对于当前枚举的 j 值，我们使用双指针的方法判断 s[i..j]s[i..j]是否为回文串：如果 s[i..j]是回文串，那么就将其加入答案数组 ans 中，并以 j+1j+1 作为新的 i 进行下一层搜索，并在未来的回溯时将 s[i..j] 从 ans 中移除。如果我们已经搜索完了字符串的最后一个字符，那么就找到了一种满足要求的分割方法。

但是如此操作，会产生重复的计算，于是我们使用动态规划将回文串预处理出来，加入答案数组 ans 中。

于是设 f(i,j) 表示 s[i..j]是否为回文串，那么有状态转移方程：

$$f(i,j)=\begin{cases}\texttt{True}, & i\geq j\\ f(i+1,j-1)\wedge(s[i]=s[j]), & \text{otherwise}\end{cases}$$

预处理完成之后，我们只需要 O(1)的时间就可以判断任意 s[i..j]是否为回文串了，

于是我们修改主体代码为如下：

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

```
        for i in range(n - 1, -1, -1):
            for j in range(i + 1, n):
                f[i][j] = (s[i] == s[j]) and f[i + 1][j - 1]
        ret = list()
        ans = list()
        def dfs(i: int):
            if i == n:
                ret.append(ans[:])
                return
            for j in range(i, n):
                if f[i][j]:
                    ans.append(s[i:j+1])
                    dfs(j + 1)
                    ans.pop()
        dfs(0)
        return ret
```

3.3.3 分析

回溯算法+动态规划预处理复杂度分析:

时间复杂度：$O(N \cdot 2^N)$

其中 N 是字符串 s 的长度。在最坏情况下,s 包含 N 个完全相同的字符，因此它的任意一种划分方法都满足要求。而长度为 n 的字符串的划分方案数为 $2^{N-1}=O(2^N)$，每一种划分方法需要 O(N)的时间求出对应的划分结果并放入答案，因此总时间复杂度为 $O(N \cdot 2^N)$。尽管动态规划预处理需要 $O(N^2)$的时间，但在渐进意义下小于 $O(N \cdot 2^N)$，因此可以忽略。

空间复杂度：$O(N^2)$

这里不计算返回答案占用的空间。数组 f 需要使用的空间为 $O(N^2)$，而在回溯的过程中，我们需要使用 O(N)的栈空间以及 O(N)用来存储当前字符串分割方法的空间。由于 O(N)在渐进意义下小于 $O(N^2)$，因此空间复杂度为 $O(N^2)$。

3 总结

本文通过对最长回文串的求解以及回文串的分割的算法求解。通过暴力解法、动态规划解法、回溯解法等算法设计的技巧求解问题，深入了解了动态规划算法，回溯算法等高级算法的解体步骤及其思路。

对最长回文串以及回文串的分割算法中均用到了动态规划的解法，动态规划常常适用于有重叠子问题和最优子结构性质的问题，动态规划方法所耗时间往往远少于朴素解法。动态规划背后的基本思想非常简单。大致上，若要解一个给定问题，我们需要解其不同部分（即子问题），再根据子问题的解以得出原问题的解。通常许多子问题非常相似，为此动态规划法试图仅仅解决每个子问题一次，从而减少计算量：一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。这种做法在重复子问题的数目关于输入的规模呈指数增长时特别有用。

对回文串的分割算法中用到了回溯算法，对于某些计算问题而言，回溯法是一种可以找出所有（或一部分）解的一般性算法，尤其适用于约束补偿问题，在解决约束满足问题时，我们逐步构造更多的候选解，并且在确定某一部分候选解不可能补全成正确解之后放弃继续搜索这个部分候选解本身及其可以拓展出的子候选解，转而测试其他的部分候选解。

在计算计科学与技术专业中，算法设计与分析是一门非常重要的课程，很多人为它如痴如醉。很多问题的解决，程序的编写都要依赖它，在软件还是面向过程的阶段，就有程序=算法+数据结构这个公式。算法的学习对于培养一个人的逻辑思维能力是有极大帮助的，这门课培养了我思考分析问题，解决问题的能力。

如果一个算法有缺陷，或不适合某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂性和时间复杂度来衡量。算法可以使用自然语言、伪代码、流程图等多种不同的方法来描述。计算机系统中的操作系统、语言编译系统、数据库管理系统以及各种各样的计算机应用系统中的软件，都必须使用具体的算法来实现。算法设计与分析是计算机科学与技术的一个核心问题。因此，学习算法无疑会增强自己的竞争力，提高自己的修为，为自己增彩。

致谢

感谢南京信息工程大学庞亚伟老师的悉心指导。

附录-完整代码

最长回文串-暴力解法（java）

```
private boolean isPalindrome(String s, int start, int end) {
    while (start < end) {
        if (s.charAt(start++) != s.charAt(end--))
            return false;
    }
    return true;
}
public String longestPalindrome(String s) {
    if (s.length() < 2)
        return s;
    int start = 0;
    int maxLen = 0;
    for (int i = 0; i < s.length() - 1; i++) {
        for (int j = i; j < s.length(); j++) {
            if (isPalindrome(s, i, j)) {
                if (maxLen < j - i + 1) {
                    start = i;
                    maxLen = j - i + 1;
                }
            }
        }
    }
    return s.substring(start, start + maxLen);
}
```

最长回文串-中心扩展法（java）

```
class Solution {
    public String longestPalindrome(String s) {
        if (s == null || s.length() < 1) {
            return "";
        }
        int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);
            int len2 = expandAroundCenter(s, i, i + 1);
```

```
            int len = Math.max(len1, len2);
            if (len > end - start) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return s.substring(start, end + 1);
    }

    public int expandAroundCenter(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) ==
s.charAt(right)) {
            --left;
            ++right;
        }
        return right - left - 1;
    }
}
```

最长回文串-动态规划（java）

```
public class Solution {

    public String longestPalindrome(String s) {
        int len = s.length();
        if (len < 2) {
            return s;
        }

        int maxLen = 1;
        int begin = 0;
        // dp[i][j] 表示 s[i..j] 是否是回文串
        boolean[][] dp = new boolean[len][len];
        // 初始化：所有长度为 1 的子串都是回文串
        for (int i = 0; i < len; i++) {
            dp[i][i] = true;
        }
```

```
        char[] charArray = s.toCharArray();
        // 递推开始
        // 先枚举子串长度
        for (int L = 2; L <= len; L++) {
            // 枚举左边界，左边界的上限设置可以宽松一些
            for (int i = 0; i < len; i++) {
                // 由 L 和 i 可以确定右边界，即 j - i + 1 = L 得
                int j = L + i - 1;
                // 如果右边界越界，就可以退出当前循环
                if (j >= len) {
                    break;
                }

                if (charArray[i] != charArray[j]) {
                    dp[i][j] = false;
                } else {
                    if (j - i < 3) {
                        dp[i][j] = true;
                    } else {
                        dp[i][j] = dp[i + 1][j - 1];
                    }
                }

                // 只要 dp[i][L] == true 成立，就表示子串 s[i..L] 是回
文，此时记录回文长度和起始位置
                if (dp[i][j] && j - i + 1 > maxLen) {
                    maxLen = j - i + 1;
                    begin = i;
                }
            }
        }
        return s.substring(begin, begin + maxLen);
    }
}
```

回文串分割-回溯算法（python3）

```
from typing import *

class Solution:
    def partition(self, s: str) -> List[List[str]]:
        result = []
        path = []
        #判断是否是回文串
        def pending_s(s):
            l, r = 0, len(s) - 1
            while l < r:
                if s[l] != s[r]:
                    return False
                l += 1
                r -= 1
            return True
        #回溯函数，这里的 index 作为遍历到的索引位置，也作为终止判断的条件
        def back_track(s, index):
           #如果对整个字符串遍历完成,并且走到了这一步,则直接加入 result
            if index == len(s):
                result.append(path[:])
                return
            #遍历每个子串
            for i in range(index, len(s)):
                #剪枝，因为要求每个元素都是回文串，那么我们只对回文串进行递归，不是回文串的部分直接不 care 它
                #当前子串是回文串
                if pending_s(s[index : i + 1]):
                    #加入当前子串到 path
                    path.append(s[index: i + 1])
                    #从当前 i+1 处重复递归
                    back_track(s, i + 1)
                    #回溯
                    path.pop()
        back_track(s, 0)
        return result
```

```
ex = Solution()
print(ex.partition('aab'))
```

回文串分割-回溯算法+动态规划预处理（python3）

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

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

        ret = list()
        ans = list()

        def dfs(i: int):
            if i == n:
                ret.append(ans[:])
                return

            for j in range(i, n):
                if f[i][j]:
                    ans.append(s[i:j+1])
                    dfs(j + 1)
                    ans.pop()

        dfs(0)
        return ret

ex = Solution()
print(ex.partition('aab'))
```

分工安排	
任务：	完成人员：
第一章 绪论	叶成宇/孙艺纱
第二章 回文串的匹配	孙艺纱
第三章 回文串的分割	叶成宇
排版	叶成宇
附录 java 代码	孙艺纱
附录 python 代码	叶成宇
总体分工比例：5:5	