宫水三叶的刷题日记

回文串问题

Author : 宫水三叶
Date : 2021/10/07
QQ Group: 703311589
WeChat : oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「回文串问题」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「回文串问题」即可获取最新下载链接。

💡**下面介绍使用本合集的最佳使用实践：**

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「回文串问题」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **5. 最长回文子串** ，难度为 **中等**。

Tag : 「模拟」、「回文串」

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"
```

示例 3：

```
输入：s = "a"
输出："a"
```

示例 4：

```
输入：s = "ac"
输出："a"
```

提示：

- 1 <= s.length <= 1000
- s 仅由数字和英文字母（大写和/或小写）组成

朴素解法

这道题有一个很容易就能想到的简单做法：枚举字符串 `s` 中的每一位，作为回文串的中心点，左右进行扩展，直到达到边界或者不满足回文串定义为止。

这样做的思路必然是正确的。

但很显然这是一个朴素（暴力）做法，那么我们如何确定这一做法是否可行呢？

还记得我们上一节的分析思路吗？当我们有了一个简单的实现方法之后，需要从**题目的数据规模**、**计算机的处理速度**和**实现方法的计算量**出发，判断这样的做法是否不会超时。

由于字符串长度最多只有 1000，而我们的实现方法是 $O(n^2)$，因此我们算法的计算量应该在 10^6 以内，是在计算机每秒的处理范围内的。

首先枚举回文串的中心 i，然后分两种情况向两边扩展边界，直到达到边界或者不满足回文串定义为止：

- 回文串长度是奇数，则依次判断 `s[i − k] == s[i + k], k = 1,2,3…`

- 回文串长度是偶数，则依次判断 `s[i − k] == s[i + k − 1], k = 1,2,3…`

代码：

```
class Solution {
    public String longestPalindrome(String s) {
        String ans = "";
        for (int i = 0; i < s.length(); i++) {
            // 回文串为奇数
            int l = i - 1, r = i + 1;
            String sub = getString(s, l, r);
            if (sub.length() > ans.length()) ans = sub;

            // 回文串为偶数
            l = i - 1;
            r = i + 1 - 1;
            sub = getString(s, l, r);
            if (sub.length() > ans.length()) ans = sub;
        }
        return ans;
    }
    String getString(String s, int l, int r) {
        while (l >= 0 && r < s.length() && s.charAt(l) == s.charAt(r)) {
            l--;
            r++;
        }
        return s.substring(l + 1, r);
    }
}
```

- 时间复杂度：先枚举了 `s` 中的每个字符作为回文串的中心点，再从中心点出发左右扩展，最多扩展到边界。复杂度是 $O(n^2)$
- 空间复杂度：$O(1)$

Manacher 算法

这是一个比较冷门的算法，使用范围也比较单一，只能用于解决「回文串」问题。

Manacher 确实是「回文串」问题的最优解。

但事实上我还没有见过必须使用 Manacher 算法才能过的回文串题。

因此我这里直接给解决方案（可以直接当做模板来使用），而不再讨论算法的具体实现原理。

Manacher 算法较长，为了避免回文串长度奇偶问题的分情况讨论，我会对原字符进行处理，在边界和字符之间插入占位符。

使用了这样的技巧之后，**当非占位字符作为回文串的中心时，对应了回文串长度为奇数的情况；当占位字符作为回文串的中心时，对应了回文串长度为偶数的情况。**。

举个例子：

> 原字符："babad"，转换后："*b*a*b*a*d*"，得到的回文串："*b*a*b*"，然后再去除占位符输出："bab"。

解释："aba" 同样是符合题意的答案。

代码：

```java
class Solution {
    public String longestPalindrome(String s) {
        if (s.length() == 1) return s;

        char[] chars = manacherString(s);
        int n = chars.length;
        int[] pArr = new int[n];
        int C = -1, R = -1, pos = -1;
        int max = Integer.MIN_VALUE;
        for (int i = 0; i < n; i++) {
            pArr[i] = i < R ? Math.min(pArr[C * 2 - i], R - i) : 1;
            while (i + pArr[i] < n && i - pArr[i] > -1) {
                if (chars[i + pArr[i]] == chars[i - pArr[i]]) {
                    pArr[i]++;
                } else {
                    break;
                }
            }
            if (i + pArr[i] > R) {
                R = i + pArr[i];
                C = i;
            }
            if (pArr[i] > max) {
                max = pArr[i];
                pos = i;
            }
        }
        int offset = pArr[pos];
        StringBuilder sb = new StringBuilder();
        for (int i = pos - offset + 1; i <= pos + offset - 1; i++) {
            if (chars[i] != '#') sb.append(chars[i]);
        }
        return sb.toString();
    }
    char[] manacherString(String s) {
        char[] chars = new char[s.length() * 2 + 1];
        for (int i = 0, idx = 0; i < chars.length; i++) {
            chars[i] = (i & 1) == 0 ? '#' : s.charAt(idx++);
        }
        return chars;
    }
}
```

- 时间复杂度：只对字符串进行了一次扫描。复杂度为 $O(n)$
- 空间复杂度：$O(1)$

总结

今天这道题目，三叶除了提供常规的、时间复杂度为 $O(n^2)$ 的朴素解法以外，还给你提供了关于「回文串」的最优解 Manacher 算法模板，建议有余力的同学可以背过。

背过这样的算法的意义在于：相当于大脑里有了一个时间复杂度为 $O(n)$ 的 api 可以使用，这个 api 传入一个字符串，返回该字符串的最大回文子串。

同时借助 Manacher 算法，还给大家介绍了如何避免回文串长度的分情况讨论，这个技巧只要涉及「回文串」问题都适用（无论是否使用 Manacher 算法）。

对于想要背过 Manacher 算法的同学，建议先敲 3 遍，默写 2 遍，然后过了 24 小时，再默写 2 遍，一周后，再进行重复，直到熟练。

不要害怕遗忘，遗忘是正常的，多进行几次重复便会形成肌肉记忆。LeetCode 周赛上常年占据第一页的选手，无不都是对算法套路和模板极其熟练。加油 ~

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **9. 回文数** ，难度为 **简单**。

Tag : 「数学」、「回文串」

给你一个整数 x ，如果 x 是一个回文整数，返回 true ；否则，返回 false 。

回文数是指正序（从左向右）和倒序（从右向左）读都是一样的整数。

例如，121 是回文，而 123 不是。

示例 1：

```
输入：x = 121
输出：true
```

示例 2：

```
输入：x = -121
输出：false
解释：从左向右读, 为 -121 。 从右向左读, 为 121- 。因此它不是一个回文数。
```

示例 3：

```
输入：x = 10
输出：false
解释：从右向左读, 为 01 。因此它不是一个回文数。
```

示例 4：

```
输入：x = -101
输出：false
```

提示：

- -2^{31} <= x <= 2^{31} - 1

进阶：你能不将整数转为字符串来解决这个问题吗？

字符串解法

虽然进阶里提到了不能用字符串来解决，但还是提供一下吧。

代码：

```java
class Solution {
    public boolean isPalindrome(int x) {
        String s = String.valueOf(x);
        StringBuilder sb = new StringBuilder(s);
        sb.reverse();
        return sb.toString().equals(s);
    }
}
```

- 时间复杂度：数字 n 的位数，数字大约有 log_{10}^{n} 位，翻转操作要执行循环。复杂度为 $O(log_{10}^{n})$

- 空间复杂度：使用了字符串作为存储。复杂度为 $O(log_{10}^{n})$

非字符串解法（完全翻转）

原数值 `x` 的不超过 int 的表示范围，但翻转后的值会有溢出的风险，所以这里使用 `long` 进行接收，最后对比两者是否相等。

代码：

```
class Solution {
    public boolean isPalindrome(int x) {
        if (x < 0) return false;
        long ans = 0;
        int t = x;
        while (x > 0) {
            ans = ans * 10 + x % 10;
            x /= 10;
        }
        return ans - t == 0;
    }
}
```

- 时间复杂度：数字 n 的位数，数字大约有 log_{10}^{n} 位。复杂度为 $O(log_{10}^{n})$
- 空间复杂度：$O(1)$

非字符串解法（部分翻转）

如果在进阶中增加一个我们熟悉的要求：环境中只能存储得下 32 位的有符号整数。

那么我们就连 `long` 也不能用了，这时候要充分利用「回文」的特性：前半部分和后半部分（翻转）相等。

这里的前半部分和后半部分（翻转）需要分情况讨论：

- 回文长度为奇数：回文中心是一个独立的数，即 `忽略回文中心后，前半部分 == 后半部分（翻转）`。如 1234321 回文串
- 回文长度为偶数：回文中心在中间两个数中间，即 `前半部分 == 后半部分（翻转）`。

如 123321

代码：

```
class Solution {
    public boolean isPalindrome(int x) {
        // 对于 负数 和 x0、x00、x000 格式的数，直接返回 flase
        if (x < 0 || (x % 10 == 0 && x != 0)) return false;
        int t = 0;
        while (x > t) {
            t = t * 10 + x % 10;
            x /= 10;
        }
        // 回文长度的两种情况：直接比较 & 忽略中心点（t 的最后一位）进行比较
        return x == t || x == t / 10;
    }
}
```

- 时间复杂度：数字 n 的位数，数字大约有 log_{10}^n 位。复杂度为 $O(log_{10}^n)$
- 空间复杂度：$O(1)$

> **🍭 更多精彩内容，欢迎关注：**公众号** / Github / LeetCode / **知乎** **

题目描述

这是 LeetCode 上的 **131. 分割回文串** ，难度为 **中等**。

Tag : 「回文串」、「回溯算法」、「动态规划」

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。

回文串 是正着读和反着读都一样的字符串。

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

动态规划 + 回溯算法

求所有的分割方案，凡是求所有方案的题基本上都没有什么优化方案，就是「爆搜」。

问题在于，爆搜什么？显然我们可以爆搜每个回文串的起点。如果有连续的一段是回文串，我们再对剩下连续的一段继续爆搜。

为什么能够直接接着剩下一段继续爆搜？

因为任意的子串最终必然能够分割成若干的回文串（最坏的情况下，每个回文串都是一个字母）。

所以我们每次往下爆搜时，只需要保证自身连续一段是回文串即可。

举个🌰 来感受下我们的爆搜过程，假设有样例 `abababa`，刚开始我们从起点第一个 a 进行爆搜：

1. 发现 `a` 是回文串，先将 `a` 分割出来，再对剩下的 `bababa` 进行爆搜
2. 发现 `aba` 是回文串，先将 `aba` 分割出来，再对剩下的 `baba` 进行爆搜
3. 发现 `ababa` 是回文串，先将 `ababa` 分割出来，再对剩下的 `ba` 进行爆搜
4. 发现 `abababa` 是回文串，先将 `abababa` 分割出来，再对剩下的 `` 进行爆搜

...

然后再对下一个起点（下个字符） `b` 进行爆搜？

不需要。

因为单个字符本身构成了回文串，所以以 `b` 为起点，`b` 之前构成回文串的方案，必然覆盖在我们以第一个字符为起点所展开的爆搜方案内（在这里就是对应了上述的第一步所展开的爆搜方

案中）。

因此我们只需要以首个字符为起点，枚举以其开头所有的回文串方案，加入集合，然后对剩下的字符串部分继续爆搜。就能做到以任意字符作为回文串起点进行分割的效果了。

一定要好好理解上面那句话 ~

剩下的问题是，我们如何快速判断连续一段 `[i, j]` 是否为回文串，因为爆搜的过程每个位置都可以作为分割点，复杂度为 $O(2^n)$ 的。

因此我们不可能每次都使用双指针去线性扫描一遍 `[i, j]` 判断是否回文。

一个直观的做法是，我们先预处理除所有的 `f[i][j]`，`f[i][j]` 代表 `[i, j]` 这一段是否为回文串。

预处理 `f[i][j]` 的过程可以用递推去做。

要想 `f[i][j] == true` ，必须满足以下两个条件：

1. `f[i + 1][j - 1] == true`
2. `s[i] == s[j]`

由于状态 `f[i][j]` 依赖于状态 `f[i + 1][j - 1]`，因此需要我们左端点 `i` 是**从大到小**进行遍历；而右端点 `j` 是**从小到大**进行遍历。

因此，我们的遍历过程可以整理为：**右端点 `j` 一直往右移动（从小到大），在 `j` 固定情况下，左端点 `i` 在 `j` 在左边开始，一直往左移动（从大到小）**

代码：

```java
class Solution {
    public List<List<String>> partition(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();
        // f[i][j] 代表 [i, j] 这一段是否为回文串
        boolean[][] f = new boolean[n][n];
        for (int j = 0; j < n; j++) {
            for (int i = j; i >= 0; i--) {
                // 当 [i, j] 只有一个字符时，必然是回文串
                if (i == j) {
                    f[i][j] = true;
                } else {
                    // 当 [i, j] 长度为 2 时，满足 cs[i] == cs[j] 即回文串
                    if (j - i + 1 == 2) {
                        f[i][j] = cs[i] == cs[j];

                    // 当 [i, j] 长度大于 2 时，满足 (cs[i] == cs[j] && f[i + 1][j - 1]) 即回
                    } else {
                        f[i][j] = cs[i] == cs[j] && f[i + 1][j - 1];
                    }
                }
            }
        }
        List<List<String>> ans = new ArrayList<>();
        List<String> cur = new ArrayList<>();
        dfs(s, 0, ans, cur, f);
        return ans;
    }
    /**
     * s: 要搜索的字符串
     * u: 以 s 中的那一位作为回文串分割起点
     * ans: 最终结果集
     * cur: 当前结果集
     * f: 快速判断 [i,j] 是否为回文串
     */
    void dfs(String s, int u, List<List<String>> ans, List<String> cur, boolean[][] f) {
        int n = s.length();
        if (u == n) ans.add(new ArrayList<>(cur));
        for (int i = u; i < n; i++) {
            if (f[u][i]) {
                cur.add(s.substring(u, i + 1));
                dfs(s, i + 1, ans, cur, f);
                cur.remove(cur.size() - 1);
            }
        }
    }
}
```

```
}
```

- 时间复杂度：动态规划预处理的复杂度为 $O(n^2)$；爆搜过程中每个字符都可以作为分割点，并且有分割与不分割两种选择，方案数量为 2^{n-1}，每个字符都需要往后检查剩余字符的分割情况，复杂度为 $O(n)$。整体复杂度为 $O(n * 2^n)$
- 空间复杂度：动态规划部分的复杂度为 $O(n^2)$；方案数量最多为 2^{n-1}，每个方案都是完整字符串 `s` 的分割，复杂度为 $O(n)$，整体复杂度为 $O(n * 2^n)$

总结

对于此类要枚举所有方案的题目，我们都应该先想到「回溯算法」。

「回溯算法」从算法定义上来说，不一定要用 DFS 实现，但通常结合 DFS 来做，难度是最低的。

「回溯算法」根据当前决策有多少种选择，对应了两套模板。

每一次独立的决策只对应 选择 和 不选 两种情况：

1. 确定结束回溯过程的 base case
2. 遍历每个位置，对每个位置进行决策（做选择 -> 递归 -> 撤销选择）

```
void dfs(当前位置, 路径(当前结果), 结果集) {
    if (当前位置 == 结束位置) {
        结果集.add(路径);
        return;
    }

    选择当前位置;
    dfs(下一位置, 路径(当前结果), 结果集);
    撤销选择当前位置;
    dfs(下一位置, 路径(当前结果), 结果集);
}
```

每一次独立的决策都对应了多种选择（通常对应了每次决策能选择什么，或者每次决策能选择多少个 ...）：

1. 确定结束回溯过程的 base case

2. 遍历所有的「选择」
3. 对选择进行决策 (做选择 -> 递归 -> 撤销选择)

```
void dfs(选择列表, 路径(当前结果), 结果集) {
    if (满足结束条件) {
        结果集.add(路径);
        return;
    }

    for (选择 in 选择列表) {
        做选择;
        dfs(路径', 选择列表, 结果集);
        撤销选择;
    }
}
```

拓展

刚好最近在更新「回溯算法」的相关题解，以下题目可以加深你对「回溯」算法的理解和模板的运用：

17. 电话号码的字母组合(中等)：从一道「回溯算法」经典题与你分享回溯算法的基本套路

39. 组合总和(中等)：DFS + 回溯算法，以及如何确定一道题是否应该使用 DFS + 回溯来求解

40. 组合总和 II(中等)：【回溯算法】求目标和的组合方案（升级篇）

216. 组合总和 III(中等)：【回溯算法】借助最后一道「组合总和」问题来总结一下回溯算法

37. 解数独(困难)：【数独问题】经典面试题：解数独

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **132. 分割回文串 II** ，难度为 **困难**。

Tag :「回文串」、「线性 DP」

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。

返回符合要求的 最少分割次数 。

示例 1：

```
输入：s = "aab"
输出：1
解释：只需一次分割就可将 s 分割成 ["aa","b"] 这样两个回文子串。
```

示例 2：

```
输入：s = "a"
输出：0
```

示例 3：

```
输入：s = "ab"
输出：1
```

提示：

- 1 <= s.length <= 2000
- s 仅由小写英文字母组成

动态规划

如果在 131. 分割回文串 你有使用到 DP 进行预处理的话。

这道题就很简单了，就是一道常规的动态规划题。

为了方便，我们约定所有下标从 1 开始。

即对于长度为 n 的字符串，我们使用 $[1, n]$ 进行表示。估计不少看过三叶题解的同学都知道，这样做的目的是为了减少边界情况判断，这本身也是对于「哨兵」思想的运用。

- **递推「最小分割次数」思路**

我们定义 $f[r]$ 为将 $[1, r]$ 这一段字符分割为若干回文串的最小分割次数，那么最终答案为 $f[n]$ 。

不失一般性的考虑 $f[r]$ 如何转移：

1. 从「起点字符」到「第 r 个字符」能形成回文串。那么最小分割次数为 0，此时有 $f[r] = 0$
2. 从「起点字符」到「第 r 个字符」不能形成回文串。此时我们需要枚举左端点 l，如果 $[l, r]$ 这一段是回文串的话，那么有 $f[r] = f[l-1] + 1$

在 2 中满足回文要求的左端点位置 l 可能有很多个，我们在所有方案中取一个 $\min$ 即可。

- **快速判断「任意一段子串是否回文」思路**

剩下的问题是，我们如何快速判断连续一段 $[l, r]$ 是否为回文串，做法和昨天的 131. 分割回文串 一模一样。

PS. 在 131. 分割回文串，数据范围只有 16，因此我们可以不使用 DP 进行预处理，而是使用双指针来判断是否回文也能过。但是该题数据范围为 2000（数量级为 10^3），使用朴素做法判断是否回文的话，复杂度会去到 $O(n^3)$（计算量为 10^9），必然超时。

因此我们不可能每次都使用双指针去线性扫描一遍 $[l, r]$ 判断是否回文。

一个合理的做法是，我们先预处理出所有的 $g[l][r]$，$g[l][r]$ 代表 $[l, r]$ 这一段是否为回文串。

预处理 $g[l][r]$ 的过程可以用递推去做。

要想 $g[l][r] = true$ ，必须满足以下两个条件：

1. $g[l+1][r-1] = true$
2. $s[i] = s[j]$

由于状态 $f[l][r]$ 依赖于状态 $f[l+1][r-1]$，因此需要我们左端点 l 是「从大到小」进行遍历；而右端点 r 是「从小到大」进行遍历。

因此最终的遍历过程可以整理为：**右端点 r 一直往右移动（从小到大），在 r 固定情况下，左端点 l 在 r 在左边开始，一直往左移动（从大到小）**

代码：

```java
class Solution {
    public int minCut(String s) {
        int n = s.length();
        char[] cs = s.toCharArray();

        // g[l][r] 代表 [l,r] 这一段是否为回文串
        boolean[][] g = new boolean[n + 1][n + 1];
        for (int r = 1; r <= n; r++) {
            for (int l = r; l >= 1; l--) {
                // 如果只有一个字符，则[l,r]属于回文
                if (l == r) {
                    g[l][r] = true;
                } else {
                    // 在 l 和 r 字符相同的前提下
                    if (cs[l - 1] == cs[r - 1]) {
                        // 如果 l 和 r 长度只有 2；或者 [l+1,r-1] 这一段满足回文，则[l,r]属于回文
                        if (r - l == 1 || g[l + 1][r - 1]) {
                            g[l][r] = true;
                        }
                    }
                }
            }
        }

        // f[r] 代表将 [1,r] 这一段分割成若干回文子串所需要的最小分割次数
        int[] f = new int[n + 1];
        for (int r = 1; r <= n; r++) {
            // 如果 [1,r] 满足回文，不需要分割
            if (g[1][r]) {
                f[r] = 0;
            } else {
                // 先设定一个最大分割次数（r 个字符最多消耗 r - 1 次分割）
                f[r] = r - 1;
                // 在所有符合 [l,r] 回文的方案中取最小值
                for (int l = 1; l <= r; l++) {
                    if (g[l][r]) f[r] = Math.min(f[r], f[l - 1] + 1);
                }
            }
        }

        return f[n];
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n^2)$

关于「如何确定 DP 状态定义」的分享

有同学会对「如何确定 DP 的状态定义」有疑问，觉得自己总是定不下 DP 的状态定义。

首先，十分正常，不用担心。

DP 的状态定义，基本上是考经验的（猜的），猜对了 DP 的状态定义，基本上「状态转移方程」就是呼之欲出。

虽然大多数情况都是猜的，但也不是毫无规律，相当一部分是定义是与「结尾」和「答案」有所关联的。

例如本题定义 $f[i]$ 为以下标为 i 的字符作为结尾（结尾）的最小分割次数（答案）。

因此对于那些你没见过的 DP 模型题，可以从这两方面去「猜」。

Manacher 算法（非重要补充）

如果你还学有余力的话，可以看看下面这篇题解。

提供了「回文串」问题的究极答案：Manacher 算法。

由于 Manacher 算法较为局限，只能解决「回文串」问题，远不如 KMP 算法使用广泛，**不建议大家深究原理，而是直接当做「模板」背过。**

背过这样的算法的意义在于：相当于大脑里有了一个时间复杂度为 $O(n)$ 的 api 可以使用，这个 api 传入一个字符串，返回该字符串的最大回文子串。

回文串问题的究极答案：Manacher 算法

如果觉得自己背不下来，也没有问题。事实上我还没有见过必须使用 Manacher 算法才能过的回文串题。

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

💡更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「回文串问题」获取下载链接。

觉得专题不错，可以请作者吃糖🍭🍭🍭：