

string

deepwaterooo

2021 年 11 月 3 日



# 目录

<b>1 字符串</b>	<b>5</b>
1.1 总结	5
1.2 KMP 算法	5
1.3 1392. Longest Happy Prefix - Hard : Use Longest Prefix Suffix (KMP-table) or String Hashing.	5
1.3.1 解题思路与分析: KMP 算法	5
1.3.2 rolling hash	5
1.4 1910. Remove All Occurrences of a Substring - Medium 可用 KMP 算法	6
1.5 467. Unique Substrings in Wraparound String - Medium Rabin-Karp Rolling Hash 算法	6
1.6 1044. Longest Duplicate Substring - Hard	7
1.7 1156. Swap For Longest Repeated Character Substring - Medium	8
1.8 395. Longest Substring with At Least K Repeating Characters - Medium	10
1.9 1830. Minimum Number of Operations to Make String Sorted - Hard 排列组合费小马快速幂	11
1.9.1 解题思路与分析	11
1.10 1960. Maximum Product of the Length of Two Palindromic Substrings - Hard 马拉车算法 todo	12
1.10.1 解题思路与分析	12
1.10.2 DP 解: 要再好好理解消化一下	12



# Chapter 1

## 字符串

### 1.1 总结

- Rabin-Karp Rolling Hash 算法

### 1.2 KMP 算法

### 1.3 1392. Longest Happy Prefix - Hard : Use Longest Prefix Suffix (KMP-table) or String Hashing.

A string is called a happy prefix if is a non-empty prefix which is also a suffix (excluding itself).  
Given a string  $s$ , return the longest happy prefix of  $s$ . Return an empty string "" if no such prefix exists.

#### 1.3.1 解题思路与分析: KMP 算法

给定一个字符串，其长度为 $k$ 的前缀和长度为 $k$ 后缀有可能相等，问最大的 $k$ 对应的那个前缀是什么。

题目本质上是求KMP算法中的next数组（KMP有个原始版本和优化版本，这里指的是原始版本。对于原始版本， $n[i]$ 表示 $s[0:i-1]$ 中最长相等前后缀的长度，规定 $n[0] = -1$ ）。

我们考虑 $n[i]$ 和 $n[i-1]$ 的递推关系。

首先 $n[1] = 0$ ，我们是可以确定的。对于 $n[i-1]$ ，假设 $n[i-1] = t$ ，意味着 $s[0:t-1] = s[i-t-1:i-2]$ ：

1、若 $s[i-1] = s[t]$ ，那么就有 $s[0:t] = s[i-t-1:i-1]$ ，所以 $n[i] \geq t+1$ 。但如果 $n[i] > t+1$ 的话，就会导致 $s[0:t+1] = s[i-t-2:i-1]$ ，从而有 $s[0:t] = s[i-t-2:i-2]$ ，这与 $n[i-1] = t$ 矛盾了，所以有 $n[i] = t+1$ 。

2、若 $s[i-1] \neq s[t]$ ，根据next数组的定义， $s[i-1]$ 要继续和 $s[n[t]]$ 进行比较，如果不等，则继续和 $s[n[n[t]]]$ 进行比较，如此这样下去，直到与 $s[i-1] = s[n^k[t]]$ 相等为止，相等后， $n[i] = n^k[t] + 1$ ；若一直都不等，则 $n[i] = n[0] + 1 = -1 + 1 = 0$ ，也是对的。证明与1类似。代码如下：

---

```
public String longestPrefix(String ss) {
    int n = ss.length();
    char [] s = ss.toCharArray();
    int [] lps = new int [n];
    for (int i = 1, j = 0; i < n; i++) {
        while (j > 0 && s[i] != s[j])
            j = lps[j-1];
        if (s[i] == s[j])
            lps[i] = ++j;
    }
    return ss.substring(0, lps[n-1]);
}
```

---

#### 1.3.2 rolling hash

Time complexity:  $O(n)$  / worst case:  $O(n^2)$

Space complexity:  $O(1)$

---

```
public String longestPrefix(String s) { // 容易出错, KMP 写起来比较简单
    int n = s.length(), hashPre = 0, hashSuf = 0;
    int left = 0, right = n-1, pow = 1, maxLen = 0;
    String res = "";
    while (left < n-1) {
        hashPre = hashPre * 31 + s.charAt(left);
        hashSuf = hashSuf + s.charAt(right)*pow;
        if (hashPre == hashSuf) maxLen = left + 1;
        left ++;
        right --;
        pow *= 31;
    }
    return maxLen == 0 ? "" : s.substring(0, maxLen);
}
```

---

## 1.4 1910. Remove All Occurrences of a Substring - Medium 可用 KMP 算法

Given two strings *s* and *part*, perform the following operation on *s* until all occurrences of the substring *part* are removed:

Find the leftmost occurrence of the substring *part* and remove it from *s*. Return *s* after removing all occurrences of *part*.

A substring is a contiguous sequence of characters in a string.

---

```
public String removeOccurrences(String s, String part) {
    if (!s.contains(part)) return s;
    int n = s.length();
    int m = part.length();
    while (s.contains(part)) {
        int idx = s.indexOf(part);
        s = s.substring(0, idx) + (idx+m-1 == n-1 ? "" : s.substring(idx+m));
    }
    return s;
}
```

---

- 有人用了 KMP: 字符串匹配可以用 KMP 算法, 由于 *p* *pp* 始终不变, 可以先算一下 *p* *pp* 的 next 数组, 然后每次从 *s* *ss* 中找 *p* *pp* 的第一次出现, 删去之, 再重复进行这个过程

---

```
private int [] buildNext(String s) { // 找与每个位置字符不同的下一个字母的 idx
    int [] next = new int [s.length()];
    for (int i = 0, j = next[0] = -1; i < s.length()-1; ) {
        if (j == -1 || s.charAt(i) == s.charAt(j)) {
            i++;
            j++;
            next[i] = s.charAt(i) != s.charAt(j) ? j : next[j];
        } else j = next[j];
    }
    return next;
}
private int kmp(String s, String p, int [] next) { // 像是夹生饭, 半生不熟的
    for (int i = 0, j = 0; i < s.length(); ) {
        if (j == -1 || s.charAt(i) == p.charAt(j)) {
            i++;
            j++;
        } else j = next[j];
        if (j == p.length()) return i-j;
    }
    return -1;
}
public String removeOccurrences(String s, String part) {
    int [] next = buildNext(part);
    int idx = -1;
    while ((idx = kmp(s, part, next)) != -1)
        s = s.substring(0, idx) + s.substring(idx + part.length());
    return s;
}
```

---

## 1.5 467. Unique Substrings in Wraparound String - Medium Rabin-Karp Rolling Hash 算法

We define the string *s* to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so *s* will look like this:

"...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd...". Given a string  $p$ , return the number of unique non-empty substrings of  $p$  are present in  $s$ .

这道题说有一个无限长的封装字符串，然后又给了我们另一个字符串  $p$ ，问我们  $p$  有多少非空子字符串在封装字符串中。我们通过观察题目中的例子可以发现，由于封装字符串是 26 个字符按顺序无限循环组成的，那么满足题意的  $p$  的子字符串要么是单一的字符，要么是按字母顺序的子字符串。这道题遍历  $p$  的所有子字符串会 TLE，因为如果  $p$  很大的话，子字符串很多，会有大量的满足题意的重复子字符串，必须要用到 **trick**，而所谓技巧就是一般来说你想不到的方法。我们看  $abcd$  这个字符串，以  $d$  结尾的子字符串有  $abcd$ ,  $bcd$ ,  $cd$ ,  $d$ ，那么我们可以发现  $bcd$  或者  $cd$  这些以  $d$  结尾的字符串的子字符串都包含在  $abcd$  中，那么我们知道以某个字符结束的最大字符串包含其他以该字符结束的字符串的所有子字符串，说起来很拗口，但是理解了我上面举的例子就行。那么题目就可以转换为分别求出以每个字符 ( $a-z$ ) 为结束字符的最长连续字符串就行了，我们用一个数组  $cnt$  记录下来，最后在求出数组  $cnt$  的所有数字之和就是我们要的结果啦，

---

```
public int findSubstringInWraproundString(String p) {
    int n = p.length();
    int[] arr = new int[n];
    int[] cnt = new int[26];
    for (int i = 0; i < n; i++)
        arr[i] = p.charAt(i) - 'a';
    int res = 0, maxLen = 0;
    for (int i = 0; i < n; i++) {
        if (i > 0 && (arr[i-1]+1) % 26 == arr[i]) // 判断前一个位置上的字符比现位字符小 1
            ++maxLen;
        else maxLen = 1;
        cnt[arr[i]] = Math.max(cnt[arr[i]], maxLen);
    }
    for (int i = 0; i < 26; i++)
        res += cnt[i];
    return res;
}
```

---

## 1.6 1044. Longest Duplicate Substring - Hard

Given a string  $s$ , consider all duplicated substrings: (contiguous) substrings of  $s$  that occur 2 or more times. The occurrences may overlap.

Return any duplicated substring that has the longest possible length. If  $s$  does not have a duplicated substring, the answer is "".

---

```
public static int base = 26; // 256
public static int mod = (1 << 31) - 1;
public static boolean match(String str1, String str2) {
    assert str1.length() == str2.length();
    for (int i = 0; i < str1.length(); i++)
        if (str1.charAt(i) != str2.charAt(i))
            return false;
    return true;
}
private String search(String s, int v) { // v: substring length
    int n = s.length();
    long hash = 0L, mp = 1L; // to avoid overflow, long long long
    Map<Long, List<Integer>> map = new HashMap<>();
    for (int j = 0; j < v; j++) {
        hash = ((hash * base) + s.charAt(j)) % mod;
        if (j >= 1)
            mp = mp * base % mod; // 先乘好准备好，准备着备用
    }
    map.computeIfAbsent(hash, k -> new ArrayList<>()).add(0);
    for (int i = 1; i + v <= n; i++) {
        hash = ((hash - s.charAt(i-1)) * mp % mod + mod) % mod * base % mod + s.charAt(i+v-1) % mod; // mod
        if (map.containsKey(hash))
            for (int idx : map.get(hash))
                if (match(s.substring(i, i+v), s.substring(idx, idx+v)))
                    return s.substring(i, i+v);
        map.computeIfAbsent(hash, k -> new ArrayList<>()).add(i);
    }
    return null;
}
public String longestDupSubstring(String s) {
    int n = s.length();
    int l = 0, r = n;
    String res = "";
    while (l <= r) {
        int m = l + (r - l) / 2;
        String tmp = search(s, m);
        if (tmp == null) r = m - 1;
        else {
            res = tmp;
            l = m;
        }
    }
    return res;
}
```

```

        if (tmp.length() > res.length())
            res = tmp;
        l = m+1;
    }
}
return res;
}

```

## 1.7 1156. Swap For Longest Repeated Character Substring - Medium

You are given a string text. You can swap two of the characters in the text.

Return the length of the longest substring with repeated characters.

给你一个字符串，如何找最长的重复子串，博主会数连续相同的字符，若此时有一个不同字符出现了，只要后面还有相同的字符，就会继续数下去，因为有一次交换的机会，什么时候停止呢，当再次出现不同字符的时候就停止，或者是当前统计个数等于该字符出现的总个数时也停止，因为得到的结果不可能超过某个字符出现的总个数。所以可以先统计每个字符的出现次数，然后开始遍历字符，对于每个遍历到的字符，都开始数之后跟其相等的字符，新建变量  $j$ ， $cnt$ ，和  $diff$ ，当  $j$  小于  $n$ ，且当前字符和比较字符相同，或者  $diff$  等于 0，且  $cnt$  小于比较字符出现的总个数时进行遍历，若当前遍历到的字符和要比较的字符不相等，说明该使用交换操作了， $diff$  自增 1，此时将  $i$  更新为  $j-1$ ，这是一个优化操作，可以避免一些不必要的计算，下次从这个位置往后统计，也相当于重置了  $diff$ 。还有就是这个  $cnt$  小于字符出现总个数这个条件卡的非常好，即便下一个还是相同字符，也不能再统计了，因为最后的这个相同字符可能是要用来交换前面的断点位置的。每次用统计出来的  $cnt$  更新结果  $res$ ，但是一个方向的遍历可能无法应对所有情况，比如 "acbaaa"，若只是从前往后遍历，那么最终只能得到 3，而正确的答案是 4，因为可以将  $b$  和第一个  $a$  交换，所以还需要从后往前进行一次相同的操作，这样才能得到正确的答案，参见代码如下：

```

public int maxRepOpt1(String s) { // O(n^2)
    int n = s.length(), ans = 0;
    Map<Character, Integer> charCnt = new HashMap<>();
    for (char c : s.toCharArray())
        charCnt.put(c, charCnt.getOrDefault(c, 0) + 1);
    for (int i = 0; i < n; i++) {
        char cur = s.charAt(i);
        int j = i, cnt = 0, dif = 0;
        while (j < n && (cur == s.charAt(j) || dif == 0) && cnt < charCnt.get(cur)) {
            if (cur != s.charAt(j)) {
                ++dif;
                i = j-1; // exchanged once, i moves to be the repeated sequence tail
            }
            ++cnt;
            ++j;
        }
        ans = Math.max(ans, cnt);
    }
    for (int i = n-1; i >= 0; i--) {
        char cur = s.charAt(i);
        int j = i, cnt = 0, dif = 0;
        while (j >= 0 && (cur == s.charAt(j) || dif == 0) && cnt < charCnt.get(cur)) {
            if (cur != s.charAt(j)) {
                ++dif;
                i = j+1;
            }
            ++cnt;
            --j;
        }
        ans = Math.max(ans, cnt);
    }
    return res;
}

```

- O(N) 解法

上面的解法严格来说还是平方级的，再来看一种线性时间的解法，可能比较难想，由于这里需要关注的是相同字符的出现位置，所以可以将所有相同的字符的位置都放到一个数组中，那么这里就建立一个字符和其出现位置数组之间的映射。由于题目中限制了只有英文字母，所以可以按照每个字母进行遍历，直接遍历每个字符的位置数组，这里新建变量  $cnt$ ， $cnt2$ ，和  $mx$ ，其中  $cnt$  统计的是连续字母的个数， $cnt2$  相当于一个临时变量，当使用交换操作时，保存之前的  $cnt$  值， $mx$  为二者之和。在遍历完某个字母位置数组之后，最后看一下若该字母出现总个数大于  $mx$ ，则说明交换后的字母还没有统计进去，不管之前有没有使用交换操作，都需要加上这个额外的一个，参见代码如下：

```

public int maxRepOpt1(String s) { // O(n^2)
    int n = s.length(), ans = 0;
    Map<Character, List<Integer>> idxMap = new HashMap<>();
    for (int i = 0; i < n; i++)

```



```

idxMap.computeIfAbsent(s.charAt(i), k -> new ArrayList<>()).add(i);
for (char c = 'a'; c <= 'z'; c++) {
    if (!idxMap.containsKey(c)) continue;
    int cnt = 1, cntb = 0, max = 0;
    List<Integer> idxs = idxMap.get(c);
    for (int i = 1; i < idxs.size(); i++) {
        if (idxs.get(i) == idxs.get(i-1) + 1) // aa
            ++cnt;
        else {
            cntb = (idxs.get(i) == idxs.get(i-1) + 2) ? cnt : 0; // aba ?
            cnt = 1;
        }
        max = Math.max(max, cnt + cntb);
    }
    ans = Math.max(ans, max + (idxs.size() > max ? 1 : 0)); // aaaaabaaaaaca 多于两个重复子段，中间替换字符可是是相同的
}
return ans;
}

```

## 算法

### (动态规划) $O(n)$

1. 我们分别考察每种字符所能得到的最大答案。
2. 假设我们当前考察的是字符 `c`。我们通过遍历一次数组的方式，找到单字符就是 `c` 的情况下的最长单重复子串。
3. 设 `cnt` 为字符串中 `c` 出现的次数； $f(i)$  表示以  $i$  结尾且没有替换过不是 `c` 的字符时的最长但重复子串； $g(i)$  表示以  $i$  结尾且替换过不是 `c` 的字符时的最长但重复子串。
4. 当 `c == text[i]` 时，累加 `cnt`， $f(i) = f(i-1) + 1$ ， $g(i) = g(i-1) + 1$ ，也就是都可以沿着上一次的结果往后累加 1 的长度。
5. 如果 `c != text[i]`，则  $g(i) = f(i-1) + 1$ ，表示我们强制替换 `text[i]` 的字符为 `c`（注意不是交换），所以要从没有替换过的  $f$  数组转移； $f(i) = 0$ ，表示如果不替换，那么长度只能归 0。交换只是替换的一种特殊形式，我们都先假设总能交换。如果不能交换，如 `aaabaaa` 我们无法把中间的 `b` 用一个新的 `a` 交换时，可以通过答案不能超过 `cnt` 来特判。
6. 答案为  $\min(cnt, \max(f(i), g(i)))$ ，也就是所有的  $f$  和  $g$  的最大值，但结果不能超过 `cnt`。
7. 这里的  $f$  和  $g$  数组因为之和上一位有关，所以可以化简为变量。

### 时间复杂度

- 枚举每种字符时，只需要遍历数组一次，故时间复杂度为  $O(n)$ 。

### 空间复杂度

- 动态规划
- 优化后，只需要常数个变量，故空间复杂度为  $O(1)$ 。

```

private int solve(char c, String s) {
    int n = s.length(), max = 0;
    int f = 0, g = 0, cnt = 0;
    for (int i = 0; i < n; i++) {
        if (c == s.charAt(i)) {
            f++;
            g++;
            cnt++;
        } else {
            g = f + 1;
            f = 0;
        }
        max = Math.max(max, Math.max(f, g));
    }
    return Math.min(max, cnt);
}

public int maxRepOpt1(String s) {
    int n = s.length(), ans = 0;
    for (char i = 'a'; i <= 'z'; i++)
        ans = Math.max(ans, solve(i, s));
    return ans;
}

```

## 1.8 395. Longest Substring with At Least K Repeating Characters - Medium

Given a string *s* and an integer *k*, return the length of the longest substring of *s* such that the frequency of each character in this substring is greater than or equal to *k*.

---

```
// 由于字母只有 26 个，而整型 mask 有 32 位，足够用了，
// 每一位代表一个字母，如果为 1，表示该字母不够 k 次，如果为 0 就表示已经出现了 k 次，这种思路真是大聪明了，
// 隐约记得这种用法在之前的题目中也用过，但是博主并不能举一反三（沮丧脸：(），还得继续努力啊。
// 遍历字符串，对于每一个字符，都将其视为起点，然后遍历到末尾，增加 HashMap 中字母的出现次数，如果其小于 k，将 mask 的对应位改为 1，如果大于等于 k，将 mask 对应位改
// 然后看 mask 是否为 0。是的话就更新 res 结果，然后把当前满足要求的子字符串的起始位置 j 保存到 max_idx 中，等内层循环结束后，将外层循环变量 i 赋值为 max_idx+1，继续
public int longestSubstring(String s, int k) { // O(N^2)
    int n = s.length(), res = 0, i = 0;
    while (i + k <= n) {
        int[] m = new int[26];
        int mask = 0, maxIdx = i;
        for (int j = i; j < n; j++) {
            int t = s.charAt(j) - 'a';
            m[t]++;
            if (m[t] < k) mask |= (1 << t);
            else mask &= ~(1 << t);
            if (mask == 0) {
                res = Math.max(res, j - i + 1);
                maxIdx = j;
            }
        }
        i = maxIdx + 1;
    }
    return res;
}
```

---

- 双指针 sliding window O(N)

---

```
public int longestSubstring(String s, int k) {
    int n = s.length(), res = 0;
    for (int cnt = 1; cnt <= 26; cnt++) {
        int start = 0, i = 0, uniqueCnt = 0;
        int[] charCnt = new int[26];
        while (i < n) {
            boolean valid = true;
            if (charCnt[s.charAt(i++) - 'a']++ == 0) ++ uniqueCnt;
            while (uniqueCnt > cnt)
                if (--charCnt[s.charAt(start++) - 'a'] == 0) --uniqueCnt;
            for (int j = 0; j < 26; j++)
                if (charCnt[j] > 0 && charCnt[j] < k) valid = false;
            if (valid) res = Math.max(res, i - start);
        }
    }
    return res;
}
```

---

- 分治：分而治之

---

```
public int longestSubstring(String s, int k) { // str.split("[dkfldjff]")
    int n = s.length();
    if (n < k) return 0;
    if (n == k && s.chars().distinct().count() == 1) return k;
    int[] cnt = new int[26];
    for (int i = 0; i < n; i++)
        cnt[s.charAt(i) - 'a']++;
    if (Arrays.stream(cnt).max().getAsInt() < k) return 0;
    StringBuilder sb = new StringBuilder("");
    for (int i = 0; i < 26; i++)
        if (cnt[i] < k && cnt[i] != 0)
            sb.append((char)(i + 'a'));
    sb.append('');
    if (sb.length() == 2) return n;
    String[] sa = s.split(sb.toString()); // str.split("[-*=/]") pay attention to the format
    System.out.println(Arrays.toString(sa));
    int max = 0;
    for (int i = 0; i < sa.length; i++)
        max = Math.max(max, longestSubstring(sa[i], k));
    return max;
}

public int longestSubstring(String s, int k) { // 人工手动拆分
    int n = s.length(), maxIdx = 0, res = 0;
    int[] cnt = new int[128];
    boolean valid = true;
    for (char c : s.toCharArray())
        cnt[c]++;
}
```

```

for (int i = 0; i < n; i++)
    if (cnt[s.charAt(i)] < k) {
        res = Math.max(res, longestSubstring(s.substring(maxIdx, i), k));
        valid = false;
        maxIdx = i+1;
    }
return valid ? n : Math.max(res, longestSubstring(s.substring(maxIdx, n), k));
}

```

## 1.9 1830. Minimum Number of Operations to Make String Sorted - Hard

### 排列组合费小马快速幂

You are given a string  $s$  (0-indexed). You are asked to perform the following operation on  $s$  until you get a sorted string:

Find the largest index  $i$  such that  $1 \leq i < s.length$  and  $s[i] < s[i - 1]$ . Find the largest index  $j$  such that  $i \leq j < s.length$  and  $s[j] < s[i - 1]$  for all the possible values of  $k$  in the range  $[i, j]$  inclusive. Swap the two characters at indices  $i - 1$  and  $j$ . Reverse the suffix starting at index  $i$ . Return the number of operations needed to make the string sorted. Since the answer can be too large, return it modulo  $10^9 + 7$ .

#### 1.9.1 解题思路与分析

题中每次对字符串  $s$  执行的操作，是将其变为由当前字母组成的前一字典序的字符串。因此求最少操作次数，等价于求解该字符串在由当前字母组成的所有排列中的字典序；

求比当前字符串  $s$  小的排列个数，可通过排列组合公式计算得到；

排列组合公式中的阶乘逆元取模，可通过费马小定理，转化为对模数的乘方进行计算；

可通过快速乘方算法，进一步提高对乘方的计算效率。

```

private int quickmul(int base, int exp) { // 快速乘方算法
    long ans = 1L;
    while (exp > 0) {
        if ((exp & 1) == 1) // 指数是奇数，就先乘一次 base
            ans = ans * base % mod;
        base = (int)((long)base * base % mod); // 底平方 (指数变偶数之后)
        exp >>= 1; // 指数除 2，快速计算
    }
    return (int)ans;
}
int mod = (int)1e9 + 7;
public int makeStringSorted(String t) {
    int n = t.length();
    char[] s = t.toCharArray();
    int[] cnt = new int[26]; // 记录剩余字符串中各字母个数
    Arrays.fill(cnt, 0);
    for (int i = 0; i < n; i++)
        cnt[s[i] - 'a']++;
    int[] fact = new int[n+1];
    int[] finv = new int[n+1];
    fact[0] = 1;
    finv[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (int)((long)fact[i-1] * i % mod); // fac[i] = i! % mod
        finv[i] = quickmul(fact[i], mod - 2); // 费马小定理计算乘法逆元, finv[i] = (i!) ^ -1 % mod
    }
    long ans = 0L;
    for (int i = 0; i < n-1; i++) {
        int lessCnt = 0; // 比当前位置小的字母总数
        for (int j = 0; j < s[i] - 'a'; j++)
            lessCnt += cnt[j];
        long upper = (long)lessCnt * fact[n-1-i] % mod; // 排列公式分子
        for (int j = 0; j < 26; j++)
            upper = upper * finv[cnt[j]] % mod;
        ans = (ans + upper) % mod;
        cnt[s[i] - 'a']--; // 指针右移
    }
    return (int)ans;
}

```

## 1.10 1960. Maximum Product of the Length of Two Palindromic Substrings

### - Hard 马拉车算法 todo

You are given a 0-indexed string  $s$  and are tasked with finding two non-intersecting palindromic substrings of odd length such that the product of their lengths is maximized.

More formally, you want to choose four integers  $i, j, k, l$  such that  $0 \leq i \leq j < k \leq l < s.length$  and both the substrings  $s[i..j]$  and  $s[k..l]$  are palindromes and have odd lengths.  $s[i..j]$  denotes a substring from index  $i$  to index  $j$  inclusive.

Return the maximum possible product of the lengths of the two non-intersecting palindromic substrings.

A palindrome is a string that is the same forward and backward. A substring is a contiguous sequence of characters in a string.

#### 1.10.1 解题思路与分析

这个马拉车，好像还是很半生不熟，要好好理解消化一下

[https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/discuss/1393288/Java-100-0\(n\)-time-O\(n\)-space-using-Manacher's-algorithm](https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/discuss/1393288/Java-100-0(n)-time-O(n)-space-using-Manacher's-algorithm)

---

```
private static int[] manacherOdd(String str) {
    int n = str.length();
    char[] s = str.toCharArray();
    int[] ans = new int[n];
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int len = i > r ? 1 : Math.min(ans[l+r-i], r-i+1);
        int maxLen = Math.min(i, n-1-i);
        int x = i - len, y = i + len;
        while (len <= maxLen && s[x--] == s[y++]) len++;
        ans[i] = len--;
        if (i + len > r) {
            l = i - len;
            r = i + len;
        }
    }
    return ans;
}

public long maxProduct(String s) { // 这个马拉车，得多写几遍
    int n = s.length();
    int[] d = manacherOdd(s);
    int[] l = new int[n], r = new int[n];
    for (int i = 0; i < n; i++) {
        l[i+d[i]-1] = Math.max(l[i+d[i]-1], 2 * d[i]-1);
        r[i-d[i]+1] = 2 * d[i] - 1;
    }
    for (int i = n-2, j = n-1; i >= 0; i--, j--)
        l[i] = Math.max(l[i], l[j]-2);
    for (int i = 1, j = 0; i < n; i++, j++)
        r[i] = Math.max(r[i], r[j]-2);
    for (int i = 1, j = 0; i < n; i++, j++)
        l[i] = Math.max(l[i], l[j]);
    for (int i = n-2, j = n-1; i >= 0; i--, j--)
        r[i] = Math.max(r[i], r[j]);
    long ans = 1;
    for (int i = 1; i < n; i++)
        ans = Math.max(ans, (long)l[i-1] * r[i]);
    return ans;
}
```

---

#### 1.10.2 DP 解：要再好好理解消化一下