# string

deepwaterooo

October 9, 2024

# Contents

# Chapter 1

# 字符串

## 1.1 总结：几种【字符串算法】关键字

- Rabin-Karp Rolling Hash 算法、AC 自动机

## 1.2 KMP 算法

### 1.2.1 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. 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]);
}
```

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);
    }
```

### 1.2.2 214. Shortest Palindrome - Hard KMP 算法

You are given a string s. You can convert s to a palindrome by adding characters in front of it.
Return the shortest palindrome you can find by performing this transformation.

1. 解题思路与分析

```
public String shortestPalindrome(String s) {
    int n = s.length();
    int base = 131, mod = 1000000007;
    int left = 0, right = 0, mul = 1;
    int best = -1;
    for (int i = 0; i < n; ++i) {
        left = (int) (((long) left * base + s.charAt(i)) % mod);
        right = (int) ((right + (long) mul * s.charAt(i)) % mod);
        if (left == right) best = i;
        mul = (int) ((long) mul * base % mod);
    }
    String add = (best == n - 1 ? "" : s.substring(best + 1));
    StringBuffer ans = new StringBuffer(add).reverse();
    ans.append(s);
    return ans.toString();
}
```

2. 解题思路与分析

```
public String shortestPalindrome(String t) {
    // if (t.chars().distinct().count() == 1) return t;
    // if (isPalindrome(t)) return t; // 这种提前的判断也会耽误时间
    int n = t.length(), j = 0;
    char [] s = t.toCharArray();
    for (int i = n-1; i >= 0; i--) // 精髓就在这两行
        if (s[i] == s[j]) j++;     //
    if (j == n) return t;
    String suf = t.substring(j);
    System.out.println("suf: " + suf);
    return new StringBuilder (suf).reverse().toString() + shortestPalindrome(t.substring(0, j)) + suf;
}
```

3. 解题思路与分析: KMP 算法

```
public String shortestPalindrome(String t) { // 这个方法比上面的一个方法要慢
    String s = new StringBuilder (t).reverse().toString();
    s = t + "#" + s; // 原串 + “#” + 反转串
    int n = t.length(), m = s.length(), j = 0;
    int [] p = new int [m];
    for (int i = 1; i < m; i++) {
        int k = p[i-1];
        while (k > 0 && s.charAt(k) != s.charAt(i)) k = p[k-1];
        k += (s.charAt(i) == s.charAt(k) ? 1 : 0);
        p[i] = k;
    }
    return s.substring(n+1, 2 * n + 1 - p[m-1]) + t;
}
```

- 官方发布的写法

```
public String shortestPalindrome(String s) {
    int n = s.length();
    int[] fail = new int[n];
    Arrays.fill(fail, -1);
    for (int i = 1; i < n; ++i) {
        int j = fail[i - 1];
        while (j != -1 && s.charAt(j + 1) != s.charAt(i))
            j = fail[j];
        if (s.charAt(j + 1) == s.charAt(i))
            fail[i] = j + 1;
    }
    int best = -1;
    for (int i = n - 1; i >= 0; --i) {
        while (best != -1 && s.charAt(best + 1) != s.charAt(i))
            best = fail[best];
        if (s.charAt(best + 1) == s.charAt(i))
            ++best;
    }
```

```
        String add = (best == n - 1 ? "" : s.substring(best + 1));
        StringBuffer ans = new StringBuffer(add).reverse();
        ans.append(s);
        return ans.toString();
    }
```

### 1.2.3 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;
}
```

### 1.2.4 1316. Distinct Echo Substrings - Hard

Return the number of distinct non-empty substrings of text that can be written as the concatenation of some string with itself (i.e. it can be written as a + a where a is some string).

1. 解题思路与分析: 枚举 我们在 text 中枚举位置 i 和 j，若字符串 text[i:j] 和 text[j:j*2-i] 相等，那么字符串 text[i:j*2-i] 就是一个满足条件的子串，其中 text[x:y] 表示字符串 text 中以位置 x 开始，位置 y 结束并且不包含位置 y 的子串。

   由于题目需要求出不同的子串数目，因此我们还需要使用哈希集合（HashSet）对所有满足条件的子串进行去重操作。

```
    public int distinctEchoSubstrings(String t) {
        int n = t.length(), j = 0;
        Set<String> ss = new HashSet<>();
        for (int d = 1; d <= n/2; d++)
            for (int i = 0; i+d <= n-d; i++) {
                if (t.substring(i, i+d).equals(t.substring(i+d, i+d+d)))
                    ss.add(t.substring(i, i+d));
            }
        return ss.size();
    }
```

**复杂度分析**

- 时间复杂度：$O(N^3)$，其中 $N$ 是字符串 `text` 的长度。我们需要两重循环枚举位置 `i` 和 `j`，时间复杂度为 $O(N^2)$，而在比较字符串 `text[i:j]` 和 `text[j:j*2-i]` 时，最坏的时间复杂度为 $O(N)$，并且将字符串加入哈希集合中的时间复杂度也为 $O(N)$，因此总时间复杂度为 $O(N^3)$。但由于本题的测试数据较弱，它可以在规定时间内通过所有的测试数据。
- 空间复杂度：$O(N^2)$ 或 $O(N^3)$，在最坏情况下，哈希集合中会存储 $O(N^2)$ 个字符串，如果我们在哈希集合中存储的是字符串视图（例如 `C++` 中的 `std::string_view`），那么每个字符串视图使用的空间为 $O(1)$，总空间复杂度为 $O(N^2)$；如果我们在哈希集合中存储的是字符串本身（例如 `C++` 中的 `std::string` 和 `Python3` 中的 `str`），那么每个字符串使用的空间为 $O(N)$，总空间复杂度为 $O(N^3)$。

2. 解题思路与分析: 滚动哈希 + 前缀和

本方法需要一些关于「滚动哈希」或「Rabin-Karp 算法」的预备知识，其核心是将字符串看成一个 k 进制的整数，其中 k 是字符串中可能出现的字符种类，本题中字符串只包含小写字母，即 k = 26（也可以取比 k 大的整数，一般来说可以取一个质数，例如 29 或 31）。这样做的好处是绕开了字符串操作，将字符串看成整数进行比较，并可以在常数时间内将字符串加入哈希集合中。

关于「滚动哈希」或「Rabin-Karp 算法」的知识，可以参考 1044. 最长重复子串的官方题解或使用搜索引擎，这里对算法本身的流程不再赘述。

**使用滚动哈希**

我们仍然在 `text` 中枚举位置 `i` 和 `j`，并判断字符串 `text[i:j]` 和 `text[j:j*2-i]` 是否相等。但与方法一不同的是，我们比较这两个字符串的哈希值，而并非字符串本身。如果仅使用 Rabin-Karp 算法计算这两个字符串的哈希值，我们仍然需要使用 $O(N)$ 的时间，与直接比较字符串的时间复杂度相同。那么我们如何在更短的时间内计算出字符串的哈希值呢？

我们可以使用类似前缀和的方法。记 `prefix[i]` 表示字符串 `text[0:i]` 的哈希值。特别地，`prefix[0]` 的值为 `0`，为空串的哈希值。我们可以在 $O(N)$ 的时间内计算出数组 `prefix` 中的所有元素，即：

$$\text{prefix}[i] = \text{prefix}[i-1] * k + \text{text}[i]$$

当我们得到了前缀和数组 `prefix` 之后，如何计算任意字符串 `text[i:j]` 的哈希值呢？观察 `prefix[i]` 和 `prefix[j]` 这两项，它们的表达式为：

$$\begin{aligned}\text{prefix}[i] &= \text{text}[0] * k^{i-1} + \text{text}[1] * k^{i-2} + \cdots + \text{text}[i-1] \\ \text{prefix}[j] &= \text{text}[0] * k^{j-1} + \text{text}[1] * k^{j-2} + \cdots + \text{text}[j-1]\end{aligned}$$

如果将 `prefix[i]` 乘以 $k^{j-i}$，那么等式两侧会变为：

$$k^{j-i} * \text{prefix}[i] = \text{text}[0] * k^{j-1} + \text{text}[1] * k^{j-2} + \cdots + \text{text}[i-1] * k^{j-i}$$

将上式与 `prefix[j]` 的表达式相减，得到：

$$\text{prefix}[j] - k^{j-i} * \text{prefix}[i] = \text{text}[i] * k^{j-i-1} + \cdots + \text{text}[j-1]$$

与 `text[i:j]` 的哈希值相等，因此我们可以在 $O(1)$ 的时间计算出任意字符串 `text[i:j]` 的哈希值。在此之前，我们还需要预处理计算出所有 `k` 的次幂，不然在进行乘法操作时，仍然需要超过 $O(1)$ 的时间。

注意：上面的所有等式都省略了取模操作（如果你不知道为什么要取模，那么你对「Rabin-Karp 算法」的预备知识还没有掌握透彻），但在实际的代码编写中不能忽略，并且在取模的意义下，做减法时会产生负数。通用的写法如下所示，它同时考虑了取模和负数（在 `C++` 等语言中，还需要注意乘法可能产生的溢出）：

$$\text{hash\_value}(\text{text}[i:j]) = (\text{prefix}[j] - k^{j-i} * \text{prefix}[i] \% \text{mod} + \text{mod}) \% \text{mod}$$

回到我们原来的问题，当我们用字符串 `text[i:j]` 和 `text[j:j*2-i]` 的哈希值代替其本身判断是否相等后，如果两者相等，字符串 `text[i:j*2-i]` 就是一个满足条件的子串。但我们还需要进行去重操作，才能得到最终的答案。

在「判断」这一步中，由于我们只对两个字符串进行比较，因此引入哈希冲突（在下方的注意事项中也有提及）的概率极小。然而在「去重」这一步中，最坏情况下字符串的数量为 $O(N^2)$，大量的字符串造成哈希冲突的概率极大。为了减少字符串的数量以降低冲突的概率，我们可以使用 `N` 个哈希集合，分别存放不同长度的字符串，即第 `m` 个哈希集合存放长度为 `m + 1` 的字符串的哈希值。这样每个哈希集合只对某一固定长度的字符串进行去重操作，并且其中最多只有 `N` 个字符串，冲突概率非常低。

```
static final int mod = (int)1e9 + 7;
public int distinctEchoSubstrings(String t) {
    int n = t.length(), ans = 0;
    char [] s = t.toCharArray();
    int base = 31;
    int [] pre = new int [n+1], mul = new int [n+1];
    mul[0] = 1;
    for (int i = 1; i <= n; i++) {
        pre[i] = (int)(((long)pre[i-1] * base + s[i-1]) % mod);
        mul[i] = (int)((long)mul[i-1] * base % mod);
    }
    Set<Integer> [] vis = new HashSet[n];
    for (int i = 0; i < n; i++) vis[i] = new HashSet<>();
```

```
        for (int i = 0; i < n; i++)
            for (int j = i+1; j < n; j++) {
                int l = j - i;
                if (j + l <= n) {
                    int hash_left = gethash(pre, mul, i, j-1);
                    if (!vis[l-1].contains(hash_left) && hash_left == gethash(pre, mul, j, j+l-1)) {
                        ++ans;
                        vis[l-1].add(hash_left);
                    }
                }
            }
        return ans;
    }
    private int gethash(int [] pre, int [] mul, int l, int r) {
        return (int)((pre[r+1] - (long)pre[l] * mul[r-l+1] % mod + mod) % mod);
    }
```

- 注意事项

由于 Rabin-Karp 算法会将字符串对应的整数值进行取模，那么：

如果字符串 S1 和 S2 对应的整数值 I1 和 I2 不相等，那么 S1 和 S2 一定不相等；

如果字符串 S1 和 S2 对应的整数值 I1 和 I2 相等，并不代表 S1 和 S2 一定相等；

这与实际应用中使用的哈希算法也是一致的，即先判断两个实例的哈希值是否相等，再判断它们本质上是否相等。而在竞赛题目中，由于数据量较少，几乎不会产生哈希冲突，因此我们可以直接用 I1 和 I2 的相等代替 S1 和 S2 的相等，减少时间复杂度。但需要牢记在实际应用中，这样做是不严谨的。

### 1.2.5 467. 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;
}
```

### 1.2.6 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 {
            if (tmp.length() > res.length())
                res = tmp;
            l = m+1;
        }
    }
    return res;
}
```

### 1.2.7 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

- 优化后，只需要常数个变量，故空间复杂度为 $O(1)$。

- 动态规划

```
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;
}
```

### 1.2.8 395. 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，这种思路真是太聪明了，
//          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
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("[dkfldjf]")
    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) { // 人工手动折分
    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));
}
```

### 1.2.9 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 <= i < s.length and s[i] < s[i - 1]. Find the largest index j such that i <= j < s.length and s[k] < 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 109 + 7.

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, facinv[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;
    }
```

### 1.2.10 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 <= i <= j < k <= 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. 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-O(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;
}
```

2. DP 解：要再好好理解消化一下

### 1.2.11 854. K-Similar Strings - Hard

Strings s1 and s2 are k-similar (for some non-negative integer k) if we can swap the positions of two letters in s1 exactly k times so that the resulting string equals s2.

Given two anagrams s1 and s2, return the smallest k for which s1 and s2 are k-similar.

1. 解题思路与分析

```
public int kSimilarity(String s, String t) {
    if (s.equals(t)) return 0;
    ArrayDeque<String> q = new ArrayDeque<>();
    Set<String> vis = new HashSet<>();
    q.offerLast(s);
    vis.add(s);
    int cnt = 0;
    while (!q.isEmpty()) {
        for (int z = q.size()-1; z >= 0; z--) {
            String cur = q.pollFirst();
            // if (cur.equals(t)) return cnt;
            for (String next : getAllNeighbour(cur, t)) {
```

```
                if (vis.contains(next)) continue;
                if (next.equals(t)) return cnt + 1;
                q.offerLast(next);
                vis.add(next);
            }
        }
        cnt++;
    }
    return -1;
}
List<String> getAllNeighbour(String ss, String tt) {
    int n = ss.length(), i = 0;
    char [] s = ss.toCharArray();
    char [] t = tt.toCharArray();
    List<String> ans = new ArrayList<>();
    while (i < n && s[i] == t[i]) i++;
    for (int j = i+1; j < n; j++)
        if (s[j] == t[i]) {
            swap(i, j, s);
            ans.add(new String(s));
            swap(i, j, s);
        }
    return ans;
}
void swap(int i, int j, char [] s) {
    char c = s[i];
    s[i] = s[j];
    s[j] = c;
}
```

### 1.2.12 Queue ArrayDeque LinkedList 效率比较

1. 为什么 ArrayDeque 比 LinkedList 快

   了解完数据结构特点之后，接下来我们从两个方面分析为什么 ArrayDeque 作为队列使用时可能比 LinkedList 快。

   - 从速度的角度：ArrayDeque 基于数组实现双端队列，而 LinkedList 基于双向链表实现双端队列，数组采用连续的内存地址空间，通过下标索引访问，链表是非连续的内存地址空间，通过指针访问，所以在寻址方面数组的效率高于链表。
   - 从内存的角度：虽然 LinkedList 没有扩容的问题，但是插入元素的时候，需要创建一个 Node 对象，换句话说每次都要执行 new 操作，当执行 new 操作的时候，其过程是非常慢的，会经历两个过程：类加载过程、对象创建过程。
     - 类加载过程
       * 会先判断这个类是否已经初始化，如果没有初始化，会执行类的加载过程
       * 类的加载过程：加载、验证、准备、解析、初始化等等阶段，之后会执行 <clinit>() 方法，初始化静态变量，执行静态代码块等等
     - 对象创建过程
       * 如果类已经初始化了，直接执行对象的创建过程
       * 对象的创建过程：在堆内存中开辟一块空间，给开辟空间分配一个地址，之后执行初始化，会执行 <init>() 方法，初始化普通变量，调用普通代码块

| 集合类型 | 数据结构 | 初始化及扩容 | 插入/删除时间复杂度 | 查询时间复杂度 | 是否是线程安全 |
|---|---|---|---|---|---|
| ArrqyDeque | 循环数组 | 初始化：16 扩容：2 倍 | 0(n) | 0(1) | 否 |
| LinkedList | 双向链表 | 无 | 0(1) | 0(n) | 否 |

   - 最好的做法是自己用 arrayDeque 封装一个一端进出的 stack

## 1.3 AC 自动机：与 Trie 等数据结构的适用总结

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**
- 先前，亲爱的表哥的活宝妹，只是【机械】地参考答案，理解了答案，但感觉离消化理解透彻、转化为解题能力，还差一步归纳总结。多找几个类似题目，写三五个，消化掉知识点里点点滴滴的细节。把类似题型放一起

### 1.3.1 3292. Minimum Number of Valid Strings to Form Target II

- You are given an array of strings words and a string target.

A string x is called valid if x is a
prefix of any string in words.

Return the minimum number of valid strings that can be concatenated to form target. If it is not possible to form target, return -1.

```
// 【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】
public class Node {
    Node [] n;
    Node f; // 当 cur.son[i] 不能匹配 target 中的某个字符时，cur.fail.son[i] 即为下一个待匹配节点（等于 root 则表示没有匹配）
    int l; // l: length
    public Node(int l) {
        n = new Node [26];
        this.l = l;
    }
}
public class AhoCorasick {
    Node root = new Node(0);
    public void put(String s) {
        Node r = root;
        for (char j : s.toCharArray()) {
            j -= 'a';
            if (r.n[j] == null)
                r.n[j] = new Node(r.l + 1);
            r = r.n[j];
        }
    }
    public void buildFail() {
        Node r = root;
        r.f = r; // 【根节点】： f 指向自己，方便转圈【避开、不必要的、非空检测】???
        ArrayDeque<Node> q = new ArrayDeque<>();
        // 处理：【根节点】所有可能的子节点
        for (int i = 0; i < 26; i++) {
            Node s = r.n[i];
            if (s == null) // 它说，这里是，【虚拟指针】。。什么意思。。
                // 不存在，就指向父节点的，这个指针
                // r.n[i] = r.f.n[i]; // 【写错了】：根节点，不存在还是不存在；就是后来的子节点，能转什么吗 ???
                r.n[i] = r; // 根节点，不存在的子节点，就指向自己
            else {
                // s.f = r.f.n[i]; // 【写错了】：根节点 f 指向自己
                s.f = r;
                q.offerFirst(s);
            }
        }
        // 处理后代：跟【根节点】写法一模一样，为什么要写两遍？
        while (!q.isEmpty()) {
            r = q.pollLast(); // 全部、非空
            for (int i = 0; i < 26; i++) {
                Node s = r.n[i];
                if (s == null) // 它说，这里是，【虚拟指针】。。什么意思。。
                    // 虚拟子节点 cur.son[i]，和 cur.fail.son[i] 是同一个
                    // 方便失配时直接跳到下一个可能匹配的位置（但不一定是某个 words[k] 的最后一个字母）
                    r.n[i] = r.f.n[i]; // 当 r.f 非空，就指向了 r.f.n[i] 节点
                else {
                    // 下面：一次【赋值跳转】，但在【字典树】中，因为【自顶向下】的 f 指针构建
                    s.f = r.f.n[i];
                      q.offerFirst(s);
                  }
              }
          }
      }
    public int minValidStrings(String [] sa, String S) {
        int n = S.length(); char [] s = S.toCharArray();
          AhoCorasick ac = new AhoCorasick();
          for (String si : sa) ac.put(si);
          ac.buildFail();
          int [] f = new int [n+1];
          Node r = ac.root;
          for (int i = 0; i < n; i++) {
              // 下面：如果没有匹配，相当于移动到，父节点 r 的 fail 节点的 son[s[i]-'a']
              r = r.n[s[i] - 'a']; // 隐藏在幕后的、无数链接跳转。。。
              // 没有任何字符串的前缀与 target[..i] 的后缀匹配
              if (r == ac.root)
                  return -1;
              f[i+1] = f[i+1 - r.l] + 1;
          }
          return f[n];
      }
```

## 1.4 KMP 算法：细节

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**

```
// 【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】
// 最原始【暴力解法】: O(N^3)
static int[] prefix_function(String t) {
    int n = t.length(); char [] s = t.toCharArray();
    int [] f = new int [n];
    for (int i = 0; i < n; i++) { // 遍历：以【当前下标 i】为结尾
        for (int j = i; j >= 0; j--) // 贪心:【从最大可能长度、到偏小长度】遍历
            if (t.substring(0, j).equals(t.substring(i-(j-1), i+1))) {
                f[i] = j;
                break; // 找到一个：最长的，就不用再遍历了
            }
    }
    return f;
}
// 【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】
// 优化：原始优化，当前动规 f[i] 最大取值为 f[i]=f[i-1]+1,
// 建立在 s[i] = s[f[i-1]] 基础上，即【当前后缀、结尾下标字符 s[i] ，与对应前缀、最后字符 s[f[i-1]]】一样
// 经过此次优化，计算前缀函数只需要进行 O(n) 次字符串比较，总复杂度降为了 O(n^2)。
static int[] prefix_function(String t) {
    int n = t.length(); char [] s = t.toCharArray();
    int [] f = new int [n];
    for (int i = 1; i < n; i++) // 遍历：以【当前下标 i】为结尾
        for (int j = f[i-1]+1; j >= 0; j--) // 仅只优化：当前动规 f[i] 的最大取值、长度
            if (t.substring(0, j).equals(t.substring(i-(j-1), i+1))) {
                f[i] = j;
                break;
            }
    return f;
}
```

- **【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】**
- 最后一步：相对更本质的优化，就里截张图，自己推理不出来，能够记住别人的推理总结，也算掌握

> 在第一个优化中，我们讨论了计算 $\pi[i+1]$ 时的最好情况：$s[i+1] = s[\pi[i]]$，此时 $\pi[i+1] = \pi[i] + 1$。现在让我们沿着这个思路走得更远一点：讨论当 $s[i+1] \neq s[\pi[i]]$ 时如何跳转。
>
> 失配时，我们希望找到对于子串 $s[0 \dots i]$，仅次于 $\pi[i]$ 的第二长度 $j$，使得在位置 $i$ 的前缀性质仍得以保持，也即 $s[0 \dots j-1] = s[i-j+1 \dots i]$：
>
> $$\overbrace{\underbrace{s_0\ s_1}_{j}\ s_2\ s_3}^{\pi[i]} \dots \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{j}}^{\pi[i]}\ s_{i+1}$$
>
> 如果我们找到了这样的长度 $j$，那么仅需要再次比较 $s[i+1]$ 和 $s[j]$。如果它们相等，那么就有 $\pi[i+1] = j+1$。否则，我们需要找到子串 $s[0 \dots i]$ 仅次于 $j$ 的第二长度 $j^{(2)}$，使得前缀性质得以保持，如此反复，直到 $j = 0$。如果 $s[i+1] \neq s[0]$，则 $\pi[i+1] = 0$。
>
> 观察上图可以发现，因为 $s[0 \dots \pi[i]-1] = s[i-\pi[i]+1 \dots i]$，所以对于 $s[0 \dots i]$ 的第二长度 $j$，有这样的性质：
>
> $$s[0 \dots j-1] = s[i-j+1 \dots i] = s[\pi[i]-j \dots \pi[i]-1]$$
>
> 也就是说 $j$ 等价于子串 $s[\pi[i]-1]$ 的前缀函数值，即 $j = \pi[\pi[i]-1]$。同理，次于 $j$ 的第二长度等价于 $s[j-1]$ 的前缀函数值，$j^{(2)} = \pi[j-1]$
>
> 显然我们可以得到一个关于 $j$ 的状态转移方程：$j^{(n)} = \pi[j^{(n-1)}-1]$，$(j^{(n-1)} > 0)$

```
// 在第一个优化中，我们讨论了计算 f[i+1] 时的最好情况：s[i+1]=s[f[i]]，此时 f[i+1] = f[i]+1。
// 现在让我们沿着这个思路走得更远一点：讨论当 s[i+1] != s[f[i]] 时如何跳转 ???
// 再优化：原始优化，当前动规 f[i] 最大取值为 f[i]=f[i-1]+1,
// 建立在 s[i] = s[f[i-1]] 基础上，即【当前后缀、结尾下标字符 s[i] ，与对应前缀、最后字符 s[f[i-1]]】一样
static int[] prefix_function(String t) {
    int n = t.length(); char [] s = t.toCharArray();
    int [] f = new int [n];
    for (int i = 1; i < n; i++) { // 遍历：以【当前下标 i】为结尾
        // int j = f[i-1] + 1; //  【初始化】: 当前【后缀结尾下标 i】最大最长、可能的长度
        int j = f[i-1]; //  【初始化】: 当前【后缀结尾下标 i】【最大最长、可能的长度-1】: - 1 是为源码 s[j] 0-index 下标书写方便
```

```
    // 贪心：最长可能片段，前后缀，【最大长度】的、最后一字条不匹配，长度缩短至：
    // 贪心：此时，两个字符不匹配时，仍然，【最大长度】用 s[i] 试着去匹配——快速跳转的 s[j]??
        while (j > 0 && s[i] != s[j]) { // 连跳、快速跳。。。s[j-1] 是因为 0-index 字符串
            // 贪心：此时，2 字符不匹配时，仍然，【最大长度】: s[i]== s[f[j-1]]??? 这里还是迷糊的。。。
            //  【难点】：亲爱的表哥的活宝妹，自己、看不出、推不出、这样的结论。。只能去理解它人的总结
            j = f[j-1]; // 此时，最长可能长度是：长度缩短 1 或更多：j-1 下标下最大长度【感觉不对!!】f[j-1]+1
        }
        // 退出上面的循环后， s[i] = s[j] || j <= 0
        if (s[i] == s[j])
            j++;
        f[i] = j;
    }
    return f;
}
```