

Table of Contents

Part I Introduction	1
1 3. Longest Substring Without Repeating Characters	1
2 4. Median of Two Sorted Arrays	3
3 5. Longest Palindromic Substring	5
Index	0

I Introduction

Enter to iic text here.

1.1 3. Longest Substring Without Repeating Characters

Longest substring Without Repeating Chatacters

Brutal Force

最简单最暴力的解法就元用两重循环，外层循环逐个遍历字符串数组的每个元素，然后再用内的循环向后探索寻找无重复字符的子串 这种做法的时间复杂度为 $O(n^2)$ 此处使用容器 vector 来保存找到的不重复字符。

j 不重复的元素放入vector中，并使用c++的算法函数find查找

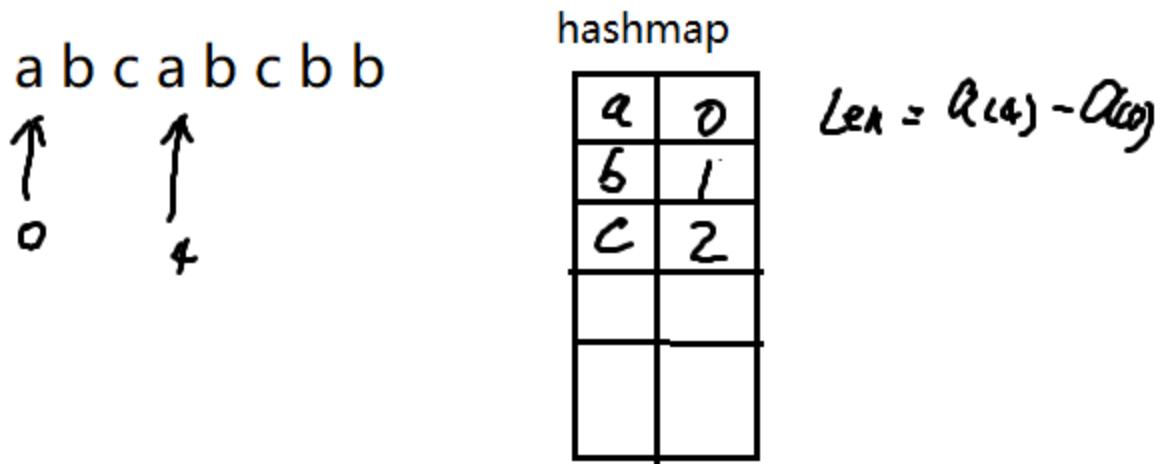
begin
all

C++ Code

```
class Solution {public:
    int lengthOfLongestSubstring(string s) {
        int max_len = 0;
        for (int i = 0; i < s.length(); i++) {
            vector<int> vec;
            int j = i;
            while (j < s.length() && find(vec.begin(), vec.end(), s[j]) == vec.end()) {
                vec.push_back(s[j]);
                j++;
            }
            max_len = max(max_len, (int)vec.size());
        }
        return max_len;
    }
};
```

优化化法

在求最长的无重复字母的子串过程中自然是希望在一次数组的遍历之后就可以得到答案也就是希望时间复杂度为 $O(n)$ 。这样的要求就需要子为路径记录的容器能够记录更多的信息在下图的示例中我们使用了哈希表存放的键值对是字符本身和字符在字符串中出现的位置那么当重复字符出现的时候就能明符的出重出字符第一次的出现位置 我们要求的不重复子串的长度就可以通过重字符当前下标减去重复字符前一次下标求得：



所以你肯定期待以下的代码

```
if (hashmap.find(s[i]) != hashmap.end()) {
    len = i - hashmap[s[i]];
    hashmap[s[i]] = i;
```

那这就是最后的解答吗

?No!

考虑这样一个问题 如果输入字符串本身就没有重复字符 那如何相减 如 $aabb$ 这样的字符串 你会发现根本没有机会执行长度的计算 那如果要将两个情况合并的话 只能采用以下的方式 这种方式的目的是为了适应没有重复元素的情况下的长度计算 所以在每一个点都要计算一次该点到某个起点的距离 没有重复的情况下该起点就是第一个字符的下标位置 如果重复的话就是重复字符第一次出现位置 为什么 是为了保证这个点是指向一个新的无重复字符串的起始位置。所以这个地方 $+1$ 的含义已经很明确了 就是一个无重复字符串子串的首字符下标。

```
int vval = 0;
if (hashmap.find(s[i]) != hashmap.end()) {
    val = hashmap[s[i]] + 1;
    hashmap[s[i]] = i ;
}
len = i - val + 1;
```

以上的解答还有一个方面没有考虑到 如下图所示当访问到第二个 b 的时候按照上面的算法 的值将被更新成 0 这直接导致计算出的长度值变为 2 仔细分析原因就会发现在更新v 值的时候发生了回退 也就是说新的v 的值比原来的值还小 这是导致错误的根源。

$$\begin{array}{r}
 a \quad b ; \quad b \quad a \\
 o \quad ; \quad : \quad - \quad 3
 \end{array}$$

a	0
b	1

要避免这样简问题其实很简单简需要在更改 val 的值的时候确认不会回退。

```
val = max(val, hashmp[s[]]);
```

C++ Code

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        map<char, int> hashmap;
        int max_len = 0;
        int start = 0;
        for (int i = 0; i < s.length(); i++) {
            if (hashmap.find(s[i]) != hashmap.end()) {
                start = max(start, hashmap[s[i]] + 1);
            }
            hashmap[s[i]] = i;
            int len = i - start + 1;
            max_len = max(max_len, len);
        }
        return max_len;
    }
};
```

1.2 4. Median of Two Sorted Arrays

Median of two sorted arrays

本题的难度在于需要在 $\log n$ 的时间复杂度内完成 而能在 $\log n$ 的时间复杂度内完成的解法一
般就是二分法。

可以使用主定理推导得出 由如下公式可以看出 在 $\log n$ 的时间内将问题的规模减少了一半
在这种情况下 时间复杂度就可以达到 $O(1)$

$$\underbrace{T(n)}_{\log n} = T(n/2) + O(1)$$

可以将问题改变为寻找第 k 大的数。根据题意采用递归的二分法，现在要求第 k 大的数 如果
每次能够排除 k 个元素那相当于在 $\log k$ 的时间内将问题的规模缩小了一半 那么这个思路
用到这道题上具体应该怎么做呢 有下图的例子现在求 A 和 B 两个数组所有元素的的中间数
也就是第 k 大的数 也就是 $A[k/2-1]$ 那么 在这个情况下比较 $A[k/2-1]$ 和 $B[k/2-1]$ 的大小也就
 5 , $k=5$, $k/2=2$, $A[2]$ 和 $B[2]$ 的大小也就

是 和 可知 所以可以确定第 大的数肯定不在 的 之间而应该在 A[1] 和 B[1] 中产生由此排除了 个数 接下来就可以用相同的方法求剩下的数中的第 大的数 由此我们将求第 个大的数的问题通过 的时间缩减为求第 个大的数的问题成功将问题规模缩减了差不多一半。 O(1) 3 ,

A: 1 10 17 18
B: 1 2 3 6 9

但我们还需要考虑一些特殊情况 以下例为说明

A: 5
B: 2 7 8 9 10 17 21 30

此时 是 是 要想在 中取 很不幸不存在 取不到这个数 但很明显 我们不能忽然将 A[0, k/2] 抛弃 而应该抛弃 A[0, 0] 然后在如下的数组中求第 大的数 A[0], B[0, 1], 3 ,

A: 5
B: 8 9 10 17 21 30

现在 而 此时显然应该抛弃 抛弃 之后 这个数组就 彻底消失了 所以剩下的第 大的数就只能在 中获取 那就只能是 A[0], A[0], A[1], B[1]=9 在上面的例子中有两个不存在需要仔细分析区别析这两种情况两代码中有所不同。

- 1) A 数组存在的 情况 取第 个元素 越界
- 2) A 数组所有元素均已抛弃的情况下 取第 个元素。

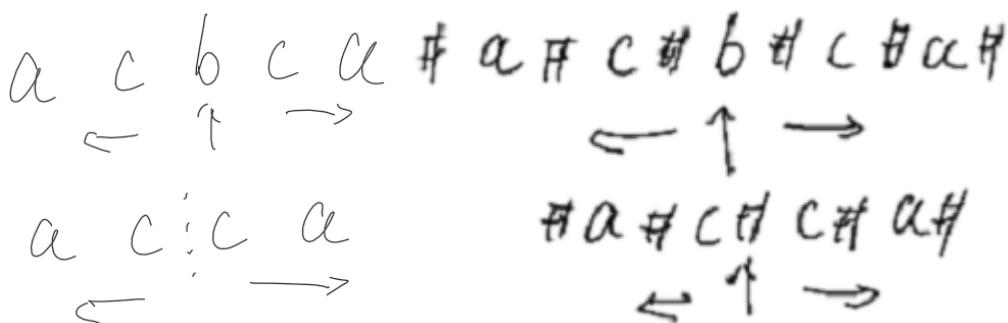
C++ Code

```
class Solution {
public:
    double findKth(vector<int> &nums1,
    int num1_pos, vector<int> &nums2, int num2_pos, int k) {
        if (num1_pos >= nums1.size()) {
            return nums2[num2_pos + k - 1];
        }
        if (num2_pos >= nums2.size()) {
            return nums1[num1_pos + k - 1];
        }
        if (k == 1) {
            return min(nums1[num1_pos], nums2[num2_pos]);
        }
        if (num1_pos + k / 2 - 1 >= nums1.size()) {
            return findKth(nums1, num1_pos, nums2,
            num2_pos + k / 2, k - k / 2);
        }
        if (num2_pos + k / 2 - 1 >= nums2.size()) {
            return findKth(nums1, num1_pos + k / 2, nums2,
            num2_pos, k - k / 2);
        }
        if (nums1[num1_pos + k / 2 - 1] >
        nums2[num2_pos + k / 2 - 1]) {
            return findKth(nums1, num1_pos + k / 2, nums2,
            num2_pos + k / 2, k - k / 2);
        } else {
            return findKth(nums1, num1_pos + k / 2, nums2, num2_pos, k - k / 2);
        }
    }
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int len = nums1.size() + nums2.size();
        if ((len & 1) == 0) {
            return (findKth(nums1, 0, nums2, 0, len / 2) +
            findKth(nums1, 0, nums2, 0, len / 2 + 1)) / 2.0;
        } else {
            return findKth(nums1, 0, nums2, 0, len / 2 + 1);
        }
    }
};
```

1.3 5. Longest Palindromic Substring**LongestPalindromic Substring****算法****Manacher**

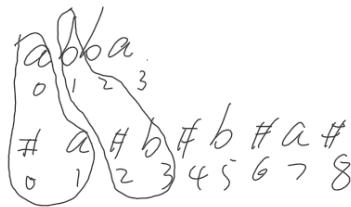
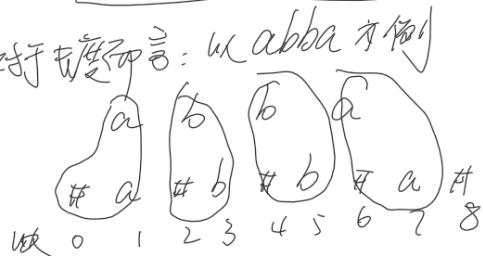
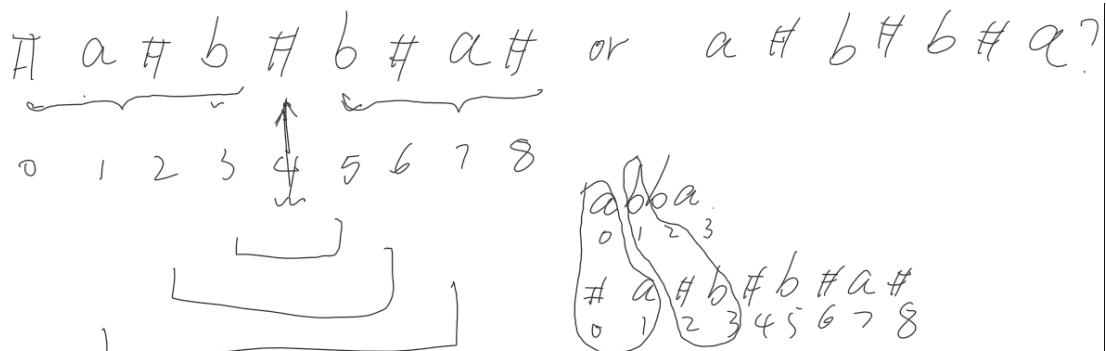
最自然的解法就是以一个字符为中点依次左右展开对比，但是这样的做法面临奇数偶数的问题。也就是回文串存在以下的两种形式 在这种情况下面临着使用两套算法来判断的问题极大的增加了解答问题的难度那么此处使用一种更灵巧的方法 在每个字符之间增加一个分割符

#



需首先将输入的字符串用特殊字符分割 但是分割的形式如何 出现在新字符串的首尾吗 还是说仅式起到字符串中间的分割功能 先讨论第一种情况 在这种情况下如何判断回文 从左向右逐个遍历在遍历过程中左右对比的方式寻找下长回文串。在这种处理方式中最重的就是将带#的字符串的射回以前的字符串。

#



- $1 \rightarrow 0$ 1/2 从以起始点.
- $3 \rightarrow 1$ 3/2 可以用起始点的
- $5 \rightarrow 2$ 5/2 位置旋转 /2 得到.
- $7 \rightarrow 3$ 7/2 那长度呢?

对于映射而言：
start: $\# - 4 = 4 - 4 = 0$ 距离等于
end: $\# + 4 = 8$ 4
所以长度为 $8 - 0 = 8$

以字符串 $head[0:i]$ 为例由图中的颜色区分可以得知每一个井的下标除以 2 就可以得到其后的字符在源串中的位置 所以可以得到公式

α | # b # c # a # d # a # c # d # e # g # h
 ↓ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
 β | b c a d a c d e g h
 ↓ 0 1 2 3 4 5 6 7 8 9
 α : 2 ~ 12 回文段 11
 β : 1 ~ 5 回文段 5

最后下标 / 2 - 起始下标 / 2 = 原字符串取的长度

有了以上的基础之后便有了 算法上一个的法最没有效率的地方就是没有选择的在每一个点进行了展开。如图所示的中对红色标注的点并没有太多展开的必要那么我们是否有办法可以跳过这些点的展开呢 是的 色算法通过 中的镜像特征可以可知中心点之 的点的 数组的值情况从而可以跳过一些不可能的中心点。

α : a c b c a b a c b c a b a c d
 β : 0 0 2 0 0 3 0 0 5 0 0 2 0 0 0

该算法的第一步自然还是用 来分割字符串以排除字符串的奇偶问题 这个算法的另外一个重要特性就是 $\#$ Mirrorin。

我们可以定义一个非常重要的整形数组 用来存放以每个字符为中心的回文串的长度 这个数组的重要性在于它可以帮助我们跳过一些不必要的点而直奔最有可能成为最长回文串的下一个中心点。

我们注意第一个 对应的值在 中是 如果从 这个成向左向右展开的话 所出的展开就是指以 为 中心出发向左向右去寻找回文串 可以得到回文串 如果仔细观察我们会发现 左边和右边的字符串相互之间互成镜像关系 这就给了我们一些灵感 这就是这个算法所利用的重要原理。

之的的展开算法使用穷举式的对比方法其时间复杂度很明。是 的而 算法可以将时间复杂度降低到 而它的核心思想就是要减少 点的展开 也就是想办法在遍历各点的过程中跳过一些点避免将其做为 这 的点展开那么如何选择这些 点呢？ 这些点必须是很有可能成为下一选最长回文串的点 如何判断有这样细潜力点呢？这个口诀便是 寻找这样的一个中心点 使以它为中心的回文串扩充到刚刚访问过的中心点的右边界。“举例如下”

从中查找最长的回文子串 这里面的回文子串包含中心点 中心点 中心点 那么现在按照新的算法进行推导

5:1~8 baxabaxab, 7:5~9 baxab 前四个点同样的是按照前面的穷举算法进穷推导到中心点 也就法 的时候 我们已经访问了 的字符区间。

0~6 个点成做下一个中心点呢 是 吗 如果这样做那么就失去了以 为 中心的回文串 那我 们是否又应到直接查找 呢 当然我们可以这样做 但这样就回到了穷举算法 说好的不一样呢 那到底谁是 下一个中心点呢 ？

按照前面的口诀观察 发现它并没有扩展到以 为 中心的回文串的右边界 所以 不可能成为下一个中心 原因是以 为 中心的回文串必然包含在以 为 中心的回文串之中。所以 必然得不到比 更长的回文串

那么又如何呢以为中心的回文串的确可以扩展到以为中心的回文串的右边界所以很有潜力可以做为下一个中心点的候选 $x(3)$

那本身呢同理的也是的右边界所以本身也是一个中心点的候选。
 $a(6), b(5)$
 $a(6) ? a(6) x(3) a(6)$

但是比更有可能成为下一个候选因为子回文串扩充到至少是的右边界那么就有很大可能继续扩充突破右边界方面的字符串内容还没有访问所以这句话很容易理解所以就自然是下一个选择。 $a(6)$
 $a(6), b(5)$

解释到这理就自然产生一个疑问你是如何知道到底能不能扩展到为中心的回文串的右边界我们刚刚还说过不能分别去扩展的呀
 $a(4), a(5), a(6) x(3)$
 $a(4), b(5), a(6) ?$

到这理就该讲解 算法的核心理念因为在已知是回文串的时候中心点两边的项是相互对应的也就是说其数组的值是相对应的。有了这个特性我们就可以预先得知右边项的数组值的一些特征而这些特征可以很好的帮助我们为选取下一个中心点做出决策。为了说明这个问题我们重新看一个例子。

有以下的字符串以及其对应的数组。

p : $0 \mid 2 \quad 3 \quad ? \quad ? \quad ?$

$A_{c1} \quad B_{c2} \quad A_{c3} \quad B_{c4} \quad A_{c5} \quad B_{c6} \quad A_{c7}$

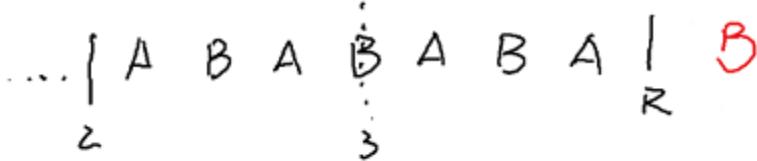
如果们以实为中心折叠的是我们可以发现其实的数组的值其实和左边的值是一样的也就是说以中心点为轴左右两部分的数组值也是对称的这也是的一个理论基础但这不是全部而折是一个开始。 p Mirroring

$B_{c4})$

2	$A_{c3})$	$A_{c5})$	$?$	$\rightarrow 2$
1	$B_{c2})$	$B_{c6})$	$?$	$\rightarrow 1$
0	$A_{c1})$	$A_{c7})$	$?$	$\rightarrow 0$

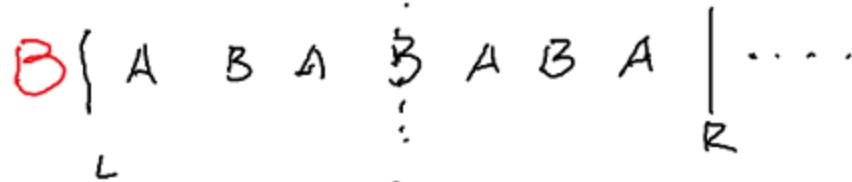
上面的例子没有考虑左右情况的还可虑有其它字符的情况那现在我们来假设左右都可以扩展的情况。

若右边是以括展的所以在扩展计算以后可以知道右边数组的值可能比其左边的相对项的值大。但其值可以确认至少与左边的相对项的值相等。实际是多少还需要展开后才能知道。



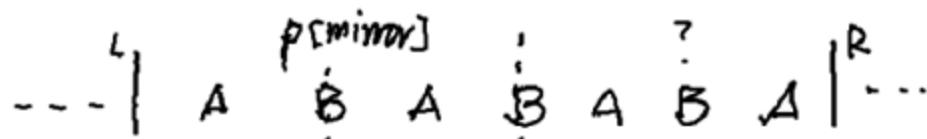
$$\begin{array}{r}
 & B \\
 \begin{array}{r} 2 \\ 1 \\ 0 \end{array} & \begin{array}{r} A \\ B \\ A \end{array} & \begin{array}{r} A \\ B \\ A \end{array} & \begin{array}{r} 2?x \rightarrow 3 \\ 1?x \rightarrow 2 \\ 0?x \rightarrow 1 \end{array} \\
 \hline & B & A & B
 \end{array}$$

若左边是可以扩展的 扩展左边可以得到 数组的值 但对于右边的相对项而言 这个值是不真实的 右边项的值其实还是扩展到右边界值。



$$\begin{array}{r}
 & B \\
 \begin{array}{r} 3 \\ 2 \\ 1 \end{array} & \begin{array}{r} A \\ B \\ A \end{array} & \begin{array}{r} A \\ B \\ A \end{array} & \begin{array}{r} 3?x \rightarrow 2 \\ 2?x \rightarrow 1 \\ 1?x \rightarrow 0 \end{array} \\
 \hline & B & A & B
 \end{array}$$

由上面的左右扩展情况其实我们可以得到一个估算 数组值的规律



如果 跨越了 边界那么 的值就不能由 能的值来决定此时的公式应
该 $p[B-mirror]$ L $p[B]$ $p[B-mirror]$

$$np[B] = R - B's\ ind]x$$

如果 在 边界之内 那么

$$2) p[B-mirror] L p[B] = p[B-mirror]$$

由此我们得到选取下一个中心点的 一个原则 一选择三拒绝
 4 ()
 一个选择

1) palindrome expands till right edge & its mirror palindrome is prefix (prefix
 p[mirror]的值正好到L边界,没有超出L边界。
 三放弃

1) Totally contained under current palindrome

2) Current palindrome expands till end of input

特别的重要仔细思想可以明白 不选择这点的原因就是 边界后面的点必然不能与前面的点形成回文 如果形成了那意味着其实 边界应该继续向右扩充 这与已知的事实冲突 典型的反证思路。 这一点

那么最后我们从头到尾用一个例子来说明该算法。前逻辑项我们都需要进行无选择的展开之后辑逻辑按照下面推进。

a b a x a b a x a b b
 0 1 2 3 4 5 6 7 8 9 10
 p: 1 0 1 0 3 = 0 3 1 = 0 镜像原则,所以b(5)肯定是下一个中心点

2) 0 1 0 3 0 4 = 0 3 3 = 0 3 | x 镜像的p数组值其实已经越过了L左
 由上面的一选择三放弃原则可以知道
 左边界而需要放弃,而a(8)则是其p数

3)

✓ 最后只能选择b(10)做为下一个中

C++ Code

```

class Solution {
public:
    string prepareString(string s) {
        string res = "#";
        for (int i = 0; i < s.length(); i++) {
            res += s[i];
            res += "#";
        }
        return res;
    }

    string longestPalindrome(string s) {
        string res;
        if (s.length() == 0) {
            return res;
        }
        string preStr = prepareString(s);
        vector<int> p(preStr.length(), 0);
        int i = 1;
        int max_pval = 0;
        int cnt = 0;
        while (i < preStr.length() - 1) {
            while (i - cnt >= 0 && i + cnt < preStr.length() && preStr[i + cnt] == preStr[i - cnt]) {
                cnt++;
            }
            p[i] = cnt - 1;
            if (p[i] > max_pval) {
                res = s.substr((i - p[i]) / 2, (i + p[i]) / 2 - (i - p[i]) / 2);
                max_pval = p[i];
            }
            if (i + cnt == preStr.length()) {
                break;
            }
            bool clear = true;
            int center = i + cnt;
            for (int j = i + 1; j < i + cnt; j++) {
                int j_mirror = 2 * i - j;
                p[j] = min(p[j_mirror], i + cnt - j - 1);
                if (p[j_mirror] == i + cnt - j - 1) {
                    center = j;
                    clear = false;
                }
            }
            if (clear) {
                cnt = 0;
            }
            i = center;
        }
        return res;
    };
}

```

动态规划

动这规划就是将复杂的问题分解为更为简单的子问题的集合。将这些子问题的解存放在一个数据结构采题个子问题只解决一次 不重复计算 存储它们的解多是采用数组 并没有使用太复杂的题据结构。

在本题中 我们采用一个二维数组 这个二维数组用行来代表子串的起始位置 用列来代表子串的结束位置而用具体的一个 来代表这个行起列止的子串是否是回文串。行和列的大小都是整个输入字符串的长度。所以如果字符串 是回文串的话 否则

$$[i,j] \quad \text{cell}(i,j) == \text{true}, \quad \text{cell}(i,j)$$

$= \text{false}$

有了这个数据结构的帮助之后 就可以讨论具体的解答步骤了。可以看出无论奇偶回文串都可以按照一定的方式递归的进行判断。字符串首字母和字符串最后一个字母是否相等 如果相等 那么去掉字符串的首尾字符 取其中的子串按照同样的方法递归的进行判断。这是一种递归的方式是由大到小的一种处理方式 而动态规划则是逆向行驶 由一个一个字符开始 到两个字符 到三个字符 再到整个字符串 一个字符的判断结果为两个字符的判断提供信息 而两个字符的判断则为三个字符的判断提供信息 以此类推。中间所有结果或者信息都由二维数组保存。

