Manacher算法详解

Manacher

Manacher算法是一个用来查找一个字符串中的最长回文子串(不是最长回文序列)的线性算法。它的优点就是把时间复杂度为$O(n^2)$的暴力算法优化到了O(n)。首先先让我们来看看最原始的暴力扩展，分析其存在的弊端，以此来更好的理解Manacher算法。

暴力匹配

暴力匹配算法的原理很简单，就是从原字符串的首部开始，依次向尾部进行遍历，每访问一个字符，就以此字符为中心向两边扩展，记录该点的最长回文长度。那么我们可以想想，这样做存在什么弊端，是不是可以求出真正的最长回文子串?

答案是显然不行的，我们从两个角度来分析这个算法

1.不适用于偶数回文串

我们举两个字符串做例子，它们分别是 "aba"，"abba"，我们通过肉眼可以观察出，它们对应的最长回文子串长度分别是3和4，然而我们要是用暴力匹配的方法去对这两个字符串进行操作就会发现，"aba" 对应的最长回文长是 "131"，"abba" 对应的最长回文长度是 "1111"，我们对奇数回文串求出了正确答案，但是在偶数回文串上并没有得到我们想要的结果，通过多次测试我们发现，这种暴力匹配的方法不适用于偶数回文串

2.时间复杂度$O(n^2)$

这里的时间复杂度是一个平均时间复杂度，并不代表每一个字符串都是这个复杂度，但因为每到一个新位置就需要向两边扩展比对，所以平均下来时间复杂度达到了$O(n^2)$。

Manacher算法本质上也是基于暴力匹配的方法，只不过做了一点简单的预处理，且在扩展时提供了加速

Manacher对字符串的预处理

我们知道暴力匹配是无法解决偶数回文串的，可Manacher算法也是一种基于暴力匹配的算法，那它是怎么来实现暴力匹配且又不出错的呢？它用来应对偶数字符串的方法就是——做出预处理，这个预处理可以巧妙的让所有字符串都变为奇数回文串，不论它原本是什么。操作实现也很简单，就是将原字符串的首部和尾部以及每两个字符之间插入一个特殊字符，这个字符是什么不重要，不会影响最终的结果(具体原因会在后面说)，这一步预处理操作后的效果就是原字符串的长度从n改变成了2*n+1，也就得到了我们需要的可以去做暴力扩展的字符串，并且从预处理后的字符串得到的最长回文字符串的长度除以2就是原字符串的最长回文子串长度，也就是我们想要得到的结果。

这里解释一下为什么预处理后不会影响对字符串的扩展匹配

比如我们的原字符串是 "aa"，假设预处理后的字符串是 "#a#a#"，我们在任意一个点，比如字符 '#'，向两端匹配只会出现 'a' 匹配 'a'，'#' 匹配 '#' 的情况，不会出现原字符串字符与特殊字符匹配的情况，这样就能保证我们不会改变原字符串的匹配规则。通过这个例子，你也可以发现实际得到的结果与上述符合。

Manacher算法核心

Manacher算法的核心部分在于它巧妙的令人惊叹的加速，这个加速一下把时间复杂度提升到了线性，让我们从暴力的算法中解脱出来，我们先引入概念，再说流程，最后提供实现代码。

概念：

ManacherString：经过Manacher预处理的字符串，以下的概念都是基于ManasherString产生的。

回文半径和回文直径：因为处理后回文字符串的长度一定是奇数，所以回文半径是包括回文中心在内的回文子串的一半的长度，回文直径则是回文半径的2倍减1。比如对于字符串 "aba"，在字符 'b' 处的回文半径就是2，回文直径就是3。

最右回文边界R：在遍历字符串时，每个字符遍历出的最长回文子串都会有个右边界，而R则是所有已知右边界中最靠右的位置，也就是说R的值是只增不减的。

回文中心C：取得当前R的第一次更新时的回文中心。由此可见R和C时伴生的。

半径数组：这个数组记录了原字符串中每一个字符对应的最长回文半径。

流程：

步骤1：预处理原字符串

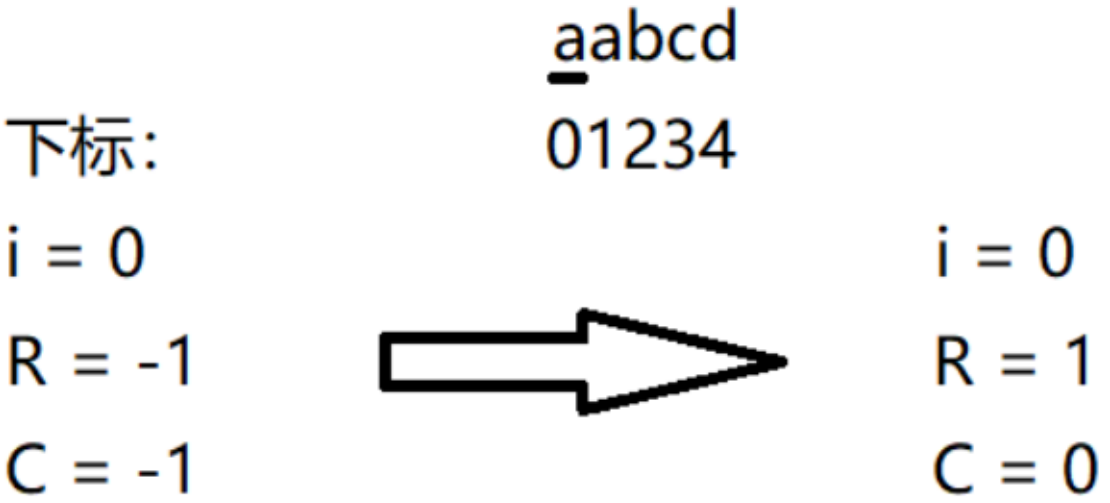
先对原字符串进行预处理，预处理后得到一个新的字符串，这里我们称为S，为了更直观明了的让大家理解Manacher的流程操作，我们在下文的S中不显示特殊字符(这样并不影响结果)。

步骤2：R和C的初始值为-1，创建半径数组pArr

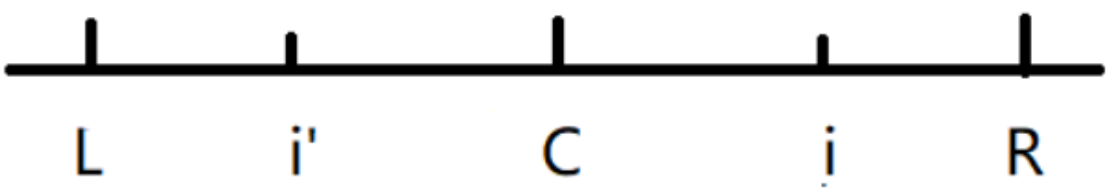
这里有点与概念相差的小偏差，就是R实际是最右边界位置的右一位。

步骤3：开始从下标 i = 0去遍历字符串S

分支1：i > R ，也就是i在R外，此时没有什么花里胡哨的方法，直接暴力匹配，此时记得看看C和R要不要更新。

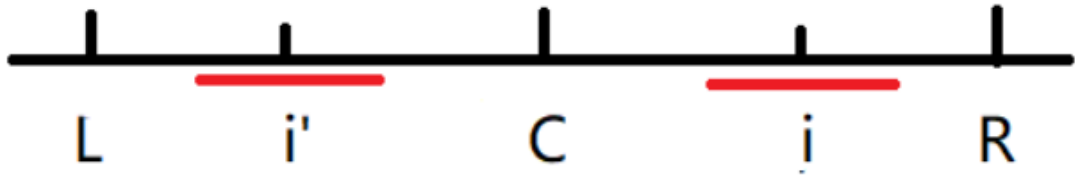


分支2：i <= R，也就是i在R内，此时分三种情况，在讨论这三个情况前，我们先构建一个模型



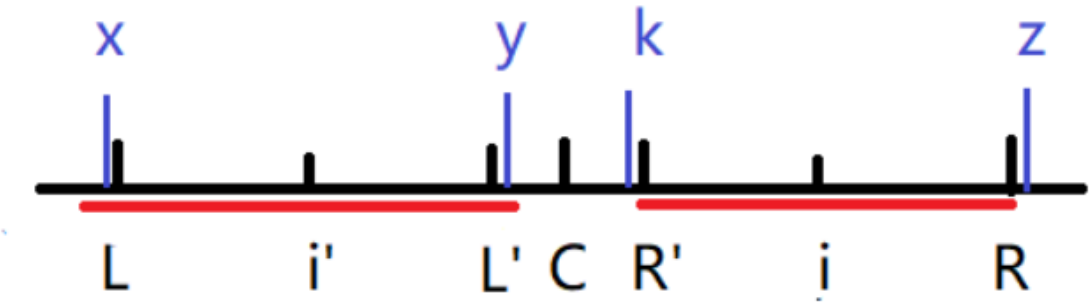
L是当前R关于C的对称点，i'是i关于C的对称点，可知 i' = 2*C - i，并且我们会发现，i'的回文区域是我们已经求过的，从这里我们就可以开始判断是不是可以进行加速处理了

情况1：i'的回文区域在L-R的内部，此时i的回文直径与 i' 相同，我们可以直接得到i的回文半径，下面给出证明



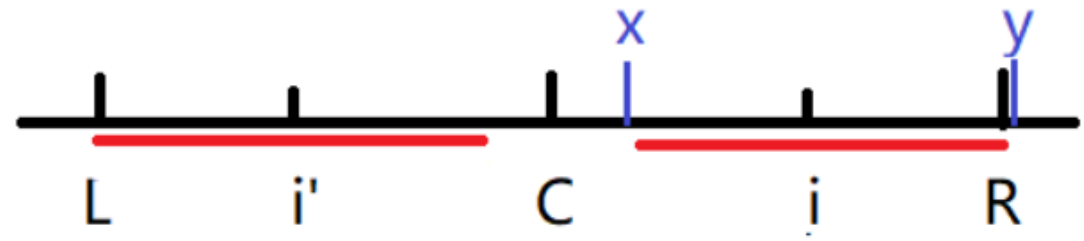
红线部分是 i' 的回文区域，因为整个L-R就是一个回文串，回文中心是C，所以i形成的回文区域和i'形成的回文区域是关于C对称的。

情况2：i'的回文区域左边界超过了L，此时i的回文半径则是i到R,下面给出证明



首先我们设L点关于i'对称的点为L'，R点关于i点对称的点为R'，L的前一个字符为x，L'的后一个字符为y，k和z同理，此时我们知道L - L'是i'回文区域内的一段回文串，故可知R' - R也是回文串，因为L - R是一个大回文串。所以我们得到了一系列关系，x = y，y = k，x != z，所以 k != z。这样就可以验证出i点的回文半径是i - R。

情况3：i' 的回文区域左边界恰好和L重合，此时i的回文半径最少是i到R，回文区域从R继续向外部匹配，下面给出证明



因为 i' 的回文左边界和L重合，所以已知的i的回文半径就和i'的一样了，我们设i的回文区域右边界的下一个字符是y，i的回文区域左边界的上一个字符是x，现在我们只需要从x和y的位置开始暴力匹配，看是否能把i的回文区域扩大即可。

总结一下，Manacher算法的具体流程就是先匹配 -> 通过判断i与R的关系进行不同的分支操作 -> 继续遍历直到遍历完整个字符串

时间复杂度：

我们可以计算出时间复杂度为何是线性的，分支一的情况下时间时间复杂度是O(n)，分支二的前两种情况都是O(1)，分支二的第三种情况，我们可能会出现O(1)——无法从R继续向后匹配，也可能出现O(n)——可以从R继续匹配，即使可以继续匹配，R的值也会增大，这样会影响到后续的遍历匹配复杂度，所以综合起来整个算法的时间复杂度就是线性的，也就是O(n)。

实现代码：

整个代码并不是对上述流程的生搬硬套(那样会显得代码冗长)，代码进行了精简优化，具体如何我会在代码中进行注释

```
#include<iostream>
#include<string>
#include<cstring>
#include<algorithm>
#include<vector>
#include<cmath>
using namespace std;
//算法主体
int maxLcsplength(string str) {
    //空字符串直接返回0
    if (str.length() == 0) {
        return 0;
    }
    //记录下manacher字符串的长度，方便后面使用
    int len = (int)(str.length() * 2 + 1);
    //开辟动态数组chaArr记录manacher化的字符串
    //开辟动态数组pArr记录每个位置的回文半径
    char *chaArr = new char[len];
    int* pArr = new int[len];
    int index = 0;
    for (int i = 0; i < len;i++) {
        chaArr[i] = (i & 1) == 0 ? '#' : str[index++];
    }
    //到此完成对原字符串的manacher化
    //R是最右回文边界，C是R对应的最左回文中心，maxn是记录的最大回文半径
    int R = -1;
    int C = -1;
    int maxn = 0;
    //开始从左到右遍历
    for (int i = 0; i < len; i++) {
        //第一步直接取得可能的最短的回文半径，当i>R时，最短的回文半径是1，反之，最短的回文半径可能是i对应的i'的回文半径或者i到R的距离
        pArr[i] = R > i ? min(R - i, pArr[2 * C - i]) : 1;
        //取最小值后开始从边界暴力匹配，匹配失败就直接退出
        while (i + pArr[i]<len && i - pArr[i]>-1) {
            if (chaArr[i + pArr[i]] == chaArr[i - pArr[i]]) {
                pArr[i]++;
            }
            else {
                break;
            }
        }
        //观察此时R和C是否能够更新
        if (i + pArr[i] > R) {
            R = i + pArr[i];
            C = i;
        }
```

```
        //更新最大回文半径的值
        maxn = max(maxn, pArr[i]);
    }
    //记得清空动态数组哦
    delete[] chaArr;
    delete[] pArr;
    //这里解释一下为什么返回值是maxn-1，因为manacherstring的长度和原字符串不同，所以这里得到的最大回文半径其实是原字符串的最大回文子串长度加1，有兴趣的可以自己验证试试
    return maxn - 1;
}
int main()
{
    string s1 = "";
    cout << maxLcsplength(s1) << endl;
    string s2 = "abbbca";
    cout << maxLcsplength(s2) << endl;
    return 0;
}
```

下面附上java代码

```
public class Manacher {

    public static char[] manacherString(String str) {
        char[] charArr = str.toCharArray();
        char[] res = new char[str.length() * 2 + 1];
        int index = 0;
        for (int i = 0; i != res.length; i++) {
            res[i] = (i & 1) == 0 ? '#' : charArr[index++];
        }
        return res;
     }

    public static int maxLcpsLength(String str) {
        if (str == null || str.length() == 0) {
            return 0;
        }
        char[] charArr = manacherString(str);
        int[] pArr = new int[charArr.length];
        int C = -1;
        int R = -1;
        int max = Integer.MIN_VALUE;
        for (int i = 0; i != charArr.length; i++) {
            pArr[i] = R > i ? Math.min(pArr[2 * C - i], R - i) : 1;
            while (i + pArr[i] < charArr.length && i - pArr[i] > -1) {
                if (charArr[i + pArr[i]] == charArr[i - pArr[i]])
                    pArr[i]++;
                else {
                    break;
                }
            }
            if (i + pArr[i] > R) {
                R = i + pArr[i];
                C = i;
            }
            max = Math.max(max, pArr[i]);
        }
```

```
        return max - 1;
    }

    public static void main(String[] args) {
        String str1 = "abc123321cba";
        System.out.println(maxLcpsLength(str1));
    }

}
```

2020/11/13 wz摘自 https://www.cnblogs.com/cloudplankroader/p/10988844.html