找回文字符串——Manacher's Algorithm

Author: **Brandon Lin**, *October 16, 2023*, Shanghai

定义

- 待找字符串：s_0，例子：`'ababaabc'`
- 填充后的字符串：s，例子：`'^#a#b#a#b#a#a#b#c#$'`
- 每个字符的位置：i，上例中第一个 `a` 的 $i = 2$.
- 【函数】以位置为 i 字符为中心，所能找到的最长回文字符串的向单向展开的长度 $l(c)$：

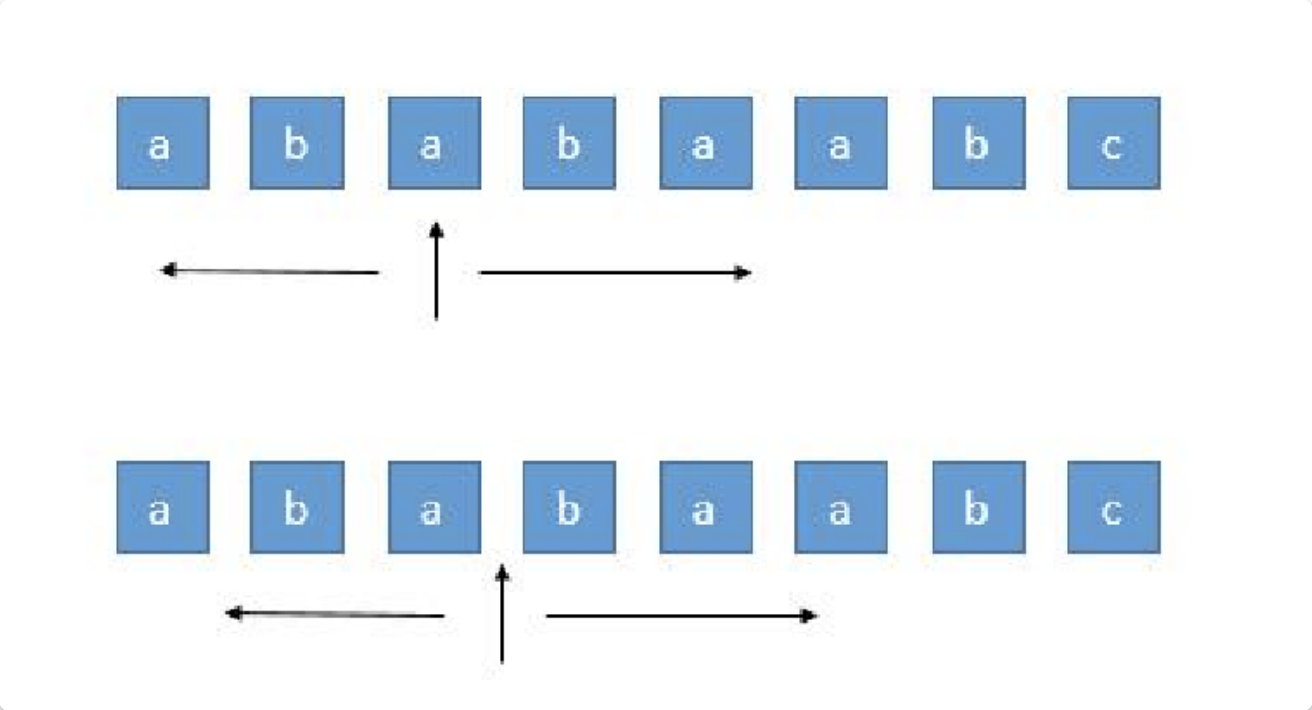
$$l : i \mapsto l(i)$$

- 以 $i = c$ 中心的字符位置，其能找到的最大回文串的左右位置分别定义为 $L = c - l(c)$ 和 $R = c + l(c)$.
- 在循环中，定义 r 为从 $i = c$ 位置向右走的步数

算法建立与介绍

算法基础：扩展中心法

每次循环寻找一个中心，进行扩展。



算法问题：

1. 奇偶性问题，需要分别以字符串上的字符、以字符串字符间的空格作为中心进行扩展
2. 重复判断，浪费效率

解决奇偶性问题

我们以以下逻辑进行填充：

1. 在每个字符串中间插入 `#`，对首尾字符同样进行操作：`'ababaabc'` $\Longrightarrow$ `'#a#b#a#b#a#a#b#c#'`
2. 在数列开始的地方插入：`^`，在数列结束的地方插入：`$`，命名为 s：`^#a#b#a#b#a#a#b#c#$`

证明 $\mathrm{card}(s)$ 是奇数：对于 n 个字符，一定有 $n+1$ 个空格（算上首尾位置），填充后的字符一共有 $n+n+1+2=2n+3$ 位，一定是奇数。

每一个字符为中心，所能找到的最长回文串的长度——定义与性质

我们定义函数 l：以位置为 i 字符为中心，所能找到的向单向展开的最长回文字符串的长度 $l(c)$：

$$l:i\mapsto l(i)$$

在 `^#a#b#a#b#a#a#b#c#$` 例子中：

	^	#	a	#	b	#	a	#	b	#	a	#	a	#	n	#	c	#
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$l(i)$	0	0	1	0	3	0	5	0	3	0	1	2	1	0	1	0	1	0

$l(i)$ 的“部分”对称性

我们发现以 $i=6$ 为中心，$l(i)$ 呈现了一定的对称性。这启发我们通过知道了 $i=[\![1,6]\!]$ 的 $l(i)$，即可以自动填充 $i=[\![7,12]\!]$ 位置的 $l(i)$，其中有些特例我们会在下个小节讨论。

【条件1】我们定义中心的字符位置 $i=c$，其能找到的最大回文串的左右位置分别定义为 $L=c-l(c)$ 和 $R=c+l(c)$. 在只考虑由 $l=c$ 位置得到的最大回文串中（这个前提非常非常重要！后面我们会改进），由 $l(c)$ 的定义我们可以得到：以 $l=c$ 位置向右走 r 步，

$$\forall r\in[\![0,l(c)]\!],\ l(c+r)=l(c-r)$$

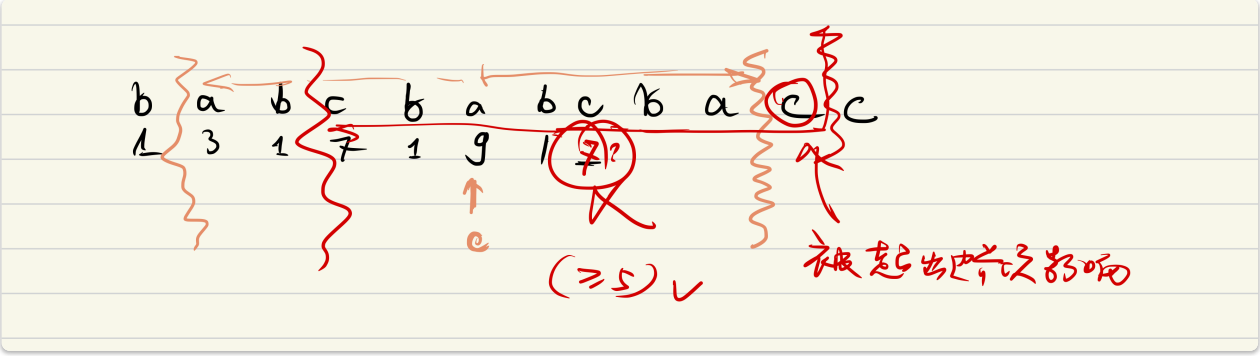
这使得我们通过上面的表达式自动得到了 $\{l(c+r),\ r\in[\![0,l(c)]\!]\}$ 的值。

由此我们设计算法【最初版，有问题】：

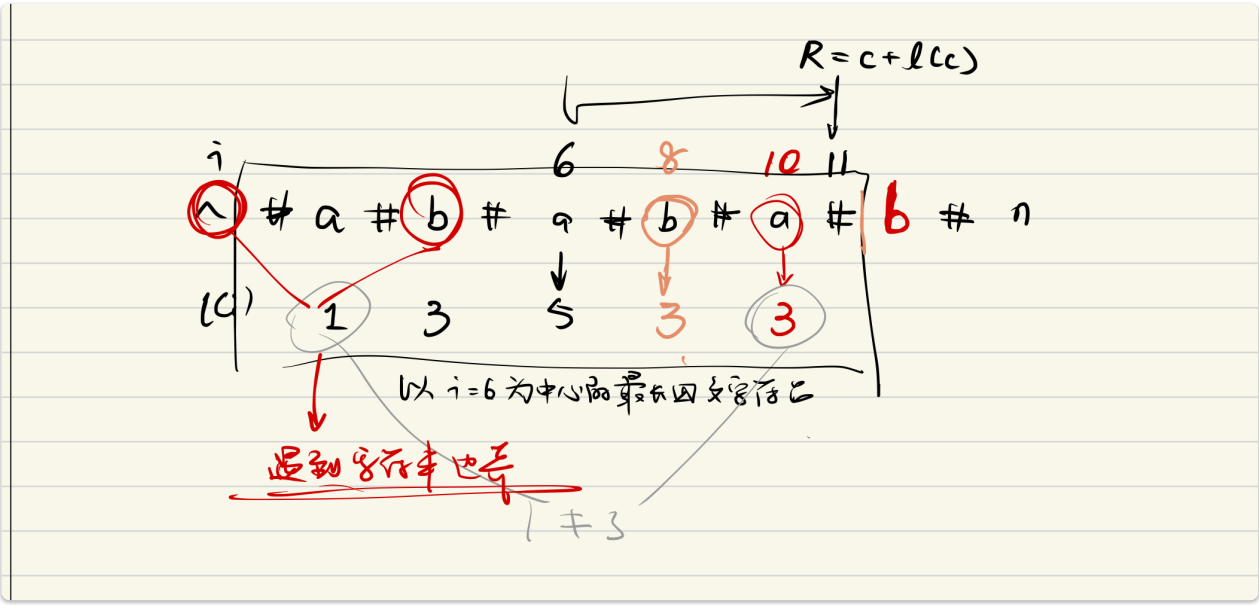
- 初始化：$c=0$（即以 $i=0$ 作为中心开始）
- 对于每一个 c：
  1. 用扩展中心法求得 $l(c)$，从而得到 $L=c-l(c),\ R=c+l(c)$
  2. 从位置 $i=c$ 向右走 r 步，其中 $r\in[\![0,l(c)]\!]$：得到
  $$\forall r\in[\![0,l(c)]\!],\ l(c+r)=l(c-r)$$
  3. 跳过 $[\![c+1,c+l(c)]\!]$ 部分（当 $l(c)=0$ 时直接就走到下一位 $i=c+1$），计算位置为 $i=c+l(c)+1$ 所能得到的最大回文串的长度，重复循环。

一些不那么对称的例子与原因

1. 【问题1】我们在上面的论述中提出：只考虑由 $l=c$ 位置得到的回文串中是这么得到 $l(c+r)$ 的。什么时候 $l(c+r)$ 会被由【以其他位置为中心的最大回文字符串】影响呢？根据 l 的定义我们得到，当
   $$c+r+l(c+r)>R=c+l(c)$$
   的时候，【$i=r$ 位置的字符所包含的最大回文字符串】已经到达了【以 $i=c$ 位置的字符所包含的最大回文字符串】的右端，这个时候需要重新计算：比较是否 $s[r+l(c+r)+1]=s[r-l(c+r)-1]$ 用扩展中心法继续。



2. 【问题2】 观察【以 $i=c$ 位置的字符所包含的最大回文字符串】左端值 $L=c-l(c)$. 很显然这个值大于 0. 但是，左侧的值有可能是因为【遇到字符起止符号】而得到，因此得从 $i=c$ 位置重新开始用扩展中心法：



换一个视角——从向前跑动的指针的眼中看问题

我们之前的介绍都是以位置为 c 向两边展开的视角遍历完整个字符串。在每次求 c 对应的 $l(c)$ 的过程中，我们都有一个向前跑动的“指针”，在最广泛的情况下达到 $c+l(c)$ 为止；接着，这个“指针”又接着从这一位的下一位向前跑动：这启发我们，这是一个从 0 位置直到 $\text{len}(s)$ 位置跑动的变量，我们可以基于这个变量写循环。在代码中，我们将其命名为 `ptr_str`，下面的数学推导中写为 p.

与之对应的，以 c 为对称的“指针” $p_{inv}=2c-p$

最终算法

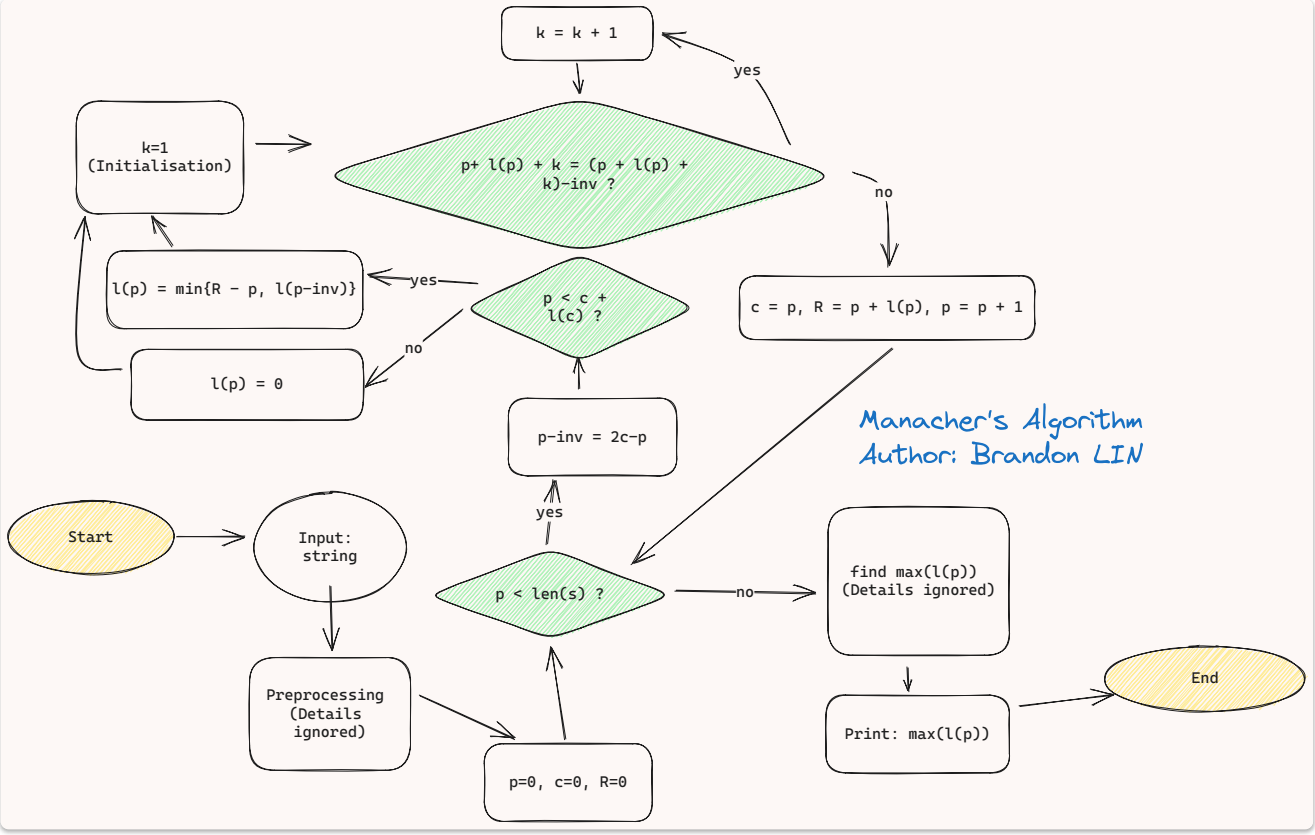
算法

输入：字符串 s_0

- 填充字符串：$s_0 \to s$，求得 $n=\text{len}(s)$
- 初始化：$p=0$，$c=0$（即以 $i=0$ 作为中心开始）
- 初始化：$R=0$
- 对于 $p=[\![0,n-1]\!]$：
  - $p_{inv}=2c-p$
  - 【条件1】如果 p 保持在 $R=c+l(c)$ 的左端：（$p<c+l(c)$）
    - 【问题2】 如果 $l(p_{inv})$ 受到字符串起始位置影响：这一块可以简化成取最小值

        - $l(p) = R - p$
        - 否则 $l(p) = l(p_{inv})$
      - 否则 $l(p) = 0$
    - 如果 $\widetilde{p} = p + l(p)$ 的下一位（即 $\widetilde{p} + 1$与 $(\widetilde{p} + 1)_{inv}$ ）相同，则需要在已经得到的 $l(p)$ 上继续加 1，直到达到不再相同的位数为止（扩展中心法）
    - 【更新 R，c，问题1】：如果 $p + l(p) > R$，我们就以这一个点为中心
      - $c = p$
      - $R = p + l(p)$
- 找到 $\max l(p)$.

流程图



复杂度

- 最外层的 p 遍历从 0 至 $n - 1$. 观察到循环内： $\widetilde{p}$ 有一定程度扩展：从 $p + l(p)$ 的地方直到 R，随后更新 R，新的 R 值在 $\widetilde{p}$ 所在的位置的右侧。
- 需要提醒的是，当 $p + l(p) > R$ 的时候，两边字符才有相等的可能，因此整个步骤为：从 $R + 1$ 的地方开始比较直到某一个地点，并将其设为新的 R 值。
- 因此可以简单证明每一个字符最多被遍历两次、不会再被循环内的 $\widetilde{p}$ 遍历到第三次
- 剩余都是简单操作，因此复杂度为 $O(\text{len}(s)) = O(\text{len}(s_0))$.

代码的一些备注

程序接受长度最高为100的字符串，且！按 Enter 即终止输入！
输出：如果找到的最长回文字符串为 1，那就输出没找到（很直观的解释，长度为 1 就是不存在）；若大于 1，则输出字符串长度和回文字符串内容。

附录

```c
#include <stdio.h>
#include <string.h>

#define MAX_STR_LENGTH 100

// 函数声明
void readString(char *string, int string_size);
void printString(char *string);
int preprocessString(char *string, char *string_preprocessed);
int manacher(char *string, char *result, int str_len);

int main() {
    char string_input[MAX_STR_LENGTH];              // 存储输入的原始字符串
    char string_process[2 * MAX_STR_LENGTH + 3];    // 存储预处理后的字符串
    char result[MAX_STR_LENGTH];                    // 存储最长回文子串
    int str_len = 0;                                // 输入字符串的长度
    int max_len = 0;                                // 最长回文子串的长度

    printf("Type in the string. \nPress return to end the input: ");
    readString(string_input, sizeof(string_input));  // 从标准输入读取字符串
    printString(string_input);                        // 打印原始输入字符串

    str_len = preprocessString(string_input, string_process);  // 预处理输入字符串
    // printString(string_process);  //  (可选) 打印预处理后的字符串

    max_len = manacher(string_process, result, str_len);  // 使用Manacher算法找到最长回
文子串
    if (max_len != 1) {
        printf("The longest palindrome we found has length %d. ", max_len);
        printString(result);  // 打印最长回文子串
    } else {
        printf("No palindrome has been found.\n");
    }

    printf("End of program.\n");

    return 0;
}

// 从标准输入读取字符串，直到回车键为止
void readString(char *string, int string_size) {
    int ch;
    int index = 0;

    while ((ch = getchar()) != '\n' && ch != EOF) {
        if (index < string_size - 1) {
            string[index] = (char)ch;  // 确保读入的是char类型
            index++;
        }
    }
    if (ch == '\n') {
        string[index] = '\0';  // 字符串以'\0'结尾
    }
}

// 打印字符串
```

```c
void printString(char *string) {
    char *pointer_to_string = string;

    printf("The string is: ");
    while (*pointer_to_string != '\0') {
        printf("%c", *pointer_to_string++);
    }
    printf("\n");
}

// 预处理输入字符串，添加'$'和'#'字符
int preprocessString(char *string, char *string_preprocessed) {
    int str_pre_id = 0;
    string_preprocessed[str_pre_id++] = '$';
    string_preprocessed[str_pre_id++] = '#';

    for (int str_id = 0; string[str_id] != '\0'; str_id++) {
        string_preprocessed[str_pre_id++] = string[str_id];
        string_preprocessed[str_pre_id++] = '#';
    }

    string_preprocessed[str_pre_id] = '\0';

    return str_pre_id - 1; // 返回预处理后的字符串长度，最后一位省略
}

// Manacher算法计算最长回文子串
int manacher(char *string, char *result, int str_len) {
    int max_len = 0;

    int longest_pa[MAX_STR_LENGTH];
    int ptr_str = 0, center = 0, Right_pos = 0, ptr_str_inv = 0;

    for (ptr_str = 0; ptr_str <= str_len; ptr_str++) {
        ptr_str_inv = 2 * center - ptr_str;

        if (ptr_str < Right_pos) {
            // 计算当前字符位置的最长回文半径
            longest_pa[ptr_str] = (Right_pos - ptr_str < longest_pa[ptr_str_inv] ?
                                   Right_pos - ptr_str : longest_pa[ptr_str_inv]);
        } else {
            longest_pa[ptr_str] = 0;
        }

        while (string[ptr_str + longest_pa[ptr_str] + 1] ==
               string[ptr_str - longest_pa[ptr_str] - 1]) {
            longest_pa[ptr_str] += 1;
        }

        if (ptr_str + longest_pa[ptr_str] > Right_pos) {
            center = ptr_str;
            Right_pos = ptr_str + longest_pa[ptr_str];
        }
    }

    int max_len_center_pos = 0;
    for (int id = 0; id <= str_len; id++) {
```

```
        if (max_len < longest_pa[id]) {
            max_len = longest_pa[id];
            max_len_center_pos = id;
        }
    }

    if (max_len) {
        int max_len_start_pos = max_len_center_pos - max_len + 1;
        int result_id = 0;
        int string_id = max_len_start_pos;

        for (; result_id < max_len; result_id++) {
            if (string[string_id] != '#' ||
                string[string_id] != '^' ||
                string[string_id] != '$') {
                    result[result_id] = string[string_id];
            }
            string_id += 2;
        }

        result[result_id] = '\0';
    }

    return max_len; // 返回最长回文子串的长度
}
```