

# 寻找最大重复子串

江苏金陵中学 林希德

## 【关键字】

后缀树 KMP 推广算法 最优子串

## 【摘要】

关于字符串处理，无论是国内的个大竞赛还是中文的经典宝书，都相对较少涉及。曾见各位高手在信息学论坛上就某前辈提出的此类问题讨论得热火朝天，于是我决心翻阅多方资料仔细研究，仅望对大家发表高见起抛砖引玉的小作用。

本文第一章给出了问题的详细描述和我对该问题的拙见，第二、三章简述了字符串处理的两个常用算法——后缀树和 KMP，第四章阐明了寻找最大重复子串的主算法。

## [目录]

- 一 [问题提出和初步认识](#)
  - 1.1 [问题描述](#)
  - 1.2 [初步分析](#)
- 二 [后缀树](#)
  - 2.1 [后缀树的定义](#)
  - 2.2 [后缀树的构建](#)
  - 2.3 [后缀链接](#)
  - 2.4 [性能分析](#)
- 三 [模式匹配](#)
  - 3.1 [朴素模式匹配](#)
  - 3.2 [KMP模式匹配](#)
  - 3.3 [前缀函数](#)
  - 3.4 [性能分析](#)
  - 3.5 [KMP 推广算法](#)
- 四 [主算法](#)
  - 4.1 [问题转化](#)
  - 4.2 [字符串分解](#)
  - 4.3 [范围限定](#)
  - 4.4 [找到循环节](#)
  - 4.5 [辅助函数](#)
  - 4.6 [性能分析](#)
- 五 [论文小结](#)

## [正文]

## 一、问题提出和初步分析

## 1.1 章节

## 定义一：

对于字符串 $S$ ，如果存在正整数 $p$ 使得

$$\forall 1 \leq x \leq |S| - p: S_x = S_{x+p}.$$

那么称 $p$ 是 $S$ 的**循环周期**(Period)， $e = |S| / p$ 是 $S$ 的**指数**(Power)，任何长度为 $p$ 的 $S$ 的子串都是 $S$ 的一个**循环节**(Repetend)，特别的，当 $e \geq 2$ 时， $S$ 是个大小为 $p$ 的**重复字符串**(Repetition or P-power word)。

## 问题描述：

给出一个由大写字母组成的字符串 $W$ ， $W$ 长度为 $1 \leq n \leq 10^5$ ，请在 $W$ 的所有子串中找出循环周期最长的那个重复字符串（Finding Maximal Repetition），即**最大重复子串**。

## 1.2 章节

## 定义二：

为方便表达，我们用符号 $W(u,v)$ 表示开始于位置 $u$ 结束于位置 $v$ 的 $W$ 的子串。

## 初步分析：

一个 $O(n^2)$ 的算法是乍一看最直观的收获。我们可以首先从 $n$ 到1枚举循环周期 $p$ ，然后针对 $p$ 从位置1到 $n$ 搜索重复子串，一旦找到立即输出并退出循环，如下：

```

For p = n → 1 do
  For i = 1 → n do
    If i > p 并且  $W_i = W_{i-p}$  then
       $G_i = \text{Max}(G_{i-1} + 1, 1)$ 
    Else
       $G_i = 0$ 
    End if
    If  $G_i = p$  then
      输出  $W(i-2p+1, i)$ ；退出循环
  
```

这个 $O(n^2)$ 的算法简单易编，但速度较慢。那么，怎样让复杂度从 $O(n^2)$ 降到 $O(n)$ 呢？为说清楚这个线性算法，我们不得不先介绍一种关于字符串处理的数据结构——后缀树，还有就是模式匹配的KMP。

## 二、后缀树

### 2.1 章节

#### 后缀树定义：

令  $W = W + 'S'$ ，这样  $W$  的最后一个字符从未在前面的任何一个位置上出现过。添置 ' $S$ ' 的目的，将在下一小节中予以说明。

后缀树其实就是一棵**检索树**(Trie)，和普通检索树一样我们不断往树中插入字符串和添置顶点，只不过，后缀树还有一些特殊的性质：

i) 我们从大到小依次往树中插入  $W$  的后缀，这其实是后缀树的名称由来，也是它的**构建过程**。

ii) 树上每条边  $E(u, v)$  有两个参数  $r$  和  $\lambda$ ，记为  $E(u, v) = (r, \lambda)$ ，代表子串  $W(r, \lambda)$ 。

iii) 树上每个节点均有 26 个儿子，代表 26 个大写字母。如果父亲节点  $u$  的第 1 个儿子是节点  $a$ ，那么  $E(u, a)$  上的子串一定以字母  $A$  开头；如果第 2 个儿子是节点  $b$ ，那么  $E(u, b)$  上的子串一定以字母  $B$  开头……以此类推。

#### 定义三：

如果从根  $Root$  到节点  $x$  途径若干条边，将这些边上的子串顺次连接得到字符串  $S_x$ ，那么称  $S_x$  是  $x$  的**对应子串**， $x$  是  $S_x$  的**对应节点**。易知，“节点 对应子串”和“字符串 对应节点”都是一一映射的关系。

iv) 后缀树有  $n$  个叶子节点  $Leaf_1, Leaf_2, \dots, Leaf_n$ ， $Leaf_i$  的对应子串实际就是  $W$  的后缀  $W(i, n)$ 。这一点通过性质 i) 就可以得到，这里无非明确一下。

### 2.2 章节

#### 定义四：

$Head_i$  是  $W(i, n)$  和  $W(j, n)$  的最长公共前缀，其中  $j$  是小于  $i$  的任意正整数， $Tail_i$  使得  $Head_i + Tail_i = W(i, n)$ 。

#### 定义五：

只要子串  $S = W(u, v)$  满足  $u \leq x$ ，我们就说  $S$  在**范围  $x$  内出现过**。 $Head_i$  其实就是在范围  $i-1$  内出现过的  $W(i, n)$  的最长前缀。

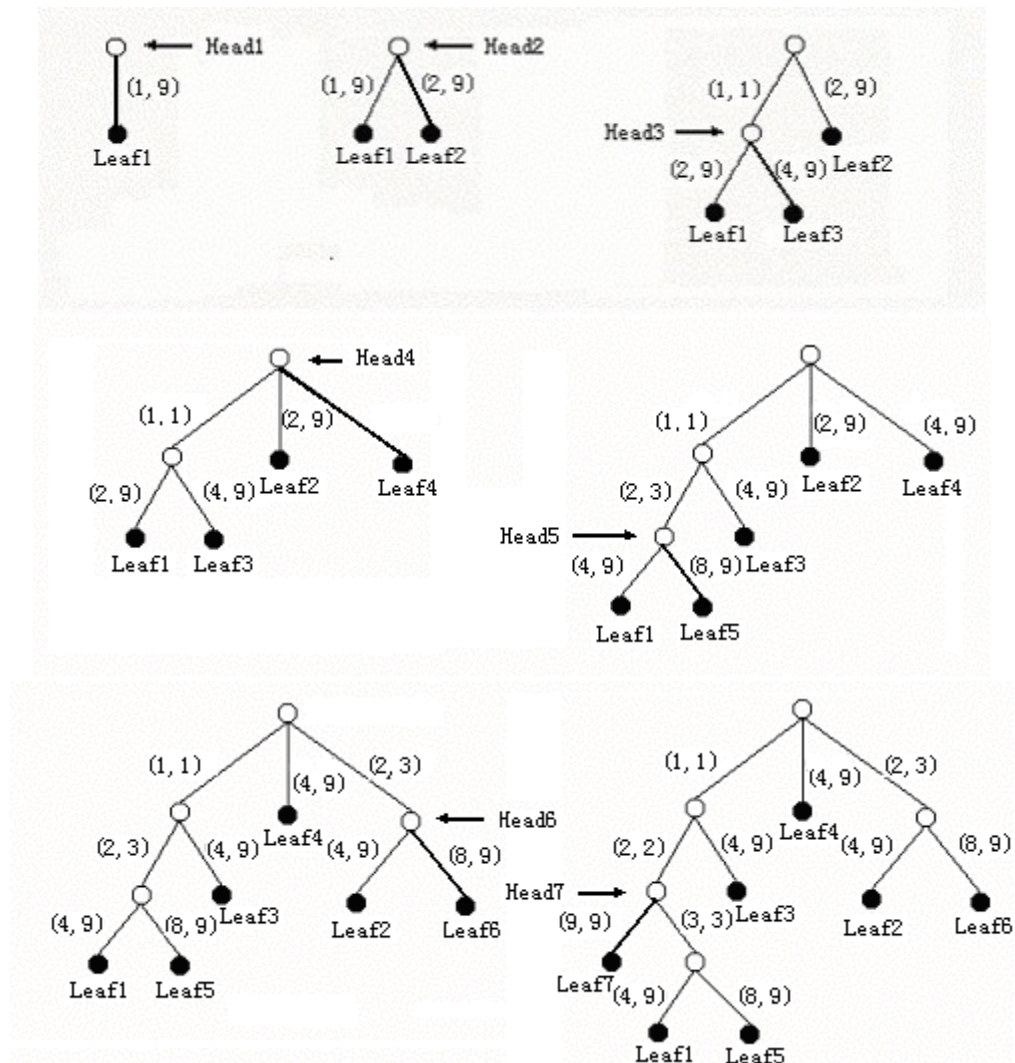
#### 后缀树的构建：

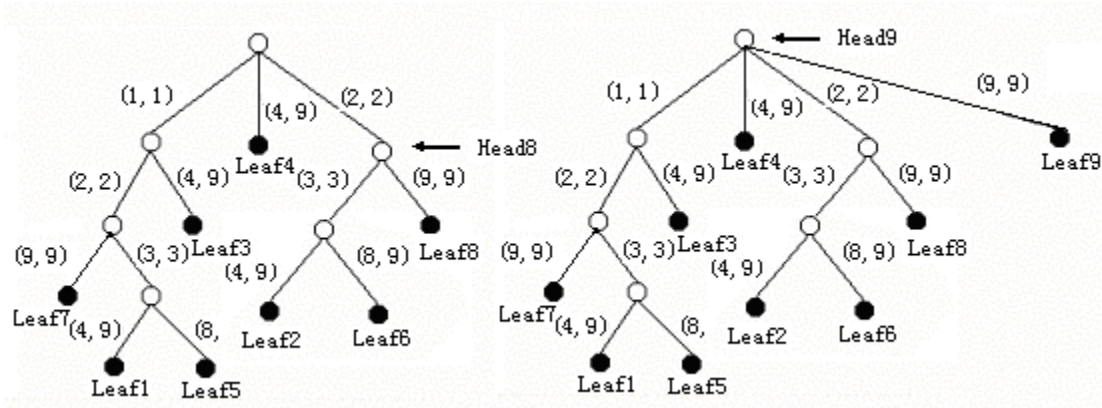
```

初始化后缀树为只有根结点的树
For i = 1 → n do
    找到  $Head_i$  的对应节点  $h_i$ ；
    增加叶节点  $Leaf_i$ ，使得  $E(h_i, Leaf_i) = (|Head_i|+1, n)$ ；
End for
  
```

### 举例一：

下面让我们来看看字符串  $W = \text{"AGATAGAG\$"}$  的后缀树被逐步构建的具体例子，其中黑色圆圈是叶节点，圆括号内的是边参数  $(r, \lambda)$ ：





增加Leaf<sub>i</sub>只是O(1)的工作，关键问题是如何找到Head<sub>i</sub>的对应节点h<sub>i</sub>。最直观也是最野蛮的方法是从根节点开始对W(i,n)实行**逐个字符地扫描**：

```

hi = Find(Root, W(i,n)) ;
Function Find(实参：开始节点 p; 字符串 S): 指针类型;
While true do
    e = p 的第 ord(S1) - 64 个儿子节点;
    If e 不存在 then
        返回指针 p, Break;
    End if
    逐个字符比较找出 S 与 E(p, e) 的最长公共前缀 U
    L = |U|
    IF L < |S| then
        p = e
        S = S(L+1, |S|)
    Else
        增加新节点 q, 令 V = E(p, e)
        令 q 成为 p 的儿子, E(p, q) = V(1, L)
        令 e 成为 q 的儿子, E(q, e) = V(L+1, |V|)
        返回指针 q, Break;
    End if
End while
End function

```

请看 [图 10.1.1](#)，或许你会担心V(L+1, |V|)可能是个无意义的空串，其实，L = |V|的情况一定不会出现。

**反证法一：**

如果L = |V|，那么就说明W(i,n)是某个W(j,n) (1 ≤ j < i) 的前缀，也就是说W的末尾字符S'至少出现过两次。这与[图 10.1.1](#)的规定违背。

故，任何增添进后缀树中的**边都不是空串**。

这种野蛮的扫描算法使得时间复杂度高达O(n<sup>2</sup>)，还外加一个不小的常数因子。其实，这个

Find函数并不是完全的没有用处，只是，为加快寻找 $h_i$ 的速度我们需要使用辅助结构——**后缀链接**。

### 2.3 章节

#### 后缀链接的定义 (McCreight Arithmetic) :

令 $Head_{i-1} = az$ ，其中 $a$ 是字符串 $W$ 的第 $i-1$ 位字符。由于 $z$ 在范围 $i$ 内出现过至少两次，所以一定有 $|Head_i| \geq |z|$ ， $z$ 是 $Head_i$ 的前缀。所谓 $h_{i-1}$ 的后缀链接 (Suffix Link) 实际是由 $h_{i-1}$ 指向 $z$ 对应节点 $d$ 的指针 $Link_{h_{i-1}}$ 。当然， $z$ 有可能是空串，此时 $Link_{h_{i-1}}$ 由 $h_{i-1}$ 指向根节点 $Root$ 。

#### 创建方法:

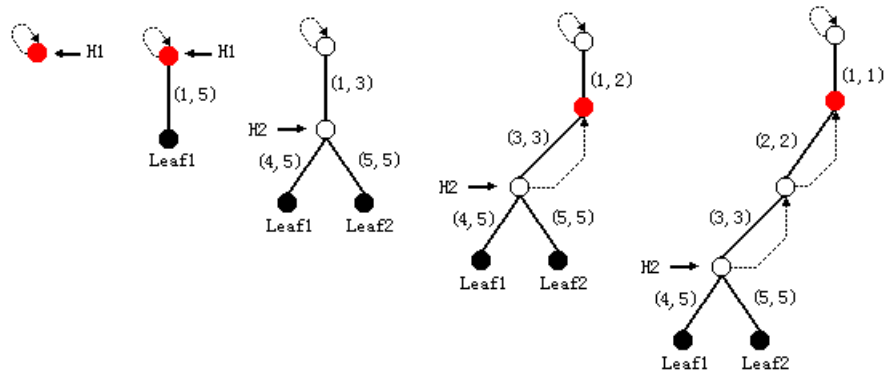
通过后缀链接，我们在 $O(1)$ 时间内找到节点 $d$ ，然后再使用函数 $Find(d, W(i+|z|, n))$ 找到 $h_i$ ，速度自然要快很多倍。现在的问题是，怎样在每次建立节点 $h_i$ 后创建 $Link_{h_i}$ 呢？规定：

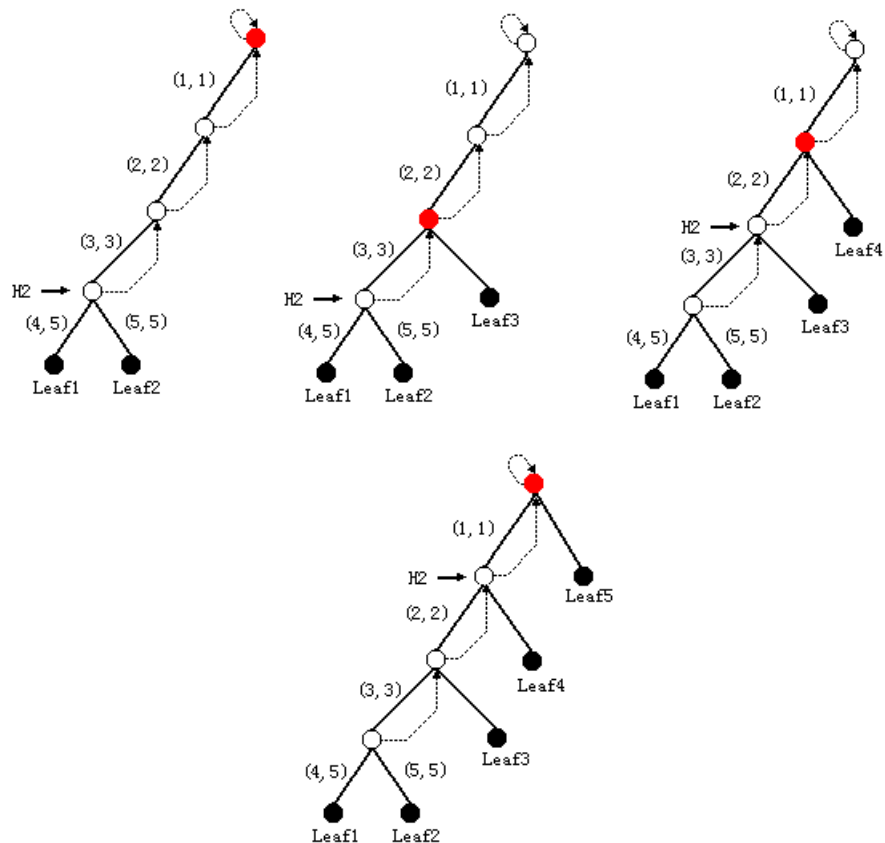
- 1) 根节点 $Root$ 的后缀链接指向它自身
- 2) 任何一个非叶节点的后缀链接都在该节点出现以后被立即创建

根据这两条规定， $h_i$ 的父亲节点 $c$ 的后缀链接一定存在。我们从 $Link_c$ 出发，**向下搜索**，即可找到 $z$ 的对应节点 $d$ 。

#### 举例二:


下面让我们来看看字符串 $W = "AAAA\$"$ 是如何利用后缀链接加快寻找 $h_i$ 的速度的。图中，虚线箭头表示相应节点的后缀链接，黑色圆圈是叶子节点，红色圆圈是当前后缀链接指向的节点。






创建后缀链接的伪代码如下：

	<pre> p = h<sub>1</sub> While Link p 不存在 do     c = p 的父亲节点     V = E(c, p)     If c = Root then         Link p = Down(Link c, V(2,  V )) </pre>
函数	<pre> Function Down(实参: 开始节点 p; 字符串 S): 指针类型;     While true do         e = p 的第 ord(S<sub>1</sub>) - 64 个儿子节点;         If e 不存在 or S 是空串 then             返回指针 p, Break;         End if         L = Min( S ,  E(p, e) )         IF L ≤  S  then             p = e             S = S(L+1,  S )         Else             ➡ 增加新节点 q, 令 V = E(p, e)             令 q 成为 p 的儿子, E(p, q) = V(1, L)             令 e 成为 q 的儿子, E(q, e) = V(L+1,  V )             返回指针 q, Break;         End if     End while End function </pre>

观察一下就会发现，函数 Down 和 Find 几乎一模一样，只不过  处，L 不是通过逐个的字符比较而是直接的长度比较得出。为什么呢？

**反正法二：**

令  $L = |S \text{ 和 } E(p, e) \text{ 的最长公共前缀}|$ 。

如果  $L < \min(|S|, |E(p, e)|)$ ，那么就证明 z 在范围 i 内从未出现过。这于  处的结论矛盾。

故，逐个字符比较的结果一定是  $L = \min(|S|, |E(p, e)|)$ 。

## 2.4 章节

**算法主框架回顾：**

回顾和整理一下算法主框架，看看我们究竟做了些什么？

```

For i = 1 → n do
    步骤 1、函数 Find 从 Link  $h_{i-1}$  开始向下搜索找到节点  $h_i$ 
    步骤 2、增添叶子节点  $Leaf_i$ 
    步骤 3、函数 Down 创建  $h_i$  的后缀链接 Link  $h_i$ 
End for
  
```

**后缀树性能分析：**

接着刚才文本框内的伪代码来谈论。对于给定的 i，步骤 2 的复杂度为  $O(1)$ ，但由于无法确定 Link  $h_{i-1}$  到  $h_i$  之间的节点个数，所以不能保证步骤 1 总是线性的。

局部估算失败，不妨从整体入手。有一点是肯定的，那就是  $i + |Head_i|$  总随着 i 的递增而递增。因此，W 中的每个字符只会被 Find 函数遍历 1 次，**总体复杂度是  $O(n)$  的。**

## 三、模式匹配的 KMP 算法

如何判断字符串 T（称 T 为模式）是否是字符串 S（称 S 为主串）的子串呢？

至今为止，解决这个问题最优秀的算法是 1xxx 年由 D. E. Knuth、J. H. Morris 和 V. R. Pratt 共同提出的模式匹配 KMP 算法。大多数选手早已熟练掌握 KMP，故，我在此仅为论文的完整性对它



进行简要叙述。

### 3.1 章节

为方便说明，我们令  $n = |S|$ ， $m = |T|$ 。

#### 朴素的模式匹配算法

朴素模式匹配的基本思想是：将主串  $S$  的第一个字符和模式  $T$  的第一个字符进行比较，若相等则进一步比较二者的后续字符；否则，从  $S$  的第二个字符开始重新和  $T$  的第一个字符进行比较……以此类推，直至模式  $T$  和主串  $S$  中的某一个子串相等，则称匹配成功，否则称匹配失败。该算法复杂度是  $O(nm)$ ，显然令我们很不满意。

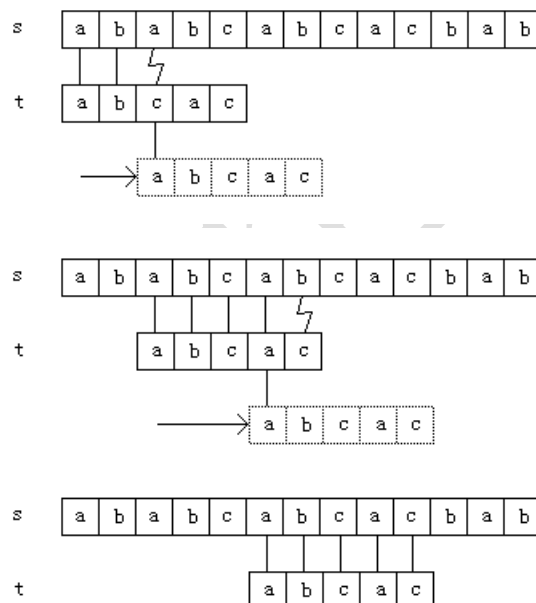
### 3.2 章节

#### KMP模式匹配

如果说索引指针指向了当前有待比较的  $S$  和  $T$  中的两字符，那么朴素模式匹配算法效率不高的主要原因就是没有充分利用匹配过程中已经得到的部分匹配信息，而任由索引指针回溯。KMP 正是针对这点缺陷对朴素算法做了实质性的改进，它主要的思想是：当出现字符比较不相等时，让模式在主串上尽可能的向右滑动，然后接着进行比较。

#### 举例三：

例如下图，其中， $S = \text{"a b a b c a b c a c b a b"}$ ， $T = \text{"a b c a c"}$ ， $\swarrow$  表示索引指针。



### 3.3 章节

## 前缀函数

一切问题集中在如何让模式T尽可能向右滑动上。

假设当前索引指针正向右移动，在它指向字符  $T_j$  和  $S_i$  时发现  $T_j \neq S_i$ ，那么对于一切正整数  $k$  ( $i - j + 1 < k \leq n - m + 1$ )，满足  $S(k, k + m) = T$  的必要条件（记为※号）是  $k \leq i$  或者  $T(1, i - k)$  是  $T(1, j - 1)$  的后缀。为防止重复或者遗漏，模式T向右滑动的距离应当等于  $\text{Min}\{x: i - j + 1 + x \text{ 满足必要条件※}\}$ 。因为空串  $T(1, i - i)$  一定是  $T(1, j - 1)$  的后缀，所以必要条件中的  $k \leq i$  完全可以省略。

KMP算法的核心就是：令  $\pi[j-1] = \text{Max}\{x: T(1, x) \text{ 是 } T(1, j-1) \text{ 的后缀}\}$ ，然后将模式T向右滑动，使得索引指针指向  $S_i$  和T的第  $\pi[j-1] + 1$  个字符。 $\pi[i]$  的大小与主串S并无关联，所以我们应该事先通过预处理求出并保存所有  $\pi[i]$  ( $1 \leq i \leq m$ ) 值，即所谓的前缀函数。过程如下：

```

 $\pi[0] = -1$ 
For  $i = 1 \rightarrow m$ 
   $K = \pi[i-1]$ 
  While ( $K > 0$ ) 并且 ( $t$  的第  $K+1$  个字符  $\neq t$  的第  $i$  个字符)
     $K = \pi[K]$ 
  End while
   $\pi[i] = K + 1;$ 
End for

```

有了前缀函数，KMP主算法就如下方框所述般简单明了：

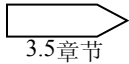
```

 $Q = 0$ 
For  $i = 1 \rightarrow m$ 
  While ( $Q > 0$ ) and ( $t$  的第  $Q+1$  个字符  $\neq s$  的第  $i$  个字符)
     $Q = \pi[Q]$ 
  End while
  If  $t$  的第  $Q+1$  个字符 =  $s$  的第  $i$  个字符 then
     $Q = Q + 1$ 
    If  $Q = m$  then
      找到了模式匹配，退出循环。
    End if
  End if
End for

```

由于不清楚  $\pi$  到底循环了多少次，所以前缀函数的复杂度似乎不好分析。不过，因为  $K$  总是  $\leq 0$  的，所以“操作  $K = \pi[K]$  造成的  $K$  值减少量”一定不会比“操作  $K = \pi[i-1]$  和  $\pi[i] = K + 1$  造成的  $K$  值增加量”多，因而  $\pi$  处循环总次数  $\leq m$ 。

主算法复杂度的证明同上，亦为线性，所以 KMP 算法的时间复杂度同空间复杂度一样，均为  $O(n+m)$ 。



定义六：

函数  $B_i = |T \text{ 与 } S(i,n) \text{ 的最长公共前缀}|$ ，这里  $1 \leq i \leq n$ 。

### KMP 推广算法

普通 KMP 算法只是在判断是否存在整数  $i$  满足  $B_i = m$ ，但如果希望求出所有  $B_1 \rightarrow B_n$  的函数值并保持复杂度不变，又该怎么办呢？很自然的，我们继承经典 KMP 的核心思想但尝试对其进行改造以得到推广算法。

定义七：

函数  $A_i = |T \text{ 与 } T(i,m) \text{ 的最长公共前缀}|$ ，这里  $2 \leq i \leq m$ 。

推广算法将借助函数  $A$  求出函数  $B$ 。

### 函数 $A$ 的求解

函数  $A$  的求解类似于数学归纳法。

- 1、首先通过逐个的比较字符求出  $A_2$ ，同时令  $k = 2$ 。
- 2、假设已经求出函数  $A_2, A_3 \dots A_{i-1}$ ，并且知道  $k$  满足  $1 < k < i$ ， $k + A_k$  最大。现在我们希望借助  $A_2 \rightarrow A_{i-1}$  求出函数  $A_i$ 。
- 3、令  $Len = k + A_k - 1$ ， $L = A_{i-k+1}$ ，然后分类讨论：
  - a) 如果  $Len \leq i$  并且  $L < Len - i + 1$ ，那么  $A_i = L$ 。  
理由：因为子串  $T(i, k + A_k - 1) = \text{子串 } T(i - k + 1, A_k)$ ，所以字符  $T_{i+L} = T_{i-k+1+L} \neq T_L$ ， $A_i$  延伸到长度  $L$  时恰好不能匹配。
  - b) 如果  $Len \leq i$  并且  $L \leq Len - i + 1$ ，那么  $A_i \leq L$ 。  
理由同上。  
此时，我们先令  $A_i = L$ ，然后通过逐个的比较字符将  $A_i$  延伸到它应有的函数值大小，最后赋值  $k = i$ 。
  - c) 如果  $Len < i$ ，那么我们无法通过  $A_2 \rightarrow A_{i-1}$  得到任何关于  $A_i$  的信息。  
此时，我们先令  $A_i = 0$ ，然后通过逐个的比较字符将  $A_i$  延伸到它应有的函数值大小，最后赋值  $k = i$ 。

### 复杂度分析

无论哪种情况， $k + A_k$ 的值都只增不减——这就是KMP算法的核心思想——所以逐个比较字符的总次数是 $O(m)$ 的，求函数A的复杂度为 $O(m)$ ，过程如下：

```

j = 0
While 字符  $T_{1+j}$  = 字符  $T_{2+j}$  do
    j = j + 1
End While
 $A_2 = j$ ,  $k = 2$ 
For i = 3  $\rightarrow$  m do
    Len = k +  $A_k$  - 1, L =  $A_{i-k+1}$ 
    If L < Len - i + 1 then
         $A_i = L$ 
    Else
        j = Max(0, Len - i + 1)
        While 字符  $T_{i+j}$  = 字符  $T_{1+j}$  do
            j = j + 1
        End While
         $A_i = j$ ,  $k = i$ 
    End if
End for

```

### 函数B的求解

函数B的求解和函数A的求解几乎一模一样，因而我就不再赘言而只给出求解过程的伪代码以供参考：

```

j = 0
While 字符  $T_{1+j}$  = 字符  $S_{1+j}$  do
    j = j + 1
End While
 $B_1 = j$ ,  $k = 1$ 
For i = 2  $\rightarrow$  n do
    Len = k +  $B_k$  - 1, L =  $A_{i-k+1}$ 
    If L < Len - i + 1 then
         $B_i = L$ 
    Else
        j = Max(0, Len - i + 1)
        While 字符  $T_{1+j}$  = 字符  $S_{i+j}$  do
            j = j + 1
        End While
         $B_i = j$ ,  $k = i$ 
    End if
End for

```

函数B有着和函数A一样的复杂度分析，所以求解函数B的复杂度是 $O(n)$ 的。

综上，我们彻底阐述完了KMP推广算法的整个求解过程，推广算法的复杂度和普通KMP一样均是 $O(n+m)$ 的。

## 四、主算法

### 4.1 章节

#### 最优子串的定义：

对于子串  $S = W(u, v)$  和正整数  $L$ ，如果满足：

- 1)  $S$  是循环周期为  $L$  的重复子串
- 2)  $S$  不能向左扩展，即  $u = 1$  或者  $W(u-1, v)$  不满足条件1)
- 3)  $S$  不能向右扩展，即  $v = n$  或者  $W(u, v+1)$  不满足条件1)

那么称  $S$  是一个周期为  $L$  的**最优子串**。

#### 举例三：

例如当  $W = \text{"ABAABABAABAABA"}$  时，“ABABA”是周期为2的最优子串，而“ABAB”或者“BABA”则不是。“ABAABABAABA”是周期为5的最优子串，而“ABAABABAAB”或者“BAABABAABA”则不是。需要说明的是：即便周期相同，甚至部分重叠，两个不同子串也可能同时是最优子串。例如前缀“ABAABA”和后缀“ABAABAABA”就都是周期为3的最优子串，尽管它们有重叠部分  $W(4, 3)$  是。

#### 问题的转化：

如果我们能够**求出所有最优子串连同它们的周期**，那么，从中找出周期最大的那个最优子串，最大重复子串的问题便迎刃而解。

怎样才能找出最优子串  $S$  呢？算法的主要思想是：

- 1、找到一个长度为  $L$  的循环节  $D$
  - 2、分别将  $D$  向左和向右扩展到不能扩展为止
  - 3、判断扩展以后的  $D$  是否长度  $\geq 2L$ 。
- 如果是，便找到了一个周期为  $L$  的最优子串  $S$ 。

以下，4.2、4.3 章节将说明如何寻找循环节  $D$ ，4.4 章节将说明如何将  $D$  快速地向两边扩展，4.5 章节将进行总结。

### 4.2 章节

**字符串分解：**

将W分解成  $W = U_1 + U_2 + U_3 + \dots + U_m$  的形式，其中 $U_i$ 定义如下：

$$P = U_1 + U_2 + \dots + U_{i-1}$$

Q 是 W 除去 P 的剩余部分，即  $W = P + Q$

如果字母 $Q_1$ 从未在范围|P|中出现过，

那么  $U_i = Q_1$  单个字母组成的字符串

否则  $U_i =$  范围 $i-1$ 中出现过的Q的最长前缀

**举例四：**

字符串  $W = \text{"ABAABABAABAAB"}$  应该被怎样划分呢？

第一步：显然  $U_1 = \text{"A"}$ ；

第二步：P = "A"，Q = "BAABABAABAAB"， $W_2$  未在范围|P|中出现过，所以  $U_2 = \text{"B"}$ ；

第三步：P = "AB"，Q = "AABABAABAAB"， $W_3$  在范围|P|中出现过，所以  $U_3 = PQ$  的最长公共前缀 "A"；

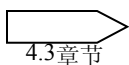
第四步：P = "ABA"，Q = "ABABAABAAB"， $W_4$  在范围|P|中出现过，所以  $U_4 = PQ$  的最长公共前缀 "AB"；

第五步：P = "ABAAB"，Q = "ABAABAAB"， $W_5$  在范围|P|中出现过，所以  $U_5 = PQ$  的最长公共前缀 "ABAAB"；

第六步：P = "ABAABABAAB"，Q = "AAB"， $W_{11}$  在范围|P|中出现过，所以  $U_6 = PQ$  的最长公共前缀 "AAB"；

最终，W 被划分成为 |A|B|A|AB|ABAAB|AAB|。

你大概已经知道，**字符串分解是利用后缀树算法实现的**，因为 $U_i$ 要么是单个的字符，要么就等于 $\text{Head}_{|P|+1}$ 。至此，我们终于可以诠释后缀树的作用。但是问题又来了，你不禁想问字符串分解的意义何在？



任何一个最优子串 S 的结束位置  $\text{End}(S)$  一定在某个片段  $U_i$  之内。如果我们暂且假设  $i$  是个已知常量，那么  $|S|$  和周期  $L$  是否都有一定的范围限制？为回答这些问题，我们需要对 S 的**存在形式**进行前后**4层**的分类讨论。

**第1层：**

$\text{End}(S)$  在  $U_i$  内， $\text{Ini}(S)$  又在何处？

情况 1)  $\text{Ini}(S) < \text{Ini}(U_i)$

情况 2)  $\text{Ini}(S) \boxed{\phantom{0000}} \text{Ini}(U_i)$

由于  $U_i$  是范围 P 内出现过的 Q 的最长前缀，所以被  $U_i$  包含的子串 S 一定在范围 P 内出现过。

情况 2) 实际可以被情况 1) 包含。为避免重复劳动，**情况 2) 将被忽略。**

### 定义七:

我们把  $S(|S|-L+1, |S|)$  称为  $S$  的**最末循环节**，用符号  $R$  表示。

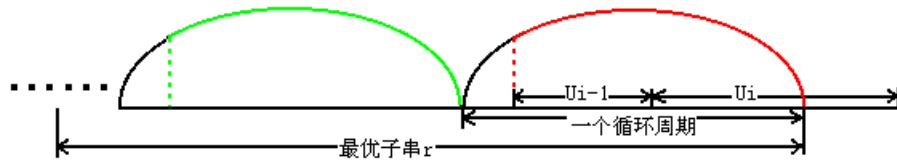
### 第2层:

既然  $\text{Ini}(S) < \text{Ini}(U_i)$ ，那么  $\text{Ini}(R)$  又应当在何处呢？

情况甲)  $\text{Ini}(R) > \text{Ini}(U_{i-1})$

情况乙)  $\text{Ini}(R) \boxed{<} \text{Ini}(U_{i-1})$

请看下图，图中一段弧线表示一个循环节：



显然，红色和绿色弧线标示了相同的子串，所以根据字符串分解的定义  $U_{i-1}$  应当等于红色弧线或者长度更长的子串。但事实上， $|U_{i-1}| < \text{红色弧线的水平长度}$ 。矛盾！**情况甲不会出现。**

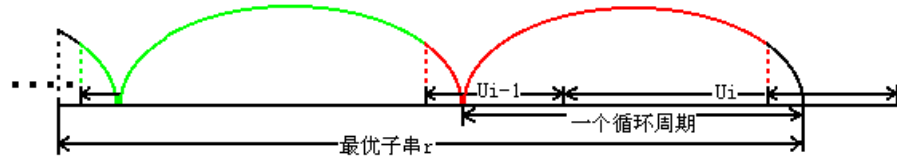
### 第3层:

虽然已经证明  $\text{Ini}(S) < \text{Ini}(U_i)$ ，不过我们还想进一步知道有关  $\text{Ini}(S)$  的信息。

情况 A)  $\text{Ini}(S) - \text{Ini}(U_{i-1}) < |U_{i-1} + U_i|$

情况 B)  $\text{Ini}(S) - \text{Ini}(U_{i-1}) \boxed{>} |U_{i-1} + U_i|$

再请看下图，图中所示为情况 B)：



因为周期  $L < |U_{i-1} + U_i|$ ，所以  $\text{Ini}(S) - \text{Ini}(U_{i-1}) > L$ 。同样道理，红色和绿色弧线标示了相同的子串，根据字符串分解的定义  $U_{i-1}$  应当等于红色弧线或者长度更长的子串。矛盾！**情况 B) 不存在。**

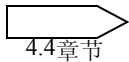
讨论完毕，综上所述，我们得出一个**重要结论**：

如果令  $V = \text{长度为 } |U_i| + 2|U_{i-1}| \text{ 的 } P \text{ 后缀，}$

$$U = U_i$$

那么，最优子串  $S$  的开始位置在  $V$  内，结束位置在  $U$  内。

至此，我们终于明白：字符串分解的重大意义在于**严格限制了最优子串的存在形式**。



讨论了这么多，我们还是没有把S的循环节给找出来。其实，只要再进一小步分类，循环节就出来了。


#### 第4层：

因为  $\text{Ini}(S) < \text{Ini}(U_i) \leq \text{End}(S)$  并且  $|S| \leq 2L$ ，所以以下两种情况S必然至少居其一：


情况i) S在V中的长度  $\geq L$

情况ii) S在U中的长度  $\geq L$



情况i) 和情况ii) 类似，为简便表达，我们仅以情况ii) 为例进行说明。易知， $U(1, L)$  就是S的一个循环节。

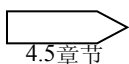
上述4层分类都基于  处的假设，现在把假设去掉，并整理一下得：

```

Answer = 0
For i = 2 → m do
    P = U1 + U2 + ... + Ui-1
    V = 长度为 |Ui| + |Ui-1| * 2 的 P 的后缀
    U = Ui
    For L = 1 → |U| do
         确定 U(1, L) 为循环节 R
        将 R 向左向右扩展到不能扩展为止
        判断扩展以后的R是否 |R| ≤ 2L,
            如果是，那么 Answer = Max(Answer, R(1, 2L))
    . . .

```

 枚举L的复杂度是  $O(|U|)$ ，而U的总长度又是  $O(n)$  的，所以我们必须在  $O(1)$  时间内完成  处的工作，才能保证算法总复杂度是线性的。



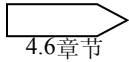
#### 定义辅助函数：

$Lp_L = U$  与  $U(L+1, |U|)$  的最长公共前缀

$Ls_L = V$  与  $V + U(1, L)$  的最长公共后缀



所谓将循环节向两边扩展，无非就是求得函数  $L_p$  和  $L_s$ 。如果单独求解某个  $L_{p_L}$  的值是不能保证复杂度是  $O(1)$  的，但局部策略失败，不妨从整体入手，我们通过一次 KMP 推广算法求出所有  $L_{p_1} \rightarrow L_{p_{|u|}}$  的函数值，从而使得平摊复杂度为  $O(1)$ 。



### 主算法性能分析

其实主算法的复杂度已经在刚才几小节中陆陆续续的提到并证明了，现在我们将做的工作只是把这些证明汇总重申一遍。

1、首先是字符串划分。这步工作需要借助辅助结构后缀树，由于后缀树时间空间复杂度均是线性的，所以该步工作也是线性的。

2、然后是辅助函数。 $L_p$  和  $L_s$  的求解借助 KMP 推广算法，对于特定的  $V$  和  $U$ ，该步复杂度为  $O(|V+U|)$ ，因而总体复杂度为  $O(L)$ 。

3、最后是找出最优重复子串  $S$ 。 $S$  分为完整周期在  $v$  和完整周期在  $u$  两种情况，对于每种情况， $S$  可以根据周期  $j$  唯一地确定，由于周期  $j$  的范围是  $O(|V+U|)$  的，所以  $S$  的求解也是线性  $O(|V+U|)$  的。

### [小结]

后缀树和 KMP 尽管巧妙但卑之无甚高论，寻找最大重复子串算法本身也并没有多少可以广泛借鉴的价值。但是，如何将所学算法融会贯通、综合使用，却值得所有人认真思考。

#### 参考文献

- [1] 算法与数据结构（第二版） 傅清祥 王晓东 编著
- [2] The Art of Computer Programming Donald E. Knuth 著
- [3] Handbook of Computer Science and Engineering
- [4] Finding Maximal Repetition Roman Kolpakov 著
- [5] Discrete Applied Mathematics 1989
- [6] jsOI 内部资料