**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](#E1)

1.1[问题描述](#E11)

1.2[初步分析](#E12)

二 [后缀树](#E2)

2.1[后缀树的定义](#E21)

2.2[后缀树的构建](#E22)

2.3[后缀链接](#E23)

2.4[性能分析](#E24)

三 [模式匹配](#E3)

3.1[朴素模式匹配](#E31)

3.2 [KMP模式匹配](#E32)

3.3[前缀函数](#E33)

3.4[性能分析](#E34)

3.5 [KMP推广算法](#E35)

四 [主算法](#E4)

4.1[问题转化](#E41)

4.2[字符串分解](#E42)

4.3[范围限定](#E43)

4.4[找到循环节](#E44)

4.5[辅助函数](#E45)

4.6[性能分析](#E46)

五 [论文小结](#E5)

[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

1  x  |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  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  n  105，请你在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(n2)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 Wi = Wi-p then

Gi = Max(Gi-1 + 1，1)

Else

Gi = 0

End if

If Gi = p then

输出W(i-2p+1,i)；退出循环

End for

End for

这个O(n2)的算法简单易编，但速度较慢。那么，怎样让复杂度从O(n2)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 + '$'，这样W的最后一个字符从未在前面的任何一个位置上出现过。添置'$'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г和，记为E(u,v) = (г,），代表子串W(г,)。

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Sx，那么称Sx是x的**对应子串**，x是Sx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Leaf1、Leaf2… …Leafn，Leafi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：**

**Headi**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，Taili使得Headi + Taili = W(i,n)。

**定义五：**

只要子串S = W(u,v)满足u  x，我们就说S**在范围x内出现过**。Headi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

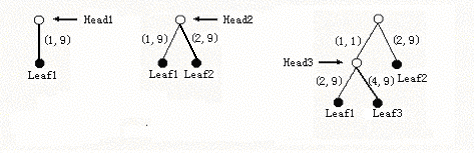
找到Headi的对应节点hi；

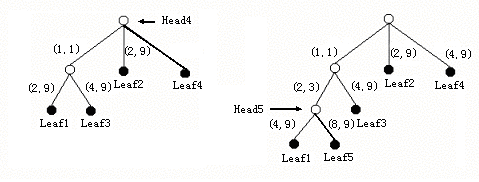
增加叶节点Leafi，使得E(hi,Leafi) = (|Headi|+1,n) ；

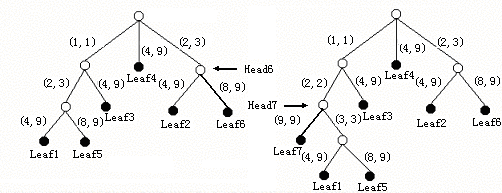
End for

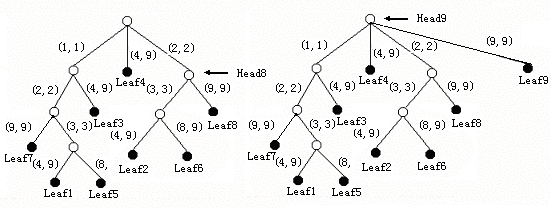
**举例一：**

下面让我们来看看字符串W = “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(г,）：









增加Leafi只是O(1)的工作，关键问题是如何找到Headi的对应节点hi。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 [处](#Red)，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)（1j<i）的前缀，也就是说W的末尾字符'$'至少出现过两次。这与刚才 [处](#Green)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(n2)，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hi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Headi-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|Headi|  |z|，z是Headi的前缀。所谓hi-1的后缀链接（Suffix Link）实际是**由hi-1**

**指向z对应节点d的指针Link hi-1**。当然，z有可能是空串，此时Link hi-1由hi-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hi，速度自然要快很多倍。现在的问题是，怎样在每次建立节点hi后创建Link hi呢？规定：

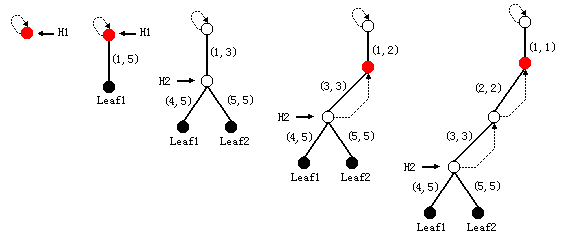
1. 根节点Root的后缀链接指向它自身
2. **任何一个非叶节点的后缀链接都在该节点出现以后被立即创建**

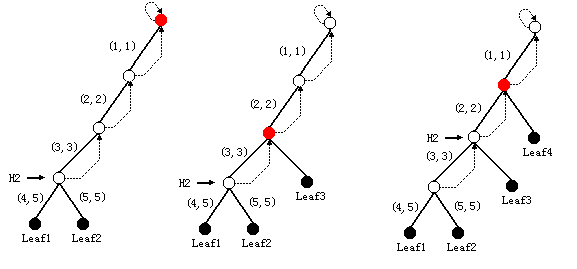
根据这两条规定，hi的父亲节点c的后缀链接一定存在。我们从Link c出发，**向下搜索**，即

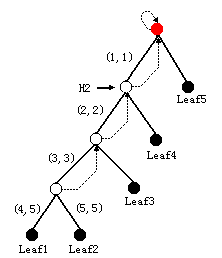
可找到z的对应节点d。

**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hi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：

p = hi

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|))

Else

Link p = Down(Link c，V)

End if

p = Link p

End while

函数Down的伪代码如下：

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(S1) - 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

观察一下就会发现，函数Down和Find几乎一模一样，只不过 [处](#Blue)，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和E(p,e)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](#Cyan)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 hi-1开始向下搜索找到节点hi

步骤2、增添叶子节点Leafi

步骤3、函数Down创建hi的后缀链接Link hi

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 hi-1到hi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 + |Headi|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.**K**nuth、J.H.**M**orris和V.R.**P**ratt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。

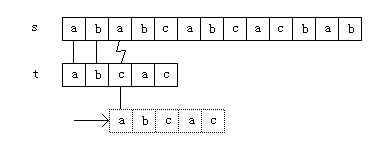
3.2章节

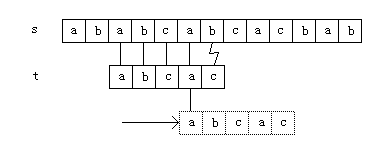
**KMP模式匹配**

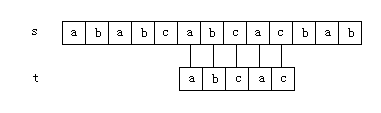
如果说**索引指针**指向了当前有待比较的S和T中的两字符，那么朴素模式匹配算法效率不高的主要原因就是没有充分利用匹配过程中已经得到的部分匹配信息，而任由索引指针回溯。KMP正是针对这点缺陷对朴素算法做了实质性的改进，它主要的思想是：**当出现字符比较不相等时，让模式在主串上尽可能的向右滑动**，然后接着进行比较。

**举例三：**

例如下图，其中，S = “a b a b c a b c a c b a b”，T = “a b c a c”，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 Tj 和 Si 时发现Tj ≠ Si，那么对于一切正整数k（i – j + 1 ＜ k  n – m + 1）， 满足S(k,k + m) = T的必要条件（记为※号）是k  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Min{x：i – j + 1 + x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  i完全可以省略。

KMP算法的核心就是：令[j-1] = Max{x：T(1,x)是T(1,j - 1)的后缀}，然后将模式T向右滑动，使得索引指针指向Si和T的第[j-1] + 1个字符。[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[i]（1 i  m）值，即所谓的**前缀函数**。过程如下：

[0] = -1

For i = 1 🡪 m

K = [i-1]

While (K  0) 并且（t的第K+1个字符 ≠ t的第i个字符）

K = [K]

End while

[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 🡪 m

While (Q > 0) and (t的第Q+1个字符 ≠ s的第i个字符)

Q = [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

3.4章节

**KMP复杂度计算**

由于不清楚 [处](#Purple)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0的，所以“操作K = [K]造成的K值减少量”一定不会比“操作K = [i-1]和[i] = K + 1造成的K值增加量”多，因而 处循环总次数  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)**。

3.5章节

**定义六：**

函数Bi = |T与S(i,n)的最长公共前缀|，这里1in。

**KMP推广算法**

普通KMP算法只是在判断是否存在整数i满足Bi = m，但如果希望求出所有B1🡪Bn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Ai = |T与T(i,m)的最长公共前缀|，这里2im。

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A2，同时令k = 2。
2. 假设已经求出函数A2、A3…Ai-1，并且知道k满足1<k<i，k + Ak最大。现在我们希望借助A2🡪Ai-1求出函数Ai。
3. 令Len = k + Ak - 1，L = Ai-k+1，然后分类讨论：
   1. 如果Leni并且L< Len – i + 1，那么Ai = L。

理由：因为子串T(i，k + Ak - 1) = 子串T(i – k + 1，Ak)，所以字符Ti+L = Ti-k+1+L

≠ TL，Ai延伸到长度L时恰好不能匹配。

* 1. 如果Leni并且LLen – i + 1，那么AiL。

理由同上。

此时，我们先令Ai = L，然后通过逐个的比较字符将Ai延伸到它应有的函数值大小，最后赋值k = i。

* 1. 如果Len<i，那么我们无法通过A2🡪Ai-1得到任何关于Ai的信息。

此时，我们先令Ai = 0，然后通过逐个的比较字符将Ai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 + Ak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T1+j = 字符T2+j do

j = j + 1

End While

A2 = j，k = 2

For i = 3 🡪 m do

Len = k + Ak – 1，L = Ai-k+1

If L < Len – i + 1 then

Ai = L

Else

j = Max(0,Len –i +1)

While 字符Ti+j = 字符T1+j do

j = j + 1

End While

Ai = 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T1+j = 字符S1+j do

j = j + 1

End While

B1 = j，k = 1

For i = 2 🡪 n do

Len = k + Bk – 1，L = Ai-k+1

If L < Len – i + 1 then

Bi = L

Else

j = Max(0,Len –i +1)

While 字符T1+j = 字符Si+j do

j = j + 1

End While

Bi = 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 = “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 = U1 + U2 + U3 + … … + Um 的形式，其中Ui定义如下：

P = U1 + U2 + … … + Ui-1

Q是W除去P的剩余部分，即W = P + Q

如果字母Q1从未在范围|P|中出现过,

那么 Ui = Q1单个字母组成的字符串

否则 Ui = 范围i-1中出现过的Q的最长前缀

**举例四：**

字符串W = “ABAABABAABAAB”应该被怎样划分呢？

第一步：显然U1 = “A”；

第二步：P =“A”，Q =“BAABABAABAAB”，W2未在范围|P|中出现过，所以U2 =“B”；

第三步：P =“AB”，Q =“AABABAABAAB”，W3在范围|P|中出现过，所以U3 = PQ的最长公共前缀“A”；

第四步：P =“ABA”，Q =“ABABAABAAB”，W4在范围|P|中出现过，所以U4 = PQ的最长公共前缀“AB”；

第五步：P =“ABAAB”，Q =“ABAABAAB”，W6在范围|P|中出现过，所以U5 = PQ的最长公共前缀“ABAAB”；

第六步：P =“ABAABABAAB”，Q =“AAB”，W11在范围|P|中出现过，所以U6 = 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Ui要么是单个的字符，要么就等于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？

4.3章节

任何一个最优子串S的结束位置End(S)一定在某个片段Ui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：**

End(S)在Ui内，Ini(S)又在何处？

情况1）Ini(S) < Ini(Ui)

情况2）Ini(S)  Ini(Ui)

由于Ui是范围P内出现过的Q的最长前缀，所以被Ui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。

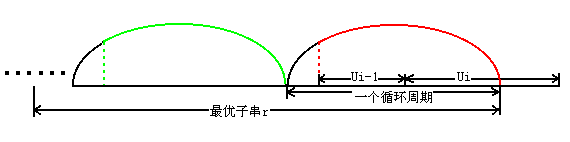
**第2层：**

既然Ini(S) < Ini(Ui)，那么Ini(R)又应当在何处呢？

情况甲）Init(R) > Ini(Ui-1)

情况乙) Init(R)  Ini(Ui-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Ui-1应当等于红色弧线或者长度更长的子串。但事实上，|Ui-1| < 红色弧线的水平长度。矛盾！**情况甲不会出现**。

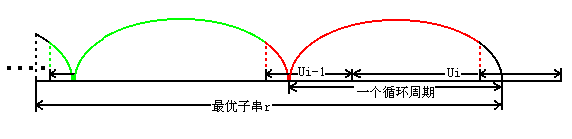
**第3层：**

虽然已经证明Ini(S) < Ini(Ui)，不过我们还想进一步知道有关Ini(S)的信息。

情况A）Ini(S) - Ini(Ui-1) < |Ui-1 + Ui|

情况B）Ini(S) - Ini(Ui-1)  |Ui-1 + Ui|

再请看下图，图中所示为情况B）：



因为周期L < |Ui-1 + Ui|，所以Ini(S)–Ini(Ui-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Ui-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 = 长度为|Ui|+2|Ui-1|的P后缀，

U = Ui

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。**

4.4章节

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Ini(S) < Ini(Ui)  End(S)并且|S|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L

情况ii）S在U中的长度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](#Orange)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))

End for

End for

输出Answer

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**](#Amethyst)**的工作**，才能保证算法总复杂度是线性的。

4.5章节

**定义辅助函数：**

LpL = U与U(L+1,|U|)的最长公共前缀

LsL = 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Lp和Ls。如果单独求解某个LpL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Lp1 🡪 Lp|U|的函数值**，从而使得平摊复杂度为O(1)。

4.6章节

**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。Lp和Ls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