中国计算机学会 China Computer Federation

信息学竞赛中的字符串问题

杭州第二中学 李建

一、马拉车

- 问题：求一个字符串的最长回文子串
- 暴力：枚举中心点，枚举长度，枚举奇偶数
- 优化：在字符与字符中间增加’#’，减少奇偶数的枚举
- 进一步优化：思考减少冗余计算

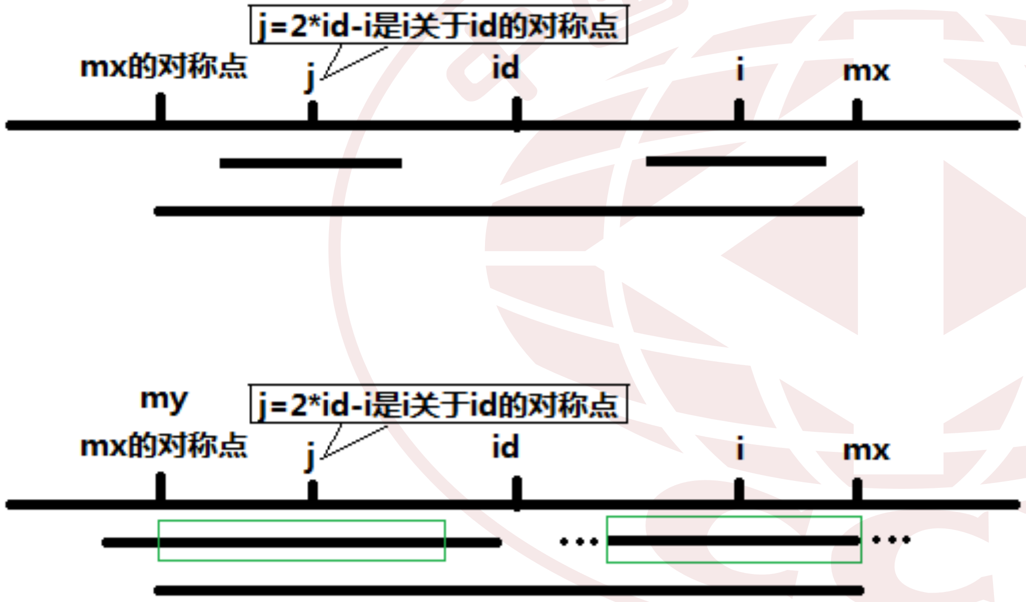
以字符串12212321为例

- 经过上一步，变成了 S[] = "#1#2#2#1#2#3#2#1# “
- P[i] 来记录以字符S[i]为中心的最长回文子串向左/右扩张的长度（包括S[i]，也就是把该回文串“对折”以后的长度）
- S # 1 # 2 # 2 # 1 # 2 # 3 # 2 # 1 #
  P 1 2 1 2 5 2 1 4 1 2 1 6 1 2 1 2 1

- 两个辅助变量

  id 为已知的 {右边界最大} 的回文子串的中心

  mx则为id+P[id]，也就是这个子串的右边界

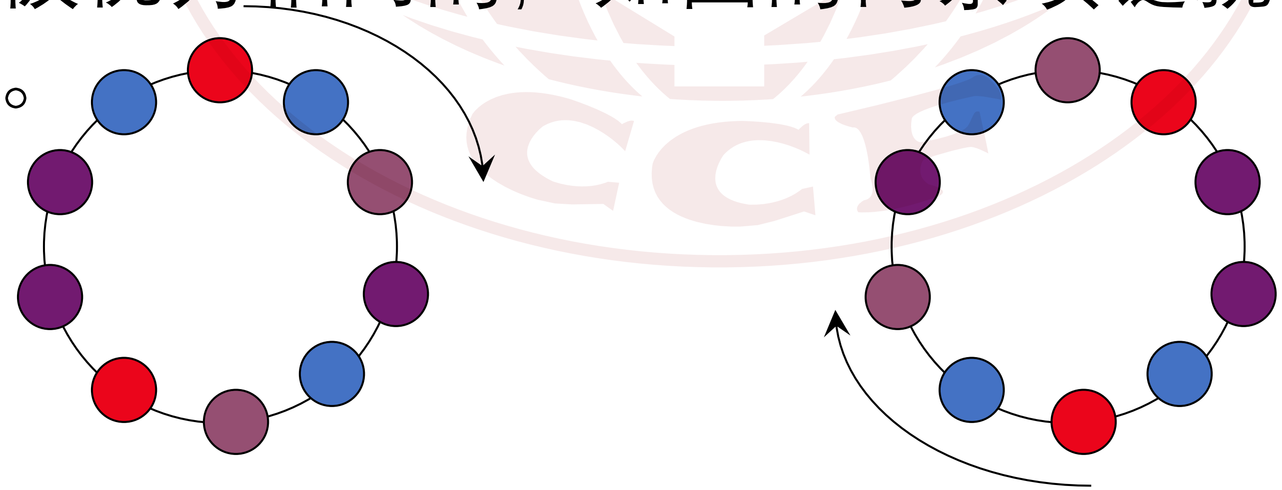


二、最小表示法

有两条环状的项链，每条项链上各有N个多种颜色的珍珠，相同颜色的珍珠，被视为相同。

问题：判断两条项链是否相同。

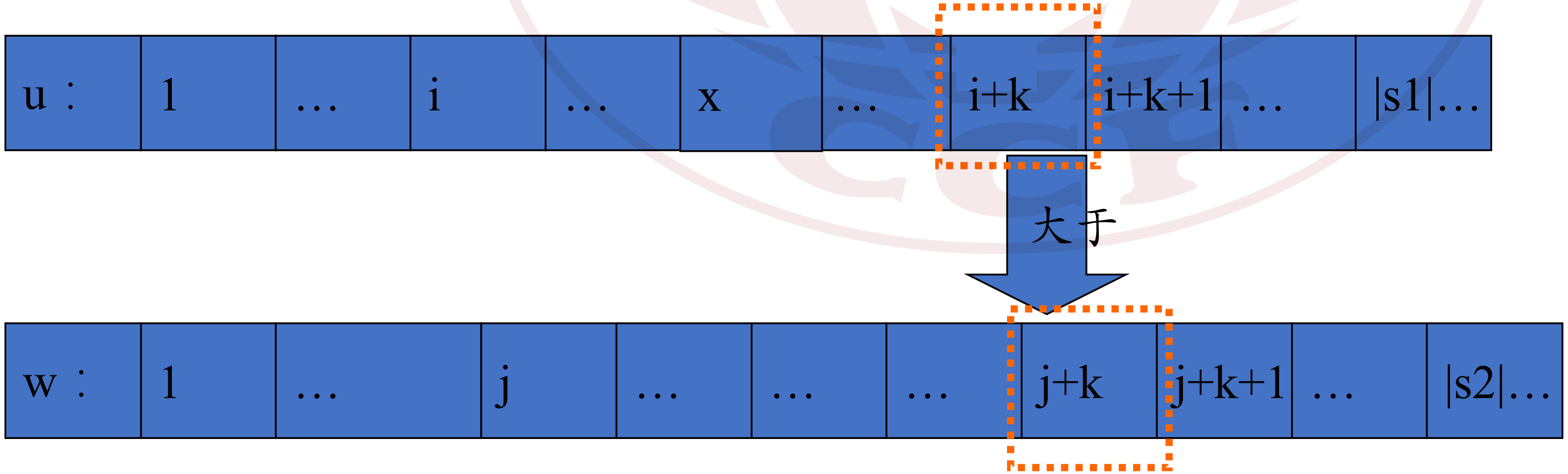
简单分析：由于项链是环状的，因此循环以后的项链被视为相同的，如图的两条项链就是一样的。



- 暴力解法一：固定一个串，枚举另外一个串的起点，逐位比较
- 暴力解法二：把所有的字符串排序，比较最小的字符串是否一样
- 最小表示法：

设指针i,j分别向后滑动k个位置后比较失败(k≥0)，即有u[i+k]≠w[j+k]

设u[i+k]>w[j+k]，同理可以讨论u[i+k]<w[j+k]的情况。



也就是说，两指针向后滑动比较失败以后，指向较大字符的指针向后滑动k+1个位置。（失败位置的下一个位置）

三、KMP

- 给定两个字符串，回答其中一个是否是另一个的子串
- 经典字符串匹配问题，暴力解法同最小表示法

假如，A="abababaababacb"，B="ababacb"，我们来看看KMP是怎么工作的。我们用两个指针i和j分别表示，A[i-j+ 1..i]与B[1..j]完全相等。也就是说，i是不断增加的，随着i的增加j相应地变化，且j满足以A[i]结尾的长度为j的字符串正好匹配B串的前 j个字符（j当然越大越好），现在需要检验A[i+1]和B[j+1]的关系。当A[i+1]=B[j+1]时，i和j各加一；什么时候j=m了，我们就说B是A的子串（B串已经整完了），并且可以根据这时的i值算出匹配的位置。当A[i+1]<>B[j+1]，KMP的策略是调整j的位置（减小j值）使得A[i-j+1..i]与B[1..j]保持匹配且新的B[j+1]恰好与A[i+1]匹配（从而使得i和j能继续增加）。我们看一看当 i=j=5时的情况。

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B = a b a b a c b
j = 1 2 3 4 5 6 7
```

此时，A[6]<>B[6]。这表明，此时j不能等于5了，我们要把j改成比它小的值j'。j'可能是多少呢？仔细想一下，我们发现，j'必须要使得B[1..j]中的头j'个字母和末j'个字母完全相等（这样j变成了j'后才能继续保持i和j的性质）。这个j'当然要越大越好。在这里，B [1..5]="ababa"，头3个字母和末3个字母都是"aba"。而当新的j为3时，A[6]恰好和B[4]相等。于是，i变成了6，而j则变成了 4：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =     a b a b a c b
j =     1 2 3 4 5 6 7
```

从上面的这个例子，我们可以看到，新的j可以取多少与i无关，只与B串有关。我们完全可以预处理出这样一个数组P[j]，表示当匹配到B数组的第j个字母而第j+1个字母不能匹配了时，新的j最大是多少。P[j]应该是所有满足B[1..P[j]]=B[j-P[j]+1..j]的最大值。

再后来，A[7]=B[5]，i和j又各增加1。这时，又出现了A[i+1]<>B[j+1]的情况：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =     a b a b a c b
j =     1 2 3 4 5 6 7
```

由于P[5]=3，因此新的j=3：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =         a b a b a c b
j =         1 2 3 4 5 6 7
```

这时，新的j=3仍然不能满足A[i+1]=B[j+1]，此时我们再次减小j值，将j再次更新为P[3]：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =             a b a b a c b
j =             1 2 3 4 5 6 7
```

现在，i还是7，j已经变成1了。而此时A[8]居然仍然不等于B[j+1]。这样，j必须减小到P[1]，即0：

```
i = 1 2 3 4 5 6 7 8 9 ……
A = a b a b a b a a b a b …
B =               a b a b a c b
j =             0 1 2 3 4 5 6 7
```

终于，A[8]=B[1]，i变为8，j为1。事实上，有可能j到了0仍然不能满足A[i+1]=B[j+1]（比如A[8]="d"时）。因此，准确的说法是，当j=0了时，我们增加i值但忽略j直到出现A[i]=B[1]为止。

```
for (int i = 1, j = 0; i <= M; i++)
{
    while (j > 0 && B[j + 1] != A[i]) j = P[j];
    if (B[j + 1] == A[i]) j++;
    if (j == N)
    {
        Len++;//成功找到一个匹配
        j = P[j];
    }
}
```

预处理不需要按照P的定义写成O(m^2)甚至O(m^3)的。我们可以通过P[1],P[2],...,P[j-1]的值来获得P[j]的值。对于刚才的B="ababacb"，假如我们已经求出了P[1],P[2],P[3]和P[4]，看看我们应该怎么求出P[5]和P[6]。P[4]=2，那么P [5]显然等于P[4]+1，因为由P[4]可以知道，B[1,2]已经和B[3,4]相等了，现在又有B[3]=B[5]，所以P[5]可以由P[4] 后面加一个字符得到。P[6]也等于P[5]+1吗？显然不是，因为B[P[5]+1]<>B[6]。那么，我们要考虑“退一步”了。我们考虑P[6]是否有可能由P[5]的情况所包含的子串得到，即是否P[6]=P[P[5]]+1。这里想不通的话可以仔细看一下：

```
    1 2 3 4 5 6 7
B = a b a b a c b
P = 0 0 1 2 3 ?
```

P[5]=3是因为B[1..3]和B[3..5]都是"aba"；而P[3]=1则告诉我们，B[1]、B[3]和B[5]都是"a"。既然P[6]不能由P[5]得到，或许可以由P[3]得到（如果B[2]恰好和B[6]相等的话，P[6]就等于P[3]+1了）。显然，P[6]也不能通过P[3]得到，因为B[2]<>B[6]。事实上，这样一直推到P[1]也不行，最后，我们得到，P[6]=0。

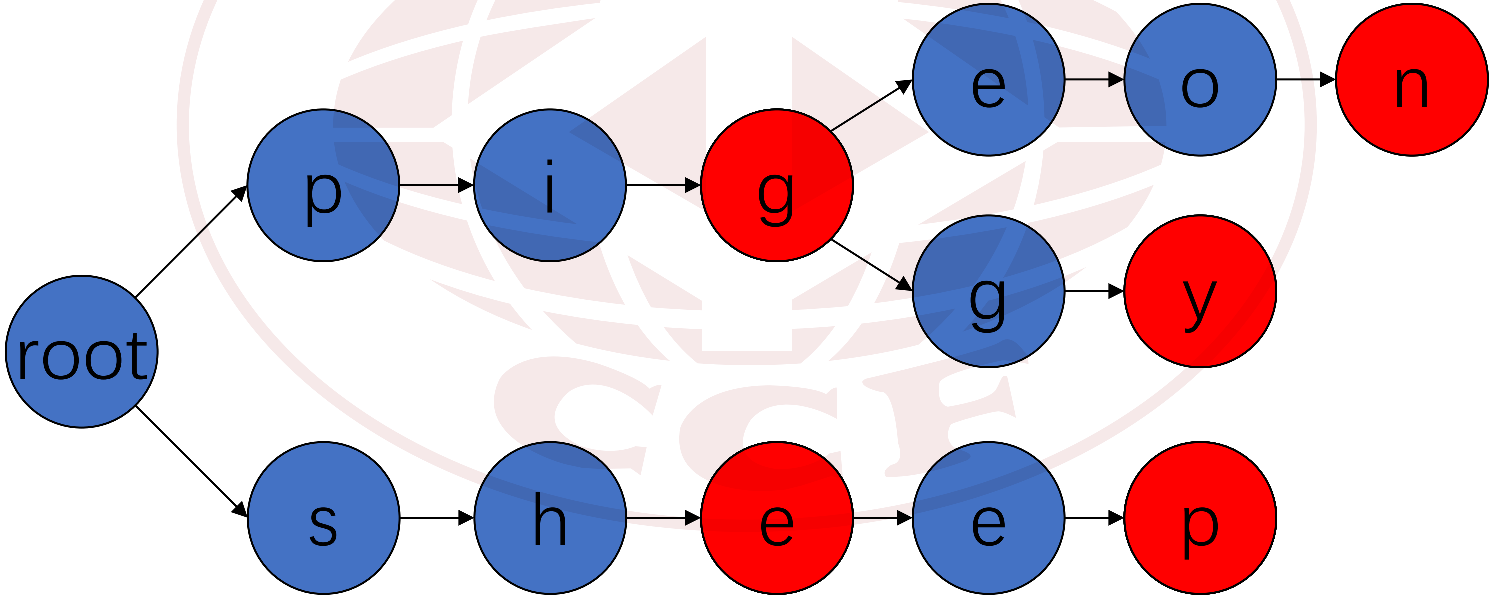
```
P[1] = 0;
for (int i = 2, j = 0; i <= N; i++)
{
    while (j > 0 && B[j + 1] != B[i]) j = P[j];
    if (B[j + 1] == B[i]) j++;
    P[i] = j;
}
```

四、AC自动机

- 给你一本字典，有N个单词(N <= 10000)
- 单词长度 <= 50，都是小写字母
- 给你一个长度不超过1000000的字符串，问你有多少个单词出现在这个字符串之中
- 多次出现只算一次

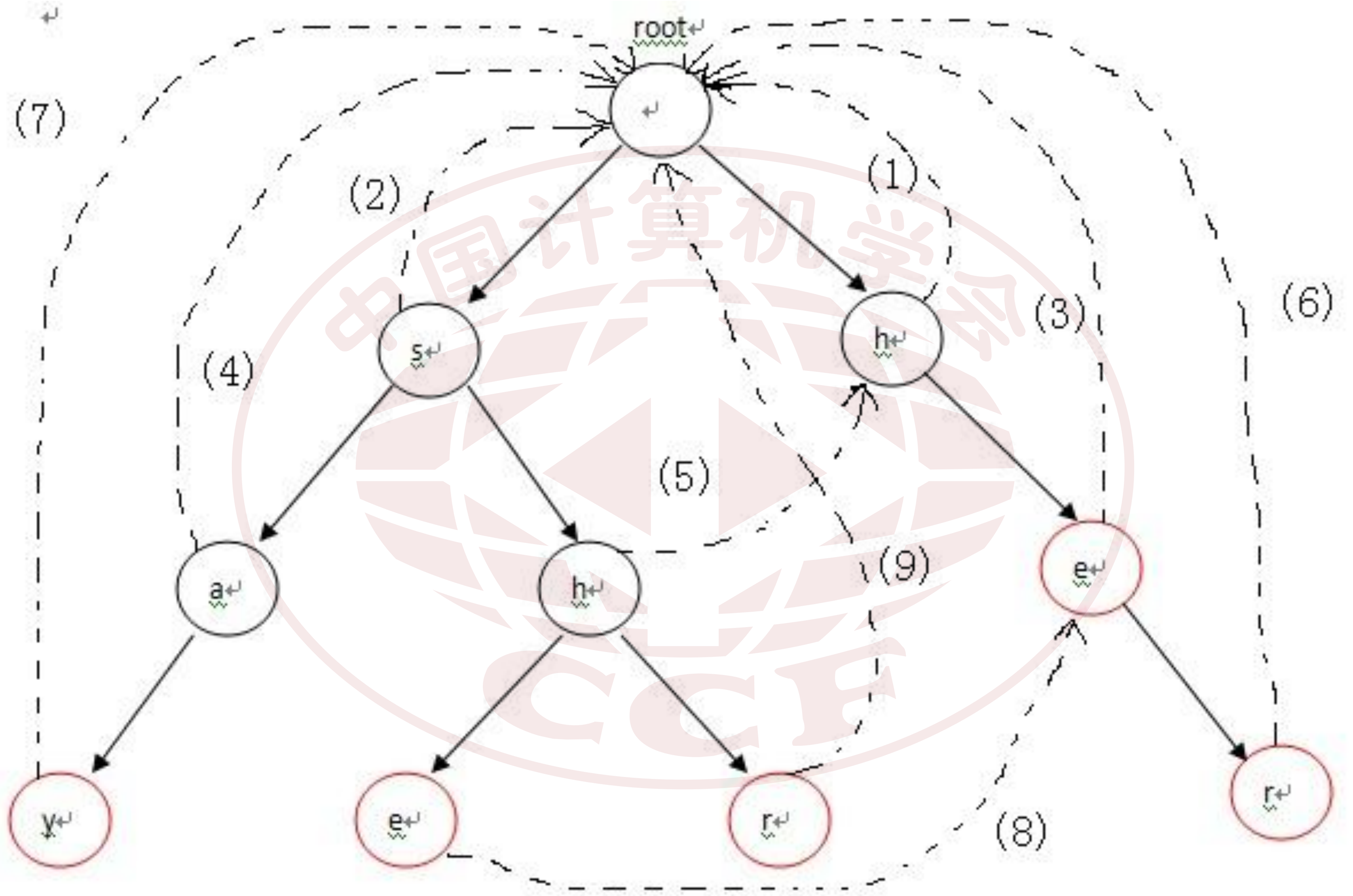
Trie树

- Trie树是一种字符串的基本储存结构
- 下图所示，存储了5个单词



AC自动机

- 以Trie为基础，增加一个Fail指针
  - **该节点代表的字符串的最长后缀**
  - Fail指针所代表的单词
  - 两者匹配吻合
- KMP的算法即为单串匹配自动机


root
(7)
(2)
(1)
(3)
(6)
(4)
s
h
(5)
a
h
e
(9)
y
e
r
r
(8)

构建AC自动机

- Trie树明显是一个拓扑图，按层遍历BFS
  - 一个节点的Fail指针指向节点必然深度比较小
- 为了方便，对于根的几个约定
  - 对于根节点，如果它的儿子是空，那么把它的这个儿子指向自己
  - 根节点儿子的失败指针指向根
- **对于*i*，*i*的儿子的失败指针通过*i*的失败指针一直寻找，直到某个位置有所对应的儿子，或者到根为止**

解法

- 首先对所有单词建立AC自动机
- 用长串*S*来遍历该自动机
  - 每次找到对应的儿子向下走即可
  - 如果对应的儿子不存在，沿Fail指针一路向根，直到对应的儿子不为空为止
- 对于每个节点，沿着Fail指针一路向根，沿途经过的所有节点所表示的串都可以被访问到
  - 访问到一个点就标记之

五、扩展KMP

- 给定母串S，和子串T
- 定义n=|S|, m=|T|，extend[i]=S[i..n]与T的最长公共前缀长度。请在线性的时间复杂度内，求出所有的extend[1..n]

转自刘雅琼PPT

- 容易发现，如果有某个位置i满足extend[i]=m，那么T就肯定在S中出现过，并且进一步知道出现首位置是i——而这正是经典的KMP问题。
- 因此可见“扩展的KMP问题”是对经典KMP问题的一个扩充和加难。

- 例子

S=' aaaaaaaaaabaaa
'

'

T='

extend[1]=10

aaaaaaaaaabaaa

aaaaaaaaaaa

红色箭头表示失配
在第 11 个位置失配

这里为了计算 extend[1]，我们进行了 11 次比较运算。

然后我们要算 extend[2]：

aaaaaaaaaabaaa

aaaaaaaaaaa

红色箭头表示失配
在第 11 个位置失配

- extend[2]=9。为了计算extend[2]，我们是不是也要进行10次比较运算呢？不然。
- 因为通过计算extend[1]=10，我们可以得到这样的信息：

S[1..10]=T[1..10]

S[2..10]=T[2..10]。

- 计算extend[2]的时候，实际上是S[2]开始匹配T。因为S[2..10]=T[2..10]，所以在匹配的开头阶段是"以T[2..10]为母串，T为子串"的匹配。

- 设辅助函数next[i]表示T[i..m]与T的最长公共前缀长度。
- 对上述例子，next[2]=10。也就是说：

  T[2..11]=T[1..10]

  T[2..10]=T[1..9]

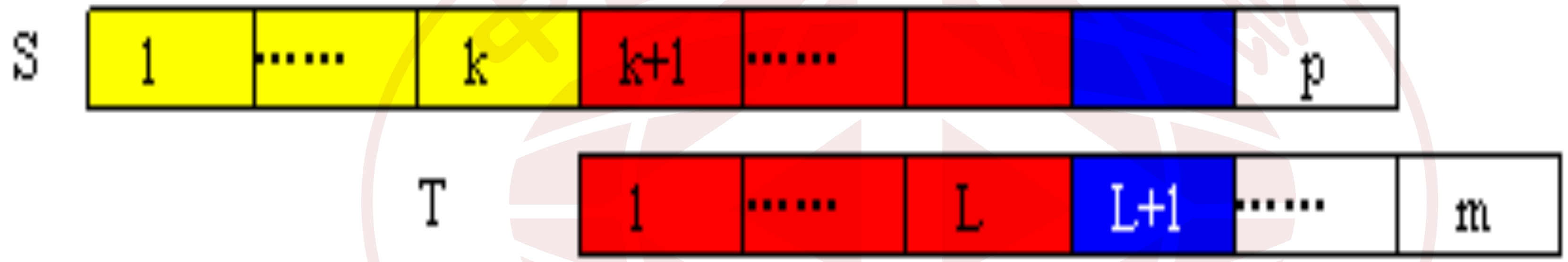
  S[2..10]=T[1..9]。
- 这就是说前9位的比较是完全可以避免的！我们直接从S[11]→T[10]开始比较。这时候一比较就发现失配，因此extend[2]=9。

- 下面提出一般的算法。
- 设extend[1..k]已经算好，并且在以前的匹配过程中到达的最远位置是p。最远位置严格的说就是i+extend[i]-1的最大值，其中i=1,2,3,…,k；不妨设这个取最大值的i是a。(下图黄色表示已经求出来了extend的位置)

S	1	……	k	k+1			p		

根据定义 S[a..p]=T[1..p-a+1] ➔S[k+1..p]=T[k-a+2..p-a+1]，令 L=next[k-a+2]。有两种情况。

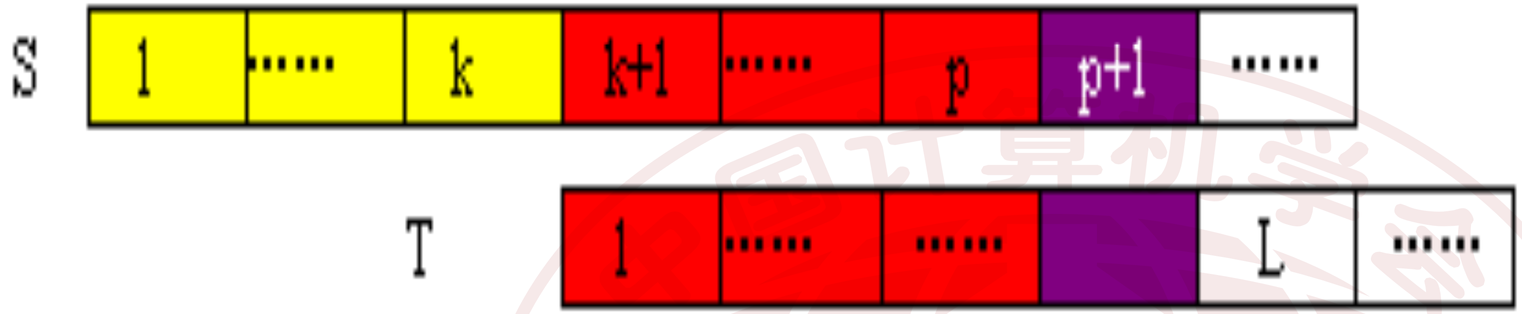
第一种情况 k+L<p，如下图：



上面的红色部分是相等的。蓝色部分肯定不相等，否则就违反了“next[i]表示 T[i..m]与 T 的**最长**公共前缀长度”的定义。（因为 next[k-a+2]=L，如果蓝色部分相等的话，那么就有 next[k-a+2]=L+1 或者更大，矛盾）。

这时候我们无需任何比较就可以知道 extend[k+1]=L。同时 a, p 的值都保持不变，k←k+1，继续上述过程。

第二种情况 k+L>=p。如下图：



上图的紫色部分是未知的。因为在计算 extend[1..k]的时候，到达过的最远地方是 p，所以 p 以后的位置从未被探访过，我们也就无从紫色部分是否相等。

这种情况下，就要从 S[p+1]⇔T[p-k+1]开始匹配，直到失配为止。匹配完之后，比较 extend[a]+a 和 extend[k+1]+(k+1)的大小，如果后者大，就更新 a。

- 整个算法描述结束。
- 上述算法是**线性**算法。原因如下：

容易看出，在计算的过程中，凡是访问过的点，都不需要重新访问了。一旦比较，都是比较以前从不曾探访过的点开始。因此总的时间复杂度是O(n+m),是线性的。

中国计算机学会 China Computer Federation

- 我们发现计算**next实际上以T为母串、T为子串的一个特殊“扩展的KMP”**。用上文介绍的完全相同的算法计算next即可。（用next本身计算next，具体可以参考标准KMP）此不赘述。

六、后缀数组

- 问题：将一个字符串的所有后缀排序
- 定义：Suffix(S,i)=S[i..n](n为字符串长度)

Suffix Array

- 后缀数组SA：
  - 保存1..n的某一个排列,SA[1]...SA[n].并且保证Suffix(SA[i])<Suffix(SA[i+1])
  - 也就是把S的n个后缀从小到大排序后，每个后缀的首字母的序号。
- 名次数组Rank：
  - Rank = SA^{-1} 也就是说,如果SA[i]=j,则Rank[j]=i. Rank[i]就是Suffix(i)在所有后缀从小到大的排列中的名次(位置)

Suffix Array的构造

- 最简单的做法:
  - 直接将n个后缀进行排序。
  - 即使使用最好的排序方法，也要O(n^2)

- 低效的原因 —— **把后缀仅仅当作普通的、独立的字符串，忽略了后缀之间存在的有机联系。**

Suffix Array的构造-倍增算法

对字符串u，定义

$$u^k = \begin{cases} u[1..k] & ,len(u) \geq k \\ u & ,len(u) < k \end{cases}$$

定义k-前缀比较关系$<_k$，$=_k$和$\leq_k$

对两个字符串u,v,

$u <_k v$　当且仅当　$u^k < v^k$

$u =_k v$　当且仅当　$u^k = v^k$

$u \leq_k v$　当且仅当　$u^k \leq v^k$

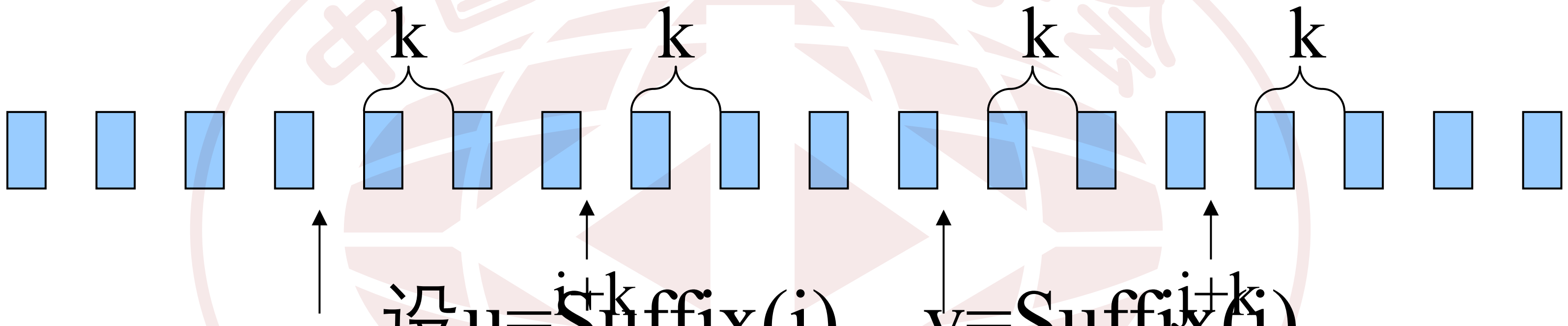
Suffix Array的构造-倍增算法

- 性质1 ： 如果k>=n , $Suffix(i) <_k Suffix(j) \Leftrightarrow Suffix(i) < Suffix(j)$

- 性质2 ： $Suffix(i) =_{2k} Suffix(j) \Leftrightarrow Suffix(i) =_k Suffix(j)$ 且 $Suffix(i+k) =_k Suffix(j+k)$

- 性质3 ：$Suffix(i) <_{2k} Suffix(j) \Leftrightarrow Suffix(i) <_k Suffix(j)$或 $(Suffix(i) =_k Suffix(j)$ 且 $Suffix(i+k) <_k Suffix(j+k))$

Suffix Array的构造-倍增算法



设u=Suffix(i)，v=Suffix(j)

比较红色字符相当于在k-前缀意义下比较

在2k-前缀意义下比较两个后缀可以转化成

对u、v在2k-前缀意义下进行比较

Suffix(i) 和 Suffix(j)

在k-前缀意义下比较两个后缀

比较绿色字符相当于在k-前缀意义下比较

Suffix(i+k) 和 Suffix(j+k)

Suffix Array的构造-倍增算法

把n个后缀按照k-前缀意义下的大小关系从小到大排序

将排序后的后缀的开头位置顺次放入数组SA_k中，称为

k-后缀数组

用$Rank_k[i]$保存Suffix(i)在排序中的名次，称数组$Rank_k$为

k-名次数组

Suffix Array的构造-倍增算法

利用SA_k可以在O(n)时间内求出$Rank_k$

利用$Rank_k$可以在常数时间内对两个后缀进行k-前缀意义下的大小比较

Suffix Array的构造-倍增算法

如果已经求出$Rank_k$

可以在常数时间内对两个后缀进行k-前缀意义下的比较

可以在常数时间内对两个后缀进行2k-前缀意义下的比较

可以很方便地对所有的后缀在2k-前缀意义下排序

- 采用快速排序O(nlogn)
- 采用基数排序O(n)

可以在O(n)时间内由$Rank_k$求出SA_{2k}

也就可以在O(n)时间内求出$Rank_{2k}$

Suffix Array的构造-倍增算法

1-前缀比较关系实际上是对字符串的第一个字符进行比较

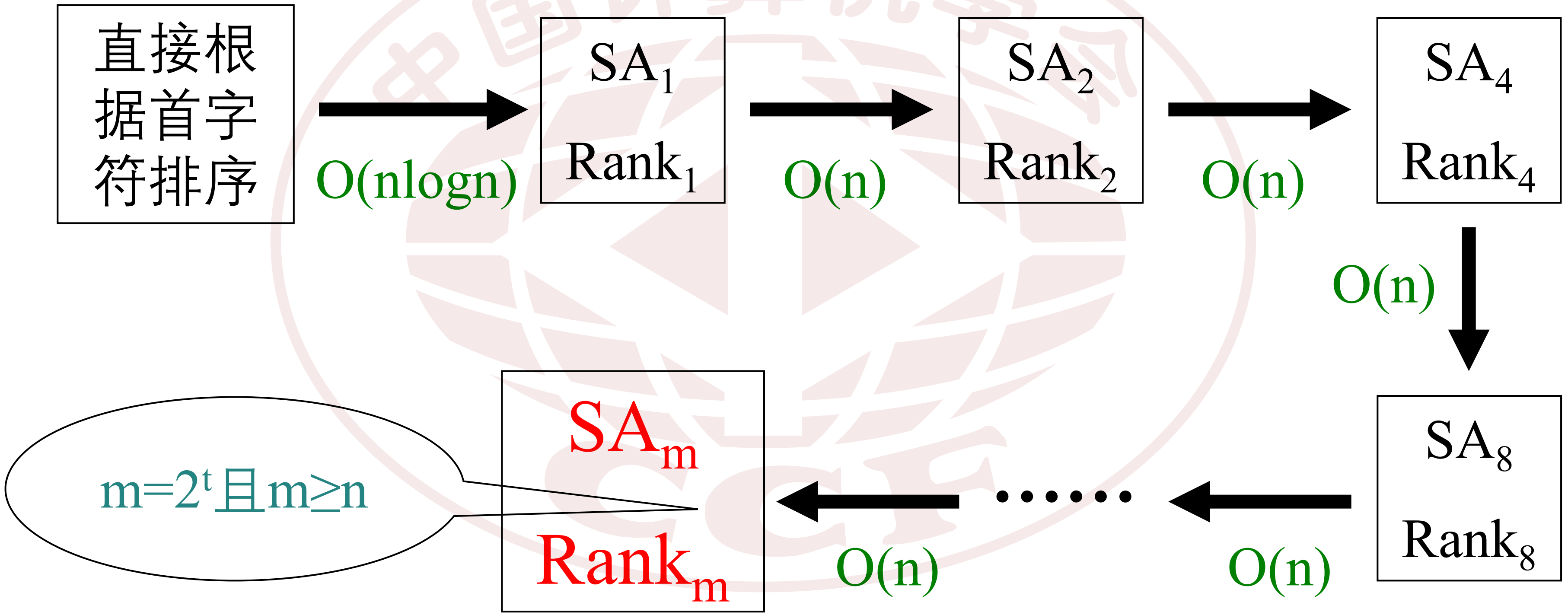
可以直接根据开头字符对所有后缀进行排序求出SA_1

- 采用快速排序，复杂度为O(nlogn)

然后根据SA_1在O(n)时间内求出$Rank_1$

可以在O(nlogn)时间内求出SA_1和$Rank_1$

Suffix Array的构造-倍增算法



O(nlogn)的操作一次

O(n)的操作⌈logn⌉次

总复杂度为

O(nlogn)

Suffix Array的构造-倍增算法

当$m \geq n$时，对任意u=Suffix(x)，$u^m=u$

Suffix(i)$\leq_m$Suffix(j) $\Longleftrightarrow$ Suffix(i)$\leq$Suffix(j) $\Longleftrightarrow$ Suffix(i)$<$Suffix(j)

O(nlogn)求出

SA_m和$Rank_m$

$$\mathbf{SA_m=SA}$$

$$\mathbf{Rank_m=Rank}$$

可以在O(nlogn)时间内求出后缀数组SA和名次数组Rank

Suffix Array的构造-倍增算法

$m \geq n,$

$SA_m = SA$

$Rank_m = Rank$

我们已经在O(nlogn)的时间内构造出了

后缀数组SA 和 **名次数组Rank**

Suffix Array 构造方法总结

利用到后缀之间的联系

用k-前缀比较关系来表达2k-前缀比较关系

每次可以将参与比较的前缀长度加倍 } **倍增思想**

根据SA_k、$Rank_k$求出SA_{2k}、$Rank_{2k}$

参与比较的前缀长度达到n以上时结束

Suffix Array

- 有了后缀数组，我们已经可以解决一些问题。例如上文提到的多串匹配问题，我们可以在O(nLgn + k * m*Lgn)的时间内解决。

后缀数组的最佳搭档——LCP

Suffix Array - LCP

- 定义
- LCP(i , j) = lcp(Suffix(SA[i]) , Suffix(SA[j]))
- 这里我们只考虑i < j 的情况,因为 LCP(i,j)=LCP(j,i)
- 而LCP(i , i) = Len(Suffix(SA[i]))

- LCP Theorem

  LCP(i , j) = min{LCP(k-1,k)|i<k<=j}

Suffix Array - LCP

- 因为LCP(i , j) = min{LCP(k-1,k)|i<k<=j}
- 有一个推论:
- 对于i <= j < k , LCP(j,k) >= LCP(i , k)

- 定义Height的数组
- Height[i] = LCP(i , i + 1) (i=1..n-1)
- Height[n] = 0

Suffix Array - LCP

- LCP(i , j) = min{ Height[i..j-1] }

- LCP(i,j)的计算转化为了查询一个一维数组Height某一个区间内的最小值。这就是非常经典RMQ问题了。

- RMQ : Sparse Table , Winner Tree…

Suffix Array – Height的计算

- 经验：不应该把n个后缀当成互不相关的普通字符串，要尽量利用它们之间的关系!!

- 规律:
- Height[Rank[i+1]]
- >=
- Height[Rank[i]] - 1

Suffix Array – Height的计算

- 几个事实:
- 如果i<n , j<n , 且有lcp(Suffix(i),Suffix(j))>1
- Fact.1
- Suffix(i)<Suffix(j) ⬄ Suffix(i+1)<Suffix(j+1)

- Fact.2
- lcp(Suffix(i),Suffix(j))= lcp(Suffix(i+1),Suffix(j+1)) + 1

Suffix Array – Height的计算

- Height[Rank[i+1]] >=Height[Rank[i]] -1
- 可以这么想象: Suffix(i)和比Suffix(i)略大的那个后缀Suffix(j)的lcp 为 L
- 那么Suffix(i + 1) …的lcp 至少为L – 1
- 把他们的第一个字母去掉,就有L-1了,如果中间又来了其他的后缀,根据LCP Theorem的推论,只会令他们的lcp更大!
- 具体证明: 略

Suffix Array – Height的计算

- 依次求Height[Rank[1]],Height[Rank[2]]......
- 时间复杂度为O(n)
- 证明：略

Suffix Array

- 我们在O(nlgn)的时间内求出了后缀数组与名次数组
- 在O(n)时间内求出了Height数组
- 在O(nlgn)(或者O(n))时间的预处理之后，支持O(1)时间查询两个后缀的LCP。

例1: Dominating Patterns
(Hefei 2009)

- 给你N个单词($N <= 150$)
- 单词长度 <= 70，都是小写字母
- 给你一个长度不超过1000000的字符串
- 要求
  - 求出Ans = 出现次数最多的单词出现了多少次
  - 输出所有出现次数为Ans的单词

解法

- 解法与上题大致相同
  - 首先建自动机，然后遍历该自动机
  - 每访问到一个节点，把这个节点的Count + 1
- 每个单词访问到的次数就是以**这个单词为后缀的所有节点**的Count的总和
- 也就是所有能够**通过Fail指针到达这个点**的节点的Count的总和

两种处理

- 方法1：将所有节点的Count一路沿着Fail指针向上传，用暴力法
- 方法2：根据Fail指针反向建立拓扑图，DFS
- 方法3：将所有节点的Count一路沿着Fail指针向上传，BFS

例2: Searching the String
(ZOJ Monthly, July 2009)

- 给出一个长度 <= 100000的字符串
- 给出*Q*(*Q* <= 100000)个询问，询问分两种
  - 0 *Str* 求*Str*在原字符串中出现次数，重叠算多次
  - 1 *Str* 求*Str*在原字符串中出现次数，重叠算一次
  - ***Str*长度 <= 6 （这个条件居然是有用的）**
- 可以离线处理

- http://acm.zju.edu.cn/onlinejudge/showProblem.do?problemCode=3228

解法

- 离线算法，基本思想和上一题相同
- 对于每个点记录两种询问，用上一题的方法1进行
  - 因为长度最大只有6，所以这个长度的常数可以忽略
- 注意两种询问的差别就可以了

例3: Censored!
(NEERC 2001, NS)

- 某种语言字符集大小为*M*（不一定是小写字母），给出*P*个字符串
- 求出长度为*N*，不包含给定的*P*个字符串中任意一个的该种语言字符串的数量
- *N*, *M*, *P* <= 50
- **需要使用高精度**

- http://acm.pku.edu.cn/JudgeOnline/problem?id=1625

解法

- 与自动机有关的动态规划
- 建立自动机的时候注意如果一个节点为非法，那么所有可以通过Fail指针指向它的都是非法节点
- 具体实现只要看当前节点的Fail指针是非法节点，因为前面已经全部被算过了
  - $Invaild[k] = Invaild[Fail[k]]$ or $Invaild[k]$
- $F[i][j]$表示长度为i的，现在停在自动机节点j的串的个数

状态转移伪代码

```
F[0][RootID] = 1
for i = 0 to N - 1
    for j = 1 to 自动机状态数
        for each k in 字符集
        {
            P = j;
            while (P的儿子中没有k)
                P = Fail[P]
            P = Next[P][k]
            if (P是合法节点)
                F[i + 1][P] += F[i][j]
        }
```

例4: DNA Repair
(Hefei 2008 Preliminary Contest)

- 字符集仅含{A, C, G, T}
- 给定*N*个长度不超过20的病毒串
  - $N <= 50$
- 给出一个长度不超过1000的DNA串，要求改变最少个字符，使得不出现任何一个病毒串
  - 无解输出-1

解法

- $F[i][j]$表示已经处理前i个，现在停在自动机节点j，最少的变化量
- 转移就是沿着某个儿子走一步
- 与上题基本相同
- 枚举当前节点的儿子v，然后如果没有尾标记，则有dp[i+1][v] = min(dp[i+1][v], dp[i][j]+T)，其中当v代表的字符和s[i]相同时T=0，否则T=1。