KMP算法

我们这里说的KMP不是拿来放电影的（虽然我很喜欢这个软件），而是一种算法。KMP算法是拿来处理字符串匹配的。换句话说，给你两个字符串，你需要回答，B串是否是A串的子串（A串是否包含B串）。比如，字符串A="I'm matrix67"，字符串B="matrix"，我们就说B是A的子串。你可以委婉地问你的MM："假如你要向你喜欢的人表白的话，我的名字是你的告白语中的子串吗？"

解决这类问题，通常我们的方法是枚举从A串的什么位置起开始与B匹配，然后验证是否匹配。假如A串长度为n，B串长度为m，那么这种方法的复杂度是O (mn)的。虽然很多时候复杂度达不到mn（验证时只看头一两个字母就发现不匹配了），但我们有许多"最坏情况"，比如，A= "aaaaaaaaaaaaaaaaaaaaaaaaaab"，B="aaaaaaaab"。我们将介绍的是一种最坏情况下O(n)的算法（这里假设 m<=n），即传说中的KMP算法。

之所以叫做KMP，是因为这个算法是由Knuth、Morris、Pratt三个提出来的，取了这三个人的名字的头一个字母。这时，或许你突然明白了AVL 树为什么叫AVL，或者Bellman-Ford为什么中间是一杠不是一个点。有时一个东西有七八个人研究过，那怎么命名呢？通常这个东西干脆就不用人名字命名了，免得发生争议，比如"3x+1问题"。扯远了。

个人认为KMP是最没有必要讲的东西，因为这个东西网上能找到很多资料。但网上的讲法基本上都涉及到"移动(shift)"、"Next函数"等概念，这非常容易产生误解（至少一年半前我看这些资料学习KMP时就没搞清楚）。在这里，我换一种方法来解释KMP算法。

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 =               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。

这个过程的代码很短（真的很短），我们在这里给出：

程序代码

```
j:=0;
for i:=1 to n do
   begin
        while (j>0) and (B[j+1]<>A[i]) do
           j:=P[j];
        if B[j+1]=A[i] then j:=j+1;
        if j=m then
           begin
                writeln('Pattern occurs with shift ',i-m);
                j:=P[j];
           end;
end;
```

最后的j:=P[j]是为了让程序继续做下去，因为我们有可能找到多处匹配。

这个程序或许比想像中的要简单，因为对于i值的不断增加，代码用的是for循环。因此，这个代码可以这样形象地理解：扫描字符串A，并更新可以匹配到B的什么位置。

现在，我们还遗留了两个重要的问题：一，为什么这个程序是线性的；二，如何快速预处理P数组。

为什么这个程序是O(n)的？其实，主要的争议在于，while循环使得执行次数出现了不确定因素。我们将用到时间复杂度的摊还分析中的主要策略，简单地说就是通过观察某一个变量或函数值的变化来对零散的、杂乱的、不规则的执行次数进行累计。KMP的时间复杂度分析可谓摊还分析的典型。我们从上述程序的j 值入手。每一次执行while循环都会使j减小（但不能减成负的），而另外的改变j值的地方只有第五行。每次执行了这一行，j都只能加1；因此，整个过程中j最多加了n个1。于是，j最多只有n次减小的机会（j值减小的次数当然不能超过n，因为j永远是非负整数）。这告诉我们，while循环总共最多执行了n次。按照摊还分析的说法，平摊到每次for循环中后，一次for循环的复杂度为O(1)。整个过程显然是O(n)的。这样的分析对于后面P数组预处理的过程同样有效，同样可以得到预处理过程的复杂度为O(m)。

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]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。

怎么这个预处理过程跟前面的KMP主程序这么像呢？其实，KMP的预处理本身就是一个B串“自我匹配”的过程。它的代码和上面的代码神似：

程序代码

```
P[1]:=0;
j:=0;
for i:=2 to m do
  begin
     while (j>0) and (B[j+1]<>B[i]) do
        j:=P[j];
     if B[j+1]=B[i] then j:=j+1;
     P[i]:=j;
  end;
```

最后补充一点：由于KMP算法只预处理B串，因此这种算法很适合这样的问题：给定一个B串和一群不同的A串，问B是哪些A串的子串。

串匹配是一个很有研究价值的问题。事实上，我们还有后缀树，自动机等很多方法，这些算法都巧妙地运用了预处理，从而可以在线性的时间里解决字符串的匹配。我们以后来说。