KMP 算法详解---- by matrix67

如果机房马上要关门了，或者你急着要和 MM 约会，请直接跳到第六个自然段。

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 =   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。

这个过程的代码很短（真的很短），我们在这里给出：

```
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]、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。

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：

```
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。

昨天发现一个特别晕的事，知道怎么去掉 BitComet 的广告吗？把界面语言设成英文就行了。

还有，金山词霸和 Dr.eye 都可以去自杀了，Babylon 素王道。