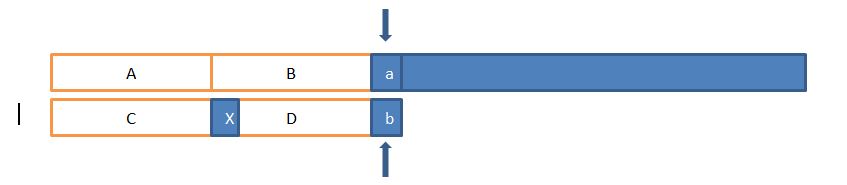
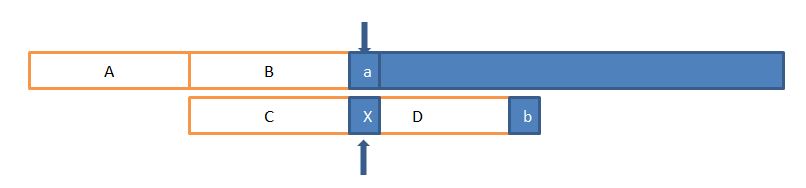
这里备注一个KMP算法的疑惑，采用证明的方式来说明KMP算法的正确性。

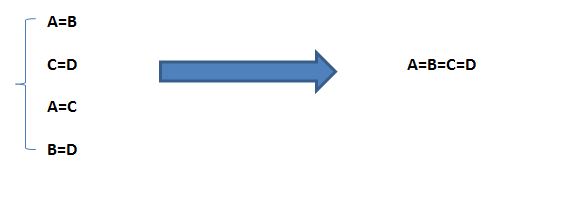
1. 都知道，当模式匹配过程中失配时，利用next数组尽力使模式串往后滑动，省去一些不必要的比较。但是为什么就可以忽略中间的那么多次比较，而不是省去的那么多次滑动中的**某个比较状态**正好是匹配的呢？为了解释这个以下用图形说明。

假设有如下的比较状态：

1. 如上图所示：上面的那个串为匹配主串，下面的为模式串，当前匹配状态为失配状态，其中主串为a，而模式串当前为b，使用next数组发现在模式串中，当前状态b失配时，应该使得模式串向后滑动尽量多的长度。

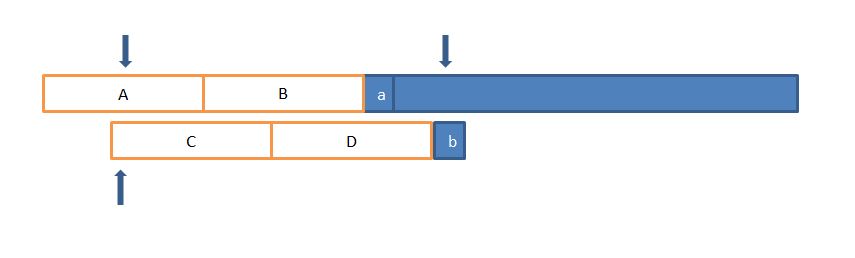
得到下面的状态：



1. 由模式匹配的next数组含义得，如果变动成上面的情况，则主串和模式串中应有：****

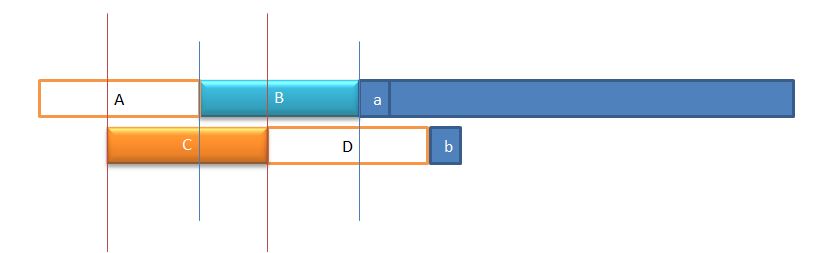
**这里有一点需要注意的是：这里的C、D是最长的，没有比这个更长的且满足要求的了**。

1. 假设存在一个中间状态满足要求。

即如下：在上面那样的匹配过程中有模式串和主串一起从头开始匹配，然后一直匹配到最后，完成整个模式串的查找并找到了位置。

1. **真的是这样吗？**

让我们来具体分析下上面的那种情况，假如真的如上面想象的那样，则应有：

**

F

BB

AA

AA区域和BB区域是相等的，可得图中箭头对应的部分是是相等的（F），

即有B区域的前半部分，和C的后半部分是相等的，又因为B=C，所以得:

**C+F=F+D**

其中前面的F区域是在D区域中的前半个，而后面的那个F是C区域中的后半部分。

这样就有一个新的满足next性质的串，可得上面最初提到的那个等式是不成立的。所以原式成立。

1. 关于next

假设 k=next[i],则表明有P[0..k-1]==P[i-k…i-1]，此时有:

如果:P[k]==P[i],则next[i+1]=k+1;

如果P[k]!=P[i],则如果将i指示位置作为主串位置，而k指示的作为模式串位置，则可以得到一个新的模式匹配问题此时面临的问题是：P[k]与P[i]失配，则很自然的使用k=next[k],来尝试下一个比对，这是模式匹配的算法思想。一直比对下去，迭代下去，直到某个k’使得P[i]=P[k’]成立 或者回到k=-1，还没有出现，这是置为0。

1. 关于改进的next

算法基本相同，区别在于，在由k=next[i]，不仅考虑i位和k位对next[i+1]的影响，还 考虑当k或者k’定下来后，这里的P[k+1]或者是P[k’+1]与P[i]之间的关系，因为当二者相等时，如果P[i+1]失配时，是没有必要再一次次将比较下去的，而是直接那next[k+1]或者next[k’+1]直接作为next[i+1]的值，只有当二者不相等时，则失配时，才是有必要的。

**关于具体算法请参考代码**