**引言：关于字符串**

**字符串(string)：是由0或多个字符组成的有限序列。一般写作`s = "123456..."`。s这里是主串，其中的一部分就是子串。**

其实，对于字符串大小关系不如是否相同重要。包括密码验证、hash列等。  
而字符串的存储结构有两种：顺序存储结构和链式存储结构。由于不同的字符是连在一起的，所以一般是开足够大的空间进行顺序存储，这样更符合字符串的意义。

**一、BF算法实现**

一种暴力的、朴素的模式匹配算法，是的，时间复杂度为O(M\*N)。而下面的KMP算法则是O(M+N)。不废话，直接上代码。

[复制代码](javascript:void(0);)

1 int BFfind(string base,string target,int start=0){

2 if(base.length()<1 || target.length()<1 || start<0)

3 return -1;

4 int i,j;

5 for(i=start;i<base.length();++i){

6 for(j=0;j<target.length();++j){

7 if(target[j]==base[i+j])

8 continue;

9 else

10 break;

11 }

12 if (j==target.length()){ //完全匹配

13 return i;

14 }

15 }

16 return -1; //没找到

17 }

[复制代码](javascript:void(0);)

**二、KMP算法实现思路**

来自：http://www.ruanyifeng.com/blog/2013/05/Knuth%E2%80%93Morris%E2%80%93Pratt\_algorithm.html

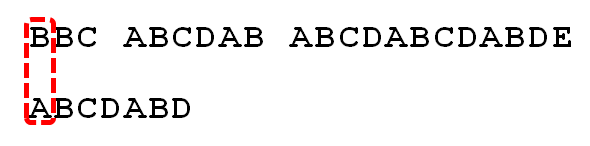
举例来说，有一个字符串"BBC ABCDAB ABCDABCDABDE"，我想知道，里面是否包含另一个字符串"ABCDABD"？



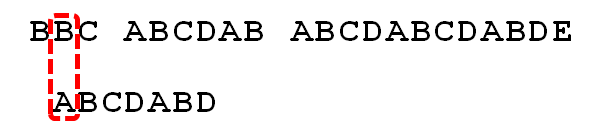
许多算法可以完成这个任务，[Knuth-Morris-Pratt算法](http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm" \t "_blank)（简称KMP）是最常用的之一。它以三个发明者命名，起头的那个K就是著名科学家Donald Knuth。



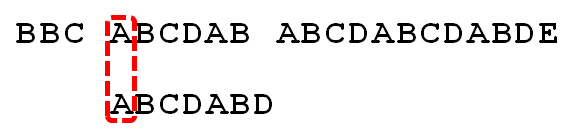
这种算法不太容易理解，网上有很多[解释](http://www.google.com/search?q=Knuth-Morris-Pratt+algorithm" \t "_blank)，但读起来都很费劲。直到读到[Jake Boxer](http://jakeboxer.com/blog/2009/12/13/the-knuth-morris-pratt-algorithm-in-my-own-words/)的文章，我才真正理解这种算法。下面，我用自己的语言，试图写一篇比较好懂的KMP算法解释。

1. 

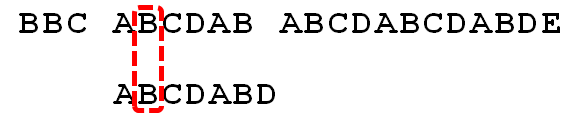
首先，字符串"BBC ABCDAB ABCDABCDABDE"的第一个字符与搜索词"ABCDABD"的第一个字符，进行比较。因为B与A不匹配，所以搜索词后移一位。

2. 

因为B与A不匹配，搜索词再往后移。

3. 

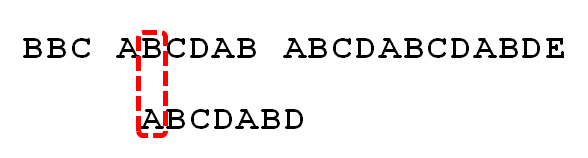
就这样，直到字符串有一个字符，与搜索词的第一个字符相同为止。

4. 

接着比较字符串和搜索词的下一个字符，还是相同。

5. 

直到字符串有一个字符，与搜索词对应的字符不相同为止。

6. 

这时，最自然的反应是，将搜索词整个后移一位，再从头逐个比较。这样做虽然可行，但是效率很差，因为你要把"搜索位置"移到已经比较过的位置，重比一遍。

7. 

一个基本事实是，当空格与D不匹配时，你其实知道前面六个字符是"ABCDAB"。KMP算法的想法是，设法利用这个已知信息，不要把"搜索位置"移回已经比较过的位置，继续把它向后移，这样就提高了效率。

8. 

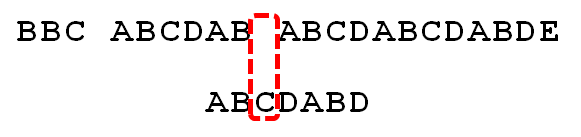
怎么做到这一点呢？可以针对搜索词，算出一张《部分匹配表》（Partial Match Table）。这张表是如何产生的，后面再介绍，这里只要会用就可以了。

9. 

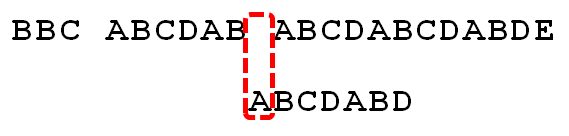
已知空格与D不匹配时，前面六个字符"ABCDAB"是匹配的。查表可知，最后一个匹配字符B对应的"部分匹配值"为2，因此按照下面的公式算出向后移动的位数：

　　移动位数 = 已匹配的字符数 - 对应的部分匹配值

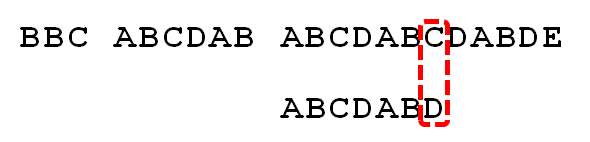
因为 6 - 2 等于4，所以将搜索词向后移动4位。

10. 

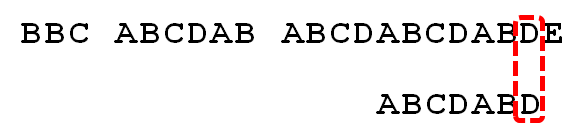
因为空格与Ｃ不匹配，搜索词还要继续往后移。这时，已匹配的字符数为2（"AB"），对应的"部分匹配值"为0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移2位。

11. 

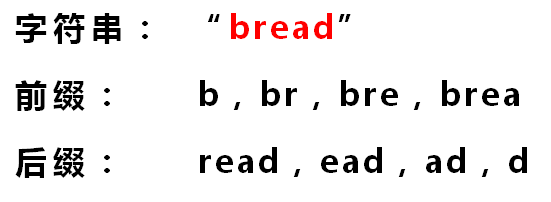
因为空格与A不匹配，继续后移一位。

12. 

逐位比较，直到发现C与D不匹配。于是，移动位数 = 6 - 2，继续将搜索词向后移动4位。

13. 

逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，再将搜索词向后移动7位，这里就不再重复了。

14. 

下面介绍《部分匹配表》是如何产生的。

首先，要了解两个概念："前缀"和"后缀"。 "前缀"指除了最后一个字符以外，一个字符串的全部头部组合；"后缀"指除了第一个字符以外，一个字符串的全部尾部组合。

15. 

"部分匹配值"就是"前缀"和"后缀"的最长的共有元素的长度。以"ABCDABD"为例，

　　－　"A"的前缀和后缀都为空集，共有元素的长度为0；

　　－　"AB"的前缀为[A]，后缀为[B]，共有元素的长度为0；

　　－　"ABC"的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度0；

　　－　"ABCD"的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为0；

　　－　"ABCDA"的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为"A"，长度为1；

　　－　"ABCDAB"的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为"AB"，长度为2；

　　－　"ABCDABD"的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD, D]，共有元素的长度为0。

16.

"部分匹配"的实质是，有时候，字符串头部和尾部会有重复。比如，"ABCDAB"之中有两个"AB"，那么它的"部分匹配值"就是2（"AB"的长度）。搜索词移动的时候，第一个"AB"向后移动4位（字符串长度-部分匹配值），就可以来到第二个"AB"的位置。

**三、KMP算法的代码实现**

首先是Next数组的代码：重要的思路是在不匹配的时候怎么进行回溯（第6、7行）。

[复制代码](javascript:void(0);)

1 void GetNext(string target,int \*next){

2 int index,k;//k:最大后缀长度 ;index:字符串下标

3 int len = target.length();

4 next[0] = 0;

5 for(index=1,k=0;index<len;++index){

6 while(k>0 && target[index]!=target[k])

7 k= next[k-1];//回溯找到长度为k-1的最大后缀

8 if(target[index]==target[k])

9 ++k;

10 next[index] = k;

11 }

12 }

[复制代码](javascript:void(0);)

然后是KMP的主体部分。这里用两重循环实现的，为了让代码易懂，所以多开了几个变量来增强可读性，比如说：用来存放长度变量等

最重要的地方是运用了之前提到的 “移动位数 = 已匹配的字符数 - 对应的部分匹配值“ 这个公式。

[复制代码](javascript:void(0);)

1 int KMPfind(string base,string target){

2 int \*next = new int [target.length()];

3 GetNext(target,next);

4

5 int m = base.length(), n=target.length();//m,n分别为base和target的长度

6 int step;//移动步数

7 int num = 0;//已经匹配的字符数

8 int i,j;

9 for(i=0;i<m;){

10 num = 0;

11 for(j=0;j<n;++j){

12 if(target[j]==base[i+j])

13 ++num;

14 else

15 break;

16 }

17 if (num==n){

18 return i;

19 }

20 if(num)

21 step = num-next[num-1];

22 else

23 step = 1;

24 //cout<<"i is:"<<i<<"step is:"<<step<<endl;

25 i = i+step;

26 //cout<<"i is:"<<i<<"step is:"<<step<<endl;

27 }

28 return -1;

29 }