# 算法

## 链表

### 反转单链表

* 从原链表的头部一个一个取节点并插入到新链表的头部，两个链表，一个新链表，一个原链表

|  |
| --- |
| ListNode\* ReverseList(ListNode\* pHead) {  ListNode\* newh = NULL;  for(ListNode\* p = pHead; p; )//p为工作指针  {  ListNode\* tmp = p -> next;//temp保存下一个结点  p -> next = newh;  newh = p;  p = tmp;  }  return newh;  } |

* 每次都将原第一个结点之后的那个结点放在新的表头后面。一个链表即可

## 字符串

### 旋转字符串

给定一个字符串，要求把字符串前面的若干个字符移动到字符串的尾部，如把字符串“abcdef”前面的2个字符'a'和'b'移动到字符串的尾部，使得原字符串变成字符串“cdefab”。请写一个函数完成此功能，要求对长度为n的字符串操作的时间复杂度为 O(n)，空间复杂度为 O(1)。

* 三步反转法 原串abcdef,k=3 => defabc

abc => cba，def => fed，cbafed => defabc

* 模拟字符移动，每次只将某个字符移动到它的正确位置，并用一个变量idx记录移动之后的位置，和cur记录被覆盖的值【即下一个要移动的值】

|  |  |  |
| --- | --- | --- |
| abcdef,idx=0,cur=’a’ | abcaef,idx=3,cur=’d’ | dbcaef,idx=0,cur=’a’ |

上面的转化就出现了问题，因为整个串长度为6，k=3,6恰好是3的倍数，这样就会到循环位置0 => 3 => 0 => 3，而不会遍历位置1 => 4 => 1 和 2

所以，我们需要另一个变量distance来让遍历的轨道发生变更,思路就是利用distance来记录一次循环将要发生，我们人为增加一次idx，让它到下一个轨道。

|  |  |  |
| --- | --- | --- |
| abcdef,idx=0,cur=’a’ | abcaef,idx=3,cur=’d’ | dbcaef,idx=1,cur=’b’ |

代码实现:

|  |
| --- |
| public class Solution {  public void rotate(int[] nums, int k) {  if(nums == null) return;  int len = nums.length;  /\*\*  \* idx:当前的位置  \* next:要替换的位置  \* cur:要替换的值  \* distance:发现循环发生的时刻，即只在一个轨道上循环  \*\*/  int idx = 0, cur = nums[0], next = 0, distance = 0;  int temp = 0;  //这个for循环只是为了保证替换发生了len次  for(int i=0; i<len; i++) {  next = (idx+k)%len;  temp = nums[next];  nums[next] = cur;  idx = next;  cur = temp;  if((distance=(distance+k)%len) == 0) {  idx = (idx+1)%len;  cur = nums[idx];  }  }  }  } |

类似题目：

|  |
| --- |
| 1、链表翻转。给出一个链表和一个数k，比如，链表为1→2→3→4→5→6，k=2，则翻转后  2→1→6→5→4→3，若k=3，翻转后3→2→1→6→5→4，若k=4，翻转后4→3→2→1→6→5，用程序实现。  2、编写程序，在原字符串中把字符串尾部的m个字符移动到字符串的头部，要求：长度为n的字符串操作时间复杂度为O(n)，空间复杂度为O(1)。  例如，原字符串为”Ilovebaofeng”，m=7，输出结果为：”baofengIlove”。  3、单词翻转。输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变，句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。例如，输入“I am a student.”，则输出“student. a am I”。 |

### 字符串包含

给定两个分别由字母组成的字符串A和字符串B，字符串B的长度比字符串A短。请问，如何最快地判断字符串B中所有字母是否都在字符串A里？

* HashTable
* 对字符串A，用位运算（26bit整数表示)计算出一个“签名”，再用B中的字符到A里面进行查找。

|  |
| --- |
| bool StringContain(string &a,string &b)  {  int hash = 0;  for (int i = 0; i < a.length(); ++i)  {  //精华，a[i]-‘A’得到字符的对应位置，1左移令字符对应位置为1，然后或操作  hash |= (1 << (a[i] - 'A'));  }  for (int i = 0; i < b.length(); ++i)  {  //做与操作，如果某一次与操作结果为0，就代表该字符未出现  if ((hash & (1 << (b[i] - 'A'))) == 0)  {  return false;  }  }  return true;  } |

### 字符串转换成整数

题意很明确，但注意事项很多，如下：

|  |
| --- |
| 1. 字符串为null，或空字符串时，返回整数0 2. 整数是有符号的，’+’ 或 ‘-’ 3. 去掉无效字符，从字符串第一个非空有效字符开始，到第一个无效字符(不包含)截止 4. 可能会出现整数溢出的情况，即大于Integer.MAX\_VALUE或小于Integer.MIN\_VALUE |

注意以上易错点之后，代码实现如下：

|  |
| --- |
| public class Solution {  public int myAtoi(String str) {  int index = 0, sign = 1, total = 0;  //1. Empty string  if(str.length() == 0) return 0;    //2. Remove Spaces  while(str.charAt(index) == ' ' && index < str.length())  index ++;    //3. Handle signs  if(str.charAt(index) == '+' || str.charAt(index) == '-'){  sign = str.charAt(index) == '+' ? 1 : -1;  index ++;  }    //4. Convert number and avoid overflow  while(index < str.length()){  int digit = str.charAt(index) - '0';  if(digit < 0 || digit > 9) break;    //check if total will be overflow after 10 times and add digit  if(Integer.MAX\_VALUE/10 < total || Integer.MAX\_VALUE/10 == total && Integer.MAX\_VALUE %10 < digit)  return sign == 1 ? Integer.MAX\_VALUE : Integer.MIN\_VALUE;    total = 10 \* total + digit;  index ++;  }  return total \* sign;  }  } |

上面的代码实现，思路清晰，效率高。

对比自己之前实现的代码：

|  |
| --- |
| public class Solution {  public int myAtoi(String str) {  if(str == null || str.equals("")) return 0;    String strTemp = str.trim();    int symbol = 1;  if(strTemp.charAt(0) == '-' || strTemp.charAt(0) == '+') {  symbol = strTemp.charAt(0) == '-' ? -1 : 1;  strTemp = strTemp.substring(1);  }    for(int i=0; i<strTemp.length(); i++) {  if(strTemp.charAt(i) < '0' || strTemp.charAt(i) > '9') strTemp = strTemp.substring(0, i);  }    double temp = 0.0;  boolean tag = false;  for(int i=0; i<strTemp.length(); i++) {  if(strTemp.charAt(i) != '0' || tag) {  temp = temp \* 10 + (strTemp.charAt(i) - '0');  tag = true;  }  }    if(temp\*symbol > Integer.MAX\_VALUE)  return Integer.MAX\_VALUE;  else if(temp\*symbol < Integer.MIN\_VALUE)  return Integer.MIN\_VALUE;    return (int)(temp\*symbol);  }  } |

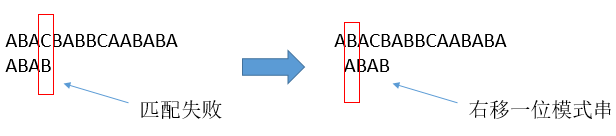
### KMP算法

给一个源字符串s : ABACBABBCAABABA 和一个模式pattern串p : ABABAC，判断源字符串S中是否出现模式串P。

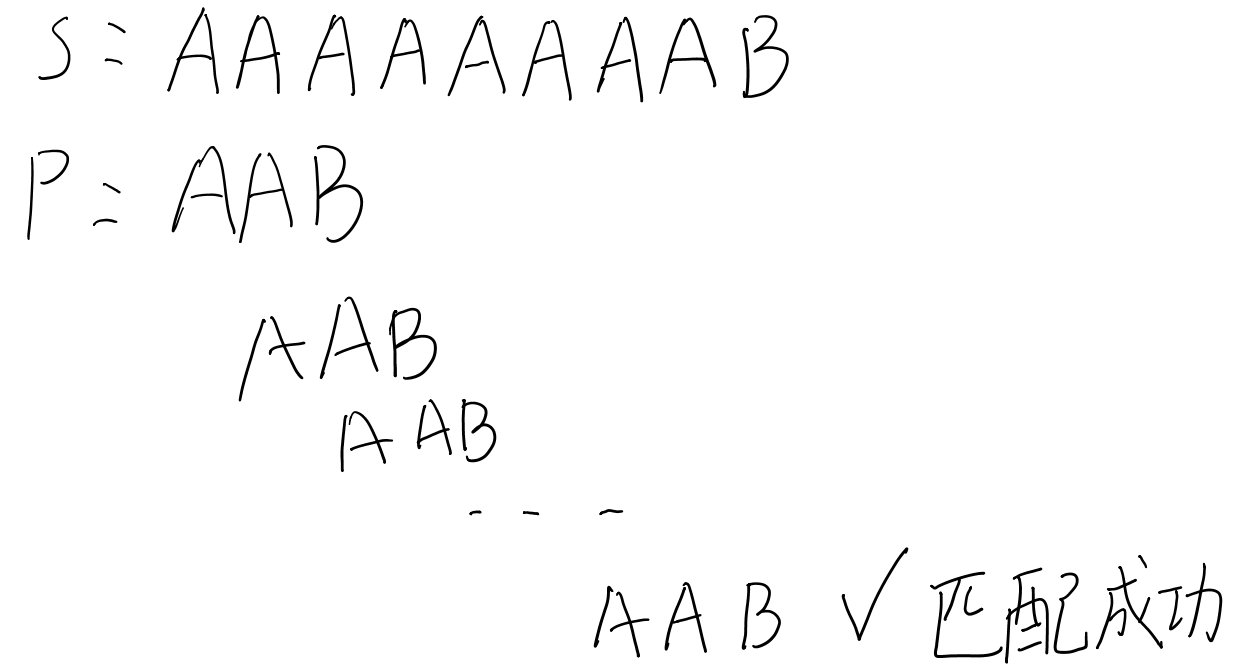
问题很明确，也很容易想出简单的解题方法

* 暴力解法

只需将模式串沿着源字符串移动匹配，如果失败，就将模式串向后一个位置，再继续匹配。

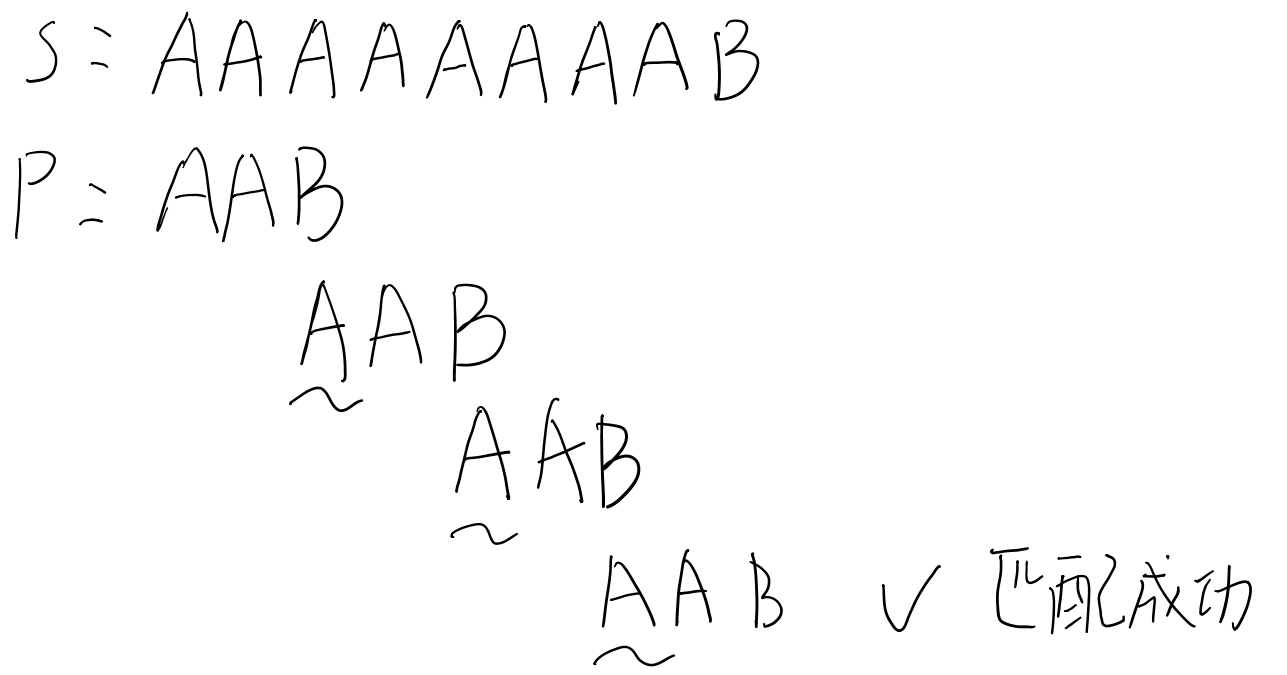


最糟糕的情况如下:



* KMP算法

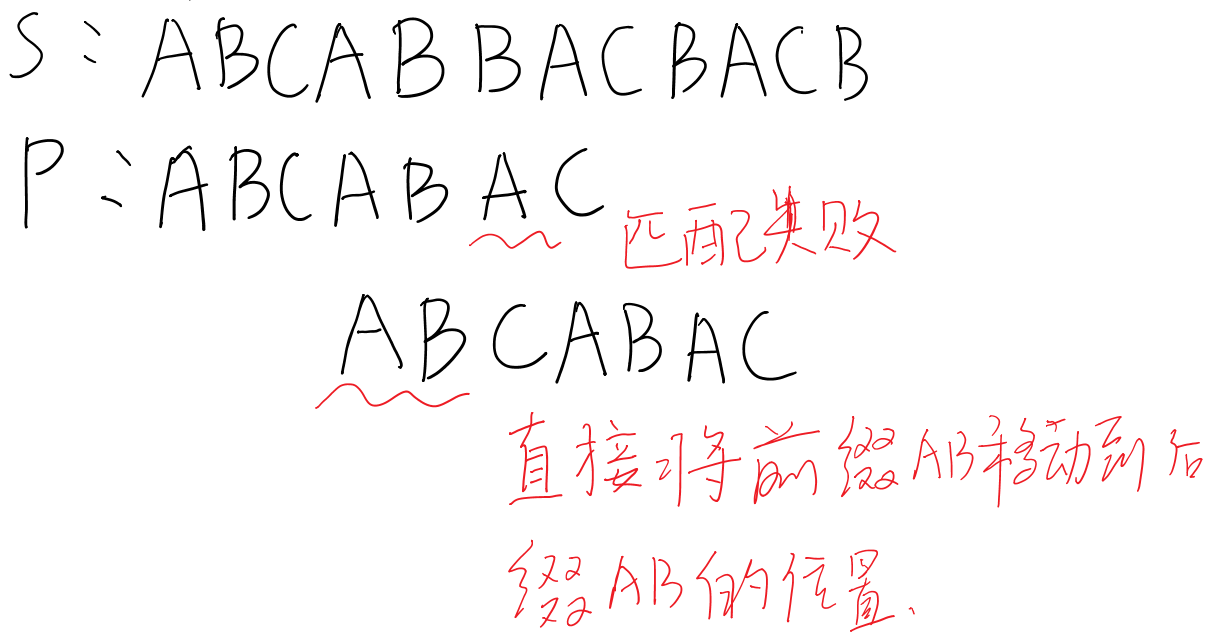
通过观察暴力解法可以观察到，每次匹配失败后，都是将模式串右移一位，重新匹配，这时间复杂度就在O(m\*n)，如上图最糟糕的情况。出现这种情况的原因就是，模式串每次匹配的时候，有些信息没有记录，或者说没有发现模式串的特点导致每次都需要重新开始匹配。其实，我们是可以更快的，例如上面的那种匹配，可以进化到下面:



显然这种匹配方式的时间复杂度是O(m+n)。

KMP算法有两种思路实现这种匹配方式，如下：

1. 利用前缀和后缀完全匹配最长的长度，如s:ABCAB···，其中ABCAB前后缀最长匹配长度为2。不妨设该模式串对应的前后缀长度数组为next[]，源字符串下标为i（从0开始），模式串下标j（从0开始），这样当匹配失败时，使模式串向右移动j-next[i]位。如上图第一次匹配失败时，模式串移动的位数为2-next[2]=2。



**主要思想：**

这种匹配方式的思想主要是当在p[4]的’B’后面的字符匹配失败时，我们就直接将模式串向后移动使前缀AB跑到后缀AB的位置来，为什么呢？因为我们知道源字符串s[3-4]肯定是AB的，因为已经匹配到s[5]失败了，虽然在s[5]失败了，但是这个AB还是要匹配的，所以，我们就直接将前缀AB移动到后缀AB的位置。之所以这里能确定是AB，因为我们知道在p[5]之前的前缀和后缀完全匹配最长的长度就是2。 一般地，我们就要前后缀能匹配最长的长度，这样就可以直接将最长前缀移动到最长后缀那里

**Next数组求解（动态规划）**



已知next[j] = k, 求next[j+1]?

因为next[j] = k, 所以p[0, k-1] = p[j-k, j-1]。

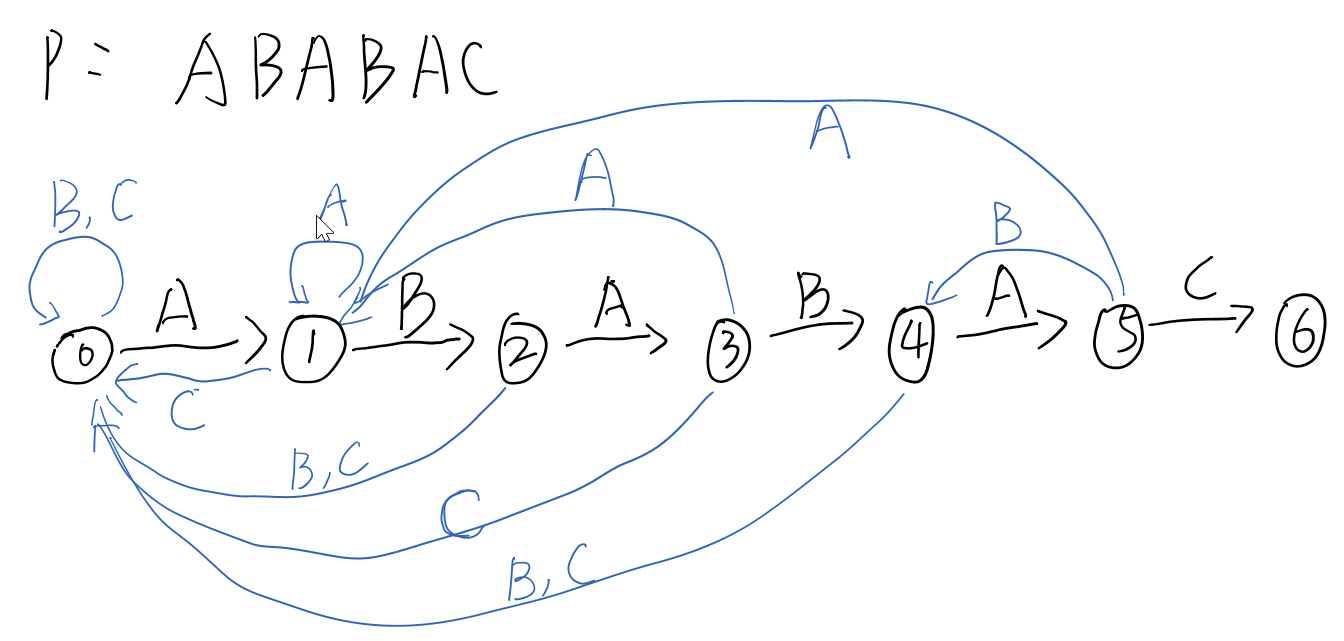
显然，如果p[k] == p[j], next[j+1] = next[j] + 1;

如果p[k] != p[j], 情况会复杂一点。虽然p[k] != p[j], 但p[0, k-1]和p[j-k, j-1]还是匹配的，然而我们在算next[j+1]的时候，必须要p[j]在内，所以，我们现在看看怎么能在保证p[?, j-1]和前缀匹配的情况下，还能加上p[j]。

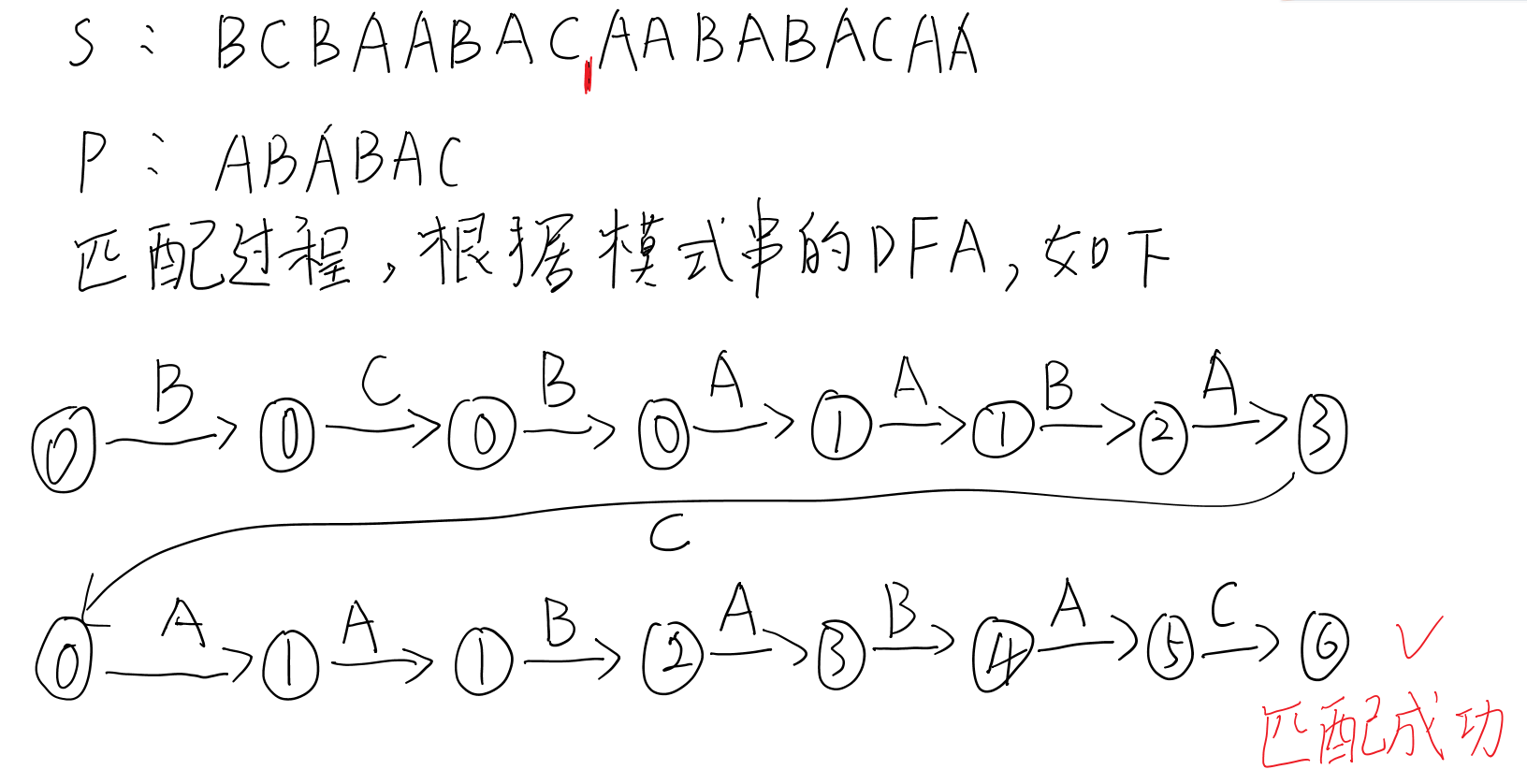
这里，根据next数组定义，我们有p[0, next[k]-1]和p[k-next[k], k-1]是匹配的，又有p[k-next[k], k-1]和p[j-next[k], j-1]是匹配的，因为p[0, k-1]和p[j-k, j-1]还是匹配的，前者是它们的一部分。所以p[0, next[k]-1]和p[j-next[k], j-1]是匹配的。这样就保证了p[?, j-1]和前缀是匹配的，现在只需要再看p[next[k]]和p[j]是否相等即可，若相等则next[j+1] = next[k]， 否则迭代，直到next[j+1] = 0。

1. 利用DFA有限状态机，模拟模式串匹配的过程，对于模式串的每个位置，源字符串出现任何字符都会对应一个状态，只有每个字符都完全匹配了，才会达到最终状态，才能匹配成功。

例如模式串ABABAC的DFA如下：



根据上面模式串的DFA去匹配源字符串，过程如下：

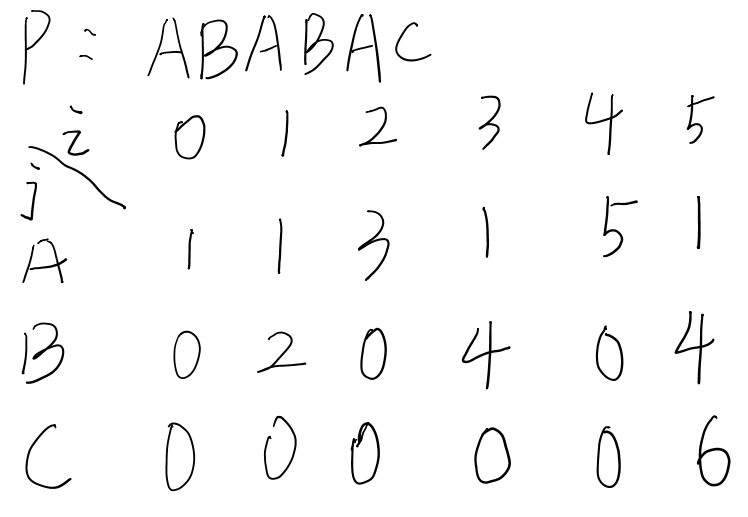


**主要思想：**

我们把模式串看成一个DFA，而源字符串是一系列字符输入，DFA根据源字符串的输入进行状态变更，如果DFA能达到最终状态，就证明源字符串中包含模式串，因为只有按照模式串的字符顺序输入，DFA才能达到最终状态，也就意味着匹配成功；如果没有达到最终状态，那就代表匹配失败。现在，关键就在于DFA的构造。

**DFA构造：**

这里我们用一个二维数组dfa[][]来表示DFA，dfa[i][j]的含义是当前状态是j，遇到了i字符之后的状态。例如对于上面的模式串ABABAC，dfa[‘B’][1] = 2。完整dfa[][]如下：



那么，我们的问题变成了求解dfa[][],再求解dfa[][]之前，我们先来了解一个概念，模式串的重启状态，这个重启状态是针对模式串的每个位置来讲的。

重启状态：当在源串s的位置i的字符s[i]和模式串p的位置j的字符p[j]匹配失败时，我们移动模式串，当模式串又能成功匹配到s[i]时，这时模式串所对应的状态，不妨令这时模式串中要与s[i]匹配的是p[k]，说白了p[j]的重启状态就是k。为什么呢？因为模式串在k之前有k-1个字符，只有经过了0,1,2,…,k-1才能到k,而每经过一个字符，模式串的状态都升一级，从0升到k。其实这里含义就是，当我们在p[j]匹配失败时，我们直接将状态设为k，再从s[i]开始继续匹配就好。重启状态也可以这么理解,除了模式串p[0, j]与源串s[?, i]完全匹配之外，到达s[i]之前，模式串p能有多少与源串匹配成功。不妨设为t，那么我们直接以状态t从匹配失败的位置开始继续匹配。

重启状态的求解：根据以上定义，我们可以得出，若模式串第j位的重启状态为X，那么第j+1位的重启状态为dfa[p[j]][X]，即在第j位之前，我们匹配了X位，状态是X，那么第j+1位的重启状态就是X遇到p[j]之后的状态。

dfa[][]的求解：既然第j位的重启状态是X，那dfa[?][j] = dfa[?][X]，除了dfa[p[j]][j] = j+1。因为只有出现p[j]时，才匹配成功，状态升级，其他都会匹配失败。那根据上面讲的，匹配失败相当于模式串以重启状态X开始，从s[i]继续匹配，那dfa[?][j] = dfa[?][X]。

代码实现：

|  |
| --- |
| public class KMP {  private String pat;  private int[][] dfa;  public KMP(String pat) {  this.pat = pat;  int M = pat.length();  int R = 256;  dfa = new int[R][M];  dfa[pat.charAt(0)][0] = 1;  for(int X=0, j=1; j<M; j++) {  //计算dfa[?][j]  for(int c=0; c<R; c++) {  dfa[c][j] = dfa[c][X];  }  dfa[pat.charAt(j)][j] = j+1;  X = dfa[pat.charAt(j)][X];  }  }  public int search(String txt) {  //在txt上模拟DFA的运行  int i, j, N = txt.length(), M = pat.length();  for(i=0, j=0; i<N && j<M; i++) {  j = dfa[txt.charAt(i)][j];  }  if(j == M) return i-M; // 找到匹配（到达模式字符串的结尾）  else return N; //未找到匹配（达到文本字符串的结尾）  }  } |

## 动态规划

### 1 KMP算法