KMP& Trie & AC自动机

算法设计与分析

字符串入门简介
以及KMP算法

By peterpan

引言

在字符串初级算法里，最基础的算法是字符串匹配算法，而其中最经典的模型问题 就是判断一个串是否是另一个串的子串。关于字符串算法有一个通用的解题方法，即抓住问题相关的字符串的某种性质，通过这个字符串性质来高效解决问题。很多人可能会认为字符串题目很有难度，不知道从何入手，变化多端，难以琢磨。其实不然，字符串算法的不同模型可能有很多，

引言

但，只要能把特定模型的基本方法、原理弄透，不管题目如何变化多端，都万变不离其中，可以用特定的方法去解决，无非有时需要配上一些额外技巧、科技罢了。所以，对于初学算法的人，切勿一开始就对字符串形成望而生怯的毛病，而应该养成正确思考字符串问题的好习惯——用算法对应的字符串性质来解题.

KMP算法

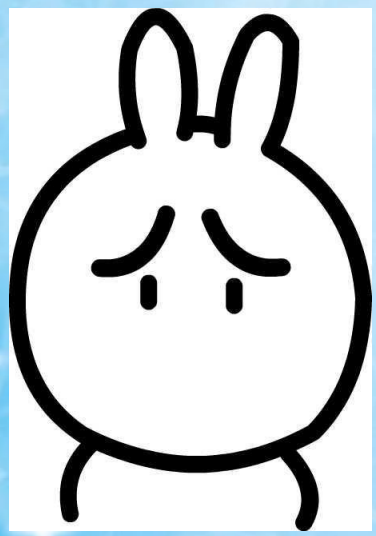
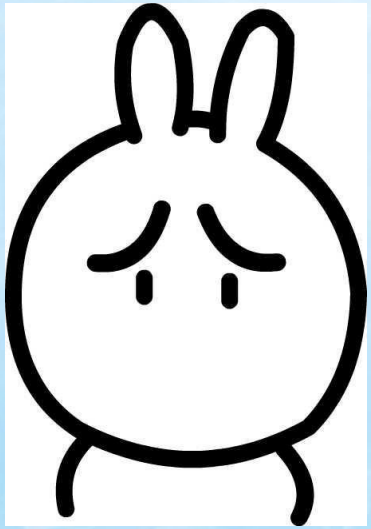
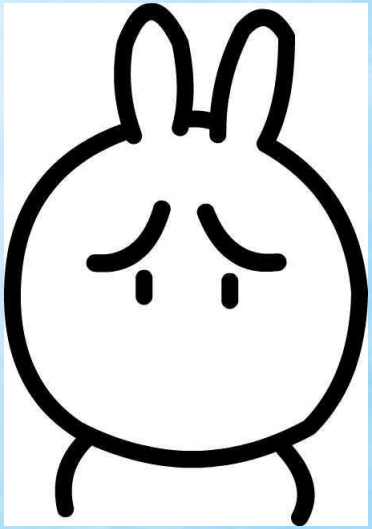
- 在字符串算法里，最简单、基础的就是KMP算法。下面就来看一下
- KMP算法能处理什么样的问题?
- KMP算法用到了什么字符串性质?
- KMP算法实现原理是什么?
- KMP算法的时间复杂度?
- KMP算法的拓展应用?

问题

下面我们先来看这么一个问题：

给两个字符串S1和S2，问S2是否是S1的子串？

数据范围：0<|S2|<=|S1|<=100000

输入和输出

Input

第一行输入一个字符串S1，第二行输入一个字符串S2，保证0<|S2|<=|S1|<=100000

Output

如果S2是S1的子串，输出“YES”.

否则，输出“NO”.

样例

- Sample Input
- AAABAAABAAABAAAD
- AAABAAAD

- ABAAB
- ABB

- Sample Output
- YES
- NO

一个字符串的子串指的是字符串某一段连续的部分（比如第一个例子），可以是其本身。
而不连续的部分，一般称作为子序列！（比如第二个例子，ABB是ABAAB的子序列而不是子串）

KMP算法用到了什么字符串性质

- 下面介绍下KMP用到的字符串性质
- 是否还记得刚才说的?
- 每个字符串算法都对应它的字符串性质!
- 我们定义这么一个字符串性质，叫前缀后缀最大值!
- 光从定义来看，似乎是和字符串的前缀、后缀有关系，同时告诉你，最大值指的是长度最大，什么长度最大？下面具体来看下这个性质。

前缀后缀最大值

- 一个长度为N的字符串S，它有N+1个前缀（包括空前缀），它有N+1个后缀（包括空后缀）
- 比如ABC，有4个前缀，空，A，AB，ABC
  有4个后缀，空，C，BC，ABC
- 比如AAA，有4个前缀，空，A，AA，AAA
  有4个后缀，空，A，AA，AAA

前缀后缀最大值

举一个容易看出性质的例子，S=ABABABA

前缀	后缀	相等
空	空	yes
A	A	yes
AB	BA	no
ABA	ABA	yes
ABAB	BABA	no
ABABA	ABABA	yes
ABABAB	BABABA	no
ABABABA	ABABABA	yes

容易发现，S有5个前缀与后缀相等，如果我们不算自身，即前缀ABABABA不算，后缀ABABABA不算，那么，在所有相等的<前缀，后缀>里，长度最大的就是ABABA，则前缀后缀最大值就是5！

前缀后缀最大值

一个字符串S，长度为N.

找出它的N+1个前缀（包括空前缀）

找出它的N+1个后缀（包括空后缀）

按照长度划分，得到N+1对序偶<前缀，后缀>

删除前缀、后缀等于S的<前缀，后缀>，得到N对<前缀，后缀>。

在这N对中，找到一对满足：

1. 前缀 = 后缀　　2. 前缀后缀的长度最大

该长度就是S的前缀后缀最大值！

前缀后缀最大值

下面我们拿暴力匹配算法中的模板串S=AAABAAAD 来具体阐述！

我们定义一个数组：int next[N];

next[i]表示 S[0...i-1] 这个前缀的前缀后缀最大值！

接下来，我们分析字符串S的每一个前缀的next值，即每一个前缀的前缀后缀最大值。

然后，我们再介绍如果使用这个next数组！

重要的话要多讲几遍！

next[i]表示 S[0...i-1] 这个前缀的前缀后缀最大值！

next[i]表示 S[0...i-1] 这个前缀的前缀后缀最大值！

请务必牢记这个定义！KMP核心部分就是这个next数组。对next数组的理解透彻与否决定你能否快速、准确地解决KMP相关的题目！注意，是S[0...i-1] 不是S[0...i]！

同时务必正确理解前缀后缀最大值的含义！

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1								

S=AAABAAAD，第0个前缀：空前缀

空前缀的next值我们直接定义为 -1

即next[0]=-1. 记得之前的“删除前缀、后缀等于S的<前缀，后缀>”的操作吗?

删除后找不到<前缀，后缀>，next值为-1

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0							

S=AAABAAAD，第1个前缀：A

next[1]=0. 表示S[0...0]的前缀后缀最大值是0

一个前缀：空

一个后缀：空

显然，next[1]=|空| = 0

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1						

S=AAABAAAD，第2个前缀：AA

next[2]=1. 表示S[0...1]的前缀后缀最大值是1

两个前缀：空，A

两个后缀：空，A

显然，next[2]=|A|=1

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2					

S=AAABAAAD，第3个前缀：AAA

next[3]=2. 表示S[0...2]的前缀后缀最大值是2

三个前缀：空，A，AA

三个后缀：空，A，AA

显然，next[3]=|AA|=2

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0				

S=AAABAAAD，第4个前缀：AAAB

next[4]=0. 表示S[0...3]的前缀后缀最大值是0

四个前缀：空，A，AA，AAA

四个后缀：空，B，AB，AAB

显然，next[4]=|空|=0

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1			

S=AAABAAAD，第5个前缀：AAABA

next[5]=1. 表示S[0...4]的前缀后缀最大值是1

五个前缀：空，A，AA，AAA，AAAB

五个后缀：空，A，BA，ABA，AABA

显然，next[5]=|A|=1

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1	2		

S=AAABAAAD，第6个前缀：AAABAA

next[6]=2. 表示S[0...5]的前缀后缀最大值是2

空, A, AA, AAA, AAAB, AAABA

空, A, AA, BAA, ABAA, AABAA

显然，next[6]=|AA|=2

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1	2	3	

S=AAABAAAD，第7个前缀：AAABAAA

next[7]=3. 表示S[0...6]的前缀后缀最大值是3

空, A, AA, AAA, AAAB, AAABA, AAABAA

空, A, AA, AAA, BAAA, ABAAA, AABAAA

显然，next[7]=|AAA|=3

分析S每一个前缀的next值

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1	2	3	0

S=AAABAAAD，第8个前缀：AAABAAAD

next[8]=0. 表示S[0...7]的前缀后缀最大值是0

空,A,AA, AAA, AAAB, AAABA, AAABAA, AAABAAA

空,D,AD,AAD, AAAD, BAAAD, ABAAAD, AABAAAD

显然，next[8]=|空|=0

得到next数组

i	0	1	2	3	4	5	6	7	8
S	A	A	A	B	A	A	A	D	
next	-1	0	1	2	0	1	2	3	0

我们得到S串的next数组

再次回忆，next[i]表示S[0...i-1]这个前缀的前缀后缀最大值。

那么，有什么用呢？！

它与子串匹配有什么关系呢?

回顾与分析

给两个字符串S1和S2，问S2是否是S1的子串?

S1=AAABAAABAAABAAAD

S2=AAABAAAD

在这个问题中，我们得到文本串S1和模板串S2.

现在想判断S2(单词)是否是S1(文章)的子串。

我们先对模板串S2，构建next数组，得到S2每一个前缀的前缀后缀的最大值。

接下来，开始KMP匹配过程!

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

设两个变量 int t1，t2；t1表示当前扫到S1串的位置，t2表示当前扫到S2串的位置。起初，t1=t2=0;

S1[t1]=S2[t2]=A，t1++，t2++；//t1=1，t2=1

S1[t1]=S2[t2]=A，t1++，t2++；//t1=2，t2=2

S1[t1]=S2[t2]=A，t1++，t2++；//t1=3，t2=3

S1[t1]=S2[t2]=B，……

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

此时t1=t2=3;

S1[t1]=S2[t2]=B，t1++，t2++；//t1=t2=4

S1[t1]=S2[t2]=A，t1++，t2++；//t1=t2=5

S1[t1]=S2[t2]=A，t1++，t2++；//t1=t2=6

S1[t1]=S2[t2]=A，t1++，t2++；//t1=t2=7

此时发现，S1[t1] != S2[t2]

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

此时，t1=7, t2=7, S1[t1]=B, S2[t2]=D 出现不相等！

在之前的暴力匹配算法中，这说明了什么？

说明了从 t1=0 这个起始位置开始，往后长度为|S2|的子串不与S2匹配，那么，在暴力算法中，接下来应该枚举下一个起始位置 t1=1, 再往下判断, 是吧？但是！在KMP中有所不同！……

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

在KMP中，S1[7] !=S2[7]，此时出现了失配!

怎么办呢？我们查询7号位置的next值，即next[7]=3.

然后, 我们直接令t2=next[t2]=next[7] (next[7]=3);相当于把S2右移了4格，下面，我们先来看下这个神奇的变化！再分析下这么做的理由和优势。

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2`	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							
S2					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			

S1未动，S2整体往右移动了 |S2| - next[7] 个格子

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			

此时，我们只需从 t1=7，t2=3 这个匹配对开始往下继续匹配即可。而S1[4...6]与S[0...2]相当于已经匹配成功了，不需要再匹配。

想想看，为什么可以这么做?

（以下过程ppt不是很好演示，可能需要板书）

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

我们在匹配时，是不是已经得到S1[0...6]=S2[0...6]了？

然后，通过next数组，next[7]=3，是不是我们就知道S2[0...2]=S2[4...6]这个性质？于是我们就可以推得:S2[0...2]=S1[4...6],又由于next[7]表示S2[0..6]这个前缀的前缀后缀最大值！所以，这样挪动是正确的！

正确可行的匹配一定会延续到S1的位置7，这样挪动和将i退回到1是等价的，退回到1，新的匹配也是S1的后缀和S2的前缀匹配。

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2	A	A	A	B	A	A	A	D								
j	0	1	2	3	4	5	6	7	8							
next	-1	0	1	2	0	1	2	3	4							

换句话说，我们在S1[7]与S2[7]处出现了失配，此时t1不需要返回，只需改变t2。在7处失配，则只需查询S2[0...6]处的前缀后缀最大值，表示某一段前缀等于后缀，而又是长度最大的！那么移动后，失配点的前段一定还是匹配的，而只需在从失配点往下匹配即可，若失配点还是失配，再继续改变t2

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			

失配点是 i=7,j=3,失配点的前一段S1[4...6]=S2[0..2]仍然匹配，现在只需从失配点，t1=7，t2=3，继续往下匹配即可...

当next[t2]==-1,表示S2的首个字符和S1的第i个字符都不匹配，那么i就加1

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			

好！我们继续往下匹配，发现t1=11, t2=7的时候，又出现了失配！与刚才同样的步骤，令t2=next[t2]=next[7].

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2`					A	A	A	B	A	A	A	D				
j					0	1	2	3	4	5	6	7	8			
next					-1	0	1	2	0	1	2	3	4			
S2									A	A	A	B	A	A	A	D
j									0	1	2	3	4	5	6	7
next									-1	0	1	2	0	1	2	3

S1未动，S2整体往右移动了 |S2| - next[7] 个格子

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2									A	A	A	B	A	A	A	D
j									0	1	2	3	4	5	6	7
next									-1	0	1	2	0	1	2	3

从 i=11，j=3这个匹配对，继续往下匹配...

前一段S1[8...10]与S2[0...2]已经匹配成功！

KMP匹配

S1	A	A	A	B	A	A	A	B	A	A	A	B	A	A	A	D
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S2									A	A	A	B	A	A	A	D
j									0	1	2	3	4	5	6	7
next									-1	0	1	2	0	1	2	3

继续往下匹配，都能匹配成功！

发现S2[0...7]与S1[8...15]完全匹配成功！

S2是S1的子串。

代码：构建next数组

```
void get_next(int *next,char *s2,int lens){
  //用于构建s2的next数组，kmp的前奏
   int t1=0,t2;
   next[0]=t2=-1;
   while(t1<lens){
      if(t2==-1||s2[t1]==s2[t2]){
         next[t1+1]=t2+1;
         t1++;
         t2++;
      }
      else t2=next[t2];
   }
}
```

代码：KMP匹配

```
bool kmp(int *next,char *s1,int lens1,char *s2,int
lens2){ //用于判断S2是否是S1的子串
    int t1=0,t2=0;
    while(t1<lens1&&t2<lens2){
        if(t2==-1||s1[t1]==s2[t2]){
            t1++;
            t2++;
        }
        else t2=next[t2];
    }
    if(t2==lens2) return true;//S2是S1子串
    else return false;//S2不是S1子串
}
```

代码：KMP匹配2

```
int kmp(int *next,char *s1,int lens1,char *s2,int lens2){
    int t1=0,t2=0;
    int times=0;
    while(t1<lens1){
        if(t2==-1||s1[t1]==s2[t2]){
            t1++;
            t2++;
        }
        else t2=next[t2];
        if(t2==lens2){
            times+=1;
            t2=next[t2];
        }
    } //求S2在S1中出现了多少次
    return times;
}
```

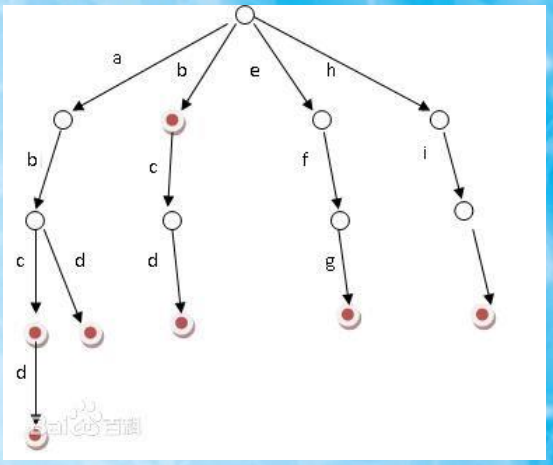
KMP的用途

1. 判断一个串是否是另一个串的子串。
2. 判断一个串在另一个串出现了多少次。
3. 求一个字符串的最小循环节.
4. 进行各式各样的字符串匹配，模糊匹配等

（kmp匹配只是最经典的匹配一种，很多时候是需要在失配函数上做文章，完成另类的匹配）

5. …… 等等

Trie树



- Trie树
  - 又称单词查找树，Trie树，是一种树形结构，是一种哈希树的变种。典型应用是用于统计，排序和保存大量的字符串（但不仅限于字符串），所以经常被搜索引擎系统用于文本词频统计。它的优点是：利用字符串的公共前缀来减少查询时间，最大限度地减少无谓的字符串比较，查询效率比哈希表高。
  - 三个性质:
    - 根节点不包含字符，除根节点外每一个节点都只包含一个字符； 从根节点到某一节点，路径上经过的字符连接起来，为该节点对应的字符串； 每个节点的所有子节点包含的字符都不相同。
  - 搜索字典项目的方法为：
    - (1) 从根结点开始一次搜索；
    - (2) 取得要查找关键词的第一个字母，并根据该字母选择对应的子树并转到该子树继续进行检索；
    - (3) 在相应的子树上，取得要查找关键词的第二个字母,并进一步选择对应的子树进行检索。
    - (4) 迭代过程……
    - (5) 在某个结点处，关键词的所有字母已被取出，则读取附在该结点上的信息，即完成查找。
  - 其他操作类似处理

Trie树

- Trie树的结构
  - Trie树的入口称为root，root点除引出字符外不保存任何信息。
  - 其他节点保存的信息有：
    - count：到此节点处截止的单词个数，如没有单词以该节点结尾，则count值为-1
    - next[26]：记录下面的节点（26对应a-z,若大小写混合，next数组开到52即可）

```
#define MAX 26//字符集大小
typedef struct TrieNode
{
    int nCount;//记录该字符出现次数
    struct TrieNode* next[MAX];
}TrieNode;

TrieNode Memory[1000000];
int allocp=0;

/*初始化*/
void InitTrieRoot(TrieNode* *pRoot)
{
    *pRoot=NULL;
}

/*创建新结点*/
TrieNode* CreateTrieNode()
{
    int i;
    TrieNode *p;
    p=&Memory[allocp++];
    p->nCount=1;
    for(i=0;i<MAX;i++)
    {
        p->next[i]=NULL;
    }
    return p;
```

```
/*插入*/
void InsertTrie(TrieNode* *pRoot,char *s)
{
    int i,k;
    TrieNode*p;
    if(!(p=*pRoot))//*pRoot为NULL是，表示要初始化树
    {
        p=*pRoot=CreateTrieNode();
    }
    i=0;
    while(s[i])//假设字符串以0结尾
    {
        k=s[i++]-'a';//确定branch
        if(!p->next[k])//还没有这个字符
            p->next[k]=CreateTrieNode();
         else
            p->next[k]->nCount++;
        p=p->next[k];
    }
}
```

```
//查找
int SearchTrie(TrieNode* *pRoot,char *s)
{
    TrieNode *p;
    int i,k;
    if(!(p=*pRoot))
    {
        return0;
    }
    i=0;
    while(s[i])
    {
        k=s[i++]-'a';
        if(p->next[k]==NULL) return 0;
        p=p->next[k];
    }
    return p->nCount;
}
```

AC自动机

- 自动机
  - 计算机控制系统的控制程序具有有限状态自动机（FA）的特征，可以用有限状态机理论来描述。有限自动机（Finite Automata Machine）是计算机科学的重要基石，它在软件开发领域内通常被称作有限状态机（Finite State Machine），是一种应用非常广泛的软件设计模式。
  - 有限状态机，（英语：Finite-state machine, FSM），又称有限状态自动机，简称状态机，是表示有限个状态以及在这些状态之间的转移和动作等行为的数学模型。
  - 状态存储关于过去的信息，就是说：它反映从系统开始到现在时刻的输入变化。转移指示状态变更，并且用必须满足来确使转移发生的条件来描述它。动作是在给定时刻要进行的活动的描述。有多种类型的动作：
  - 进入动作（entry action）：在进入状态时进行
  - 退出动作：在退出状态时进行
  - 输入动作：依赖于当前状态和输入条件进行
  - 转移动作：在进行特定转移时进行

AC自动机

- 关于AC自动机
  - AC自动机：Aho-Corasick automation，该算法在1975年产生于贝尔实验室，是著名的多模匹配算法之一。一个常见的例子就是给出n个单词，再给出一段包含m个字符的文章，让你找出有多少个单词在文章里出现过。要搞懂AC自动机，先得有模式树（字典树）Trie和KMP模式匹配算法的基础知识。AC自动机算法分为3步：构造一棵Trie树，构造失败指针和模式匹配过程。
  - 简单来说，AC自动机是用来进行多模式匹配（单个主串，多个模式串）的高效算法。

AC自动机

- AC自动机的构造过程
  - 使用Aho-Corasick算法需要三步：
  - 1.建立模式串的Trie
  - 2.给Trie添加失败路径
  - 3.根据AC自动机，搜索待处理的文本

我们以下面这个例子来介绍AC自动机的运作过程

例子：给定5个单词：say she shr he her，然后给定一个字符串　yasherhs。问一共有多少单词在这个字符串中出现过。

AC自动机

- 确定数据结构
  - 首先，我们需要确定AC自动机所需的数据存储结构，它们的用处之后会讲到。

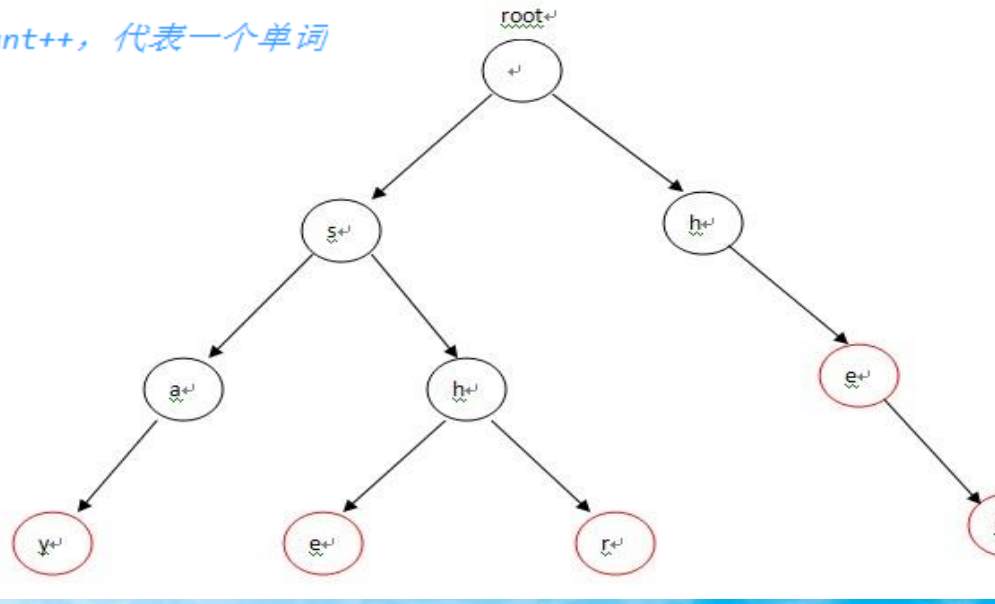
```
struct node
{
    node *fail;          //失败指针
    node *next[26]; //Tire每个节点的个子节点（最多个字母）
    int count;           //是否为该单词的最后一个节点
    node()               //构造函数初始化
    {
        fail=NULL;
        count=0;
        memset(next,NULL,sizeof(next));
    }
}*q[500001];             //队列，方便用于bfs构造失败指针
char keyword[51];        //输入的单词
char str[1000001];       //模式串
int head,tail;           //队列的头尾指针
```

AC自动机

- 第一步：构建Trie

```
void insert(char *str,node *root)
{
    node *p=root;
    int i=0,index;
    while(str[i])
    {
        index=str[i]-'a';
        if(p->next[index]==NULL) p->next[index]=new node();
        p=p->next[index];
        i++;
    }
    p->count++; //在单词的最后一个节点count++，代表一个单词
}
```

– 构建完成后的效果

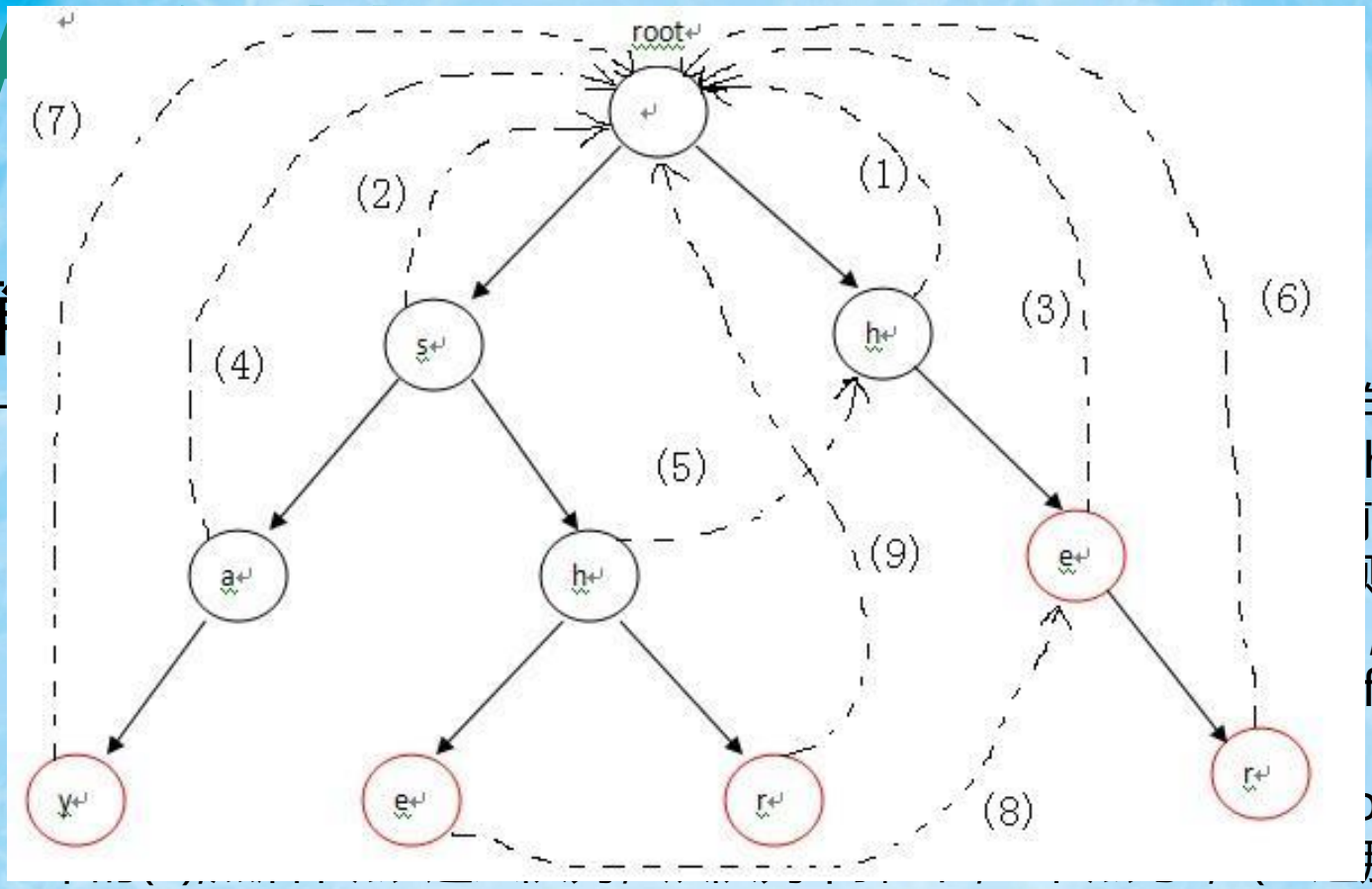


AC自动机

- 第二步：构建失败指针
  - 构建失败指针是AC自动机的关键所在，可以说，若没有失败指针，所谓的AC自动机只不过是Trie树而已。
  - 失败指针原理：
  - 构建失败指针，使当前字符失配时跳转到另一段从root开始每一个字符都与当前已匹配字符段某一个后缀完全相同且长度最大的位置继续匹配，如同KMP算法一样，AC自动机在匹配时如果当前字符串匹配失败，那么利用失配指针进行跳转。由此可知如果跳转，跳转后的串的前缀必为跳转前的模式串的后缀，并且跳转的新位置的深度（匹配字符个数）一定小于跳之前的节点（跳转后匹配字符数不可能大于跳转前，否则无法保证跳转后的序列的前缀与跳转前的序列的后缀匹配）。所以可以利用BFS在Trie上进行失败指针求解。
  - 失败指针利用：
  - 如果当前指针在某一字符s[m+1]处失配，即(p->next[s[m+1]]==NULL)，则说明没有单词s[1...m+1]存在，此时，如果当前指针的失配指针指向root，则说明当前序列的任何后缀不是某个单词的前缀，如果指针的失配指针不指向root，则说明当前序列s[i...m]是某一单词的前缀，于是跳转到当前指针的失配指针，以s[i...m]为前缀继续匹配s[m+1]。
  - 对于已经得到的序列s[1...m]，由于s[i...m]可能是某单词的后缀，s[1...j]可能是某单词的前缀，所以s[1...m]中可能会出现单词，但是当前指针的位置是确定的，不能移动，我们就需要temp临时指针，令temp=当前指针，然后依次测试s[1...m],s[i...m]是否是单词。
  - >>>简单来说，失败指针的作用就是将主串某一位之前的所有可以与模式串匹配的单词快速在Trie树中找出。

AC自动机

- 第二步：构建失败指针
  - 在构造完Tire树之后，接下去的工作就是构造失败指针。构造失败指针的过程概括起来就一句话：设这个节点上的字母为C，沿着它父亲节点的失败指针走，直到走到一个节点，它的子结点中也有字母为C的节点。然后把当前节点的失败指针指向那个字母也为C的儿子。如果一直走到了root都没找到，那就把失败指针指向root。具体操作起来只需要：先把root加入队列(root的失败指针指向自己或者NULL)，这以后我们每处理一个点，就把它的所有儿子加入队列。



- 第

– 与NULL
h节点相
i后缀，
收指针的
，h所连
fail指针

ot,对应图
那个)相

连，先遍历到a节点，扫描指针指向a节点的父节点s节点的fail指针指向的节点，也就是root，root->next['a']==NULL,并且root->fail==NULL,说明匹配序列为空,则把节点a的fail指针指向root,对应图中的(4),然后节点a进入队列。接着遍历到h节点，扫描指针指向h节点的父节点s节点的fail指针指向的节点，也就是root，root->next['h']!=NULL,所以把节点h的fail指针指向右边那个h,对应图中的(5),然后节点h进入队列...由此类推，最终失配指针如下图所示。

AC自动机

- 第二步：构建失败指针
  - 为什么上述那个方法是可行的，是可以保证从root到所跳转的位置的那一段字符串长度小于当前匹配到的字符串长度且与当前匹配到的字符串的某一个后缀完全相同且长度最大呢?
  - 显然我们在构建失败指针的时候都是从当前节点的父节点的失败指针出发，由于Trie树将所有单词中相同前缀压缩在了一起，所以所有失败指针都不可能平级跳转（到达另一个与自己深度相同的节点），因为如果平级跳转，很显然跳转所到达的那个节点肯定不是当前匹配到的字符串的后缀的一部分，否则那两个节点会合为一个，所以跳转只能到达比当前深度小的节点，又因为是由当前节点父节点开始的跳转，所以这样就可以保证从root到所跳转到位置的那一段字符串长度小于当前匹配到的字符串长度。另一方面，我们可以类比KMP求NEXT数组时求最大匹配数量的思想，那种思想在AC自动机中的体现就是当构建失败指针时不断地回到之前的跳转位置，然后判断跳转位置的下一个字符是否包含当前字符，如果是就将失败指针与那个跳转位置连接，如果跳转位置指向NULL就说明当前匹配的字符在当前深度之前没有出现过，无法与任何跳转位置匹配，而若是找到了第一个跳转位置的下一个字符包含当前字符的的跳转位置，则必然取到了最大的长度，这是因为其余的当前正在匹配的字符必然在第一个跳转位置的下一个字符包含当前字符的的跳转位置深度之上，而那样的跳转位置就算可以，也不会是最大的（最后一个字符的深度比当前找到的第一个可行的跳转位置的最后一个字符的深度小，串必然更短一些）。
  - 这样就证明了这种方法构建失败指针的可行性。

AC自动机

- 第二步：构建失败指针
  - 代码:

```
void build_ac_automation(node *root)
{
    int i;
    root->fail=NULL;
    q[head++]=root;
    while(head!=tail)
    {
        node *temp=q[tail++];
        node *p=NULL;
        for(i=0; i<26; i++)
        {
            if(temp->next[i]!=NULL)
            {
                if(temp==root) temp->next[i]->fail=root;
                else
                {
                    p=temp->fail;
                    while(p!=NULL)
                    {
                        if(p->next[i]!=NULL)
                        {
                            temp->next[i]->fail=p->next[i];
                            break;
                        }
                        p=p->fail;
                    }
                    if(p==NULL) temp->next[i]->fail=root;
                }
                q[head++]=temp->next[i];
            }
        }
    }
}
```

AC自动机

- 第三步：匹配
  - 最后，我们便可以在AC自动机上查找字符串中出现过哪些单词了。匹配过程分两种情况：(1)当前字符匹配，表示从当前节点沿着树边有一条路径可以到达目标字符，此时只需沿该路径走向下一个节点继续匹配即可，目标字符串指针移向下个字符继续匹配；(2)当前字符不匹配，则去当前节点失败指针所指向的字符继续匹配，匹配过程随着指针指向root结束。重复这2个过程中的任意一个，直到模式串走到结尾为止。
  - 对例子来说：字符串为yasherhs。对于i=0,1。Trie中没有对应的路径，故不做任何操作；i=2,3,4时，指针p走到左下节点e。因为节点e的count信息为1，所以cnt+1，并且将节点e的count值设置为-1，表示该单词已经出现过了，防止重复计数，最后temp指向e节点的失败指针所指向的节点继续查找，以此类推，最后temp指向root，退出while循环，这个过程中count增加了2。表示找到了2个单词she和he。当i=5时，程序进入第5行，p指向其失败指针的节点，也就是右边那个e节点，随后在第6行指向r节点，r节点的count值为1，从而cnt+1，循环直到temp指向root为止。最后i=6,7时，找不到任何匹配，匹配过程结束。
    - AC自动机时间复杂性为：O(n+m)其中m是模式串总长度，n是字符串总长度。

AC自动机

- 第三步：匹配
  - 代码：

```
int query(node *root)
{
    int i=0,cnt=0,index,len=strlen(str);
    node *p=root;
    while(str[i])
    {
        index=str[i]-'a';
        while(p->next[index]==NULL && p!=root) p=p->fail;
        p=p->next[index];
        p=(p==NULL)?root:p;
        node *temp=p;
        while(temp!=root)
        {
            if(temp->count>=0)
            {
                cnt+=temp->count;
                temp->count=-1;
            }
            else break;
            temp=temp->fail;
        }
        i++;
    }
    return cnt;
}
```

  - 这就没什么多说的了，代码都是次要的，主要是原理。

AC自动机

- AC自动机的讲解就这么愉快地结束了
  - 我们可以发现KMP算法与AC自动机有着相通之处。
  - 然后我们思考一个问题，AC自动机如何打出匹配位置？
  - 我们是不是可以这样：
    - 给每个模式串指定一个编号，编号不重复
    - 模式串Trie树中每个包含单词的结点添加一个变量表示这个单词的编号
    - 因为模式串如果被找到，必然是主串当前搜索位置之前的子字符串的后缀，就用当前搜索位置i+1-strlen(模式串)即可
    - 事实证明这是正确的，详见std

习题

- 习题1
  - Hdu 2222 Keywords Search
  - http://acm.hdu.edu.cn/showproblem.php?pid=2222
  - 题意：给出N个单词，和一个10^6长的字符串，求这些单词在字符串中出现的频率。
  - 思路：将N个单词构造成一棵字典树，然后利用AC自动机建立fail指针，将字符串在自动机上匹配，得到单词频率。

习题

- 习题2
  - Hdu 2896 病毒侵袭
  - http://acm.hdu.edu.cn/showproblem.php?pid=2896
  - 题意：N种病毒特征-128种字符组成的字符串；M个网站-10000长字符串，查询M个字符串中存在哪些病毒，由小到大输出存在的病毒编号，并在最后输出含有病毒的网站的个数。
  - 思路：与hdu2222两个区别就是本题需要输出含有字符串的编号，就加个变量记录病毒串的编号，在查询的时候存入一个ans[]数组中就OK；本题不能再查询完病毒串之后进行标记的优化，因为有多个网站所以又多组查询。

```
while(ptr != root && ptr->flag != -1){
        if(ptr->flag) ans[V++] = ptr->flag;
        //ptr->flag = -1;
        //can't flag visited , cause mutiple cases
        ptr = ptr->fail;
    }
```

习题

- 习题3
  - 训练2-5第一题"Homework"
  - 利用KMP来做

囧囧的作业

【问题描述】

在美好的暑假，囧囧参加了 Flyioi 补习班。有一天老师给同学们布置了一项作业：

对于一个非负整数 n，b_n 表示 N 的二进制表示中 1 的个数的奇偶性(当 n 的二进制中有偶数个 1 时 $b_n=0$，否则 $b_n=1$)，对于 $n\geqslant 0$，b_n 形成了一个无限长的 01 序列。给定一个长度为 p 的序列 c[0..p-1]，请找出 c 在 b 中第一次出现的位置(即最小的非负整数 k，对于 $0\leqslant i<p$，$c_i=b_{i+k}$)

然后老师给了同学们一些 p 很小的数列，让同学们完成。当然老师希望同学们用电脑完成这个任务。但是囧囧认为此题过于 easy，决定全部用手算出答案。

老师知道了这件事后很生气，后果很严重。因此决定单独给囧囧布置一项作业：

给定一个长度为 p 的序列 c[0..p-1]，求出 c 的每一个前缀在 b 中第一次出现的位置。

当然，为了防止囧囧再次用手算出答案，老师准备了许多 p 超大的数据。这下囧囧真的囧了，你能帮帮他吗？

【输入格式】

第一行有一个正整数 T，表示数据组数。

每个数据包含两行，

第一行有一个正整数 p

第二行包含 p 个用空格隔开的 01 序列。

【输出格式】

每个数据输出一行，如果序列在 b 中不存在，则输出-1

【输入样例】

```
1
9
1 0 0 1 0 1 1 1 0
```

【输出样例】

```
1 2 4 4 8 8 8 -1 -1
```

【样例说明】

b 的前 16 项为 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0

【数据规模和约定】

在 20%的数据中，$p\leqslant 100$

在 100%的数据中，$T\leqslant 1000$，$p\leqslant 10^6$，单个数据的 $\Sigma p\leqslant 2*10^6$

习题

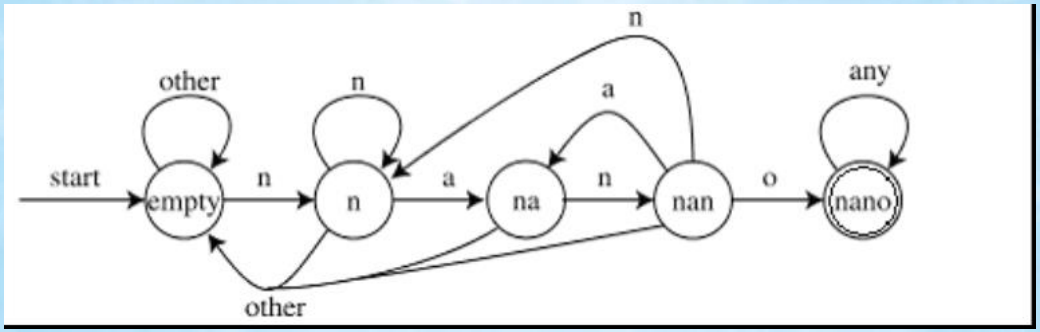
- 习题4
  - POJ 3691 DNA repair
  - 题目大意:
  - 给你N个致病基因(长度各不超过20), 再给你一个DNA一条链的碱基序列(长度M不超过1000), 要求尽量少的修改碱基, 使得序列不含致病基因, 不能的话输出-1.

习题

- Sample Input
- 2
- AAA
- AAG
- AAAG
- 2
- A
- TG
- TGAATG
- 4
- A
- G
- C
- T
- AGT
- 0
- Sample Output
- Case 1: 1
- Case 2: 4
- Case 3: -1

习题

- 习题4
  - 这道题用AC自动机+dp做与其他匹配多模式串的AC自动机最大的不同就是：一个结点j的KIND个孩子不会有空，就是说当从结点通过一个不能往下连接的字符时，孩子要通过j的fail结点来找到，或者回到根。
  - 如图:箭头就是next[]的指向



  - 所以在Trie树中安全结点中遍历时，就能得到到某个结点的修改值了。
  - 母串的前i个前缀在Trie树中遍历，到达某个安全结点j需要修改的值为dp[i][j]
  - dp[i][son]=min{dp[i][son],dp[i-1][j]+(str[i-1]!=char(j->son))} ;//son是j的孩子str是从0开始的
  - 初始状态为:dp[0][0]=0，其他为无穷　　//0是根结点

参考资料

- 参考资料
  - http://www.cnblogs.com/xudong-bupt/p/3433506.html
  - http://www.cppblog.com/mythit/archive/2009/04/21/80633.html
  - http://www.cppblog.com/myjfm/archive/2012/09/12/190396.html
  - 《ACM-ICPC程序设计系列 - 算法设计与实现》