

# 字符串模式匹配中DFA的应用

北京大学信息学院 郭炜

GWPL@PKU.EDU.CN

本讲义部分内容引用北大信息科学技术学院贺一骏讲义

# POJ1204 Word Puzzles

题目大意:

给出一个 $N*L$ 的字符矩阵，再给出 $M$ 个字符串，问这 $M$ 个字符串在这个字符矩阵中出现的位置。

MARGARITA

ALEMA

BARBECUE

数据范围:

$N, L \leq 1000$

$M \leq 1000$

时间限制: 5s

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	Q	W	S	P	I	L	A	A	T	I	R	A	G	R	A	M	Y	K	E	I
1	A	G	T	R	C	L	Q	A	X	L	P	O	I	J	L	F	V	B	U	Q
2	T	Q	T	K	A	Z	X	V	M	R	W	A	L	E	M	A	P	K	C	W
3	L	I	E	A	C	N	K	A	Z	X	K	P	O	T	P	I	Z	C	E	O
4	F	G	K	L	S	T	C	B	T	R	O	P	I	C	A	L	B	L	B	C
5	J	E	W	H	J	E	E	W	S	M	L	P	O	E	K	O	R	O	R	A
6	L	U	P	Q	W	R	N	J	O	A	A	G	J	K	M	U	S	J	A	E
7	K	R	Q	E	I	O	L	O	A	O	Q	P	R	T	V	I	L	C	B	Z
8	Q	O	P	U	C	A	J	S	P	P	O	U	T	M	T	S	L	P	S	F
9	L	P	O	U	Y	T	R	F	G	M	M	L	K	I	U	I	S	X	S	W

# 将问题抽象

将 **$N \times L$** 的字符矩阵中的每行、每列、每斜行，单独抽出得到了 **$N+L+2 \times (N+L-1)$** 个字符串，加上它们的各自的逆序，则得到的字符串的数目是：

$$2 \times (N+L+2 \times (N+L-1)) = 6N+6L-2$$

然后，现在的问题是判断之后给出的 **$M$** 个字符串出现在以上的那些字符串的什么位置。这里我们称前面抽象出来的 **$6N+6L-2$** 个串为原串，之后给出的 **$M$** 个串为模式串。

# 思考...

强行匹配？ 时间复杂度：  $O(NLMlen)$  ( $len$ 是模式串的平均长度)

$O(10^{12})$       太不靠谱了！！

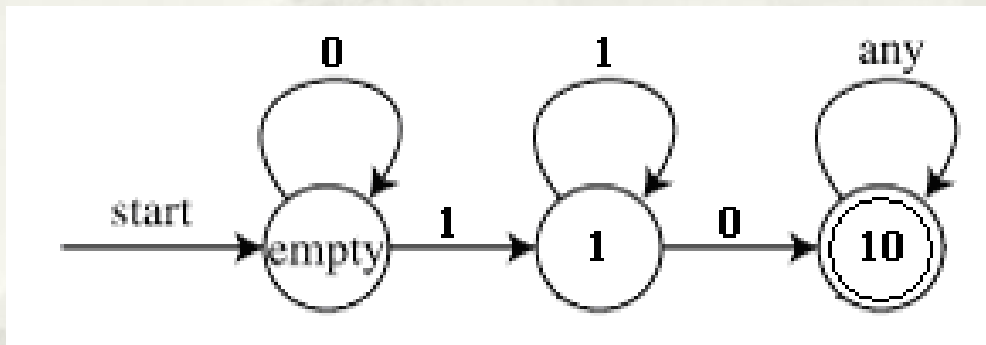
KMP？ 时间复杂度：  $O(NLM)$

$O(10^9)$       还是不能忍！！

# 确定性有限状态自动机

## DFA(deterministic finite automata)

**DFA**使用一个五元组( $Q, q_0, A, \Sigma, \delta$ )来描述, 这里 $Q$ 为状态集;  $q_0$ 为起始状态;  $A$ 为终态集;  $\Sigma$ 为字母表,  $\delta$ 为转移函数。用一个图来描述一个自动机:



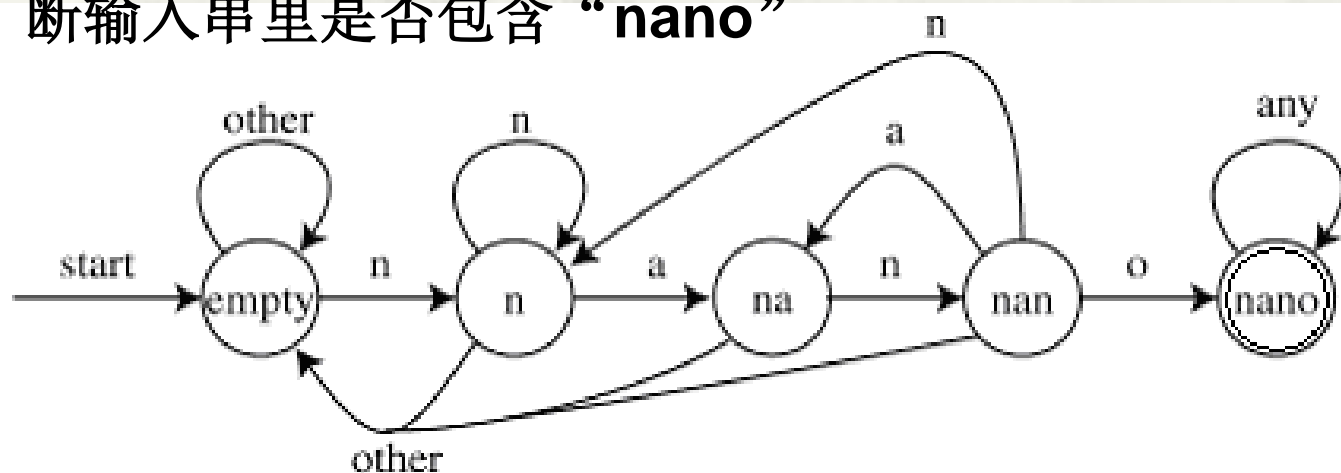
这是一个字符集为**01**的  
**DFA**

**S="001110"** 可以匹配它

图中圆圈代表状态, 箭头代表转移, 例如从状态“1”有一条字符**0**的边指向状态“10”, 就是说在状态“1”如果碰到输入是'**0**'那么就转移到状态“10”。状态**empty**之前有一个**start**标记, 我们称**empty**状态为初态; 状态“10”多加了一个圆圈, 我们称他为终态。自动机的初态只有一个而终态可以由若干个。

# 确定性有限状态自动机

**DFA**是一个图结构的数据结构，每一个节点都有字符集字符数条有向边，并且之所以称之为确定性的，是由于任何一个节点，都不会存在标有相同字符的有向边指向不同的节点。为了更好的理解，我们再给出一个复杂一点的例子，终态为'**nano**'的自动机如下图所示，能够判断输入串里是否包含“**nano**”

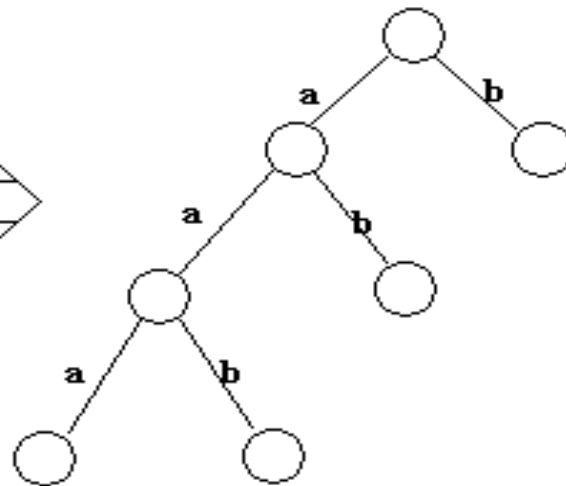
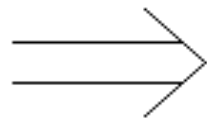


为了解决多串匹配问题，我们下面将介绍一种**DFA**，他是树结构的模型（一般图模型的**DFA**在应用中并不是很多）。

# 单词前缀树(trie)

4个模式串

b  
ab  
aab  
aaa



这个树有一个性质，那就是m个模式串中的前缀所组成的集合A与根节点到每一个树中的节点的路径上的字符组成的字符串S所组成的集合B，是一个满射的关系，即树中任一节点，都对应于某个模式串的前缀。



# 单词前缀树(trie)

将串s插入到trie的代码描述如下：

```
void build(string s)
{
    trienode* p=root;
    for (int i=0;i<s.size();++i)
    {
        if (p->child[s[i]-'a']==NULL)
            p->child[s[i] -'a'] = new trienode(); //初始化新的节点
        p=p->child[s[i] -'a'];
    }
}
```

```
struct trienode
{
    trienode * child[26] ;
    //假设所有字符就是26个字母
    trienode() {
        memset(child,0, sizeof(child));
    }
};
```

可以看出将n个模式串插入到一棵单词前缀树的时间复杂度为 $O(\sum \text{len}(i))$ ，其中 $\text{len}(i)$ 为第i个模式串的长度。



# Trie图

trie图是一种DFA，可以由trie树为基础构造出来，

对于插入的每个模式串，其插入过程中使用的最后一个节点都作为DFA的一个终止节点。

如果要求一个母串包含哪些模式串，以用母串作为DFA的输入，在DFA上行走，走到终止节点，就意味着匹配了相应的模式串(没能走到终止节点，并不意味着一定不包含模式串)。

# Trie图

模式串：

abcd

abc

abe

ae

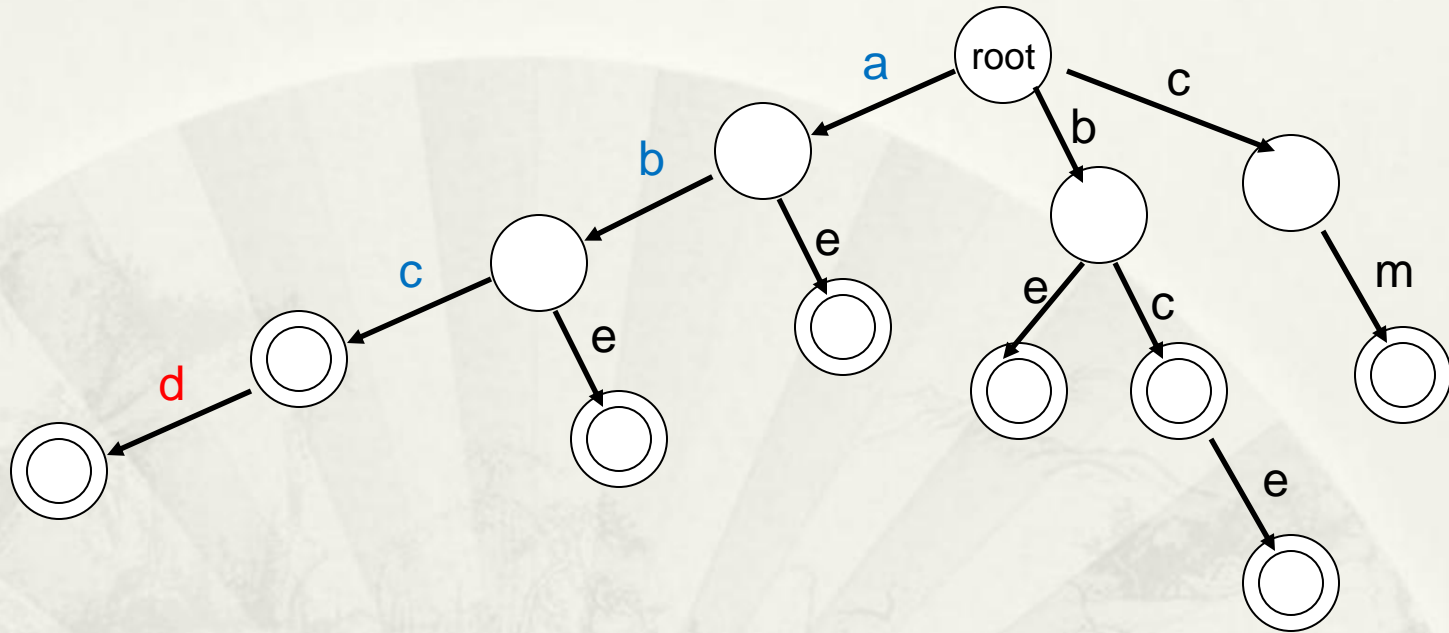
bc

be

bce

cm

母串：kc**abc**mgh




走不动了就回到root，从**b**开始继续在DFA上走。这样，母串指针就回溯了。如何避免母串指针回溯，即如何做到只要回到root,就直接从母串的下一个未访问过的字符开始继续走？

# Trie图

KMP算法是如何避免母串指针回溯的？

母串： aabcbdaa**k**g.....  
子串： aabcbdaa**f**



# Trie图

母串为什么要回溯：假设前 $n-1$ 个字符匹配，第 $n$ 个失配：



母串：  $a_1 a_2 a_3 \dots a_{n-1} a_n \dots$

子串：  $b_1 b_2 b_3 \dots b_{n-1} b_n \dots$

母串指针如果不回溯，直接用 $a_n$ 和 $b_1$ 比，可能就会忽略下面的情况：

母串：  $a_1 a_2 a_3 a_4 a_5 \dots a_{n-1} a_n \dots$

子串：  $b_1 b_2 \dots b_{i-1} b_i \dots b_n \dots$

$a_4 a_5 \dots a_{n-1} a_n$  和  $b_1 b_2 \dots b_{i-1} b_i$  匹配上

# Trie图

若发生了下述情况:

母串:  $a_1 a_2 a_3 a_4 a_5 \dots a_{n-1} a_n \dots$

子串:  $b_1 b_2 \dots b_{i-1} b_i \dots b_n \dots$

$a_4 a_5 \dots a_{n-1} a_n$  和  $b_1 b_2 \dots b_{i-1} b_i$  匹配

则  $b_1 b_2 \dots b_{i-1}$  是  $a_1 a_2 a_3 a_4 a_5 \dots a_{n-1}$  的后缀, 也是  $a_1 a_2 a_3 a_4 a_5 \dots a_{n-1}$  的前缀

类似情况可能有多种, 考察其中  $b_1$  位置最靠左的.  
则  $b_1 b_2 \dots b_{i-1}$  就是所有既是  $a_1 \dots a_{n-1}$  前缀又是  $a_1 \dots a_{n-1}$  后缀的串中最长的(不包括  $a_1 \dots a_{n-1}$  )。

# Trie图

发生了下述情况时：

母串：  $a_1 a_2 a_3 a_4 a_5 \dots a_{n-1} a_n \dots$

子串：  $b_1 b_2 \dots b_{i-1} b_i \dots b_n \dots$

$a_4 a_5 \dots a_{n-1} a_n$  和  $b_1 b_2 \dots b_{i-1} b_i$  匹配

接下来要比较的就是  $a_n$  和  $b_i$


那么，不如当初母串指针不要回溯，直接比较  $a_n$  和  $b_i$  -----前提是：

找到了一个既是  $a_1 \dots a_{n-1}$  前缀又是  $a_1 \dots a_{n-1}$  后缀的最长的串  $b_1 b_2 \dots b_{i-1}$ 。

# Trie图

KMP算法是如何避免母串指针回溯的？

母串： aabcbdaa**k**g.....  
子串：       aab**b**cbdaafc





# Trie图

KMP算法是如何避免母串指针回溯的？


母串： aabcbdaa**k**g.....  
子串：       a**a**bcdaafc



# Trie图

KMP算法是如何避免母串指针回溯的？


母串： aabcbdaa**k**g.....  
子串：       **a**abcbdaafc



# Trie图

KMP算法是如何避免母串指针回溯的？

母串： aab**c**daakg.....  
子串：        aab**c**daafc



# Trie图

如果有多个子串：

母串： abcde**k**g.....  
          ↓

子串1： abcde**g**c

子串2： abcdeuae

子串3： cden

子串4： dek

子串5： sdecse

# Trie图

如果有多个子串：

母串： abcdekkg.....  
          ↓

子串1： abcdegc

子串2： abcdeuae

子串3： cden

子串4： dek

子串5： sdecse

# Trie图

如果有多个子串：

母串： abcdekkg.....  
          ↓

子串1： abcdegc

子串2： abcdeuae

子串3： cden

子串4： dek

子串5： sdecse

# Trie 图

## 如果有多个子串：

母串:    abcde**kg**.....

子串 1: abcde**g**c

子串2: abcdeuae

子串3:      cde**n**

子串4: dek

## 子串5: sdecse

希望母串指针不回溯，  
要记住已经匹配了哪  
些子串的前缀。

在一个子串的失配位置，还应该和别的子串进行比较，是有可能匹配上的。



# Trie图

如果有多个子串：

母串：     abcdek<sup>↓</sup>g.....  
子串1：     a<sup>↓</sup>bcdegc  
子串2：     a<sup>↓</sup>bcdeuae  
子串3：     c<sup>↓</sup>den  
子串4：     s<sup>↓</sup>decse

如果都匹配不上，再从每个子串的开头开始匹配,相当于在trie图上回到root

在trie图上回到root就意味着现在还没有任何一个子串被匹配上任何一个字符（以前即便完整匹配了，也是以前的事情）

# Trie图

如果有多个子串：

母串：    abcdek<sup>↓</sup>g.....  
子串1：        abcdegc  
子串2：        abcdeuae  
子串3：        cden  
子串4：        sdecse

# Trie图

模式串：

abcd

abc

abe

ae

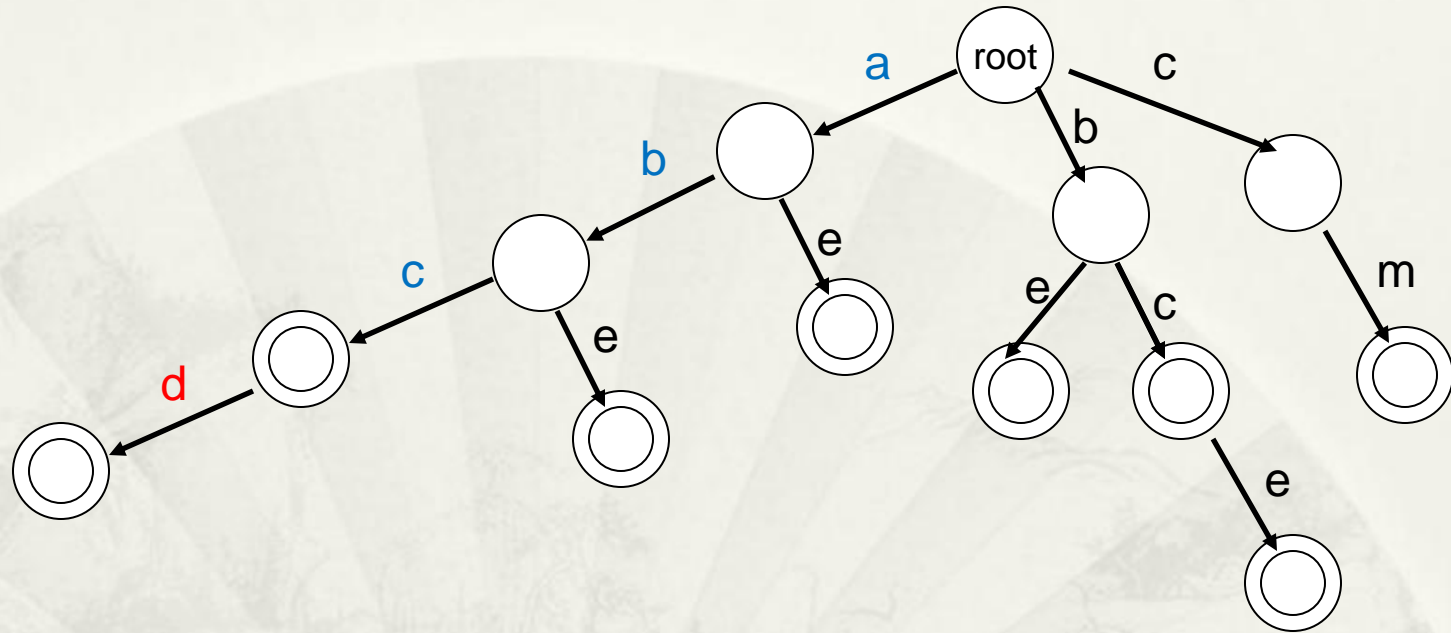
bc

be

bce

cm

母串：kc**abc**mgh



走不动了就回到root，从**b**开始继续在DFA上走。这样，母串指针就回溯了。如何避免母串指针回溯，即如何做到只要回到root,就直接从母串的下一个字符开始继续走？

**关键：要记住哪些模式串的哪个前缀已经被匹配**

# Trie图

模式串：

abcd

abc

abe

ae

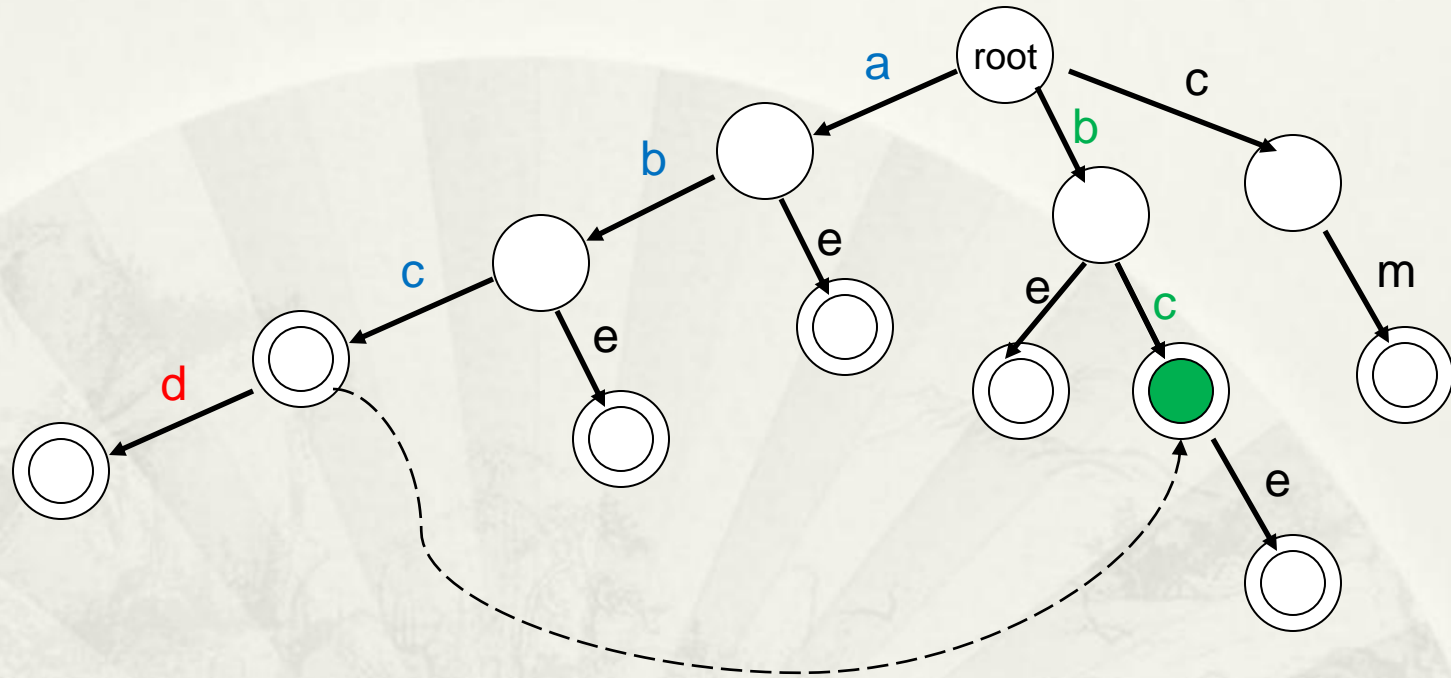
bc

be

bce

cm

母串：kc**abc**mgh



发现绿色节点对应的字符串是已经匹配的母串 **abc** 的后缀(**bc**一定是某个子串的前缀)，那么就应该让母串中 **abc** 的后一个字符继续和绿色节点出发的字符匹配，即试图匹配一个前缀为**bc**的子串。

# Trie图

模式串：

abcd

abc

abe

ae

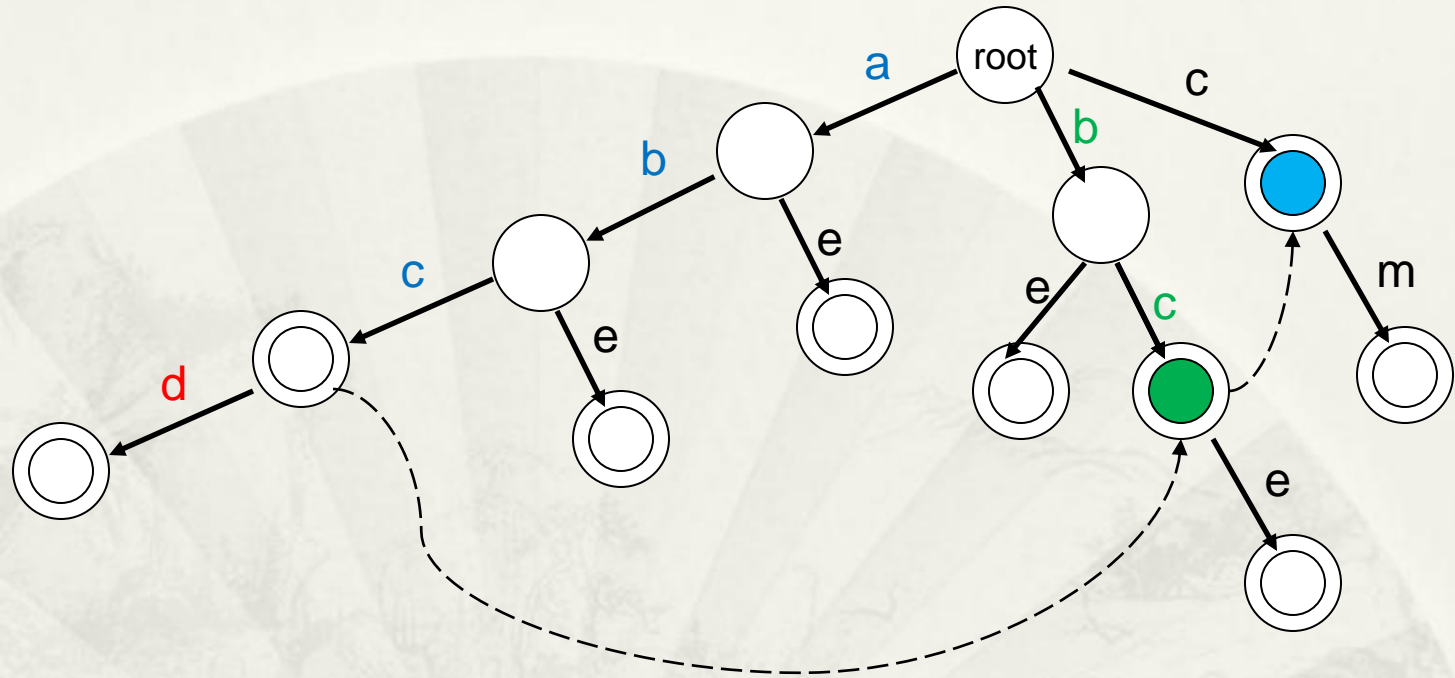
bc

be

bce

cm

母串：kc**abc**mgh



m和e失配。但是发现蓝色节点节点对应的字符串是已经匹配的母串 **abc** 的后缀**c**，那么就应该让母串中 **abc**的后一个字符继续和蓝色节点出发的字符匹配

# Trie图

模式串：

abcd

abc

abe

ae

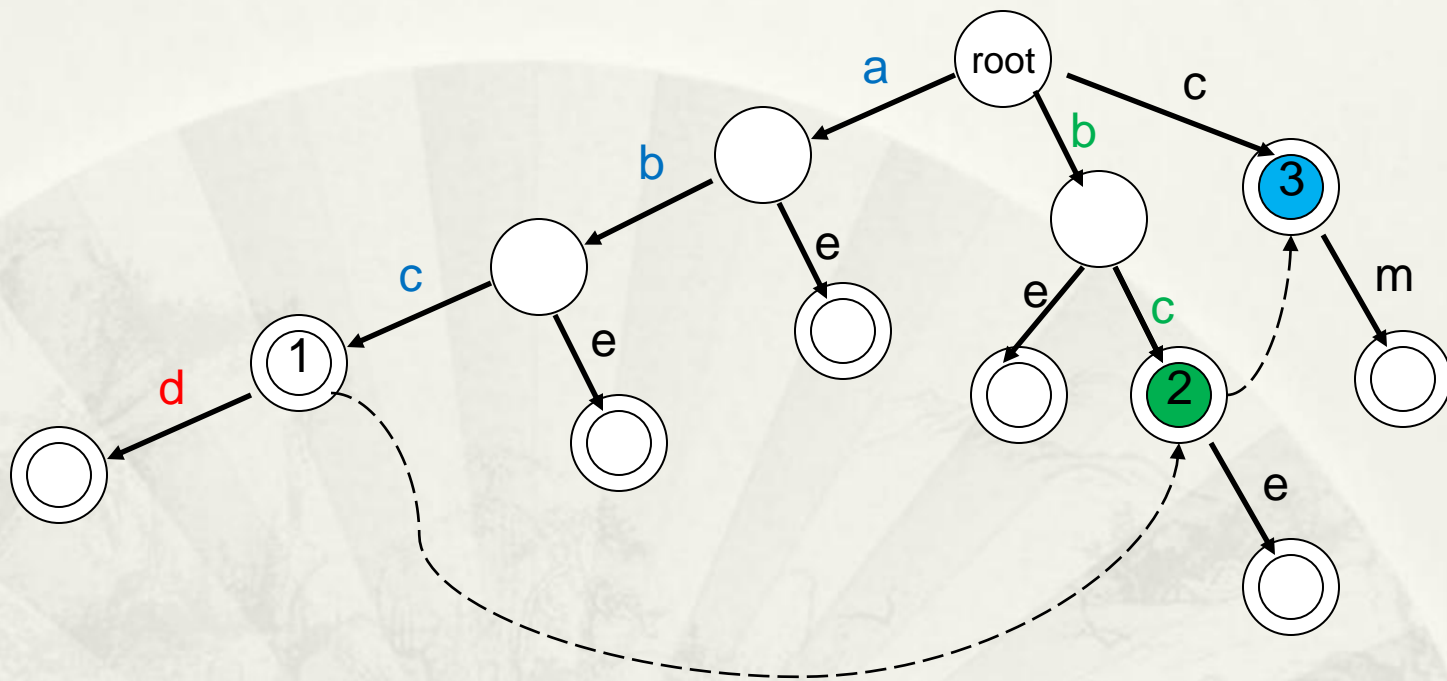
bc

be

bce

cm

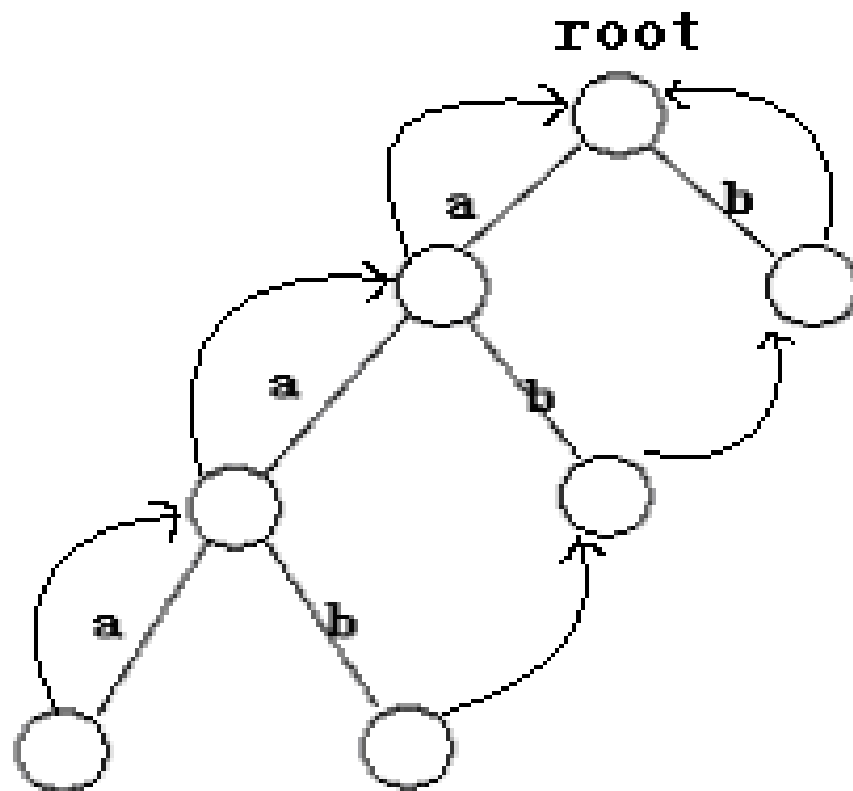
母串：kc**abc**mgh



总之，要想办法在Trie图上走时，可以从 1->2->3

# 前缀指针

仿照**KMP**算法的**Next**数组，我们也对树上的每一个节点建立一个前缀指针。这个前缀指针的定义和**KMP**算法中的**next**数组相类似，从根节点沿边到节点**p**我们可以得到一个字符串**S**，节点**p**的前缀指针定义为：指向树中出现过的**S**的最长的后缀(不能等于**S**)。





# 如何高效的构造前缀指针

步骤为：根据深度一一求出每一个节点的前缀指针。对于当前节点，设他的父节点与他的边上的字符为**Ch**，如果他的父节点的前缀指针所指向的节点的儿子中，有通过**Ch**字符指向的儿子，那么当前节点的前缀指针指向该儿子节点，否则通过当前节点的父节点的前缀指针所指向点的前缀指针，继续向上查找，直到到达根节点为止。

# 如何高效的构造前缀指针

接下来我们构造前缀指针！

ROOT1号节点的所有连出的

边都连向ROOT1号

节点是1号节点，连接字符为

A、B

为B查找父亲的前缀指针

号节点是是否通过A连接

的儿子。

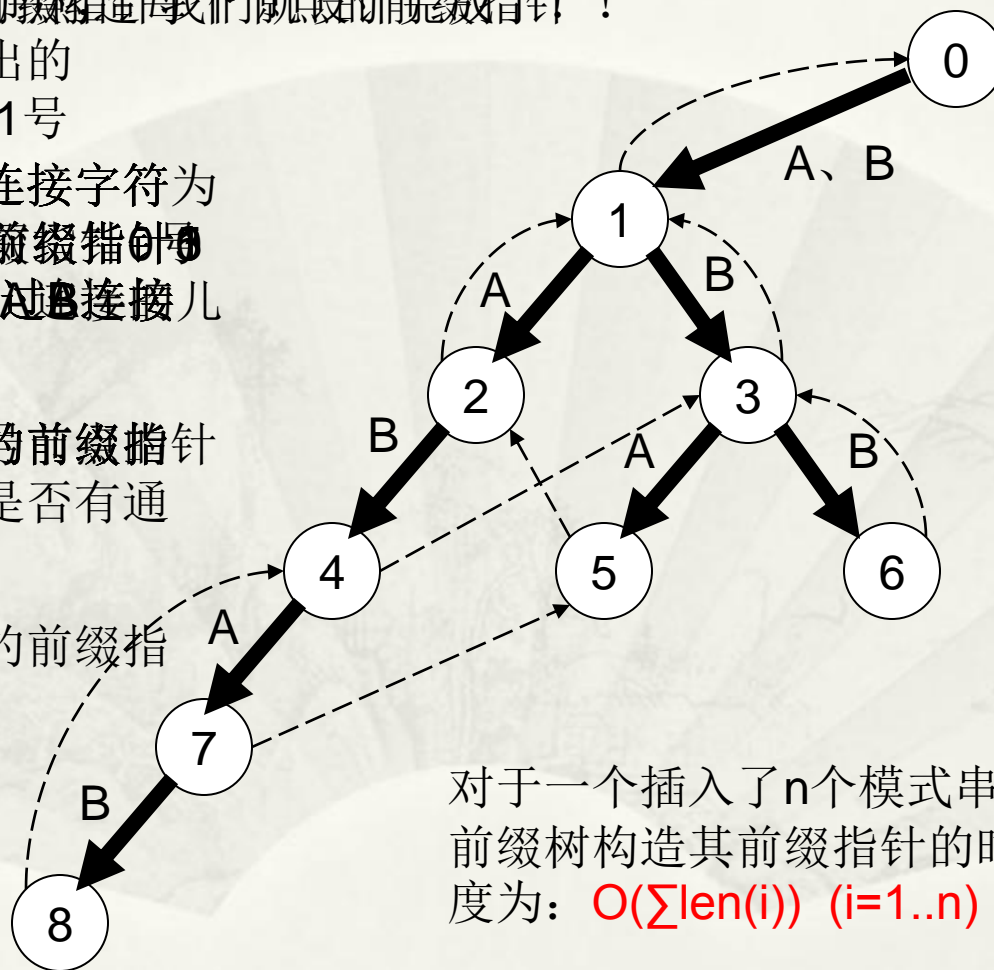
没有！于是2号节点的前缀指针

指向1号节点

通过B连接的儿子。

有！于是4号节点的前缀指针

指向2号节点



对于一个插入了n个模式串的单词  
前缀树构造其前缀指针的时间复杂度为： $O(\sum \text{len}(i))$  ( $i=1..n$ )

# 如何在建立好的Trie图上遍历

以上的单词前缀树+前缀指针就是确定性有限状态自动机的树形结构图(即trie图)的基本构造方式了。

接下来要解决的问题是，已知一个串S，如何利用这个串在当前已经建立好的DFA上进行遍历，看其是否包含某个模式串，以及其时间复杂度。

**危险节点**的概念：

- 1) 终止节点是危险节点
- 2) 如果一个节点的前缀指针指向危险节点，那么它也是危险节点。

# 如何在建立好的Trie图上遍历

遍历的方法如下：从ROOT出发，按照当前串的下一个字符ch来进行在树上的移动。若当前点P不存在通过ch连接的儿子，那么考虑P的前缀指针指向的节点Q，如果还无法找到通过ch连接的儿子节点，再考虑Q的前缀指针...直到找到通过ch连接的儿子，再继续遍历。如果遍历过程中经过了某个终止节点，则说明S包含该终止节点代表的模式串。

如果遍历过程中经过了某个非终止节点的危险节点，则可以断定S包含某个模式串。要找出是哪一个，沿着危险节点的前缀指针链走，碰到终止节点即可。

这样遍历一个串S的时间复杂度是 $O(\text{len}(S))$

为什么在trie图上遍历母串S的时间复杂度是 $O(\text{len}(S))$ ?

- 1) 母串每过掉一个字符，不论该字符是匹配上了还是没匹配上，在trie图上最多往下走一层(层的概念来自原trie树)。
- 2) 一个节点的前缀指针总是指向层次更高的节点，所以每沿着前缀指针走一步，节点的层次就会向上一层
- 3) 母串 S 最终被过掉了  $\text{len}(S)$  个字符，所以最多向下走了  $\text{len}(S)$  次。

最多向下走 $\text{len}(S)$ 次，那么就不可能向上走超过 $\text{len}(S)$ 次，因此沿前缀指针走的次数，最多不会超过 $\text{len}(S)$

# 回到原问题

下面我们回到原来的那个多串模式匹配问题。

有了**DFA**这个好工具之后，这道题目就可以很简单的解决了。具体步骤如下：

- 1.按照**M**个模式串构造出一个单词前缀树
- 2.将这个单词前缀树加上前缀指针
- 3.将原来 **$6N+6L-2$** 个原串在这个建立的**DFA**上进行遍历，将遍历到的终态的位置记录下来，便可以得到每个模式串在原串中出现的位置了。



# 回到原问题

现在来考虑一下，我们完成这个问题的算法的时间复杂度(M个模式串的总长度为LEN):

1. $O(LEN)$

2. $O(LEN)$

3. $O((6N+6L-2)*len(i))=O(NL)$

因此总的时间复杂度为 $O(LEN+NL)$

一个很不错的算法！！



# 最纯粹的Trie图题目

给N个模式串，每个不超过个字符，再给M个句子，句子长度<100 判断每个句子里是否包含模式串

$N < 10$ ,  $M < 10$ , 字符都是小写字母

5 8

abcde

defg

cdke

ab

f

abcdkef

abkef

bcd

bca

add

ab

qab

f

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
#define LETTERS 26
int nNodesCount = 0;
struct CNode
{
    CNode * pChlds[LETTERS];
    CNode * pPrev; //前缀指针
    bool bBadNode; //是否是危险节点
    void Init() {
        memset(pChlds,0,sizeof(pChlds));
        bBadNode = false;
        pPrev = NULL;
    }
};
CNode Tree[200]; //10个模式串，每个10个字符，每个字符一个节点，
也只要100个节点
```

```
void Insert( CNode * pRoot, char * s)
{ //将模式串s插入trie树
    for( int i = 0; s[i]; i ++ ) {
        if( pRoot->pChlds[s[i]-'a'] == NULL) {
            pRoot->pChlds[s[i]-'a'] =
                Tree + nNodesCount;
            nNodesCount ++;
        }
        pRoot = pRoot->pChlds[s[i]-'a'];
    }
    pRoot->bBadNode = true;
}
```

```
void BuildDfa( )  
{ //在trie树上加前缀指针  
    for( int i = 0; i < LETTERS ; i ++ )  
        Tree[0].pChilds[i] = Tree + 1;  
    Tree[0].pPrev = NULL;  
    Tree[1].pPrev = Tree;  
    deque<CNode * > q;  
    q.push_back(Tree+1);  
    while( ! q.empty() ){  
        CNode * pRoot = q.front();  
        q.pop_front();  
    }  
}
```

```

for( int i = 0; i < LETTERS ; i ++ ) {
    CNode * p = pRoot->pChlds[i];
    if( p) {
        CNode * pPrev = pRoot->pPrev;
        while( pPrev ) {
            if( pPrev->pChlds[i] ) {
                p->pPrev =
                    pPrev->pChlds[i];
                if( p->pPrev-> bBadNode)
                    p-> bBadNode = true;
                //自己的pPrev指向的节点是危险节点，则自己也是危险节点
                break;
            }
            else
                pPrev = pPrev->pPrev;
        }
        q.push_back(p);
    }
} //对应于while( ! q.empty() )

```

```
bool SearchDfa(char * s)
{ //返回值为true则说明包含模式串
```

```
    CNode * p = Tree + 1;
```

```
    for( int i = 0; s[i] ; i ++ ) {
```

```
        while(true) {
```

```
            if( p->pChlds[s[i]-'a']) {
```

```
                p = p->pChlds[s[i]-'a'];
```

```
                if( p->bBadNode)
```

```
                    return true;
```

```
                break;
```

```
            }
```

```
            else
```

```
                p = p->pPrev;
```

```
        }
```

```
    }
```

```
    return false;
```

```
}
```

```
int main()
{
    nNodesCount = 2;
    int M,N;
    scanf("%d%d",&N,&M);           //N个模式串， M个句子
    for( int i = 0;i < N; i ++ ) {
        char s[20];
        scanf("%s",s);
        Insert(Tree + 1,s);
    }
    BuildDfa();
    for( int i = 0 ;i < M;i ++ ) {
        char s[200];
        scanf("%s",s);
        cout << SearchDfa(s) << endl;
    }
    return 0;
}
```

## 2010 福州赛区题目

### Computer Virus on Planet Pandora

---

有 $n$ 个各不相同的长度不超过1,000的模式串（ $0 < n \leq 250$ ），和一个长度不超过5,100,000的母串。如果母串包含某个模式串或其反转（“ABCD”的反转是“DCBA”），则称母串被模式串感染。问母串一共被多少个模式串感染。



## 2010 福州赛区题目

### Computer Virus on Planet Pandora

---

1) 给模式串编上号，**Trie**图上每个终止节点记下它所对应的模式串的编号。为每个模式串设置一个是否已经被匹配的标记。走到某终止节点，就能知道哪个模式串被匹配，就设置其标记。

2) 要注意的是，本题中，模式串里有可能一个串**A**是另一个模式串**B**的子串，在这种情况下，用母串在**Trie**图上跑一遍，就可能会发生走到**B**串的终止节点，即作出**B**被匹配的结论，但是却忽略了匹配**B**的过程中**A**其实也已经能够匹配这个情况。

## POJ3987 2010 福州赛区题目

### Computer Virus on Planet Pandora

#### 2) 的解决办法:

- 1> 对Trie图上的每个节点设置一个“是否计算过”的标记，初始值全部为假。
- 2> 假定在原Trie树上的节点x所对应的字符串为s，那么x节点“已计算过”，就意味着s的所有模式子串，都已经被标记为“已匹配”。
- 3> 在用母串在Trie图上跑的过程中，每到达一个“未计算过”的危险节点，就将该节点标记为“已计算过”，然后就要将该节点对应的s的所有后缀模式串(既是s的后缀，又是模式串的字符串)都标记为“已匹配”(s的非后缀模式子串在扫描到s之前就已经被标记为“已匹配”了)。

## POJ3987 2010 福州赛区题目

### Computer Virus on Planet Pandora


---

4> 标记过程: 沿着节点的后缀指针一直走到根节点, 将路径上(包括起点)的每个模式串的终止节点所对应的模式串, 都标记为“已匹配”。

这条路上的每个节点在Trie树上对应的字符串, 都是s的后缀, 而且所有既是s的后缀, 且在Trie树上有对应节点的字符串, 其对应节点也一定会出现在这条路径上。

## 2012 浙江金华邀请赛 Problem I. Hrinity

给定2500个模式串，和长度为5,000,000的母串，问母串中包含多少个模式串。如果模式串s1是模式串s2的子串，而且s2被母串包含，则s1应被忽略。



模式串：

b

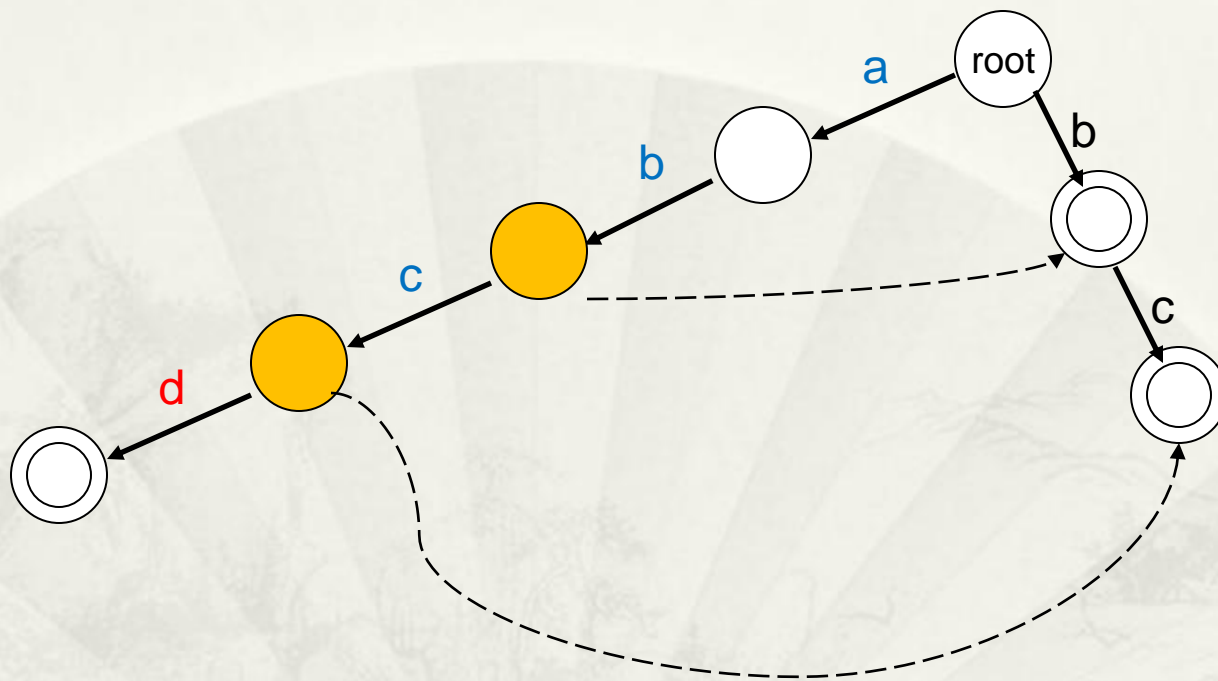
bc

abcd

母串

abc

abcd



模式串:

b

e

abcd

abcdef

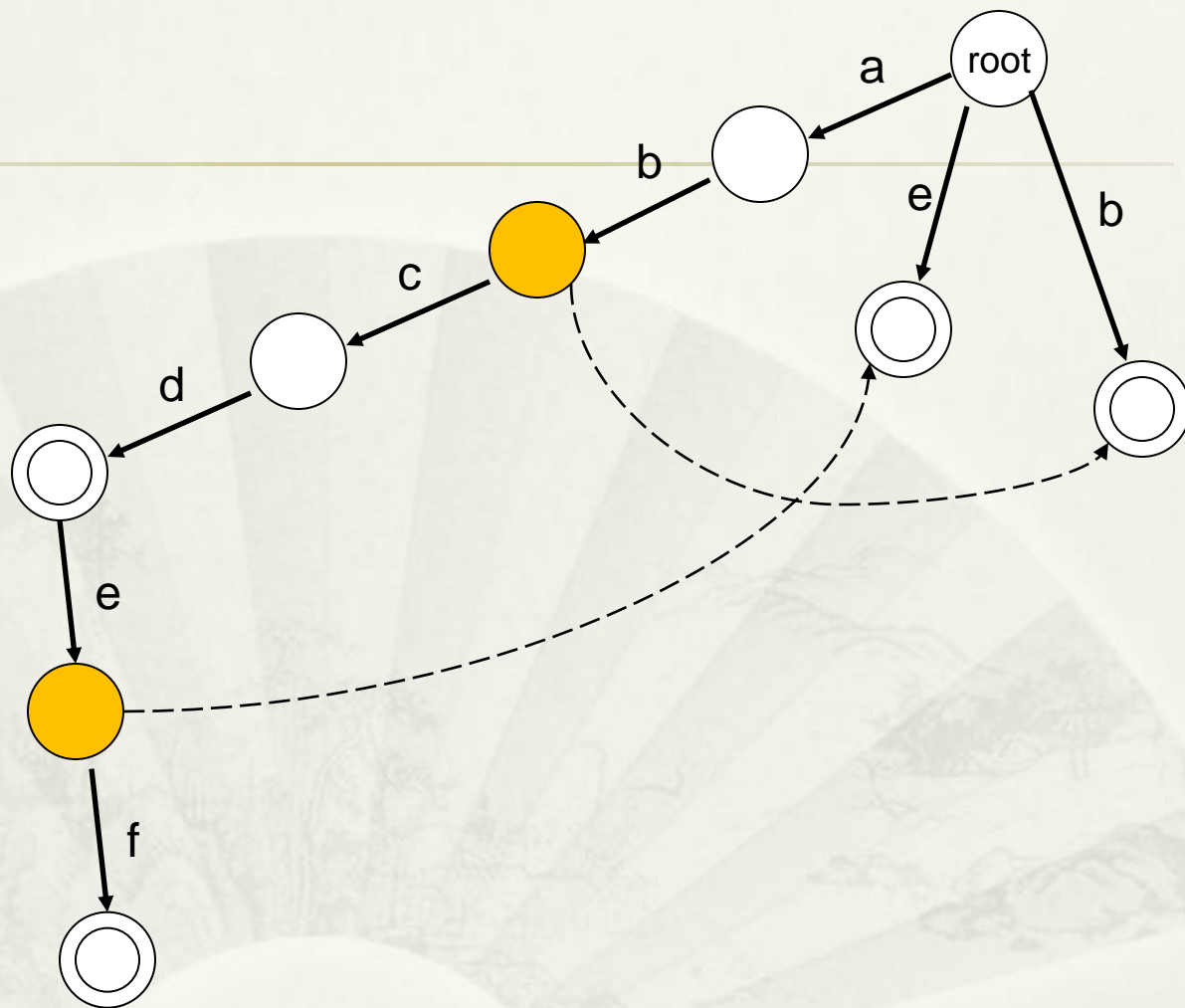
母串

abc

abcd

abce

abcdef



## 2012 浙江金华邀请赛 Problem I. Hrinity

对终止节点定义两种操作：全忽略和半忽略

若 $a$ 是终止节点，则全忽略 $a$ 表示：

- 1) 若 $a$ 的父节点为根，则全忽略就是忽略 $a$
- 2) 其他情况，全忽略 $a$ 即为：

忽略 $a$ ，并且全忽略 $a$ 的前缀指针链上的全部终止节点

对 $a$ 到根的路径上的所有危险节点 $k_i$ ，沿着 $k_i$ 的前缀指针链全忽略所有终止节点（若 $k_i$ 是终止节点，则全忽略 $k_i$ 即可）

半忽略 $a$ ：

在全忽略操作中,去掉“忽略 $a$ ”，剩下的操作就是半忽略。



解法：

用母串 **S** 在 **trie** 图上遍历，走到未被忽略的终止节点 **x**，则将 **x** 标记为已经匹配，并且半忽略 **x**。

走到危险但非终止的节点 **y**，则沿着 **y** 的前缀指针链找到第一个终止节点 **x**，将 **x** 标记为已经匹配，并且半忽略 **x**。

统计匹配且未被忽略的模式串数目

设置标记，处理过的危险节点和终止节点就不用再处理。



## POI #7 题：病毒

已知若干个01串，他们是病毒的特征代码。如果一段程序（由01组成）不含有任何病毒特征代码，则称它为安全程序。请判断是否存在无限长的安全程序。

“无限长”的安全母串，在Trie图上，从根节点出发，可以永远不停地走下去，即走无限步，都不会路过“不安全节点”，而Trie图节点是有限的……

模式串对应的终止节点是“不安全”节点，如果有一个节点，它的前缀指针指向的节点是“不安全”节点，那么它也是“不安全节点”。

“无限长”的安全母串，在Trie图上，从根节点出发，可以永远不停地走下去，即走无限步，都不会路过“不安全节点”，而Trie图节点是有限的  
.....

只需判断，Trie图上，由安全节点构成的子图上，有没有环即可，有，就能无限走下去

事先要对trie图做额外处理，即

```
if( pNode->pChildds[i] == NULL)
    pNode->pChildds[i] = pNode->pPrev->pChildds[i];
```

然后Dfs求环的时候，只考虑字母边

## Trie图上进行动归：POJ 3691 DNA repair

给定不超过50个由 'A', 'G', 'C', 'T'. 四个字母组成的模式串，每个模式串长度不超过20，再给一个不超过1000个字符长的同样由上述字母组成的母串S，问在S中至少要修改多少个字符，才能使其不包含任何模式串

## 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

# 试试DFA吧

对于这样的多字符串模式匹配问题，我们想到了DFA，那就试试吧。

首先按照给出的P个模式串构造一棵DFA出来。这时候，我们发现DFA给我们创建了一个很好的动态规划的平台。迅速给出状态：

$Ans[i][j]$ 表示若要用长度为i的母串的前缀遍历DFA树，使之达到节点j，至少要修改多少个字符。j必须不是“危险”节点。

初始情况：

$Ans[0][1] = 0$  //1是DFA的初始节点

其他所有 $Ans[i][j]$  为无穷大

问题的答案就是

$\text{Min}\{ \text{Ans}[\text{len}][j] \mid j \text{ 是DFA的安全节点} \}$

len是母串的长度

# 试试DFA吧

递推公式为：

由Ans[i][j] 可以推出：

$$\text{Ans}[i+1][\text{son}[j]] = \min \{ \text{Ans}[i+1][\text{son}[j]],$$
$$\text{Ans}[i][j] + ( \text{Char}(j,\text{son}[j]) \neq \text{str}[i] ) \}$$

( Char(j,son[j]) != str[i] ) 表达式值为0或1

Char(j,son[j])表示从节点j走到节点son[j]所经过的字母

str是母串,下标从0开始算



# 试试DFA吧

$\text{Char}(j, \text{son}[j])$  表示从  $j$  到  $\text{son}[j]$  经过的字母，假设是  $a$ 。

这里所说的“经过”，不仅指从  $j$  通过一条字母  $a$  边直接到达  $\text{son}[j]$ ，也可以是通过若干前缀指针后再通过一个字母  $a$  边到达  $\text{son}[j]$

所以  $\text{son}[j]$  并不一定是  $j$  的子节点。

## POJ1625 Censored!

题目大意：

给出一个字符集V和P个模式串(长度小于10)，问由这个字符集中字符组成的长度为N的且不包含任意一个模式串的字符串有多少个？(字符集大小,  $N \leq 50$ ,  $P \leq 10$ )

样例解释：

样例：

aaa

输入：

aab

2 3 1 //N=3 P=1

aba

ab //字符集为ab

baa

bb

bab

以上5个

输出：

5

# 解题思路

这种类型的题目，如果使用搜索算法必然会导致超时的出现。因此，动态规划似乎是一种可行的方案，我们来尝试构造一个状态。

$Ans[i][j]$ 表示长度为 $i$ 且能走到节点 $j$ 的字符串个数（节点 $j$ 是安全节点）

初始 $Ans[0][1] = 1$ , 其他 $Ans[i][j] = 0$

由 $Ans[i][j]$  可以导出，每个由 $j$ 可以到达的安全节点 $son[j]$ ,都应执行：

$Ans[i+1][son[j]] += Ans[i][j]$

因为从根走 $i$ 步到达节点 $j$ 有  $n$ 种走法，那么走 $i+1$ 步到达 $son[j]$ 的走法就要加 $n$  (因为 $j$ 以外节点也可能通过一步走到 $son[j]$ )

这里所说的“到达”，不仅指从j(或从其他节点) 通过一条字母边直接到达son[j], 也可以是通过若干前缀指针后再通过一个字母边到达son[j]

(son[j]并不一定是j的子节点)

整个问题最终的答案就是：

$\sum \{ Ans[N][j] \mid j \text{ 是安全节点} \}$

此题需要高精度计算

# 总结

---

像例题的题目还有很多，类似例题2的题目还有：

[POJ2778 DNA Sequence](#)