GZOI

后缀自动机
Suffix Automaton

后缀自动机

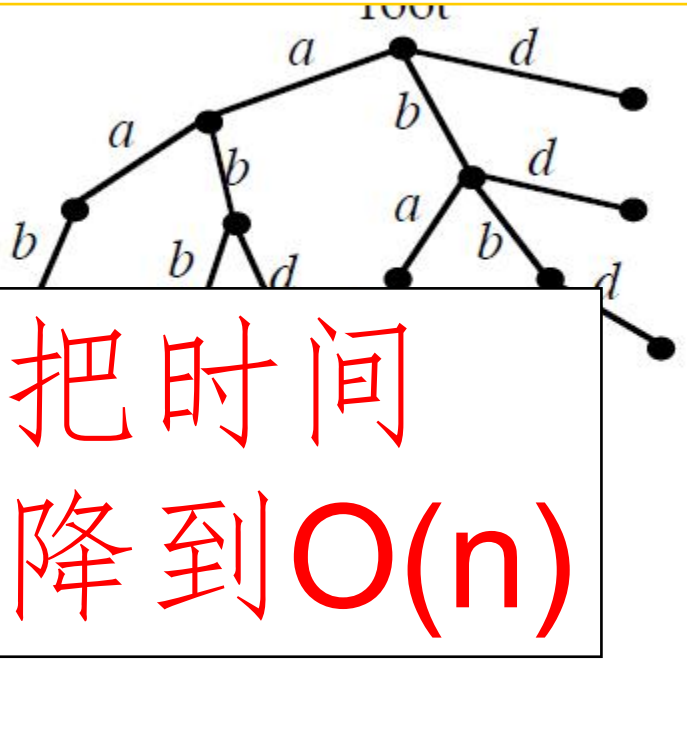
- 给定字符串S
- S的后缀自动机suffix automaton(SAM)是一个能够识别S的所有后缀的自动机。
- 即SAM(x) = True，当且仅当x是S的后缀
- 同时后面可以看出，后缀自动机也能用来识别S所有的子串。

简单实现——trie

- 把所有的后缀都放进一个trie里面，比如串

状态数量太多了！怎么破？

长度为n的串，会有n^2的节点



我们目的是把时间和空间复杂度降到O(n)

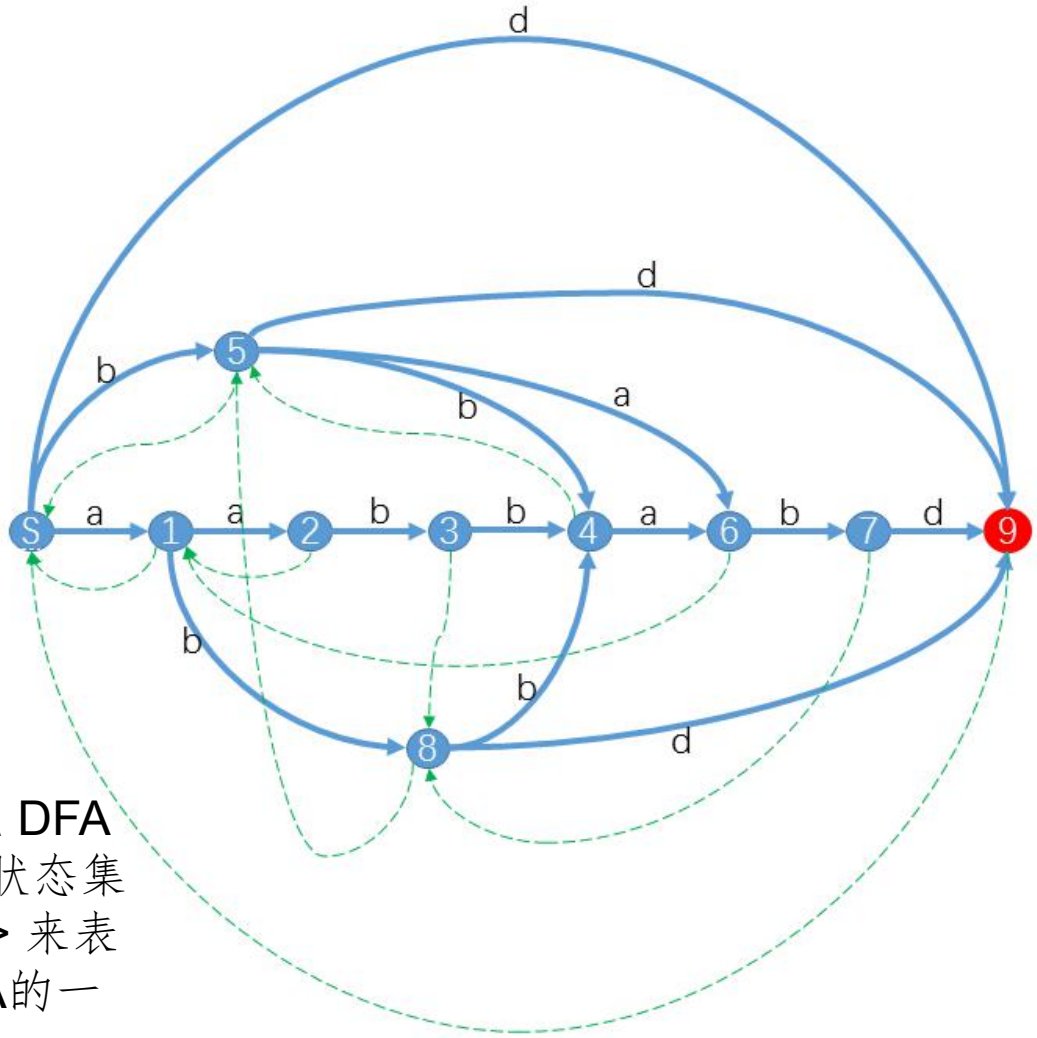
aabbabd

其中 红色状态 是终结状态。你可以发现对于S的后缀，我们都可以从S出发沿着字符标示的路径(蓝色实线)转移，最终到达终结状态。

特别的，对于S的子串，最终会到达一个合法状态。而对于其他不是S子串的字符串，最终会“无路可走”。

SAM 本质上是一个有限状态集自动机 DFA，DFA 可以用一个五元组 <字符集、状态集、转移函数、起始状态、终结状态集> 来表示。至于那些 绿色虚线 虽然不是DFA的一部分，却是SAM的重要部分。

其中比较重要的是 **状态集** 和 **转移函数**。



其中一个状态指的是从起点开始到这个点的所有路径组成的子串的集合。

例如上图中状态 4为 {bb,abb,aabb} 。

SAM状态集——endpos

- 子串的结束位置集合 endpos
  对于 S 的一个子串 t，endpos(t)表示子串t在 S 中所有出现时的结束位置集合。
- 如：S=aabbabd
  endpos(ab)={3,6}
  因为 ab 一共出现了2 次，结束位置分别是 3和 6 。
  同理 endpos(a)={1,2,5}
  endpos(abba)={5}

S=aabbabd

- 把 S 的所有子串的endpos 都求出来。
- 如果两个子串的 endpos相等，就把这两个子串归为一类,称为等价类

  如： endpos(aabb)={4}

  endpos(abb)={4}

  endpos(bb)={4}

- 最终这些 endpos的等价类就构成的 SAM 的状态集合。

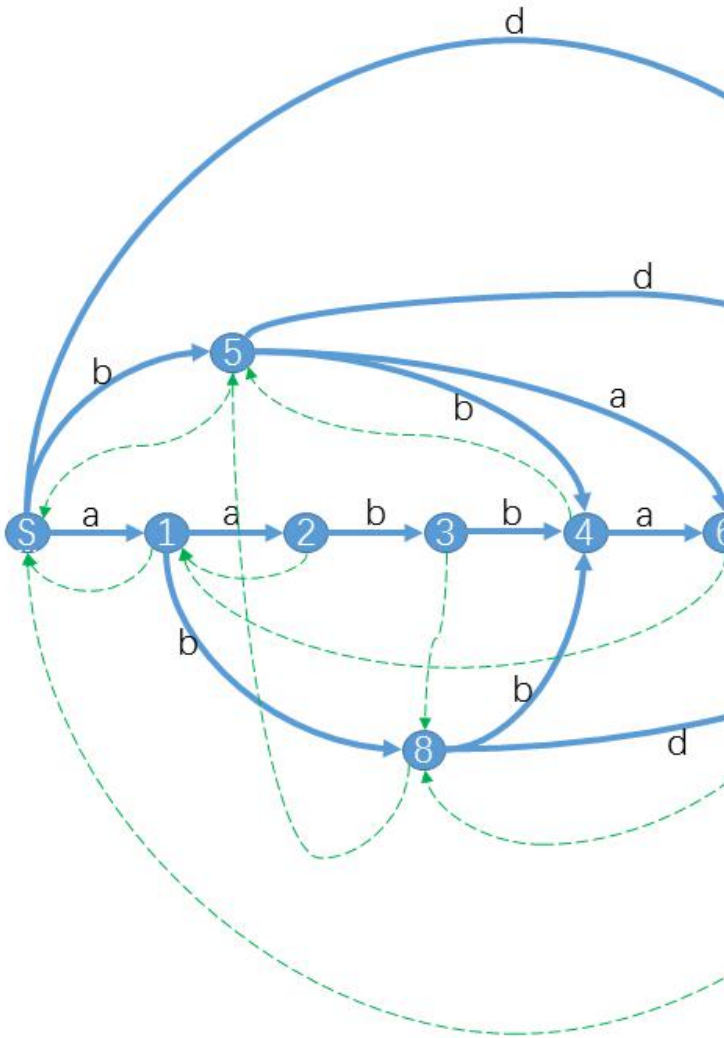
aabbabd的SAM状态集合

状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2,5}
2	aa	{2}
3	aab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4,6}
6	aabba,abba,bba,ba	{5}
7	aabbab,abbab,bbab,bab	{6}
8	ab	{3,6}
9	aabbabd,abbabd,bbabd,babd,abd,bd,d	{7}

- 而这个endpos集合与SAM的状态集合刚好对应

aabbabd的endpos集

状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2,5}
2	aa	{2}
3	aab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4,6}
6	aabba,abba,bba,ba	{5}
7	aabbab,abbab,bbab,bab	{6}
8	ab	{3,6}
9	aabbabd,abbabd,bbabd,babd,abd,bd,d	{7}



- 而这个endpos集合与SAM的状态集合刚好对应

性质一

- 令 s_1, s_2 为 S 的两个子串，不妨设 $|s_1| \leq |s_2|$。
- 则s_1是s_2的后缀当且仅当

$$endpos(s_1) \supseteq endpos(s_2)$$

- s_1不是 s_2的后缀当且仅当

$$endpos(s_1) \cap endpos(s_2) = \varnothing$$

对于两个集合有A⊇B，表示B中的元素都在A中出现过

证明

首先证明s_1是s_2的后缀 $\Rightarrow$ $endpos(s_1) \supseteq endpos(s_2)$

因为每次出现 s_2 时候，s_1 一定会伴随出现。

然后证明 $endpos(s_1) \supseteq endpos(s_2) \Rightarrow$ s_1 是 s_2 的后缀

。显然 $endpos(s_2) \neq \varnothing$ ，那么意味着每次 s_2 结束的时候 s_1 也会结束，且 $|s_1| \leq |s_2|$ ，显然成立。

所以这两个互为充要条件。那么 s_1 不是 s_2 的后缀当且仅当 $endpos(s_1) \cap endpos(s_2) = \varnothing$ 就是其中的推论了，后者是前者的必要条件。

性质二

- SAM中的一个状态中包含的子串都具有相同的 endpos，长度短的子串一定是较长的子串的后缀。

其中一个状态指的是从起点开始到这个点的所有路径组成的子串的集合。

例如上图中状态 4为 {bb,abb,aabb} 。

性质三

- 定义substrings(st)表示状态st中包含的所有子串集合，longest(st)表示st包含的最长的子串，shortest(st)表示st包含的最短的子串
- 那么：
  对于一个状态 st，只要任意串s∈substrings(st)，都有 s 是 longest(st) 的后缀

如状态 7，substring(7)={aabbab,abbab,bbab,bab}， longest(7)=aabbab，shortest(7)=bab

例一　后缀自动机——基本概念

- 输入　　hihocoder 1441

  第一行包含一个字符串S，S长度不超过50。

  第二行包含一个整数N，表示询问的数目。(1 <= N <= 10)

  以下N行每行包括一个S的子串s，s不为空串。

- 输出

  对于每一个询问s，求出包含s的状态st，输出一行依次包含shortest(st)、longest(st)和endpos(st)。其中endpos(st)由小到大输出，之间用一个空格分割。

样例输入
```
aabbabd
5
b
abbab
aa
aabbab
bb
```

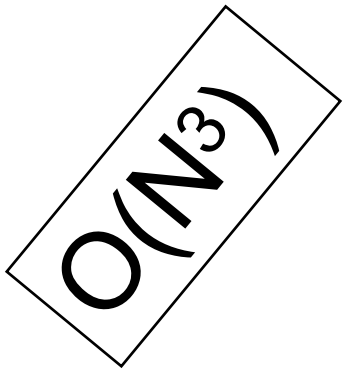
样例输出
```
b b 3 4 6
bab aabbab 6
aa aa 2
bab aabbab 6
bb aabb 4
```

http://hihocoder.com/problemset/problem/1441

法一

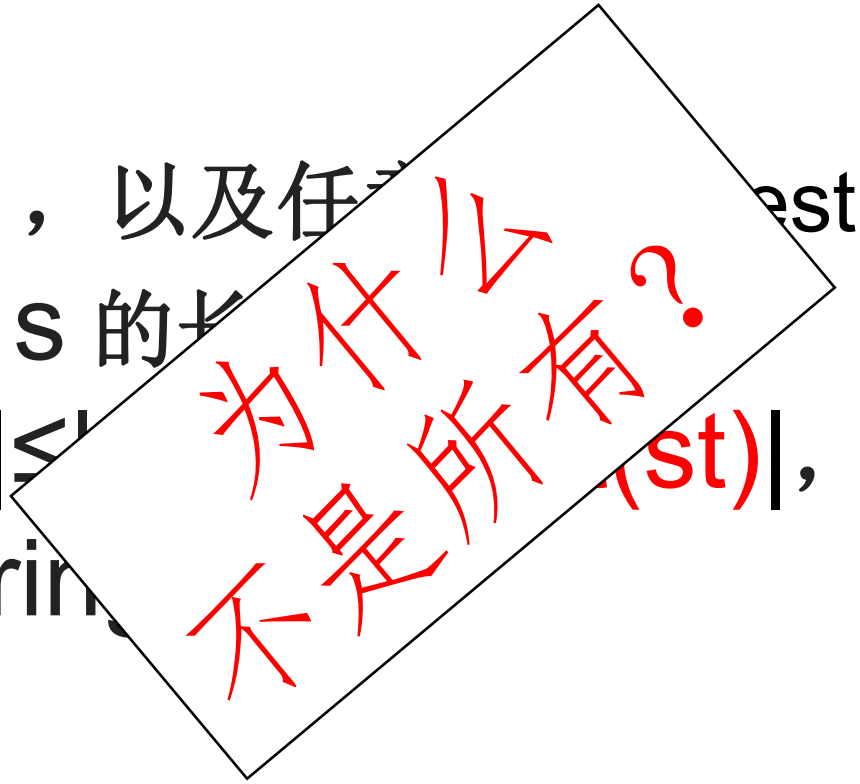
- 暴力枚举每一个子串
- 暴力查找每一个子串出现的位置，用二进制形式记录子串的结尾字符出现的位置，作为状态st
- 计算状态st的同时，更新longset(st)和short(st)并记录每一个子串对应的st

```
int main(){
    string s;    int n, m;
    map<long long , string> shortest, longest;
    map<string , long long> dict;
    cin>>s>>n;
    m = s.length();
    for (int i = 0; i <m ; ++i) {
        for (int j = 1; j <= m-i ; ++j) {
            string x = s.substr(i, j);          //枚举所有子串
            long long st = 0;
            for (int k = 0; k <= m-j ; ++k)
                    if( x == s.substr(k,j) )  st  |= (1LL<<(k+j));    //跟子串x相同的，记录进st (二进制记录)
            if( shortest[st].length()==0 || j< shortest[st].length() ) shortest[st] = x;
            if( j > longest[st].length() ) longest[st] = x;
             dict[x] = st;        //记录子串所在的st
        }
    }
    for (int l = 0; l <n ; ++l)    {
        string x;        cin>>x;
        long long st = dict[x];
        cout<<shortest[st]<<" "<<longest[st];
        for (int i = 0; i < 50 ; ++i)       {
            if( (1LL<<i)&st )
                cout<<" "<<i;
        }
        cout<<endl;
    }
    return 0;
}
```



性质四

- 对于一个状态 st，以及任…est(st)的后缀 s，如果 s 的长…|shortest(st)|≤…(st)|，那么 s∈substrin…



所以 substrings(st) 包含的是longest(st)的一系列 连续 后缀。

对于元素A和集合B，有A∈B，读作A属于B，表示A是集合B中的元素

状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2,5}
2	aa	{2}
3	aab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4,6}
6	aabba,abba,bba,ba	{5}
7	aabbab,abbab,bbab,bab	{6}
8	ab	{3,6}
9	aabbabd,abbabd,bbabd,babd,abd,bd,d	{7}

SAM的后缀链接

- substrings(st) 包含的是longest(st) 的一系列 连续 后缀。这连续的后缀，并不是所有的后缀，它会在某个地方会“断掉”。

- 如状态 7 ，包含的子串依次是 aab,abbab,bbab,bab。按照连续的规律下一个子串应该是 ab，但是ab没在状态 7里。
- 为什么呢？

- aabbab,abbab,bbab,bab的 endpos 都是{6}，下一个后缀 ab 当然也在结束位置 6，但是 ab还在结束位置 3 出现过，所以ab出现次数更多，于是就被分配到一个新的状态中了。

- 当 longest(st) 的某个后缀 s 在新的位置出现时，就会“断掉”，s 会属于新的状态

- 接上面ab属于状态8

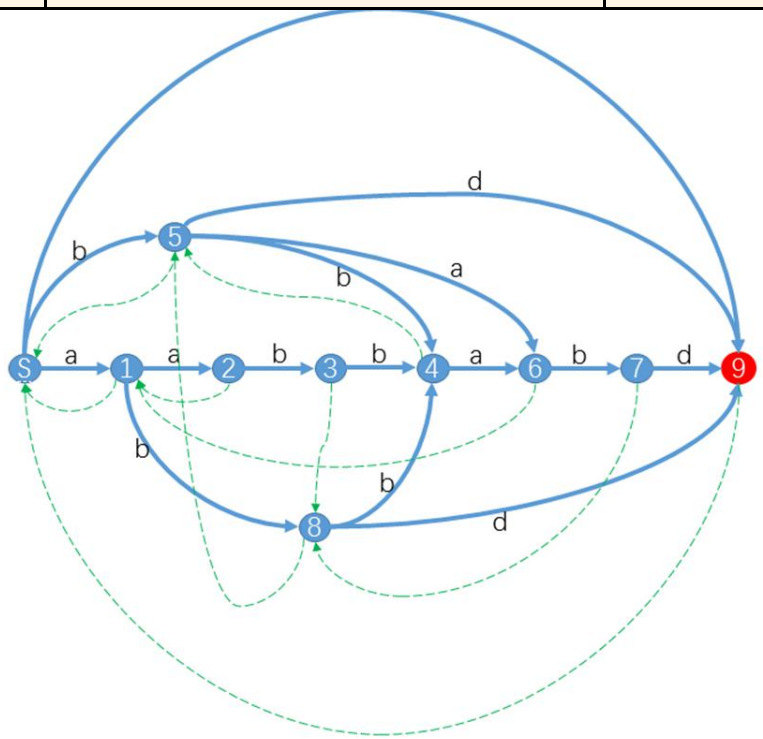
  endpos(ab)={3,6}

ab的下一个后缀b，也有类似情况：b在状态4出现过，endpos(b)={3,4,6}

b属于状态5。接下来处理b的后缀，会遇到空串，endpos()={0,1,2,3,4,5,6}

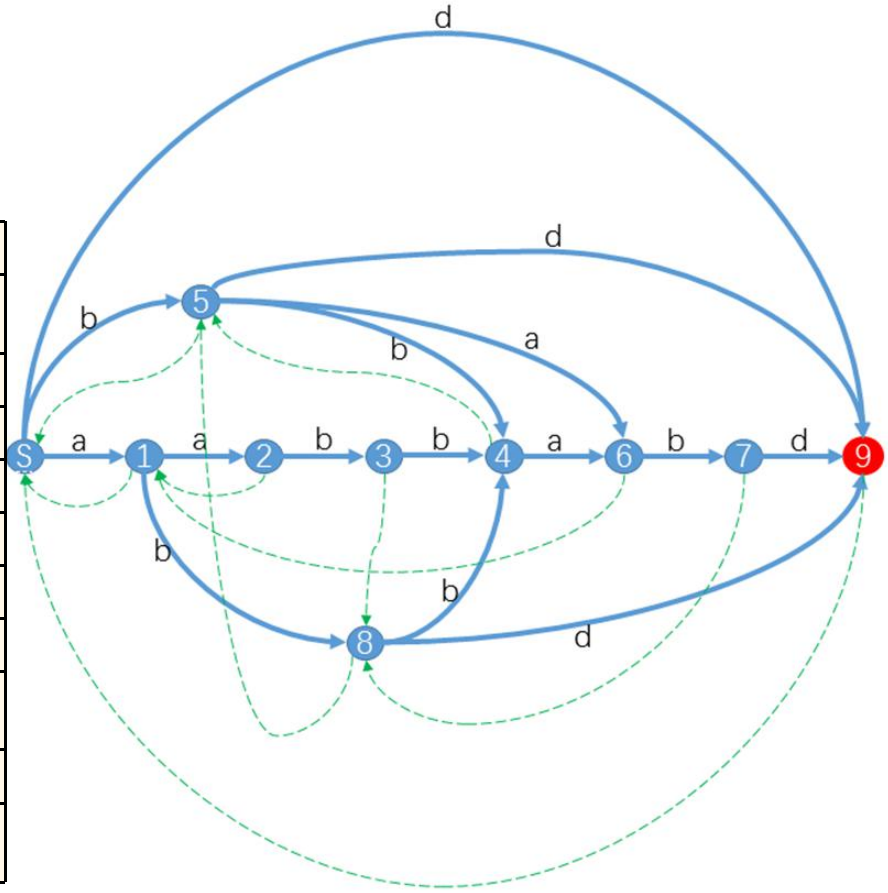
这个就是状态S

状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2,5}
2	aa	{2}
3	aab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4,6}
6	aabba,abba,bba,ba	{5}
7	aabbab,abbab,bbab,bab	{6}
8	ab	{3,6}
9	aabbabd,abbabd,bbabd,babd,abd,bd,d	{7}



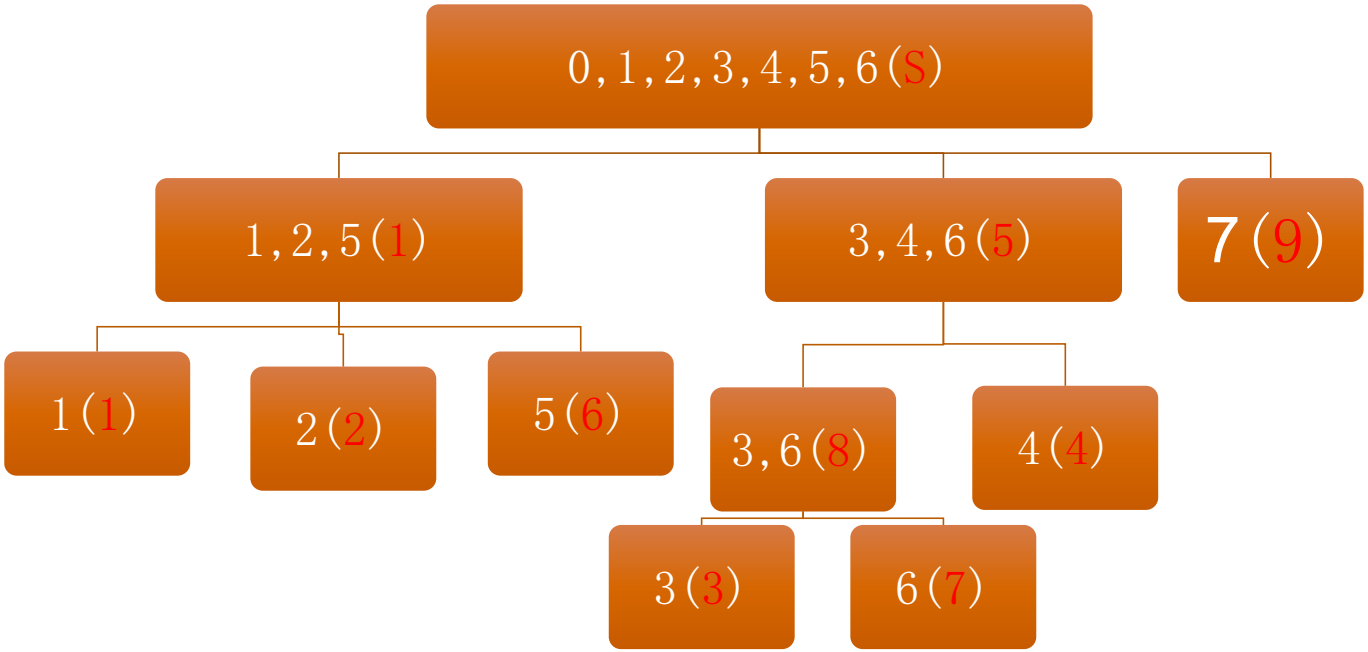
SAM的后缀链接

状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2,5}
2	aa	{2}
3	aab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4,6}
6	aabba,abba,bba,ba	{5}
7	aabbab,abbab,bbab,bab	{6}
8	ab	{3,6}
9	aabbabd,abbabd,bbabd,babd,abd,bd,d	{7}



- 表示longest(7)即aabbab的后缀依次在状态7,8,5,S中。
- 我们称这条状态序列为后缀链接SuffixLink
- 这条link就是图中绿色虚线
- 根据link链接，反向后，我们可以构成一颗endpos树

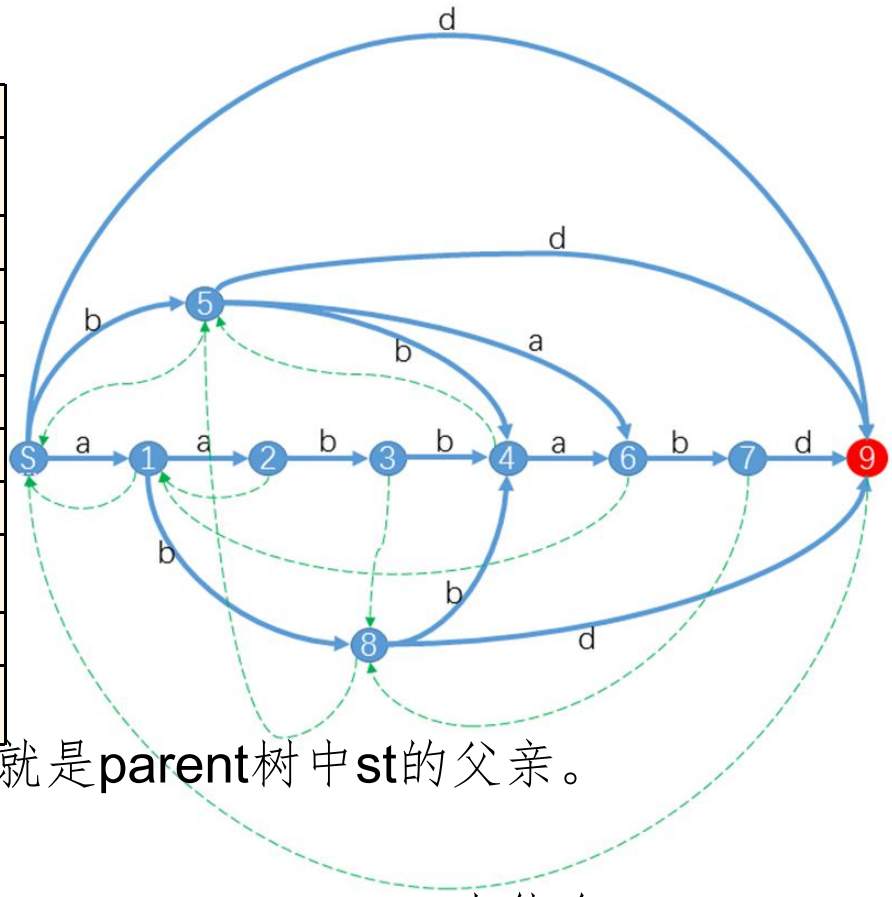
endpos集合树



状态	
S	空串
1	a
2	aa
3	aab
4	aabb,ab
5	b
6	aabba,a
7	aabbab,
8	ab
9	aabbabc

括号里的为节点，括号外的为endpos集合
从图中可以看出，endpos集合实际上构成了一个树形结构，不妨称其为Parent树。
在这个树中，叶子节点个数只有n个，同时每个内部节点至少有2个孩子，容易证明树的大小必然是O(n);
最多节点情况是h形，为2n-1个节点
而每一个节点也就是SAM的一个状态，所以最多只有2n-1个状态

状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2,5}
2	aa	{2}
3	aab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4,6}
6	aabba,abba,bba,ba	{5}
7	aabbab,abbab,bbab,bab	{6}
8	ab	{3,6}
9	aabbabd,abbabd,bbabd,babd,abd,bd,d	{7}



- 设某一状态st,令状态fa=link(st)，也就是parent树中st的父亲。
- 那么有endpos(fa) ⊇endpos(st)
- 考虑长度，st的子串长度范围是[minlen(st),maxlen(st)],为什么minlen(st)-1不符合st要求？
  随着长度的变小，出现的地方越来越多，那么minlen(st)-1就必然属于fa范围。
- maxlen(fa)=minlen(st)-1

SAM转移函数

- 定义next(st)表示状态st遇到的下一字符集合

  $next(st)=\{S[i+1]|i\in endpos(st)\}$

- 如：
- S=aabbabd

next(S)={S[1],S[2],S[3],S[4],S[5],S[6],S[7]}

={a,b,d}

next(8)={S[4],S[7]}={b,d}

SAM转移函数

- 对于一个状态st,它的substrings(st) 中的所有子串后面接上一个字符c (c∈next(st)),得到新的子串都会同属于一个状态st'。

- 如:
  状态 4 ， next(4)={a} ， aabb,abb,bb,后面接上字符 a得到 aabba,abba,bba， 这些子串都属于状态 6 。

SAM转移函数

- 对于一个状态 st和一个字符 c∈next(st)，可以定义转移函数:

  $$trans(st,c)=\{z|s\in substrings(z)\}$$

  $$s=longest(st)+c$$

- 在longest(st)后面接上一个字符 c 组成一个新的子串s，找到包含子串 s 的状态 z (若不存在，则新建这个状态z)，那么 trans(st,c)就等于z。
- 当然，这里不一定要longest(st)，其它属于st的子串也是一样的。

算法流程

- 对于状态st我们保存以下数据：

maxlen[st]
minlen[st]
trans[st][1..c]
link[st]

- 使用增量法构造 SAM。
- 从初始状态开始，每次考虑添加一个字符 S[1],S[2],...,S[N]，依次构造可以识别 S[1],S[1..2],S[1..3],… ,S[1..N]=S的 SAM

- 每添加一个字符会增加多少个后缀？
- 假设已经构造好S[1..i]的SAM，现在要添加字符S[i+1],那么新增了i+1个S[1..i+1]的后缀要识别：
  S[1..i+1],S[2..i+1],...S[i..i+1],S[i+1]
- 状态呢？
- 这些新增状态分别是从S[1..i],S[2..i],S[3..i],… ,S[i],_ (空串)通过添加字符 S[i+1] 转移过来
- 即通过S[1..i]的所有后缀所在的状态转移的

- 而S[1..i]的所有后缀对应的状态们恰好就是其后缀链接ink连接起来路径上的所有状态

假设 S[1..i] 对应的状态是 u，等价于 S[1..i]∈substrings(u)。根据上面的讨论我们知道 S[1..i],S[2..i],S[3..i],...,S[i],_ (空串)对应的状态们恰好就是从 u 到初始状态 S 的由后缀链接link 连接起来路径上的所有状态，不妨称这条路径(上所有状态集合)是 suffix−path(u→S)

也就是说，对于 S[1..i]的后缀对于其他 s要么存在于 u 这个状态中，要么存在于前面的后缀链接link连接的状态中。

- 而对于S[2..i+1],...S[i..i+1],S[i+1] 这些新增加的后缀可能在以前出现过，即可能被SAM识别，能识别则进行状态转移即可，不能识别，则需要新建状态。

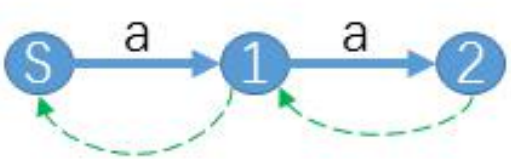
- S[1..i+1]这个子串肯定是不能被以前SAM识别的，所以必定要新添加一个状态z

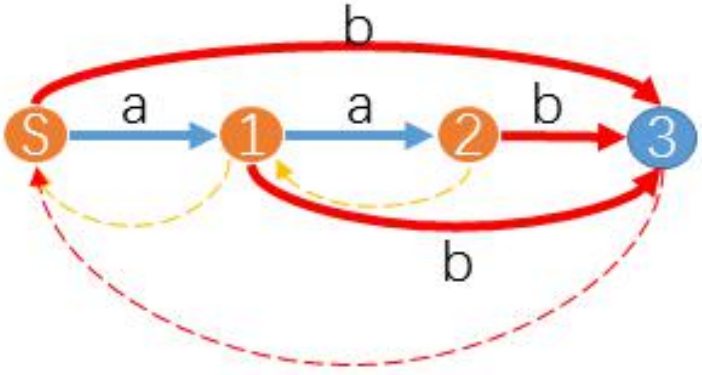
构造SAM

- 情况一
- 对于 suffix-path(u→S) 的任一状态 v，都有

  trans[v][S[i+1]]=NULL
- 这时我们只要令 trans[v][S[i+1]]=z，并且 link[st]=S

- 例如我们已经得到了 aa的 SAM，现在希望构造 aab的 SAM 。就如下图所示：





此时 u=2,新建z,z=3， suffix−path(u→S) 是 桔色状态 组成的路径 2−1−S 。并且这 3 个状态都没有对应字符 b 的转移。所以我们只要添加红色转移 trans[2][b]=trans[1][b]=trans[S][b]=z即可。当然link[3]=S 。

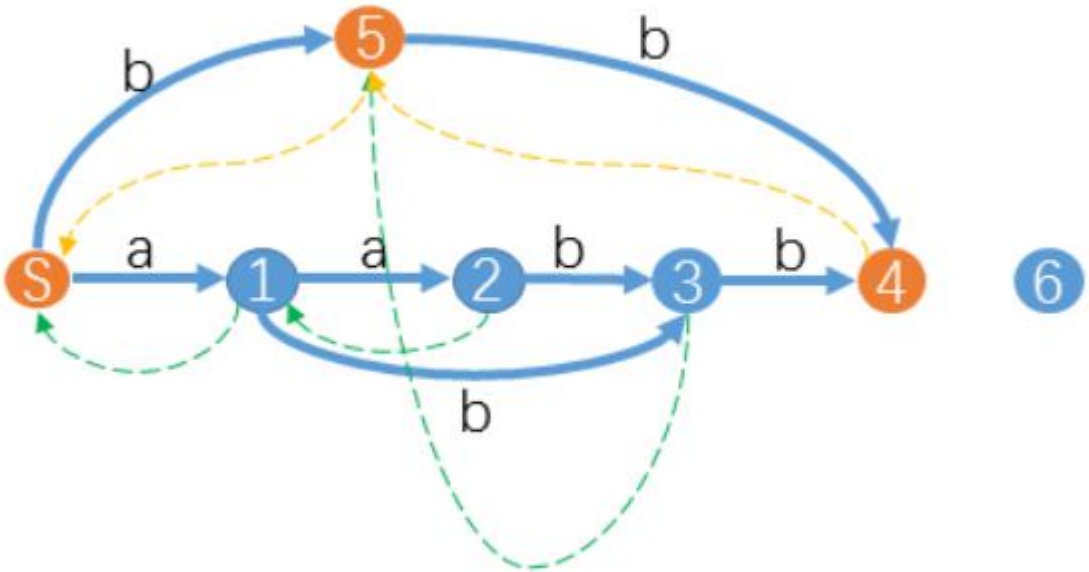
情况二

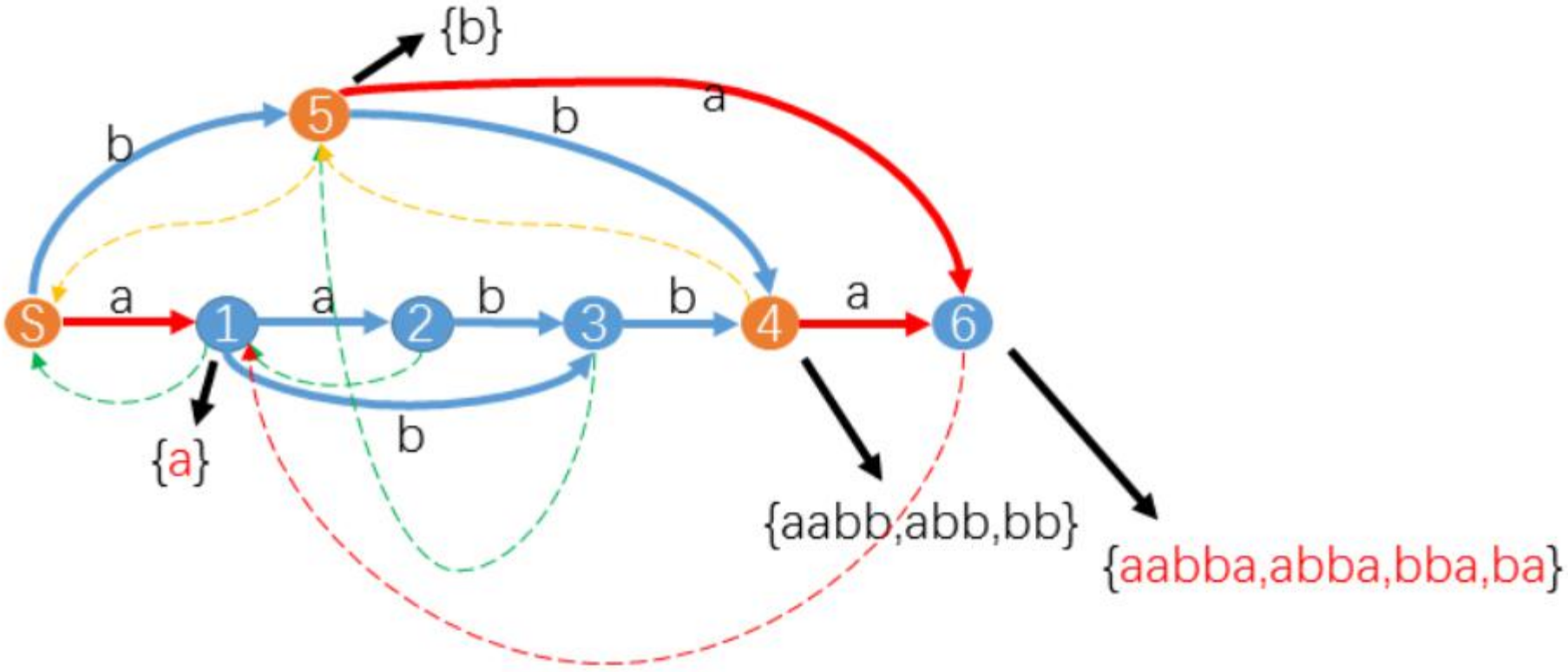
- suffix-path(u→S) 上有一个节点 v，使得

  trans[v][S[i+1]]≠NULL

2.1

如：假设已经构造aabb的SAM，现在要增加一个字符a构造aabba的SAM





- 这时 u=4，新建z=6，suffix−path(u→S) 是 桔色状态 组成的路径 4−5−S。对于状态 4 和状态 5，由于它们都没有对应字符 a 的转移，所以我们只要添加红色转移 trans[4][a]=trans[5][a]=z=6即可。
- 但此时 trans[S][a]=1已经存在了，此时link[z]=1即可，即红色虚线

2.1

- 在 suffix-path(u→S) 遇到的状态 v，此时 trans[v][S[i+1]]=x，如果状态x中包含的最长子串就是 v 中包含的最长子串接上字符 S[i+1]，即 longest(x)!=lonest(v)+c

  则有maxlen(v)+ 1 =maxlen(x)

- 只要使 link[z]=x 即可。

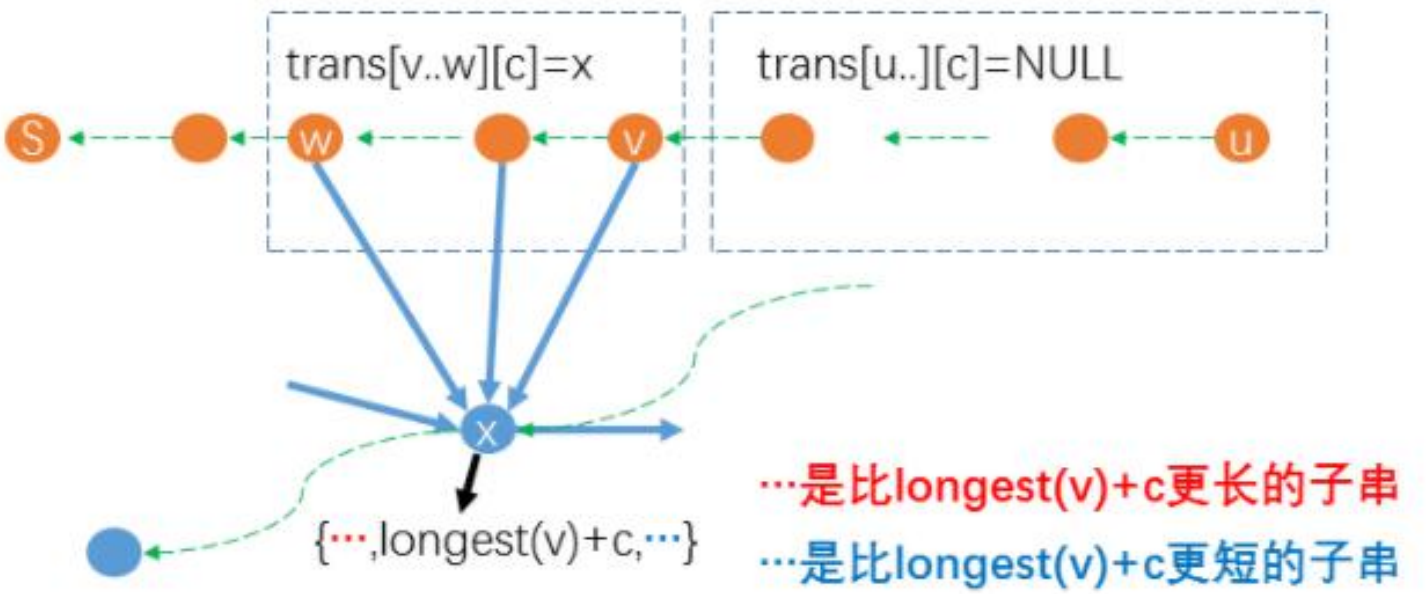
如在上面的例子里，v=S,x=1，longest(v) 是空串，longest(x)=a就是 longest(v)+a

我们将状态 6 link到状态 1就行了。因为此时 z 只缺少了这个 suffix-path(x→S) 的状态

2.2

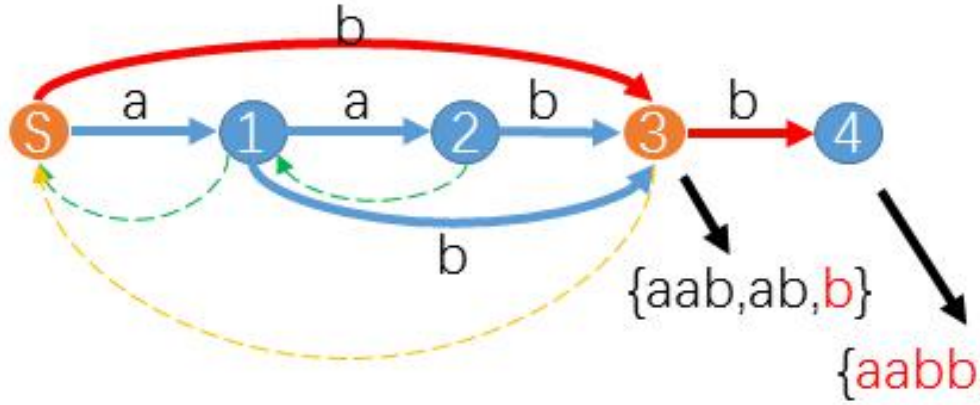
- 如果x 中包含的最长子串不是 v中包含的最长子串接上字符 S[i+1]，即 longest(x)!=lonest(v)+c

同时：maxlen(x)> maxlen(v)+ 1



在 suffix-path(u→S) 这条路径上，从 u 开始有一部分连续的状态满足 trans[u..][c]=NULL，对于这部分状态只需增加 trans[u..][c]=z 。紧接着有一部分连续的状态 v..w 满足 trans[v..w][c]=x ，并且 longest(v)+c 不等于 longest(x) 。

- 假设已经构造 aab的SAM如图，现在我们要增加一个字符bb 构造 aabb的 SAM。

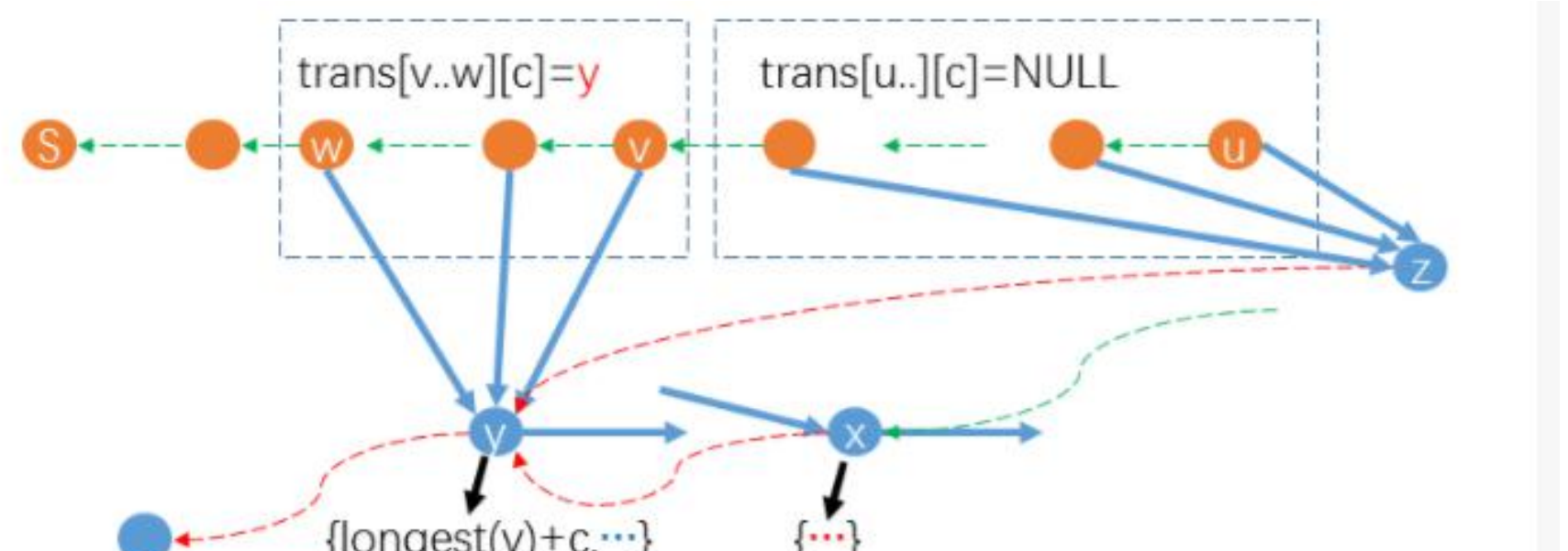


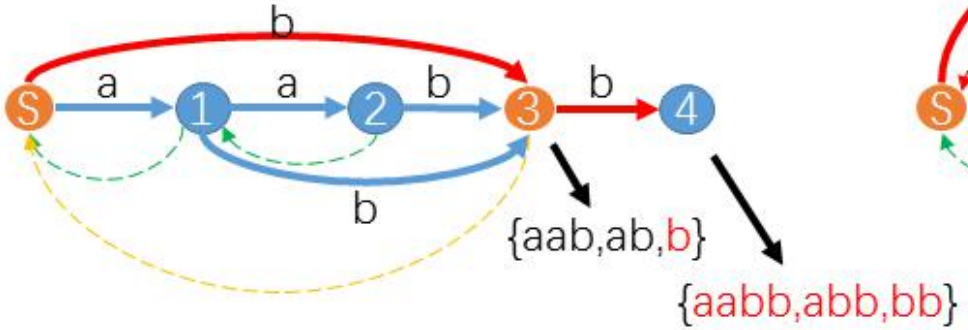
状态	子串	endpos
S	空串	{0,1,2,3}
1	a	{1,2}
2	aa	{2}
3	aab,ab,b	{3}

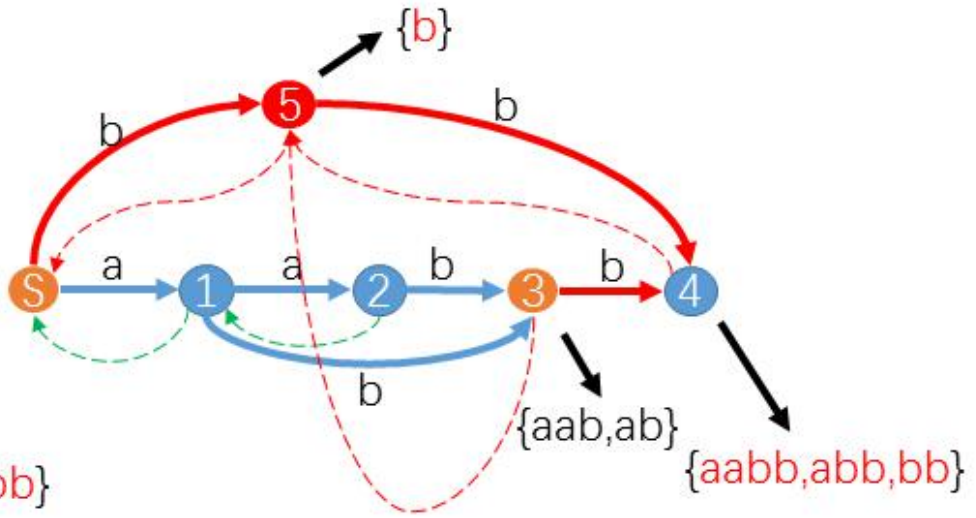
首先，新建状态4，在 suffix−path(u→S) 即3->S的状态 3 时，
trans[3][b]=null,所以trans[3][b]=z

然后处理状态 S 时，遇到 trans[S][b]=3
但longest(3)=aab ，longest(S)+b=b,两者不相等
即意味着增加了新字符后 endpos(aab) 不等于 endpos(b)

- 这时需要从 x 拆分出新的状态 y，并且把原来 x 中长度小于等于 maxlen(v)+1 的子串分给 y，其余子串留给 x。同时令：
- memcpy(trans[y],trans[x]), link[y]=link[x];
- trans[v..w][c]=y, link[x]=link[z]=y;
- 即y先复制x的转移，同时继承x的link，并且x前面断开的 substrings 都转移到 y 中，最后link[x]=link[z]=y。





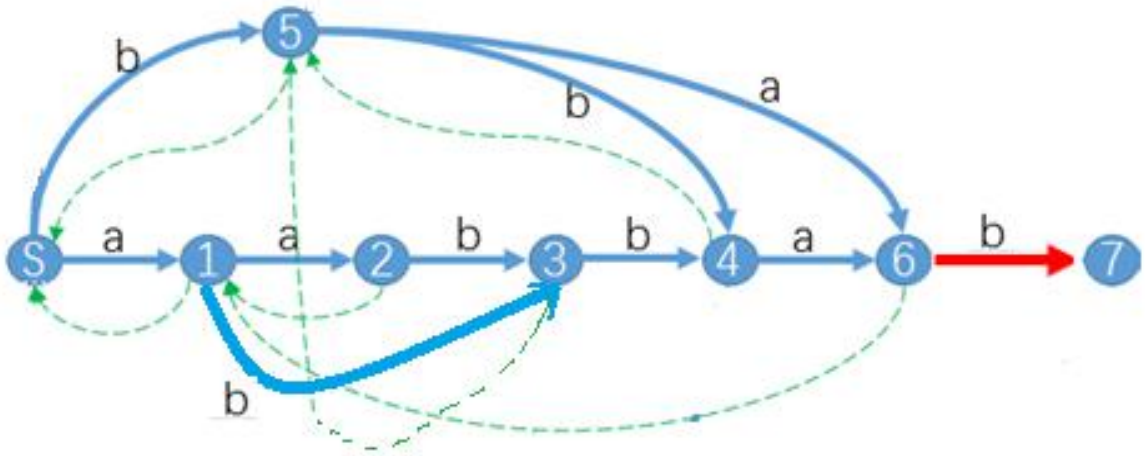


然后处理状态 S 时，遇到 trans[S][b]=3
但 longest(3)=aab，longest(S)+b=b,两者不相等
即意味着增加了新字符后 endpos(aab) 不等于 endpos(b)，
所以这两个子串不能同属一个状态3 。

这时就要从 3 中新拆分出一个新状态 5，把b及其后缀分给5，其余的子串留给3 。
同时令 trans[S][c]=5, link[5]=link[3]=S, link[3]=link[6]=5

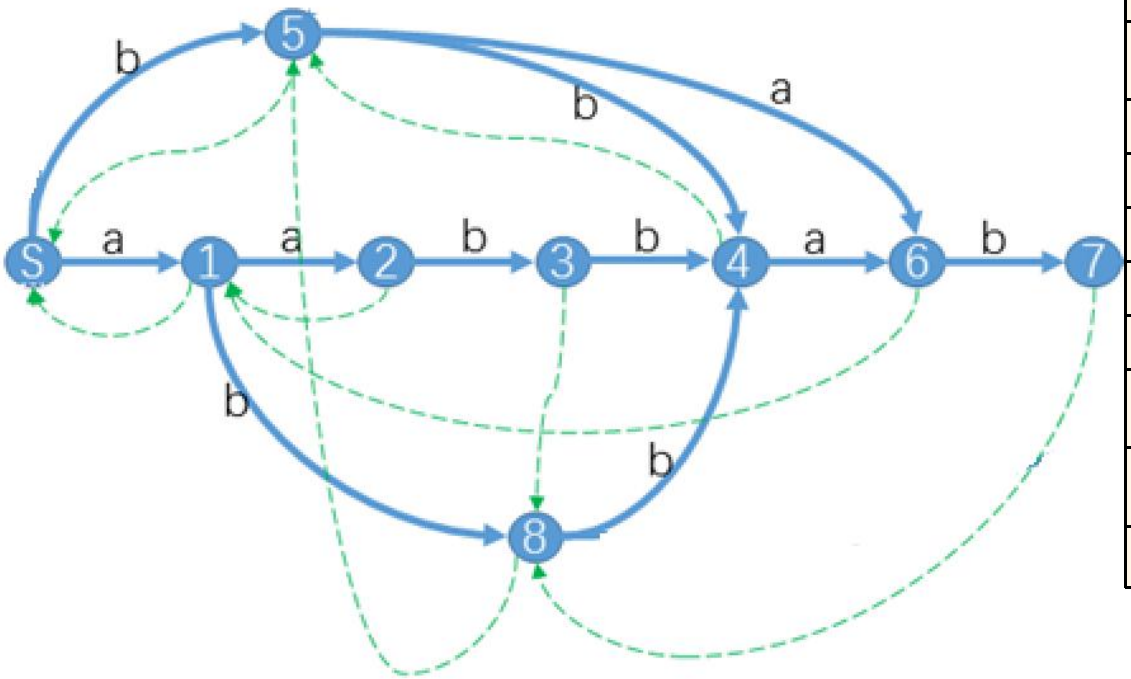
状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2}
2	aa	{2}
3	aab,ab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4}

aabb b在3和4的位置都出现了，所以不再只属于状态3。



状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2,5}
2	aa	{2}
3	aab,ab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4}
6	aabba,abba,bba,ba	{5}
7	aabbab,abbab,bbab,bab	
8	ab	

- 又如，若已经构成了aabba的SAM，现在要加一字符b，构造aabbab的SAM
- 在 suffix-path(u→S) 即6-1-S上的状态 1 时 ,trans[1][b]=3 longest(1)+b!=longest(3)
- 所以新建一状态8？
- 如果不新建状态，状态7的ab后缀怎么处理？



状态	子串	endpos
S	空串	{0,1,2,3,4,5,6}
1	a	{1,2,5}
2	aa	{2}
3	aab	{3}
4	aabb,abb,bb	{4}
5	b	{3,4,6}
6	aabba,abba,bba,ba	{5}
7	aabbab,abbab,bbab,bab	{6}
8	ab	{3,6}

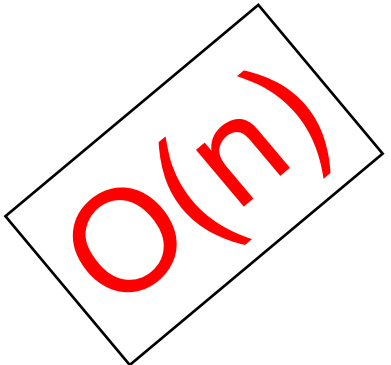
- 新建状态8后，复制状态3的转移即
- tran[8][b]=tran[3][b]=4
- 继承状态3的link，即link[8]=link[3]=5
- 然后把断开的接到y上， trans[v..w][c]=8
- 最后，更新x,z的link link[3]=link[7]=8

算法流程回顾

- 一、新建状态z
- 二、对于 suffix-path(u→S) 路径上的任一状态 v，若trans[v][S[i+1]]=NULL，使trans[v][S[i+1]]=z
  若全是NULL的，使link[z]=S
- 三、若后缀链接路径上有trans[v][S[i+1]]!=NULL;
  则令 x=trans[v][S[i+1]],
  若maxlen(x)=maxlen(v)+ 1， link[z]=x，
  若maxlen(x)>maxlen(v)+ 1 ,则需要新建状态

时空复杂度

- **状态的数量**
- 长度为 n 的字符串 s 建立的后缀自动机的状态个数不超过 $2n-1$（$n\geq 3$）
- **转移的数量**
- 长度为 n 的字符串 s 建立的后缀自动机中，转移的数量不超过 $3n-4$（$n\geq 3$）。



状态的数量证明：

- 上面描述的算法证明了这一性质（最初自动机包含一个初始节点，第一步和第二步都会添加一个状态，余下的 n−2步每步由于需要分割，至多增加两个状态）。

- 所以就是 1+2+(n−2)×2=2n−1了。

转移的数量证明：

我们计算 连续的 转移个数。考虑以 S 为初始节点的自动机的最长路径树。这棵树将包含所有连续的转移，树的边数比结点个数小 1，这意味着连续的转移个数不超过 2n−2。

我们再来计算 不连续 的转移个数。考虑每个不连续转移；假设该转移——转移 (p,q)，标记为 c。对自动机运行一个合适的字符串 u+c+w，其中字符串 u 表示从初始状态到 p 经过的最长路径，w 表示从 q 到任意终止节点经过的最长路径。

一方面，对所有不连续转移，字符串 u+c+w 都是不同的（因为字符串 u 和 w 仅包含连续转移）。另一方面，每个这样的字符串 u+c+w，由于在终止状态结束，它必然是完整串 s 的一个后缀。由于 s 的非空后缀仅有 n 个，并且完整串 s 不能是某个 u+c+w（因为完整串 s 匹配一条包含 n 个连续转移的路径），那么不连续转移的总共个数不超过 n−1。

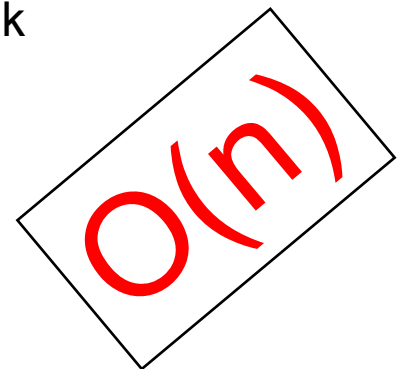
有趣的是，仍然存在达到转移个数上限的数据：abbb...bbbc

代码

- const int Maxn = 2*1000;　//字符串最大长度
- int st[Maxn];　　//状态数Maxn*2-1
- int x,y,z;　　//与上文一致，x表示trans[st][c]
  //y表示复制的节点，z新建的状态
- int size=last=0; //size 当前的状态总数
  // last 为上次插入的状态的状态编号
- int maxlen[Maxn], trans[Maxn][26], link[Maxn];

memset(link,-1,sizeof(link));　//link初始为-1

```
void Suffix_Automata (char c) {
        int z= ++size;
         maxlen[z] = maxlen[last] + 1;
        int p;    //link路径
        for (p=last;  p!=-1&&!trans[p][c]; p=link[p])  trans[p][c] = z;
        if (p==-1) link[z] = 0;                  //路径上全是null,则 link[z] = S
        else {
                int x = trans[p][c];
                 if (maxlen[x] == maxlen[p] + 1) link[z] = x;
                else {
                        int  y= ++size;
                         maxlen[y] = maxlen[p] + 1;
                         memcpy( trans[y],trans[x],sizeof(trans[x]));  //复制X的转移
                         link[y]=link[x];          //继承x的link
                         for (; p!=-1&& trans[p][c] == x; p = link[p])
                                                  trans[p][c] = y;    //断开的转移到y
                         link[x] = link[z]= y;           //更新x和z的link
                }
        }
        last = z;
}
```



例一　后缀自动机——基本概念

- 输入　　hihocoder 1441

  第一行包含一个字符串S，S长度不超过50。

  第二行包含一个整数N，表示询问的数目。(1 <= N <= 10)

  以下N行每行包括一个S的子串s，s不为空串。

- 输出

  对于每一个询问s，求出包含s的状态st，输出一行依次包含shortest(st)、longest(st)和endpos(st)。其中endpos(st)由小到大输出，之间用一个空格分割。

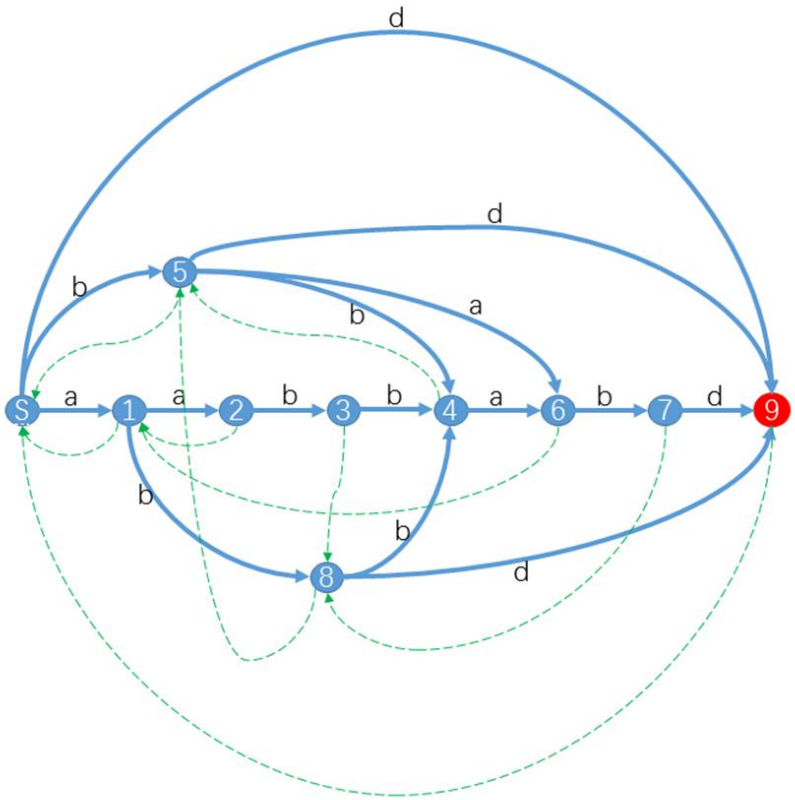
样例输入
```
aabbabd
5
b
abbab
aa
aabbab
bb
```

样例输出
```
b b 3 4 6
bab aabbab 6
aa aa 2
bab aabbab 6
bb aabb 4
```

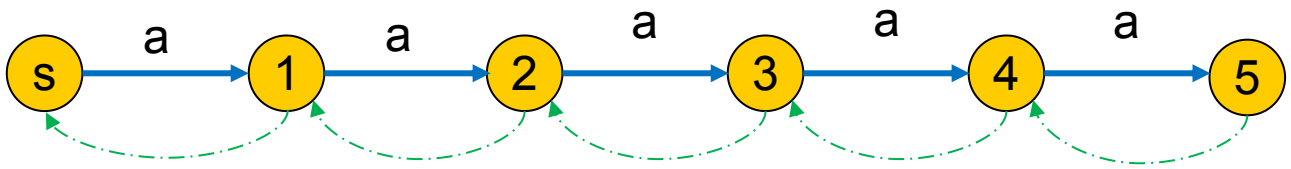
http://hihocoder.com/problemset/problem/1441

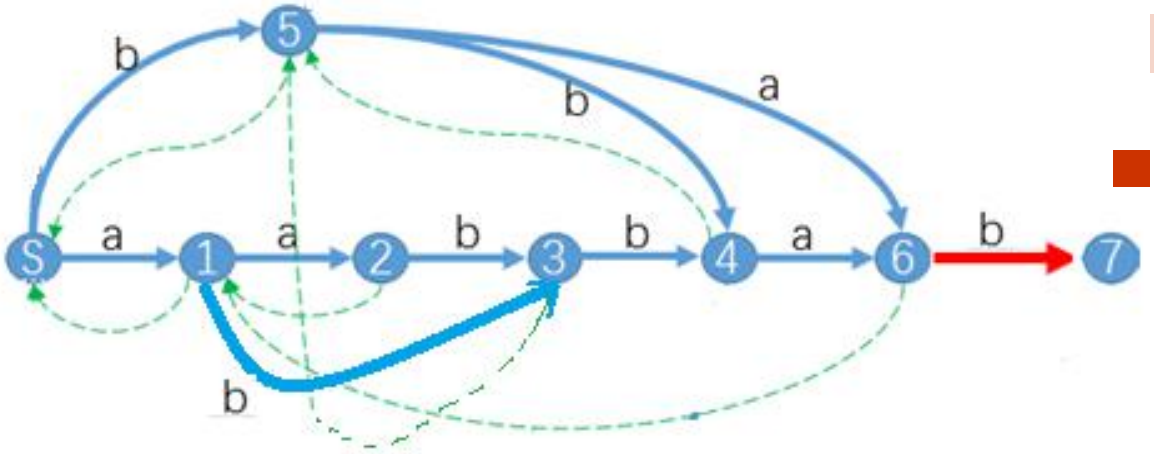
法二

- 构造SAM,同时每一个状态节点记录endpos的st (二进制)
- dfs,记录每个子串属于哪个节点，同时更新longest(st)和shortest(st)



```
void Suffix_Automata (int c) {
        int z= ++size;
        maxlen[z] = maxlen[last] + 1;
        st[z]|= (1LL<<k);  //更新endpos
        int p;  //link路径
        for (p=last;  p!=-1&&!trans[p][c]; p=link[p])  trans[p][c] = z;
        if (p==-1) link[z] = 0;     //路径上全是null,则 link[z] = S
        else {
              int x = trans[p][c];
             if (maxlen[x] == maxlen[p] + 1) {
                 link[z] = x;
                 for (int pp=x; pp!=-1; pp = link[pp]) st[pp]|= (1LL<<k);  //沿着link，更新endpos
         } else {
                      int  y= ++size;
                      st[y]=st[x];     ////复制x的st
                       st[y]|= (1LL<<k);  //更新endpos
                         maxlen[y] = maxlen[p] + 1;
                memcpy( trans[y],trans[x],sizeof(trans[x]));  //复制X的转移
                link[y]=link[x];            //继承x的link
               for (; p!=-1&& trans[p][c] == x; p = link[p]) trans[p][c] = y;    //断开的转移到y
               link[x] = link[z]= y;             //更新x和z的link
                for (int pp=link[y]; pp!=-1; pp = link[pp]) st[pp]|= (1LL<<k); //沿着link，更新endpos
                      }
        }
        last = z;
}
```

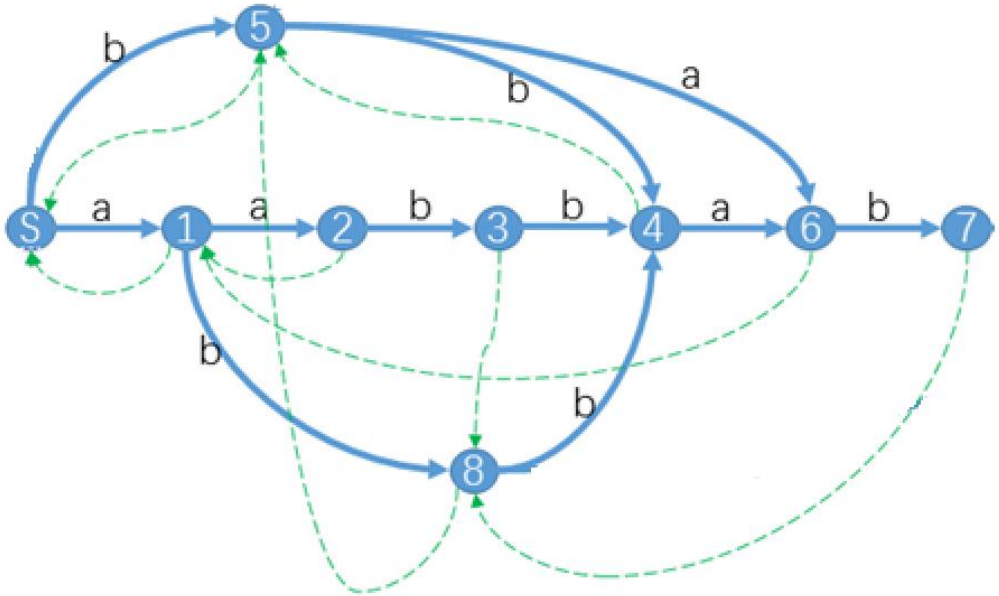

a
a
a
a
a
s
1
2
3
4
5



- 更新时节点3作为x被拆分后，新建了节点8，
- st[y]|= (1LL<<k);

8节点的st更新了，但link[3]=5，5节点的st并没有更新，

所以要沿着link继续更新。



```
void Suffix_Automata (int c) {
        int z= ++size;
        maxlen[z] = maxlen[last] + 1;
        st[z]|= (1LL<<k);  //更新endpos
        int p;  //link路径
        for (p=last;  p!=-1&&!trans[p][c]; p=link[p])  trans[p][c] = z;
        if (p==-1) link[z] = 0;     //路径上全是null,则 link[z] = S
        else {
                int x = trans[p][c];
                 if (maxlen[x] == maxlen[p] + 1) {
                    link[z] = x;          st[x]|= (1LL<<k);  //更新endpos
                 } else {
                   int  y= ++size;
                   st[y]=st[x];           //复制x的st
                   maxlen[y] = maxlen[p] + 1;
                   memcpy( trans[y],trans[x],sizeof(trans[x]));  //复制X的转移
                    link[y]=link[x];          //继承x的link
                for (; p!=-1&& trans[p][c] == x; p = link[p]) trans[p][c] = y;    //断开的转移到y
                link[x] = link[z]= y;           //更新x和z的link
                }
        for (int pp=link[x]; pp!=-1; pp = link[pp]) st[pp]|= (1LL<<k);  //沿着link，更新endpos
        }
        last = z;
}
```

两种情况的更新写在一起

```
map<int , string> shortest, longest;
map<string , int> dict;
void dfs(int root,string ss){
        for(int i=0;i<26;i++){
          if(trans[root][i]) {
                string x=ss+(char)(i+'a');     //每到一个节点，生成新的子串
                int xson=trans[root][i];
                dict[x] = xson;
                int lens=x.length();
           if(shortest[xson].length()==0 || lens< shortest[xson].length())
                                shortest[xson] = x;
           if(lens > longest[xson].length() ) longest[xson] = x;

            dfs(xson,x);
                }
        }
}
```

例二　生成魔咒

SDOI2016

Description

魔咒串由许多魔咒字符组成，魔咒字符可以用数字表示。例如可以将魔咒字符1、2拼凑起来形成一个魔咒串[1,2]。

一个魔咒串S的非空子串被称为魔咒串S的生成魔咒。例如S=[1,2,1]时，它的生成魔咒有[1]、[2]、[1,2]、[2,1]、[1,2,1] 五种。S=[1,1,1]时，它的生成魔咒有[1] 、[1,1]、[1,1,1] 三种。

最初S为空串。共进行n次操作，每次操作是在S的结尾加入一个魔咒字符。每次操作后都需要求出，当前的魔咒串S共有多少种生成魔咒。

Input

第一行一个整数n。

第二行n个数，第i个数表示第i次操作加入的魔咒字符。

Output

输出n行，每行一个数。第i行的数表示第i次操作后S的生成魔咒数量。

Sample Input	Sample Output
7	1
1 2 3 3 3 1 2	3
	6
	9
	12
	17
	22

HINT

对于 10% 的数据，$1 \leq n \leq 10$；

对于 30% 的数据，$1 \leq n \leq 100$；

对于 60% 的数据，$1 \leq n \leq 1000$；

对于 100% 的数据，$1 \leq n \leq 100000$。

用来表示魔咒字符的数字 x 满足 $1 \leq x \leq 10^9$。

分析

- 1.x的范围较大，1<=x<=10^9，n也较大，如果trans[2*maxn][x]必定暴空间，怎么处理？
- 2.后缀自动机如何计算子串个数？

- 对于问题1有两种办法：离散和使用map影射

```
map<int , int> trans[2*maxn];
void Suffix_Automata (int c) {
        int z= ++size;
        maxlen[z] = maxlen[last] + 1;
        int p;  //link路径
        for (p=last;  p!=-1&&!trans[p][c]; p=link[p])  trans[p][c] = z;
        if (p==-1) link[z] = 0;       //路径上全是null,则 link[z] = S
        else {
             int x = trans[p][c];
            if (maxlen[x] == maxlen[p] + 1) link[z] = x;
             else {
                    int  y= ++size;
                     maxlen[y] = maxlen[p] + 1;
                     trans[y]=trans[x];
                     link[y]=link[x];          //继承x的link
                    for (; p!=-1&& trans[p][c] == x; p = link[p]) trans[p][c] = y;    //断开的转移到y
                     link[x] = link[z]= y;           //更新x和z的link
                }
            }
        last = z;
        ans+=maxlen[z]-maxlen[link[z]];     //累加子串个数
}
```

广义后缀自动机

- 传统后缀自动机是解决单个主串的匹配问题，广义后缀自动机可以用来解决多个主串的匹配问题。
- 如何将多个主串构建成广义后缀自动机？
- 先将一个主串建立成后缀自动机，让后将重置last，令last=root，下一个字符串再从头节点开始建立，下一状态如果不存在，则以后缀自动机的规则进行建立新节点。

例三　　串　　BZOJ3277

Description
字符串是oi界常考的问题。现在给定你n个字符串，询问每个字符串有多少子串（不包括空串）是所有n个字符串中至少k个字符串的子串（注意包括本身）。

Input
第一行两个整数n，k。
接下来n行每行一个字符串。
n,k,l<=100000

Output
输出一行n个整数，第i个整数表示第i个字符串的答案。

Sample Input
3 1
abc
a
ab

Sample Output
6 1 3

- 把n个串的广义后缀自动机建出来，然后统计每个点所代表的串出现的次数。这个数值相当于它的fail子树中每个点代表的字符串的并集的大小。
- 求出dfs序后，这个问题就相当于区间询问有多少个不同的数字，把询问按照右端点排序，每新加入一个数字就在它出现的上一个位置减掉，这样询问就是区间求和。求出来这个出现次数以后，如果它大于等于k，这个点的贡献就是val[u]-val[fail[u]]【因为如果这个点有贡献，这个点的fail也一定有贡献】，否则没有贡献。
- 然后再把这个贡献按照拓扑序更新一次，得到以每个点为结尾的子串的贡献，最后把每个串在自动机上dfs一遍就可以统计出答案了。

广义后缀自动机的题：

POJ3294： **题意**：给定一些模板字符串，求一个最长公共字串，这个最长公共字串至少在一半以上的字符串里出现过。

对比：如果是后缀数组，则要+二分+RMQ；而广义后缀自动机只需要记录出现的位置，最后传递即可。

SPOJ8093 **题意**：给定一些模板串，询问每个匹配串在多少个模板串里出现过。

对比：同上。传递的两种方式：每加一个字符传递一次；也可以用bitset记录在哪里出现过等到加完所有字符串后再拓扑排序，然后"亦或"向上传递。

后缀自动机的应用

- 两个串的最长公共子串

  建出A串的后缀自动机，然后B串在后缀自动机上跑

- 统计本质不同的子串的个数
- 计算任意子串出现的次数
- 统计所有本质不同子串的权值和
- 找第**K**大的子串
- 求最小循环表示
- 找回文串

判断子串

- 直接在后缀自动机上跑边，跑完串还未跑到 NULL 则为原串子串。

统计本质不同的子串的个数

方法一：用dfs处理处每个点能扩展出多少个字符串$sum[x]=\sum sum[t[x].son[i]]+1$，其实可以不用dfs，拓扑一下（按len从小到大）然后倒着做。最后sum[1]就是所有子串的个数。
方法二：$ans=\sum t[x].len-t[t[x].fa].len$，这里运用了性质1，因为节点i表示的字符串大小范围是从len[fa[i]]+1...len[i]的，那么包含不同串的个数=$len[i]-(len[fa[i]]+1)+1=len[i]-len[fa[i]]$

HihoCoder 1445

其实这个可以用后缀数组做，具体来说，答案就是 $\sum_{i=1}^{n}(n-sa[i]+1)-height[i]$ 。我们考虑 sa 相邻两个后缀。首先多出了 $(n-sa[i]+1)$ 个后缀，然后 LCP 长度为 $height[i]$ 的子串重复计算过，减去就行了。

用 SAM 的话，其实就是统计所有状态包含的子串总数，也就是 $\sum_{i=1}^{Size}maxlen[i]-minlen[i]+1$ ，建完直接算就行了。注意前面讲过的 $minlen[i]=maxlen[link[i]]+1$ 。

找第K大的子串

1、找不同串的第K大：预处理出每个状态可以构出多少个字符串，可以用dfs做，也可以对自动机拓扑一下（其实就相当于把len从小到大排序，因为len小的拓扑序也会小），然后倒着求一下（相当于DAG上的DP），然后dfs去找第K大的就好了。

2、找相同串的第K大：除了要预处理出上面的东西，还要预处理出所有状态right集合的大小（每个串在原串中出现多少次），这个会影响上面的要求的值，然后在做dfs的时候同时处理一下就好了。

BZOJ3998弦论　　洛谷**P3975**

求最小循环串

- 给一个字符串S，每次可以将它的第一个字符移到最后面，求这样能得到的字典序最小的字符串。
- 如*BBAAB*，最小的就是*AABBB*

- 把原串复制一遍到后面即s+s，建立ss的后缀自动机，然后从root开始走length(s)步，每次走转移最小的。
- 由于 SAM 可以接受 SS 所有的子串，而字典序最小的字符串也必定是 SS 的子串，因此按照上面的规则移动就可以找到一个字典序最小的子串。

[POJ1509]Glass Beads

找回文串

构造原串的后缀自动机，求出每个节点endpos集合的rmax，然后把反串放到后缀自动机上面运行，如果当前的匹配串在原串中的范围[l...r]覆盖了当前节点的rmax，那么[l...rmax]就是一个回文串。

总结

- 其实后缀数组能干的很多事情都可以用后缀自动机来干，后缀自动机因为有树形结构所以加上了树链剖分可以用很多数据结构来维护，它的代码简介，常数又小，速度又快，但是需要多加思考才能解决题目。

参考 https://www.cnblogs.com/zjp-shadow/p/9218214.html

后缀数组sa和后缀自动机SAM

- 1、SAM往往是增量法实现,所以对于在串后动态增加的问题SAM可以做，sa则不行。
- 2、sa是转到数组上，SAM是转到树上,一般数组上较容易
- 3、处理多串问题的时候，如果是一对多的关系SAM使用较多。