冬令营上我犯了最大的一个错误，就是在陈立杰讲后缀自动机的时候睡觉。

这导致了，我在冬令营之后只能花费好几个不眠之夜来思考后缀自动机到底是什么。

突然，在某天的梦里，我看到了一幅神奇的图景，我突然发现，一切都是那么的明晰了。  
  
先说说后缀自动机(Suffix Automaton)是什么东西。一个串A的后缀自动机是一个有限状态自动机(DFA)，它能够且仅能够接受A的后缀，并且我们要求它的状态数最少。

有一个强大的定理说明，N个字符的后缀自动机的状态数与转移数都不会超过O(N)。

举个例子：串A="abaaaba"

通过一番努力，我们能够看出，它的后缀自动机是：

　　　　　　　　　　　　ｂ－－－－－＞4－－ａ－－＞6

　　　　　　　　　　　　｜　　　　　　＾  
　　　　　　　　　　　　｜　　　　　　ｂ

　　　　　　　　　　　　｜　　　　　　｜

S－－ａ－＞１－ａ－－＞3－－ａ－－＞5

｜　　　　｜　　　　　　　　　　　　　＾

｜　　　　ｂ　　　　　　　　　　　　　ａ

｜　　　　ｖ　　　　　　　　　　　　　｜

ｂ－－－＞2－ａ－－＞7－ａ－－－－＞８

其中，红色的S表示开始状态，蓝色的表示接受状态，绿色的表示非接受状态。

后缀自动机有什么性质呢？又该如何构造后缀自动机呢？

先看第一个神奇的性质：

我们观察从每个点出发，能够走到接受状态的所有路径（用/来表示空路径）：

S：a ba aba aaba aaaba baaaba abaaaba

1：/ ba aba aaba baaaba

2：a aaaba

3：ba aba

4：a

5：ba

6：/

7：/ aaba

首先，所有的路径都是串A的后缀（显然）。

其次，对于任何两个状态，它们能够接受的路径要么没有交集，要么是包含关系。

这个直接保证了，状态数是O(N)的。

让我们来看第二个性质：

　　　　　　　　　　　　ｂ－－－－－＞4－－ａ－－＞6

　　　　　　　　　　　　｜　　　　　　＾  
　　　　　　　　　　　　｜　　　　　　ｂ

　　　　　　　　　　　　｜　　　　　　｜

S－－ａ－＞１－ａ－－＞3－－ａ－－＞5

｜　　　　｜　　　　　　　　　　　　　＾

｜　　　　ｂ　　　　　　　　　　　　　ａ

｜　　　　ｖ　　　　　　　　　　　　　｜

ｂ－－－＞2－ａ－－＞7－ａ－－－－＞８

我们再观察从起点出发，能走到每个点的路径：

S：/

1：a

2：b ab

3：aa

4：aab aaab baaab abaaab

5：aaa baaa abaaa

6：aaba aaaba baaaba abaaaba

7：ba aba

8：baa abaa

它们的特点是：

首先，这些路径两两不同，共同组成了串A的所有子串（显然）。

其次，能走到一个状态的路径都是某个串在某个长度区间里的后缀。（例如走到6的串是abaaaba的长度为4~7的后缀）。

这个性质是否让你浮想联翩？你是否发现了一个后缀树？

是的！不过，它是A的逆序的后缀树！

　　　　　　　　　　　　S

　　　　　　　　　　　／　＼

　　　　　　　　　　ａ　　　ｂａ

　　　　　　　　　／　　　　　＼

　　　　　　　　１　　　　　　　２

　　　　　　　／　＼　　　　　　　＼

　　　　　　ａ　　　ｂａ　　　　　　ａａｂａ

　　　　　／　　　　　＼　　　　　　　＼

　　　　３　　　　　　　７　　　　　　　４

　　　／　＼　　　　　　　＼

　　ｂａ　ａｂａ　　　　ａａｂａ

　　／　　　　＼　　　　　　　＼

　８　　　　　　５　　　　　　　６

（真不幸，这个例子里的A是回文串。。。）

我们发现，后缀树中的每个节点和后缀自动机里的节点一一对应。树上，从S走到某个节点的路径，对应了自动机里从某个节点走回起点的路径。

这个使得如果我们能够建立一个串的后缀自动机，我们就能够建立它的逆的后缀树。这样，任何后缀树（后缀数组）能够做的事情，后缀自动机都能够做。

后缀自动机是可以O(N)时间内构造的！怎么构造？这等于问我：如何在O(N)的时间里构造一个串的后缀树。

线性构造后缀树？听起来挺难的，不过，后缀自动机给了我们一个超级简单的方法：增量法。

对于A="abaaaba"，我们往自动机里从短往长添加A的前缀，注意，是前缀！

a ab aba abaa abaaa abaaab abaaaba

对于后缀树来说，就是依次添加前缀的逆序：

a ba aba aaba aaaba baaaba abaaaba

我们记录树上每个节点p的父亲Par[p]和它代表的串的长度Lth[p]，同时记录后缀自动机里它的转移Trans[p][x]

注意，Trans[p][x]的意义在树上说就是：p代表的串，往前添加一个字符x，得到的串是谁的前缀？

假设我们已经构造好了n-1个字符的后缀自动机，从开始状态走n-1步得到的节点是u（也就是说，树上代表A[n-1..1]的节点是u），我们要添加一个字符x，那么就是说要往树里插入一个字符串'x'+A[n-1...1]。

新建一个节点p，设定好长度为n。

p该插入到哪里呢？

我们想：如果Trans[u][x]不是NULL，那么p就应该插入到Trans[u][x]的下面。（根据Trans在树上的意义，Trans[u][x]代表的是，u对应的串的头部添加一个字符得到的串对应的节点，刚好就是p应该去的地方！）。

如果Trans[u][x]是NULL呢？那么，我们就应该看Trans[Par[u]][x]是否是NULL、Trans[Par[Par[u]]][x]是否是NULL。。。

直到找到某个祖宗w，它的Trans[w][x]不是NULL。

令ｑ=Trans[w][x]

　　　　　　　　　　　　　　　　　　S

　　　　　　　　　　　　　　　　／　　　＼

　　　　　　　　　　　　　．．．　　　　．．．

　　　　　　　　　　　　　／　　　　　　　＼

　　　　　　　　　　　　ｗ　　　　　　　　　ｑ

　　　　　　　　　　　／　　　　　　　　　　　＼　　　　　　ｐ？

　　　　　　　　　　．．．　　　　　　　　　　．．．

　　　　　　　　　／

　　　　　　　　ｕ

容易看出，p和q享有长度为Lth[w]+1共同的前缀，它们应该变成兄弟。

　　　　　　　　　　　　　　　　　　S

　　　　　　　　　　　　　　　　／　　　＼

　　　　　　　　　　　　　．．．　　　　．．．

　　　　　　　　　　　　　／　　　　　　　＼

　　　　　　　　　　　　ｗ　　　　　　　　　ｒ(Lth[r]=Lth[w]+1)

　　　　　　　　　　　／　　　　　　　　　／　＼

　　　　　　　　　　．．．　　　　　　　ｐ　　　ｑ

　　　　　　　　　／　　　　　　　　　　　　　　　＼

　　　　　　　　ｕ　　　　　　　　　　　　　　　　　．．．

注意，如果Lth[w]+1恰好等于Lth[q]，那么p应该变成q的孩子。

　　　　　　　　　　　　　　　　　　S

　　　　　　　　　　　　　　　　／　　　＼

　　　　　　　　　　　　　．．．　　　　．．．

　　　　　　　　　　　　　／　　　　　　　＼

　　　　　　　　　　　　ｗ　　　　　　　　　ｑ(Lth[q]==Lth[w]+1)

　　　　　　　　　　　／　　　　　　　　　／　＼

　　　　　　　　　　．．．　　　　　　　ｐ　　　．．．

　　　　　　　　　／

　　　　　　　　ｕ

这样，我们就通过之前定义好的Trans链条找到了p的位置！

计算出Par[p]之后，如何更新Trans[][]呢？

只有两部分的Trans[][x]需要修改：

u~w的Trans[][x]应该修改为p

w及祖先中，所有Trans[?][x]==q的应该替换成r

讨论完以上问题之后，我们发现了一个超级短的后缀自动机的建立代码（和你的后缀数组比比？）

（变量的标号和文中不一样）

void build(){

root=curnode=new node(0,NULL);//最开始的后缀自动机只有一个节点，长度是0，父亲是空

for (int i=0;i<N;i++){

int x=A[i]-'a';//增加一个字符

node \*p=curnode;

curnode=new node(i+1,NULL);//建立一个Lth为i+1的节点

for (;p && p->trans[x]==NULL;p=p->p)p->trans[x]=curnode;//沿祖先向上，寻找插入位置。同时更新Trans

if (!p)curnode->p=root;//插入到根的下面

else{

node \*q=p->trans[x];

if (q->l==p->l+1)curnode->p=q;//成为q的孩子

else{

node \*r=new node();r[0]=q[0];r->l=p->l+1;//新建一个节点，表示curnode和q的公共前缀

q->p=r;curnode->p=r;//兄弟

for (;p && p->trans[x]==q;p=p->p)p->trans[x]=r;//更新第二部分的Trans

}

}

}

}

震撼吧？线性构造后缀树，或者说，线性构造后缀自动机，原来这么容易！

为什么时间复杂度是线性的？嗯，势能分析吧。。。总之，真的是线性的。

构造完成之后，我们拿它有什么用呢？

首先，是解决LCS（最长公共前缀）的查询。因为我们能够构造出后缀树，所以最长公共前缀就成为了最近公共祖先的查询。

然后，是解决子串计数。一个子串在原串中出现多少次，等于从开始状态沿着这个子串转移到达某个状态之后，还有多少条路径能够走到接受态。这个可以通过把自动机拓扑排序（或者，更暴力点，按Lth排序）之后dp来实现。

接下来，是解决字符串匹配。建立模板串的后缀自动机。然后设定一个初始指向开始状态的指针p，依次读取文本串的字符x，如果Trans[p][x]不是空，就沿着Trans[p][x]走，匹配长度加一，否则让p沿着Par[p]回退，同时调整匹配长度为Lth[p]，直到Trans[p][x]不是空，再沿着Trans[p][x]走。这样，我们就能够知道，当前文本串出现在模式串中最长的后缀的长度。（其实建立文本串的后缀自动机也可以。。。）

for (int i=0,l=0;i<M;i++){

int x=B[i]-'a';

while (Par[p]!=-1 && Trans[p][x]==-1)p=Par[p],l=Lth[p];

if (Trans[p][x]!=-1){

result=max(result,l);

l++;

p=Trans[p][x];

}else p=0,l=0;

}

最后，最本质的一点，就是能够解决任何后缀树能够解决的问题。

举几个例子吧。（来自陈立杰）

SPOJ NSUBSTR（问出现次数为i的子串有多少个）

这个的做法我们已经讨论过了。计算每个状态有多少种方法走到接受态（等于它在后缀树的子孙里有多少个接受状态，这两种计算方法都是可以的）。用这个数更新这个状态代表的串的长度的结果。之后在用长的结果去更新短的结果。

SPOJ SUBLEX（问字典序第x小的后缀）

可以利用后缀树来搞。或者，计算走到每个状态之后有多少种走法（利用dp）。然后dfs一遍即可。

SPOJ LCS（最长公共连续子串）

当然可以把两个串拼起来之后用后缀树来搞。也可以用上面提到的方法，计算第二个串的每个前缀出现在第一个串中的最长的前缀。

SPOJ LCS2（多个串的最长公共前缀）

建立一个串的自动机。之后利用扫描和treeDP，计算每个状态代表的串出现在所有子串中最长的后缀。

By fanhq666