后缀自动机（姬）

算法思想

后缀自动机简称 SAM □

我们记 $\sum$ 为字符集□ $|\sum|$ 为字符集大小。

以下问题可以通过 SAM 在线性时间内解决。

1.在另一个字符串中搜索一个字符串的所有出现位置□□ kmp □哈希也可以）

2.计算给定的字符串中有多少个不同的子串。（后缀数组也可以线性做）

在直观上，我们可以把 SAM 理解为将字符串的所有子串压缩在一个树上。对于长度为 n 的字符串□ SAM 的空间复杂度是 $O(n)$ 的，此外构造 SAM 的时间复杂度也是 $O(n)$ 的，小结论：一个 SAM 最多有 $2n-1$ 个结点和 $3n-4$ 条转移边。

我们把结点称作状态，边称为状态之间的转移。我们有一个初始的源点作为初始状态，其他各个结点都可以从这个源点出发到达。每个转移都标有一些字母，从一个结点出发的所有转移都不同。存在一个或多个终止状态，路径上所有转移连接起来一定是字符串的一个后缀，并且每一个后缀都可以用一条从源点到终止状态的路径构成。所以子串就是后缀的前缀也就是从源点开始到任意一个点的路径，所以可以说一个点对应着一个原字符串的子串。

结束位置 $endpos$ □考虑字符串 s 的任意非空子串 t □我们记 $endpos(t)$ 为在字符串 s 中 t 的所有结束位置（从 0 开始），比如现在有一个字符串叫 $abcbc$ □我们有 $endpos("bc")=2,4$ □不同子串 t_{1} 和 t_{2} 的 $endpos$ 集合可能相等，比如刚才的 c 和 bc □所以我们可以根据 $endpos$ 集合的不同将 s 的非空子串分为若干等价类。

SAM 中的每个状态对应一个 $endpos$ 相同的等价类，还有一个初始状态，所以 SAM 的状态个数等于等价类的个数 $+1$ 。

假设字符串 s 的两个非空子串 u 和 v 的 $endpos$ 相同，显然有短的字符串是长的字符串的后缀□ $endpos$ 之间要么相交是空（不为后缀关系），要么是被包含的关系（其中一个是另一个的后缀），且满足这个等价类的子串的长度是一个连续的区间。

后缀链接 $link(v)$ 连接到对应于比 v 等价类最短的还短一个字符的字符串的等价类。所有的后缀链接构成一棵根节点为 t_{0} 的树。如果我们从任意状态 v_{0} 开始沿着后缀链接遍历，总会到达初始状态 t_{0} □这种情况会得到一个互不相交的区间 $[minlen(v_{i}),len(v_{i})]$ □并且他们的并集正好是 $ [0,len(v_{0})]$ □

后缀自动机的构造是在线的，我们可以逐个加入字符串的每个字符，并且每一步维护 SAM □我们为了保证线性的空间复杂度，只保存 len 和 $link$ 的值和每个状态的转移列表，不会标记终止状态。一开始 SAM 只包含一个状态 t_{0} □编号为 0，为了方便我们规定 t_{0} 的 $len=0,link=-1$ □ -1 表示虚拟状态）。现在我们开始插入字符：令 $last$ 为添加字符 c 之前，整个字符串对应的状态（每次的最后一步都会更新这个值）。创建一个新的状态 cur □因为整个字符串一定是第一次出现的），并将 $len(cur)$ 赋值为 $len(last)+1$ □这时 $link(cur)$ 的值还未知。

从状态 $last$ 开始，如果没有字符 c 的转移，我们就添加一个到状态 cur 的转移，遍历后缀链接，如果在某个点已经存在到字符 c 的转移，我们就停下来，并将这个状态标记为 p □

如果没有找到这个 p □则到达了 -1，此时我们将 $link(cur)$ 赋值为 0 并退出。

如果找到了，且我们知道经过字符 c 转移后的状态为 q 现在分情况进行讨论：

如果 $len(p)+1=len(q)$ 我们只需要将 $link(cur)$ 赋值为 q 并退出。

否则我们需要复制 q 我们创建一个新的状态 $clone$ 复制 q 除了 len 的值之外的所有信息（包括后缀链接和转移），然后将 $len(clone)$ 赋值为 $len(p)+1$ 之后我们将 $link(cur)$ 指向 $clone$ 也将 $link(q)$ 指向 $clone$ 这样就是 q 发现了一个介于原来两个之间的状态，于是需要把 $link(q)$ 指向 $clone$

无论哪种情况，最后我们都将 $last$ 的值更新为 cur

这里遍历后缀链接的原因是：我们创建了一个新的状态之后，对后缀链接这些状态进行遍历，尝试添加通过一个字符 c 到新状态 cur 的转移，但是我们不能覆盖之前的合法转移，当没有找到时，说明这个字符从未出现过，所以后缀链接为 0。如果出现了，说明我们正在向自动机内添加一个已经存在了的子串，如果存在 $len(q)=len(p)+1$ 说明之前已经有一个状态的转移是一样的，直接连到转移后的就可以了；如果不存在，说明转移是不连续的，即 q 不仅对应于长度为 $len(p)+1$ 的后缀，还对应着更长的子串，就需要拆开状态 q 来创建这样的状态。但是这样还没完，我们需要把一些本来转移到 q 的转移重定向到 $clone$ 我们需要继续沿着后缀链接遍历，从结点 p 直到 -1 或者转移到不是状态 q 的一个转移。

因为我们只为 s 的每个字符创建了一个或者两个新状态，所以 SAM 只包含线性个状态。

算法实现

如果你用 map 存储转移列表，时间复杂度会变成 $O(nlog|\sum|)$ 当字符集为较小的常数，比如 26 时，就将转移数组设为 $int[26]$ 即可。

给 SAM 赋予树形结构，树的根为 0，其余结点 v 的父亲为 $link(v)$ 则 $S_{1...p}$ 和 $S_{1...q}$ 的最长公共后缀对应的字符串就是 v_{p} 和 v_{q} 对应的 LCA 的字符串。显然每个状态对应的子串种类数是 $len(i)-len(link(i))$

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN=1001000;
struct NODE {
    int ch[26];
    int len,fa;
    NODE() {
        memset(ch,0,sizeof(ch));
        len=0;fa=0;
    }
} dian[MAXN<<1];
int las=1,tot=1;
void add(int c) {
    int p=las;
    int np=las=++tot;
    dian[np].len=dian[p].len+1;
    for(; p&&!dian[p].ch[c]; p=dian[p].fa)dian[p].ch[c]=np;
    if(!p)dian[np].fa=1;//以上为case 1
```

```
    else {
        int q=dian[p].ch[c];
        if(dian[q].len==dian[p].len+1)dian[np].fa=q;//以上为case 2
        else {
            int nq=++tot;
            dian[nq]=dian[q];
            dian[nq].len=dian[p].len+1;
            dian[q].fa=dian[np].fa=nq;
            for(; p&&dian[p].ch[c]==q; p=dian[p].fa)dian[p].ch[c]=nq; //以上为case 3
        }
    }
}
char s[MAXN];
int len;
int main() {
    scanf("%s",s);
    len=strlen(s);
    for(int i=0; i<len; i++)add(s[i]-'a');
    return 0;
}
```

算法练习

1.给一个文本串 T 和多个模式串 P □询问 P 是否作为 T 的一个子串出现。

首先用 $O(|T|)$ 的时间对 T 构造后缀自动机，从 t_{0} 开始根据 P 开始转移，如果在某个点无法转移下去，则不是子串，反之则出现过，为子串。时间复杂度为 $O(|P|)$ □且可以求出 P 在文本串中出现的最大前缀长度（其实后缀数组也可以做，连起来用字符分割，用 $O(n)$ 的算法构造，然后看那个位置 rk 数组值加一位置的 $height$ 数组值，即为最长前缀长度）。

2.给一个字符串 S □计算不同子串的个数

做法 1 ：后缀数组显然可以做，这里可以对 S 构造后缀自动机，每一个 S 的子串都对应自动机的路径，所以不同子串的个数等于自动机中以 t_{0} 为起点的不同路径的条数。树形 dp 的思想，我们设 $dp[v]$ 为状态 v 开始的路径数量（包括空串），则可以有 $dp[v]=1+\sum_{(v,w,c)exists}dp[w]$ □也就是说每一棵子树的所有情况都可以包括进来（包括空，因为有一条边这样肯定不是空串了），最后再加上自己的那个空串，所以最后加 1，所以最后不同字串的个数是 $dp[t_{0}]-1$ □把空串排除）。复杂度 $O(|S|)$ □

做法 2 ：每个结点对应的子串数量是 $len(i)-len(link(i))$ 对所有结点求和即可，复杂度也是 $O(|S|)$ □

3.给定一个字符串 S □计算所有不同子串的总长度

做法 1 ：对两个部分进行树形 dp □不同字串的数量 $dp[v]$ 和他们的总长度 $ans[v]$ □上一问的做法我们把 $dp[v]$ 搞定之后□ $ans[v]=\sum_{(v,w,c)exists}dp[w]+ans[w]$ □因为有多少个串，都会多一个长度，所以在原有的答案上再加上有多少个串就可以了）。复杂度仍然是 $O(|S|)$ 的。

做法 2 ：同上一问，后缀长度连续，为 $len(i)$ 到 $len(link(i))+1$ □所以等差数列求和，对所有顶点相加即可，复杂度同样是 $O(|S|)$ 的。

4.字典序第 k 大子串。多组询问，每次询问 k

字典序第 k 大对应着 SAM 中第 k 大的路径。我们计算完每一个儿子有多少个子串之后，从根开始找第一个和超过 k 的位置，然后减去之前的那些，继续递归着找就可以了，复杂度大概是 $O(|\sum||ans|)$ 其中 $|ans|$ 为查询的答案。显然后缀数组更加可做。

5.最小循环移位

貌似不用后缀自动机也是 $O(n)$ 的。这里介绍后缀自动机的做法。

需要倍增字符串（老套路），所以这个题变成了在 $S+S$ 的后缀自动机上，寻找最小的长度为 $|S|$ 的路径，这个路径代表的字符串一定是原来 S 的子串，所以这样找到的子串一定是原来的最小循环移位。所以从初始状态 t_{0} 开始，贪心地访问能访问的最小字符即可。

```
void find() {
    int tlas=1;
    for(int i=0;i<len;i++) {
        for(int j=0;j<26;j++) {
            if(dian[tlas].ch[j]) {
                tlas=dian[tlas].ch[j];
                printf("%c",'a'+j);
                break;
            }
        }
    }
}

int main() {
    scanf("%s",s);
    len=strlen(s);
    for(int i=0;i<len;i++) {
        add(s[i]-'a');
    }
    for(int i=0;i<len;i++) {
        add(s[i]-'a');
    }
    find();
    return 0;
}
```

6.给定一个文本串 T 多组询问，每次询问字符串 P 你需要回答 P 在字符串 T 中作为子串出现了多少次。

将模式串沿着后缀自动机跑，跑不下去答案为 0，如果跑完，答案就是该节点的终点集合大小。

预处理：对于自动机中的每个状态 v 处理 cnt_{v} 使之等于 $endpos(v)$ 集合的大小（也就是等价类中包含了多少个串）。显然这个值可以用后缀链接来算，后缀链接显然编号更小，于是为了保证计算加上的都是已经计算正确的，可以沿长度降序遍历 $cnt_{link(v)}+=cnt_{v}$ 这里我们把每个状态，如果不是通过复制创建的且不是初始状态 t_{0} 我们就讲它的 cnt 初始化为 1。（不是通过复制获得的状态，恰好有 $|T|$ 个，我们需要计算的是它们所对应位置的数量，所以他们的 cnt 赋值为 1，

其他的赋值为 0 ）。而 $cnt_{link(v)}+=cnt_{v}$ 的含义是，如果一个字符串 v 出现了 cnt_{v} 次，那么它的所有后缀也在这个位置结束，所以后缀链接那些字符串也在这里结束，所以需要加上 cnt_{v} 次。所以我们用 $O(T)$ 的时间计算出了所有状态的 cnt 值。

所以每次询问只需要找到 cnt_{t} 也就是这个字符串代表的状态，所以单次时间复杂度为 $O(|P|)$

7.给定一个文本串 T 多组查询，每次查询字符串 P 在 T 中第一次出现的位置（开头位置）。

我们对于每个状态，需要预处理 $firstpos$ 第一次出现这个状态的末端位置，也就是每一个 $endpos$ 集合中最小的元素）。我们需要分情况讨论：

当此状态是新创建的状态 cur 时，我们令 $firstpos(cur)=len(cur)-1$

当此状态是结点 q 复制到 $clone$ 时，我们令 $firstpos(clone)=firstpos(q)$

所以答案就是 $firstpos(t)-|P|+1$ t 是 P 末尾的状态，单次时间复杂度是 $O(|P|)$ 的。

8.查询模式串在文本串出现的所有位置

和上一问类似，我们为所有状态计算位置 $firstpos$ 设模式串为 T 在后缀自动机中对应状态为 t 显然 $firstpos(t)$ 是答案的一部分。我们显然需要找到所有可以通过后缀链接到达 t 的状态。我们存一下每个状态的后缀引用列表，然后从 t 结点 dfs 下去，把所有状态的 $firstpos$ 值都输出就可以了。这个复杂度是 $O(ans(P))$ 访问了多少个结点就是多少个答案，并且一个结点只会访问一次。但是这样子遇到复制出来的结点，他们的 $firstpos$ 是一样的，所以需要标记是否是复制出来的，如果是则不输出就可以了。

```
struct NODE {
    bool is_clone;
    map<int,int> ch;
    int len,fa,firstpos;
    vector<int> vec;
    NODE() {
        ch.clear();
        is_clone=len=fa=firstpos=0;
        vec.clear();
    }
}dian[MAXN<<1];

int add(int c,int id) {
    int p=las;
    int np=las=++tot;
    dian[np].len=dian[p].len+1;
    dian[np].firstpos=id;
    for(; p&&!dian[p].ch[c]; p=dian[p].fa)dian[p].ch[c]=np;
    if(!p)dian[np].fa=1;//以上为case 1
    else {
        int q=dian[p].ch[c];
        if(dian[q].len==dian[p].len+1)dian[np].fa=q;//以上为case 2
        else {
            int nq=++tot;
            dian[nq].is_clone=true;
            dian[nq]=dian[q];
```

```
            dian[nq].len=dian[p].len+1;
            dian[q].fa=dian[np].fa=nq;
            for(; p&&dian[p].ch[c]==q; p=dian[p].fa)dian[p].ch[c]=nq; //以上为case 3
        }
    }
    return dian[las].len-dian[dian[las].fa].len;
}

void dfs(int now,int length) {
    if(!dian[now].is_clone) printf("%d\n",dian[now].firstpos-length+1);
    for(int i=0;i<dian[now].vec.size();i++) {
        dfs(dian[now].vec[i],length);
    }
}

//然后main函数里就是把这个字符串的终止位置找到，然后从终止位置开始dfs就可以了。
```

From:
https://wiki.cvbbacm.com/ - **CVBB ACM Team**

Permanent link:
https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:%E7%8E%8B%E6%99%BA%E5%BD%AA:%E5%90%8E%E7%BC%80%E8%87%AA%E5%8A%A8%E6%9C%BA&rev=1627632503

Last update: **2021/07/30 16:08**