字符串

Hash

```
const int N=1e5+5;
const int mod=998244353,mod2=1e9+7;
const int base=13331,base2=131;
int p[N],p2[N];
void init(){
    p[0]=p2[0]=1;
    for(int i=1;i<N;i++){
        p[i]=p[i-1]*base%mod;
        p2[i]=p2[i-1]*base2%mod2;
    }
}
struct Hash{
    vector<int>h,h2;
    Hash():h(1),h2(1){}
    void push_back(char ch){
        h.push_back((h.back()*base+ch)%mod);
        h2.push_back((h2.back()*base2+ch)%mod2);
    }
    void pop_back(){
        h.pop_back();
        h2.pop_back();
    }
    void init(string s){
        for(auto i:s)push_back(i);
    }
    pair<int,int> get(int l,int r){
        int len=r-l+1;
        int v1=(h[r]-h[l-1]*p[len]%mod+mod)%mod;
        int v2=(h2[r]-h2[l-1]*p2[len]%mod2+mod2)%mod2;
        return {v1,v2};
    }
};
int Hashe(string s){
    int len=s.length();
    int ans=0;
    for(int i=0;i<s.length();i++){
        ans=(ans*base+s[i])%mod;
    }
    return ans;
}
```

Manacher

```
// O(n)处理以某个点为回文中心的最大回文串长度
// 回文中心对应的数组下标为：奇数长度:(1,3...2*i+1...2*n-1)   |   偶数长度:
(0,2...2*i...2*n)
vector<int> manacher(string s){
```

```
    string t="#";
    for(auto i:s){
        t+=i;
        t+='#';
    }
    int n=t.size();
    vector<int>r(n);
    for(int i=0,j=0;i<n;i++){
        if(2*j-i>=0 && j+r[j]>i)
            r[i]=min(r[2*j-i],j+r[j]-i);
        while(i-r[i]>=0 && i+r[i]<n && t[i-r[i]]==t[i+r[i]])
            r[i]++;
        if(i+r[i]>j+r[j])j=i;
    }
    for(auto&i:r)i--;
    return r;
}
```

KMP

```
int kmp[N],n;   // n 为模式串长度
string pat;     // 模式串
void kmp_init(){
    n=pat.length();
    pat=' '+pat;
    for(int i=2,j=0;i<=n;i++){
        while(j && pat[i]!=pat[j+1])
            j=kmp[j];
        if(pat[i]==pat[j+1])j++;
        kmp[i]=j;
    }
}
vector<int>match(string s){    // 出现匹配串的位置
    int len=s.length();
    s=' '+s;
    vector<int>ans;
    for(int i=1,j=0;i<=len;i++){
        while(j && s[i]!=pat[j+1])j=kmp[j];
        if(s[i]==pat[j+1])j++;
        if(j==n){
            ans.push_back(i-n+1);
            j=kmp[j];   // 出现串可以重叠
            // j=0  |   出现串不能重叠
        }
    }
    return ans;
}
```

扩展KMP（Z函数）

```
// z[i] 表示 s[i] 开头的后缀和 s 的lcp
// 求 s 和 t 的每一个后缀的lcp，把 s 和 t 拼起来跑 z 函数即可
vector<int>getZ(string s){
    int n=s.length();
```

```
    vector<int>z(n+1);
    z[0]=n;
    for(int i=1,j=1;i<n;i++){
        z[i]=max(0ll,min(j+z[j]-i , z[i-j]));
        while(i+z[i]<n && s[z[i]]==s[i+z[i]])
            z[i]++;
        if(i+z[i]>j+z[j])j=i;
    }
    return z;
}
```

失配树

```
const int N=1e6+5;
int kmp[N],n;   // n 为模式串长度
string pat;     // 模式串
vector<int>fail[N];     // 失配树
void kmp_init(){
    n=pat.length();
    pat=' '+pat;
    fail[0].push_back(1);
    for(int i=2,j=0;i<=n;i++){
        while(j && pat[i]!=pat[j+1])
            j=kmp[j];
        if(pat[i]==pat[j+1])j++;
        kmp[i]=j;
        fail[j].push_back(i);
    }
}
int siz[N],top[N],son[N],f[N],dep[N];
void dfs1(int x){
    siz[x]=1;
    dep[x]=dep[f[x]]+1;
    for(auto y:fail[x]){
        if(y==f[x])continue;
        f[y]=x;
        dfs1(y);
        siz[x]+=siz[y];
        // 根节点为 0，此处要特判 son[x]==0
        if(son[x]==0 || siz[y]>siz[son[x]])son[x]=y;
    }
}
void dfs2(int x,int id){
    top[x]=id;
    if(son[x])dfs2(son[x],id);
    for(auto y:fail[x]){
        if(y==f[x]||y==son[x])continue;
        dfs2(y,y);
    }
}
int lca(int u,int v){
    while(top[u]!=top[v]){
        if(dep[top[u]]<dep[top[v]])v=f[top[v]];
        else u=f[top[u]];
    }
```

```
    return (dep[u]<dep[v])?u:v;
}
int common_border(int x,int y){
    return lca(kmp[x],kmp[y]);
}
```

NTT 字符串匹配

```
// 完全匹配：定义一个匹配函数，判断 f(x) 是否为 0
// 部分匹配：对每种字符分别考虑，计算总失配次数是否满足条件
#define int long long
#define VI vector<int>
const int LG=21; //内存参数，约为log(多项式长度)      LG==21 -> 30MB (注意调节)
const int N=(1<<LG);    // 有用的，不能乱设
const int mod=998244353;
int qpow(int a,int b){
    int ans=1;
    while(b){
        if(b&1)ans=ans*a%mod;
        b>>=1;
        a=a*a%mod;
    }
    return ans;
}
int w[N],inv[N];
int cp=[]{
    w[N/2]=1;
    int s=qpow(3,(mod-1)/N);
    for(int i=N/2+1;i<N;i++){
        w[i]=s*w[i-1]%mod;
    }
    for(int i=N/2-1;i>0;i--){
        w[i]=w[i*2];
    }
    inv[1]=1;
    for(int i=2;i<N;i++)
        inv[i]=(mod-mod/i)*inv[mod%i]%mod;
    return 0;
}();
void dft(int*a,int n){
    for(int k=n/2;k;k>>=1)
        for(int i=0;i<n;i+=2*k)
            for(int j=0;j<k;j++){
                int t=a[i+j+k];
                a[i+j+k]=(a[i+j]-t+mod)*w[k+j]%mod;
                a[i+j]=(a[i+j]+t)%mod;
            }
}
void idft(int*a,int n){
    for(int k=1;k<n;k*=2)
        for(int i=0;i<n;i+=2*k)
            for(int j=0;j<k;j++){
                int u=a[i+j],v=a[i+j+k]*w[j+k]%mod;
                a[i+j+k]=(u-v+mod)%mod;
                a[i+j]=(u+v)%mod;
```

```
            }
    for(int i=0;i<n;i++)
        a[i]=a[i]*(mod-(mod-1)/n)%mod;
    reverse(a+1,a+n);
}
VI operator*(VI a,VI b){
    int n=a.size()+b.size()-1;
    int s=(1<<__lg(2*n-1));
    a.resize(s),b.resize(s);
    dft(a.data(),a.size());
    dft(b.data(),b.size());
    for(int i=0;i<s;i++){
        a[i]=a[i]*b[i]%mod;
    }
    idft(a.data(),a.size());
    a.resize(n);
    return a;
}

int ans[N],n,m;
string s1,s2;
void cal(char c){
    VI a(n+1,0),b(m+1,0);
    for(int i=1;i<=n;i++){
        if(s1[i]==c)a[i]=1;      // 表示该位置为C
        else a[i]=0;
    }
    for(int i=1;i<=m;i++){       // 表示该位置不为C
        if(s2[m+1-i]==c)b[i]=0;
        else b[i]=1;
    }
    a=a*b;
    for(int i=m+1;i<=n+1;i++)ans[i]+=a[i];
}
void solve(){
    cin>>s1>>s2;
    n=s1.length(),m=s2.length();
    for(int i=1;i<=n+m+1;i++)ans[i]=0;
    s1=' '+s1,s2=' '+s2;
    cal('A');        // 对每个字符分别匹配
    cal('T');
    cal('C');
    cal('G');
    int cnt=0;
    for(int i=m+1;i<=n+1;i++)cnt+=(ans[i]<=3);
    cout<<cnt<<endl;
}
```

最小表示法

```
int get_min(string s){
    int n=s.length();
    int i=0,j=1,k=0;
    while(i<n&&j<n){
        for(k=0;k<n;k++){
```

```
            if(s[(i+k)%n]>s[(j+k)%n]){
                i=i+k+1;
                if(i==j)i++;
                break;
            }
            else if(s[(i+k)%n]<s[(j+k)%n]){
                j=j+k+1;
                if(i==j)j++;
                break;
            }
        }
        if(k==n)break;
    }
    return min(i,j);    //返回首位下标
}
```

Lyndon 分解

```
// 位置编号为 1 -- n
vector<int>Lyndon(string s){
    int n=s.length();
    s=' '+s;
    vector<int>pos;     // 所有 Lyndon 串的结尾 (endpos)
    int now=1;
    while(now<=n){
        int i=now,j=now+1;
        while(j<=n && s[i]<=s[j]){
            if(s[i]<s[j])i=now;
            else i++;
            j++;
        }
        while(now<=i){
            pos.push_back(now+j-i-1);
            now+=j-i;
        }
    }
    return pos;
}
```

Runs

```
struct RunsGetter{
    Hash h;
    string s;
    int n;
    vector<tuple<int,int,int>>Runs;         // 储存串 s 的所有 Runs
    RunsGetter(string ss){
        s=ss;
        n=s.length();
        h.init(s);
        getRuns();
    }
    int lcp(int i,int j){           // 后缀 i 和 j 的 最长公共前缀
        if(s[i-1]!=s[j-1])return 0;
```

```
        int l=1,r=n-j+1;
        while(l<r){
            int mid=(l+r+1)>>1;
            if(h.get(i,i+mid-1)==h.get(j,j+mid-1))l=mid;
            else r=mid-1;
        }
        return l;
    }
    int lcp_r(int i,int j){          // 前缀 i 和 j 的 最长公共后缀
        if(s[i-1]!=s[j-1])return 0;
        int l=1,r=i;
        while(l<r){
            int mid=(l+r+1)>>1;
            if(h.get(i-mid+1,i)==h.get(j-mid+1,j))l=mid;
            else r=mid-1;
        }
        return l;
    }
    bool le(int l1,int r1,int l2,int r2){
        if(l1==l2)return r1<r2;
        int l=min({lcp(l1,l2), r1-l1+1, r2-l2+1});
        if(l1+l>r1 || l2+l>r2)return r1-l1<r2-l2;
        return s[l1+l-1]<s[l2+l-1];
    }
    int stk[N],top,ed[N];
    void lyndon(){
        top=0;
        for(int i=n;i>0;i--){
            stk[++top]=i;
            while(top>1 && le(i,stk[top],stk[top]+1,stk[top-1]))top--;
            ed[i]=stk[top];
        }
        for(int i=1;i<=n;i++){
            int j=ed[i],Lcs=lcp_r(i-1,j),Lcp=lcp(i,j+1);
            int l=i-Lcs,r=j+Lcp;
            if((r-l+1)/(j-i+1)>1)Runs.emplace_back(l,r,j-i+1);
        }
    }
    void getRuns(){
        lyndon();
        for(auto&i:s)i=(char)('z'-i+'a');
        lyndon();
        sort(Runs.begin(),Runs.end());
        Runs.erase(unique(Runs.begin(),Runs.end()),Runs.end());
    }
};
```

平方串

```
void solve(){
    RunsGetter getter(t);
    vector<tuple<int,int,int>>runs=getter.Runs;
    Hash hs;
    hs.init(t);
    map<pair<int,int>,int>mp;          //   [hs,cnt]   |   储存每个平方子串的出现次数
```

```
    vector<tuple<int,int,pair<int,int>>>vec;      // {endpos,len,hash}    |    储存所有平方子串
    for(auto&[l,r,p]:runs){
        for(int len=2*p;l+len-1<=r;len+=2*p){       // 枚举平方子串长度
            for(int L=l;L<l+p && L+len-1<=r;L++){
                int R=L+len-1;
                if(mp.find(hs.get(L,R))==mp.end()){
                    mp[hs.get(L,R)]=(r-R)/p+1;
                    vec.emplace_back(R,len,hs.get(L,R));
                }
                else mp[hs.get(L,R)]+=(r-R)/p+1;
            }
        }
    }
    // -------------------------------
    for(auto&[l,r,p]:runs){         // 找出所有本原平方串
        int len=2*p;
        for(int L=l;L<l+p && L+len-1<=r;L++){
            int R=L+len-1;
            if(mp.find(hs.get(L,R))==mp.end()){
                mp[hs.get(L,R)]=(r-R)/p+1;
                vec.emplace_back(R,len,hs.get(L,R));
            }
            else mp[hs.get(L,R)]+=(r-R)/p+1;
        }
    }
}
```

后缀数组

```
const int N=2e6+5;
// 开二倍空间
struct SA{
    int n;
    vector<int>sa,rk,h;
    // sa, rk 的下标为 [0,n-1], h 的下标为 [0,n-2], 需要 st 表时变换成 [1,n-1]
    // rk[i] 表示后缀i的排名，sa[i] 表示排名为i的后缀
    SA(){}
    SA(const string&s){
        n=s.length();
        sa.resize(n);
        rk.resize(n);
        h.resize(n-1);
        iota(sa.begin(),sa.end(),0);
        sort(sa.begin(),sa.end(),[&](int a,int b){
            return s[a]<s[b];
        });
        rk[sa[0]]=0;
        for(int i=1;i<n;i++){
            rk[sa[i]]=rk[sa[i-1]]+(s[sa[i]]!=s[sa[i-1]]);
        }
        int k=1;
        vector<int>tmp,cnt(n);
        tmp.reserve(n);
        while(rk[sa[n-1]]<n-1){
```

```
            tmp.clear();
            for(int i=0;i<k;i++)tmp.push_back(n-k+i);
            for(auto i:sa){
                if(i>=k)tmp.push_back(i-k);
            }
            fill(cnt.begin(),cnt.end(),0);
            for(int i=0;i<n;i++)cnt[rk[i]]++;
            for(int i=1;i<n;i++)cnt[i]+=cnt[i-1];
            for(int i=n-1;i>=0;i--){
                sa[--cnt[rk[tmp[i]]]]=tmp[i];
            }
            swap(rk,tmp);
            rk[sa[0]]=0;
            for(int i=1;i<n;i++){
                rk[sa[i]]=rk[sa[i-1]]+(tmp[sa[i-1]]<tmp[sa[i]] || sa[i-1]+k==n
|| tmp[sa[i-1]+k]<tmp[sa[i]+k]);
            }
            k*=2;
        }
        // 求 height 数组
        for(int i=0,j=0;i<n;i++){
            if(rk[i]==0)j=0;
            else{
                j-=(j>0);
                while(i+j<n && sa[rk[i]-1]+j<n && s[i+j]==s[sa[rk[i]-1]+j])
                    j++;
                h[rk[i]-1]=j;
            }
        }
    }
};

int st[N][20],Log[N],n;
SA sa;
void st_init(){
    // 注意定义域 [1,n-1]
    for(int i=1;i<=n-1;i++){
        st[i][0]=sa.h[i-1];
    }
    int mx=Log[n];
    for(int j=1;j<=mx;j++){
        for(int i=1;i<=n-1;i++){
            st[i][j]=min(st[i][j-1],st[i+(1<<(j-1))][j-1]);
        }
    }
}
int lcp(int l,int r){       // 后缀 [l,n] 和 后缀 [r,n] 的 lcp, l,r 的域为[1,n]
    if(l==r)return n+1-l;
    l=sa.rk[l-1],r=sa.rk[r-1];
    if(l>r)swap(l,r);
    int k=Log[r-l];
    return min(st[l+1][k],st[r-(1<<k)+1][k]);
}
void log_init(){
    Log[0]=-1;
```

```
    for(int i=1;i<N;i++){
        if(i&(i-1))Log[i]=Log[i-1];
        else Log[i]=Log[i-1]+1;
    }
}
void solve(){
    string s;
    cin>>s;
    sa=SA(s);
    for(int i=1;i<=n;i++)cout<<sa.sa[i-1]+1<<' ';
}
```

自动机

kmp 自动机

```
int nxt[N][26],kmp[N];      // nxt[i][j] 表示从 i 开始走 j 字符可以和 s 匹配的最长
border
void solve(){
    string s;
    cin>>s;
    int n=s.length();
    s=' '+s;
    for(int i=2,j=0;i<=n;i++){
        while(j && s[i]!=s[j+1])j=kmp[j];
        if(s[i]==s[j+1])j++;
        kmp[i]=j;
    }
    for(int i=0;i<=n;i++){
        for(int j=0;j<26;j++){
            if(i+1<=n && s[i+1]-'a'==j)nxt[i][j]=i+1;
            else nxt[i][j]=nxt[kmp[i]][j];
        }
    }
    int q;
    cin>>q;
    while(q--){
        string t;
        cin>>t;     // 把 t 拼在 s 后面跑 kmp
        int m=t.length();
        t=' '+t;
        for(int i=n+1,j=kmp[n];i<=n+m;i++){
            while(j && (j>=n && t[i-n]!=t[j+1-n]))j=kmp[j];
            if(j<n)j=nxt[j][t[i-n]-'a'];
            else if(t[i-n]==t[j+1-n])j++;
            kmp[i]=j;
        }
    }
}
```

AC自动机（ACAM）

```
struct ACAM{
    static const int M=26;
    struct Node{
        int len;
        int link;
        int cnt;
        array<int,M>nxt;
        Node():len(0),link(0),cnt(0),nxt{}{}
    };
    vector<Node>t;
    ACAM(){init();}
    void init(){
        t.clear();
        t.assign(2,Node());
        t[0].nxt.fill(1);
        t[0].len=-1;
    }
    int newNode(){
        t.emplace_back();
        return t.size()-1;
    }
    int ins(const string&a){
        int p=1;
        for(auto i:a){
            int x=i-'a';
            if(t[p].nxt[x]==0){
                t[p].nxt[x]=newNode();
                t[t[p].nxt[x]].len=t[p].len+1;
            }
            p=t[p].nxt[x];
        }
        return p;
    }
    void work(){
        queue<int>q;
        q.push(1);
        while(!q.empty()){
            int x=q.front();
            q.pop();
            for(int i=0;i<M;i++){
                if(t[x].nxt[i]==0){
                    t[x].nxt[i]=t[t[x].link].nxt[i];
                }
                else{
                    t[t[x].nxt[i]].link=t[t[x].link].nxt[i];
                    q.push(t[x].nxt[i]);
                }
            }
        }
    }
    void query(const string& s){
        int p=1;
        for(auto i:s){
```

```
            int x=i-'a';
            p=t[p].nxt[x];
            t[p].cnt++;
        }
    }
    int nxt(int p,int x){
        return t[p].nxt[x];
    }
    int link(int p){
        return t[p].link;
    }
    int len(int p){
        return t[p].len;
    }
    int size(){
        return t.size()-1;
    }
    int cnt(int p){
        return t[p].cnt;
    }
};
int pos[N];
vector<int>G[N];
ACAM ac;
void dfs(int x){
    for(auto y:G[x]){
        dfs(y);
        ac.t[x].cnt+=ac.t[y].cnt;
    }
}
void solve(){
    int n;
    cin>>n;
    ac.init();
    for(int i=1;i<=n;i++){
        string s;
        cin>>s;
        pos[i]=ac.ins(s);
    }
    ac.work();
    int m=ac.size();
    for(int i=1;i<=m;i++){          // 建 fail 树
        G[ac.link(i)].push_back(i);
    }
    string s;
    cin>>s;
    ac.query(s);
    dfs(1);
    for(int i=1;i<=n;i++)cout<<ac.cnt(pos[i])<<endl;
}
```

子序列自动机

```
// n 为长度，m 为值域
// m 较小时 二维数组维护，m 较大时 可持久化线段树维护
const int N=1e5+5,M=26;
int n,m,nxt[N][M],a[N];
void init(){
    for(int i=n;i>=1;i--){
        for(int j=0;j<m;j++){
            nxt[i-1][j]=nxt[i][j];
        }
        nxt[i-1][a[i]]=i;
    }
}

// 可持久化线段树
int cnt=0,rt[N],a[N],n,m,q;
struct Segment{
    int ls,rs;
    int nxt;
#define mid ((l+r)>>1)
}seg[10*N];
void ins(int&p,int pre,int l,int r,int x,int y){
    seg[++cnt]=seg[pre];
    p=cnt;
    if(l==r){
        seg[p].nxt=y;
        return;
    }
    if(x<=mid)ins(seg[p].ls,seg[pre].ls,l,mid,x,y);
    else ins(seg[p].rs,seg[pre].rs,mid+1,r,x,y);
}
int query(int p,int l,int r,int x){
    if(p==0)return 0;
    if(l==r)return seg[p].nxt;
    if(x<=mid)return query(seg[p].ls,l,mid,x);
    else return query(seg[p].rs,mid+1,r,x);
}
int get(vector<int>v){      // v 为要匹配的子序列
    int pos=0;
    for(auto i:v){
        pos=query(rt[pos],1,m,i);
        if(pos==0)return 0;
    }
    return pos;
}
void solve(){
    cin>>n>>m>>q;
    for(int i=1;i<=n;i++)cin>>a[i];
    for(int i=n;i>=1;i--){      // 从后往前，更新这一位的 nxt 指针
        ins(rt[i-1],rt[i],1,m,a[i],i);
    }
    while(q--){
        vector<int>v;
        if(get(v))cout<<"Yes"<<endl;
```

```
        else cout<<"No"<<endl;
    }
}
```

回文自动机（PAM）

```
const int N=1e6+5;
string s;
int n;      // s.length()
int tr[N][26],fail[N],len[N],num[N];
int cnt,last;
void init(){
    cnt=1;
    fail[0]=1,fail[1]=1;
    len[0]=0,len[1]=-1;
}
int getfail(int x,int pos){
    while(pos-len[x]-1<0||s[pos]!=s[pos-len[x]-1])x=fail[x];
    return x;
}
void ins(int w,int pos){
    int x=getfail(last,pos);
    if(!tr[x][w]){
        len[++cnt]=len[x]+2;
        int tmp=getfail(fail[x],pos);
        fail[cnt]=tr[tmp][w];
        num[cnt]=num[fail[cnt]]+1;
        tr[x][w]=cnt;
    }
    last=tr[x][w];
}
void solve(){
    cin>>s;
    n=s.length();
    s=' '+s;
    init();
    int ans;
    for(int i=1;i<=n;i++){
        ins(s[i]-'a',i);
        ans=num[last];
        cout<<ans<<' ';     //每一位结尾的回文串个数
    }
    cout<<cnt-1<<endl;      //本质不同的回文串个数
    return;
}
```

后缀自动机（SAM）

```
// 每个 endpos 等价类处串的长度是一个区间，为 [len(link(p))+1,len(p)]，且是当前串的后缀
// link(p) 是 p 的后缀，endpos(p) 是 endpos(link(p)) 的子集
// 状态数为 2n，转移数为 3n
// 每次加入字符，对 本质不同子串个数 的增量为 len(p)-len(link(p))
int fa[N][20];      // fail 树倍增数组
int ed[N];          // 前缀的结束位置
```

```cpp
int cnt[N];          // 每个子串的出现次数，通过在 parent 树上 dfs 统计
struct SAM{
    struct Node{
        int len;
        int link;        // 表示最长的 endpos 不同的后缀
        array<int,26>nxt;        // 表示子串
        //map<int,int>mp;        | 值域较大时，用map存
        Node():len{},link{},nxt{}{}
    };
    vector<Node>t;
    SAM(){init();}
    void init(){
        t.assign(2,Node());
        t[0].nxt.fill(1);                // 值域较大时，fill改成extend时特判
        t[0].len=-1;
    }
    int newNode(){
        t.emplace_back();
        return t.size()-1;
    }
    int extend(int p,int c){
        if(t[p].nxt[c] || p==0){
            // if(!p)t[p].nxt[c]=1;      |    值域较大时，需要特判p==0
            int q=t[p].nxt[c];
            if(t[q].len==t[p].len+1){
                return q;
            }
            int r=newNode();
            t[r].len=t[p].len+1;
            t[r].link=t[q].link;
            t[r].nxt=t[q].nxt;
            t[q].link=r;
            while (t[p].nxt[c]==q){
                t[p].nxt[c]=r;
                p=t[p].link;
            }
            return r;
        }
        int cur=newNode();
        t[cur].len=t[p].len+1;
        while(!t[p].nxt[c]){
            t[p].nxt[c]=cur;
            p=t[p].link;
        }
        t[cur].link=extend(p,c);
        cnt[cur]=1;          // 新增状态，出现次数+1
        return cur;
    }
    int nxt(int p,int x){
        return t[p].nxt[x];
    }
    int link(int p){
        return t[p].link;
    }
    int len(int p){
```

```
        return t[p].len;
    }
    int size(){
        return t.size();
    }
};


vector<int>G[N];    // parent 树
void dfs(int x){
    for(int i=1;i<=19;i++){
        fa[x][i]=fa[fa[x][i-1]][i-1];
    }
    for(auto y:G[x]){
        dfs(y);
        cnt[x]+=cnt[y];
    }
}
SAM sam;
int get(int l,int r){       // 子串 [l,r] 的出现次数
    int now=ed[r];
    for(int i=19;i>=0;i--){
        if(fa[now][i]>1 && sam.len(fa[now][i])>=r-l+1)
            now=fa[now][i];
    }
    return cnt[now];
}
void solve(){
    string s;
    cin>>s;
    int p=1,n=s.length();       // 1 为根节点
    for(int i=1;i<=n;i++){
        p=sam.extend(p,s[i-1]-'a');
        ed[i]=p;
    }
    n=sam.size()-1;         // 节点个数(包括根节点1)
    for(int i=1;i<=n;i++){
        G[sam.link(i)].push_back(i);
        fa[i][0]=sam.link(i);
    }
    dfs(1);
}
```