> @zeroy.site

正向表

```
int tot=0,h[M];
struct edge{
    int nxt;
    int to,cost;
}G[3*M];
void add(int a,int b,int c){
    G[++tot]=(edge){h[a],b,c};
    h[a]=tot;
}
//图论题存图用的
//遍历：
for(int i=h[x];i;i=G[i].nxt){
    int u=G[i].to,v=G[i].cost;
    //...
}
```

基础数据结构

树状数组

```
void Add(int x,int d){
    while(x<=n){
        C[x]+=d;
        x+=(x&-x);
    }
}
int Sum(int x){
    int res=0;
    while(x){
        res+=C[x];
        x-=(x&-x);
    }
    return res;
}
//在线单点加值，查询前缀和，单次操作复杂度均为O(logn)
```

ST表

预处理

```
void Init_RMQ(int n){
    for(int i=1;i<=n;i++)f[i][0]=A[i];
    lg2[1]=0;
    for(int i=2;i<=n;i++)lg2[i]=lg2[i>>1]+1;
    for(int j=1;(1<<j)<=n;j++)
        for(int i=1;i+(1<<j)-1<=n;i++)
            f[i][j]=max(f[i][j-1],f[i+(1<<(j-1))][j-1]);
}
```

查询

```
int query(int l,int r){
    int k=lg2[r-l+1];
    return max(f[l][k],f[r-(1<<k)+1][k]);
}
//预处理复杂度为O(nlogn),单次查询复杂度为O(1)
```

线段树

```
struct Segment_tree{
    #define fa tree[p]
    #define lson tree[p<<1]
    #define rson tree[p<<1|1]
    struct node{
        int l,r;
        LL add;//懒标记
        LL sum;
        int len(){return r-l+1;}
    }tree[M<<2];
    void up(int p){
        fa.sum=lson.sum+rson.sum;
    }
    void down(int p){
        if(fa.add==0)return;
        lson.add+=fa.add;
        lson.sum+=fa.add*lson.len();
        rson.add+=fa.add;
        rson.sum+=fa.add*rson.len();
        fa.add=0;
    }
    void build(int l,int r,int p){
        fa.l=l;fa.r=r;fa.sum=fa.add=0;
        if(l==r){fa.sum=A[l];return;}
        int mid=(l+r)>>1;
        build(l,mid,p<<1);
        build(mid+1,r,p<<1|1);
        up(p);
    }
    void update(int l,int r,LL d,int p){
        if(fa.l==l&&fa.r==r){
            fa.sum+=1LL*fa.len()*d;
            fa.add+=d;
            return;
        }
        int mid=(fa.l+fa.r)>>1;
        down(p);
```

```
        if(r<=mid)update(l,r,d,p<<1);
        else if(l>mid)update(l,r,d,p<<1|1);
        else {
            update(l,mid,d,p<<1);
            update(mid+1,r,d,p<<1|1);
        }
        up(p);
    }
    LL query(int l,int r,int p){
        if(fa.l==l&&fa.r==r)return fa.sum;
        down(p);
        int mid=(fa.l+fa.r)>>1;
        if(r<=mid)return query(l,r,p<<1);
        else if(l>mid)return query(l,r,p<<1|1);
        return query(l,mid,p<<1)+query(mid+1,r,p<<1|1);
    }
}T;
//区间加减，区间查询和，单次操作复杂度均为O(logn)
//可实现的变种：区间乘积，区间染色问题等等。
```

并查集

```
int getfa(int x){
    return fa[x]==x?x:fa[x]=getfa(fa[x]);
}
//初始状态下所有fa[x]=x
void Union(int x,int y){//两个集合合并
    fa[getfa(x)]=getfa(y);
}
```

树上算法

LCA

跳重链

```
void dfs(int x,int f,int d){
    dep[x]=d;
    sz[x]=1;
    fa[x]=f;son[x]=0;
    for(int i=0;i<G[x].size();i++){
        int u=G[x][i];
        if(u==f)continue;
        dfs(u,x,d+1);
        if(sz[u]>sz[son[x]])son[x]=u;
        sz[x]+=sz[u];
    }
}
void dfs_top(int x,int tp){
    top[x]=tp;
    if(son[x])dfs_top(son[x],tp);
    for(int i=0;i<G[x].size();i++){
        int u=G[x][i];
        if(u==fa[x]||u==son[x])continue;
```

```
        dfs_top(u,u);
    }
}
int LCA(int a,int b){
    while(top[a]!=top[b]){
        if(dep[top[a]]>dep[top[b]])a=fa[top[a]];
        else b=fa[top[b]];
    }
    return dep[a]>dep[b]?b:a;
}
```

倍增

```
void dfs(int x,int f,int d){
    dep[x]=d;
    fa[x][0]=f;
    for(int i=0;i<G[x].size();i++){
        int u=G[x][i];
        if(u==f)continue;
        dfs(u,x,d+1);
    }
}
int LCA(int a,int b){
    if(dep[a]>dep[b])swap(a,b);
    int step=dep[b]-dep[a];
    for(int i=19;i>=0;i--)
        if(step&1<<i)b=fa[b][i];
    if(a==b)return a;
    for(int i=19;i>=0;i--)
        if(fa[a][i]!=fa[b][i])a=fa[a][i],b=fa[b][i];
    return fa[a][0];
}
for(int j=1;j<=19;j++)
    for(int i=1;i<=n;i++)
        fa[i][j]=fa[fa[i][j-1]][j-1];
```

树链剖分

寻找重儿子

```
void dfs(int x,int f,int d){
    dep[x]=d;fa[x]=f;sz[x]=1;son[x]=0;
    for(int i=h[x];i;i=G[i].nxt){
        int u=G[i].to;
        if(u==f)continue;
        dfs(u,x,d+1);
        if(sz[u]>sz[son[x]])son[x]=u;
        sz[x]+=sz[u];
    }
}
```

处理top

```
void dfs_top(int x,int tp){
    top[x]=tp;ID[x]=++tt;ln[tt]=x;
    if(son[x])dfs_top(son[x],tp);
    for(int i=h[x];i;i=G[i].nxt){
        int u=G[i].to;
        if(u==son[x]||u==fa[x])continue;
        dfs_top(u,u);
    }
}
```

query

```
while(top[u]!=top[v]){
    if(dep[top[u]]>dep[top[v]]){
        query(ID[top[u]],ID[u],1);
        u=fa[top[u]];
    }
    else {
        query(ID[top[v]],ID[v],1);
        v=fa[top[v]];
    }
}
if(dep[u]>dep[v])query(ID[v],ID[u],1);
else query(ID[u],ID[v],1);
```

数学

扩展欧几里得算法

```
void exgcd(ll a,ll b,ll &d,ll &x,ll &y){
    if(!b){d=a;x=1;y=0;return;}
    exgcd(b,a%b,d,y,x);y-=a/b*x;
}
```

逆元

```
int inv(int a){
    ll x,y,d;
    exgcd(a,m,d,x,y);
    return (x%MOD+MOD)%MOD;
}
//当a与mod互质时也可使用费马小定理，qkpow(a,mod-2)
int qkpow(int a,int b){
    int res=1;
    while(b){
        if(b&1)res=res*a%mod;
        a=a*a%mod;
        b>>=1;
    }
    return res;
}
//注意不要爆int或者long long
```

线性筛逆元

```
void Init(){
    fac[0]=1;rev[1]=1;
    for(int i=1;i<=n;i++)
        fac[i]=(fac[i-1]*i)%MOD;
    for(int i=2;i<=n;i++)//线性筛逆元
        rev[i]=(MOD-MOD/i)*rev[MOD%i]%MOD;
}
```

FFT

```
struct Complex{
    double x,y;
    Complex(){}
    Complex(double _x,double _y):x(_x),y(_y){}
    Complex operator + (const Complex &res) const{
        return (Complex){x+res.x,y+res.y};
    }
    Complex operator - (const Complex &res) const{
        return (Complex){x-res.x,y-res.y};
    }
    Complex operator * (const Complex &res) const{
        return (Complex){x*res.x-y*res.y,x*res.y+y*res.x};
    }
};
Complex A[M],B[M];
double pi=acos(-1.0);
void FFT(Complex *y,int n,int f){
    if(n==1)return;
    Complex L[n>>1],R[n>>1];
    for(int i=0;i<n;i+=2)L[i>>1]=y[i],R[i>>1]=y[i+1];
    FFT(L,n>>1,f);FFT(R,n>>1,f);
    Complex wn(cos(2*pi/n),f*sin(2*pi/n)),w(1,0);
    for(int i=0;i<(n>>1);i++,w=w*wn){
        y[i]=L[i]+w*R[i];
        y[i+(n>>1)]=L[i]-w*R[i];
    }
}
double q[M],b[M],C[M];
int main(){
    //...
    int nn=n,mm=n;
    mm+=nn;
    for(nn=1;nn<=mm;nn<<=1);
    FFT(A,nn,1);FFT(B,nn,1);
    for(int i=0;i<=nn;i++)A[i]=A[i]*B[i];
    FFT(A,nn,-1);
    //...
}
```

字符串算法

KMP

给出两个字符串S1和S2，若S1的区间[l,r]与S2完全相同，则称S2在S1中出现了，其出现位置为l。

现在请你求出S2在S1中所有出现的位置。

定义一个字符串S的border为S的一个非S本身的子串t，满足t既是S的前缀，又是S的后缀。

对于S2，你还需要求出对于每个前缀S'的最长border t'的长度。

```
#include<bits/stdc++.h>
#define M 1000005
using namespace std;
char s1[M],s2[M];
int f[M];
void getf(char *s,int l){
    f[0]=f[1]=0;
    for(int i=1;i<l;i++){
        int j=f[i];
        while(j&&s[i]!=s[j])j=f[j];
        if(s[i]==s[j])j++;
        f[i+1]=j;
    }
}
int main(){
    scanf("%s%s",s1,s2);
    int l1=strlen(s1);
    int l2=strlen(s2);
    getf(s2,l2);
    for(int i=0,j=0;i<l1;i++){
        while(j&&s2[j]!=s1[i])j=f[j];
        if(s2[j]==s1[i])j++;
        if(j==l2)
            printf("%d\n",i-l2+2);
    }
    for(int i=1;i<=l2;i++)
        printf("%d ",f[i]);
    return 0;
}
```