多项式 2

FFT

算法简介

$O(n\log n)$ 时间实现多项式点值表示法与系数表示法之间的转化，主要用于加速多项式乘法。

算法实现

假设 $f(x)$ 为 $n-1$ 次多项式，且 n 是 2 的幂次。(如果不满足条件可以将高次系数视为 0)

快速傅里叶变换

考虑如何实现 $O(n\log n)$ 时间根据多项式系数表示法得到多项式点值表示法。

设 $f(x)=a_0+a_1x+a_2x^2+\cdots a_{n-1}x^{n-1}$

构造辅助函数 $g(x)=a_0+a_2x+\cdots a_{n-2}x^{\frac n2-1},h(x)=a_1+a_3x+\cdots a_{n-1}x^{\frac n2-1}$

于是有 $f(x)=g\left(x^2\right)+xh\left(x^2\right)$

记 $\omega_n=\cos \frac {2\pi}n+\sin \frac {2\pi}ni$将 $x=\omega_n^k,x=\omega_n^{k+\frac n2}(k=0,1\cdots \frac n2-1)$ 代入，有

$$f(\omega_n^k)=g(\omega_n^{2k})+\omega_n^kh(\omega_n^{2k})=g(\omega_{\frac n2}^k)+\omega_n^kh(\omega_{\frac n2}^k)$$

$$f(\omega_n^{k+\frac n2})=g(\omega_n^{2k+n})+\omega_n^{k+\frac n2}h(\omega_n^{2k+n})=g(\omega_{\frac n2}^k)-\omega_n^kh(\omega_{\frac n2}^k)$$

于是根据 $g(x)$ 的点值表示法 $\{\left(\omega_{\frac n2}^k,g(\omega_{\frac n2}^k)\right)\}$ 以及 $h(x)$ 的点值表示法 $\{\left(\omega_{\frac n2}^k,h(\omega_{\frac n2}^k)\right)\}$

可以 $O(n)$ 时间计算出 $f(x)$ 的点值表示法 $\{\left(\omega_n^k,f(\omega_n^k)\right)|k=0,1\cdots n-1\}$

于是利用分治算法，即可 $O(n\log n)$ 解决上述问题。

递归边界为当分治到多项式只剩下一个常数项时，多项式点值表示法即为系数表示法，于是直接返回。

快速傅里叶逆变换

考虑如何实现 $O(n\log n)$ 时间根据多项式系数点值法得到多项式系数表示法。

记 $y_i=f(\omega_n^i)$ 假设 $f(x)$ 系数表示法为 ${a_0,a_1\cdots a_{n-1}}$ 点值表示法为 $\{(\omega_n^0,y_0),(\omega_n^1,y_1)\cdots (\omega_n^{n-1},y_{n-1})\}$

构造多项式 $A(x)=\sum_{i=0}^{n-1}y_ix^i$ 将 $x=\omega_n^{-k}(k=0,1\cdots n-1)$ 代入，有

$$A\left(\omega_n^{-k}\right)=\sum_{i=0}^{n-1}\omega_n^{-ki}\sum_{j=0}^{n-1} a_j\omega_n^{ij}=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_j\omega_n^{i(j-k)}=\sum_{j=0}^{n-1}a_j\sum_{i=0}^{n-1}\omega_n^{i(j-k)}$$

如果 $j=k$ 则 $\sum_{i=0}^{n-1}\omega_n^{i(j-k)}=\sum_{i=0}^{n-1}1=n$

如果 $j\neq k$ 则 $\sum_{i=0}^{n-1}\omega_n^{i(j-k)}=\frac{\omega_n^{n(j-k)}-1}{\omega_n^{(j-k)}-1}=\frac{1-1}{\omega_n^{(j-k)}-1}=0$

于是有 $A\left(\omega_n^{-k}\right)=na_k$ 考虑快速傅里叶变换计算 $A(x)$ 的点值表示法即可快速解决上述问题。

递归版 FFT 板子

```
complex temp[MAXN<<2];
void FFT(complex *f,int n,int type){//type=1为正变换 type=-1为逆变换，逆变换最终结果需要除以n
    if(n==1)return;
    int m=n>>1;
    memcpy(temp,f,sizeof(complex)*n);
    for(int i=0;i<n;i+=2)
    f[i>>1]=temp[i],f[(i>>1)+m]=temp[i+1];
    complex *f1=f,*f2=f+m;
    FFT(f1,m,type);FFT(f2,m,type);
    complex cur(1.0,0.0),w(cos(2.0*pi/n),type*sin(2.0*pi/n));
    for(int i=0;i<m;i++){
        temp[i]=f1[i]+cur*f2[i];
        temp[i+m]=f1[i]-cur*f2[i];
        cur=cur*w;
    }
    memcpy(f,temp,sizeof(complex)*n);
}
```

算法优化

蝴蝶变换

考虑倍增模拟分治过程，于是需要调整运算顺序。观察下面分治过程

$$\{x_0,x_1,x_2,x_3,x_4,x_5,x_6,x_7\}$$

$$\{x_0,x_2,x_4,x_6\},\{x_1,x_3,x_5,x_7\}$$

$$\{x_0,x_4\},\{x_2,x_6\},\{x_1,x_5\},\{x_3,x_7\}$$

$$\{x_0\},\{x_4\},\{x_2\},\{x_6\},\{x_1\},\{x_5\},\{x_3\},\{x_7\}$$

发现将 x_i 的下标用二进制表示，然后将其翻转就可以得到 x_i 的最终位置。例如 $3=011,6=110$，于是 x_3 与 x_6 位置互换。

非递归版 FFT 板子

```
int rev[MAXN<<2];
int build(int k){
    int n,pos=0;
    while((1<<pos)<=k)pos++;
    n=1<<pos;
    _for(i,0,n)rev[i]=(rev[i>>1]>>1)|((i&1)<<(pos-1));
    return n;
}
void FFT(complex *f,int n,int type){
    _for(i,0,n)if(i<rev[i])
    swap(f[i],f[rev[i]]);
    complex t1,t2;
    for(int i=1;i<n;i<<=1){
        complex w(cos(pi/i),type*sin(pi/i));
        for(int j=0;j<n;j+=(i<<1)){
            complex cur(1.0,0.0);
            _for(k,j,j+i){
                t1=f[k],t2=cur*f[k+i];
                f[k]=t1+t2,f[k+i]=t1-t2;
                cur=cur*w;
            }
        }
    }
    if(type==-1)_for(i,0,n)
    f[i].x/=n;
}
```

算法练习

洛谷p3803

题意

给定两个多项式的系数表示法，求两个多项式乘积的系数表示法。

题解

记这两个多项式为 $g(x)=b_0+b_1x+\cdots b_nx^n,h(x)=c_0+c_1x+\cdots c_mx^m$所求多项式为 $f(x)=a_0+a_1x+\cdots a_{n+m}x^{n+m}$

设 s 是 2 的幂次且 $\frac s2\le n+m\lt s$

对 $g(x)=b_0+b_1x+\cdots b_nx^n+0x^{n+1}\cdots +0x^{s-1}$ 套用快速傅里叶变换求出点值表示法 $\{(w_s^0,g_0)\cdots (w_s^{s-1},g_{s-1})\}$

同样对 $h(x)$ 进行上述操作，于是得到 $f(x)$ 的点值表示法 $\{(w_s^0,g_0h_0)\cdots (w_s^{s-1},g_{s-1}h_{s-1})\}$

再根据快速傅里叶逆变换，即可得到 $f(x)$ 的系数表示法，时间复杂度 $O(s\log s)$

```
const int MAXN=1e6+5;
const double pi=acos(-1.0);
struct complex{
    double x,y;
    complex(double x=0.0,double y=0.0):x(x),y(y){}
    complex operator + (const complex &b){
        return complex(x+b.x,y+b.y);
    }
    complex operator - (const complex &b){
        return complex(x-b.x,y-b.y);
    }
    complex operator * (const complex &b){
        return complex(x*b.x-y*b.y,x*b.y+y*b.x);
    }
}a[MAXN<<2],b[MAXN<<2];
int rev[MAXN<<2];
int build(int k){
    int n,pos=0;
    while((1<<pos)<=k)pos++;
    n=1<<pos;
    _for(i,0,n)rev[i]=(rev[i>>1]>>1)|((i&1)<<(pos-1));
    return n;
}
void FFT(complex *f,int n,int type){
    _for(i,0,n)if(i<rev[i])
    swap(f[i],f[rev[i]]);
    complex t1,t2;
    for(int i=1;i<n;i<<=1){
        complex w(cos(pi/i),type*sin(pi/i));
        for(int j=0;j<n;j+=(i<<1)){
            complex cur(1.0,0.0);
            _for(k,j,j+i){
                t1=f[k],t2=cur*f[k+i];
                f[k]=t1+t2,f[k+i]=t1-t2;
                cur=cur*w;
            }
        }
```

```
    }
    if(type==-1)_for(i,0,n)
    f[i].x/=n;
}
int main()
{
    int n1=read_int(),n2=read_int(),n=build(n1+n2);
    _rep(i,0,n1)
    a[i].x=read_int();
    _rep(i,0,n2)
    b[i].x=read_int();
    FFT(a,n,1);FFT(b,n,1);
    _for(i,0,n)
    a[i]=a[i]*b[i];
    FFT(a,n,-1);
    _rep(i,0,n1+n2)
    space((int)(a[i].x+0.5));
    return 0;
}
```

优化

考虑式子 $\left(g(x)+h(x)i\right)^2=g^2(x)-h^2(x)+2g(x)h(x)i=g^2(x)-h^2(x)+2f(x)i$

于是将 $g(x)$ 作为实部，$h(x)$ 作为虚部，两次 FFT 计算出 $\left(g(x)+h(x)i\right)^2$ 表达式系数，即可得到 $f(x)$ 系数。

```
const int MAXN=1e6+5;
const double pi=acos(-1.0);
struct complex{
    double x,y;
    complex(double x=0.0,double y=0.0):x(x),y(y){}
    complex operator + (const complex &b){
        return complex(x+b.x,y+b.y);
    }
    complex operator - (const complex &b){
        return complex(x-b.x,y-b.y);
    }
    complex operator * (const complex &b){
        return complex(x*b.x-y*b.y,x*b.y+y*b.x);
    }
}a[MAXN<<2];
int rev[MAXN<<2];
int build(int k){
    int n,pos=0;
    while((1<<pos)<=k)pos++;
    n=1<<pos;
    _for(i,0,n)rev[i]=(rev[i>>1]>>1)|((i&1)<<(pos-1));
    return n;
```

```
}
void FFT(complex *f,int n,int type){
    _for(i,0,n)if(i<rev[i])
    swap(f[i],f[rev[i]]);
    complex t1,t2;
    for(int i=1;i<n;i<<=1){
        complex w(cos(pi/i),type*sin(pi/i));
        for(int j=0;j<n;j+=(i<<1)){
            complex cur(1.0,0.0);
            _for(k,j,j+i){
                t1=f[k],t2=cur*f[k+i];
                f[k]=t1+t2,f[k+i]=t1-t2;
                cur=cur*w;
            }
        }
    }
    if(type==-1)_for(i,0,n)
    f[i].y/=n;
}
int main()
{
    int n1=read_int(),n2=read_int(),n=build(n1+n2);
    _rep(i,0,n1)
    a[i].x=read_int();
    _rep(i,0,n2)
    a[i].y=read_int();
    FFT(a,n,1);
    _for(i,0,n)
    a[i]=a[i]*a[i];
    FFT(a,n,-1);
    _rep(i,0,n1+n2)
    space((int)(a[i].y/2+0.5));
    return 0;
}
```

应用

洛谷p1919

进行高精度乘法时，将 10 进制数视为 $x=10$ 多项式，进行多项式乘法后考虑进位情况即可。

```
const int MAXN=1e6+5;
const double pi=acos(-1.0);
struct complex{
    double x,y;
    complex(double x=0.0,double y=0.0):x(x),y(y){}
    complex operator + (const complex &b){
        return complex(x+b.x,y+b.y);
```

```
    }
    complex operator - (const complex &b){
        return complex(x-b.x,y-b.y);
    }
    complex operator * (const complex &b){
        return complex(x*b.x-y*b.y,x*b.y+y*b.x);
    }
}a[MAXN<<2],b[MAXN<<2];
int rev[MAXN<<2];
int build(int k){
    int n,pos=0;
    while((1<<pos)<=k)pos++;
    n=1<<pos;
    _for(i,0,n)rev[i]=(rev[i>>1]>>1)|((i&1)<<(pos-1));
    return n;
}
void FFT(complex *f,int n,int type){
    _for(i,0,n)if(i<rev[i])
    swap(f[i],f[rev[i]]);
    complex t1,t2;
    for(int i=1;i<n;i<<=1){
        complex w(cos(pi/i),type*sin(pi/i));
        for(int j=0;j<n;j+=(i<<1)){
            complex cur(1.0,0.0);
            _for(k,j,j+i){
                t1=f[k],t2=cur*f[k+i];
                f[k]=t1+t2,f[k+i]=t1-t2;
                cur=cur*w;
            }
        }
    }
    if(type==-1)_for(i,0,n)
    f[i].x/=n;
}
char s1[MAXN<<2],s2[MAXN<<2];
int ans[MAXN<<2];
int main()
{
    scanf("%s%s",s1,s2);
    int n1=strlen(s1)-1,n2=strlen(s2)-1,n=build(n1+n2);
    _rep(i,0,n1)
    a[i].x=s1[n1-i]-'0';
    _rep(i,0,n2)
    b[i].x=s2[n2-i]-'0';
    FFT(a,n,1);FFT(b,n,1);
    _for(i,0,n)
    a[i]=a[i]*b[i];
    FFT(a,n,-1);
    _for(i,0,n){
        ans[i]+=(int)(a[i].x+0.5);
        ans[i+1]+=ans[i]/10;
```

```
        ans[i]%=10;
    }
    while(n>=0&&!ans[n])n--;
    if(n==-1)putchar('0');
    else
    while(n>=0)putchar(ans[n--]+'0');
    return 0;
}
```

NTT

算法简介

$O(n\log n)$ 时间实现多项式特定模数意义下的点值表示法与系数表示法之间的转化，无精度误差。

理论基础

阶

定义

设 $n\gt 1,gcd(a,n)=1$，称 $a^x\equiv 1\pmod n$ 的最小正整数解为 a 对 n 的阶，记为 $\delta_n(a)$。

性质

设 $a^x\equiv 1\pmod n$，则 $\delta_n(a)\mid x$，特别的，$\delta_n(a)\mid \varphi(n)$。

原根

定义

如果 (n,a) 满足 $a\gt 1,\delta_n(a)=\varphi(n)$，则称 a 为 n 的原根。

性质

1. 一个正整数 n 具有原根的充要条件为 $n=2,4,p^\alpha,2p^\alpha$，其中 p 为素数
2. 如果 n 具有原根，则 n 具有 $\varphi\left(\varphi(n)\right)$ 个原根
3. 如果 g 为 n 的原根，则 $1,g^1,g^2\cdots g^{\varphi(n)}$ 构成 n 的最简剩余系
4. 设 $\varphi(n)=p_1^{\alpha_1}p_2^{\alpha_2}\cdots p_k^{\alpha_k}$，且 $n\nmid a^{\frac{\varphi(n)}{p_i}}-1(i=1,2\cdots k)$，则 a 为 n 的原根

算法实现

当模数是素数 p 时，记 $\omega_n\equiv g^{\frac {p-1}n}\pmod p$发现 ω_n 满足以下性质

1. $\omega_n^{2k}=\omega_{\frac n2}^k$
2. $\omega_n^{k+\frac n2}\equiv -\omega_n^k\pmod p$
3. $p-1 \nmid k,\sum_{i=0}^{n-1}\omega_n^{ki} \equiv 0\pmod p$

于是考虑用 $g^{\frac {p-1}n}$ 替代 FFT 中的 ω_n其余过程与 FFT 类同。

注意算法过程中需要保证 $\frac {p-1}n$ 为整数，故需要选择含 2 的幂次较多的模数 p

一般常见模数为 $998244353,1004535809,469762049$，这三个数的原根均包含 3。

```
const int MAXN=1e6+5,Mod=998244353,G=3,Inv_G=332748118;
int rev[MAXN<<2];
int build(int k){
    int n,pos=0;
    while((1<<pos)<=k)pos++;
    n=1<<pos;
    _for(i,0,n)rev[i]=(rev[i>>1]>>1)|((i&1)<<(pos-1));
    return n;
}
void NTT(int *f,int n,int type){
    _for(i,0,n)if(i<rev[i])
    swap(f[i],f[rev[i]]);
    int t1,t2;
    for(int i=1;i<n;i<<=1){
        int w=quick_pow(type==1?G:Inv_G,(Mod-1)/(i<<1));
        for(int j=0;j<n;j+=(i<<1)){
            int cur=1;
            _for(k,j,j+i){
                t1=f[k],t2=1LL*cur*f[k+i]%Mod;
                f[k]=(t1+t2)%Mod,f[k+i]=(t1-t2)%Mod;
                cur=1LL*cur*w%Mod;
            }
        }
    }
    if(type==-1){
        int div=quick_pow(n,Mod-2);
        _for(i,0,n)
        f[i]=(1LL*f[i]*div%Mod+Mod)%Mod;
    }
}
```

MTT

洛谷p4245

算法简介

$O(n\log n)$ 时间实现多项式任意模数意义下乘法。

算法实现

三模数NTT

设所求多项式系数表示法为 $\{a_0,a_1\cdots a_{n-1}\}$，考虑取用三个不同的大模数 $469762049,998244353,1004535809$。

跑三轮 NTT，可以得到三组同余方程，利用中国剩余定理将同余方程合并，可以得到 $a_i\equiv b_i\pmod {4710643227511944407909662 73}$

如果 $0\le a_i\le 10^{26}\lt 4710643227511944407909662 73$，显然有 $a_i=b_i$。

设两条多项式的最高次幂为 n，系数最大值为 v，则两多项式相乘得到的多项式的最大系数不超过 $(n+1)v^2$。

故绝大多数题目的数据范围都满足上述约束条件。

关于同余方程的合并，为防止整型溢出，考虑先合并其中两条，设结果为 $a_i \equiv A\pmod M$，令 $a_i=A+MK$，代入第三条同余方程。

于是有 $A+MK\equiv t\pmod m$，移项，得 $K\equiv (t-A)M^{-1}\pmod m$。

由此解得 A 与 K 后即可在乘法不溢出得情况下计算 a_i 在给定模数意义下的结果。

```
int quick_pow(int a,int b,int mod){
    int ans=1;
    while(b){
        if(b&1)
        ans=1LL*ans*a%mod;
        a=1LL*a*a%mod;
        b>>=1;
    }
    return ans;
}
LL mul(LL a,int b,LL mod){
    LL ans=0;
    while(b){
        if(b&1)
        ans=(ans+a)%mod;
        b>>=1;
        a=(a<<1)%mod;
    }
    return ans;
}
const int m[3]={469762049,998244353,1004535809},G=3;
```

```
int rev[MAXN<<2];
int build(int k){
    int n,pos=0;
    while((1<<pos)<=k)pos++;
    n=1<<pos;
    _for(i,0,n)rev[i]=(rev[i>>1]>>1)|((i&1)<<(pos-1));
    return n;
}
void NTT(int *f,int n,int type,int mod){
    _for(i,0,n)if(i<rev[i])
    swap(f[i],f[rev[i]]);
    int t1,t2,Inv_G=quick_pow(G,mod-2,mod);
    for(int i=1;i<n;i<<=1){
        int w=quick_pow(type==1?G:Inv_G,(mod-1)/(i<<1),mod);
        for(int j=0;j<n;j+=(i<<1)){
            int cur=1;
            _for(k,j,j+i){
                t1=f[k],t2=1LL*cur*f[k+i]%mod;
                f[k]=(t1+t2)%mod,f[k+i]=(t1-t2)%mod;
                cur=1LL*cur*w%mod;
            }
        }
    }
    if(type==-1){
        int div=quick_pow(n,mod-2,mod);
        _for(i,0,n)
        f[i]=(1LL*f[i]*div%mod+mod)%mod;
    }
}
int f2[MAXN<<2],g2[MAXN<<2],temp[3][MAXN<<2];
void MTT(int *f,int n1,int *g,int n2,int *ans,int mod){
    int n=build(n1+n2);
    _for(i,0,3){
        memcpy(f2,f,sizeof(f2));memcpy(g2,g,sizeof(g2));
        NTT(f2,n,1,m[i]);NTT(g2,n,1,m[i]);
        _for(j,0,n)temp[i][j]=1LL*f2[j]*g2[j]%m[i];
        NTT(temp[i],n,-1,m[i]);
    }
    LL A,K,M=1LL*m[0]*m[1];
    int
inv1=quick_pow(m[1],m[0]-2,m[0]),inv2=quick_pow(m[0],m[1]-2,m[1]),inv3=quick
_pow(M%m[2],m[2]-2,m[2]);
    _rep(i,0,n1+n2){
A=(mul(1LL*temp[0][i]*m[1]%M,inv1,M)+mul(1LL*temp[1][i]*m[0]%M,inv2,M))%M;
        K=((temp[2][i]-A)%m[2]+m[2])%m[2]*inv3%m[2];
        ans[i]=(((K%mod)*(M%mod)+A%mod)%mod+mod)%mod;
    }
}
```

优化版拆系数FFT

考虑提高 FFT 精度。取 $m=\sqrt v$($v=$ 给定多项式系数的最大值)，进行下述转化

$$f(x)=f_1(x)m+f_2(x),g(x)=g_1(x)m+g_2(x)$$

于是有

$$f(x)g(x)=f_1(x)g_1(x)m^2+(f_2(x)g_1(x)+f_1(x)g_2(x))m+f_2(x)g_2(x)$$

发现只要求出 $f_1(x)g_1(x),f_2(x)g_1(x),f_1(x)g_2(x),f_2(x)g_2(x)$ 这四条多项式的系数即可求出 $f(x)g(x)$ 的系数。

同时 $f_1(x),f_2(x),g_1(x),g_2(x)$ 系数范围为 $[0,\sqrt v]$使得计算过程中浮点误差减小。

接下来考虑如何快速计算 $f_1(x)g_1(x),f_2(x)g_1(x),f_1(x)g_2(x),f_2(x)g_2(x)$ 这四条多项式的系数。

首先，构造多项式 $P(x)=A(x)+B(x)i,Q(x)=A(x)-B(x)i$不难验证有 $P(x)=\overline{Q(\overline x)}$

于是根据 $P(x)$ 点值表示法可以 $O(n)$ 求出 $Q(x)$ 点值表示法，同时又可以 $O(n)$ 求解 $A(x)=\frac {P(x)+Q(x)}2,B(x)=\frac {P(x)-Q(x)}{2i}$

令 $A=f_1(x),B=f_2(x)$ 于是可以一次 FFT 求出 $f_1(x),f_2(x)$ 的点值表示法。

同理可以一次 FFT 求出 $g_1(x),g_2(x)$ 的点值表示法。

再次构造多项式 $P(x)=f_1(x)g_1(x)+f_2(x)g_1(x)i,Q(x)=f_1(x)g_2(x)+f_2(x)g_2(x)i$

根据 $f_1(x),f_2(x),g_1(x),g_2(x)$ 的点值表示法可以 $O(n)$ 求出 $P(x),Q(x)$ 的点值表示法。

再次对 $P(x),Q(x)$ 使用 FFT 即可求出 $P(x),Q(x)$ 的系数表示法，恰好对应所求的 $f_1(x)g_1(x),f_2(x)g_1(x),f_1(x)g_2(x),f_2(x)g_2(x)$

此方法合计只使用了四次 FFT算法效率优于三模数 NTT

为保证精度，需要使用 long double 数据类型。

```
const long double pi=acos(-1.0);
struct complex{
    long double x,y;
    complex(long double x=0.0,long double y=0.0):x(x),y(y){}
    complex operator + (const complex &b){
        return complex(x+b.x,y+b.y);
    }
    complex operator - (const complex &b){
        return complex(x-b.x,y-b.y);
    }
    complex operator * (const complex &b){
        return complex(x*b.x-y*b.y,x*b.y+y*b.x);
    }
};
int rev[MAXN<<2];
```

```
int build(int k){
    int n,pos=0;
    while((1<<pos)<=k)pos++;
    n=1<<pos;
    _for(i,0,n)rev[i]=(rev[i>>1]>>1)|((i&1)<<(pos-1));
    return n;
}
void FFT(complex *f,int n,int type){
    _for(i,0,n)if(i<rev[i])
    swap(f[i],f[rev[i]]);
    complex t1,t2;
    for(int i=1;i<n;i<<=1){
        complex w(cos(pi/i),type*sin(pi/i));
        for(int j=0;j<n;j+=(i<<1)){
            complex cur(1.0,0.0);
            _for(k,j,j+i){
                t1=f[k],t2=cur*f[k+i];
                f[k]=t1+t2,f[k+i]=t1-t2;
                cur=cur*w;
            }
        }
    }
    if(type==-1)_for(i,0,n)
    f[i].x/=n,f[i].y/=n;
}
void FFT2(complex *f1,complex *f2,int n){
    FFT(f1,n,1);
    f2[0].x=f1[0].x,f2[0].y=-f1[0].y;
    _for(i,1,n)
    f2[i].x=f1[n-i].x,f2[i].y=-f1[n-i].y;
    complex t1,t2;
    _for(i,0,n){
        t1=f1[i],t2=f2[i];
        f1[i]=complex((t1.x+t2.x)*0.5,(t1.y+t2.y)*0.5);
        f2[i]=complex((t1.y-t2.y)*0.5,(t2.x-t1.x)*0.5);
    }
}
complex f1[MAXN<<2],f2[MAXN<<2],g1[MAXN<<2],g2[MAXN<<2],temp[2][MAXN<<2];
void MTT(int *f,int n1,int *g,int n2,int *ans,int mod){
    int n=build(n1+n2),m=4e4;
    _rep(i,0,n1)f1[i].x=f[i]/m,f1[i].y=f[i]%m;
    _rep(i,0,n2)g1[i].x=g[i]/m,g1[i].y=g[i]%m;
    FFT2(f1,f2,n);FFT2(g1,g2,n);
    complex I(0.0,1.0);
    _for(i,0,n){
        temp[0][i]=f1[i]*g1[i]+I*f2[i]*g1[i];
        temp[1][i]=f1[i]*g2[i]+I*f2[i]*g2[i];
    }
    FFT(temp[0],n,-1);FFT(temp[1],n,-1);
    LL a,b,c;
    _rep(i,0,n1+n2){
```

```
a=temp[0][i].x+0.5,b=temp[0][i].y+temp[1][i].x+0.5,c=temp[1][i].y+0.5;
        ans[i]=((a%mod*m%mod*m%mod+b%mod*m%mod+c%mod)%mod+mod)%mod;
    }
}
```

FWT

算法简介

$O(n\log n)$ 时间解决形如这样的问题 $C_i=\sum_{i=j\oplus k}A_jB_k$其中 $\oplus$ 为某种位运算符。

算法实现

该算法基本按照以下四步执行：

1. 构造函数 $FWT[A]_i$使得 $FWT[C]_i=FWT[A]_i\times FWT[B]_i$
2. 根据 A_i,B_i 在 $O(n\log n)$ 时间计算出 $FWT[A]_i,FWT[B]_i(i=0,1,\cdots n-1)$
3. 根据 $FWT[A]_i,FWT[B]_i$ 在 $O(n)$ 时间计算出 $FWT[C]_i(i=0,1,\cdots n-1)$
4. 根据 $FWT[C]_i$ 在 $O(n\log n)$ 时间反演出 $C_i(i=0,1,\cdots n-1)$

或运算

构造函数 $FWT[A]_i=\sum_{j|i=i}A_j$于是有

$$FWT[A]_i\times FWT[B]_i=\sum_{j|i=i}A_j\sum_{k|i=i}B_k=\sum_{(j|k)|i=i}A_jB_k=FWT[C]_i$$

接下来需要快速计算出 $FWT[A]_i,FWT[B]_i(i=0,1,\cdots n-1)$利用分治算法，有

$$FWT[A]=\text{Merge}(FWT[A_0],FWT[A_0]+FWT[A_1])$$

其中 A_0 表示当前序列 A 中下标最高位为 0 的区间(左半区间)A_1 表示当前序列 A 中下标最高位为 1 的区间(右半区间)。

$FWT[A_0]+FWT[A_1]$ 可以理解为对应位置相加Merge 表示序列拼接。

注意分治算法是先抹去最高位，计算出 A_0,A_1再补上最高位，重新计算结果的。

显然 A_0 之间补上最高位后贡献不变A_1 之间贡献也不变，唯一变化的是补上最高位后 A_0 的贡献需要计入 A_1 中，于是上式成立。

接下来考虑反演。反演考虑上述过程的逆运算即可，于是有

$$A=\text{Merge}(A_0,A_1-A_0)$$

```
void OR(int *f,int n,int type){
    for(int i=1;i<n;i<<=1)
    for(int j=0;j<n;j+=(i<<1))
    _for(k,j,j+i)
```

```
    f[k+i]=(f[k+i]+type*f[k])%Mod;
}
```

与运算

构造函数 $FWT[A]_i=\sum_{j\And i=i}A_j$ 于是有

$$FWT[A]_i\times FWT[B]_i=\sum_{j\And i=i}A_j\sum_{k\And i=i}B_k=\sum_{(j\And k)\And i=i}A_jB_k=FWT[C]_i$$

类似的，有

$$FWT[A]=\text{Merge}(FWT[A_0]+FWT[A_1],FWT[A_1])$$

$$A=\text{Merge}(A_0-A_1,A_1)$$

```
void AND(int *f,int n,int type){
    for(int i=1;i<n;i<<=1)
    for(int j=0;j<n;j+=(i<<1))
    _for(k,j,j+i)
    f[k]=(f[k]+type*f[k+i])%Mod;
}
```

异或运算

定义 $a\otimes b$ 表示 $a\And b$ 二进制下的 1 的个数模 2 意义下数值。

于是有 $(i\otimes j)\oplus (i\otimes k)=i\otimes (j\oplus k)$ 构造函数 $FWT[A]_i=\sum_{j\otimes i=0}A_j-\sum_{j\otimes i=1}A_j$

$$FWT[A]_i\times FWT[B]_i=(\sum_{j\otimes i=0}A_j-\sum_{j\otimes i=1}A_j)(\sum_{k\otimes i=0}B_k-\sum_{k\otimes i=1}B_k)=\sum_{(j\oplus k)\otimes i=0}A_jB_k-\sum_{(j\oplus k)\otimes i=1}A_jB_k=FWT[C]_i$$

接下来需要快速计算出 $FWT[A]_i,FWT[B]_i(i=0,1,\cdots n-1)$ 利用分治算法，考虑左右区间之间的相互贡献，有

$$FWT[A]=\text{Merge}(FWT[A_0]+FWT[A_1],FWT[A_0]-FWT[A_1])$$

因为 A_0 补上最高位后相互间 $\And$ 运算结果二进制下的 1 的个数不变，于是对自己贡献仍然为正。

但 A_1 补上最高位后相互间 $\And$ 运算结果二进制下的 1 的个数 $+1$，导致奇偶性改变，对自己贡献变为负。

同时补上最高位后 A_0 A_1 间 $\And$ 运算结果二进制下的 1 的个数不变，于是相互贡献为正。

接下来考虑反演式，有

$$FWT[A]=\text{Merge}\left(\frac{FWT[A_0]+FWT[A_1]}2,\frac{FWT[A_0]-FWT[A_1]}2\right)$$

```
void XOR(int *f,int n,int type){
```

```
    int t1,t2,t3=type==1?1:quick_pow(2,Mod-2);
    for(int i=1;i<n;i<<=1)
    for(int j=0;j<n;j+=(i<<1))
    _for(k,j,j+i){
        t1=f[k],t2=f[k+i];
        f[k]=1LL*(t1+t2)*t3%Mod;
        f[k+i]=1LL*(t1-t2)*t3%Mod;
    }
}
```

算法练习

牛客暑期多校(第二场) E 题

题意

给定 n 个数 $a_1,a_2\cdots a_n(0\le a_i\lt 2^{18})$

要求输出 n 个数，第 i 个数表示取 i 个数(可以重复取)可以得到的最大值。

题解

记 $f_{i,j}$ 表示选取 i 个数能否取到数值 $j$$ans_i$ 表示选取 i 个数能取到的最大值。

显然有 $f_{2,i}=\sum_{i=j\oplus k}f_{1,j}f_{1,k}$事实上有递推式 $f_{n,i}=\sum_{i=j\oplus k}f_{n-1,j}f_{n-1,k}$

注意 $f_{i,j}$ 只考虑是否能取到，不需要统计方案数，于是每轮计算后考虑将不为 0 的 $f_{i,j}$ 都赋值为 1。这样可以防止最终计算结果溢出。

于是时间复杂度变为 $O(2^{18}m)$其中 m 为卷积次数。考虑最大有效卷积次数。

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

Permanent link:
https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:jxm2001:%E5%A4%9A%E9%A1%B9%E5%BC%8F_2&rev=1596678943

Last update: **2020/08/06 09:55**