多项式exp

OIWiki上，多项式全家桶里面有多项式exp

exp

在多项式exp当中，调用了polyln

```
long long exp_t[20005];

void polyexp(long long h[],const int n,long long f[])
{
    memset(exp_t,0,sizeof(exp_t));
    std::fill(f,f+n+n,0);
    f[0]=1;
    int t;
    for(t=2;t<=n;t<<=1)
    {
        const int t2=t<<1;
        polyln(f,t,exp_t);
        exp_t[0]=(h[0]+1-exp_t[0]+MOD)%MOD;
        long long i;
        for(i=1;i!=t;++i)
        {
            exp_t[i]=(h[i]-exp_t[i]+MOD)%MOD;
        }
        std::fill(exp_t+t,exp_t+t2,0);
        NTT(f,t2,1);
        NTT(exp_t,t2,1);
        for(i=0;i!=t2;++i)
        {
            f[i]=f[i]*exp_t[i]%MOD;
        }
        NTT(f,t2,-1);
        std::fill(f+t,f+t2,0);
    }
}
```

ln

在多项式ln当中，调用了polyinv，derivative和integrate

```
long long ln_t[20005];
```

```
void polyln(long long h[],const int n,long long f[])
{
    memset(ln_t,0,sizeof(ln_t));
    const int t=n<<1;
    derivative(h,n,ln_t);
    std::fill(ln_t+n,ln_t+t,0);
    polyinv(h,n,f);
    NTT(ln_t,t,1);
    NTT(f,t,1);
    long long i;
    for(i=0;i!=t;++i)
    {
        ln_t[i]=ln_t[i]*f[i]%MOD;
    }
    NTT(ln_t,t,-1);
    integrate(ln_t,n,f);
}
```

逆

多项式求逆什么都不需要调用。

```
long long inv_t[20005];

void polyinv(long long h[],const int n,long long f[])
{
    memset(inv_t,0,sizeof(inv_t));
    std::fill(f,f+n+n,0);
    f[0]=QPow(h[0],MOD-2);
    int t;
    for(t=2;t<=n;t<<=1)
    {
        const int t2=t<<1;
        std::copy(h,h+t,inv_t);
        std::fill(inv_t+t,inv_t+t2,0);
        NTT(f,t2,1);
        NTT(inv_t,t2,1);
        long long i;
        for(i=0;i!=t2;++i)
        {
            f[i]=f[i]*((2LL-(f[i]*inv_t[i])%MOD+MOD)%MOD)%MOD;
        }
        NTT(f,t2,-1);
        std::fill(f+t,f+t2,0);
    }
}
```

求导和积分

在积分中，需要提前把逆元都求出来。这里建议是通过阶乘的方法比较方便。

```
void derivative(long long h[],const int n,long long f[])
{
    long long i;
    for(i=1;i!=n;++i)
    {
        f[i-1]=(h[i]*i)%MOD;
    }
    f[n-1]=0;
}

void integrate(long long h[],const int n,long long f[])
{
    long long i;
    for(i=n-1;i;--i)
    {
        f[i]=(h[i-1]*((FEG[i]*GAM[i-1])%MOD))%MOD;
    }
    f[0]=0;
}
```

NTT

使用这个NTT板子，传入长度必须是2的幂，建议为2048。

```
long long rev[20005];

void NTT(long long A[],long long n,int inv)//数组A□长度n□逆变换（共轭）符号inv
{
    int bit=0;
    while((1<<bit)<n)
    {
        bit++;//根据数组长度n□确定单位根次数
    }
    long long i;
    for(i=0;i<n;i++)//初始化□rev数组存储位逆序置换
    {
        rev[i]=(rev[i>>1]>>1)|((i&1)<<(bit-1));
        if(i<rev[i])
        {
            long long temp=A[i];
            A[i]=A[rev[i]];
            A[rev[i]]=temp;
```

```
        }
    }
    long long mid,j;
    for(mid=1;mid<n;mid<<=1)//mid是准备合并序列的长度的二分之一
    {
        long long now=mid<<1;//now是准备合并序列的长度
        long long wn=QPow(ROOT,(MOD-1)/now);//单位根
        if(inv==-1)
        {
            wn=QPow(wn,MOD-2);//逆变换时逆元
        }
        for(i=0;i<n;i+=now)//i是合并到了哪一位
        {
            long long w=1;
            for(j=0;j<mid;j++,w=1ll*w*wn%MOD)//蝴蝶变换
            {
                long long x=A[i+j];
                long long y=1ll*w*A[i+j+mid]%MOD;
                A[i+j]=(x+y)%MOD;
                A[i+j+mid]=(x-y+MOD)%MOD;
            }
        }
    }
    if(inv==-1)
    {
        long long p=QPow(n,MOD-2);
        for(i=0;i<n;i++)
        {
            A[i]=1LL*A[i]*p%MOD;
        }
    }
}
```

阶乘初始化

其实用这个也可以很方便的大量计算逆元。

```
long long GAM[20010];
long long FEG[20010];

void init()
{
    GAM[0]=1;
    long long i;
    for(i=1;i<20005;i++)
    {
        GAM[i]=(GAM[i-1]*i)%MOD;
```

```
    }
    FEG[20004]=QPow(GAM[20004],MOD-2);
    for(i=20003;i>=0;i--)
    {
        FEG[i]=(FEG[i+1]*(i+1))%MOD;
    }
}
```

快速幂

是基础。这里的ROOT是原根（生成元），在NTT中用到。

```
const long long MOD=998244353;
const long long ROOT=3;

long long QPow(long long bas,long long t)
{
    long long ret=1;
    for(;t;t>>=1,bas=(bas*bas)%MOD)
    {
        if(t&1LL)
        {
            ret=(ret*bas)%MOD;
        }
    }
    return ret;
}
```