## 较完整多项式模板

# 依赖+多项式求导+多项式积分

```
#include <bits/stdc++.h>
#define ll long long
using namespace std;
const double PI = 3.14159265358979323846;
const int MAXN = 2e5 + 33;//数据范围
const int MOD = 998244353, wroot = 3; //NTT依赖数据

//定义Complex类
class Complex
{
public:
    double r, i;
    Complex(double rr = 0, double ii = 0) { r = rr, i = ii; }
    Complex operator+(const Complex &op) const { return Complex(r + op.r, i
    Complex operator-(const Complex &op) const { return Complex(r - op.r, i
    Complex operator*(const Complex &op) const { return Complex(r * op.r -
};

//取整
int Round(double x)
{
    return int(x + 0.5);
}

//快速幂
ll qp(ll x, ll y, ll MOD)
```

```
ll qp(ll x, ll y, ll MOD)
{
    ll ret = 1;
    while (y > 0)
    {
        if (y & 1)
            ret = ret * x % MOD;
        x = x * x % MOD;
        y >>= 1;
    }
    return ret;
}

int numinv[MAXN << 1];
//返回模MOD意义下x的逆元 MOD为质数
int get_num_inv(int x)
{
    if (x < (MAXN << 1))
    {
        if (numinv[x])
            return numinv[x];
        return numinv[x] = qp(x, MOD - 2, MOD);
    }
    return qp(x, MOD - 2, MOD);
}

//对n次多项式p求导得模MOD意义下的多项式res
void Poly_derivative(const int *p, int *res, const int n)
{
```

```
    for (int i = 0; i <= n - 1; i++)
        res[i] = (ll)p[i + 1] * (i + 1) % MOD;
    res[n] = 0;
}
//对n次多项式p积分得模MOD意义下的多项式res 且res[0]=0;
void Poly_integral(const int *p, int *res, const int n)
{
    for (int i = n; i >= 1; i--)
        res[i] = (ll)p[i - 1] * (ll)get_num_inv(i) % MOD;
    res[0] = 0;
}
```

# FFT+NTT

## FFT

```
namespace FFT_template
{
Complex temp1[MAXN << 2], temp2[MAXN << 2];
void brc(Complex *p, const int N)
{
    int i, j, k;
    for (i = 1, j = N >> 1; i < N - 2; i++)
    {
        if (i < j)
            swap(p[i], p[j]);
        for (k = N >> 1; j >= k; k >>= 1)
```

```
            j -= k;
        if (j < k)
            j += k;
    }
}
void FFT(Complex *p, const int N, const int op) //op==1 为正变换，op==-1为逆
{
    brc(p, N);
    double p0 = PI * op;
    for (int h = 2; h <= N; h <<= 1, p0 *= 0.5)
    {
        int hf = h >> 1;
        Complex unit(cos(p0), sin(p0));
        for (int i = 0; i < N; i += h)
        {
            Complex w(1.0, 0.0);
            for (int j = i; j < i + hf; j++)
            {
                Complex u = p[j], v = w * p[j + hf];
                p[j] = u + v;
                p[j + hf] = u - v;
                w = w * unit;
            }
        }
    }
    if (op == -1)
        for (int i = 0; i < N; i++)
            p[i].r /= N;
```

```
}
} // namespace FFT_template
```

## NTT

```
namespace NTT_templates
{
int wi[MAXN << 2];
void brc(int *p, const int N)
{
    int i, j, k;
    for (i = 1, j = N >> 1; i < N - 2; i++)
    {
        if (i < j)
            swap(p[i], p[j]);
        for (k = N >> 1; j >= k; k >>= 1)
            j -= k;
        if (j < k)
            j += k;
    }
}
void NTT_init(const int N) //使用NTT之前调用,且N要保持一致,为2的幂
{
    wi[0] = 1;
    wi[1] = qp(wroot, (MOD - 1) / N, MOD);
    for (int i = 2; i <= N; i++)
        wi[i] = (ll)wi[i - 1] * (ll)wi[1] % MOD;
```

```
}
void NTT(int *p, const int N, const int op)
{
    brc(p, N);
    for (int h = 2; h <= N; h <<= 1)
    {
        int unit = ((op == -1) ? (N - N / h) : (N / h));
        int hf = h >> 1;
        for (int i = 0; i < N; i += h)
        {
            int w = 0;
            for (int j = i; j < i + hf; j++)
            {
                int u = p[j], v = (ll)wi[w] * (ll)p[j + hf] % MOD;
                if ((p[j] = u + v) >= MOD)
                    p[j] -= MOD;
                if ((p[j + hf] = u - v) < 0)
                    p[j + hf] += MOD;
                if ((w += unit) >= N)
                    w -= N;
            }
        }
    }
    if (op == -1)
    {
        int inv = qp(N, MOD - 2, MOD);
        for (int i = 0; i < N; i++)
```

```
            p[i] = (ll)p[i] * (ll)inv % MOD;
    }
}
} // namespace NTT_templates
```

# 多项式乘法

```
namespace Polynomial_mul
{
using namespace NTT_templates;
using namespace FFT_template;
int temp11[MAXN << 2], temp22[MAXN << 2];
//对于给定n次多项式a和b,多项式res=a*b (可能有精度损失)
void Poly_mul_FFT(const int *a, const int *b, int *res, const int n) //n为最
{
    int N = 2;
    while (N <= n + n)
        N <<= 1;
    Complex *A = temp1,
            *B = temp2;
    for (int i = 0; i < N; i++)
        A[i] = (i <= n ? Complex(a[i], 0.0) : Complex(0.0, 0.0));

    for (int i = 0; i < N; i++)
        B[i] = (i <= n ? Complex(b[i], 0.0) : Complex(0.0, 0.0));
    FFT(A, N, 1);
    FFT(B, N, 1);
    for (int i = 0; i < N; i++)
```

```
        A[i] = A[i] * B[i];
    FFT(A, N, -1);
    for (int i = 0; i < N; i++)
        res[i] = Round(A[i].r);
}
//对于给定n次多项式a和b,求出模MOD以及x^(2*n+1)下的多项式res=a*b
void Poly_mul(const int *a, const int *b, int *res, const int n)
{
    int N = 2;
    while (N <= n + n)
        N <<= 1;
    NTT_init(N);
    int *A = temp11,
        *B = temp22;
    for (int i = 0; i < N; i++)
        A[i] = (i <= n ? a[i] : 0);
    for (int i = 0; i < N; i++)
        B[i] = (i <= n ? b[i] : 0);
    NTT(A, N, 1);
    NTT(B, N, 1);
    for (int i = 0; i < N; i++)
        A[i] = (ll)A[i] * (ll)B[i] % MOD;

    NTT(A, N, -1);
    for (int i = 0; i <= n + n; i++)
        res[i] = A[i];
}
} // namespace Polynomial_mul
```

# 多项式求逆

```
namespace Polynomial_inv
{
using namespace NTT_templates;
int tmp[MAXN << 2], tmp2[MAXN << 2], tmp3[MAXN << 2];
void get_poly_inv(const int *p, int *res, const int N) //N是2的幂,一般不调用
{
    if (N <= 1)
    {
        res[0] = qp(p[0], MOD - 2, MOD);
        return;
    }
    get_poly_inv(p, res, N >> 1);
    int K = N << 1;
    int *temp = tmp;
    for (int i = 0; i < N; i++)
        temp[i] = p[i];
    for (int i = N; i < K; i++)
        temp[i] = res[i] = 0;
    NTT_init(K);
    NTT(temp, K, 1);
    NTT(res, K, 1);
    for (int i = 0; i < K; i++)
    {
        res[i] = (ll)res[i] * (2 - (ll)temp[i] * (ll)res[i] % MOD) % MOD;
```

```
        if (res[i] < 0)
            res[i] += MOD;
    }
    NTT(res, K, -1);
    for (int i = N; i < K; i++)
        res[i] = 0;
}
//对给定n次多项式p,求出其模MOD以及x^(n+1)意义下的逆多项式res
void Poly_inv(const int *p, int *res, const int n) //n是最高次项次数 模x^(n-
{
    int N = 2;
    while (N <= n)
        N <<= 1;
    int dN = N << 1;
    int *temp_in = tmp3,
        *temp_out = tmp2;
    for (int i = 0; i < N; i++)
        temp_in[i] = (i <= n ? p[i] : 0);
    for (int i = 0; i < dN; i++)
        temp_out[i] = 0;
    get_poly_inv(temp_in, temp_out, N);
    for (int i = 0; i <= n; i++)
        res[i] = temp_out[i];
}
} // namespace Polynomial_inv
```

## 多项式求对数(ln)

```
namespace Polynomial_ln
{
using namespace NTT_templates;
int tmp[MAXN << 2], tmp2[MAXN << 2], tmp3[MAXN << 2];
void get_poly_ln(const int *p, int *res, const int N) //N为2的幂 为项数,而非
{
    int *temp = tmp;
    Polynomial_inv::get_poly_inv(p, temp, N);
    int K = N << 1;
    Poly_derivative(p, res, N - 1);
    for (int i = N; i < K; i++)
        res[i] = 0;
    NTT_init(K);
    NTT(res, K, 1);
    NTT(temp, K, 1);
    for (int i = 0; i < K; i++)
        res[i] = (ll)res[i] * temp[i] % MOD;
    NTT(res, K, -1);
    Poly_integral(res, res, N - 1);
}
//对给定n次多项式p (且p[0]==1) ,求出其模MOD以及x^(n+1)意义下的多项式res==ln
void Poly_ln(const int *p, int *res, const int n) //n为最高项次数
{
    int N = 2;
    while (N <= n)
        N <<= 1;
    int *temp_in = tmp3,
```

```
        *temp_out = tmp2;
    for (int i = 0; i < N; i++)
        temp_in[i] = (i <= n ? p[i] : 0);
    get_poly_ln(temp_in, temp_out, N);
    for (int i = 0; i <= n; i++)
        res[i] = temp_out[i];
}
} // namespace Polynomial_ln
```

# 多项式求指数(exp)

```
namespace Polynomial_exp
{
using namespace NTT_templates;
int tmp[MAXN << 2], tmp2[MAXN << 2], tmp3[MAXN << 2];
void get_poly_exp(const int *p, int *res, const int N) //N为2的幂 为项数,而非
{
    if (N <= 1)
    {
        res[0] = 1;
        return;
    }
    get_poly_exp(p, res, N >> 1);
    int *temp = tmp;
    Polynomial_ln::get_poly_ln(res, temp, N);
    int K = N << 1;
```

```
    for (int i = 0; i < N; i++)
    {
        temp[i] = p[i] - temp[i];
        if (temp[i] < 0)
            temp[i] += MOD;
    }
    if ((++temp[0]) == MOD)
        temp[0] = 0;
    for (int i = N; i < K; i++)
        temp[i] = res[i] = 0;
    NTT_init(K);
    NTT(temp, K, 1);
    NTT(res, K, 1);
    for (int i = 0; i < K; i++)
        res[i] = (ll)res[i] * (ll)temp[i] % MOD;
    NTT(res, K, -1);
    for (int i = N; i < K; i++)

        res[i] = 0;
}
//对给定n次多项式p（且p[0]==0），求出其模MOD以及x^(n+1)意义下的多项式res==ex
void Poly_exp(const int *p, int *res, const int n)
{
    int N = 2;
    while (N <= n)
        N <<= 1;
    int *temp_out = tmp2,
        *temp_in = tmp3;
```

```
    for (int i = 0; i < N; i++)
        temp_in[i] = (i <= n ? p[i] : 0);
    get_poly_exp(temp_in, temp_out, N);
    for (int i = 0; i <= n; i++)
        res[i] = temp_out[i];
}
} // namespace Polynomial_exp
```

# 多项式除法 & 取模

```
namespace Polynomial_div
{
int tmp[MAXN << 2], tmp2[MAXN << 2];
//翻转n次多项式p的系数,得到多项式res
void Polynomial_flip(const int *p, int *res, const int n)
{
    for (int i = 0; i <= n; i++)
    {
        res[i] = p[n - i];
    }
}
//对给定n次多项式p,m次多项式q ,p=T*q+R,求出多项式T和R 要求n>=m
void Poly_div(const int *p, const int n, const int *q, const int m, int *re
{
    Polynomial_flip(q, tmp, m);
    Polynomial_inv::Poly_inv(tmp, tmp2, n - m);
    Polynomial_flip(p, tmp, n);
```

```
    Polynomial_mul::Poly_mul(tmp2, tmp, tmp2, n - m);
    Polynomial_flip(tmp2, tmp, n - m);
    for (int i = n - m + 1; i <= n; i++)
        tmp[i] = 0;
    Polynomial_mul::Poly_mul(q, tmp, tmp2, max(n - m, m));
    for (int i = 0; i <= n; i++)
    {
        resR[i] = (p[i] - tmp2[i]) % MOD + MOD;
        if (resR[i] >= MOD)
            resR[i] -= MOD;
    }
    for (int i = 0; i <= n - m; i++)
        resT[i] = tmp[i];
    for (int i = n - m + 1; i <= n; i++)
        resT[i] = 0;
}
} // namespace Polynomial_div
```

2020-01-01

Eserinc

Love life, love blank.