

依赖+多项式求导+多项式积分

```
1. #include <bits/stdc++.h>

2. #define ll long long

3. using namespace std;

4. const double PI = 3.14159265358979323846;

5. const int MAXN = 2e5 + 33; //数据范围

6. const int MOD = 998244353, wroot = 3; //NTT依赖数据

7.

8. //定义Complex类

9. class Complex

10. {

11. public:

12.     double r, i;

13.     Complex(double rr = 0, double ii = 0) { r = rr, i = ii; }

14.     Complex operator+(const Complex &op) const { return Complex(r + op.r, i + op.i); }

15.     Complex operator-(const Complex &op) const { return Complex(r - op.r, i - op.i); }

16.     Complex operator*(const Complex &op) const { return Complex(r * op.r - i * op.i, r * op.i + i * op.r); }

17. };

18.

19. //取整

20. int Round(double x)

21. {

22.     return int(x + 0.5);

23. }

24.

25. //快速幂

26. ll qn(ll x, ll v, ll MOD)
```

```
26. // 求 x 的逆元，即 y，使 xy = 1
```

```
27. {
```

```
28.     ll ret = 1;
```

```
29.     while (y > 0)
```

```
30.     {
```

```
31.         if (y & 1)
```

```
32.             ret = ret * x % MOD;
```

```
33.         x = x * x % MOD;
```

```
34.         y >>= 1;
```

```
35.     }
```

```
36.     return ret;
```

```
37. }
```

```
38.
```

```
39. int numinv[MAXN << 1];
```

```
40. //返回模MOD意义下x的逆元 MOD为质数
```

```
41. int get_num_inv(int x)
```

```
42. {
```

```
43.     if (x < (MAXN << 1))
```

```
44.     {
```

```
45.         if (numinv[x])
```

```
46.             return numinv[x];
```

```
47.         return numinv[x] = qp(x, MOD - 2, MOD);
```

```
48.     }
```

```
49.     return qp(x, MOD - 2, MOD);
```

```
50. }
```

```
51.
```

```
52. //对n次多项式p求导得模MOD意义下的多项式res
```

```
53. void Poly_derivative(const int *p, int *res, const int n)
```

```
54. {
```

```

55.     for (int i = 0; i <= n - 1; i++)
56.         res[i] = (ll)p[i + 1] * (i + 1) % MOD;
57.     res[n] = 0;
58. }
59. //对n次多项式p积分得模MOD意义下的多项式res 且res[0]=0;
60. void Poly_integral(const int *p, int *res, const int n)
61. {
62.     for (int i = n; i >= 1; i--)
63.         res[i] = (ll)p[i - 1] * (ll)get_num_inv(i) % MOD;
64.     res[0] = 0;
65. }

```

FFT+NTT

FFT

```

1. namespace FFT_template
2. {
3.     Complex templ[MAXN << 2], temp2[MAXN << 2];
4.     void brc(Complex *p, const int N)
5.     {
6.         int i, j, k;
7.         for (i = 1, j = N >> 1; i < N - 2; i++)
8.             {
9.                 if (i < j)
10.                    swap(p[i], p[j]);
11.                 for (k = N >> 1; j >= k; k >>= 1)

```

```

12.         j -= k;
13.     if (j < k)
14.         j += k;
15.     }
16. }

17. void FFT(Complex *p, const int N, const int op) //op==1 为正变换, op== -1为逆
18. {
19.     brc(p, N);
20.     double p0 = PI * op;
21.     for (int h = 2; h <= N; h <<= 1, p0 *= 0.5)
22.     {
23.         int hf = h >> 1;
24.         Complex unit(cos(p0), sin(p0));
25.         for (int i = 0; i < N; i += h)
26.         {
27.             Complex w(1.0, 0.0);
28.             for (int j = i; j < i + hf; j++)
29.             {
30.                 Complex u = p[j], v = w * p[j + hf];
31.                 p[j] = u + v;
32.                 p[j + hf] = u - v;
33.                 w = w * unit;
34.             }
35.         }
36.     }
37.     if (op == -1)
38.         for (int i = 0; i < N; i++)
39.             p[i].r /= N;

```

```

40. }

41. } // namespace FFT_template

```

NTT

```

1. namespace NTT_templates

2. {

3. int wi[MAXN << 2];

4. void brc(int *p, const int N)

5. {

6.     int i, j, k;

7.     for (i = 1, j = N >> 1; i < N - 2; i++)

8.     {

9.         if (i < j)

10.            swap(p[i], p[j]);

11.         for (k = N >> 1; j >= k; k >>= 1)

12.             j -= k;

13.         if (j < k)

14.             j += k;

15.     }

16. }

17. void NTT_init(const int N) //使用NTT之前调用,且N要保持一致,为2的幂

18. {

19.     wi[0] = 1;

20.     wi[1] = qp(wroot, (MOD - 1) / N, MOD);

21.     for (int i = 2; i <= N; i++)

22.         wi[i] = (ll)wi[i - 1] * (ll)wi[1] % MOD;

```

```

23. }

24. void NTT(int *p, const int N, const int op)

25. {

26.     brc(p, N);

27.     for (int h = 2; h <= N; h <<= 1)

28.     {

29.         int unit = ((op == -1) ? (N - N / h) : (N / h));

30.         int hf = h >> 1;

31.         for (int i = 0; i < N; i += h)

32.         {

33.             int w = 0;

34.             for (int j = i; j < i + hf; j++)

35.             {

36.                 int u = p[j], v = (11)wi[w] * (11)p[j + hf] % MOD;

37.                 if ((p[j] = u + v) >= MOD)

38.                     p[j] -= MOD;

39.                 if ((p[j + hf] = u - v) < 0)

40.                     p[j + hf] += MOD;

41.                 if ((w += unit) >= N)

42.                     w -= N;

43.             }

44.         }

45.     }

46.     if (op == -1)

47.     {

48.         int inv = qp(N, MOD - 2, MOD);

49.         for (int i = 0; i < N; i++)

```

```

50.             p[i] = (ll)p[i] * (ll)inv % MOD;

51.     }

52. }

53. } // namespace NTT_templates

```

多项式乘法

```

1. namespace Polynomial_mul

2. {

3.     using namespace NTT_templates;

4.     using namespace FFT_template;

5.     int temp11[MAXN << 2], temp22[MAXN << 2];

6.     //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)

7.     void Poly_mul_FFT(const int *a, const int *b, int *res, const int n) //n为最i

8.     {

9.         int N = 2;

10.        while (N <= n + n)

11.            N <<= 1;

12.        Complex *A = temp1,

13.            *B = temp2;

14.        for (int i = 0; i < N; i++)

15.            A[i] = (i <= n ? Complex(a[i], 0.0) : Complex(0.0, 0.0));

16.        for (int i = 0; i < N; i++)

17.            B[i] = (i <= n ? Complex(b[i], 0.0) : Complex(0.0, 0.0));

18.        FFT(A, N, 1);

19.        FFT(B, N, 1);

20.        for (int i = 0; i < N; i++)

```

```

21.         A[i] = A[i] * B[i];

22.     FFT(A, N, -1);

23.     for (int i = 0; i < N; i++)

24.         res[i] = Round(A[i].r);

25. }

26. //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

27. void Poly_mul(const int *a, const int *b, int *res, const int n)

28. {

29.     int N = 2;

30.     while (N <= n + n)

31.         N <<= 1;

32.     NTT_init(N);

33.     int *A = temp11,

34.         *B = temp22;

35.     for (int i = 0; i < N; i++)

36.         A[i] = (i <= n ? a[i] : 0);

37.     for (int i = 0; i < N; i++)

38.         B[i] = (i <= n ? b[i] : 0);

39.     NTT(A, N, 1);

40.     NTT(B, N, 1);

41.     for (int i = 0; i < N; i++)

42.         A[i] = (1l)A[i] * (1l)B[i] % MOD;

43.     NTT(A, N, -1);

44.     for (int i = 0; i <= n + n; i++)

45.         res[i] = A[i];

46. }

47. } // namespace Polynomial_mul

```


多项式求逆

```
1. namespace Polynomial_inv
2. {
3.     using namespace NTT_templates;
4.     int tmp[MAXN << 2], tmp2[MAXN << 2], tmp3[MAXN << 2];
5.     void get_poly_inv(const int *p, int *res, const int N) //N是2的幂, 一般不调用
6.     {
7.         if (N <= 1)
8.         {
9.             res[0] = qp(p[0], MOD - 2, MOD);
10.            return;
11.        }
12.        get_poly_inv(p, res, N >> 1);
13.        int K = N << 1;
14.        int *temp = tmp;
15.        for (int i = 0; i < N; i++)
16.            temp[i] = p[i];
17.        for (int i = N; i < K; i++)
18.            temp[i] = res[i] = 0;
19.        NTT_init(K);
20.        NTT(temp, K, 1);
21.        NTT(res, K, 1);
22.        for (int i = 0; i < K; i++)
23.        {
24.            res[i] = (ll)res[i] * (2 - (ll)temp[i] * (ll)res[i] % MOD) % MOD;
```

```

25.         if (res[i] < 0)
26.             res[i] += MOD;
27.     }
28.     NTT(res, K, -1);
29.     for (int i = N; i < K; i++)
30.         res[i] = 0;
31. }
32. //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
33. void Poly_inv(const int *p, int *res, const int n) //n是最高次项次数 模 $x^{(n-1)}$ 
34. {
35.     int N = 2;
36.     while (N <= n)
37.         N <<= 1;
38.     int dN = N << 1;
39.     int *temp_in = tmp3,
40.         *temp_out = tmp2;
41.     for (int i = 0; i < N; i++)
42.         temp_in[i] = (i <= n ? p[i] : 0);
43.     for (int i = 0; i < dN; i++)
44.         temp_out[i] = 0;
45.     get_poly_inv(temp_in, temp_out, N);
46.     for (int i = 0; i <= n; i++)
47.         res[i] = temp_out[i];
48. }
49. } // namespace Polynomial_inv

```

多项式求对数(ln)

```

1. namespace Polynomial_ln

2. {

3. using namespace NTT_templates;

4. int tmp[MAXN << 2], tmp2[MAXN << 2], tmp3[MAXN << 2];

5. void get_poly_ln(const int *p, int *res, const int N) //N为2的幂 为项数,而非;

6. {

7.     int *temp = tmp;

8.     Polynomial_inv::get_poly_inv(p, temp, N);

9.     int K = N << 1;

10.    Poly_derivative(p, res, N - 1);

11.    for (int i = N; i < K; i++)

12.        res[i] = 0;

13.    NTT_init(K);

14.    NTT(res, K, 1);

15.    NTT(temp, K, 1);

16.    for (int i = 0; i < K; i++)

17.        res[i] = (ll)res[i] * temp[i] % MOD;

18.    NTT(res, K, -1);

19.    Poly_integral(res, res, N - 1);

20. }

21. //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

22. void Poly_ln(const int *p, int *res, const int n) //n为最高项次数

23. {

24.     int N = 2;

25.     while (N <= n)

26.         N <<= 1;

27.     int *temp_in = tmp3,

```

```

28.         *temp_out = tmp2;

29.     for (int i = 0; i < N; i++)

30.         temp_in[i] = (i <= n ? p[i] : 0);

31.     get_poly_ln(temp_in, temp_out, N);

32.     for (int i = 0; i <= n; i++)

33.         res[i] = temp_out[i];

34. }

35. } // namespace Polynomial_ln

```

多项式求指数(exp)

```

1. namespace Polynomial_exp

2. {

3.     using namespace NTT_templates;

4.     int tmp[MAXN << 2], tmp2[MAXN << 2], tmp3[MAXN << 2];

5.     void get_poly_exp(const int *p, int *res, const int N) //N为2的幂 为项数,而非

6.     {

7.         if (N <= 1)

8.         {

9.             res[0] = 1;

10.            return;

11.        }

12.        get_poly_exp(p, res, N >> 1);

13.        int *temp = tmp;

14.        Polynomial_ln::get_poly_ln(res, temp, N);

15.        int K = N << 1;

```

```

16.     for (int i = 0; i < N; i++)
17.     {
18.         temp[i] = p[i] - temp[i];
19.         if (temp[i] < 0)
20.             temp[i] += MOD;
21.     }
22.     if ((++temp[0]) == MOD)
23.         temp[0] = 0;
24.     for (int i = N; i < K; i++)
25.         temp[i] = res[i] = 0;
26.     NTT_init(K);
27.     NTT(temp, K, 1);
28.     NTT(res, K, 1);
29.     for (int i = 0; i < K; i++)
30.         res[i] = (1l)res[i] * (1l)temp[i] % MOD;
31.     NTT(res, K, -1);
32.     for (int i = N; i < K; i++)
33.         res[i] = 0;
34. }
35. //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==exl
36. void Poly_exp(const int *p, int *res, const int n)
37. {
38.     int N = 2;
39.     while (N <= n)
40.         N <<= 1;
41.     int *temp_out = tmp2,
42.         *temp_in = tmp3;

```

```

43.     for (int i = 0; i < N; i++)
44.         temp_in[i] = (i <= n ? p[i] : 0);
45.     get_poly_exp(temp_in, temp_out, N);
46.     for (int i = 0; i <= n; i++)
47.         res[i] = temp_out[i];
48. }
49. } // namespace Polynomial_exp

```

多项式除法 & 取模

```

1. namespace Polynomial_div
2. {
3.     int tmp[MAXN << 2], tmp2[MAXN << 2];
4.     //翻转n次多项式p的系数,得到多项式res
5.     void Polynomial_flip(const int *p, int *res, const int n)
6.     {
7.         for (int i = 0; i <= n; i++)
8.             {
9.                 res[i] = p[n - i];
10.            }
11.    }
12.    //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
13.    void Poly_div(const int *p, const int n, const int *q, const int m, int *res)
14.    {
15.        Polynomial_flip(q, tmp, m);
16.        Polynomial_inv::Poly_inv(tmp, tmp2, n - m);
17.        Polynomial_flip(p, tmp, n);
18.        Poly_mul(tmp, tmp2, res, n - m + 1);
19.        Polynomial_inv::Poly_inv(res, res, n - m + 1);
20.    }
21. }

```

```

18.     Polynomial_mul::Poly_mul(tmp2, tmp, tmp2, n - m);
19.     Polynomial_flip(tmp2, tmp, n - m);
20.     for (int i = n - m + 1; i <= n; i++)
21.         tmp[i] = 0;
22.     Polynomial_mul::Poly_mul(q, tmp, tmp2, max(n - m, m));
23.     for (int i = 0; i <= n; i++)
24.     {
25.         resR[i] = (p[i] - tmp2[i]) % MOD + MOD;
26.         if (resR[i] >= MOD)
27.             resR[i] -= MOD;
28.     }
29.     for (int i = 0; i <= n - m; i++)
30.         resT[i] = tmp[i];
31.     for (int i = n - m + 1; i <= n; i++)
32.         resT[i] = 0;
33. }
34. } // namespace Polynomial_div

```

2020-01-01



Eserinc

Love life, love blank.



