多项式

生成函数

1. 序列 a 的**普通生成函数**: $F(x) = \sum a_n x^n$
2. 序列 a 的**指数生成函数**: $F(x) = \sum a_n \frac{x^n}{n!}$

泰勒展开式

1. $\frac{1}{1-x} = 1 + x + x^2 + x^3 + \cdots = \sum_{n=0}^{\infty} x^n$
2. $\frac{1}{1-x^2} = 1 + x^2 + x^4 + \cdots$
3. $\frac{1}{1-x^3} = 1 + x^3 + x^6 + \cdots$
4. $\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + \cdots$
5. $e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$
6. $e^{-x} = 1 - \frac{x^1}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \cdots$
7. $\frac{e^x + e^{-x}}{2} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots$
8. $\frac{e^x - e^{-x}}{2} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \cdots$

有穷序列的生成函数

1. $1 + x + x^2 = \frac{1-x^3}{1-x}$
2. $1 + x + x^2 + x^3 = \frac{1-x^4}{1-x}$

广义二项式定理

$$\frac{1}{(1-x)^n} = \sum_{i=0}^{\infty} \binom{n+i-1}{i} x^i$$

证明

1. 扩展域
   $(1+x)^n = \sum_{i=0}^{n} \binom{n}{i} x^i$，因 $i > n, \binom{n}{i} = 0$。
2. 扩展指数为负数
   $\binom{-n}{i} = \frac{(-n)(-n-1)\cdots(-n-i+1)}{i!} = (-1)^i \times \frac{n(n+1)\cdots(n+i-1)}{i!} = (-1)^i \binom{n+i-1}{i}$
3. 括号内的加号变减号
   $(1-x)^{-n} = \sum_{i=0}^{\infty} (-1)^i \binom{n+i-1}{i} (-x)^i = \sum_{i=0}^{\infty} \binom{n+i-1}{i} x^i$

线性凸包

```
struct Line {
  i64 a, b, r;
  bool operator<(Line l) { return pair(a, b) > pair(l.a, l.b); }
  bool operator<(i64 x) { return r < x; }
};
struct Lines : vector<Line> {
  static constexpr i64 inf = numeric_limits<i64>::max();
  Lines(i64 a, i64 b) : vector<Line>{{a, b, inf}} {}
  Lines(vector<Line>& lines) {
    if (not ranges::is_sorted(lines, less())) ranges::sort(lines, less());
    for (auto [a, b, _] : lines) {
      for (; not empty(); pop_back()) {
        if (back().a == a) continue;
        i64 da = back().a - a, db = b - back().b;
        back().r = db / da - (db < 0 and db % da);
        if (size() == 1 or back().r > end()[-2].r) break;
      }
      emplace_back(a, b, inf);
    }
  }
  Lines operator+(Lines& lines) {
    vector<Line> res(size() + lines.size());
    ranges::merge(*this, lines, res.begin(), less());
    return Lines(res);
  }
  i64 min(i64 x) {
    auto [a, b, _] = *lower_bound(begin(), end(), x, less());
    return a * x + b;
  }
};
```

NTT快速数论变换 O(nlogn)

常数很大

```
namespace ntt {

    const long long mod = 998244353;
    const long long G = 3;
    const long long Gi = 332748118; // G 在模 p 意义下的逆元

    class P {
    public:
        long long a{};

        P() = default;

        [[nodiscard]] long long get() const {
            return a;
        }

        explicit P(long long p) {
            a = p % mod;
            if (a < 0) {
                a += mod;
            }
        };

        P operator+(const P& rhs) const {
            return P{ a + rhs.a };
        }

        P operator-(const P& rhs) const {
            return P{ a - rhs.a };
        }

        P operator*(const P& rhs) const {
            return P{ a * rhs.a };
        }

        [[nodiscard]] P pow(long long n) const {
            P bas = P(a);
            P res = P(1);
```

```cpp
        while (n > 0) {
            if (n % 2 == 1) {
                res = res * bas;
            }
            bas = bas * bas;
            n /= 2;
        }
        return res;
    }

    [[nodiscard]] P inverse() const {
        return pow(mod - 2);
    }
};

class NTTMultiplier {
public:
    vector<int> input1;
    vector<int> input2;
    vector<int> bitInv;//位逆置换使用
    int size{};

    NTTMultiplier(const vector<int>& v1, const vector<int>& v2) {
        input1 = v1;
        input2 = v2;
        size = (int)(input1.size() + input2.size() - 1);
        int n = 1;
        while (n < size) {
            n <<= 1;
        }
        size = n;
        input1.resize(size);
        input2.resize(size);
        bitInv.resize(size);
        initBitInv();//位逆置换使用
    }

    void initBitInv() {
        bitInv[0] = 0;
        int log2n = (int)log2(size);
        for (int i = 1; i < size; i++) {
            int pre = (i & 1) << (log2n - 1);//第1位(奇数为1,偶数为0);
            int suf = bitInv[i >> 1] >> 1;   //第2到第n位(这是的递推公式);
            bitInv[i] = pre | suf;
        }
    }

    vector<int> multiply() {
        // 将输入转换为模数形式
        vector<P> nttInput1(input1.begin(), input1.end());
        vector<P> nttInput2(input2.begin(), input2.end());

        // 执行快速傅里叶变换
        fastNTT(nttInput1, false);
        fastNTT(nttInput2, false);
        // 对应位置相乘
        for (int i = 0; i < size; i++) {
            nttInput1[i] = nttInput1[i] * nttInput2[i];
        }
        // 执行反向快速傅里叶变换
        fastNTT(nttInput1, true);
        P invSize = P(size).inverse();
        for (int i = 0; i < size; i++) {
            nttInput1[i] = nttInput1[i] * invSize;
        }

        // 取实部并取整
        vector<int> result(size);
        for (int i = 0; i < size; i++) {
            result[i] = (int)nttInput1[i].get();
        }

        input1.clear();
        input2.clear();
        size = 0;

        return result;
    }


    void bitRev(vector<P>& arr) {
        for (int i = 1; i < arr.size(); i++) {
            if (i < bitInv[i]) {
                swap(arr[i], arr[bitInv[i]]);        //交换
            }
        }
```

```cpp
}


void fastNTT(vector<P>& data, bool inverse) {
    int n = (int)data.size();
    bitRev(data);
    P bas = inverse ? P(Gi) : P(G);
    for (int len = 2; len <= n; len *= 2) {
        long long angle = (long long)(mod - 1) / len;
        P wn = bas.pow(angle);
        for (int i = 0; i < n; i += len) {
            P w(1);
            for (int j = i; j < i + len / 2; j++) {
                P evenVal = data[j];
                P oddVal = data[j + len / 2] * w;
                data[j] = evenVal + oddVal;
                data[j + len / 2] = evenVal - oddVal;
                w = w * wn;
            }
        }
    }
}

void ntt(vector<P>& data, bool inverse) { // NOLINT(misc-no-recursion)
    int n = (int)data.size();
    if (n == 1) {
        return;
    }
    vector<P> even(n / 2);
    vector<P> odd(n / 2);
    // 分离奇偶项
    for (int i = 0; i < n / 2; i++) {
        even[i] = data[2 * i];
        odd[i] = data[2 * i + 1];
    }
    // 递归进行快速傅里叶变换
    ntt(even, inverse);
    ntt(odd, inverse);

    P bas = inverse ? P(Gi) : P(G);
    long long angle = (long long)(mod - 1) / n; // NOLINT(cppcoreguidelines-narrowing-c
    P w(1);
    P wn = bas.pow(angle);
```

```
            // 合并结果
            for (int i = 0; i < n / 2; i++) {
                P evenVal = even[i];
                P oddVal = odd[i] * w;
                data[i] = evenVal + oddVal;
                data[i + n / 2] = evenVal - oddVal;
                w = w * wn;
            }
        }
    };
}
```

jiangly

```cpp
template<class T>
constexpr T power(T a, i64 b) {
    T res = 1;
    for (; b; b /= 2, a *= a) {
        if (b % 2) {
            res *= a;
        }
    }
    return res;
}

template<int P>
struct MInt {
    int x;
    constexpr MInt() : x{} {}
    constexpr MInt(i64 x) : x{ norm(x % getMod()) } {}

    static int Mod;
    constexpr static int getMod() {
        if (P > 0) {
            return P;
        }
        else {
            return Mod;
        }
    }
    constexpr static void setMod(int Mod_) {
        Mod = Mod_;
    }
    constexpr int norm(int x) const {
        if (x < 0) {
            x += getMod();
        }
        if (x >= getMod()) {
            x -= getMod();
        }
        return x;
    }
    constexpr int val() const {
        return x;
    }
```

```
explicit constexpr operator int() const {
    return x;
}
constexpr MInt operator-() const {
    MInt res;
    res.x = norm(getMod() - x);
    return res;
}
constexpr MInt inv() const {
    assert(x != 0);
    return power(*this, getMod() - 2);
}
constexpr MInt& operator*=(MInt rhs)& {
    x = 1LL * x * rhs.x % getMod();
    return *this;
}
constexpr MInt& operator+=(MInt rhs)& {
    x = norm(x + rhs.x);
    return *this;
}
constexpr MInt& operator-=(MInt rhs)& {
    x = norm(x - rhs.x);
    return *this;
}
constexpr MInt& operator/=(MInt rhs)& {
    return *this *= rhs.inv();
}
friend constexpr MInt operator*(MInt lhs, MInt rhs) {
    MInt res = lhs;
    res *= rhs;
    return res;
}
friend constexpr MInt operator+(MInt lhs, MInt rhs) {
    MInt res = lhs;
    res += rhs;
    return res;
}
friend constexpr MInt operator-(MInt lhs, MInt rhs) {
    MInt res = lhs;
    res -= rhs;
    return res;
}
friend constexpr MInt operator/(MInt lhs, MInt rhs) {
        MInt res = lhs;
        res /= rhs;
        return res;
    }
    friend constexpr std::istream& operator>>(std::istream& is, MInt& a) {
        i64 v;
        is >> v;
        a = MInt(v);
        return is;
    }
    friend constexpr std::ostream& operator<<(std::ostream& os, const MInt& a) {
        return os << a.val();
    }
    friend constexpr bool operator==(MInt lhs, MInt rhs) {
        return lhs.val() == rhs.val();
    }
    friend constexpr bool operator!=(MInt lhs, MInt rhs) {
        return lhs.val() != rhs.val();
    }
};

template<>
int MInt<0>::Mod = 1;

template<int V, int P>
constexpr MInt<P> CInv = MInt<P>(V).inv();

constexpr int P = 998244353;
using Z = MInt<P>;

std::vector<int> rev;
template<int P>
std::vector<MInt<P>> roots{ 0, 1 };

template<int P>
constexpr MInt<P> findPrimitiveRoot() {
    MInt<P> i = 2;
    int k = __builtin_ctz(P - 1);
    while (true) {
        if (power(i, (P - 1) / 2) != 1) {
            break;
        }
        i += 1;
    }
    return power(i, (P - 1) >> k);
}

template<int P>
constexpr MInt<P> primitiveRoot = findPrimitiveRoot<P>();

template<>
constexpr MInt<998244353> primitiveRoot<998244353> {31};

template<int P>
constexpr void dft(std::vector<MInt<P>>& a) {
    int n = a.size();

    if (int(rev.size()) != n) {
        int k = __builtin_ctz(n) - 1;
        rev.resize(n);
        for (int i = 0; i < n; i++) {
            rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
        }
    }

    for (int i = 0; i < n; i++) {
        if (rev[i] < i) {
            std::swap(a[i], a[rev[i]]);
        }
    }
    if (roots<P>.size() < n) {
        int k = __builtin_ctz(roots<P>.size());
        roots<P>.resize(n);
        while ((1 << k) < n) {
            auto e = power(primitiveRoot<P>, 1 << (__builtin_ctz(P - 1) - k - 1));
            for (int i = 1 << (k - 1); i < (1 << k); i++) {
                roots<P>[2 * i] = roots<P>[i];
                roots<P>[2 * i + 1] = roots<P>[i] * e;
            }
            k++;
        }
    }
    for (int k = 1; k < n; k *= 2) {
        for (int i = 0; i < n; i += 2 * k) {
            for (int j = 0; j < k; j++) {
                MInt<P> u = a[i + j];
```

```cpp
                MInt<P> v = a[i + j + k] * roots<P>[k + j];
                a[i + j] = u + v;
                a[i + j + k] = u - v;
            }
        }
    }
}

template<int P>
constexpr void idft(std::vector<MInt<P>>& a) {
    int n = a.size();
    std::reverse(a.begin() + 1, a.end());
    dft(a);
    MInt<P> inv = (1 - P) / n;
    for (int i = 0; i < n; i++) {
        a[i] *= inv;
    }
}

template<int P = 998244353>
struct Poly : public std::vector<MInt<P>> {
    using Value = MInt<P>;

    Poly() : std::vector<Value>() {}
    explicit constexpr Poly(int n) : std::vector<Value>(n) {}

    explicit constexpr Poly(const std::vector<Value>& a) : std::vector<Value>(a) {}
    constexpr Poly(const std::initializer_list<Value>& a) : std::vector<Value>(a) {}

    template<class InputIt, class = std::_RequireInputIter<InputIt>>
    explicit constexpr Poly(InputIt first, InputIt last) : std::vector<Value>(first, last) {}

    template<class F>
    explicit constexpr Poly(int n, F f) : std::vector<Value>(n) {
        for (int i = 0; i < n; i++) {
            (*this)[i] = f(i);
        }
    }

    constexpr Poly shift(int k) const {
        if (k >= 0) {
            auto b = *this;
            b.insert(b.begin(), k, 0);
            return b;
        }
        else if (this->size() <= -k) {
            return Poly();
        }
        else {
            return Poly(this->begin() + (-k), this->end());
        }
    }
    constexpr Poly trunc(int k) const {
        Poly f = *this;
        f.resize(k);
        return f;
    }
    constexpr friend Poly operator+(const Poly& a, const Poly& b) {
        Poly res(std::max(a.size(), b.size()));
        for (int i = 0; i < a.size(); i++) {
            res[i] += a[i];
        }
        for (int i = 0; i < b.size(); i++) {
            res[i] += b[i];
        }
        return res;
    }
    constexpr friend Poly operator-(const Poly& a, const Poly& b) {
        Poly res(std::max(a.size(), b.size()));
        for (int i = 0; i < a.size(); i++) {
            res[i] += a[i];
        }
        for (int i = 0; i < b.size(); i++) {
            res[i] -= b[i];
        }
        return res;
    }
    constexpr friend Poly operator-(const Poly& a) {
        std::vector<Value> res(a.size());
        for (int i = 0; i < int(res.size()); i++) {
            res[i] = -a[i];
        }
        return Poly(res);
    }
    constexpr friend Poly operator*(Poly a, Poly b) {
        if (a.size() == 0 || b.size() == 0) {
            return Poly();
        }
        if (a.size() < b.size()) {
            std::swap(a, b);
        }
        int n = 1, tot = a.size() + b.size() - 1;
        while (n < tot) {
            n *= 2;
        }
        if (((P - 1) & (n - 1)) != 0 || b.size() < 128) {
            Poly c(a.size() + b.size() - 1);
            for (int i = 0; i < a.size(); i++) {
                for (int j = 0; j < b.size(); j++) {
                    c[i + j] += a[i] * b[j];
                }
            }
            return c;
        }
        a.resize(n);
        b.resize(n);
        dft(a);
        dft(b);
        for (int i = 0; i < n; ++i) {
            a[i] *= b[i];
        }
        idft(a);
        a.resize(tot);
        return a;
    }
    constexpr friend Poly operator*(Value a, Poly b) {
        for (int i = 0; i < int(b.size()); i++) {
            b[i] *= a;
        }
        return b;
    }
    constexpr friend Poly operator*(Poly a, Value b) {
        for (int i = 0; i < int(a.size()); i++) {
            a[i] *= b;
        }
        return a;
    }
    constexpr friend Poly operator/(Poly a, Value b) {
        for (int i = 0; i < int(a.size()); i++) {
            a[i] /= b;
        }
        return a;
    }
    constexpr Poly& operator+=(Poly b) {
        return (*this) = (*this) + b;
    }
    constexpr Poly& operator-=(Poly b) {
        return (*this) = (*this) - b;
    }
    constexpr Poly& operator*=(Poly b) {
        return (*this) = (*this) * b;
    }
    constexpr Poly& operator*=(Value b) {
        return (*this) = (*this) * b;
    }
    constexpr Poly& operator/=(Value b) {
        return (*this) = (*this) / b;
    }
    constexpr Poly deriv() const {
        if (this->empty()) {
            return Poly();
        }
        Poly res(this->size() - 1);
        for (int i = 0; i < this->size() - 1; ++i) {
            res[i] = (i + 1) * (*this)[i + 1];
        }
        return res;
    }
    constexpr Poly integr() const {
        Poly res(this->size() + 1);
        for (int i = 0; i < this->size(); ++i) {
            res[i + 1] = (*this)[i] / (i + 1);
        }
        return res;
    }
    constexpr Poly inv(int m) const {
        Poly x{ (*this)[0].inv() };
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly{ 2 } - trunc(k) * x)).trunc(k);
        }
        return x.trunc(m);
    }
    constexpr Poly log(int m) const {
        return (deriv() * inv(m)).integr().trunc(m);
    }
    constexpr Poly exp(int m) const {
        Poly x{ 1 };
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x * (Poly{ 1 } - x.log(k) + trunc(k))).trunc(k);
        }
        return x.trunc(m);
    }
    constexpr Poly pow(int k, int m) const {
        int i = 0;
        while (i < this->size() && (*this)[i] == 0) {
            i++;
        }
        if (i == this->size() || 1LL * i * k >= m) {
            return Poly(m);
        }
        Value v = (*this)[i];
        auto f = shift(-i) * v.inv();
        return (f.log(m - i * k) * k).exp(m - i * k).shift(i * k) * power(v, k);
    }
    constexpr Poly sqrt(int m) const {
        Poly x{ 1 };
        int k = 1;
        while (k < m) {
            k *= 2;
            x = (x + (trunc(k) * x.inv(k)).trunc(k)) * CInv<2, P>;
        }
        return x.trunc(m);
    }
    constexpr Poly mulT(Poly b) const {
        if (b.size() == 0) {
            return Poly();
        }
        int n = b.size();
        std::reverse(b.begin(), b.end());
        return ((*this) * b).shift(-(n - 1));
    }
    constexpr std::vector<Value> eval(std::vector<Value> x) const {
        if (this->size() == 0) {
            return std::vector<Value>(x.size(), 0);
        }
        const int n = std::max(x.size(), this->size());
        std::vector<Poly> q(4 * n);
        std::vector<Value> ans(x.size());
        x.resize(n);
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                q[p] = Poly{ 1, -x[l] };
            }
            else {
                int m = (l + r) / 2;
                build(2 * p, l, m);
                build(2 * p + 1, m, r);
                q[p] = q[2 * p] * q[2 * p + 1];
            }
            };
        build(1, 0, n);
        std::function<void(int, int, int, const Poly&)> work = [&](int p, int l, int r, const Po
            if (r - l == 1) {
                if (l < int(ans.size())) {
                    ans[l] = num[0];
                }
            }
            else {
                int m = (l + r) / 2;
                work(2 * p, l, m, num.mulT(q[2 * p + 1]).trunc(m - l));
                work(2 * p + 1, m, r, num.mulT(q[2 * p]).trunc(r - m));
            }
            };
        work(1, 0, n, mulT(q[1].inv(n)));
        return ans;
    }
};

template<int P = 998244353>
Poly<P> berlekampMassey(const Poly<P>& s) {
    Poly<P> c;
    Poly<P> oldC;
    int f = -1;
    for (int i = 0; i < s.size(); i++) {
        auto delta = s[i];
        for (int j = 1; j <= c.size(); j++) {
            delta -= c[j - 1] * s[i - j];
        }
        if (delta == 0) {
            continue;
        }
        if (f == -1) {
            c.resize(i + 1);
            f = i;
        }
        else {
            auto d = oldC;
            d *= -1;
            d.insert(d.begin(), 1);
            MInt<P> df1 = 0;
            for (int j = 1; j <= d.size(); j++) {
                df1 += d[j - 1] * s[f + 1 - j];
            }
            assert(df1 != 0);
            auto coef = delta / df1;
            d *= coef;
            Poly<P> zeros(i - f - 1);
            zeros.insert(zeros.end(), d.begin(), d.end());
            d = zeros;
            auto temp = c;
            c += d;
            if (i - temp.size() > f - oldC.size()) {
                oldC = temp;
                f = i;
            }
        }
    }
    c *= -1;
    c.insert(c.begin(), 1);
    return c;
}


template<int P = 998244353>
MInt<P> linearRecurrence(Poly<P> p, Poly<P> q, i64 n) {
    int m = q.size() - 1;
    while (n > 0) {
        auto newq = q;
        for (int i = 1; i <= m; i += 2) {
            newq[i] *= -1;
        }
        auto newp = p * newq;
        newq = q * newq;
        for (int i = 0; i < m; i++) {
            p[i] = newp[i * 2 + n % 2];
        }
        for (int i = 0; i <= m; i++) {
            q[i] = newq[i * 2];
        }
        n /= 2;
    }
    return p[0] / q[0];
}
```