多项式

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;
  }
};
```

多项式封装

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

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

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

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

```

```
    template<class F> explicit constexpr Poly(int n, F f) : 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(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(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) {
        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()) {
            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();
        reverse(b.begin(), b.end());
        return ((*this) * b).shift(-(n - 1));
    }
    constexpr vector<Value> eval(vector<Value> x) const {
        if (this->size() == 0) {
            return vector<Value>(x.size(), 0);
        }
        const int n = max(x.size(), this->size());
        vector<Poly> q(4 * n);
        vector<Value> ans(x.size());
        x.resize(n);
        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);
        function<void(int, int, int, const Poly &)> work = [&](int p, int l, int r,
                                                                const Poly &num) {
            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]).resize(m - l));
                work(2 * p + 1, m, r, num.mulT(q[2 * p]).resize(r - m));
            }
        };
        work(1, 0, n, mulT(q[1].inv(n)));
```

```
        return ans;
    }
};
```

离散傅里叶变换 dft 与其逆变换 idft

```
vector<int> rev;
template<int P> 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(vector<MInt<P>> &a) { // 离散傅里叶变换
    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) {
            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++;
        }
    }
```

```cpp
    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];
                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(vector<MInt<P>> &a) { // 逆变换
    int n = a.size();
    reverse(a.begin() + 1, a.end());
    dft(a);
    MInt<P> inv = (1 - P) / n;
    for (int i = 0; i < n; i++) {
        a[i] *= inv;
    }
}
```

Berlekamp-Massey 算法（杜教筛）

求解数列的最短线性递推式，最坏复杂度为 $\mathcal{O}(NM)$，其中 N 为数列长度，M 为它的最短递推式的阶数。

```cpp
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;
}
```

Linear-Recurrence 算法

```
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];
}
```

快速傅里叶变换 FFT

$\mathcal{O}(N\log N)$。

```
struct Polynomial {
    constexpr static double PI = acos(-1);
    struct Complex {
        double x, y;
        Complex(double _x = 0.0, double _y = 0.0) {
            x = _x;
            y = _y;
        }
        Complex operator-(const Complex &rhs) const {
            return Complex(x - rhs.x, y - rhs.y);
```

```
        }
        Complex operator+(const Complex &rhs) const {
            return Complex(x + rhs.x, y + rhs.y);
        }
        Complex operator*(const Complex &rhs) const {
            return Complex(x * rhs.x - y * rhs.y, x * rhs.y + y * rhs.x);
        }
    };
    vector<Complex> c;
    Polynomial(vector<int> &a) {
        int n = a.size();
        c.resize(n);
        for (int i = 0; i < n; i++) {
            c[i] = Complex(a[i], 0);
        }
        fft(c, n, 1);
    }
    void change(vector<Complex> &a, int n) {
        for (int i = 1, j = n / 2; i < n - 1; i++) {
            if (i < j) swap(a[i], a[j]);
            int k = n / 2;
            while (j >= k) {
                j -= k;
                k /= 2;
            }
            if (j < k) j += k;
        }
    }
    void fft(vector<Complex> &a, int n, int opt) {
        change(a, n);
        for (int h = 2; h <= n; h *= 2) {
            Complex wn(cos(2 * PI / h), sin(opt * 2 * PI / h));
            for (int j = 0; j < n; j += h) {
                Complex w(1, 0);
                for (int k = j; k < j + h / 2; k++) {
                    Complex u = a[k];
                    Complex t = w * a[k + h / 2];
                    a[k] = u + t;
                    a[k + h / 2] = u - t;
                    w = w * wn;
                }
            }
        }
        if (opt == -1) {
            for (int i = 0; i < n; i++) {
                a[i].x /= n;
            }
        }
    }
};
```

快速数论变换 NTT

$\mathcal{O}(N \log N)$。

```
struct Polynomial {
    vector<Z> z;
    vector<int> r;
    Polynomial(vector<int> &a) {
        int n = a.size();
        z.resize(n);
        r.resize(n);
        for (int i = 0; i < n; i++) {
            z[i] = a[i];
            r[i] = (i & 1) * (n / 2) + r[i / 2] / 2;
        }
        ntt(z, n, 1);
    }
    LL power(LL a, int b) {
        LL res = 1;
        for (; b; b /= 2, a = a * a % mod) {
            if (b % 2) {
                res = res * a % mod;
            }
        }
        return res;
    }
    void ntt(vector<Z> &a, int n, int opt) {
        for (int i = 0; i < n; i++) {
            if (r[i] < i) {
                swap(a[i], a[r[i]]);
            }
        }
        for (int k = 2; k <= n; k *= 2) {
            Z gn = power(3, (mod - 1) / k);
            for (int i = 0; i < n; i += k) {
                Z g = 1;
                for (int j = 0; j < k / 2; j++, g *= gn) {
                    Z t = a[i + j + k / 2] * g;
                    a[i + j + k / 2] = a[i + j] - t;
                    a[i + j] = a[i + j] + t;
                }
            }
        }
        if (opt == -1) {
            reverse(a.begin() + 1, a.end());
            Z inv = power(n, mod - 2);
            for (int i = 0; i < n; i++) {
                a[i] *= inv;
            }
        }
    }
};
```

拉格朗日插值

$n+1$ 个点可以唯一确定一个最高为 n 次的多项式。普通情况：$f(k)=\sum_{i=1}^{n+1} y_i \prod_{i\neq j} \frac{k-x[j]}{x[i]-x[j]}$ 。

```
struct Lagrange {
    int n;
    vector<Z> x, y, fac, invfac;
    Lagrange(int n) {
        this->n = n;
        x.resize(n + 3);
        y.resize(n + 3);
        fac.resize(n + 3);
        invfac.resize(n + 3);
        init(n);
    }
    void init(int n) {
        iota(x.begin(), x.end(), 0);
        for (int i = 1; i <= n + 2; i++) {
            Z t;
            y[i] = y[i - 1] + t.power(i, n);
        }
        fac[0] = 1;
        for (int i = 1; i <= n + 2; i++) {
            fac[i] = fac[i - 1] * i;
        }
        invfac[n + 2] = fac[n + 2].inv();
        for (int i = n + 1; i >= 0; i--) {
            invfac[i] = invfac[i + 1] * (i + 1);
        }
    }
    Z solve(LL k) {
        if (k <= n + 2) {
            return y[k];
        }
        vector<Z> sub(n + 3);
        for (int i = 1; i <= n + 2; i++) {
            sub[i] = k - x[i];
        }
        vector<Z> mul(n + 3);
        mul[0] = 1;
        for (int i = 1; i <= n + 2; i++) {
            mul[i] = mul[i - 1] * sub[i];
        }
        Z ans = 0;
        for (int i = 1; i <= n + 2; i++) {
            ans = ans + y[i] * mul[n + 2] * sub[i].inv() * pow(-1, n + 2 - i) * invfac[i - 1] *
                            invfac[n + 2 - i];
        }
        return ans;
    }
```

```
};
```

结论 from LuanXR

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

常用结论

杂

- 求 $B_i=\sum\limits_{k=i}^{n}C_k^iA_k$，即 $B_i=\dfrac{1}{i!}\sum\limits_{k=i}^{n}\dfrac{1}{(k-i)!}\cdot k!A_k$，反转后卷积。
- NTT中，$\omega_n=$ `qpow(G,(mod-1)/n))`。
- 遇到 $\sum\limits_{i=0}^{n}[i\%k=0]f(i)$ 可以转换为 $\sum\limits_{i=0}^{n}\dfrac{1}{k}\sum\limits_{j=0}^{k-1}(\omega_k^i)^jf(i)$ 。(单位根卷积)
- 广义二项式定理 $(1+x)^{\alpha}=\sum\limits_{i=0}^{\infty}\dbinom{n}{\alpha}x^i$ 。

普通生成函数 / OGF

- 普通生成函数：$A(x)=a_0+a_1x+a_2x^2+\ldots=\langle a_0,a_1,a_2,\ldots\rangle$；
- $1+x^k+x^{2k}+\ldots=\dfrac{1}{1-x^k}$；
- 取对数后 $=-\ln(1-x^k)=\sum\limits_{i=1}^{\infty}\dfrac{1}{i}x^{ki}$ 即 $\sum\limits_{i=1}^{\infty}\dfrac{1}{i}x^i\otimes x^k$（polymul_special）；
- $x+\dfrac{x^2}{2}+\dfrac{x^3}{3}+\ldots=-\ln(1-x)$；
- $1+x+x^2+\ldots+x^{m-1}=\dfrac{1-x^m}{1-x}$；
- $1+2x+3x^2+\ldots=\dfrac{1}{(1-x)^2}$（借用导数，$nx^{n-1}=(x^n)'$）；
- $C_m^0+C_m^1x+C_m^2x^2+\ldots+C_m^mx^m=(1+x)^m$（二项式定理）；
- $C_m^0+C_{m+1}^1x^1+C_{m+2}^2x^2+\ldots=\dfrac{1}{(1-x)^{m+1}}$（归纳法证明）；
- $\sum\limits_{n=0}^{\infty}F_nx^n=\dfrac{(F_1-F_0)x+F_0}{1-x-x^2}$（F 为斐波那契数列，列方程 $G(x)=xG(x)+x^2G(x)+(F_1-F_0)x+F_0$）；
- $\sum\limits_{n=0}^{\infty}H_nx^n=\dfrac{1-\sqrt{n-4x}}{2x}$（H 为卡特兰数）；
- 前缀和 $\sum\limits_{n=0}^{\infty}s_nx^n=\dfrac{1}{1-x}f(x)$；
- 五边形数定理：$\prod\limits_{i=1}^{\infty}(1-x^i)=\sum\limits_{k=0}^{\infty}(-1)^kx^{\frac{1}{2}k(3k\pm1)}$。

指数生成函数 / EGF

- 指数生成函数：$A(x)=a_0+a_1x+a_2\dfrac{x^2}{2!}+a_3\dfrac{x^3}{3!}+\ldots=\langle a_0,a_1,a_2,a_3,\ldots\rangle$；
- 普通生成函数转换为指数生成函数：系数乘以 $n!$；
- $1+x+\dfrac{x^2}{2!}+\dfrac{x^3}{3!}+\ldots=\exp x$；
- 长度为 n 的循环置换数为 $P(x)=-\ln(1-x)$，长度为 n 的置换数为 $\exp P(x)=\dfrac{1}{1-x}$（注意是**指数**生成函数）
    - n 个点的生成树个数是 $P(x)=\sum\limits_{n=1}^{\infty}n^{n-2}\dfrac{x^n}{n!}$，n 个点的生成森林个数是 $\exp P(x)$；
    - n 个点的无向连通图个数是 $P(x)$，n 个点的无向图个数是 $\exp P(x)=\sum\limits_{n=0}^{\infty}2^{\frac{1}{2}n(n-1)}\dfrac{x^n}{n!}$；
    - 长度为 $n(n\geq 2)$ 的循环置换数是 $P(x)=-\ln(1-x)-x$，长度为 n 的错排数是 $\exp P(x)$。

/END/