

多项式

线性凸包

```

1 struct Line {
2     i64 a, b, r;
3     bool operator<(Line l) { return pair(a, b) > pair(l.a, l.b); }
4     bool operator<(i64 x) { return r < x; }
5 };
6 struct Lines : vector<Line> {
7     static constexpr i64 inf = numeric_limits<i64>::max();
8     Lines(i64 a, i64 b) : vector<Line>{{a, b, inf}} {}
9     Lines(vector<Line>& lines) {
10         if (not ranges::is_sorted(lines, less())) ranges::sort(lines, less());
11         for (auto [a, b, _] : lines) {
12             for (; not empty(); pop_back()) {
13                 if (back().a == a) continue;
14                 i64 da = back().a - a, db = b - back().b;
15                 back().r = db / da - (db < 0 and db % da);
16                 if (size() == 1 or back().r > end()[-2].r) break;
17             }
18             emplace_back(a, b, inf);
19         }
20     }
21     Lines operator+(Lines& lines) {
22         vector<Line> res(size() + lines.size());
23         ranges::merge(*this, lines, res.begin(), less());
24         return Lines(res);
25     }
26     i64 min(i64 x) {
27         auto [a, b, _] = *lower_bound(begin(), end(), x, less());
28         return a * x + b;
29     }
30 };

```

多项式封装

```

1 template<int P = 998244353> struct Poly : public vector<MInt<P>> {
2     using Value = MInt<P>;
3
4     Poly() : vector<Value>() {}
5     explicit constexpr Poly(int n) : vector<Value>(n) {}
6
7     explicit constexpr Poly(const vector<Value> &a) : vector<Value>(a) {}
8     constexpr Poly(const initializer_list<Value> &a) : vector<Value>(a) {}
9
10    template<class InputIt, class = _RequireInputIter<InputIt>>
11    explicit constexpr Poly(InputIt first, InputIt last) : vector<Value>(first, last) {}
12 };

```

```

13     template<class F> explicit constexpr Poly(int n, F f) : vector<Value>(n) {
14         for (int i = 0; i < n; i++) {
15             (*this)[i] = f(i);
16         }
17     }
18
19     constexpr Poly shift(int k) const {
20         if (k >= 0) {
21             auto b = *this;
22             b.insert(b.begin(), k, 0);
23             return b;
24         } else if (this->size() <= -k) {
25             return Poly();
26         } else {
27             return Poly(this->begin() + (-k), this->end());
28         }
29     }
30     constexpr Poly trunc(int k) const {
31         Poly f = *this;
32         f.resize(k);
33         return f;
34     }
35     constexpr friend Poly operator+(const Poly &a, const Poly &b) {
36         Poly res(max(a.size(), b.size()));
37         for (int i = 0; i < a.size(); i++) {
38             res[i] += a[i];
39         }
40         for (int i = 0; i < b.size(); i++) {
41             res[i] += b[i];
42         }
43         return res;
44     }
45     constexpr friend Poly operator-(const Poly &a, const Poly &b) {
46         Poly res(max(a.size(), b.size()));
47         for (int i = 0; i < a.size(); i++) {
48             res[i] += a[i];
49         }
50         for (int i = 0; i < b.size(); i++) {
51             res[i] -= b[i];
52         }
53         return res;
54     }
55     constexpr friend Poly operator-(const Poly &a) {
56         vector<Value> res(a.size());
57         for (int i = 0; i < int(res.size()); i++) {
58             res[i] = -a[i];
59         }
60         return Poly(res);
61     }
62     constexpr friend Poly operator*(Poly a, Poly b) {
63         if (a.size() == 0 || b.size() == 0) {
64             return Poly();
65         }

```

```

66     if (a.size() < b.size()) {
67         swap(a, b);
68     }
69     int n = 1, tot = a.size() + b.size() - 1;
70     while (n < tot) {
71         n *= 2;
72     }
73     if (((P - 1) & (n - 1)) != 0 || b.size() < 128) {
74         Poly c(a.size() + b.size() - 1);
75         for (int i = 0; i < a.size(); i++) {
76             for (int j = 0; j < b.size(); j++) {
77                 c[i + j] += a[i] * b[j];
78             }
79         }
80         return c;
81     }
82     a.resize(n);
83     b.resize(n);
84     dft(a);
85     dft(b);
86     for (int i = 0; i < n; ++i) {
87         a[i] *= b[i];
88     }
89     idft(a);
90     a.resize(tot);
91     return a;
92 }
93 constexpr friend Poly operator*(Value a, Poly b) {
94     for (int i = 0; i < int(b.size()); i++) {
95         b[i] *= a;
96     }
97     return b;
98 }
99 constexpr friend Poly operator*(Poly a, Value b) {
100     for (int i = 0; i < int(a.size()); i++) {
101         a[i] *= b;
102     }
103     return a;
104 }
105 constexpr friend Poly operator/(Poly a, Value b) {
106     for (int i = 0; i < int(a.size()); i++) {
107         a[i] /= b;
108     }
109     return a;
110 }
111 constexpr Poly &operator+=(Poly b) {
112     return (*this) = (*this) + b;
113 }
114 constexpr Poly &operator-=(Poly b) {
115     return (*this) = (*this) - b;
116 }
117 constexpr Poly &operator*=(Poly b) {
118     return (*this) = (*this) * b;

```

```

119     }
120     constexpr Poly &operator*=(Value b) {
121         return (*this) = (*this) * b;
122     }
123     constexpr Poly &operator/=(Value b) {
124         return (*this) = (*this) / b;
125     }
126     constexpr Poly deriv() const {
127         if (this->empty()) {
128             return Poly();
129         }
130         Poly res(this->size() - 1);
131         for (int i = 0; i < this->size() - 1; ++i) {
132             res[i] = (i + 1) * (*this)[i + 1];
133         }
134         return res;
135     }
136     constexpr Poly integr() const {
137         Poly res(this->size() + 1);
138         for (int i = 0; i < this->size(); ++i) {
139             res[i + 1] = (*this)[i] / (i + 1);
140         }
141         return res;
142     }
143     constexpr Poly inv(int m) const {
144         Poly x{(*this)[0].inv()};
145         int k = 1;
146         while (k < m) {
147             k *= 2;
148             x = (x * (Poly{2} - trunc(k) * x)).trunc(k);
149         }
150         return x.trunc(m);
151     }
152     constexpr Poly log(int m) const {
153         return (deriv() * inv(m)).integr().trunc(m);
154     }
155     constexpr Poly exp(int m) const {
156         Poly x{1};
157         int k = 1;
158         while (k < m) {
159             k *= 2;
160             x = (x * (Poly{1} - x.log(k) + trunc(k))).trunc(k);
161         }
162         return x.trunc(m);
163     }
164     constexpr Poly pow(int k, int m) const {
165         int i = 0;
166         while (i < this->size() && (*this)[i] == 0) {
167             i++;
168         }
169         if (i == this->size() || 1LL * i * k >= m) {
170             return Poly(m);
171         }

```

```

172     value v = (*this)[i];
173     auto f = shift(-i) * v.inv();
174     return (f.log(m - i * k) * k).exp(m - i * k).shift(i * k) * power(v, k);
175 }
176 constexpr Poly sqrt(int m) const {
177     Poly x{1};
178     int k = 1;
179     while (k < m) {
180         k *= 2;
181         x = (x + (trunc(k) * x.inv(k)).trunc(k)) * CInv<2, P>;
182     }
183     return x.trunc(m);
184 }
185 constexpr Poly mult(Poly b) const {
186     if (b.size() == 0) {
187         return Poly();
188     }
189     int n = b.size();
190     reverse(b.begin(), b.end());
191     return ((*this) * b).shift(-(n - 1));
192 }
193 constexpr vector<Value> eval(vector<Value> x) const {
194     if (this->size() == 0) {
195         return vector<Value>(x.size(), 0);
196     }
197     const int n = max(x.size(), this->size());
198     vector<Poly> q(4 * n);
199     vector<Value> ans(x.size());
200     x.resize(n);
201     function<void(int, int, int)> build = [&](int p, int l, int r) {
202         if (r - l == 1) {
203             q[p] = Poly{1, -x[l]};
204         } else {
205             int m = (l + r) / 2;
206             build(2 * p, l, m);
207             build(2 * p + 1, m, r);
208             q[p] = q[2 * p] * q[2 * p + 1];
209         }
210     };
211     build(1, 0, n);
212     function<void(int, int, int, const Poly &)> work = [&](int p, int l, int r,
213                                                         const Poly &num) {
214         if (r - l == 1) {
215             if (l < int(ans.size())) {
216                 ans[l] = num[0];
217             }
218         } else {
219             int m = (l + r) / 2;
220             work(2 * p, l, m, num.mult(q[2 * p + 1]).resize(m - 1));
221             work(2 * p + 1, m, r, num.mult(q[2 * p]).resize(r - m));
222         }
223     };
224     work(1, 0, n, mult(q[1].inv(n)));

```

```

225         return ans;
226     }
227 };

```

离散傅里叶变换 dft 与其逆变换 idft

```

1  vector<int> rev;
2  template<int P> vector<MInt<P>> roots{0, 1};
3
4  template<int P> constexpr MInt<P> findPrimitiveRoot() {
5      MInt<P> i = 2;
6      int k = __builtin_ctz(P - 1);
7      while (true) {
8          if (power(i, (P - 1) / 2) != 1) {
9              break;
10         }
11         i += 1;
12     }
13     return power(i, (P - 1) >> k);
14 }
15
16 template<int P> constexpr MInt<P> primitiveRoot = findPrimitiveRoot<P>();
17 template<> constexpr MInt<998244353> primitiveRoot<998244353>{31};
18
19 template<int P> constexpr void dft(vector<MInt<P>> &a) { // 离散傅里叶变换
20     int n = a.size();
21
22     if (int(rev.size()) != n) {
23         int k = __builtin_ctz(n) - 1;
24         rev.resize(n);
25         for (int i = 0; i < n; i++) {
26             rev[i] = rev[i >> 1] >> 1 | (i & 1) << k;
27         }
28     }
29
30     for (int i = 0; i < n; i++) {
31         if (rev[i] < i) {
32             swap(a[i], a[rev[i]]);
33         }
34     }
35     if (roots<P>.size() < n) {
36         int k = __builtin_ctz(roots<P>.size());
37         roots<P>.resize(n);
38         while ((1 << k) < n) {
39             auto e = power(primitiveRoot<P>, 1 << (__builtin_ctz(P - 1) - k - 1));
40             for (int i = 1 << (k - 1); i < (1 << k); i++) {
41                 roots<P>[2 * i] = roots<P>[i];
42                 roots<P>[2 * i + 1] = roots<P>[i] * e;
43             }
44             k++;
45         }
46     }

```

```

47     for (int k = 1; k < n; k *= 2) {
48         for (int i = 0; i < n; i += 2 * k) {
49             for (int j = 0; j < k; j++) {
50                 MInt<P> u = a[i + j];
51                 MInt<P> v = a[i + j + k] * roots<P>[k + j];
52                 a[i + j] = u + v;
53                 a[i + j + k] = u - v;
54             }
55         }
56     }
57 }
58 template<int P> constexpr void idft(vector<MInt<P>> &a) { // 逆变换
59     int n = a.size();
60     reverse(a.begin() + 1, a.end());
61     dft(a);
62     MInt<P> inv = (1 - P) / n;
63     for (int i = 0; i < n; i++) {
64         a[i] *= inv;
65     }
66 }

```

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。

```

1  template<int P = 998244353> Poly<P> berlekampMassey(const Poly<P> &s) {
2      Poly<P> c;
3      Poly<P> oldc;
4      int f = -1;
5      for (int i = 0; i < s.size(); i++) {
6          auto delta = s[i];
7          for (int j = 1; j <= c.size(); j++) {
8              delta -= c[j - 1] * s[i - j];
9          }
10         if (delta == 0) {
11             continue;
12         }
13         if (f == -1) {
14             c.resize(i + 1);
15             f = i;
16         } else {
17             auto d = oldc;
18             d *= -1;
19             d.insert(d.begin(), 1);
20             MInt<P> df1 = 0;
21             for (int j = 1; j <= d.size(); j++) {
22                 df1 += d[j - 1] * s[f + 1 - j];
23             }
24             assert(df1 != 0);
25             auto coef = delta / df1;
26             d *= coef;
27             Poly<P> zeros(i - f - 1);

```

```

28         zeros.insert(zeros.end(), d.begin(), d.end());
29         d = zeros;
30         auto temp = c;
31         c += d;
32         if (i - temp.size() > f - oldc.size()) {
33             oldc = temp;
34             f = i;
35         }
36     }
37 }
38 c *= -1;
39 c.insert(c.begin(), 1);
40 return c;
41 }

```

Linear-Recurrence 算法

```

1  template<int P = 998244353> MInt<P> linearRecurrence(Poly<P> p, Poly<P> q, i64 n) {
2      int m = q.size() - 1;
3      while (n > 0) {
4          auto newq = q;
5          for (int i = 1; i <= m; i += 2) {
6              newq[i] *= -1;
7          }
8          auto newp = p * newq;
9          newq = q * newq;
10         for (int i = 0; i < m; i++) {
11             p[i] = newp[i * 2 + n % 2];
12         }
13         for (int i = 0; i <= m; i++) {
14             q[i] = newq[i * 2];
15         }
16         n /= 2;
17     }
18     return p[0] / q[0];
19 }

```

快速傅里叶变换 FFT

$\mathcal{O}(N \log N)$ 。

```

1  struct Polynomial {
2      constexpr static double PI = acos(-1);
3      struct Complex {
4          double x, y;
5          Complex(double _x = 0.0, double _y = 0.0) {
6              x = _x;
7              y = _y;
8          }
9          Complex operator-(const Complex &rhs) const {
10             return Complex(x - rhs.x, y - rhs.y);

```



```

11     }
12     Complex operator+(const Complex &rhs) const {
13         return Complex(x + rhs.x, y + rhs.y);
14     }
15     Complex operator*(const Complex &rhs) const {
16         return Complex(x * rhs.x - y * rhs.y, x * rhs.y + y * rhs.x);
17     }
18 };
19 vector<Complex> c;
20 Polynomial(vector<int> &a) {
21     int n = a.size();
22     c.resize(n);
23     for (int i = 0; i < n; i++) {
24         c[i] = Complex(a[i], 0);
25     }
26     fft(c, n, 1);
27 }
28 void change(vector<Complex> &a, int n) {
29     for (int i = 1, j = n / 2; i < n - 1; i++) {
30         if (i < j) swap(a[i], a[j]);
31         int k = n / 2;
32         while (j >= k) {
33             j -= k;
34             k /= 2;
35         }
36         if (j < k) j += k;
37     }
38 }
39 void fft(vector<Complex> &a, int n, int opt) {
40     change(a, n);
41     for (int h = 2; h <= n; h *= 2) {
42         Complex wn(cos(2 * PI / h), sin(opt * 2 * PI / h));
43         for (int j = 0; j < n; j += h) {
44             Complex w(1, 0);
45             for (int k = j; k < j + h / 2; k++) {
46                 Complex u = a[k];
47                 Complex t = w * a[k + h / 2];
48                 a[k] = u + t;
49                 a[k + h / 2] = u - t;
50                 w = w * wn;
51             }
52         }
53     }
54     if (opt == -1) {
55         for (int i = 0; i < n; i++) {
56             a[i].x /= n;
57         }
58     }
59 }
60 };

```

快速数论变换 NTT

$\mathcal{O}(N \log N)$ 。

```

1  struct Polynomial {
2      vector<Z> z;
3      vector<int> r;
4      Polynomial(vector<int> &a) {
5          int n = a.size();
6          z.resize(n);
7          r.resize(n);
8          for (int i = 0; i < n; i++) {
9              z[i] = a[i];
10             r[i] = (i & 1) * (n / 2) + r[i / 2] / 2;
11         }
12         ntt(z, n, 1);
13     }
14     LL power(LL a, int b) {
15         LL res = 1;
16         for (; b; b /= 2, a = a * a % mod) {
17             if (b % 2) {
18                 res = res * a % mod;
19             }
20         }
21         return res;
22     }
23     void ntt(vector<Z> &a, int n, int opt) {
24         for (int i = 0; i < n; i++) {
25             if (r[i] < i) {
26                 swap(a[i], a[r[i]]);
27             }
28         }
29         for (int k = 2; k <= n; k *= 2) {
30             Z gn = power(3, (mod - 1) / k);
31             for (int i = 0; i < n; i += k) {
32                 Z g = 1;
33                 for (int j = 0; j < k / 2; j++, g *= gn) {
34                     Z t = a[i + j + k / 2] * g;
35                     a[i + j + k / 2] = a[i + j] - t;
36                     a[i + j] = a[i + j] + t;
37                 }
38             }
39         }
40         if (opt == -1) {
41             reverse(a.begin() + 1, a.end());
42             Z inv = power(n, mod - 2);
43             for (int i = 0; i < n; i++) {
44                 a[i] *= inv;
45             }
46         }
47     }
48 };

```

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：
$$f(k) = \sum_{i=1}^{n+1} y_i \prod_{i \neq j} \frac{k - x[j]}{x[i] - x[j]}。$$

```

1 struct Lagrange {
2     int n;
3     vector<Z> x, y, fac, invfac;
4     Lagrange(int n) {
5         this->n = n;
6         x.resize(n + 3);
7         y.resize(n + 3);
8         fac.resize(n + 3);
9         invfac.resize(n + 3);
10        init(n);
11    }
12    void init(int n) {
13        iota(x.begin(), x.end(), 0);
14        for (int i = 1; i <= n + 2; i++) {
15            Z t;
16            y[i] = y[i - 1] + t.power(i, n);
17        }
18        fac[0] = 1;
19        for (int i = 1; i <= n + 2; i++) {
20            fac[i] = fac[i - 1] * i;
21        }
22        invfac[n + 2] = fac[n + 2].inv();
23        for (int i = n + 1; i >= 0; i--) {
24            invfac[i] = invfac[i + 1] * (i + 1);
25        }
26    }
27    Z solve(LL k) {
28        if (k <= n + 2) {
29            return y[k];
30        }
31        vector<Z> sub(n + 3);
32        for (int i = 1; i <= n + 2; i++) {
33            sub[i] = k - x[i];
34        }
35        vector<Z> mul(n + 3);
36        mul[0] = 1;
37        for (int i = 1; i <= n + 2; i++) {
38            mul[i] = mul[i - 1] * sub[i];
39        }
40        Z ans = 0;
41        for (int i = 1; i <= n + 2; i++) {
42            ans = ans + y[i] * mul[n + 2] * sub[i].inv() * pow(-1, n + 2 - i) *
invfac[i - 1] *
43                invfac[n + 2 - i];
44        }
45        return ans;
46    }

```

47 | };

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

$$1. \frac{1}{1-x} = 1 + x + x^2 + x^3 + \dots = \sum_{n=0}^{\infty} x^n$$

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

$$3. \frac{1}{1-x^3} = 1 + x^3 + x^6 + \dots$$

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

$$5. e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

$$6. e^{-x} = 1 - \frac{x^1}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} + \dots$$

$$7. \frac{e^x + e^{-x}}{2} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \dots$$

$$8. \frac{e^x - e^{-x}}{2} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \dots$$

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

$$(1+x)^n = \sum_{i=0}^n \binom{n}{i} x^i, \text{ 因 } i > n, \binom{n}{i} = 0.$$

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

普通生成函数 / OGF

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

指数生成函数 / EGF

- 指数生成函数: $A(x) = a_0 + a_1x + a_2 \frac{x^2}{2!} + a_3 \frac{x^3}{3!} + \dots = \langle a_0, a_1, a_2, a_3, \dots \rangle$;
- 普通生成函数转换为指数生成函数: 系数乘以 $n!$;
- $1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots = \exp x$;
- 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 $\exp P(x) = \frac{1}{1 - x}$ (注意是指数生成函数)
 - n 个点的生成树个数是 $P(x) = \sum_{n=1}^{\infty} n^{n-2} \frac{x^n}{n!}$, 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 $\exp P(x) = \sum_{n=0}^{\infty} 2^{\frac{1}{2}n(n-1)} \frac{x^n}{n!}$;
 - 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/