算法竞赛小记

Competitive Programming Library

Calculatelove



github.com/Calculatelove/cp-library-v4-LaTeX

LAST COMMIT

2026-11-19 b71bfa3

feat: 2026 CCPC 厦门站前更新

目录

0x11 基础 Misc 6

常用常数表 6
常用公式 6
日期公式 6
xorshift 6
xorsum 7
Sum of Powers 7
常用配置 7
makefile 7
pragma 7
ulimit 7
Linux 对拍脚本 7
Windows 对拍程序 7
随机数据生成 8
生成区间 $[l,r]$ 中的随机整数 8
生成区间 $[l,r)$ 中的随机浮点数 8
生成互不相同的随机整数 8
生成随机树 8
常用库函数重载 8
i128 8
整数除法精确取整 9
取最值 9
取模 9
精确开根号 10
精确取对数 10
快读 10
超级快读 10
高精度 11
分数类 12

0x12 基础 算法 13

主定理 13
快速乘 13
正道：i128 13
魔道：long double 13
光速幂 14

0x21 字符串 I 14

哈希 14
字符串哈希 14
集合哈希 15
kmp 15
border 理论 15
Z Algorithm 16
manacher 17
Lyndon 分解 18
最小表示法 19
带通配符的字符串匹配 19

0x22 字符串 II 19

Trie 19
AC 自动机 21
SA 23
SAM 24
PAM 28
Palindrome Series 29

0x31 数学 多项式 31

拉格朗日插值 31

FFT 32
NTT 34
MTT 35
三模 NTT 35
拆系数 FFT 37
Trick：翻转卷积 38
Trick：积卷积 38
Trick：多项式点乘向量 38
Model：分治卷积 38
Model：半在线卷积 39
Model：树上路径长度统计 40
多项式基本运算 40
多项式除法（暴力） 40
多项式导数/积分 40
多项式牛顿迭代 40
多项式求逆 40
多项式开方 41
多项式带余除法 41
多项式 ln/exp 41
多项式快速幂 42
少项式快速幂 42
转置原理 43
多项式多点求值 43
多项式快速插值 43
多项式平移 44
Chirp Z-Transform 45
多项式全家桶 45
Bostan-Mori 50
分式远处系数 50
分式远处区间系数 51
常系数齐次线性递推 51
Fiduccia 算法 51
递推式转分式 51
整式递推 52
应用：快速阶乘算法 52
多项式复合 52
拉格朗日反演 52
Power Projection 53
多项式复合逆 53

0x32 数学 集合幂级数 54

FWT 54
DWT 54
IDWT 55
位运算卷积 55
子集 dp 56
子集卷积 56

0x33 数学 组合计数 58

排列组合 58
组合数常见恒等式 58
插板法 59
Lucas 定理 59
组合数常见求法 59
求法 1：递推 59
求法 2：预处理阶乘与阶乘逆元 59
求法 3：暴力统计 60
求法 4：Lucas 定理 60
求法 5：exLucas 定理 60
求法 6：分段打表 60
邪修：快速阶乘算法 61
求法 7：阶乘分解质因数 61
斯特林数 61
第一类斯特林数 61

第二类斯特林数 61
卡特兰数 61
贝尔数 62
斐波那契数列 62
斐波那契数列常见恒等式 63
斐波那契数列模意义下的循环节 63
普通生成函数 64
指数生成函数 64
符号化方法 65
图论计数 65
有标号无向连通图计数 65
有标号无向欧拉图计数 65
有标号无向二分图计数 66
有标号 DAG 计数 66
有标号弱连通 DAG 计数 67
有标号强连通图计数 67
Misc 68
Raney 引理 68

0x34 数学 初等数论–质数 69

质数 69
阶乘的质因数分解 69
埃氏筛 69
线性筛 69
狄利克雷前缀和 70
Miller-Rabin 素性测试 70
Pollard-Rho 质因数分解 71

0x34 数学 初等数论–约数 73

约数 73
最大公约数与最小公倍数 73
gcd 和 lcm 的常见恒等式 73
最大公约数常见求法 74
求法 1：辗转相除法 74
求法 2：更相减损术 74
求法 3：递推 75
求法 4：基于值域预处理的快速 gcd 75
类欧几里得算法 76

0x34 数学 初等数论–取整 77

取整 77
取整的常见恒等式 77
数论分块 77
下取整数论分块 77
上取整数论分块 78
多维数论分块 78
积性函数与狄利克雷卷积 78
常见积性函数（及性质） 78
线性筛筛积性函数 79
狄利克雷卷积 80
常见狄利克雷卷积式 80
杜教筛 81
Min_25 筛 82

0x34 数学 初等数论–同余 82

同余 82
费马小定理与欧拉定理 83
模意义下逆元 83
模质数意义下逆元 83
单个逆元的求解 83
多个逆元的求解 84
预处理线性逆元 84
线性同余方程 84
线性同余方程组 85

阶与原根 . . . 86
离散对数 . . . 88
高次同余方程 . . . 88
Misc . . . 90
平方差 . . . 90

0x35 数学 离散数学 90

0x36 数学 线性代数 90

0x37 数学 概率论 90

0x38 数学 博弈论 90

0x41 数据结构 序列结构 90

ST 表 . . . 90
并查集 . . . 91
线段树 . . . 91
扫描线 . . . 91
势能线段树 . . . 91
笛卡尔树 . . . 93
区间笛卡尔树 . . . 93

0x42 数据结构 树上结构 93

0x51 动态规划 状态设计 93

0x52 动态规划 转移优化 93

0x65 图论 二分图与网络流 93

最大流 . . . 93
Dinic . . . 93

0x71 计算几何 I 94

计算几何基础 . . . 94

0x72 计算几何 II 94

凸壳 . . . 94
凸包 . . . 95
闵可夫斯基和 . . . 95

0x11 基础 Misc

常用常数表

n	$\log_{10} n$	$n!$	$C(n, n/2)$	$\mathrm{LCM}(1 \ldots n)$	P_n
2	0.30103000	2	2	2	2
3	0.47712125	6	3	6	3
4	0.60205999	24	6	12	5
5	0.69897000	120	10	60	7
6	0.77815125	720	20	60	11
7	0.84509804	5040	35	420	15
8	0.90308999	40320	70	840	22
9	0.95424251	362880	126	2520	30
10	1	3628800	252	2520	42
11	1.04139269	39916800	462	27720	56
12	1.07918125	479001600	924	27720	77
15	1.17609126	1.31e12	6435	360360	176
20	1.30103000	2.43e18	184756	232792560	627
25	1.39794001	1.55e25	5200300	26771144400	1958
30	1.47712125	2.65e32	155117520	2.329e12	5604
P_n	37338_{40}	204226_{50}	966467_{60}	190569292_{100}	$1e9_{114}$

$n \leq$	10	100	1e3	1e4	1e5	1e6
$\max \omega(n)$	2	3	4	5	6	7
$\max d(n)$	4	12	32	64	128	240
$\pi(n)$	4	25	168	1229	9592	78498
$n \leq$	1e7	1e8	1e9	1e10	1e11	1e12
$\max \omega(n)$	8	8	9	10	10	11
$\max d(n)$	448	768	1344	2304	4032	6720
$\pi(n)$	664579	5761455	5.08e7	4.55e8	4.12e9	3.76e10
$n \leq$	1e13	1e14	1e15	1e16	1e17	1e18
$\max \omega(n)$	12	12	13	13	14	15
$\max d(n)$	10752	17280	26880	41472	64512	103680
$\pi(n)$	Prime number theorem: $\pi(x) \sim x/\log(x)$					

Random Primes (`1e9`, `1e18`)

```
902973697 1001936473 902562365550767401 1002761300631564491
913216937 1016373329 917480027360499821 1019296038923039081
925996493 1028328907 924926893934733409 1020486568805982493
930175307 1036706831 930715715438134229 1032496198455759913
940338211 1048962763 941086105034830319 1043831183499049909
955824229 1054557227 953518393255689421 1052292637121190469
962573639 1068163763 966041672021759747 1068066450752903981
972484789 1078252283 972685471134953333 1078187098370425103
983690579 1086023041 985128320139082481 1084139330090191657
999631453 1097345849 994883032981443269 1098492456730478171
```

常用公式

日期公式

0x11 日期公式.cpp

```cpp
// 返回从 1/1/1 到 y/m/d 的天数，由于 1/1/1 是周一，所以 mod 7 可以算出是周几
int getday(int y, int m, int d) {
    if (m < 3) y --, m += 12;
    return (365 * y + y / 4 - y / 100 + y / 400 + (153 * (m - 3) + 2) / 5 + d
     ↪  - 307);
}

// 返回从 1/1/1 开始经过 n 天的日期（getday 的逆函数）
std::array<int, 3> date(int n) {
    int y, m, d;
    n += 429 + ((4 * n + 1227) / 146097 + 1) * 3 / 4;
    y = (4 * n - 489) / 1461;
    n -= y * 1461 / 4;
    m = (5 * n - 1) / 153;
    d = n - m * 153 / 5;
    if (-- m > 12) m -= 12, y ++;
    return {y, m, d};
}
```

xorshift

不适合直接作为哈希值进行结合。

0x11 xorshift.cpp

```cpp
u64 mask = std::mt19937_64(std::random_device{}())();
u64 xorshift(u64 x) {
    x ^= mask;
    x ^= x << 13, x ^= x >> 7, x ^= x << 17;
    x ^= mask;
    return x;
}
```

xorsum

$$\operatorname{xor}_{i=1}^{n} i = \begin{cases} n, & n \bmod 4 = 0 \\ 1, & n \bmod 4 = 1 \\ n+1, & n \bmod 4 = 2 \\ 0, & n \bmod 4 = 3 \end{cases}$$

0x11 xorsum.cpp

```cpp
i64 xorsum(i64 n) {
    int r = n % 4;
    if (r == 0) {
        return n;
    } else if (r == 1) {
        return 1;
    } else if (r == 2) {
        return n + 1;
    } else {
        return 0;
    }
}
```

Sum of Powers

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \quad \sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

$$\sum_{i=1}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

$$\sum_{i=1}^{n} i^5 = \frac{n^2(n+1)^2(2n^2+2n-1)}{12}$$

常用配置

makefile

```
CXXFLAGS = -std=c++23 -O2 -Wall -fsanitize=undefined

# -D_GLIBCXX_DEBUG : STL debug mode
# -fsanitize=address : 内存错误检查
# -fsanitize=undefined : UB 检查
```

pragma

```cpp
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("avx2")
```

ulimit

```
ulimit -s 1048576 # 单位 KB
```

Linux 对拍脚本

需要 `chmod +x cmp.sh`，然后才能 `./cmp.sh`。

```
while true; do
    ./gen >0.in
    ./obj <0.in >1.out
    ./std <0.in >2.out
    let cnt=cnt+1; echo "Test $cnt:"
    if diff 1.out 2.out; then
        echo ac
    else
        echo wa
        break;
    fi
done
```

Windows 对拍程序

```cpp
#include <bits/stdc++.h>
int main() {
    int T = 100;
    for (int i = 1; i <= T; i ++) {
        system("gen.exe >0.in");
        system("obj.exe <0.in >1.out");
        system("std.exe <0.in >2.out");
        std::cout << "Test " << i << ":\n";
        if (system("fc 1.out 2.out")) {
            std::cout << "wa\n";
            break;
        } else {
            std::cout << "ac\n";
        }
    }
}
```

随机数据生成

生成区间 $[l,r]$ 中的随机整数

0x11 生成区间 [l,r] 中的随机整数.cpp

```cpp
std::mt19937_64 mtrand{std::random_device{}()};

// 生成区间 [l, r] 中的随机整数
int rand(int l, int r) {
    std::uniform_int_distribution<int> range(l, r);
    return range(mtrand);
}
```

生成区间 $[l,r)$ 中的随机浮点数

0x11 生成区间 [l,r) 中的随机浮点数.cpp

```cpp
std::mt19937_64 mtrand{std::random_device{}()};

// 生成区间 [l, r) 中的随机浮点数
double rand(double l, double r) {
    std::uniform_real_distribution<double> range(l, r);
    return range(mtrand);
}
```

生成互不相同的随机整数

0x11 生成互不相同的随机整数.cpp

```cpp
// 生成 n 个在区间 [l, r] 中互不相同的随机整数
auto GenSeq(int n, int l, int r) {
    assert(n <= r - l + 1);
    std::map<int, int> buc;
    std::vector<int> res(n);
    for (int i = 0; i < n; i ++) {
        do {
            res[i] = rand(l, r);
        } while (buc.contains(res[i]));
        buc[res[i]] = 1;
    }
    return res;
}
```

生成随机树

- 均匀随机树的期望直径为 $\mathcal{O}(\sqrt{n})$ 级别。

0x11 生成随机树.cpp

```cpp
// 生成 n 个点的随机树
auto GenTree(int n) {
    std::vector<std::pair<int, int>> res;
    std::vector<int> prufer(n - 1), deg(n + 1, 1);
    for (int i = 1; i <= n - 2; i ++) {
        prufer[i] = rand(1, n);
        deg[prufer[i]] ++;
    }
    int leaf = 0, p = 0;
    for (int i = 1; i <= n; i ++) {
        if (deg[i] == 1) {
            leaf = p = i;
            break;
        }
    }
    for (int i = 1; i <= n - 2; i ++) {
        int x = prufer[i];
        res.push_back({leaf, x});
        if (-- deg[x] == 1 && x < p) {
            leaf = x;
        } else {
            p ++;
            while (deg[p] != 1) p ++;
            leaf = p;
        }
    }
    if (n > 1) {
        res.push_back({leaf, n});
    }
    return res;
}
```

常用库函数重载

i128

重载了 `__int128` 的标准输入输出流。

0x11 i128.cpp

```cpp
using i128 = __int128;

// 重载 i128 标准输入流
std::istream &operator >> (std::istream &is, i128 &x) {
    std::string s;
    is >> s;
    int n = s.length(), f = s[0] == '-';
    x = 0;
    for (int i = f; i < n; i ++) {
        x = x * 10 + s[i] - '0';
    }
    if (f) x = -x;
    return is;
}
// 重载 i128 标准输出流
std::ostream &operator << (std::ostream &os, i128 x) {
    if (x < 0) x = -x, os << '-';
    std::string s;
    do {
        s += char(x % 10 + '0'), x /= 10;
    } while (x);
    std::reverse(s.begin(), s.end());
    return os << s;
}
```

整数除法精确取整

0x11 整数除法精确取整.cpp

```cpp
// 整数除法 精确上取整
i64 ceilDiv(i64 n, i64 m) {
    // assert(m);
    if (m < 0) n = -n, m = -m;
    return n >= 0 ? (n + m - 1) / m : n / m;
}

// 整数除法 精确下取整
i64 floorDiv(i64 n, i64 m) {
    // assert(m);
    if (m < 0) n = -n, m = -m;
    return n >= 0 ? n / m : (n - m + 1) / m;
}
```

取最值

0x11 取最值.cpp

```cpp
template <class T>
inline void chmin(T &x, const T &y) {
    if (x > y) {
        x = y;
    }
}
template <class T>
inline void chmax(T &x, const T &y) {
    if (x < y) {
        x = y;
    }
}
```

取模

0x11 取模.cpp

```cpp
const int mod = 998244353; // 模数需要根据实际问题调整

template <class T>
inline int norm(T x) {
    x %= mod;
    return x < 0 ? x + mod : x;
}

inline void add(int &x, const int &y) {
    x += y; if (x >= mod) x -= mod;
}
inline void dec(int &x, const int &y) {
    x -= y; if (x < 0) x += mod;
}
inline void mul(int &x, const int &y) {
    x = 1ll * x * y % mod;
}
inline void neg(int &x) {
    if (x) x = mod - x;
}

template <class T>
constexpr int qpow(int a, T b, int p) {
    int ans = 1;
```

```
    for (; b; b >>= 1) {
        if (b & 1) ans = 1ll * ans * a % p;
        a = 1ll * a * a % p;
    }
    return ans;
}
```

精确开根号

0x11 精确开根号.cpp

```
// 精确开根号：找到最大整数 d，使得 d * d <= n
i64 isqrt(i64 n) {
    i64 d = sqrt(n);
    while (d * d > n) {
        d --;
    }
    while ((d + 1) * (d + 1) <= n) {
        d ++;
    }
    return d;
}
```

精确取对数

0x11 精确取对数.cpp

```
// 精确取对数（下取整）：找到最大整数 t，使得 a^t <= b
int ilog(i64 a, i64 b) {
    int t = 0;
    i64 v = 1;
    while (v <= b / a) {
        t ++;
        v *= a;
    }
    return t;
}

// 精确取对数（上取整）：找到最小整数 t，使得 a^t >= b
int iLog(i64 a, i64 b) {
    int t = 0;
    i64 v = 1;
    while (v < b) {
        t ++; if (v > b / a) break;
        v *= a;
    }
    return t;
}
```

快读

0x11 快读.cpp

```
// 读入一个整数（无需在意类型）
template <class T>
inline void read(T &x) {
    char s;
    bool opt;
    while (s = getchar(), (s < '0' || s > '9') && s != '-');
    x = (opt = s == '-') ? 0 : s - '0';
    while (s = getchar(), s >= '0' && s <= '9') x = x * 10 + s - '0';
    if (opt) x = -x;
}
```

超级快读

0x11 超级快读.cpp

```
/**
 * 注意，此代码的所有输出类函数，均不带“空格”与“换行”
 * 注意，虽然此代码使用了缓冲区，但不需要你手动清空，fastio 的析构函数会自动清空
 * 注意，此快读快写不能与 C/C++ 风格的 IO 混用，否则会导致 IO 的顺序混乱
 * 注意，如果你需要使用控制台调试，请在输入的最后使用 Ctrl+Z (Windows) 或 Ctrl+D (Linux/macOS) 手动输入 EOF 来结束输入
 */
struct fastio {
    static const int N = 1 << 20;

    char buf[N], *p1 = buf, *p2 = buf;
    #define gc() (p1 == p2 && (p2 = (p1 = buf) + fread(buf, 1, N, stdin), p1 == p2) ? EOF : *p1 ++)

    char pbuf[N], *pp = pbuf;
    void pc(const char &c) {
        if (pp - pbuf == N) fwrite(pbuf, 1, N, stdout), pp = pbuf;
        *pp ++ = c;
    }
```

```
    // 读入一个整数（无需在意类型）
    template <class T>
    void read(T &x) {
        char s;
        bool opt;
        while (s = gc(), (s < '0' || s > '9') && s != '-');
        x = (opt = s == '-') ? 0 : s - '0';
        while (s = gc(), s >= '0' && s <= '9') x = x * 10 + s - '0';
        if (opt) x = -x;
    }
    // 读入一个字符串
    void readString(std::string &x) {
        char s;
        while (s = gc(), std::isspace(s));
        x = s;
        while (s = gc(), !std::isspace(s)) x += s;
    }

    // 输出一个整数（无需在意类型）
    template <class T>
    void print(T x) {
        int top = 0; static int stk[40];
        if (x < 0) x = -x, pc('-');
        do {
            stk[++ top] = x % 10, x /= 10;
        } while (x);
        while (top) pc(stk[top --] + '0');
    }
    // 输出一个字符串（char 数组类型）
    void puts(const char *s) {
        while (*s) pc(*s), s ++;
    }
    // 输出一个字符串（std::string 类型）
    void puts(const std::string &s) {
        for (char ch : s) pc(ch);
    }

    ~fastio() {
        fwrite(pbuf, pp - pbuf, 1, stdout);
    }
} io;
```

高精度

0x11 高精度.cpp

```
// 高精度
struct bint {
    static const int SIZE = 4000; // 位数上限需要根据实际问题调整
    int len, num[SIZE];

    explicit bint(int len = 1) : len(len) {
        memset(num, 0, sizeof(num));
    };

    auto operator <=> (const bint &b) const {
        if (len ^ b.len) return len <=> b.len;
        for (int i = len; i >= 1; i --)
            if (num[i] ^ b.num[i]) return num[i] <=> b.num[i];
        return std::strong_ordering::equal;
    }
    auto operator == (const bint &b) const {
        if (len ^ b.len) return false;
        for (int i = len; i >= 1; i --)
            if (num[i] ^ b.num[i]) return false;
        return true;
    }

    // 高精度加法
    bint operator + (const bint &b) const {
        bint c(std::max(len, b.len));
        for (int i = 1; i <= c.len; i ++) c.num[i] = num[i] + b.num[i];
        for (int i = 1; i <= c.len; i ++)
            if (c.num[i] >= 10) c.num[i + 1] ++, c.num[i] -= 10;
        while (c.num[c.len + 1]) {
            c.len ++;
            if (c.num[c.len] >= 10) c.num[c.len + 1] ++, c.num[c.len] -= 10;
        }
        return c;
    }
    // 高精度减法
    bint operator - (const bint &b) const {
        bint c(len);
        for (int i = 1; i <= c.len; i ++) c.num[i] = num[i] - b.num[i];
        for (int i = 1; i <= c.len; i ++)
            if (c.num[i] < 0) c.num[i + 1] --, c.num[i] += 10;
        while (c.len > 1 && !c.num[c.len]) c.len --;
        return c;
```

```
    }

    // 高精度乘以低精度
    bint operator * (const int &b) const {
        bint c(len);
        i64 x = 0;
        for (int i = 1; i <= len; i ++) {
            x += 1ll * num[i] * b;
            c.num[i] = x % 10, x /= 10;
        }
        while (x) {
            c.len ++;
            c.num[c.len] = x % 10, x /= 10;
        }
        while (c.len > 1 && !c.num[c.len]) c.len --;
        return c;
    }
    // 高精度乘以高精度
    bint operator * (const bint &b) const {
        bint c(len + b.len);
        for (int i = 1; i <= len; i ++) {
            for (int j = 1; j <= b.len; j ++) {
                c.num[i + j - 1] += num[i] * b.num[j];
            }
        }
        for (int i = 1; i <= c.len; i ++) {
            c.num[i + 1] += c.num[i] / 10;
            c.num[i] %= 10;
        }
        while (c.len > 1 && !c.num[c.len]) c.len --;
        return c;
    }

    // 高精度除以低精度（带余除法）
    std::pair<bint, int> divmod(const int &b) const {
        bint c(len);
        i64 r = 0;
        for (int i = len; i >= 1; i --) {
            c.num[i] = (r * 10 + num[i]) / b;
            r = (r * 10 + num[i]) % b;
        }
        while (c.len > 1 && !c.num[c.len]) c.len --;
        return {c, r};
    }
    // 高精度除以高精度（带余除法）
    std::pair<bint, bint> divmod(const bint &b) const {
        if (*this < b) return {bint(), *this};
        bint c(len - b.len + 1);
        bint r = *this;
        for (int i = c.len; i >= 1; i --) {
            bint tmp(b.len + i - 1);
            for (int j = 1; j <= b.len; j ++) {
                tmp.num[j + i - 1] = b.num[j];
            }
            while (tmp <= r) {
                r = r - tmp, c.num[i] ++;
            }
        }
        while (c.len > 1 && !c.num[c.len]) c.len --;
        return {c, r};
    }

    friend std::istream &operator >> (std::istream &is, bint &b) { // 默认没有前导 0
        std::string s;
        is >> s;
        b.len = s.length();
        memset(b.num, 0, sizeof(b.num));
        for (int i = 0; i < b.len; i ++) {
            b.num[b.len - i] = s[i] - '0';
        }
        return is;
    }
    friend std::ostream &operator << (std::ostream &os, const bint &b) {
        for (int i = b.len; i >= 1; i --) {
            os << b.num[i];
        }
        return os;
    }
};
```

分数类

0x11 分数类.cpp

```
// 分数类
struct frac {
    i64 p, q;
    frac() : p(0), q(1) {}
    frac(i64 p, i64 q) : p(p), q(q) {} // 必要时，在此处规范化分数（保证 q>0 以及 p,q 最简）
```

```cpp

    frac operator + (const frac &rhs) const {
        return frac(p * rhs.q + rhs.p * q, q * rhs.q);
    }
    frac operator - (const frac &rhs) const {
        return frac(p * rhs.q - rhs.p * q, q * rhs.q);
    }
    frac operator * (const frac &rhs) const {
        return frac(p * rhs.p, q * rhs.q);
    }
    frac operator / (const frac &rhs) const {
        return frac(p * rhs.q, q * rhs.p);
    }

    auto operator <=> (const frac &rhs) const { // 比较大小时，必须保证 q>0
        return i128(p) * rhs.q <=> i128(rhs.p) * q;
    }
    auto operator == (const frac &rhs) const {
        return i128(p) * rhs.q == i128(rhs.p) * q;
    }

    friend std::ostream &operator << (std::ostream &os, frac x) {
        i64 g = std::gcd(x.p, x.q);
        return os << x.p / g << '/' << x.q / g;
    }
};
```

0x12 基础 算法

主定理

主定理：用于求解分治递归类型的渐近时间复杂度问题。

具体地，对于一个规模为 n 的问题，将其划分成 a 个规模为 $\frac{n}{b}$ 的子问题，附带“合并/分割”的开销 $f(n)$。

$$T(n)=aT\left(\frac{n}{b}\right)+f(n)$$

以 $n^{\log_b a}$ 为处理问题用时的基准线。

- 若 $f(n)=O(n^{\log_b a-\epsilon})$
  直觉：总工作量由“叶子层”主导。
  $$T(n)=\Theta(n^{\log_b a})$$
- 若 $f(n)=\Theta(n^{\log_b a}\log^k n)$ 且满足 $k\geq 0$
  直觉：每一层工作量相近，共 $\log_b n$ 层。
  $$T(n)=\Theta(n^{\log_b a}\log^{k+1} n)$$
- 若 $f(n)=\Omega(n^{\log_b a+\epsilon})$ 且满足正则性条件 $af(\frac{n}{b})\leq cf(n)$（其中 $c<1$）
  直觉：总工作量由顶层的合并/分割主导。
  $$T(n)=\Theta(f(n))$$

> 一些例子：
>
> - 情形 1：遍历满二叉树 $T(n)=2T(\frac{n}{2})+O(1)=O(n)$。
> - 情形 2：归并排序 $T(n)=2T(\frac{n}{2})+O(n)=\mathcal{O}(n\log n)$。
> - 情形 3：多项式牛顿迭代 $T(n)=T(\frac{n}{2})+\mathcal{O}(n\log n)=\mathcal{O}(n\log n)$。
>
> 一个主定理处理不了的例子：$T(n)=2T\left(\frac{n}{2}\right)+\mathcal{O}\left(\frac{n}{\log n}\right)=\mathcal{O}(n\log\log n)$。

快速乘

求解 $a\times b \bmod p$，其中 a,b,p 均为 `long long` 范围。

正道正确率更高，魔道效率更高。

正道：i128

0x12 快速乘（i128）.cpp

```cpp
constexpr i64 qmul(i64 a, i64 b, i64 p) {
    return i128(a) * b % p;
}
```

魔道：long double

注意到 $a\times b \bmod p=a\times b-\lfloor\frac{a\times b}{p}\rfloor\times p$，利用 `long double` 来处理 $\lfloor\frac{a\times b}{p}\rfloor$。

模数较大时可能会出锅。

0x12 快速乘（long double）.cpp

```cpp
constexpr i64 qmul(i64 a, i64 b, i64 p) {
    i64 c = static_cast<long double>(a) * b / p + 1e-8;
    i64 ans = a * b - c * p;
```

```
    if (ans < 0) ans += p;
    if (ans >= p) ans -= p;
    return ans;
}
```

光速幂

对于一个固定底数 a，多次询问 $a^i \bmod p$ 的值（其中 $0 \le i \le n$）。
取阈值 $B = \lceil \sqrt{n} \rceil$，先预处理 $a^1, a^2, \dots, a^i, \dots, a^B$，再预处理 $a^B, a^{2B}, \dots, a^{iB}, \dots, a^{\lfloor n/B \rfloor B}$。
每次只需利用 $a^i = a^{\lfloor i/B \rfloor B} \times a^{i \bmod B}$ 回答即可。
时间复杂度：预处理 $\mathcal{O}(\sqrt{n})$，查询 $\mathcal{O}(1)$。

> k 进制快速幂：利用 $a^i = (a^k)^{\lfloor i/k \rfloor} \times a^{i \bmod k}$ 递归处理。右侧预处理，左侧递归。
> 时间复杂度：预处理 $\mathcal{O}(k \log_k n)$，查询 $\mathcal{O}(\log_k n)$。

0x12 光速幂.cpp

```cpp
// 光速幂
struct flashPower {
    int b;
    std::vector<int> w1, w2;

    flashPower() {}
    flashPower(int a, int n) {
        init(a, n);
    }

    void init(int a, int n) {
        b = sqrt(n) + 1;
        w1.resize(b + 1), w2.resize(b + 1);

        w1[0] = 1;
        for (int i = 1; i <= b; i ++) {
            w1[i] = 1ll * w1[i - 1] * a % mod;
        }

        w2[0] = 1;
        for (int i = 1; i <= b; i ++) {
            w2[i] = 1ll * w2[i - 1] * w1[b] % mod;
        }
    }

    int pow(int n) {
        return 1ll * w2[n / b] * w1[n % b] % mod;
    }
} fp;
```

0x21 字符串 I

哈希

字符串哈希

字符串哈希：字符串哈希要求数据有序。将字符串的元素通过某种不具有交换律的运算结合（例如字符串拼接，矩阵乘法...）。
通常将字符串 s 视为一个 base 进制数，即

$$f(s) \equiv \sum_{i=1}^{l} s_i \times \text{base}^{l-i} \pmod M$$

其中模数 M 尽量为大质数，并且尽量对两个大质数模数 M_1, M_2 各求一次哈希（双哈希）

静态字符串区间哈希：

$$\text{pre}_i = \text{pre}_{i-1} \times \text{base} + s_i$$
$$f(s[l:r]) = \text{pre}_r - \text{pre}_{l-1} \times \text{base}^{r-l+1}$$

动态字符串区间哈希：区间哈希值可以合并。设左右两个区间的长度与哈希值分别为 $(l_1, h_1), (l_2, h_2)$，将它们拼接形成大区间的长度与哈希值为

$$(l_1 + l_2, h_1 \times \text{base}^{l_2} + h_2)$$

可以使用线段树动态维护。

卡自然溢出哈希（Hash Killer I）：
base 为偶数：此时 $\text{base}^{64} \equiv 0 \pmod{2^{64}}$。所以两个长度 > 64 且后 64 位相同但不完全相同的字符串就会哈希冲突。
base 为奇数：使用 Thue Morse 序列攻击。对于二进制串 s，设 $\bar{s}$ 表示 s 按位取反后得到的串。

Thue Morse 序列：$t_0 = \texttt{0}$，$t_i = t_{i-1} + \overline{t_{i-1}}$。此时

$$f(t_i) = \text{base}^{2^{i-1}} f(t_{i-1}) + f(\overline{t_{i-1}})$$
$$f(\overline{t_i}) = \text{base}^{2^{i-1}} f(\overline{t_{i-1}}) + f(t_{i-1})$$

记 $h_i = f(t_i) - f(\overline{t_i})$，有 $h_i = \left(\text{base}^{2^{i-1}} - 1\right) h_{i-1}$。由于

$$\text{base}^{2^{i-1}} - 1 = (\text{base}^{2^{i-2}} + 1)(\text{base}^{2^{i-3}} + 1) \dots (\text{base} + 1)(\text{base} - 1)$$

其中每一项都是偶数，所以 $2^i \mid \text{base}^{2^{i-1}} - 1$。于是 $2^{\frac{i(i+1)}{2}} \mid h_i$，因此 t_{11} 与 $\overline{t_{11}}$ 就会哈希冲突。

> Thue Morse 序列的规律：第 x $(x \geq 0)$ 位上的值等于 $\text{popcount}(x) \bmod 2$。

卡大质数哈希（Hash Killer II）：使用生日悖论攻击。

集合哈希

集合哈希：集合哈希要求数据无序。将集合的元素通过某种映射规则转化后，再通过某种具有交换律的运算结合。

可用的映射规则：用来打乱原来可能有的某些性质。

- 查表映射：给原数值 x 随机赋一个新数值 c_x 并存入表中。

可用的结合方式：

- 使用加法结合：

$$f(S) = \sum_{x \in S} \text{shift}(x)$$

- 使用异或结合（异或哈希）：

$$f(S) = \bigoplus_{x \in S} \text{shift}(x)$$

此时出现偶数次元素无贡献，出现奇数次元素只贡献一次。

异或哈希判完全平方数：定义完全异或性函数 f，对于任意正整数 x, y 都有 $f(xy) = f(x) \oplus f(y)$。先对 $1 \sim n$ 进行一次线性筛。给其中的所有质数 p 对应的 $f(p)$ 赋一个范围较大的随机数，其余合数的 f 值通过线性筛得出。

此时 x 为完全平方数，极大概率等价于 $f(x) = 0$。

当待判断的数可以表示为若干个 $[1, n]$ 范围内整数的乘积时，就可以根据这些数的 f 函数异或和求出该数的 f 函数，从而快速判断其是否为完全平方数。

kmp

字符串的 border：对字符串 s 与整数 $0 \leq r < |s|$，若 s 长度为 r 的前缀与后缀相等，则称 s 长度为 r 的前缀为 s 的 border。

字符串的周期：对字符串 s 与整数 $1 \leq p \leq |s|$，若对于任意 $i \in [1, |s| - p]$ 均有 $s_i = s_{i+p}$，则称 p 为 s 的周期。特别地，若 p 整除 $|s|$，则称 p 为 s 的整周期。

相当于 $s[1 : n - p] = s[p + 1 : n]$，所以 **$s$ 有长度为 r 的 border $\iff$ $|s| - r$ 是 s 的周期。**

kmp：记 next_i 表示字符串 $s[1 : i]$ 的最长 border 的长度。

利用 border 的 border 还是原串 border 的性质，可以 $\mathcal{O}(n)$ 求解前缀函数。

进一步，$s[1 : i]$ **的所有 border 为** $\text{next}_i, \text{next}_{\text{next}_i}, \dots$

0x21 kmp.cpp

```cpp
// net[i]: s[1:i] 真前缀与真后缀匹配的最大长度（s[1:i] 的 border）
auto kmp(const auto &s) {
    int n = s.size() - 1; // s 下标从 1 开始
    std::vector<int> net(n + 1);
    net[1] = 0;
    for (int i = 2, j = 0; i <= n; i ++) {
        while (j > 0 && s[j + 1] != s[i]) j = net[j];
        if (s[j + 1] == s[i]) j ++;
        net[i] = j;
    }
    return net;
}

// f[i]: t[1:i] 后缀与 s 前缀匹配的最大长度
auto kmp_match(const auto &s, const auto &t, const auto &net) {
    int n = s.size() - 1, m = t.size() - 1; // s, t 下标从 1 开始
    std::vector<int> f(m + 1);
    for (int i = 1, j = 0; i <= m; i ++) {
        while (j > 0 && (j == n || s[j + 1] != t[i])) j = net[j];
        if (s[j + 1] == t[i]) j ++;
        f[i] = j;
    }
    return f;
}
```

border 理论

弱周期引理：若 p, q 是 s 的周期且 $p + q \leq n$，则 $\gcd(p, q)$ 也是 s 的周期。

证明：不妨设 $p > q$，下面证明 $p - q$ 也是 s 的周期。

- 当 $i \le q$ 时，$s_i = s_{i+p} = s_{i+p-q}$。
- 当 $i > q$ 时，$s_i = s_{i-q} = s_{i+p-q}$。

所以 $p - q$ 也是 s 的周期，运用辗转相除法可知 $\gcd(p, q)$ 也是 s 的周期。

> 周期引理：若 p, q 是 s 的周期且 $p + q - \gcd(p, q) \le n$，则 $\gcd(p, q)$ 也是 s 的周期。

长 border 结构：s 的所有长度 $\ge \lfloor \frac{n}{2} \rfloor$ 的 border，它们与 n 共同构成一个等差数列，公差为 s 的最小周期。

证明：设 $n-p, n-q$ $(p, q \le \lceil \frac{n}{2} \rceil)$ 是两个不同的 border，**其中 $n-p$ 是最大的 border**。已知 p, q 是 s 的周期，由于 $p + q \le n$，可知 $\gcd(p, q)$ 也是 s 的周期。由于 $n - p$ 是最大 border，所以 p 是最小周期。结合两个条件可得 $\gcd(p, q) = p$，即 $p \mid q$。同时，周期的整数倍也一定是周期。

综上，s 的所有长度 $\ge \lfloor \frac{n}{2} \rfloor$ 的 border 与 n 共同构成一个**公差为 p（最小周期）的等差数列**。

> 短周期结构：s 的所有长度 $\le \lceil \frac{n}{2} \rceil$ 的周期，都是 s 最小周期的倍数。
> 证明同「长 border 结构」。

字符串 border 结构：s 的所有 border 可以被划分成至多 $\lceil \log_2 n \rceil$ 个等差数列。

进一步，每个等差数列都具有一定的周期性。同一个等差数列里的字符串，都可以被表示成

$$W, W + A, W + A^2, \dots, W + A^k$$

或

$$W, B + W, B^2 + W, \dots, B^k + W$$

其中 W, A, B 均为字符串。

证明：设 $[2^i, 2^{i+1})$ 中最大的 border 为 p，那么段内的其余 border 都是 $s[1 : p]$ 的 border，且长度 $\ge \lfloor \frac{p}{2} \rfloor$，由「长 border 结构」可知段内构成一个等差数列。

推论：s 的所有公差 $\ge d$ 的 border 的等差数列总大小是 $\mathcal{O}(\frac{n}{d})$ 的。

Misc：

- 若 s 在 t 中的某两个匹配 s_1, s_2 有重合，且起始位置相差 d，则 d 均为 s 与 $s_1 \cup s_2$（两个匹配区间并）的周期。
- 若 $|s| \ge \frac{|t|}{2}$，则 s 在 t 中的匹配起始位置形成等差数列。若该等差数列至少有 3 项，则公差为 s 的最小周期 $\mathrm{per}(s)$。

Z Algorithm

Z Algorithm：记 Z_i 表示字符串 s 与 $s[i : n]$ 的最长公共前缀（LCP）长度。在某些情况下，可能会记 $Z_1 = 0$。

考虑从前向后扫。维护一个右端点最靠右，且与 s 的前缀匹配的子串 $s[l : r]$。

- 若 $i \le r$，则
$$Z_i \ge \min(Z_{i-l+1}, r - i + 1)$$
- 若 $i > r$，则
$$Z_i \ge 0$$

在当前 Z_i 的基础上不断尝试扩展。

注意到单次枚举的次数为 $\max\{l + Z_l - 1\}$ 的增量，时间复杂度 $\mathcal{O}(n)$。

0x21 Z-Algorithm.cpp

```cpp
// Z[i]: lcp(s, s[i:n])
auto Zalgo(const auto &s) {
    int n = s.size() - 1; // s 下标从 1 开始
    std::vector<int> Z(n + 1);
    Z[1] = n;
    for (int i = 2, l = 0, r = 0; i <= n; i ++) {
        int &x = Z[i] = i <= r ? std::min(Z[i - l + 1], r - i + 1) : 0;
        while (i + x <= n && s[1 + x] == s[i + x]) {
            x ++;
        }
        if (i + x - 1 > r) {
            l = i, r = i + x - 1;
        }
    }
    return Z;
}

// p[i]: lcp(s, t[i:m])
auto Zalgo_match(const auto &s, const auto &t, const auto &Z) {
    int n = s.size() - 1, m = t.size() - 1; // s, t 下标从 1 开始
    std::vector<int> p(m + 1);
    for (int i = 1, l = 0, r = 0; i <= m; i ++) {
        int &x = p[i] = i <= r ? std::min(Z[i - l + 1], r - i + 1) : 0;
        while (1 + x <= n && i + x <= m && s[1 + x] == t[i + x]) {
```

```
            x ++;
        }
        if (i + x - 1 > r) {
            l = i, r = i + x - 1;
        }
    }
    return p;
}
```

manacher

奇偶回文串的处理：回文串按照长度分成奇偶两类，奇回文串的对称轴处于一个字符上，偶回文串的对称轴处于两个相邻字符的中间。

可以在一个字符串的首尾以及所有的相邻两字符之间，插入一个原串中从未出现的字符（例如 #, $, ^）。

这样原串中的每一个回文串，都可以对应到新串中的一个两端都是分隔符的奇回文串。

manacher：记 p_i 表示以位置 i 为中心的最长回文子串的半径（含中心）。

考虑从前向后扫。维护一个右端点最靠右的回文子串，不妨记该回文串的中心为 k。

- 若 $i \le k + p_k - 1$，则

$$p_i \ge \min(k + p_k - i, p_{2k-i})$$

- 若 $i > k + p_k - 1$，则

$$p_i \ge 0$$

在当前 p_i 的基础上不断尝试扩展。

注意到单次枚举的次数为 $\max\{k + p_k - 1\}$ 的增量，时间复杂度 $\mathcal{O}(n)$。

0x21 manacher.cpp

```cpp
// 转化为形如 #a#b#c# 的字符串
auto manacher_init(const std::string &s) {
    std::string t = "#";
    for (char ch : s) {
        t += ch, t += "#";
    }
    return t;
}

// p[i]: 区间 [i - p[i] + 1, i + p[i] - 1] 回文
auto manacher(const auto &s) {
    int n = s.size() - 1; // s 下标从 1 开始
    std::vector<int> p(n + 1);
    for (int i = 1, mid = 0, r = 1; i <= n; i ++) {
        int &x = p[i] = i < r ? std::min(r - i, p[mid - i]) : 0;
        while (i - x > 0 && i + x <= n && s[i - x] == s[i + x]) {
            x ++;
        }
        if (i + x > r) {
            mid = i * 2, r = i + x;
        }
    }
    return p;
}

// 以位置 i 为回文中心的最长回文子串，在原串中的长度为 p[i] - 1（转换成 #a#b#c# 形式后）
// 判断原串区间 [l, r] 是否回文：p[l + r] - 1 >= r - l + 1（转换成 #a#b#c# 形式后）
```

0x21 manacher（奇偶分开考虑）.cpp

```cpp
// p[i]: 区间 [i - p[i] + 1, i + p[i] - !is_even] 回文，奇偶分开考虑
auto manacher(const auto &s, int is_even) {
    int n = s.size() - 1; // s 下标从 1 开始
    std::vector<int> p(n + 1);
    for (int i = 1, mid = 0, r = 1; i <= n - is_even; i ++) {
        int &x = p[i] = i < r ? std::min(r - i, p[mid - i]) : 0;
        while (i - x > 0 && i + x + is_even <= n && s[i - x] == s[i + x + is_even]) {
            x ++;
        }
        if (i + x > r) {
            mid = i * 2, r = i + x;
        }
    }
    return p;
}
```

反回文串：对于长度为 n 的字符串 s，若对于任意 $i \in [1, n]$ 均有 $s_i \ne s_{n-i+1}$，则称 s 为反回文串。

01 串的反 manacher：相当于对于每两个相邻字符的中间，求以该位置为中心的最长反回文子串的半径。注意我们只需要考虑长度为偶数的串。

当字符集大小为 2 时，考虑形如 ABCD 的字符串，若已知 $A \neq B, A \neq D, B \neq C$，可以推出 $C \neq D$，所以 manacher 的对称复用逻辑还是正确的。但是字符集大小超过 2 的时候，上述推理就不成立了。

具体实现时，只需要将 manacher 偶回文串指针扩展过程中的 == 改成 != 即可。

Lyndon 分解

Lyndon 串：对于字符串 s，若 s 的字典序小于 s 的所有非空真后缀，则称 s 为 Lyndon 串。

等价定义：

- 字符串 s 的字典序严格小于 s 的其他循环移位。
- 对字符串 s 的任意非空分解 $s = uv$，都有 $u < v$。
- 对字符串 s 的任意非空分解 $s = uv$，都有 $uv < vu$。

Lyndon 串的拼接性质：

- **若字符串 u, v 均为 Lyndon 串并且 $u < v$，则 uv 为 Lyndon 串，且 $u < uv < v$。**
- 若字符串 u, v 均为 Lyndon 串并且 $u < v$，则对于任意正整数 a, b，均有 $u^a v^b$ 为 Lyndon 串，且 $u < u^a v^b < v$。
- 字符串 s ($|s| \geq 2$) 是 Lyndon 串，当且仅当 $s = uv$ ($|u| > 0, |v| > 0, \boldsymbol{u < v}$) 并且 u, v 均为 Lyndon 串。

Lyndon 串的简单性质：

- Lyndon 串不存在非空 border。
- 截取 Lyndon 串的某个前缀，将其末字符增大，结果仍是 Lyndon 串（截出的前缀本身不必是 Lyndon 串）。
- 若字符串 s_1, s_2 满足 s_1 是 Lyndon 串且 $s_1 > s_2$，则 $s_1 > s_2^2$。

Lyndon 分解：将字符串 s 分解为 $w_1 w_2 \dots w_k$，其中所有的 w_i 均为 Lyndon 串，且满足 $w_1 \geq w_2 \geq \cdots \geq w_k$。

Lyndon 分解的性质：

- Lyndon 分解存在且唯一。
- 字符串 s 是 Lyndon 串当且仅当 $k = 1$。
- 第一块 w_1：是原串最长的 Lyndon 前缀。
- 最后一块 w_k：是原串最长的 Lyndon 后缀，也是原串字典序最小的非空后缀。

Lyndon 分解（duval）：维护三个指针 i, j, k，将字符串分成三个部分：

- $s[1 : i-1]$：这部分的 Lyndon 分解已完成。
- $s[i : j-1]$：这部分可以被表示为 $w^c + v$，其中 w 是一个 Lyndon 串，v 是 w 的可空真前缀。
- $s[j : n]$：这部分还未处理，当前考虑的是位置 j。

其中 $k = j - |w|$，那么 s_k 表示继续循环下去时，位置 j 应当出现的字符。

接下来有三种情况：

- $s_k = s_j$：可以将 w 继续循环下去，令 $k \leftarrow k+1$，$j \leftarrow j+1$。
- $s_k < s_j$：此时 $w^c + v + s_j$ 本身是一个 Lyndon 串，将其作为新的 w，令 $k \leftarrow i$，$j \leftarrow j+1$。
- $s_k > s_j$：将当前的 c 个 w 依次加入 Lyndon 分解，然后从 v 的开头重新扫描。

每一轮 j 扫描的长度，不超过 i 前进长度的两倍。时间复杂度 $\mathcal{O}(n)$。

0x21 duval.cpp

```cpp
// Lyndon 分解
auto duval(const auto &s) {
    int n = s.size() - 1; // s 下标从 1 开始
    std::vector<int> res;
    for (int i = 1; i <= n;) {
        int j = i + 1, k = i;
        for (; j <= n && s[k] <= s[j]; j ++) {
            if (s[k] < s[j]) {
                k = i;
            } else {
                k ++;
            }
        }
        for (; i <= k; i += j - k) {
            res.push_back(i);
        }
    }
    return res;
}
```

标准分解：对于 Lyndon 串 s ($|s| \geq 2$)，记 s 的最长 Lyndon 真后缀为 v，此时 $s = uv$，记 (u, v) 为 s 的标准分解。

最小表示法

循环同构：若长度为 n 的字符串 s，可以选择一个起始位置 i $(1 \le i \le n)$，使得 $s[i:n]+s[1:i-1]=t$，则称 s,t 循环同构。

最小表示法：字符串 s 的最小表示法，为与 s 循环同构的所有字符串中，字典序最小的字符串。

对于两个不同的起始位置 i,j，已知它们的前 k 位相同。考虑继续比较下一位：

- 若 $s_{(i+k) \bmod n} > s_{(j+k) \bmod n}$，可以发现区间 $[i,i+k]$ 内的起始位置都不能成为答案，因为对于起始位置 $i+p$ $(p \in [0,k])$ 来说，起始位置 $j+p$ 一定更优。所以此时可以跳过区间 $[i,i+k]$ 的起始位置，令 $i \leftarrow i+k+1$，$k \leftarrow 0$。
- 若 $s_{(i+k) \bmod n} < s_{(j+k) \bmod n}$ 也是同理，跳过区间 $[j,j+k]$ 的起始位置，令 $j \leftarrow j+k+1$，$k \leftarrow 0$。
- 若 $s_{(i+k) \bmod n} = s_{(j+k) \bmod n}$，令 $k \leftarrow k+1$，然后继续比较。

在跳转的过程中，若两个起始位置 i,j 相同，则任选一个加一。最终 $\min(i,j)$ 即为最优的起始位置。

时间复杂度 $\mathcal{O}(n)$。

0x21 最小表示法.cpp

```cpp
// 最小表示法
// std::rotate(s.begin(), s.begin() + minrep(s), s.end());
int minrep(const auto &s) {
    int n = s.size(), k = 0, i = 0, j = 1; // s 下标从 0 开始
    while (k < n && i < n && j < n) {
        int d = s[(i + k) % n] - s[(j + k) % n];
        if (d == 0) {
            k ++;
        } else {
            (d > 0 ? i : j) += k + 1;
            if (i == j) i ++;
            k = 0;
        }
    }
    return std::min(i, j);
}
```

带通配符的字符串匹配

0x22 字符串 II

Trie

trie：trie 是一个纯正的自动机，形态是树。

- 一个起点，若干终点。
- 多模式串的所有前缀，与 trie 上的所有状态一一对应。
- trie 主要考虑一种转移边：表示在当前状态所表示的串后面加一个字符。

01 trie：

- 01 trie 支持可持久化，可以作为 **01 trie 的前缀和**来使用。
- 01 trie 支持合并：
  - 01 trie 单次合并复杂度，取决于两棵 01 trie 的交集大小。
  - 01 trie 合并总复杂度，取决于所有 01 trie 的节点总数。
- 01 trie 支持可持久化合并，需要在合并时新建节点。

0x22 01 trie.cpp

```cpp
// 01 trie
namespace trie {
    const int pond = 10001000; // 内存池大小，需要根据实际问题调整

    int nodeCount;
    struct node {
        int trans[2];
        int cnt;
        #define lc trans[0]
        #define rc trans[1]
    } t[pond];

    int newnode() {
        int p = ++ nodeCount;
        t[p].lc = t[p].rc = t[p].cnt = 0;
        return p;
    }

    void insert(int &p, int d, int x) {
        if (!p) p = newnode();
        t[p].cnt ++;
```

```
        if (d < 0) return;
        int v = x >> d & 1;
        insert(t[p].trans[v], d - 1, x);
    }

    int merge(int p, int q) {
        if (!p || !q) return p ^ q;
        t[p].cnt += t[q].cnt;
        t[p].lc = merge(t[p].lc, t[q].lc);
        t[p].rc = merge(t[p].rc, t[q].rc);
        return p;
    }

    // 可持久化 01 trie 合并
    // int merge(int p, int q) {
    //     if (!p || !q) return p ^ q;
    //     int u = newnode();
    //     t[u].cnt = t[p].cnt + t[q].cnt;
    //     t[u].lc = merge(t[p].lc, t[q].lc);
    //     t[u].rc = merge(t[p].rc, t[q].rc);
    //     return u;
    // }
}
```

0x22 可持久化 01 trie.cpp

```
// 可持久化 01 trie
namespace trie {
    const int pond = 10001000; // 内存池大小，需要根据实际问题调整

    int nodeCount;
    struct node {
        int trans[2];
        int cnt;
    } t[pond];

    void insert(int &p, int q, int d, int x) {
        p = ++ nodeCount, t[p] = t[q], t[p].cnt ++;
        if (d < 0) return;
        int v = x >> d & 1;
        insert(t[p].trans[v], t[q].trans[v], d - 1, x);
    }
}
```

01 trie 查询与 x 异或的最大值：从高位到低位贪心，有异走异，无异走同。

01 trie 查询与 x 异或的第 k 大值：从高位到低位

- 设当前考虑到第 i 位，设 x 的异位有 cnt 个数。
  - 若 $k \le \text{cnt}$，则查询 x 的异位中，与 x 异或的第 k 大值。
  - 若 $k > \text{cnt}$，则查询 x 的同位中，与 x 异或的第 $k - \text{cnt}$ 大值。

01 trie 查询与 x 异或 $\ge k$ 的信息：从高位到低位

- 设当前考虑到第 i 位：
  - 若 k 的第 i 位为 1，直接走 x 的异位。
  - 若 k 的第 i 位为 0，统计 x 的异位的贡献，然后走 x 的同位。
- 考虑完所有位之后，统计当前叶子节点的贡献。

01 trie 全局异或 x：从高位到低位，运用懒标记。

设当前考虑到第 i 位，若懒标记的第 i 位为 1，则交换左右儿子。

01 trie 维护集合异或和，支持插入、删除、全局 +1：从低位到高位

- 额外维护：
  - cnt：当前状态的子树内插入数的个数。
  - xr：当前状态的子树内的异或和。
- 插入、删除：递归操作即可。
- 全局 +1：交换当前节点的左右儿子，然后进入 0 转移边（未交换前是 1 转移边）的儿子进行递归交换。

（因为二进制下 +1，相当于是从低位到高位找到第一个 0 将其改为 1，然后将前面的 1 都改为 0）

0x22 01 trie 维护集合异或和.cpp

```
// 01 trie 维护集合异或和
namespace trie {
    const int pond = 10001000; // 内存池大小，需要根据实际问题调整

    int nodeCount;
    struct node {
        int trans[2];
        int cnt;
        int xr;
        #define lc trans[0]
        #define rc trans[1]
    } t[pond];
```

```
    int newnode() {
        int p = ++ nodeCount;
        t[p].lc = t[p].rc = t[p].cnt = t[p].xr = 0;
        return p;
    }

    void upd(int p) {
        t[p].cnt = t[t[p].lc].cnt + t[t[p].rc].cnt;
        t[p].xr = (t[t[p].lc].xr << 1) ^ (t[t[p].rc].xr << 1) ^
         ↪  (t[t[p].rc].cnt & 1);
    }

    void insert(int &p, int d, int x, int y) {
        if (!p) p = newnode();
        if (d == 31) {
            t[p].cnt += y;
            return;
        }
        int v = x >> d & 1;
        insert(t[p].trans[v], d + 1, x, y);
        upd(p);
    }

    void addone(int p) {
        std::swap(t[p].lc, t[p].rc);
        if (t[p].lc) addone(t[p].lc);
        upd(p);
    }
}
```

AC 自动机

AC 自动机：AC 自动机是一个状态自动机，形态是 trie。

- 一个起点，若干终点。
- 多模式串的所有前缀，与 AC 自动机上的所有状态一一对应。
- AC 自动机主要考虑两种边：
  - 普通转移边：表示在当前状态所表示的串后面加一个字符。
  - 失配指针 fail：表示当前状态所表示的串的最长匹配后缀。这类边构成一棵树，称作 fail 树。

trie 图：将原来 trie 中不存在的转移边 $\delta(p,c)$ 指向 $\delta(\text{fail}_p,c)$。本质上是路径压缩。

trie 图的普通转移边：从字符串 s 指向 $s'+c$，其中 s' 为 s 中满足 $s'+c$ 存在的最长后缀。

0x22 AC 自动机.cpp

```
// AC 自动机
namespace AC {
    const int SIZE = 1000100; // 多模式串长度总和，需要根据实际问题调整

    int nodeCount;
    struct node {
        int trans[26];
        int fail;
    } t[SIZE];

    int newnode() {
        int p = ++ nodeCount;
        for (int c = 0; c < 26; c ++) t[p].trans[c] = 0;
        t[p].fail = 0;
        return p;
    }
    void init() {
        nodeCount = 0, newnode(); // 根节点为 1
    }

    void insert(const std::string &s) {
        int p = 1;
        for (char ch : s) {
            int v = ch - 'a';
            if (!t[p].trans[v]) t[p].trans[v] = newnode();
            p = t[p].trans[v];
        }
    }

    void build_fail() {
        for (int i = 0; i < 26; i ++) t[0].trans[i] = 1;
        t[1].fail = 0;

        std::queue<int> q;
        q.push(1);

        while (q.size()) {
            int u = q.front(); q.pop();
            for (int i = 0; i < 26; i ++) {
                if (t[u].trans[i]) {
                    t[t[u].trans[i]].fail = t[t[u].fail].trans[i];
```

```
                    q.push(t[u].trans[i]);
                } else {
                    t[u].trans[i] = t[t[u].fail].trans[i];
                }
            }
        }
    }
}
```

AC 自动机（大字符集）：当字符集很大时，需要用到 `std::map` 与可持久化数组建立 trie 图。

0x22 AC 自动机（大字符集）.cpp

```cpp
// 可持久化数组
namespace SGT {
    const int pond = 10001000; // 内存池大小，需要根据实际问题调整

    int nodeCount;
    struct node {
        int lc, rc;
        int val;
    } t[pond];

    void init() {
        nodeCount = 0, t[0].val = 1;
    }

    void insert(int &p, int q, int l, int r, int x, int y) {
        p = ++ nodeCount, t[p] = t[q];
        if (l == r) { t[p].val = y; return; }
        int mid = (l + r) >> 1;
        if (x <= mid) {
            insert(t[p].lc, t[q].lc, l, mid, x, y);
        } else {
            insert(t[p].rc, t[q].rc, mid + 1, r, x, y);
        }
    }

    int ask(int p, int l, int r, int x) {
        if (l == r) return t[p].val;
        int mid = (l + r) >> 1;
        if (x <= mid) {
            return ask(t[p].lc, l, mid, x);
        } else {
            return ask(t[p].rc, mid + 1, r, x);
        }
    }
}

// AC 自动机（大字符集）
namespace AC {
    const int SIZE = 1000100; // 多模式串长度总和，需要根据实际问题调整

    int nodeCount;
    struct node {
        std::map<int, int> trans;
        int fail;
    } t[SIZE];

    int root[SIZE];

    int newnode() {
        int p = ++ nodeCount;
        t[p].trans.clear();
        t[p].fail = 0;
        return p;
    }
    void init() {
        nodeCount = 0, newnode(); // 根节点为 1
    }

    void insert(const std::vector<int> &a) {
        int p = 1;
        for (int v : a) {
            if (!t[p].trans.contains(v)) t[p].trans[v] = newnode();
            p = t[p].trans[v];
        }
    }

    void build_fail() {
        SGT::init(), root[0] = 0;
        t[1].fail = 0;

        std::queue<int> q;
        q.push(1);

        while (q.size()) {
            int u = q.front(); q.pop();
            root[u] = root[t[u].fail];
            for (auto [c, v] : t[u].trans) {
```

```
                SGT::insert(root[u], root[u], 1, V, c, v);
                t[v].fail = SGT::ask(root[t[u].fail], 1, V, c);
                q.push(v);
            }
        }
    }
}
```

AC 自动机查询所有模式串在文本串中的出现位置：

将所有询问串 t 去重之后，所有询问串 t 在文本串 s 中的出现次数之和不超过 $\mathcal{O}(|s|\sqrt{\sum|t|})$ 级别。

因为询问串的串长 $|t|$ 种类数不超过 $\mathcal{O}(\sqrt{\sum|t|})$ 级别，同一长度的询问串 endpos 各不相同。

对询问串建 AC 自动机（trie 图），每个节点记录一下向上跳 fail 指针遇到的最近询问节点 up。

然后从左到右扫描 s，扫描到右端点 r 时，维护以 r 结尾的最长匹配串对应的状态。然后在当前状态不断地跳 up 指针，去更新（记录）当前询问串在 s 中的 endpos。

SA

序列 SA 的求法：

- $\mathcal{O}(n\log^2 n)$：sort + 倍增 + hash。当比较函数时间开销较大时，尝试使用 `std::stable_sort()`。
- $\mathcal{O}(n\log n)$：倍增 + 基数排序（双关键字排序）。
- $\mathcal{O}(n)$：DC3，SA-IS。

height 数组：$\text{height}_i = \text{LCP}(\text{sa}_{i-1}, \text{sa}_i)$

height 数组求后缀 LCP：后缀 sa_i, sa_j $(i<j)$ 的最长公共前缀为

$$\text{LCP}(\text{sa}_i, \text{sa}_j) = \min_{i<k\le j}\{\text{height}_k\}$$

本质不同子串个数：

$$\frac{n(n+1)}{2} - \sum_{i=2}^{n}\text{height}_i$$

（序列 SA）height 数组的性质：$\text{height}_{\text{rk}_i} \ge \text{height}_{\text{rk}_{i-1}} - 1$

证明：

- 当 $\text{height}_{\text{rk}_{i-1}} = 0$ 时，命题成立。
- 当 $\text{height}_{\text{rk}_{i-1}} \ge 1$ 时，考虑到后缀排序将“前缀相近的后缀”放到一起，所以 height_i 可以视为字典序小于 sa_i 的后缀与 sa_i 的 LCP 最大值。记 $u = \text{sa}_{\text{rk}_{i-1}-1}$，则有 $s_u = s_{i-1}$，于是 $s[u+1:n]$ 的字典序比 $s[i:n]$ 更小，则必然存在一个字典序小于 $s[i:n]$ 的后缀使得与其的 LCP 大于等于 $\text{height}_{\text{rk}_{i-1}} - 1$，故命题成立。

0x22 SA.cpp

```cpp
// 后缀数组
struct SuffixArray {
    int n, t;
    std::vector<int> sa, rk, height;
    std::vector<std::vector<int>> f;

    SuffixArray() {}
    SuffixArray(const auto &s) {
        build(s);
    }

    void build(const auto &s) {
        n = s.size() - 1; // s 下标从 1 开始
        int m = 256;
        sa.resize(n + 1), rk.resize(n + 1), height.resize(n + 1);
        std::vector<int> cnt(std::max(n, m) + 1), id(n + 1), px(n + 1),
         ↪ tmp(2 * n + 1);

        auto same = [&] (int x, int y, int k) {
            return tmp[x] == tmp[y] && tmp[x + k] == tmp[y + k];
        };

        for (int i = 1; i <= n; i ++) cnt[rk[i] = s[i]] ++;
        for (int i = 1; i <= m; i ++) cnt[i] += cnt[i - 1];
        for (int i = n; i >= 1; i --) sa[cnt[rk[i]] --] = i;

        for (int k = 1, p = 0; k < n; k <<= 1, m = p) {
            p = 0;
            for (int i = n - k + 1; i <= n; i ++) id[++ p] = i;
            for (int i = 1; i <= n; i ++)
                if (sa[i] > k) id[++ p] = sa[i] - k;

            for (int i = 1; i <= m; i ++) cnt[i] = 0;
            for (int i = 1; i <= n; i ++) cnt[px[i] = rk[id[i]]] ++;
            for (int i = 1; i <= m; i ++) cnt[i] += cnt[i - 1];
            for (int i = n; i >= 1; i --) sa[cnt[px[i]] --] = id[i];
```

```
            p = 0;
            for (int i = 1; i <= n; i ++) tmp[i] = rk[i];
            for (int i = 1; i <= n; i ++) rk[sa[i]] = same(sa[i - 1], sa[i],
             ↪ k) ? p : ++ p;
            if (p == n) break;
        }
        if (n == 1) rk[1] = 1;

        for (int i = 1, h = 0; i <= n; i ++) {
            int j = sa[rk[i] - 1];
            if (h) h --;
            while (i + h <= n && j + h <= n && s[i + h] == s[j + h]) h ++;
            height[rk[i]] = h;
        }

        t = std::__lg(n);
        f.assign(t + 1, std::vector<int>(n + 1));
        for (int i = 1; i <= n; i ++) {
            f[0][i] = height[i];
        }
        for (int j = 1; j <= t; j ++) {
            for (int i = 1; i <= n - (1 << j) + 1; i ++) {
               f[j][i] = std::min(f[j - 1][i], f[j - 1][i + (1 << (j - 1))]);
            }
        }
    }

    int ask(int l, int r) {
        int k = std::__lg(r - l + 1);
        return std::min(f[k][l], f[k][r - (1 << k) + 1]);
    }
    int lcp(int i, int j) {
        if (i == j) return n - i + 1;
        auto [l, r] = std::minmax(rk[i], rk[j]);
        return ask(l + 1, r);
    }
};
```

SAM

SAM 与其他后缀数据结构的关系：

- 后缀 trie：一个字符串的所有后缀构成的 trie。
- 后缀 trie 上建 AC 自动机的 fail 指针：表示删除当前状态所表示的串的首字母。
- 后缀树：后缀 trie 虚树化（仅保留所有后缀在后缀 trie 上的终止节点，以及两两 LCA）。
- 后缀自动机的 parent 树：反串的后缀树。

SAM：SAM 是一个状态自动机，形态是 DAG。

- 一个起点，若干终点。
- 原串的所有本质不同的子串，与在 SAM 上从起点开始到任意点结束的所有路径一一对应，不重不漏。
- SAM 主要考虑两种边：
  - 普通转移边：表示在当前状态所表示的串后面加一个字符。
  - 后缀链接 link：表示将当前状态所表示的最短串的首字母删除。这类边构成一棵树，称作 parent 树。
- SAM 点数 $\leq 2n-1$，边数 $\leq 3n-4$。

endpos 集合：子串 s 的所有终止位置的集合，记作 $\text{endpos}(s)$。

- SAM 中的每一个状态，都对应一个 endpos 等价类。
- SAM 中的每一个状态（endpos 等价类），包含的子串长度构成一个连续段，且短串是长串的后缀。
- 对于字符串 s_1, s_2 ($|s_1| \leq |s_2|$)：
  - 若 s_1 为 s_2 的后缀，则 $\text{endpos}(s_1) \supseteq \text{endpos}(s_2)$。
  - 否则 $\text{endpos}(s_1) \bigcap \text{endpos}(s_2) = \varnothing$。

endpos 集合的维护：在 parent 树上使用（可持久化）线段树合并维护。

在 parent 树上，节点 p 的 endpos 集合为 $\text{subtree}(p)$ 中所有节点 endpos 集合的并集。为此，我们在 $s[1:i]$ 对应状态的 **endpos 集合里插入位置 i，然后使用数据结构合并子树的 endpos 集合**。

当字符集很大时，需要用到 `std::map` 建立 SAM。

0x22 SAM.cpp

```
// 后缀自动机
namespace SAM {
    const int SIZE = N * 2;

    int nodeCount, Last;
    struct node {
        int trans[26];
```

```
        int link, maxl;
    } t[SIZE];

    int newnode() {
        int p = ++ nodeCount;
        for (int c = 0; c < 26; c ++) t[p].trans[c] = 0;
        t[p].link = t[p].maxl = 0;
        return p;
    }
    void init() {
        nodeCount = 0, Last = 1, newnode(); // 根节点为 1
    }

    void extend(int c) {
        int p = Last, np = Last = newnode();

        t[np].maxl = t[p].maxl + 1;
        for (; p && t[p].trans[c] == 0; p = t[p].link) {
            t[p].trans[c] = np;
        }

        if (!p) {
            t[np].link = 1;
        } else {
            int q = t[p].trans[c];
            if (t[q].maxl == t[p].maxl + 1) {
                t[np].link = q;
            } else {
                int nq = newnode();
                t[nq] = t[q], t[nq].maxl = t[p].maxl + 1;
                t[np].link = t[q].link = nq;
                for (; p && t[p].trans[c] == q; p = t[p].link) {
                    t[p].trans[c] = nq;
                }
            }
        }
    }
}
```

SAM 子串定位：求 $s[l:r]$ 在 SAM 中的对应状态

先定位 $s[1:r]$ 在 SAM 中的对应状态（在构造 SAM 的过程中记录一下），然后在 **parent 树上倍增**，找到深度最浅的满足 $\text{maxl} \geq r-l+1$ 的状态。

SAM 子串匹配：给定模式串 S 与文本串 T，对每个右端点 r 求最左端的 l 使得 $T[l:r]$ 为 S 的子串

当 r 扩展时，不断地跳后缀链接（这里令 len 变为对应状态的 maxl），直到跳到根节点或存在相应的转移边为止。

若存在相应的转移边，则走该转移边（这里令 $\text{len} \leftarrow \text{len}+1$）。

```
int p = 1, len = 0; // 当前匹配到的状态与长度
for (int i = 1; i <= m; i ++) {
    int c = T[i] - 'a';
    while (p > 1 && !SAM::t[p].trans[c]) {
        p = SAM::t[p].link, len = SAM::t[p].maxl;
    }
    if (SAM::t[p].trans[c]) {
        p = SAM::t[p].trans[c], len ++;
    }
    // 此时的 p 与 len，就是以 i 为结尾时匹配到的最长状态与长度
}
```

子 SAM：给定模式串 S 与文本串 T，让 T 在 $S[l:r]$ 中进行匹配

判断由 $S[l:r]$ 组成的子 SAM 是否存在相应的转移边，首先在原 SAM 就要存在相应的转移边，并且原 SAM 中新状态的 endpos 集合在区间 $[l+\text{len}'-1, r]$ 中至少要有一个元素。

失配时，应先尝试 $\text{len} \leftarrow \text{len}-1$ 然后继续匹配，而非直接跳失配链接。

SAM 前端删除：在一个状态的前面删除字符，求新状态

若当前长度等于当前状态的 minl，则跳后缀链接。

SAM 前端插入：在一个状态的前面添加字符，求新状态

维护每个状态 u 对应 endpos 集合的任意一个元素 pos_u，就可以做到在原串中定位状态 u。

- 若长度等于当前状态的 maxl：则相当于在 **parent 树上向某一个儿子走**。在构建 parent 树的时候，预处理出每个状态的极长串前面加一个字符 c 后会走到哪个儿子即可（枚举 u 以及其儿子 v，根据 pos_v 在原串中定位，即可得知从 u 到 v 需要添加什么字符）。
- 若长度小于当前状态的 maxl：相当于要考虑当前状态是否能够容纳新串，根据 pos_u 在原串中定位，判断新加的字符是否与原串对应位置上的字符匹配。

0x22 SAM 前端操作.cpp

```
// 后缀自动机（前端操作）
namespace SAM {
    const int SIZE = N * 2;

```

```
    int strL, str[N];
    int nodeCount, Last;
    struct node {
        int trans[26];
        int link, maxl, pos;
    } t[SIZE];

    int net[SIZE][26];
    std::vector<int> son[SIZE];

    int newnode() {
        int p = ++ nodeCount;
        for (int c = 0; c < 26; c ++) t[p].trans[c] = net[p][c] = 0;
        t[p].link = t[p].maxl = t[p].pos = 0;
        son[p].clear();
        return p;
    }
    void init() {
        nodeCount = 0, Last = 1, newnode(); // 根节点为 1
        str[strL = 0] = -1;
    }

    void extend(int c) {
        int p = Last, np = Last = newnode();

        str[++ strL] = c, t[np].pos = strL;
        t[np].maxl = t[p].maxl + 1;
        for (; p && t[p].trans[c] == 0; p = t[p].link) {
            t[p].trans[c] = np;
        }

        if (!p) {
            t[np].link = 1;
        } else {
            int q = t[p].trans[c];
            if (t[q].maxl == t[p].maxl + 1) {
                t[np].link = q;
            } else {
                int nq = newnode();
                t[nq] = t[q], t[nq].maxl = t[p].maxl + 1;
                t[np].link = t[q].link = nq;
                for (; p && t[p].trans[c] == q; p = t[p].link) {
                    t[p].trans[c] = nq;
                }
            }
        }
    }
    void dfs_init(int u) {
        for (int v : son[u]) {
            net[u][str[t[v].pos - t[u].maxl]] = v;
            dfs_init(v);
        }
    }
    void build_tree() {
        for (int i = 2; i <= nodeCount; i ++) {
            son[t[i].link].push_back(i);
        }
        dfs_init(1);
    }

    void push_front(int &p, int &len, int c) {
        if (len == t[p].maxl) {
            p = net[p][c];
        } else if (str[t[p].pos - len] != c) {
            p = 0;
        }
        len = p ? len + 1 : 0;
    }
}
```

SAM 补节点：在 parent 树中，补充询问对应的状态

parent 树是反串的后缀树，而后缀树又是后缀 trie 虚树化（仅保留所有后缀在后缀 trie 上的终止节点，以及两两 LCA）。

由于询问串 $s[l:r]$ 在 SAM 上的状态可能不完全覆盖 minl $\sim$ maxl（为该区间的一个前缀），于是我们离线将所有询问串在 SAM 上的状态找出来，然后将状态划分，建立有关询问串的虚拟节点（补充有关询问串的虚树结构）。此时所有询问串一定完整地覆盖了从当前状态到根的路径，且总节点数不超过 $2n+m$。

简化了 parent 树的结构，降低了讨论难度。

广义 SAM：对多模式串建立 SAM

0x22 广义 SAM（离线）.cpp

```
// 广义 SAM（离线）
namespace GSAM {
    const int SIZE = N * 2;

    int trieCount, trie[N][26], site[N];
```

```
    int nodeCount;
    struct node {
        int trans[26];
        int link, maxl;
    } t[SIZE];

    int newtrie() {
        int p = ++ trieCount;
        for (int c = 0; c < 26; c ++) trie[p][c] = 0;
        site[p] = 0;
        return p;
    }
    int newnode() {
        int p = ++ nodeCount;
        for (int c = 0; c < 26; c ++) t[p].trans[c] = 0;
        t[p].link = t[p].maxl = 0;
        return p;
    }
    void init() {
        trieCount = 0, newtrie(); // trie 根节点为 1
        nodeCount = 0, newnode(); // GSAM 根节点为 1
    }

    void insert(const std::string &s) {
        int p = 1;
        for (char ch : s) {
            int v = ch - 'a';
            if (!trie[p][v]) trie[p][v] = newtrie();
            p = trie[p][v];
        }
    }

    int extend(int Last, int c) {
        int p = Last, np = newnode();

        t[np].maxl = t[p].maxl + 1;
        for (; p && t[p].trans[c] == 0; p = t[p].link) {
            t[p].trans[c] = np;
        }

        if (!p) {
            t[np].link = 1;
        } else {
            int q = t[p].trans[c];
            if (t[q].maxl == t[p].maxl + 1) {
                t[np].link = q;
            } else {
                int nq = newnode();
                t[nq] = t[q], t[nq].maxl = t[p].maxl + 1;
                t[np].link = t[q].link = nq;
                for (; p && t[p].trans[c] == q; p = t[p].link) {
                    t[p].trans[c] = nq;
                }
            }
        }
        return np;
    }

    void build() {
        std::queue<int> q;
        q.push(1), site[1] = 1;

        while (q.size()) {
            int u = q.front(); q.pop();
            for (int i = 0; i < 26; i ++) {
                if (trie[u][i]) {
                    site[trie[u][i]] = extend(site[u], i);
                    q.push(trie[u][i]);
                }
            }
        }
    }
}
```

0x22 广义 SAM（在线）.cpp

```
// 广义 SAM（在线）
// 在插入一个串之前，需要令 GSAM::Last = 1
namespace GSAM {
    const int SIZE = N * 2;

    int nodeCount, Last;
    struct node {
        int trans[26];
        int link, maxl;
    } t[SIZE];

    int newnode() {
        int p = ++ nodeCount;
        for (int c = 0; c < 26; c ++) t[p].trans[c] = 0;
        t[p].link = t[p].maxl = 0;
```

```
        return p;
    }
    void init() {
        nodeCount = 0, Last = 1, newnode(); // 根节点为 1
    }

    void extend(int c) {
        int p = Last;
        if (t[p].trans[c]) {
            int q = t[p].trans[c];
            if (t[q].maxl == t[p].maxl + 1) {
                Last = q;
            } else {
                int nq = newnode();
                t[nq] = t[q], t[nq].maxl = t[p].maxl + 1;
                t[q].link = nq;
                for (; p && t[p].trans[c] == q; p = t[p].link) {
                    t[p].trans[c] = nq;
                }
                Last = nq;
            }
        } else {
            int np = Last = newnode();

            t[np].maxl = t[p].maxl + 1;
            for (; p && t[p].trans[c] == 0; p = t[p].link) {
                t[p].trans[c] = np;
            }

            if (!p) {
                t[np].link = 1;
            } else {
                int q = t[p].trans[c];
                if (t[q].maxl == t[p].maxl + 1) {
                    t[np].link = q;
                } else {
                    int nq = newnode();
                    t[nq] = t[q], t[nq].maxl = t[p].maxl + 1;
                    t[np].link = t[q].link = nq;
                    for (; p && t[p].trans[c] == q; p = t[p].link) {
                        t[p].trans[c] = nq;
                    }
                }
            }
        }
    }
```

```
}
```

PAM

PAM：PAM 是一个状态自动机，形态为由两棵树构成的森林。

- 两个起点（偶根为 0，奇根为 1），若干终点。
- 原串的所有本质不同的回文子串，与 PAM 上的所有状态一一对应，不重不漏。
- PAM 主要考虑两种边：
  - 普通转移边：表示在当前状态所表示的串前后各加一个字符。
  - 后缀链接 link：表示当前状态所表示的串的最长回文后缀。这类边构成一棵树，称作 parent 树。
- 偶根的 link 指向奇根，而我们不关心奇根的 link（因为奇根不可能失配，奇根转移出的下一个状态是单个字符，必定为回文串）。
- PAM 点数（本质不同回文子串个数）$\leq n$。因为在任意串后面加一个字符，由反证法可知本质不同回文子串至多 $+1$。

0x22 PAM.cpp

```
// 回文自动机
namespace PAM {
    int strL, str[N];
    int nodeCount, Last;
    struct node {
        int trans[26];
        int link, len;
    } t[N];

    int newnode() {
        int p = ++ nodeCount;
        for (int c = 0; c < 26; c ++) t[p].trans[c] = 0;
        t[p].link = t[p].len = 0;
        return p;
    }
    void init() {
        nodeCount = -1, Last = 1, newnode(), newnode(); // 偶根为 0，奇根为 1
        str[strL = 0] = -1;
        t[0].len = 0, t[0].link = 1;
        t[1].len = -1;
    }

```

```
    int find(int p) {
        while (str[strL - t[p].len - 1] != str[strL]) {
            p = t[p].link;
        }
        return p;
    }
    void extend(int c) {
        str[++ strL] = c;
        int p = find(Last);
        if (!t[p].trans[c]) {
            int np = newnode();
            t[np].len = t[p].len + 2;
            t[np].link = t[find(t[p].link)].trans[c];
            t[p].trans[c] = np;
        }
        Last = t[p].trans[c];
    }
}
```

PAM half 指针：表示当前状态所表示的串，长度小于等于该串一半的最长回文后缀。

0x22 PAM half.cpp

```
// 回文自动机 half
namespace PAM {
    int strL, str[N];
    int nodeCount, Last;
    struct node {
        int trans[26];
        int link, half, len;
    } t[N];

    int newnode() {
        int p = ++ nodeCount;
        for (int c = 0; c < 26; c ++) t[p].trans[c] = 0;
        t[p].link = t[p].half = t[p].len = 0;
        return p;
    }
    void init() {
        nodeCount = -1, Last = 1, newnode(), newnode(); // 偶根为 0，奇根为 1
        str[strL = 0] = -1;
        t[0].len = 0, t[0].link = 1;
        t[1].len = -1;
    }

    int find(int p) {
        while (str[strL - t[p].len - 1] != str[strL]) {
            p = t[p].link;
        }
        return p;
    }
    int find_half(int p, int L) {
        while (t[p].len + 2 > L || str[strL - t[p].len - 1] != str[strL]) {
            p = t[p].link;
        }
        return p;
    }
    void extend(int c) {
        str[++ strL] = c;
        int p = find(Last);
        if (!t[p].trans[c]) {
            int np = newnode();
            t[np].len = t[p].len + 2;
            t[np].link = t[find(t[p].link)].trans[c];
            t[np].half = t[np].len > 2 ? t[find_half(t[p].half, t[np].len / 2)].trans[c] : t[np].link;
            t[p].trans[c] = np;
        }
        Last = t[p].trans[c];
    }
}
```

Palindrome Series

当 t 是回文串 s 的后缀时，t 是 s 的 border $\iff$ t 是回文串。

所以回文串 s 的回文后缀一定是 **border**，反之亦然。

Palindrome Series：回文串 s 的所有回文后缀，可以被划分成至多 $\lceil \log_2 n \rceil$ 个等差数列。进一步，每个等差数列都具有一定的周期性。同一个等差数列里的字符串，都可以被表示成

$$W, W + A, W + A^2, \dots, W + A^k$$

或

$$W, B + W, B^2 + W, \dots, B^k + W$$

其中 W, A, B 均为字符串。

PAM series-link：对于 PAM 上的每个状态 u

- $\text{diff}_u = \text{len}_u - \text{len}_{\text{link}_u}$。

- slink_u 表示 u 不断跳后缀链接 link，跳到的第一个满足 $\text{diff}_v \neq \text{diff}_u$ 的节点 v。

我们称 **diff 值相同的连续 link 链为一个 Series**，Series 中的回文串长度都构成一个等差数列。那么 slink_u 就表示 u 所在的 Series 对应的下一个 Series 的最长回文节点。

上述理论可以用来优化一类 dp 问题：

$$f_i = \bigoplus_{s[j+1:i] \text{ is Palindrome}} f_j$$

（其中 $\oplus$ 表示一种运算，例如 $+$ 或者 $\min$）

考虑将一个 Series 表示的所有回文串的 dp 值都记录到最长的回文串上，设 g_u 表示 u 所在 Series 的 dp 贡献之和（其中 u 作为该 Series 中最长回文节点）。

若节点 u 最近一次在位置 i 结尾，且 $\text{diff}_u = \text{diff}_{\text{link}_u}$，则 link_u 在 i 之前（严格早于 i）的最近一次出现，结尾于位置 $i - \text{diff}_u$。

假设当前处理到位置 i，沿着 slink 处理每一个 Series 的贡献。设当前节点为 u，则 u 所在 Series 中最短回文串长度为 $\text{diff}_u + \text{len}_{\text{slink}_u}$，除了最短回文串的其他贡献，恰好等于在位置 $i - \text{diff}_u$ 时已经维护好的 g_{link_u}。因此有转移：

$$g_u = \begin{cases} f_{i-\text{diff}_u-\text{len}_{\text{slink}_u}} & \text{diff}_u \neq \text{diff}_{\text{link}_u} \\ f_{i-\text{diff}_u-\text{len}_{\text{slink}_u}} \oplus g_{\text{link}_u} & \text{diff}_u = \text{diff}_{\text{link}_u} \end{cases}$$

0x22 PAM series-link.cpp

```cpp
// 回文自动机 series-link
namespace PAM {
    int strL, str[N];
    int nodeCount, Last;
    struct node {
        int trans[26];
        int link, len;
        int diff, slink;
    } t[N];
    int g[N];

    int newnode() {
        int p = ++ nodeCount;
        for (int c = 0; c < 26; c ++) t[p].trans[c] = 0;
        t[p].link = t[p].len = 0;
        t[p].diff = t[p].slink = 0;
        g[p] = 0x3f3f3f3f;
        return p;
    }
    void init() {
        nodeCount = -1, Last = 1, newnode(), newnode(); // 偶根为 0，奇根为 1
        str[strL = 0] = -1;
        t[0].len = 0, t[0].link = 1;
        t[1].len = -1;
    }

    int find(int p) {
        while (str[strL - t[p].len - 1] != str[strL]) {
            p = t[p].link;
        }
        return p;
    }
    void extend(int c) {
        str[++ strL] = c;
        int p = find(Last);
        if (!t[p].trans[c]) {
            int np = newnode();
            t[np].len = t[p].len + 2;
            t[np].link = t[find(t[p].link)].trans[c];
            t[np].diff = t[np].len - t[t[np].link].len;
            t[np].slink = t[np].diff == t[t[np].link].diff ?
                t[t[np].link].slink : t[np].link;
            t[p].trans[c] = np;
        }
        Last = t[p].trans[c];
    }

    int ask(int x, int i) {
        int ans = 0x3f3f3f3f;
        for (int u = x; u > 1; u = t[u].slink) {
            g[u] = f[i - t[u].diff - t[t[u].slink].len];
            if (t[u].diff == t[t[u].link].diff) {
                chmin(g[u], g[t[u].link]);
            }
            chmin(ans, g[u]);
        }
        return ans;
    }
}
```

0x31 数学 多项式

拉格朗日插值

多项式的插值：给出 $n+1$ 个点 $(x_1,y_1),(x_2,y_2),\dots,(x_{n+1},y_{n+1})$，其中横坐标（模意义下）两两不同。求一个至多 n 次的多项式，使得这 $n+1$ 个点都在 $f(x)$ 上。

拉格朗日插值：

$$f(x)=\sum_{i=1}^{n+1}y_i\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

可以发现 $f(x)$ 是一个至多 n 次的多项式，并且可以证明拉格朗日插值是唯一的插值多项式。

朴素实现的时间复杂度为 $\mathcal{O}(n^2)$，可以使用多项式优化到 $\mathcal{O}(n\log^2 n)$（详见多项式快速插值）。

0x31 拉格朗日插值.cpp

```cpp
// 拉格朗日插值
int lagrange(std::vector<std::pair<int, int>> seq, int k) {
    int n = seq.size(); // seq 下标从 0 开始，至多 n-1 次多项式需要 n 个点值
    int ans = 0;
    for (int i = 0; i < n; i ++) {
        int p = 1, q = 1;
        for (int j = 0; j < n; j ++) {
            if (i == j) continue;
            mul(p, k - seq[j].first);
            mul(q, seq[i].first - seq[j].first);
        }
        ans = (ans + 1ll * p * qpow(q, mod - 2, mod) % mod * seq[i].second)
         ↪ % mod;
    }
    return norm(ans);
}
```

拉格朗日插值求系数：相当于是直接将整个多项式求出来。运用了暴力除以一次式的技巧。

- 记系数部分 $a_i=\frac{y_i}{\prod_{j\neq i}(x_i-x_j)}$。
- 记多项式 $g(x)=\prod_j(x-x_j)$，记多项式 $h_i(x)=\prod_{j\neq i}(x-x_j)$。
- 先 $\mathcal{O}(n^2)$ 暴力多项式乘法求出 $g(x)$。考虑由 $g(x)$ 得到 $h_i(x)$，相当于是多项式 $g(x)$ 除以一次式 $(x-x_i)$，可以 $\mathcal{O}(n)$ 暴力除以一次式。
- 具体地，由于 $(x-x_i)h_i(x)=g(x)$，则对于 $g(x)$ 的 j $(j\geq 1)$ 次项系数，有 $[x^{j-1}]h_i-x_i[x^j]h_i=[x^j]g$。于是

$$[x^n]h_i=[x^{n+1}]g$$
$$[x^j]h_i=[x^{j+1}]g+x_i[x^{j+1}]h_i$$

从后往前递推即可求出 $h_i(x)$。

- 求出了 a_i 与 $h_i(x)$ 之后，计算 $f(x)=\sum_{i=1}^{n+1}a_ih_i(x)$。时间复杂度 $\mathcal{O}(n^2)$。

0x31 拉格朗日插值求系数.cpp

```cpp
// 拉格朗日插值求系数
std::vector<int> lagrange(std::vector<std::pair<int, int>> seq) {
    int n = seq.size(); // seq 下标从 0 开始，至多 n-1 次多项式需要 n 个点值

    std::vector<int> a(n); // 系数部分 a[]
    for (int i = 0; i < n; i ++) {
        int q = 1;
        for (int j = 0; j < n; j ++) {
            if (i == j) continue;
            mul(q, seq[i].first - seq[j].first);
        }
        a[i] = 1ll * seq[i].second * qpow(q, mod - 2, mod) % mod;
    }

    std::vector<int> g(n + 1); // 总多项式 g
    g[0] = 1;
    for (int i = 0; i < n; i ++) {
        for (int j = i + 1; j >= 1; j --) {
            g[j] = (1ll * g[j] * -seq[i].first + g[j - 1]) % mod;
        }
        g[0] = 1ll * g[0] * -seq[i].first % mod;
    }

    std::vector<int> f(n); // 答案多项式 f
    for (int i = 0; i < n; i ++) {
        std::vector<int> h(n); // 分多项式 h_i
        h[n - 1] = g[n];
        for (int j = n - 2; j >= 0; j --) {
            h[j] = (g[j + 1] + 1ll * h[j + 1] * seq[i].first) % mod;
        }

        for (int j = 0; j < n; j ++) {
            f[j] = (f[j] + 1ll * a[i] * h[j]) % mod;
        }
```

```
    }

    for (int &x : f) x = norm(x);
    return f;
}

int func(const std::vector<int> &f, int k) {
    int n = f.size();
    int y = 0;
    for (int i = n - 1; i >= 0; i --) {
        y = (1ll * y * k + f[i]) % mod;
    }
    return norm(y);
}
```

连续点值的拉格朗日插值：给出 $n+1$ 个连续点值 $f(1), f(2), \dots, f(n+1)$，代入拉格朗日插值公式，得

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

现在考虑给定一个自变量 x，求出因变量 $f(x)$。考虑右边的乘积项怎么求

- 分子：记 $\text{pre}_i = \prod_{1 \le j \le i} (x-j)$，$\text{suf}_i = \prod_{i \le j \le n+1} (x-j)$，则分子为 $\text{pre}_{i-1} \times \text{suf}_{i+1}$。
- 分母：$j < i$ 的部分是 $i-1$ 到 1 乘起来。$j > i$ 的部分是 -1 到 $-(n+1-i)$ 乘起来。故分母为 $(i-1)!(n+1-i)!(-1)^{n+1-i}$。

故最后的式子为

$$f(x) = \sum_{i=1}^{n+1} y_i \frac{\text{pre}_{i-1} \times \text{suf}_{i+1}}{(i-1)!(n+1-i)!(-1)^{n+1-i}}$$

求出单个点值的时间复杂度为 $\mathcal{O}(n)$。

> 若给出 $n+1$ 个连续点值 $f(h), f(h+1), \dots, f(h+n)$，分母的求法是完全不变的，只需将分子改成 $x - x_j$ 相乘即可。
> 或者可以考虑将坐标系向左平移 $h-1$。

0x31 连续点值的拉格朗日插值.cpp

```
// 连续点值的拉格朗日插值
int lagrange(std::vector<int> y, int k) {
    int n = y.size() - 1; // y 下标从 1 开始，连续点值为 f(1), ..., f(n)
    if (k >= 1 && k <= n) {
        return y[k];
    }

    std::vector<int> pre(n + 2), suf(n + 2);
    pre[0] = 1;
    for (int i = 1; i <= n; i ++) {
        pre[i] = 1ll * pre[i - 1] * (k - i) % mod;
    }
    suf[n + 1] = 1;
    for (int i = n; i >= 1; i --) {
        suf[i] = 1ll * suf[i + 1] * (k - i) % mod;
    }

    std::vector<int> fact(n + 1), facv(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++) {
        fact[i] = 1ll * fact[i - 1] * i % mod;
    }
    facv[n] = qpow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i --) {
        facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
    }

    int ans = 0;
    for (int i = 1; i <= n; i ++) {
        int cur = y[i];
        cur = 1ll * cur * pre[i - 1] % mod * suf[i + 1] % mod;
        cur = 1ll * cur * facv[i - 1] % mod * facv[n - i] % mod;
        if ((n - i) & 1) cur = -cur;
        add(ans, norm(cur));
    }

    return ans;
}
```

FFT

单位根：n 次单位根，即为 $x^n = 1$ 的所有复数解。共 n 个，依次表示为 $\omega_n^0, \omega_n^1, \omega_n^2, \dots, \omega_n^{n-1}$，在单位圆上的辐角分别为 $0, \frac{2\pi}{n}, \frac{4\pi}{n}, \dots, \frac{2(n-1)\pi}{n}$。由欧拉公式 $\omega_n^1 = e^{\frac{2\pi i}{n}} = \cos(\frac{2\pi}{n}) + i \sin(\frac{2\pi}{n})$。

单位根的简单性质：

- $\omega_n^i \times \omega_n^j = \omega_n^{i+j}$，$(\omega_n^i)^j = \omega_n^{ij}$。

- $\omega_{2n}^{2k}=\omega_n^k$
- $\omega_{2n}^{k}=-\omega_{2n}^{k+n}$

补充界：在进行 FFT 之前，需要对其最高位补 0，将其补成一个最高次数为 2^P-1 的多项式。方便后续的奇偶性分组。

DFT：DFT 的思想：“系数表示法” 转 “点值表示法”。

已知系数 $a_0,\dots,a_{n-1}$，计算点值 $F(\omega_n^0),F(\omega_n^1),\dots,F(\omega_n^{n-1})$。

记 $n=2^P$，考虑一个 2^P-1 次的多项式 $F(x)$。将每一项按照次数的奇偶性，划分成两部分。记 $F_L(x)=a_0+a_2x+\cdots+a_{n-2}x^{n/2-1}$，$F_R(x)=a_1+a_3x+\cdots+a_{n-1}x^{n/2-1}$，则

$$F(x)=F_L(x^2)+xF_R(x^2)$$

对于 $k<n/2$，将 ω_n^k 与 $\omega_n^{k+n/2}$ 代入上式，由单位根的性质可得 $F(\omega_n^k),F(\omega_n^{k+n/2})$ 等于 $F_L(\omega_{n/2}^k)\pm\omega_n^kF_R(\omega_{n/2}^k)$。

因此由 $F_L(x),F_R(x)$ 在 $\omega_{n/2}^0,\dots,\omega_{n/2}^{n/2-1}$ 处的点值，可以 $\mathcal{O}(n)$ 推出 $F(x)$ 在 $\omega_n^0,\dots,\omega_n^{n-1}$ 处的点值。

这是一个分治的过程，不断分治直到剩下一个项即可。时间复杂度 $\mathcal{O}(n\log n)$。

IDFT：IDFT 的思想：“点值表示法” 转 “系数表示法”。

已知点值 $F(\omega_n^0),F(\omega_n^1),\dots,F(\omega_n^{n-1})$，求出系数 $a_0,\dots,a_{n-1}$。

结论：将 DFT 中的 ω_n^1 换成 ω_n^{-1}，做完以后除以 n 即可。

由单位根反演的经典结论 $[n\mid k]=\frac{1}{n}\sum_{i=0}^{n-1}\omega_n^{ik}$，可以证明：

$$g_k=\sum_{i=0}^{n-1}a_i\left(\omega_n^k\right)^i\iff a_k=\frac{1}{n}\sum_{i=0}^{n-1}g_i\left(\omega_n^{-k}\right)^i$$

再进一步，发现 $F(\omega_n^{-i})=F(\omega_n^{n-i})$，于是 IDFT 可以看作将 DFT 后得到的点值进行翻转（注意 $F(\omega_n^0)$ 这一项不参与翻转），然后再除以 n。套用 DFT 的板子即可。

位逆序置换（FFT 倍增实现）：

以 $n=8$ 为例，模拟分治的过程：

$$\{a_0,a_1,a_2,a_3,a_4,a_5,a_6,a_7\}$$
$$\{a_0,a_2,a_4,a_6\}\{a_1,a_3,a_5,a_7\}$$
$$\{a_0,a_4\}\{a_2,a_6\}\{a_1,a_5\}\{a_3,a_7\}$$
$$\{a_0\}\{a_4\}\{a_2\}\{a_6\}\{a_1\}\{a_5\}\{a_3\}\{a_7\}$$

分治自顶向下，每层按照下标奇偶性划分；倍增则自底向上，因此需要预先确定每个 a_i 在最底层的位置。

将 i 用二进制数表示，则 a_i 在最底层的位置即为**该二进制数的翻转**（例如 `011000` 变成 `000110`）。证明：每次根据奇偶性将 a_i 进行分类，相当于是由最低位决定最高位。以此类推，次低位决定次高位，第三低位决定第三高位...

蝶形运算（FFT 倍增实现）：

自底向上合并，对于当前长度为 n 的块，已知左右两块分别存放 F_L,F_R 的 $n/2$ 个点值。

- 更新前：$F_L(\omega_{n/2}^k)$ 储存在数组下标为 k 的位置，$F_R(\omega_{n/2}^k)$ 储存在数组下标为 $k+n/2$ 的位置。
- 更新后：$F(\omega_n^k)$ 储存在数组下标为 k 的位置，$F(\omega_n^{k+n/2})$ 储存在数组下标为 $k+n/2$ 的位置。

因此可以直接在数组下标为 k 和 $k+n/2$ 的位置进行覆写，而不用开额外的辅助数组。

FFT 多项式乘法：

对于两个多项式 A,B，现在要求出多项式 $C=A\times B$。

考虑对 A,B 进行 DFT，经过 DFT 之后得到一系列关于 A,B 的点值，将关于 A,B 的点值对应位置相乘即可得到 C 的点值，再对 C 的点值进行 IDFT 即可得到 C 的系数。

时间复杂度 $\mathcal{O}((n+m)\log(n+m))$。

循环卷积：对长度为 L 的两个序列进行 DFT，将点值对应位置相乘，再 IDFT 回去。实际上是做长度为 L 的循环卷积，即 a_ib_j 贡献给了 $c_{(i+j)\bmod L}$。当补充界 $L\geq n+m+1$ 时，即为正常卷积。只需要部分项时，可以据此缩小补充界（见半在线卷积的卡常技巧）。

0x31 FFT.cpp

```cpp
const double pi = acos(-1);
using comp = std::complex<double>;

std::vector<comp> roots{0, 1};
void dft(std::vector<comp> &a) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i ++) {
        for (int k = n >> 1; (j ^= k) < k; k >>= 1);
        if (i < j) std::swap(a[i], a[j]);
    }
    for (int k = roots.size(); k < n; k <<= 1) {
        comp w(cos(pi / k), sin(pi / k));
        roots.resize(k << 1);
```

```
        for (int i = k >> 1; i < k; i ++) {
            roots[i * 2] = roots[i];
            roots[i * 2 + 1] = roots[i] * w;
        }
    }
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < n; i += (k << 1)) {
            for (int j = 0; j < k; j ++) {
                comp u = a[i + j], v = a[i + j + k] * roots[k + j];
                a[i + j] = u + v, a[i + j + k] = u - v;
            }
        }
    }
}
void idft(std::vector<comp> &a) {
    int n = a.size();
    std::reverse(a.begin() + 1, a.end());
    dft(a);
    for (int i = 0; i < n; i ++) {
        a[i] /= n;
    }
}

struct poly : public std::vector<comp> {
    using std::vector<comp>::vector;
    explicit poly(const std::vector<comp> &a) : std::vector<comp>(a) {}

    friend poly operator * (poly a, poly b) {
        int tot = a.size() + b.size() - 1;
        if (tot < 128) {
            poly c(tot);
            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;
        }
        int L = 1;
        while (L < tot) L <<= 1;
        a.resize(L), b.resize(L);
        dft(a), dft(b);
        for (int i = 0; i < L; i ++) {
            a[i] *= b[i];
        }
        idft(a);
        a.resize(tot);
        return a;
    }
};
```

NTT

在模意义下，我们希望找到一个东西来代替单位根。主要是要满足 FFT 过程中，用到的单位根的三个性质：

- ω_n 的阶为 n。
- $\omega_{2n}^{2k} = \omega_n^k$。
- $\omega_{2n}^{k} = -\omega_{2n}^{n+k}$。

对于一个质数 m，考虑 m 的一个原根 g。此时 g 的阶为 $m-1$，显然原根 g 不能直接代替 n 阶单位根。

考虑 g 的若干次幂。g^k 的阶为 $\frac{m-1}{\gcd(m-1,k)}$。**于是当 $n \mid (m-1)$ 时，可以构造 $g^{\frac{m-1}{n}}$ 的阶恰好等于 n**。尝试使用 $g^{\frac{m-1}{n}}$ 代替 n 阶单位根：

- 性质 1, 2 的证明显然。
- 性质 3 相当于是要证明 $g^{\frac{m-1}{2}} \equiv -1 \pmod m$，可以使用二次剩余中的欧拉判别法来证明（原根必定不是二次剩余）。

于是当 $n \mid (m-1)$ 时，我们完全可以使用 $g^{\frac{m-1}{n}}$ 来代替 n 阶单位根 ω_n。

现在还要满足 $n \mid (m-1)$ 才可以代替，对模数 m 的取值有特殊的要求。注意到 FFT 过程中的 n 都是 2 的若干次幂，所以 $m-1$ 只需要包含大量的因子 2 即可。

例如 $998244353 = 2^{23} \times 7 \times 17 + 1$，就是一个经典的 NTT 模数。

常用的 NTT 模数：

- `998244353`：原根为 3。
- `1004535809`：原根为 3。
- `469762049`：原根为 3。
- `167772161`：原根为 3。
- `924844033`：原根为 5。
- `4179340454199820289`（4×10^{18} 级别，以时间换精度）：原根为 3。

0x31 NTT.cpp

```cpp
std::vector<int> roots{0, 1};
void dft(std::vector<int> &a) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i ++) {
        for (int k = n >> 1; (j ^= k) < k; k >>= 1);
        if (i < j) std::swap(a[i], a[j]);
    }
    for (int k = roots.size(); k < n; k <<= 1) {
        int w = qpow(3, (mod - 1) / (k << 1), mod); // 原根根据实际模数调整
        roots.resize(k << 1);
        for (int i = k >> 1; i < k; i ++) {
            roots[i * 2] = roots[i];
            roots[i * 2 + 1] = 1ll * roots[i] * w % mod;
        }
    }
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < n; i += (k << 1)) {
            for (int j = 0; j < k; j ++) {
               int u = a[i + j], v = 1ll * a[i + j + k] * roots[k + j] % mod;
                add(a[i + j] = u, v), dec(a[i + j + k] = u, v);
            }
        }
    }
}
void idft(std::vector<int> &a) {
    int n = a.size(), inv = (1 - mod) / n + mod;
    std::reverse(a.begin() + 1, a.end());
    dft(a);
    for (int i = 0; i < n; i ++) {
        mul(a[i], inv);
    }
}

struct poly : public std::vector<int> {
    using std::vector<int>::vector;
    explicit poly(const std::vector<int> &a) : std::vector<int>(a) {}

    friend poly operator * (poly a, poly b) {
        int tot = a.size() + b.size() - 1;
        if (tot < 128) {
            poly c(tot);
            for (int i = 0; i < a.size(); i ++) {
                for (int j = 0; j < b.size(); j ++) {
                    c[i + j] = (c[i + j] + 1ll * a[i] * b[j]) % mod;
                }
            }
            return c;
        }
        int L = 1;
        while (L < tot) L <<= 1;
        a.resize(L), b.resize(L);
        dft(a), dft(b);
        for (int i = 0; i < L; i ++) {
            mul(a[i], b[i]);
        }
        idft(a);
        a.resize(tot);
        return a;
    }
};
```

MTT

当模数不为常用的 NTT 模数时，我们就需要使用一些技巧使得多项式乘法可以继续进行。

首先不可以简单地使用 FFT，因为 `double` 的有效精度只有 $15 \sim 16$ 位十进制有效数字，而对于常见的数据范围 $n, m \leq 10^5$ 以及 $a_i, b_i \leq 10^9$，卷积结果的最大值接近 $10^5 \times 10^9 \times 10^9 = 10^{23}$，一定会掉精度。

三模 NTT

选择三个 10^9 级别的常用 NTT 模数，**用三个模数分别做一次 NTT，将得到的三个结果使用 CRT 合并。**

常用的 NTT 模数选法如下，因为这三个 NTT 模数均有一个原根为 3。

$$\begin{aligned} \text{mod1} &= 998244353 \\ \text{mod2} &= 1004535809 \\ \text{mod3} &= 469762049 \end{aligned}$$

该做法的优点是，可以通过 CRT 精确得到未经取模的卷积结果，对值域的范围要求比较友好。

0x31 MTT（三模 NTT）.cpp

```cpp
template <const int mod>
struct ntt {
    inline void add(int &x, const int &y) { x += y; if (x >= mod) x -= mod; }
    inline void dec(int &x, const int &y) { x -= y; if (x < 0) x += mod; }
    inline void mul(int &x, const int &y) { x = 1ll * x * y % mod; }

    std::vector<int> roots{0, 1};
    void dft(std::vector<int> &a) {
        int n = a.size();
        for (int i = 1, j = 0; i < n; i ++) {
            for (int k = n >> 1; (j ^= k) < k; k >>= 1);
            if (i < j) std::swap(a[i], a[j]);
        }
        for (int k = roots.size(); k < n; k <<= 1) {
            int w = qpow(3, (mod - 1) / (k << 1), mod);
            roots.resize(k << 1);
            for (int i = k >> 1; i < k; i ++) {
                roots[i * 2] = roots[i];
                roots[i * 2 + 1] = 1ll * roots[i] * w % mod;
            }
        }
        for (int k = 1; k < n; k <<= 1) {
            for (int i = 0; i < n; i += (k << 1)) {
                for (int j = 0; j < k; j ++) {
                    int u = a[i + j], v = 1ll * a[i + j + k] * roots[k + j] %
                      ↪ mod;
                    add(a[i + j] = u, v), dec(a[i + j + k] = u, v);
                }
            }
        }
    }
    void idft(std::vector<int> &a) {
        int n = a.size(), inv = (1 - mod) / n + mod;
        std::reverse(a.begin() + 1, a.end());
        dft(a);
        for (int i = 0; i < n; i ++) {
            mul(a[i], inv);
        }
    }

    auto mul(std::vector<int> a, std::vector<int> b) {
        for (int &x : a) x %= mod;
        for (int &x : b) x %= mod;
        dft(a), dft(b);
        for (int i = 0; i < a.size(); i ++) {
            mul(a[i], b[i]);
        }
        idft(a);
        return a;
    }
};

const int mod1 = 998244353, mod2 = 1004535809, mod3 = 469762049;
ntt<mod1> ntt1;
ntt<mod2> ntt2;
ntt<mod3> ntt3;

int CRT(int x1, int x2, int x3) {
    static const i64 M12 = 1ll * mod1 * mod2;
    static const int
        inv1 = qpow(mod1, mod2 - 2, mod2),
        inv12 = qpow(M12 % mod3, mod3 - 2, mod3);
    int t1 = 1ll * (x2 - x1 + mod2) * inv1 % mod2;
    i64 x12 = x1 + 1ll * mod1 * t1;
    int t2 = (x3 - x12 % mod3 + mod3) * inv12 % mod3;
    return (x12 % mod + (M12 % mod) * t2) % mod;
}

// int CRT(int x1, int x2, int x3) {
//     static const i128 M = i128(mod1) * mod2 * mod3;
//     static const i64 M1 = M / mod1, M2 = M / mod2, M3 = M / mod3;
//     static const int
//         V1 = qpow(M1 % mod1, mod1 - 2, mod1),
//         V2 = qpow(M2 % mod2, mod2 - 2, mod2),
//         V3 = qpow(M3 % mod3, mod3 - 2, mod3);
//     i128 x = 0;
//     x += i128(x1) * M1 * V1;
//     x += i128(x2) * M2 * V2;
//     x += i128(x3) * M3 * V3;
//     x %= M; // 此时的 x 即为 CRT 的解
//     return x % mod;
// }

struct poly : public std::vector<int> {
    using std::vector<int>::vector;
    explicit poly(const std::vector<int> &a) : std::vector<int>(a) {}

    friend poly operator * (poly a, poly b) {
        int tot = a.size() + b.size() - 1;
        if (tot < 128) {
```

```
            poly c(tot);
            for (int i = 0; i < a.size(); i ++) {
                for (int j = 0; j < b.size(); j ++) {
                    c[i + j] = (c[i + j] + 1ll * a[i] * b[j]) % mod;
                }
            }
            return c;
        }
        int L = 1;
        while (L < tot) L <<= 1;
        a.resize(L), b.resize(L);
        auto c1 = ntt1.mul(a, b), c2 = ntt2.mul(a, b), c3 = ntt3.mul(a, b);
        poly c(tot);
        for (int i = 0; i < tot; i ++) {
            c[i] = CRT(c1[i], c2[i], c3[i]);
        }
        return c;
    }
};
```

拆系数 FFT

当 $\mathrm{mod} < 2^{30}$ 时，将每个系数拆成两个 15 位的数（$x = x_0 + x_1 \times 2^{15}$），然后将低位 x_0 和高位 x_1 分别作为复数的实部和虚部进行 DFT。此时卷积结果就可以被 `double` 精确表示。

该做法的优点是，只需要进行 4 次 DFT，效率较高，代码也相对好写。缺点是 n 较大时，可能会掉精度。

0x31 MTT（拆系数 FFT）.cpp

```
const double pi = acos(-1);
using comp = std::complex<double>;

std::vector<comp> roots{0, 1};
void dft(std::vector<comp> &a) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i ++) {
        for (int k = n >> 1; (j ^= k) < k; k >>= 1);
        if (i < j) std::swap(a[i], a[j]);
    }
    for (int k = roots.size(); k < n; k <<= 1) {
        comp w(cos(pi / k), sin(pi / k));
        roots.resize(k << 1);
        for (int i = k >> 1; i < k; i ++) {
            roots[i * 2] = roots[i];
            roots[i * 2 + 1] = roots[i] * w;
        }
    }
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < n; i += (k << 1)) {
            for (int j = 0; j < k; j ++) {
                comp u = a[i + j], v = a[i + j + k] * roots[k + j];
                a[i + j] = u + v, a[i + j + k] = u - v;
            }
        }
    }
}
void idft(std::vector<comp> &a) {
    int n = a.size();
    std::reverse(a.begin() + 1, a.end());
    dft(a);
    for (int i = 0; i < n; i ++) {
        a[i] /= n;
    }
}

struct poly : public std::vector<int> {
    using std::vector<int>::vector;
    explicit poly(const std::vector<int> &a) : std::vector<int>(a) {}

    friend poly operator * (poly a, poly b) {
        int tot = a.size() + b.size() - 1;
        if (tot < 128) {
            poly c(tot);
            for (int i = 0; i < a.size(); i ++) {
                for (int j = 0; j < b.size(); j ++) {
                    c[i + j] = (c[i + j] + 1ll * a[i] * b[j]) % mod;
                }
            }
            return c;
        }
        int L = 1;
        while (L < tot) L <<= 1;
        std::vector<comp> P(L), Q(L), R(L), S(L);
        for (int i = 0; i < a.size(); i ++) {
            P[i] = comp(a[i] & 32767, a[i] >> 15);
        }
        for (int i = 0; i < b.size(); i ++) {
            Q[i] = comp(b[i] & 32767, b[i] >> 15);
        }
```

```
        dft(P), dft(Q);
        for (int i = 0; i < L; i ++) {
            int j = (L - i) & (L - 1);
            comp a0 = (P[i] + std::conj(P[j])) * 0.5;
            comp a1 = (P[i] - std::conj(P[j])) * comp(0, -0.5);
            R[i] = a0 * Q[i], S[i] = a1 * Q[i];
        }
        idft(R), idft(S);
        poly c(tot);
        for (int i = 0; i < tot; i ++) {
            i64 c0 = llround(R[i].real()) % mod;
            i64 c1 = (llround(R[i].imag()) + llround(S[i].real())) % mod;
            i64 c2 = llround(S[i].imag()) % mod;
            c[i] = ((c2 << 30) + (c1 << 15) + c0) % mod;
        }
        return c;
    }
};
```

Trick：翻转卷积

给出序列 $a_0,\dots,a_n$ 与序列 $b_0,\dots,b_m$，对于每一个 k 需要求出 $c_k=\sum_i a_ib_{i-k}$，即下标差恒为 k 的卷积式（相当于 a_ib_j 贡献给了 c_{i-j}）。

将序列 b 翻转得到 b'，其中 $b'_{m-j}=b_j$。对序列 a,b' 进行普通卷积得到序列 c'，此时 a_ib_j 贡献给了 c'_{m+i-j}，于是有 $c_k=c'_{k+m}$。

Trick：积卷积

给出序列 $a_1,\dots,a_{p-1}$ 与序列 $b_1,\dots,b_{p-1}$，对于每一个 $k\in[1,p)$ 需要求出 $c_k=\sum_{i\cdot j \bmod p=k} a_ib_j$。

当模数为较小的质数 p 时，可以使用“离散对数”，将模 p 意义下的乘法转换为模 $p-1$ 意义下的加法。

Trick：多项式点乘向量

给出多项式 $A(x)=\sum_{i=0}^n a_ix^i$ 与向量 $\mathbf{B}=(b_0,\dots,b_n)$，现在我们要求多项式 A 点乘向量 $\mathbf{B}$ 的值

$$A\cdot\mathbf{B}=\sum_{i=0}^n a_ib_i$$

记翻转多项式 $B^R(x)=\sum_{i=0}^n b_{n-i}x^i$，则点乘的值即为 $[x^n](A\times B^R)$，即 $A\times B^R$ 的其中一项。

> e.g. 给出多项式序列 $p_1,\dots,p_n$，欲求 $\left(\prod_{i=l}^r p_i\right)\cdot\mathbf{B}$，即求 $\text{pre}_r\times\text{ipre}_{l-1}\times B^R$ 的第 n 项。
>
> 预处理出所有的 $\text{pre}_i\times B^R$ 以及 ipre_i 即可。

Model：分治卷积

分治 NTT 模型 1：给出 n 个数 $a_1,\dots,a_n$，求出 n 个一次式的积 $\prod_{i=1}^n(a_ix+1)$。

设 $f(l,r)$ 表示区间 $[l,r]$ 中所有一次式的积，$f(l,r)$ 是一个次数为 $r-l+1$ 的多项式(与区间长度有关)。

取 $\text{mid}=\lfloor\frac{l+r}{2}\rfloor$，则有

$$f(l,r)=f(l,\text{mid})\times f(\text{mid}+1,r)$$

这是一个分治的过程，不断分治直到 $l=r$ 即可。

时间复杂度 $T(n)=2T\left(\frac{n}{2}\right)+\mathcal{O}(n\log n)=\mathcal{O}(n\log^2 n)$。

> By the way：求出 n 个多项式的积，次数之和为 m。
>
> 可以每次取出两个次数最小的多项式相乘后放回，时间复杂度 $\mathcal{O}(m\log^2 m)$。
>
> 其实使用分治求解的时间复杂度也是相同的。

分治 NTT 模型 2：给出 n 个数 $a_1,\dots,a_n$，求出 n 个一次分式的和 $\sum_{i=1}^n\frac{1}{1-a_ix}$。

对于分式 $\frac{A_1}{A_2},\frac{B_1}{B_2}$，若我们已知分子 A_1,B_1 与分母 A_2,B_2，由通分可得 $\frac{A_1}{A_2}+\frac{B_1}{B_2}=\frac{A_1B_2+A_2B_1}{A_2B_2}$，由此可以得到新的分子 $A_1B_2+A_2B_1$ 与分母 A_2B_2。

设 $A(l,r),B(l,r)$ 分别表示区间 $[l,r]$ 中所有一次分式之和通分后所得的分子与分母。

$A(l,r),B(l,r)$ 的次数同样与区间长度有关。

取 $\text{mid}=\lfloor\frac{l+r}{2}\rfloor$，则有

$$A(l,r)=A(l,\text{mid})\times B(\text{mid}+1,r)+A(\text{mid}+1,r)\times B(l,\text{mid})$$
$$B(l,r)=B(l,\text{mid})\times B(\text{mid}+1,r)$$

通过分治求出 $A(1,n),B(1,n)$，最后求逆一次即可。

时间复杂度 $\mathcal{O}(n\log^2 n)$。

分治 NTT 模型 3：给出 n 个数 $a_1,\dots,a_n$，求出 $\sum_{i=1}^n\prod_{j\neq i}(a_jx+1)$。

设 $f(l,r)=\sum_{i=l}^r\prod_{j\in[l,r],j\neq i}(a_jx+1)$，$g(l,r)=\prod_{i=l}^r(a_ix+1)$，$f(l,r),g(l,r)$ 的次数同样与区间长度有关。

取 $\text{mid}=\lfloor\frac{l+r}{2}\rfloor$，则有

$$f(l,r)=f(l,\text{mid})\times g(\text{mid}+1,r)+f(\text{mid}+1,r)\times g(l,\text{mid})$$
$$g(l,r)=g(l,\text{mid})\times g(\text{mid}+1,r)$$

时间复杂度 $\mathcal{O}(n\log^2 n)$。

分治 NTT 模型 4：若递归到分治树的节点 u 时，只需要保留多项式中的 $\mathcal{O}(s_u)$ 个有效项（其余的项可以特殊处理从而无需继续递归），并且节点 u 上需要进行规模为 $\mathcal{O}(s_u)$ 的卷积。

记整棵分治树所有节点的 s_u 之和等于 m，则时间复杂度 $\mathcal{O}(m\log m)$。

分治 NTT 模型 5（djq 分治）：Q 次询问组合数前缀和。每次询问给出 n,m，求 $\sum_{i=0}^{m}\binom{n}{i}\bmod 998244353$。

注意到 $\binom{n}{m}=\frac{n\times(n-1)\times\cdots\times(n-m+1)}{m!}$，可以将 $\binom{n}{m}$ 看成一个关于 n 的 m 次多项式。

记多项式 $F_i(x)=\frac{x-i+1}{i}$（特别地 $F_0(x)=1$），则单次查询的答案为

$$\sum_{i=0}^{m}\prod_{j=0}^{i}F_j(n)$$

记 $M=\max\{m\}$，对区间 $[0,M+1]$ 建立线段树。将询问 n,m 挂在叶子节点 $[m+1,m+1]$ 上。

对于线段树上表示区间 $[l,r]$ 的节点 u，预处理出：

- $\prod_{i=l}^{r}F_i(x)$。
- $\sum_{i=l}^{r}\prod_{j=l}^{i}F_j(x)$。

在线段树上递归，假设当前递归到了表示区间 $[l,r]$ 的节点 u，维护当前的 $\prod_{i=0}^{l-1}F_i(x)$ 与 $\sum_{i=0}^{l-1}\prod_{j=0}^{i}F_j(x)$。在递归的过程中，两个多项式的次数太大，无法承受。

考虑运用多项式多点求值的经典套路 $f(x)\equiv f(n)\pmod{x-n}$。

递归到节点 u 的时候，将两个多项式均对 $\prod_{m+1\in[l,r]}(x-n)$ 取模（也就是对区间内所有询问的 $x-n$ 之积取模，同样在线段树上预处理出，注意叶子可能有多个询问，乘积需要单独计算），此时两个多项式的次数均不超过区间内询问个数。

最后递归到叶子时，由于叶子可能有多个询问，对当前多项式进行多点求值即可。

时间复杂度 $\mathcal{O}(M\log^2 M+Q\log^2 Q)$，该做法的优点是时间复杂度与 n 无关。

> 还有一种基于莫队的做法。记 $S_{n,m}=\sum_{i=0}^{m}\binom{n}{i}$，则有
>
> $$S_{n+1,m}=2S_{n,m}-\binom{n}{m}$$
> $$S_{n,m+1}=S_{n,m}+\binom{n}{m+1}$$

Model：半在线卷积

半在线卷积：给出 $g_1,\dots,g_n$，求出 $f_0,\dots,f_n$，其中 $f_0=1$，$f_i=\sum_{j=1}^{i}f_{i-j}g_j$。

使用**按中序遍历转移的 CDQ 分治**：设 $\text{solve}(l,r)$ 表示求出区间 $[l,r]$ 中所有的 f 值，取 $\text{mid}=\lfloor\frac{l+r}{2}\rfloor$。

- 先递归 $\text{solve}(l,\text{mid})$，求出 $f_l,\dots,f_{\text{mid}}$。
- **计算 $f_{[l,\text{mid}]}$ 与 $g_{[0,r-l]}$ 的卷积结果，求出左半区间对右半区间的贡献。**
- 后递归 $\text{solve}(\text{mid}+1,r)$，求出 $f_{\text{mid}+1},\dots,f_r$。
- 递归到叶子 $l=r$ 时，请注意根据具体递推式调整 f_i（e.g. $f_i=g_i-\sum_{j=1}^{i-1}\binom{i-1}{j}f_{i-j}g_j$）。

时间复杂度 $T(n)=2T\left(\frac{n}{2}\right)+\mathcal{O}(n\log n)=\mathcal{O}(n\log^2 n)$。

> 卡常技巧：计算 $f_{[l,\text{mid}]}$ 与 $g_{[0,r-l]}$ 的卷积结果时，**只需将补充界设定成 $\geq\text{len}$ 即可（而非 $\geq\frac{3\cdot\text{len}}{2}$）**。
>
> 因为循环卷积只会污染 $0\sim\frac{\text{len}}{2}$ 次项，但我们只需要 $\frac{\text{len}}{2}\sim\text{len}$ 次项。

全在线卷积：求出 $f_0,\dots,f_n$，其中 $f_0=1$，$f_i=\sum_{j=1}^{i}f_{i-j}g_j$，但是 g_i 与 $f_0,\dots,f_{i-1}$ 强相关。

仍然使用按中序遍历转移的 CDQ 分治，但 $g_{[0,r-l]}$ 我们不一定提前知道。进一步分析分治树的结构，**发现仅有 $l=0$ 时不知道 $g_{[0,r-l]}$**。所以我们需要修改计算贡献的方式。

- 当 $l=0$ 时，**计算 $f_{[0,\text{mid}]}\times g_{[0,\text{mid}]}$ 的贡献**（此时还有 $f_{[0,\text{mid}]}\times g_{[\text{mid}+1,r]}$ 的贡献没被计算）。
- 当 $l>0$ 时，**计算 $f_{[l,\text{mid}]}\times g_{[0,r-l]}+g_{[l,\text{mid}]}\times f_{[0,r-l]}$ 的贡献**（后半部分即为 $l=0$ 中漏掉的贡献）。
- 递归到叶子 $l=r$ 时，先根据 $f_0,\dots,f_{l-1}$ 求出 g_l，再**计算 $f_0\times g_l$ 的贡献**。

可以证明，点对的贡献不重不漏。

时间复杂度 $\mathcal{O}(n\log^2 n)$。

Model：树上路径长度统计

树上路径长度统计：有一棵包含 n 个点的树，对于每个 k $(1 \le k < n)$ 求出有多少条路径的长度为 k。

对树进行点分治，设当前的分治重心为 c，考虑统计经过当前分治重心的路径贡献。设删除分治重心 c 之后，剩下的连通块内的点到 c 的距离分布多项式分别为 $b_1(x), \dots, b_m(x)$，其中 $[x^k]b_i(x)$ 表示第 i 个连通块有多少个距离 c 为 k 的点。

则分治重心 c 的贡献多项式为

$$\sum_{i=1}^{m} b_i(x) \left(\sum_{j=1}^{i-1} b_j(x) + 1 \right)$$

将多项式 $b_1(x), \dots, b_m(x)$ 按照次数从小到大排序后，维护一个前缀多项式之和，计算上式即可。

时间复杂度 $\mathcal{O}(n \log^2 n)$。

> 还有一种基于长链剖分的做法，时间复杂度 $\mathcal{O}(n \log n)$，详见 zak 的博客。

多项式基本运算

多项式除法（暴力）

多项式除法：给出两个多项式 $A(x), B(x)$，求出多项式 $C(x)$，满足 $B(x)C(x) = A(x)$。

设 $A(x) = \sum_{i=0}^{n} a_i x^i, B(x) = \sum_{i=0}^{m} b_i x^i, C(x) = \sum_{i=0}^{n-m} c_i x^i$。先对 b 的前导零进行处理，设 b 的前导零有 t 位，则需要使得 a, b 左移 t 位。然后直接根据定义 $\sum_{j=0}^{i} c_j b_{i-j} = a_i$ 暴力展开。

初值

$$c_0 = \frac{a_0}{b_0}$$

递推式

$$c_i = \frac{1}{b_0} \left(a_i - \sum_{j=0}^{i-1} c_j b_{i-j} \right)$$

但递推式里的 j 只需要枚举到 $i - j \le m$ 即 $j \ge i - m$ 就可以了。

时间复杂度 $\mathcal{O}(nm)$。实际上是 $(n-m)m$。

多项式导数/积分

多项式形式导数：

$$f'(x) = \left(\sum_{i=0}^{n} a_i x^i \right)' = \sum_{i=1}^{n} i a_i x^{i-1}$$

多项式形式积分：

$$\int f(x) \mathrm{dx} = \int \left(\sum_{i=0}^{n} a_i x^i \right) \mathrm{dx} = \sum_{i=0}^{n} \frac{1}{i+1} a_i x^{i+1}$$

多项式牛顿迭代

多项式牛顿迭代：已知多项式函数 g，求出满足 $g(h(x)) = 0$ 的多项式 h，一般我们只需要求出 $h \bmod x^n$。

（在这里，我们将 $f(x)$ 看成函数 $g(f(x))$ 的自变量，则我们要求的就是 $g(f(x)) = 0$ 模 x^n 意义下的零点）。

当 $n = 1$ 时，$[x^0]h(x)$ 需要单独求出。

当 $n > 1$ 时，记 $h_0 = h \bmod x^{\lceil \frac{n}{2} \rceil}, h_1 = h \bmod x^n$，假设现在已经求出了 h_0，希望通过 h_0 推出 h_1。考虑将函数 $g(f(x))$ 在 $f(x) = h_0(x)$ 处泰勒展开。

$$g(f(x)) = \sum_{i=0}^{\infty} \frac{g^{(i)}(h_0(x))}{i!} (f(x) - h_0(x))^i$$

将 $h_1(x)$ 代入自变量 $f(x)$，观察到 $h_1(x) - h_0(x)$ 的前 $\lceil \frac{n}{2} \rceil$ 项均为 0，所以二次及以上的项均可忽略。进一步可得

$$g(h_0(x)) + g'(h_0(x)) (h_1(x) - h_0(x)) \equiv 0 \pmod{x^n}$$

移项可得

$$h_1(x) \equiv h_0(x) - \frac{g(h_0(x))}{g'(h_0(x))} \pmod{x^n}$$

时间复杂度 $T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$。

多项式求逆

多项式逆：给出一个多项式 $A(x)$，求出多项式 $B(x)$，满足 $A(x)B(x) = 1$。一般只需要求出 $B \bmod x^m$。

多项式逆存在条件：多项式 $A(x)$ 存在逆，当且仅当 $[x^0]A(x) \neq 0$。

（注意：一个多项式的逆可能是有无穷项的，例如 $\frac{1}{1-x} = 1 + x + x^2 + \dots$）

多项式求逆（牛顿迭代）：

构造函数 $g(f(x))=\frac{1}{f(x)}-A(x)$，代入牛顿迭代法得

$$h_1(x)\equiv h_0(x)-\frac{\frac{1}{h_0(x)}-A(x)}{-\frac{1}{h_0^2(x)}}\pmod{x^m}$$

$$h_1(x)\equiv 2h_0(x)-A(x)h_0^2(x)\pmod{x^m}$$

初始时，$A^{-1}(x)$ 的常数项即为 $A(x)$ 常数项的逆元。

时间复杂度 $\mathcal{O}(m\log m)$。

多项式开方

多项式开方：给出一个多项式 $A(x)$，求出多项式 $B(x)$，满足 $B^2(x)=A(x)$。一般只需要求出 $B \bmod x^m$。

多项式开方存在条件：当 $[x^0]A(x)\neq 0$ 时，多项式 $A(x)$ 存在开方，当且仅当 $[x^0]A(x)$ 是模意义下的二次剩余。

构造函数 $g(f(x))=f^2(x)-A(x)$，代入牛顿迭代法得

$$h_1(x)\equiv h_0(x)-\frac{h_0^2(x)-A(x)}{2h_0(x)}\pmod{x^m}$$

$$h_1(x)\equiv \frac{h_0^2(x)+A(x)}{2h_0(x)}\pmod{x^m}$$

初始时，$\sqrt{A(x)}$ 的常数项即为 $A(x)$ 常数项模意义下的平方根。当 $A(x)$ 常数项不等于 1 时，还需要通过二次剩余求解常数项。

时间复杂度 $\mathcal{O}(m\log m)$。

> By the way：注意到 $\sqrt{A(x)}=e^{\frac{1}{2}\ln A(x)}$，若 $[x^0]A(x)=1$，可以直接套用多项式 ln/exp。

多项式带余除法

多项式带余除法：给出一个 n 次多项式 $A(x)$ 和一个 m 次多项式 $B(x)$，求出多项式 $Q(x),R(x)$ 满足：

- $Q(x)$ 的次数为 $n-m$，$R(x)$ 的次数 $<m$。
- $A(x)=Q(x)\times B(x)+R(x)$。

考虑构造变换 $f^R(x)=x^{\deg f}f\left(\frac{1}{x}\right)$，实际上即为翻转 f 的系数。

将 $A(x)=Q(x)\times B(x)+R(x)$ 中的 x 替换为 $\frac{1}{x}$，并且等式两边同乘以 x^n

$$x^nA\left(\frac{1}{x}\right)=x^{n-m}Q\left(\frac{1}{x}\right)\times x^mB\left(\frac{1}{x}\right)+x^{n-m+1}x^{m-1}R\left(\frac{1}{x}\right)$$

$$A^R(x)=Q^R(x)\times B^R(x)+x^{n-m+1}x^{m-1}R\left(\frac{1}{x}\right)$$

$$A^R(x)\equiv Q^R(x)\times B^R(x)\pmod{x^{n-m+1}}$$

由于 $Q(x)$ 的次数小于 $n-m+1$，故对 x^{n-m+1} 取模没有影响。

故在模 x^{n-m+1} 意义下求出 $\frac{A^R(x)}{B^R(x)}$ 再翻转一遍即可求出 $Q(x)$，再使用 $A(x)-Q(x)\times B(x)$ 即可求出 $R(x)$。

时间复杂度 $\mathcal{O}(n\log n)$。

多项式 ln/exp

> e 在模 998244353 下没有意义。同理，$\ln 2$ 在模 998244353 下没有意义。
>
> 简单的解释：考察 $e=\sum_{i=0}^{\infty}\frac{1}{i!}$，在 $i\geq 998244353$ 时，分式就已经失去了意义。即使有意义，无穷项求和在模 998244353 下也不会收敛。

多项式 ln 与多项式 exp 的定义：由麦克劳林级数定义

$$\ln f(x)=\sum_{i=1}^{\infty}(-1)^{i+1}\frac{(f(x)-1)^i}{i}$$

$$=-\sum_{i=1}^{\infty}\frac{(1-f(x))^i}{i}$$

$$\exp f(x)=\sum_{i=0}^{\infty}\frac{f^i(x)}{i!}$$

多项式 ln 存在条件：多项式 $f(x)$ 存在 **ln**，当且仅当 $[x^0]f(x)=1$，并且由此可得 $[x^0]\ln f(x)=0$。

多项式 ln：给出一个多项式 $A(x)$，求出多项式 $B(x)=\ln A(x)$。一般只需要求出 $B \bmod x^m$。

将 $B(x)=\ln A(x)$ 等式两边对 x 求导可得

$$B'(x)=\frac{A'(x)}{A(x)}$$

两边同时积分可得

$$B(x)=\int\frac{A'(x)}{A(x)}\mathrm{dx}$$

注意：虽然多项式先求导后积分会丢掉常数项的信息，但是容易验证当 $A(x)$ 的常数项为 1 时，$B(x)$ 的常数项必定为 0。

时间复杂度 $\mathcal{O}(m\log m)$。

多项式 exp 存在条件：多项式 $f(x)$ 存在 **exp**，当且仅当 $[x^0]f(x)=0$，并且可以推出 $[x^0]\exp f(x)=1$。

多项式 exp：给出一个多项式 $A(x)$，求出多项式 $B(x)=\exp A(x)$。一般只需要求出 $B \bmod x^m$。

构造函数 $g(f(x))=\ln f(x)-A(x)$，代入牛顿迭代得

$$h_1(x)\equiv h_0(x)-\frac{\ln h_0(x)-A(x)}{\frac{1}{h_0(x)}}\pmod{x^m}$$

$$h_1(x)\equiv h_0(x)\left(1-\ln h_0(x)+A(x)\right)\pmod{x^m}$$

初始时，$\exp A(x)$ 的常数项必为 1。

时间复杂度 $\mathcal{O}(m\log m)$。

多项式快速幂

多项式快速幂：给出一个多项式 $A(x)$，你需要求出 $A^k(x) \bmod x^m$。

多项式快速幂（倍增法）：具体算法流程同快速幂。

时间复杂度 $\mathcal{O}(m\log m\log k)$，但可以处理的问题形式更加灵活。

> e.g. 多项式快速幂对另一多项式取模。
> e.g. 多项式快速幂且保留次数在 $[-m,m]$ 的项。

多项式快速幂（先 ln 后 exp）：注意到 $A^k(x)=e^{k\ln A(x)}$，考虑直接套用多项式 ln 与多项式 exp 的模板。

但由于多项式 ln 需要保证 $[x^0]A(x)=1$，首先要对 $A(x)$ 进行一些操作。

找出一个最小的 i，满足 $[x^i]A(x)\neq 0$（相当于是找出 $A(x)$ 的第一个非零系数），记 $v=[x^i]A(x)$。则有

$$A(x)=\left(\frac{A(x)}{x^iv}\right)x^iv$$

注意到 $\frac{A(x)}{x^iv}$ 满足常数项为 1 的限制，可以进行多项式 ln。

然后有

$$A^k(x)=\left(\frac{A(x)}{x^iv}\right)^k x^{ik}v^k$$

先使用多项式 ln 与多项式 exp 求出 $\left(\frac{A(x)}{x^iv}\right)^k \bmod x^{m-ik}$ 的值，最后再乘上 $x^{ik}v^k$ 即可。（注意：这里的乘 x^{ik} 相当于多项式系数向右移动 ik 位。上面的除以 x^i 同理，相当于多项式系数向左移动 i 位，即舍弃掉前 i 项系数）

时间复杂度 $\mathcal{O}(m\log m)$。

> 设模数为质数 P。当 $k\le P$ 时，可以放心应用该做法；当 $k>P$ 时，需要进一步讨论 k 的取模相关问题。
> 由 $(a+b)^P\equiv a^P+b^P\pmod P$ 以及费马小定理可得 $f^P(x)\equiv f(x^P)\pmod P$。
> 在一般情况下 $P>m$，所以 $f^P(x)\equiv f(x^P)\equiv[x^0]f(x)\pmod{x^m}$，由于我们进行了一系列操作使得 $[x^0]f(x)=1$，所以 $f^P(x)\equiv 1\pmod{x^m}$。于是：
> - 多项式 **ln/exp** 部分：k 需要对 P 取模。
> - v^k 部分：k 需要对 $\varphi(P)=P-1$ 取模（因为是数值计算）。
> - x^{ik} 部分：用原始的 k 判断，当 $i>0$ 且 $k\ge m$ 时答案必定为 0。

少项式快速幂

少项式快速幂：给出一个次数不高的多项式 $A(x)$，求出 $A^k(x) \bmod x^m$。

该做法需要满足 $[x^0]A(x)\neq 0$，可以对前导 0 特殊处理。

设 $A(x)=\sum_{i=0}^{n}a_ix^i, A^k(x)=\sum_{i=0}^{m-1}b_ix^i$。考虑对 $A^k(x)$ 求导

$$A^k(x)'=kA^{k-1}(x)A'(x)$$

$$A^k(x)'A(x)=kA^k(x)A'(x)$$

注意到 $A^k(x)$ 与 $A^k(x)'$ 有一定的递推关系。具体地，考虑等式左右两边的 i 次项

$$\sum_{j=0}^{i}(j+1)b_{j+1}a_{i-j}=k\cdot\sum_{j=0}^{i}(i-j+1)a_{i-j+1}b_j$$

等式左边只需枚举到 $j\ge i-n$，等式右边只需枚举到 $j\ge i-n+1$。可以解得

$$b_{i+1}=\frac{\left(k\cdot\sum\limits_{j=i-n+1}^{i}(i-j+1)a_{i-j+1}b_j\right)-\left(\sum\limits_{j=i-n}^{i-1}(j+1)b_{j+1}a_{i-j}\right)}{(i+1)a_0}$$

初值 $b_0 = a_0^k$，上式可以在已知 $b_0, b_1, \dots, b_i$ 的情况下递推出 b_{i+1}。
需要线性预处理出 $1, 2, \dots, m-1$ 的逆元，递推式：$i^{-1} \equiv -\lfloor \frac{p}{i} \rfloor (p \bmod i)^{-1} \pmod p$。
时间复杂度 $\mathcal{O}(nm)$。当次数 n 较小时有奇效。

> By the way：从这个递推式进行归纳可知，当 $a_0 = 1$ 时，$[x^n]A^k(x)$ 是一个关于 k 的至多 n 次多项式。

转置原理

~~待填坑。~~

多项式多点求值

多项式多点求值：给出一个多项式 $f(x)$ 和 n 个点 $x_1, \dots, x_n$，求 $f(x_1), \dots, f(x_n)$。
多项式多点求值（多项式取模）：
设 $\text{mid} = \lfloor \frac{1+n}{2} \rfloor$，将给定的点集分成两部分

$$X_1 = \{x_1, \dots, x_{\text{mid}}\}, \quad X_2 = \{x_{\text{mid}+1}, \dots, x_n\}$$

构造多项式

$$g_1(x) = \prod_{x_i \in X_1} (x - x_i), \quad g_2(x) = \prod_{x_i \in X_2} (x - x_i)$$

将 $f(x)$ 对 $g_1(x)$ 取模，记 $f(x) = Q_1(x) \times g_1(x) + R_1(x)$。对于 X_1 中的元素 x_i，由于 $g_1(x_i) = 0$，则有 $f(x_i) = R_1(x_i)$。又由于 $R_1(x)$ 的次数小于区间长度的一半，于是问题规模成功减半，分治求解问题即可。对于 X_2 中的元素也是同理。
多项式 g 可以通过分治 NTT 预处理。
时间复杂度 $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n \log n) = \mathcal{O}(n \log^2 n)$。

> 另一种理解角度：$f(x) \equiv f(x_i) \pmod{x - x_i}$。证明只需将 $x^n = ((x - x_i) + x_i)^n$ 二项式定理展开即可。

0x31 多项式多点求值（多项式取模）.cpp

```cpp
// 多项式多点求值（多项式取模做法），作为 poly 的成员函数
std::vector<int> eval(std::vector<int> x) const {
    if (empty()) return std::vector<int>(x.size());
    if (x.empty()) return {};

    int n = x.size();
    std::vector<poly> t(n * 4);
    std::vector<int> ans(n);

    std::function<void(int, int, int)> build = [&] (int p, int l, int r) {
        if (l == r) {
            t[p] = poly{norm(-x[l]), 1};
            return;
        }
        int mid = (l + r) >> 1;
        build(p * 2, l, mid), build(p * 2 + 1, mid + 1, r);
        t[p] = t[p * 2] * t[p * 2 + 1];
    };
    build(1, 0, n - 1);

    std::function<void(int, int, int, const poly &)> solve = [&] (int p, int
     ↪ l, int r, const poly &f) {
        if (l == r) {
            ans[l] = f[0];
            return;
        }
        int mid = (l + r) >> 1;
        solve(p * 2, l, mid, divmod(f, t[p * 2]).second);
        solve(p * 2 + 1, mid + 1, r, divmod(f, t[p * 2 + 1]).second);
    };
    solve(1, 0, n - 1, divmod(*this, t[1]).second);

    return ans;
}
```

多项式多点求值（转置原理）：

~~待填坑。~~

多项式快速插值

多项式快速插值：给出 n 个点 $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$，其中横坐标（模意义下）两两不同。快速地求一个至多为 $n-1$ 次多项式，使得这 n 个点都在 $f(x)$ 上。
考虑拉格朗日插值

$$f(x) = \sum_{i=1}^{n} y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}$$

记 $g(x)=\prod_{j=1}^{n}(x-x_j)$，则

$$\prod_{j\neq i}(x_i-x_j)=\lim_{x\to x_i}\frac{\prod_{j=1}^{n}(x-x_j)}{x-x_i}=g'(x_i)$$

首先使用分治 NTT 求出 $g'(x)$，再使用多项式多点求值，求出所有的 $g'(x_i)$。

记 $v_i=\frac{y_i}{g'(x_i)}$，则原式化为

$$f(x)=\sum_{i=1}^{n}v_i\prod_{j\neq i}(x-x_j)$$

仍然使用分治 NTT 求出上式。具体地，设 $\text{mid}=\lfloor\frac{1+n}{2}\rfloor$，设多项式

$$f_1(x)=\sum_{i=1}^{\text{mid}}v_i\left(\prod_{j\neq i,j\leq\text{mid}}(x-x_j)\right),\quad g_1(x)=\prod_{j\leq\text{mid}}(x-x_j)$$

$$f_2(x)=\sum_{i=\text{mid}+1}^{n}v_i\left(\prod_{j\neq i,j>\text{mid}}(x-x_j)\right),\quad g_2(x)=\prod_{j>\text{mid}}(x-x_j)$$

显然有 $f(x)=f_1(x)g_2(x)+f_2(x)g_1(x)$。$f_1,f_2,g_1,g_2$ 可以进一步分治求解。

时间复杂度 $\mathcal{O}(n\log^2 n)$。

多项式平移

多项式平移：给出一个多项式 $A(x)$ 以及一个常数 c，求出多项式 $A(x+c)$。

多项式平移（二项式定理法）：设 $A(x)=\sum_{i=0}^{n}a_ix^i$，考虑二项式定理

$$\begin{aligned}A(x+c)&=\sum_{i=0}^{n}a_i(x+c)^i\\&=\sum_{i=0}^{n}a_i\sum_{j=0}^{i}\binom{i}{j}x^jc^{i-j}\\&=\sum_{i=0}^{n}a_i\sum_{j=0}^{i}\frac{i!}{j!(i-j)!}x^jc^{i-j}\\&=\sum_{j=0}^{n}\frac{x^j}{j!}\cdot\left(\sum_{i=j}^{n}a_i\cdot i!\cdot\frac{c^{i-j}}{(i-j)!}\right)\end{aligned}$$

注意到括号里的求和式，下标之差恒为 j，于是可以使用“翻转卷积”的技巧求出每个关于 j 的求和式。

时间复杂度 $\mathcal{O}(n\log n)$。

连续点值平移：给出 $n+1$ 个连续点值 $f(0),f(1),\dots,f(n)$（其中 f 是一个次数至多为 n 的多项式）以及一个常数 c，求出 $f(c),f(c+1),\dots,f(c+n)$。

代入拉格朗日插值公式，得

$$\begin{aligned}f(x)&=\sum_{i=0}^{n}y_i\prod_{j\neq i}\frac{x-j}{i-j}\\&=\sum_{i=0}^{n}y_i\cdot\frac{\prod_{v=x-n}^{x}v}{x-i}\cdot\frac{(-1)^{n-i}}{i!(n-i)!}\\&=\left(\prod_{v=x-n}^{x}v\right)\sum_{i=0}^{n}\frac{1}{x-i}\cdot\frac{y_i(-1)^{n-i}}{i!(n-i)!}\end{aligned}$$

于是

$$f(c+x)=\left(\prod_{v=c+x-n}^{c+x}v\right)\sum_{i=0}^{n}\frac{1}{c+x-i}\cdot\frac{y_i(-1)^{n-i}}{i!(n-i)!}$$

注意到右边的求和式，是一个卷积的形式。但是 $\frac{1}{c+x-i}$ 的分母在模意义下可能为 0，此时 $c+x\equiv i\pmod p$，于是 $f(c+x)\equiv f(i)\equiv y_i\pmod p$，所以此时该点值不需要额外计算，直接取已有的值 y_i 即可。

记：

- $a_i=\begin{cases}\frac{1}{c-n+i} & c-n+i\not\equiv 0\pmod p\\ 0 & c-n+i\equiv 0\pmod p\end{cases}$（其中 $i=0\sim 2n$，因为分母的取值范围为 $[c-n,c+n]$）。
- $b_i=\frac{y_i(-1)^{n-i}}{i!(n-i)!}$。

则右边的求和式为 $\sum_{i=0}^{n}a_{x-i+n}b_i$。将 a,b 进行卷积得到 res。此时，若 $(c+x)\bmod p\in[0,n]$，则

$$f(c+x)=y_{(c+x)\bmod p}$$

否则

$$f(c+x)=\left(\prod_{v=c+x-n}^{c+x}v\right)\text{res}_{x+n}$$

时间复杂度 $\mathcal{O}(n\log n)$。

> 给出 $n+1$ 个等差点值 $f(b), f(b+k), \dots, f(b+nk)$（其中 f 是一个次数至多为 n 的多项式）以及一个常数 c，求出 $f(c+b), f(c+b+k), \dots, f(c+b+nk)$。
>
> 设 $g(x) = f(b+xk)$，可以看成 $g(0), g(1), \dots, g(n)$ 向右平移 $\frac{c}{k}$ 得到 $g(\frac{c}{k}), g(\frac{c}{k}+1), \dots, g(\frac{c}{k}+n)$。

Chirp Z-Transform

Chirp Z-Transform：给出一个 n 次多项式 $f(x)$ 以及整数 c, m，求出 $f(c^0), f(c^1), \dots, f(c^{m-1})$。

利用 $ij = \binom{i+j}{2} - \binom{i}{2} - \binom{j}{2}$，有

$$\begin{aligned} f(c^i) &= \sum_{j=0}^{n} a_j c^{ij} \\ &= \sum_{j=0}^{n} a_j c^{\binom{i+j}{2} - \binom{i}{2} - \binom{j}{2}} \\ &= c^{-\binom{i}{2}} \sum_{j=0}^{n} a_j c^{-\binom{j}{2}} c^{\binom{i+j}{2}} \end{aligned}$$

注意到求和式的下标之差恒为 i，于是可以使用“翻转卷积”的技巧求出每个关于 i 的求和式。

多项式全家桶

reference：jiangly 算法模板。

封装理念：

- 构建结构体 **poly** 继承自 **std::vector<int>**，自由度高且易于访问。
- 涉及“当前多项式的变换或求值”的功能均作为 poly 的内置函数；其余功能作为 poly 的外置函数。

0x31 多项式全家桶.cpp

```cpp
std::vector<int> roots{0, 1};
void dft(std::vector<int> &a) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i ++) {
        for (int k = n >> 1; (j ^= k) < k; k >>= 1);
        if (i < j) std::swap(a[i], a[j]);
    }
    for (int k = roots.size(); k < n; k <<= 1) {
        int w = qpow(3, (mod - 1) / (k << 1), mod); // 原根根据实际模数调整
        roots.resize(k << 1);
        for (int i = k >> 1; i < k; i ++) {
            roots[i * 2] = roots[i];
            roots[i * 2 + 1] = 1ll * roots[i] * w % mod;
        }
    }
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < n; i += (k << 1)) {
            for (int j = 0; j < k; j ++) {
                int u = a[i + j], v = 1ll * a[i + j + k] * roots[k + j] % mod;
                add(a[i + j] = u, v), dec(a[i + j + k] = u, v);
            }
        }
    }
}
void idft(std::vector<int> &a) {
    int n = a.size(), inv = (1 - mod) / n + mod;
    std::reverse(a.begin() + 1, a.end());
    dft(a);
    for (int i = 0; i < n; i ++) {
        mul(a[i], inv);
    }
}

// 多项式全家桶（2026-11-05）
struct poly : public std::vector<int> {
    using std::vector<int>::vector;
    explicit poly(const std::vector<int> &a) : std::vector<int>(a) {}

    // 多项式乘法
    friend poly operator * (poly a, poly b) {
        int tot = a.size() + b.size() - 1;
        if (tot < 128) {
            poly c(tot);
            for (int i = 0; i < a.size(); i ++) {
                for (int j = 0; j < b.size(); j ++) {
                    c[i + j] = (c[i + j] + 1ll * a[i] * b[j]) % mod;
                }
            }
            return c;
        }
        int L = 1;
        while (L < tot) L <<= 1;
```

```
        a.resize(L), b.resize(L);
        dft(a), dft(b);
        for (int i = 0; i < L; i ++) {
            mul(a[i], b[i]);
        }
        idft(a);
        a.resize(tot);
        return a;
    }

    // 多项式乘法（暴力）
    friend poly mul_bf(poly a, poly b) {
        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] = (c[i + j] + 1ll * a[i] * b[j]) % mod;
            }
        }
        return c;
    }
    // 多项式除法（暴力）
    friend poly div_bf(poly a, poly b) {
        int t = 0;
        while (t < b.size() && b[t] == 0) t ++;
        if (t == b.size()) return poly(a.size() - b.size() + 1); // 除以全 0
         ↪ 多项式无意义
        a = poly(a.begin() + t, a.end());
        b = poly(b.begin() + t, b.end());
        int m = b.size() - 1, inv = qpow(b[0], mod - 2, mod);
        poly c(a.size() - b.size() + 1);
        for (int i = 0; i < c.size(); i ++) {
            c[i] = a[i];
            for (int j = std::max(0, i - m); j < i; j ++) {
                dec(c[i], 1ll * c[j] * b[i - j] % mod);
            }
            mul(c[i], inv);
        }
        return c;
    }

    friend poly operator + (poly a, poly b) {
        poly c(std::max(a.size(), b.size()));
        for (int i = 0; i < a.size(); i ++) add(c[i], a[i]);
        for (int i = 0; i < b.size(); i ++) add(c[i], b[i]);
        return c;
    }

    friend poly operator - (poly a, poly b) {
        poly c(std::max(a.size(), b.size()));
        for (int i = 0; i < a.size(); i ++) add(c[i], a[i]);
        for (int i = 0; i < b.size(); i ++) dec(c[i], b[i]);
        return c;
    }
    friend poly operator - (poly a) {
        for (int &x : a) neg(x);
        return a;
    }

    friend poly operator * (poly a, int b) {
        for (int &x : a) mul(x, b);
        return a;
    }
    friend poly operator / (poly a, int b) {
        int inv = qpow(b, mod - 2, mod);
        for (int &x : a) mul(x, inv);
        return a;
    }

    poly &operator += (poly b) {
        return *this = *this + b;
    }
    poly &operator -= (poly b) {
        return *this = *this - b;
    }
    poly &operator *= (poly b) {
        return *this = *this * b;
    }
    poly &operator *= (int b) {
        return *this = *this * b;
    }
    poly &operator /= (int b) {
        return *this = *this / b;
    }

    poly mulxk(int k) const {
        auto res = *this;
        res.insert(res.begin(), k, 0);
        return res;
    }
    poly divxk(int k) const {
        if (k >= size()) return poly();
        return poly(begin() + k, end());
    }
```

```
    poly modxk(int k) const {
        if (k >= size()) return *this;
        return poly(begin(), begin() + k);
    }
    poly cirxk(int k) const {
        auto res = *this;
        for (int i = k; i < size(); i ++) {
            add(res[i % k], res[i]); // 若多项式次数 <2k，可改成 res[i - k]
        }
        return res.modxk(k);
    }

    poly flip() const {
        auto res = *this;
        std::reverse(res.begin(), res.end());
        return res;
    }

    poly mulT(poly b) const {
        if (b.empty()) return poly();
        std::reverse(b.begin(), b.end());
        return ((*this) * b).divxk(b.size() - 1);
        // 等价于“翻转卷积”，a.mulT(b) 相当于是求 a[i] * b[j] -> c[i - j]
    }

    // 多项式导数
    poly deriv() const {
        if (empty()) return poly();
        poly res(size() - 1);
        for (int i = 1; i < size(); i ++) {
            res[i - 1] = 1ll * (*this)[i] * i % mod;
        }
        return res;
    }

    // 多项式积分
    poly integ() const {
        poly res(size() + 1);
        for (int i = 0; i < size(); i ++) {
            res[i + 1] = 1ll * (*this)[i] * qpow(i + 1, mod - 2, mod) % mod;
        }
        return res;
    }

    // 多项式求逆
    poly inv(int m) const { // 需要保证常数项不为 0
        poly x{qpow(front(), mod - 2, mod)};
        int k = 1;
        while (k < m) {
            k <<= 1;
            x = (x * (poly{2} - modxk(k) * x)).modxk(k);
        }
        return x.modxk(m);
    }

    // 多项式开方
    poly sqrt(int m) const {
        poly x{1}; // 默认常数项为 1
        int k = 1;
        while (k < m) {
            k <<= 1;
            x = (x + (modxk(k) * x.inv(k)).modxk(k)) / 2;
        }
        return x.modxk(m);
    }

    // 多项式带余除法
    friend std::pair<poly, poly> divmod(poly a, poly b) {
        int n = a.size(), m = b.size(), L = n - m + 1;
        if (n < m) return {poly{0}, a};
        poly q = (a.flip().modxk(L) * b.flip().inv(L)).modxk(L).flip();
        poly r = a - b * q;
        while (r.size() > 1 && !r.back()) {
            r.pop_back();
        }
        return {q, r};
    }

    // 多项式 ln
    poly ln(int m) const { // 需要保证常数项为 1
        return (deriv() * inv(m)).integ().modxk(m);
    }

    // 多项式 exp
    poly exp(int m) const { // 需要保证常数项为 0
        poly x{1};
        int k = 1;
        while (k < m) {
            k <<= 1;
            x = (x * (poly{1} - x.ln(k) + modxk(k))).modxk(k);
        }
        return x.modxk(m);
```

```
    }

    // 多项式快速幂
    poly pow(int k, int m) const {
        int i = 0;
        while (i < size() && (*this)[i] == 0) i ++;
        if (i == size() || 1ll * i * k >= m) return poly(m);
        int v = (*this)[i];
        poly f = divxk(i) / v;
        return (f.ln(m - i * k) * k).exp(m - i * k).mulxk(i * k) * qpow(v,
         ↪ k, mod);
        /*
            当实际 k < mod 时，可以放心使用
            当实际 k >= mod 时，请注意：
            - 当 i > 0 时，直接返回 poly(m)
            - 当 i = 0 时
              - 在多项式 ln 与多项式 exp 部分，k 需要对 mod 取模
              - 在 qpow(v, k, mod) 部分，k 需要对 mod - 1 取模
        */
    }
    // 少项式快速幂
    poly pow_bf(int k, int m) const { // 需要保证常数项不为 0
        std::vector<int> v(m); // 需要保证 m > 1
        v[1] = 1;
        for (int i = 2; i < m; i ++) {
            v[i] = 1ll * v[mod % i] * (mod - mod / i) % mod;
        }

        const poly a = *this;
        int n = a.size() - 1, inv = qpow(a[0], mod - 2, mod);
        poly b(m);
        b[0] = qpow(a[0], k, mod);
        for (int i = 0; i + 1 < m; i ++) {
            for (int j = std::max(0, i - n + 1); j <= i; j ++) {
                add(b[i + 1], 1ll * (i - j + 1) * a[i - j + 1] % mod * b[j] %
                 ↪ mod);
            }
            mul(b[i + 1], k);
            for (int j = std::max(0, i - n); j < i; j ++) {
              dec(b[i + 1], 1ll * (j + 1) * b[j + 1] % mod * a[i - j] % mod);
            }
            mul(b[i + 1], v[i + 1]), mul(b[i + 1], inv);
        }
        return b;
    }

    // 多项式多点求值
    std::vector<int> eval(std::vector<int> x) const {
        if (empty()) return std::vector<int>(x.size());

        int n = std::max(x.size(), size());
        std::vector<poly> t(n * 4);
        std::vector<int> ans(x.size());
        x.resize(n);

        std::function<void(int, int, int)> build = [&] (int p, int l, int r)
         ↪ {
            if (l == r) {
                t[p] = poly{1, norm(-x[l])};
                return;
            }
            int mid = (l + r) >> 1;
            build(p * 2, l, mid), build(p * 2 + 1, mid + 1, r);
            t[p] = t[p * 2] * t[p * 2 + 1];
        };
        build(1, 0, n - 1);

        std::function<void(int, int, int, const poly &)> solve = [&] (int p,
         ↪ int l, int r, const poly &num) {
            if (l >= ans.size()) return;
            if (l == r) {
                ans[l] = num[0];
                return;
            }
            int mid = (l + r) >> 1;
            solve(p * 2, l, mid, num.mulT(t[p * 2 + 1]).modxk(mid - l + 1));
            solve(p * 2 + 1, mid + 1, r, num.mulT(t[p * 2]).modxk(r - mid));
        };
        solve(1, 0, n - 1, mulT(t[1].inv(n)));

        return ans;
    }

    // 多项式平移
    poly shift(int c) const {
        int n = size() - 1;
        std::vector<int> fact(n + 1), facv(n + 1);
        fact[0] = 1;
        for (int i = 1; i <= n; i ++) {
            fact[i] = 1ll * fact[i - 1] * i % mod;
        }
        facv[n] = qpow(fact[n], mod - 2, mod);
```

```
        for (int i = n - 1; i >= 0; i --) {
            facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
        }

        c = norm(c);
        poly a(n + 1), b(n + 1);
        for (int i = 0; i <= n; i ++) {
            a[i] = 1ll * (*this)[i] * fact[i] % mod;
        }
        for (int i = 0, w = 1; i <= n; i ++, mul(w, c)) {
            b[i] = 1ll * w * facv[i] % mod;
        }

        poly res = a.mulT(b);
        for (int i = 0; i <= n; i ++) {
            mul(res[i], facv[i]);
        }
        return res;
    }

    // Chirp Z-Transform
    std::vector<int> czt(int c, int m) const {
        c = norm(c);
        if (c == 0) {
            std::vector<int> res(m, front());
            res[0] = 0;
            for (int x : *this) add(res[0], x);
            return res;
        }

        poly a = *this;
        int n = a.size(), ic = qpow(c, mod - 2, mod);
        poly b(n + m - 1), v(std::max(n, m));
        b[0] = 1;
        for (int i = 1, w = 1; i < b.size(); i ++, mul(w, c)) {
            b[i] = 1ll * b[i - 1] * w % mod;
        }
        v[0] = 1;
        for (int i = 1, w = 1; i < v.size(); i ++, mul(w, ic)) {
            v[i] = 1ll * v[i - 1] * w % mod;
        }

        for (int i = 0; i < n; i ++) {
            mul(a[i], v[i]);
        }
        poly res = b.mulT(a).modxk(m);
        for (int i = 0; i < m; i ++) {
            mul(res[i], v[i]);
        }
        return res;
    }
};

// 多项式快速插值
poly lagrange(std::vector<std::pair<int, int>> seq) {
    int n = seq.size(); // seq 下标从 0 开始，至多 n-1 次多项式需要 n 个点值
    std::vector<poly> t(n * 4);
    std::vector<int> x(n);
    for (int i = 0; i < n; i ++) {
        x[i] = seq[i].first;
    }

    std::function<void(int, int, int)> build = [&] (int p, int l, int r) {
        if (l == r) {
            t[p] = poly{norm(-x[l]), 1};
            return;
        }
        int mid = (l + r) >> 1;
        build(p * 2, l, mid), build(p * 2 + 1, mid + 1, r);
        t[p] = t[p * 2] * t[p * 2 + 1];
    };
    build(1, 0, n - 1);

    auto res = (t[1].deriv()).eval(x);
    std::function<poly(int, int, int)> solve = [&] (int p, int l, int r) {
        if (l == r) {
            return poly{norm(1ll * seq[l].second * qpow(res[l], mod - 2,
             ↪  mod))};
        }
        int mid = (l + r) >> 1;
        return solve(p * 2, l, mid) * t[p * 2 + 1] + solve(p * 2 + 1, mid +
         ↪  1, r) * t[p * 2];
    };
    return solve(1, 0, n - 1);
}

// 连续点值平移
std::vector<int> shift_conti(std::vector<int> y, int c) {
    int n = y.size() - 1; // y 下标从 0 开始，连续点值为 f(0), ..., f(n)
    std::vector<int> fact(n + 1), facv(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++) {
```

```
        fact[i] = 1ll * fact[i - 1] * i % mod;
    }
    facv[n] = qpow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i --) {
        facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
    }

    poly a(2 * n + 1), b(n + 1);
    for (int i = 0; i <= 2 * n; i ++) {
        a[i] = qpow(norm(c - n + i), mod - 2, mod);
    }
    for (int i = 0; i <= n; i ++) {
        b[i] = 1ll * facv[i] * facv[n - i] % mod * norm(y[i]) % mod;
        if ((n - i) & 1) neg(b[i]);
    }

    poly res = a * b;
    int num = 1, z = 0;
    for (int i = c - n; i <= c; i ++) {
        int v = norm(i);
        if (v) mul(num, v); else z ++;
    }
    std::vector<int> ans(n + 1);
    for (int i = 0; i <= n; i ++) {
        ans[i] = z ? y[norm(c + i)] : 1ll * num * res[i + n] % mod;
        int v1 = norm(c - n + i), v2 = norm(c + i + 1);
        if (v1) mul(num, a[i]); else z --;
        if (v2) mul(num, v2); else z ++;
    }
    return ans;
}
```

Bostan–Mori

分式远处系数

分式远处系数：用于求解分式 $\frac{P(x)}{Q(x)}$ 的第 n 项 $[x^n]\frac{P(x)}{Q(x)}$。

考虑

$$\frac{P(x)}{Q(x)}=\frac{P(x)Q(-x)}{Q(x)Q(-x)}$$

由于 $Q(x)Q(-x)$ 是偶函数，仅偶次项有值。记 $U(x^2)=Q(x)Q(-x)$，然后将分子按照奇偶次项拆开

$$P(x)Q(-x)=C_{\text{even}}(x^2)+xC_{\text{odd}}(x^2)$$

于是

$$\frac{P(x)}{Q(x)}=\frac{C_{\text{even}}(x^2)}{U(x^2)}+\frac{xC_{\text{odd}}(x^2)}{U(x^2)}$$

由于右式第一项只影响偶次项，右式第二项只影响奇次项。于是我们只需要根据 n 的奇偶递归到其中一边即可。

当 n 为偶数时

$$[x^n]\frac{P(x)}{Q(x)}=[x^{\frac{n}{2}}]\frac{C_{\text{even}}(x)}{U(x)}$$

当 n 为奇数时

$$[x^n]\frac{P(x)}{Q(x)}=[x^{\frac{n-1}{2}}]\frac{C_{\text{odd}}(x)}{U(x)}$$

递归到 $n=0$ 时，有 $[x^0]\frac{f(x)}{g(x)}=\frac{f_0}{g_0}$。

时间复杂度 $\mathcal{O}(m\log m\log n)$，其中 m 表示分子分母的长度。

0x31 分式远处系数.cpp

```
// 分式远处系数
int BostanMori(poly p, poly q, int n) {
    for (; n; n >>= 1) {
        poly nq = q;
        for (int i = 1; i < nq.size(); i += 2) {
            neg(nq[i]);
        }
        p *= nq, q *= nq;

        int i = n & 1;
        while (i < p.size()) {
            p[i / 2] = p[i], i += 2;
        }
        p.resize(i / 2);

        int j = 0;
        while (j < q.size()) {
            q[j / 2] = q[j], j += 2;
        }
        q.resize(j / 2);
    }
    return 1ll * p[0] * qpow(q[0], mod - 2, mod) % mod;
}
```

分式远处区间系数

待填坑。

常系数齐次线性递推

常系数齐次线性递推：已知初值 $a_0, \dots, a_{k-1}$ 与系数 $c_1, \dots, c_k$（常数项 $c_0 = 0$），对于一个 k 阶齐次线性递推数列 $a_i = \sum_{j=1}^{k} c_j a_{i-j}$，求出第 n 项。

Fiduccia 算法

Fiduccia 算法：构造特征多项式 $\Gamma(x) = x^k - \sum_{i=1}^{k} c_i x^{k-i}$，记 $A(x) = \sum_{i=0}^{k-1} a_i x^i$，则有

$$a_n = (x^n \bmod \Gamma(x)) \cdot A(x)$$

（其中 $\cdot$ 代表点积）

于是应用多项式快速幂（倍增法）求出 $x^n \bmod \Gamma(x)$，最后求出点积即可。

时间复杂度 $\mathcal{O}(k \log k \log n)$。

该做法的优点是，当项数 n 与递推系数 c 固定时，可以做到 $\mathcal{O}(k \log k \log n)$ 预处理，$\mathcal{O}(k)$ 查询。

Fiduccia 算法的简单解释：我们想要知道 a_n 具体可以表示成怎样的 $a_0, \dots, a_{k-1}$ 的线性组合。相当于是初始有一个 a_n，每次选出最高项 a_i，根据公式 $a_i = \sum_{j=1}^{k} c_j a_{i-j}$ 将 a_i 消成更低的项。

从生成函数的角度考虑，相当于是将 x^i 替换成 $\sum_{j=1}^{k} c_j x^{i-j}$，等价于减去 $x^{i-k}\left(x^k - \sum_{j=1}^{k} c_j x^{k-j}\right)$。整个过程等价于 x^n 对 $x^k - \sum_{j=1}^{k} c_j x^{k-j}$ 取模。

0x31 常系数齐次线性递推（Fiduccia）.cpp

```cpp
// 常系数齐次线性递推（Fiduccia）
int linearRecurrence(std::vector<int> c, std::vector<int> y, int n) {
    int k = y.size(); // k 阶齐次线性递推数列
    if (n < k) return norm(y[n]);

    poly g(k + 1);
    g[k] = 1;
    for (int i = 1; i <= k; i ++) {
        g[k - i] = norm(-c[i]);
    }

    poly res{1}, a{0, 1};
    for (int b = n; b; b >>= 1) {
        if (b & 1) res = divmod(res * a, g).second;
        a = divmod(a * a, g).second;
    }

    int ans = 0;
    for (int i = 0; i < res.size(); i ++) {
        ans = (ans + 1ll * y[i] * res[i]) % mod;
    }
    return norm(ans);
}
```

递推式转分式

对于 k 阶齐次线性递推数列 $\{a_i\}$，我们想要构造多项式 $P(x), Q(x)$ 使得 $[x^n]\frac{P(x)}{Q(x)} = a_n$。构造如下

$$Q(x) = 1 - \sum_{i=1}^{k} c_i x^i$$

$$P(x) = \left(\left(\sum_{i=0}^{k-1} a_i x^i\right) \times Q(x)\right) \bmod x^k$$

检验：设 $\frac{P(x)}{Q(x)} = \sum_{i=0}^{\infty} h_i x^i$。当 $0 \le i < k$ 时，根据定义有 $h_i = a_i$。当 $i \ge k$ 时，化简得 $h_i = \sum_{j=1}^{k} c_j h_{i-j}$，正是 $\{a_i\}$ 的递推式。归纳得 $h_i = a_i$。

于是构造多项式 $P(x), Q(x)$，套用 Bostan-Mori 求解 $[x^n]\frac{P(x)}{Q(x)}$ 即可。

时间复杂度 $\mathcal{O}(k \log k \log n)$，但单次询问常数相较于 Fiduccia 算法更小。

0x31 常系数齐次线性递推（递推式转分式）.cpp

```cpp
// 常系数齐次线性递推（递推式转分式）
int linearRecurrence(std::vector<int> c, std::vector<int> y, int n) {
    int k = y.size(); // k 阶齐次线性递推数列
    if (n < k) return norm(y[n]);

    for (int &x : c) x = norm(x);
    for (int &x : y) x = norm(x);

    poly q = poly{1} - poly(c);
    poly p = (poly(y) * q).modxk(k);
    return BostanMori(p, q, n);
}
```

整式递推

应用：快速阶乘算法

快速阶乘算法：求 $n! \bmod p$，其中 p 是一个大质数。

（由于模数固定时可以分段打表，所以大概率不保证 p 是一个 NTT 模数，此时需要使用 MTT）。

设 $v = \lfloor \sqrt{n} \rfloor$，设多项式 $g(x) = \prod_{i=1}^{v} (x+i)$，则

$$n! = \left(\prod_{i=0}^{v-1} g(iv) \right) \cdot \left(\prod_{i=v^2+1}^{n} i \right)$$

首先后半部分的 $\prod_{i=v^2+1}^{n} i$ 可以 $\mathcal{O}(\sqrt{n})$ 求出。我们现在只关心前半部分。

设多项式 $g_d(x) = \prod_{i=1}^{d} (x+i)$，则 $d+1$ 个点值 $g_d(0), g_d(v), \dots, g_d(dv)$ 可以唯一确定 d 次多项式 $g_d(x)$。**所以对于当前的 d，我们都需要维护出这 $d+1$ 个点值。**

初始时 $d = 1$，只有点值 $g_1(0) = 1, g_1(v) = v+1$。考虑两种操作：

操作 1：$d \leftarrow 2d$。

- 补齐点值：补齐 $g_d((d+1)v), \dots, g_d(2dv)$，可以通过 $g_d(0), g_d(v), \dots, g_d(dv)$ 向右平移 dv 得到（即连续点值 $h(x) = g_d(xv)$ 向右平移 d）。
- 修正点值：有公式 $g_{2d}(x) = g_d(x) g_d(x+d)$，将所有点值 $g_d(x)$ 向右平移 d 得到 $g_d(x+d)$（即连续点值 $h(x) = g_d(xv)$ 向右平移 $\frac{d}{v}$），然后对应位相乘即可。

操作 2：$d \leftarrow d+1$。

- 修正点值：有公式 $g_{d+1}(x) = g_d(x) \cdot (x+d+1)$，直接相乘即可。
- 补齐点值：补齐 $g_{d+1}((d+1)v)$，直接暴力 $\mathcal{O}(d)$ 求出即可。

于是可以从 $d = 1$ 通过倍增变换到 $d = v$。倍增过程的时间复杂度为 $T(v) = T(\frac{v}{2}) + \mathcal{O}(v \log v) = \mathcal{O}(v \log v)$。

而我们只需要 $\mathcal{O}(\sqrt{n})$ 个点值，故时间复杂度 $\mathcal{O}(\sqrt{n} \log n)$。

0x31 快速阶乘算法.cpp

```cpp
// 快速阶乘算法
int fastFactorial(int n) {
    if (n == 0) return 1;
    if (n >= mod) return 0;

    int v = sqrt(n), iv = qpow(v, mod - 2, mod);
    std::function<std::vector<int>(int)> solve = [&] (int d) {
        if (d == 1) {
            return std::vector<int>{1, v + 1};
        }
        if (d & 1) {
            auto y = solve(d - 1);
            for (int i = 0; i < y.size(); i ++) {
                mul(y[i], i * v + d);
            }
            int num = 1;
            for (int i = 1; i <= d; i ++) {
                mul(num, d * v + i);
            }
            y.push_back(num);
            return y;
        } else {
            auto y = solve(d / 2);
            auto y1 = shift_conti(y, d / 2);
            y.insert(y.end(), y1.begin() + 1, y1.end());
            auto y2 = shift_conti(y, 1ll * (d / 2) * iv % mod);
            for (int i = 0; i < y.size(); i ++) {
                mul(y[i], y2[i]);
            }
            return y;
        }
    };

    auto res = solve(v);
    int ans = 1;
    for (int i = 0; i < v; i ++) {
        mul(ans, res[i]);
    }
    for (int i = v * v + 1; i <= n; i ++) {
        mul(ans, i);
    }
    return ans;
}
```

多项式复合

~~待填坑。~~

拉格朗日反演

~~待填坑。~~

Power Projection

Power Projection：给出多项式 $A(x), B(x)$ 以及正整数 n，保证 $[x^0]B(x) = 0$。求出

$$[x^{n-1}]A(x)B(x)^i \quad (0 \le i < n)$$

0x31 Power Projection.cpp

```cpp
// Power Projection
poly PowerProjection(poly a, poly b, int n) {
    // assert(n >= 1);
    // assert(!b.size() || !b[0]);

    a.resize(n), b.resize(n);
    for (int &x : b) {
        neg(x);
    }

    int n0 = n --;
    for (int m = 1; n; m <<= 1, n >>= 1) {
        int L = n * 2 + 2, len = 1;
        while (len < 2 * m * L) len <<= 1;

        poly d(len);
        for (int i = 0; i < m; i ++) {
            for (int j = 0; j <= n; j ++) {
                d[i * L + j] = b[i * (n + 1) + j];
            }
        }
        dft(d);

        int h = len >> 1;
        poly work(h);
        for (int i = 0; i < h; i ++) {
            work[i] = 1ll * d[i] * d[i + h] % mod;
            std::swap(d[i], d[i + h]);
        }
        idft(work);

        int t = n / 2 + 1;
        poly p(2 * m * t), q(2 * m * t);
        for (int i = 0; i < 2 * m - 1; i ++) {
            for (int j = 0; j < t; j ++) {
                q[i * t + j] = work[i * (n + 1) + j];
            }
        }
        for (int i = 0; i < m; i ++) {
            for (int j = 0; j < t; j ++) {
                add(q[(m + i) * t + j], b[i * (n + 1) + 2 * j]);
                add(q[(m + i) * t + j], b[i * (n + 1) + 2 * j]);
            }
        }

        poly c(len);
        for (int i = 0; i < m; i ++) {
            for (int j = 0; j <= n; j ++) {
                c[i * L + j] = a[i * (n + 1) + j];
            }
        }
        dft(c);
        for (int i = 0; i < len; i ++) {
            mul(c[i], d[i]);
        }
        idft(c);

        for (int i = 0; i < 2 * m - 1; i ++) {
            for (int j = 0; j < t; j ++) {
                p[i * t + j] = c[i * L + 2 * j + (n & 1)];
            }
        }
        for (int i = 0; i < m; i ++) {
            for (int j = 0; j < t; j ++) {
                add(p[(m + i) * t + j], a[i * (n + 1) + 2 * j + (n & 1)]);
            }
        }
        a.swap(p), b.swap(q);
    }

    std::reverse(a.begin(), a.end());
    a.resize(n0);
    return a;
}
```

多项式复合逆

~~待填坑。~~

0x32 数学 集合幂级数

FWT

位运算卷积：已知序列 $a_0,\dots,a_{2^m-1}$ 和序列 $b_0,\dots,b_{2^m-1}$，求出 $c_0,\dots,c_{2^m-1}$，其中

$$c_k=\sum_{i\oplus j=k}a_ib_j$$

其中 $\oplus$ 表示一种位运算（or / and / xor）。

DWT

> FFT 多项式乘法的核心思想：A,B 的系数转 A,B 的点值，A,B 的点值点积转 C 的点值，C 的点值转 C 的系数。

我们想要 DWT 与 IDWT 有着类似的功能。设 $\mathrm{dwt}(a)$ 表示幂级数 a 经过 DWT 变换后得到的幂级数（点值），我们希望

$$a\times b=c\iff \mathrm{dwt}(a)\cdot\mathrm{dwt}(b)=\mathrm{dwt}(c)$$

由于 DFT 是线性变换，我们希望 DWT 也是线性变换。

记序列长度 $n=2^m$，设 $\mathrm{dwt}(a)_i=\sum_{j=0}^{n-1}c(i,j)a_j$，其中变换系数 $c(i,j)$ 表示变换前下标 j 对变换后下标 i 的贡献。

由 $\mathrm{dwt}(a)\cdot\mathrm{dwt}(b)=\mathrm{dwt}(c)$，得

$$\mathrm{dwt}(c)_i=\mathrm{dwt}(a)_i\cdot\mathrm{dwt}(b)_i$$

$$\left(\sum_{p=0}^{n-1}c(i,p)c_p\right)=\left(\sum_{j=0}^{n-1}c(i,j)a_j\right)\cdot\left(\sum_{k=0}^{n-1}c(i,k)b_k\right)$$

$$\sum_{p=0}^{n-1}c(i,p)c_p=\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}c(i,j)c(i,k)a_jb_k$$

由 $a\times b=c$，得

$$\begin{aligned}\sum_{p=0}^{n-1}c(i,p)c_p&=\sum_{p=0}^{n-1}c(i,p)\sum_{t_1\oplus t_2=p}a_{t_1}b_{t_2}\\&=\sum_{p=0}^{n-1}\sum_{t_1\oplus t_2=p}c(i,t_1\oplus t_2)a_{t_1}b_{t_2}\\&=\sum_{t_1=0}^{n-1}\sum_{t_2=0}^{n-1}c(i,t_1\oplus t_2)a_{t_1}b_{t_2}\end{aligned}$$

对比上下两段的最后一个式子，**发现充要条件为** $c(i,j)c(i,k)=c(i,j\oplus k)$。

由于位运算可以按位考虑，可以这样构造 $c(i,j)$：**先得到单个二进制位的转移系数** $c([0,1],[0,1])$，满足 $c(i,j)c(i,k)=c(i,j\oplus k)$。对于所有的 $c(i,j)$，我们将 i,j 二进制分解，**将对应位的转移系数相乘**

$$c(i,j)=\prod_{x=0}^{m-1}c(i_x,j_x)$$

（其中 i_x,j_x 分别表示 i,j 的二进制下第 x 位）

代入条件 $c(i,j)c(i,k)=c(i,j\oplus k)$，可验证其成立

$$\begin{aligned}c(i,j)c(i,k)&=\left(\prod_{x=0}^{m-1}c(i_x,j_x)\right)\left(\prod_{x=0}^{m-1}c(i_x,k_x)\right)\\&=\prod_{x=0}^{m-1}c(i_x,j_x)c(i_x,k_x)\\&=\prod_{x=0}^{m-1}c(i_x,j_x\oplus k_x)\\&=c(i,j\oplus k)\end{aligned}$$

现在考虑优化计算 DWT。考虑按最高位拆分，记 i',j' 表示 i,j 去掉最高位的值，记 i_h 表示 i 的最高位

$$\begin{aligned}\operatorname{dwt}(a)_i &= \sum_{j=0}^{n-1} c(i,j)a_j \\ &= \sum_{j=0}^{n/2-1} c(i,j)a_j + \sum_{j=n/2}^{n-1} c(i,j)a_j \\ &= c(i_h,0)\left(\sum_{j=0}^{n/2-1} c(i',j')a_j\right) + c(i_h,1)\left(\sum_{j=n/2}^{n-1} c(i',j')a_j\right)\end{aligned}$$

发现括号里的式子是一个子问题。记 a_0 表示幂级数 a 最高位为 0 的部分，a_1 表示幂级数 a 最高位为 1 的部分。

- 对于 $k < n/2$，代入 $i = k$ 得

$$\operatorname{dwt}(a)_k = c(0,0)\cdot\operatorname{dwt}(a_0)_k + c(0,1)\cdot\operatorname{dwt}(a_1)_k$$

- 对于 $k < n/2$，代入 $i = k + n/2$ 得

$$\operatorname{dwt}(a)_{k+n/2} = c(1,0)\cdot\operatorname{dwt}(a_0)_k + c(1,1)\cdot\operatorname{dwt}(a_1)_k$$

如果我们知道 $\operatorname{dwt}(a_0)_i, \operatorname{dwt}(a_1)_i$ 在 $i = 0,1,\dots,n/2-1$ 处的值，就可以 $\mathcal{O}(n)$ 求出 $\operatorname{dwt}(a)_i$ 在 $i = 0,1,\dots,n-1$ 处的值。

这是一个分治的过程，不断分治直到仅剩一个项即可。

时间复杂度 $\mathcal{O}(m2^m)$。

IDWT

在 DWT 中，我们只需要

$$c = \begin{bmatrix} c(0,0) & c(0,1) \\ c(1,0) & c(1,1) \end{bmatrix}$$

这四个数即可完成变换，记该矩阵为位矩阵 c。

如果我们要进行 IDWT，则需要位矩阵 c 的逆矩阵 c^{-1}。类似地，有

$$a_i = \sum_{j=0}^{n-1} c^{-1}(i,j)\operatorname{dwt}(a)_j$$

套用 DWT 的模板即可。

在构造矩阵 c 的时候，我们需要小心一点，得确保矩阵 c 存在逆矩阵 c^{-1}。

位运算卷积

or 卷积：or 运算的位矩阵，及其逆矩阵

$$c = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix},\quad c^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}$$

or 卷积的正变换等价于子集求和，因为 $c(i,j) = [i \operatorname{and} j = j]$。

and 卷积：and 运算的位矩阵，及其逆矩阵

$$c = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix},\quad c^{-1} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$

and 卷积的正变换等价于超集求和，因为 $c(i,j) = [i \operatorname{or} j = j]$。

xor 卷积：xor 运算的位矩阵，及其逆矩阵

$$c = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix},\quad c^{-1} = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

xor 卷积的正变换，其变换系数 $c(i,j) = (-1)^{\operatorname{popcount}(i \operatorname{and} j)}$。

xor 卷积的 IDWT 相较于 DWT，每一位的变换系数需要乘以 $\frac{1}{2}$，一共有 m 位，总共的影响为 $\frac{1}{2^m} = \frac{1}{n}$。所以 xor 卷积的 IDWT 只需套用 DWT 的模板，最后乘以 $\frac{1}{n}$ 即可。

0x32 FWT.cpp

```cpp
void dwtOr(std::vector<int> &a, int opt) {
    int n = a.size();
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < n; i += (k << 1)) {
            for (int j = 0; j < k; j ++) {
                opt > 0 ? add(a[i + j + k], a[i + j]) : dec(a[i + j + k], a[i
                ↪  + j]);
            }
        }
    }
}
void dwtAnd(std::vector<int> &a, int opt) {
    int n = a.size();
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < n; i += (k << 1)) {
            for (int j = 0; j < k; j ++) {
                opt > 0 ? add(a[i + j], a[i + j + k]) : dec(a[i + j], a[i + j
                ↪  + k]);
            }
```

```
        }
    }
}
void dwtXor(std::vector<int> &a, int opt) {
    int n = a.size();
    for (int k = 1; k < n; k <<= 1) {
        for (int i = 0; i < n; i += (k << 1)) {
            for (int j = 0; j < k; j ++) {
                int u = a[i + j], v = a[i + j + k];
                add(a[i + j] = u, v), dec(a[i + j + k] = u, v);
            }
        }
    }
    if (opt < 0) {
        int inv = qpow(n, mod - 2, mod);
        for (int i = 0; i < n; i ++) {
            mul(a[i], inv);
        }
    }
}

// 位运算卷积
using dwtFunc = void(*)(std::vector<int>&, int);
std::vector<int> conv(std::vector<int> a, std::vector<int> b, dwtFunc dwt) {
    dwt(a, +1), dwt(b, +1);
    for (int i = 0; i < a.size(); i ++) {
        mul(a[i], b[i]);
    }
    dwt(a, -1);
    return a;
}
```

子集 dp

子集 dp：已知序列 $a_0,\dots,a_{2^m-1}$，求出 $b_0,\dots,b_{2^m-1}$，其中 $b_i=\sum_{j\subseteq i}a_j$。

将一个二进制状态视作一个 m 维的坐标，第 i 维的取值等于二进制下第 i 位的值。则二进制下的子集和，相当于是要求每个维度都小于等于当前坐标的所有坐标的权值和，运用高维前缀和即可。

当信息满足可减性时，可以进行子集 dp 的逆过程。

时间复杂度 $\mathcal{O}(m2^m)$。

```
// 注意：要先枚举维度 i，再枚举状态 S
for (int i = 0; i < m; i ++) {
    for (int S = 0; S < (1 << m); S ++) {
        if (S >> i & 1) {
            f[S] += f[S ^ (1 << i)];
        }
    }
}
```

半在线子集 dp：求出 $b_0,\dots,b_{2^m-1}$，其中 b_i 与 $\sum_{j\subseteq i}b_j$ 强相关。

按照 $|S|$ 从小到大分层计算。每一层都做一次子集 dp，然后更新当前层的所有状态。

时间复杂度 $\mathcal{O}(m^2 2^m)$。

```
for (int s = 1; s <= m; s ++) {
    for (int S = 0; S < (1 << m); S ++) {
        tmp[S] = f[S];
    }
    for (int i = 0; i < m; i ++) {
        for (int S = 0; S < (1 << m); S ++) {
            if (S >> i & 1) {
                tmp[S] += tmp[S ^ (1 << i)];
            }
        }
    }

    for (int S = 0; S < (1 << m); S ++) {
        if (__builtin_popcount(S) == s) {
            // 使用 tmp[S] 更新 f[S]
        }
    }
}
```

子集卷积

子集卷积：已知序列 $a_0,\dots,a_{2^m-1}$ 和序列 $b_0,\dots,b_{2^m-1}$，求出 $c_0,\dots,c_{2^m-1}$，其中

$$c_k=\sum_{i\text{ or }j=k,i\text{ and }j=\varnothing}a_ib_j$$

注意到 $i\text{ and }j=\varnothing$ 的充要条件为 $|i|+|j|=|i\text{ or }j|$。于是引入序列 f 的占位多项式 F

$$F_{p,i}=\begin{cases}f_i & |i|=p\\ 0 & |i|\neq p\end{cases}$$

将占位多项式代入到子集卷积的式子中

$$C_{s,k}=\sum_{p+q=s}\sum_{i\text{ or }j=k}A_{p,i}B_{q,j}$$

从 or 卷积的角度来看，有

$$C_s = \sum_{p+q=s} A_p \operatorname{or} B_q$$

于是将 A_p 和 B_q 进行 DWT，求出所有的 C_s 后再进行 IDWT 即可。
时间复杂度 $\mathcal{O}(m^2 2^m)$，空间复杂度 $\mathcal{O}(m2^m)$。

0x32 子集卷积.cpp

```cpp
// 子集卷积
std::vector<int> subset_conv(std::vector<int> a, std::vector<int> b) {
    int n = a.size(), m = std::__lg(n);
    std::vector<std::vector<int>> A(m + 1, std::vector<int>(n, 0));
    std::vector<std::vector<int>> B(m + 1, std::vector<int>(n, 0));
    std::vector<std::vector<int>> C(m + 1, std::vector<int>(n, 0));

    for (int i = 0; i < n; i ++) {
        int s = __builtin_popcount(i);
        A[s][i] = a[i], B[s][i] = b[i];
    }
    for (int i = 0; i <= m; i ++) {
        dwtOr(A[i], +1), dwtOr(B[i], +1);
    }
    for (int i = 0; i <= m; i ++) {
        for (int j = 0; j <= i; j ++) {
            for (int k = 0; k < n; k ++) {
                add(C[i][k], 1ll * A[j][k] * B[i - j][k] % mod);
            }
        }
    }
    for (int i = 0; i <= m; i ++) {
        dwtOr(C[i], -1);
    }

    std::vector<int> c(n);
    for (int i = 0; i < n; i ++) {
        c[i] = C[__builtin_popcount(i)][i];
    }
    return c;
}
```

半在线子集卷积：已知序列 $g_0, \ldots, g_{2^m-1}$，求出 $f_0, \ldots, f_{2^m-1}$，其中对于 $k > 0$ 满足

$$f_k = \sum_{i \operatorname{or} j = k, i \operatorname{and} j = \varnothing} f_i g_j$$

先根据具体问题的定义得出 f_0（g_0 一般来说等于 0）。
然后按照 $|S|$ 从小到大分层计算，每一层都根据前面层的 DWT 结果得到当前层的 DWT 结果，然后先 IDWT 回去，更新当前层的所有状态，然后再 DWT 回来，为后面层的计算做准备。
时间复杂度 $\mathcal{O}(m^2 2^m)$，空间复杂度 $\mathcal{O}(m2^m)$。

0x32 半在线子集卷积.cpp

```cpp
// 半在线子集卷积
std::vector<int> subset_conv(std::vector<int> g) {
    int n = g.size(), m = std::__lg(n);
    std::vector<std::vector<int>> F(m + 1, std::vector<int>(n, 0));
    std::vector<std::vector<int>> G(m + 1, std::vector<int>(n, 0));

    for (int i = 1; i < n; i ++) { // 一般来说 g_0 = 0，所以不考虑 G[0]
        G[__builtin_popcount(i)][i] = g[i];
    }
    for (int i = 1; i <= m; i ++) {
        dwtOr(G[i], +1);
    }

    int f0 = 1;
    std::fill(F[0].begin(), F[0].end(), f0); // 一般来说 f_0 = 1，这一步模拟
    ↪ 了 dwtOr(F[0], +1)

    std::vector<int> f(n);
    f[0] = f0;
    for (int i = 1; i <= m; i ++) {
        for (int j = 1; j <= i; j ++) {
            for (int k = 0; k < n; k ++) {
                add(F[i][k], 1ll * G[j][k] * F[i - j][k] % mod);
            }
        }
        dwtOr(F[i], -1);

        for (int S = 0; S < n; S ++) {
            if (__builtin_popcount(S) == i) {
                // 在此更新 F[i][S]，即为真正的 f[S]
                f[S] = F[i][S];
            } else {
                F[i][S] = 0;
            }
        }
        dwtOr(F[i], +1);
```

```
    }
    return f;
}
```

0x33 数学 组合计数

排列组合

排列数：A_n^m，从 n 个有标号元素中，取出 m 个元素排成一列的方案数（需要考虑取数顺序）。

$$A_n^m = \frac{n!}{(n-m)!} = n^{\underline{m}}$$

组合数：C_n^m 或 $\binom{n}{m}$，从 n 个有标号元素中，取出 m 个元素组成集合的方案数（不需要考虑取数顺序）。

$$\binom{n}{m} = \begin{cases} \frac{n^{\underline{m}}}{m!} & m \in \mathbb{Z}, m \geq 0 \\ 0 & m \in \mathbb{Z}, m < 0 \end{cases}$$

组合数常见恒等式

阶乘展开式：$n, m \in \mathbb{Z}$，$n \geq m \geq 0$

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

归纳恒等式：$m \in \mathbb{Z}$

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

对称恒等式：$n, m \in \mathbb{Z}$，$n \geq 0$

$$\binom{n}{m} = \binom{n}{n-m}$$

吸收恒等式：$m \in \mathbb{Z}$

$$\binom{n}{m} = \frac{n}{m}\binom{n-1}{m-1} \qquad (m \neq 0)$$

$$\binom{n}{m} = \frac{n}{n-m}\binom{n-1}{m} \qquad (n \neq m)$$

上指标反转：$m \in \mathbb{Z}$

$$\binom{n}{m} = (-1)^m \binom{m-n-1}{m}$$

三项式版恒等式：$m, k \in \mathbb{Z}$

$$\binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}$$

二项式定理：$n \in \mathbb{N}$

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

> 一些常见特例：
>
> - $x = 1$ 且 $y = a$：
>
> $$\sum_{i=0}^{n} \binom{n}{i} a^i = (1+a)^n$$
>
> - $x = 1$ 且 $y = -1$：
>
> $$\sum_{i=0}^{n} (-1)^i \binom{n}{i} = [n = 0]$$

广义二项式定理：$r \in \mathbb{R}$，$y > 0$，$|x| < y$

$$(x+y)^r = \sum_{k=0}^{\infty} \frac{r^{\underline{k}}}{k!} x^k y^{r-k}$$

平行求和法：$m \in \mathbb{Z}$

$$\sum_{0 \leq k \leq m} \binom{n+k}{k} = \binom{n+m+1}{m}$$

上指标求和法：$n, m \in \mathbb{Z}$，$n, m \geq 0$

$$\sum_{0 \leq k \leq n} \binom{k}{m} = \binom{n+1}{m+1}$$

范德蒙德卷积：$m \in \mathbb{Z}$

$$\sum_{k} \binom{s}{k}\binom{t}{m-k} = \binom{s+t}{m}$$

范德蒙德卷积变式：$m, s, t \in \mathbb{N}$

$$\sum_{0 \le k \le m} \binom{k}{s} \binom{m-k}{t} = \binom{m+1}{s+t+1}$$

斜对角线求和法：$n \in \mathbb{N}$

$$\sum_{i=0}^{n} \binom{n-i}{i} = \mathrm{Fib}_{n+1}$$

> 等价于长度为 $n-1$ 的没有相邻 1 的二进制串个数。

组合数带权和：$n \in \mathbb{N}$

$$\sum_{i=0}^{n} i \binom{n}{i} = n 2^{n-1}$$

组合数带权平方和：$n \in \mathbb{N}$

$$\sum_{i=0}^{n} i^2 \binom{n}{i} = n(n+1) 2^{n-2}$$

插板法

问题 1：将 n 个完全相同的元素分成 k 组，保证每组至少有一个元素。等价于 $\sum_{i=1}^{k} x_i = n$ 且 $\forall x_i \ge 1$ 的整数解个数。

相当于将 $k-1$ 块板，插入 $n-1$ 个间隙中。

$$\binom{n-1}{k-1}$$

问题 2：将 n 个完全相同的元素分成 k 组，允许组内元素为空。等价于 $\sum_{i=1}^{k} x_i = n$ 且 $\forall x_i \ge 0$ 的整数解个数。

给 k 组分别补一个元素，转化为问题 1。

$$\binom{n+k-1}{k-1}$$

问题 3：将 n 个完全相同的元素分成 k 组，保证第 i 组至少分到 a_i 个元素。

给第 i 组提前安排 a_i 个元素，转化为问题 2。

$$\binom{n-\sum a_i+k-1}{k-1}$$

Lucas 定理

Lucas 定理：对于质数 p

$$\binom{n}{m} \equiv \binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor} \binom{n \bmod p}{m \bmod p} \pmod p$$

相当于是将 n, m 转化为 p 进制，将每一位上对应的组合数 $\binom{n_i}{m_i}$ 相乘后对 p 取模。

组合数 mod 2：当 $p=2$ 时，$\binom{n}{m} \equiv \binom{\lfloor n/2 \rfloor}{\lfloor m/2 \rfloor} \binom{n \bmod 2}{m \bmod 2} \pmod 2$。

故 $\binom{n}{m} \bmod 2 = 1$，当且仅当在二进制下 m 是 n 的子集。

组合数常见求法

求法 1：递推

> **适用情况：对模数不做要求，可以承受平方时间的预处理（$n \le 5000$）。**
> 时间复杂度：预处理 $\mathcal{O}(n^2)$，查询 $\mathcal{O}(1)$。

初值 $\binom{n}{0} = \binom{n}{n} = 1$，递推式 $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$。

```
for (int i = 0; i <= n; i ++) {
    C[i][0] = C[i][i] = 1;
    for (int j = 1; j < i; j ++) {
        C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % mod;
    }
}
```

求法 2：预处理阶乘与阶乘逆元

> **适用情况：要求模数为质数，可以承受线性时间的预处理（$n \le 10^6$）。**
> 时间复杂度：预处理 $\mathcal{O}(n)$，查询 $\mathcal{O}(1)$。

将组合数使用阶乘形式表达

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

考虑预处理出阶乘与阶乘逆元。

设 $\mathrm{fact}_i = i!$，则有初值 $\mathrm{fact}_0 = 1$，递推式 $\mathrm{fact}_i = \mathrm{fact}_{i-1} \times i$。

设 $\mathrm{facv}_i = \frac{1}{i!}$，则有初值 $\mathrm{facv}_n = \mathrm{fact}_n^{-1}$，递推式 $\mathrm{facv}_i = \mathrm{facv}_{i+1} \times (i+1)$。

于是先正着递推一遍得到 $\text{fact}_{0,\dots,n}$，然后通过一次求逆元得到 facv_n，最后再反着递推一遍得到 $\text{facv}_{n,\dots,0}$。

0x33 组合数（预处理阶乘与阶乘逆元）.cpp

```cpp
// 组合数（预处理阶乘与阶乘逆元）
struct BinomCoef {
    std::vector<int> fact, facv;

    BinomCoef() {}
    BinomCoef(int n) {
        init(n);
    }

    void init(int n) {
        fact.resize(n + 1), facv.resize(n + 1);
        fact[0] = 1;
        for (int i = 1; i <= n; i ++) {
            fact[i] = 1ll * fact[i - 1] * i % mod;
        }
        facv[n] = qpow(fact[n], mod - 2, mod);
        for (int i = n - 1; i >= 0; i --) {
            facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
        }
    }

    int binom(int n, int m) {
        if (n < m || m < 0) {
            return 0;
        }
        return 1ll * facv[m] * facv[n - m] % mod * fact[n] % mod;
    }
    int perm(int n, int m) {
        if (n < m || m < 0) {
            return 0;
        }
        return 1ll * fact[n] * facv[n - m] % mod;
    }
} bc;
```

求法 3：暴力统计

> **适用范围：要求模数为质数，m 的范围可以承受线性枚举。**
> 时间复杂度：无预处理，查询 $\mathcal{O}(m)$。

将组合数表达为

$$\binom{n}{m} = \frac{n^{\underline{m}}}{m!}$$

注意到分子分母均只有 m 项，当 n 很大但 m 很小时，可以暴力枚举。
具体实现时，可以先分别求出分子与分母，然后再对分母求一次逆元。

求法 4：Lucas 定理

> **适用范围：要求模数为质数，模数以内的组合数可以预处理。**
> 时间复杂度：预处理 $\mathcal{O}(p)$，查询 $\mathcal{O}(\log_p n)$。

由 Lucas 定理，得

$$\binom{n}{m} \equiv \binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor}\binom{n \bmod p}{m \bmod p} \pmod p$$

先预处理出模数以内的组合数。然后不断地使用 Lucas 定理，直到 n, m 递降至模数以内。

求法 5：exLucas 定理

~~待填坑。~~

求法 6：分段打表

> **适用范围：要求模数为固定质数，代码长度限制不小，可以承受段长 $\mathcal{O}(W)$ 时间的查询。**
> **（正常数据范围 $n \le 10^9$，配合 Lucas 定理可以做到 $\text{mod} \le 10^9$ 且 $n \le 10^{18}$）**
> 时间复杂度：打表预处理，查询 $\mathcal{O}(W)$。

仍然将组合数使用阶乘形式表达，每次查询相当于要得到三个阶乘的值。
考虑分段打表。记段长为 W（取 $10^5 \sim 10^6$ 相对合适），先通过打表预处理出所有 W 的倍数的阶乘。
每次查询 $n!$ 的时候，就找到最近的 W 的倍数，根据该答案继续向上乘。

邪修：快速阶乘算法

> **适用范围：要求模数为不固定质数（$n \le 10^9$）。**
> 时间复杂度：无预处理，查询 $\mathcal{O}(\sqrt{n}\log n)$。

直接使用快速阶乘算法！得到三个阶乘的值。

求法 7：阶乘分解质因数

> **适用范围 1：对模数不做要求，可以承受线性时间的查询（$n \le 10^6$）。**
> 时间复杂度：预处理 $\mathcal{O}(n)$，查询 $\mathcal{O}(n)$。
> **适用范围 2：高精度组合数（$n \le 5000$）。**

仍然将组合数使用阶乘形式表达，我们考虑 $1 \sim n$ 中的每个质数出现的次数。
先将 $1 \sim n$ 中所有的质数提前筛出来，大约有 $\frac{n}{\ln n}$ 个质数。对于质数 p，在阶乘 $n!$ 中 p 的出现次数为

$$\sum_{i=1}^{\lfloor \log_p n \rfloor} \left\lfloor \frac{n}{p^i} \right\rfloor$$

于是将三个阶乘都进行质因数分解，得出每个质数的出现次数，然后将所有质数的贡献相乘即可。
时间复杂度 $\mathcal{O}(n)$（高精度组合数的复杂度另算）。

斯特林数

第一类斯特林数

~~待填坑。~~

第二类斯特林数

~~待填坑。~~

卡特兰数

卡特兰数：Cat_n，从原点 $(0,0)$ 出发，每次只能向 x 轴或 y 轴正方向走，且不能越过直线 $y=x$，到达 (n,n) 的方案数。
下标从 0 开始的若干项：

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, \ldots$$

卡特兰数通项公式：

$$\begin{aligned}\mathrm{Cat}_n &= \binom{2n}{n} - \binom{2n}{n-1} \\ &= \frac{\binom{2n}{n}}{n+1}\end{aligned}$$

简单推导：越过直线 $y=x$，相当于是触碰直线 $y=x+1$。
将触碰 $y=x+1$ 的第一个点以后的路径，关于 $y=x+1$ 对称，就可以得到一条从 $(0,0)$ 到 $(n-1,n+1)$ 的路径。
于是「$(0,0) \to (n,n)$ 且越过 $y=x$ 的路径」与「$(0,0) \to (n-1,n+1)$ 的路径」是一一对应的，方案数均为 $\binom{2n}{n-1}$。
正难则反，不越过直线 $y=x$ 的方案数，即为总方案数 $\binom{2n}{n}$ 减去越过直线的方案数 $\binom{2n}{n-1}$。

卡特兰数递推公式：

$$\begin{aligned}\mathrm{Cat}_0 &= 1 \\ \mathrm{Cat}_n &= \sum_{i=0}^{n-1} \mathrm{Cat}_i \mathrm{Cat}_{n-i-1}\end{aligned}$$

简单推导：枚举第一次触碰直线 $y=x$ 的位置 (i,i)，其中 $i \ge 1$。
由于在 (i,i) 之前不能触碰直线 $y=x$，相当于是 $(1,0) \to (i,i-1)$ 不越过直线 $y=x-1$，于是这部分的方案数为 $\mathrm{Cat}_{i-1}\mathrm{Cat}_{n-i}$。

卡特兰数递推公式 2：

$$\begin{aligned}\mathrm{Cat}_0 &= 1 \\ \mathrm{Cat}_n &= \frac{4n-2}{n+1}\mathrm{Cat}_{n-1}\end{aligned}$$

常见卡特兰数问题：

- 从原点 $(0,0)$ 出发，每次只能向 x 轴或 y 轴正方向走，且不能越过直线 $y=x$，到达 (n,n) 的方案数 Cat_n。
- 以 $1,2,\ldots,n$ 作为 n 个元素的进栈序列时，出栈序列的方案数 Cat_n。
- 以 n 对括号组成的合法括号序列的方案数 Cat_n。
- 以 n 个节点构造无标号、左右儿子区分的二叉树个数 Cat_n。
- 以 n 个节点构造无标号、儿子区分的有根树个数 Cat_{n-1}。
- 以圆周上的 $2n$ 个点为端点，连互不相交的 n 条弦的方案数 Cat_n。
- 将凸 n 边形用其中 $n-3$ 条对角线分割为互不重叠的 $n-2$ 个三角形的方案数 Cat_{n-2}。

- 将 $1 \sim 2n$ 中的每个数，不重不漏地填入一个 $2 \times n$ 的矩阵，每个元素大于其左方、上方元素的方案数 Cat_n。

贝尔数

贝尔数：B_n，表示将 n 个两两不同的元素，划分成若干个互不区分的非空子集的方案数。

下标从 0 开始的若干项：

$$1, 1, 2, 5, 15,$$
$$52, 203, 877, 4140,$$
$$21147, 115975, 678570, 4213597,$$
$$27644437, 190899322, 1382958545, ...$$

贝尔数递推公式：

$$B_0 = 1$$
$$B_n = \sum_{i=0}^{n-1} \binom{n-1}{i} B_i$$

简单推导：枚举与元素 n 在同一集合的元素个数 i，共有 $\binom{n-1}{i}$ 种选法，剩下的元素继续划分，于是这部分的方案数为 $\binom{n-1}{i} B_{n-1-i}$。

贝尔数级别的暴搜：当我们需要将 n 个两两不同的元素，划分成若干个互不区分的非空子集时，**若 n 较小可以直接暴搜，复杂度为 B_n 级别**。

设 $s_1, \ldots, s_t$ 表示当前时刻已经存在的集合，每次加入一个新元素 x 时有两种转移

- 选择一个已经存在的集合 s_i，将 x 加入 s_i。
- 新开一个集合 s_{t+1}，将 x 加入 s_{t+1}。

贝尔数与第二类斯特林数的关系：

$$B_n = \sum_{i=0}^{n} \left\{ {n \atop i} \right\}$$

因为第二类斯特林数 $\left\{ {n \atop i} \right\}$ 表示将 n 个两两不同的元素，划分成 i 个互不区分的非空子集的方案数。

斐波那契数列

斐波那契数列：Fib_n。

斐波那契数列递推公式：

$$\mathrm{Fib}_0 = 0, \mathrm{Fib}_1 = 1$$
$$\mathrm{Fib}_n = \mathrm{Fib}_{n-1} + \mathrm{Fib}_{n-2}$$

斐波那契数列递推公式（矩阵形式）：

$$\begin{bmatrix} \mathrm{Fib}_n & \mathrm{Fib}_{n-1} \end{bmatrix} = \begin{bmatrix} \mathrm{Fib}_{n-1} & \mathrm{Fib}_{n-2} \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

即

$$\begin{bmatrix} \mathrm{Fib}_n & \mathrm{Fib}_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

该转移矩阵可以求逆，从而实现逆向转移。

斐波那契数列通项公式：

$$\mathrm{Fib}_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

当 5 是模 p 意义下的二次剩余时（一个常见模数是 $10^9 + 9$），可以直接暴力预先找出 $\sqrt{5}$，然后使用通项公式计算。

否则，考虑使用扩展数域进行计算，定义扩展数域 $\mathbb{F}_p(\sqrt{5})$ 为

$$\{a + b\sqrt{5} \mid a, b \in \mathbb{F}_p\}$$

可以验证，$\mathbb{F}_p(\sqrt{5})$ 关于加、减、乘、除封闭。记 (a, b) 表示 $a + b\sqrt{5}$，则有

$$(a, b) \pm (c, d) = (a \pm c, b \pm d)$$
$$(a, b) \times (c, d) = (ac + 5bd, ad + bc)$$
$$(a, b) \div (c, d) = \left(\frac{ac - 5bd}{c^2 - 5d^2}, \frac{bc - ad}{c^2 - 5d^2}\right)$$

广义斐波那契数列：将递推公式反过来写，有 $\mathrm{Fib}_n = \mathrm{Fib}_{n+2} - \mathrm{Fib}_{n+1}$。

考虑将斐波那契数列扩展到负数域，经过推导可得

$$\mathrm{Fib}_{-n} = (-1)^{n-1} \mathrm{Fib}_n$$

类斐波那契数列：已知数列 $G_0 = a, G_1 = b$，$G_n = G_{n-1} + G_{n-2}$，则有

$$G_n = \mathrm{Fib}_{n-1} \times a + \mathrm{Fib}_n \times b$$

类斐波那契数列相加，仍然是类斐波那契数列。只需将相应的 a, b 相加即可。

斐波那契数列常见恒等式

斐波那契数列求和：

$$\sum_{i=1}^{n} \mathrm{Fib}_i = \mathrm{Fib}_{n+2} - 1$$

斐波那契数列平方和：

$$\sum_{i=1}^{n} \mathrm{Fib}_i^2 = \mathrm{Fib}_n \mathrm{Fib}_{n+1}$$

卡西尼性质：

$$\mathrm{Fib}_{n-1}\mathrm{Fib}_{n+1} - \mathrm{Fib}_n^2 = (-1)^n$$

简单推导：$\mathrm{Fib}_{n-1}\mathrm{Fib}_{n+1} - \mathrm{Fib}_n^2 = \det \begin{pmatrix} \mathrm{Fib}_{n+1} & \mathrm{Fib}_n \\ \mathrm{Fib}_n & \mathrm{Fib}_{n-1} \end{pmatrix} = \det \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = (-1)^n$。

附加性质：

$$\mathrm{Fib}_{n+m} = \mathrm{Fib}_{n+1}\mathrm{Fib}_m + \mathrm{Fib}_n\mathrm{Fib}_{m-1}$$

简单推导：可以使用“类斐波那契数列”的知识解释。

记 $G_0 = \mathrm{Fib}_n, G_1 = \mathrm{Fib}_{n+1}$，$G_k = G_{k-1} + G_{k-2}$，则有 $\mathrm{Fib}_{n+m} = G_m$。

斐波那契数列相邻项互质性质：

$$\gcd(\mathrm{Fib}_n, \mathrm{Fib}_{n+1}) = 1$$

斐波那契数列 gcd 性质：

$$\gcd(\mathrm{Fib}_n, \mathrm{Fib}_m) = \mathrm{Fib}_{\gcd(n,m)}$$

简单推导：不妨设 $n < m$，则有

$$\begin{aligned} \gcd(\mathrm{Fib}_n, \mathrm{Fib}_m) &= \gcd(\mathrm{Fib}_n, \mathrm{Fib}_{n+(m-n)}) \\ &= \gcd(\mathrm{Fib}_n, \mathrm{Fib}_{n+1}\mathrm{Fib}_{m-n} + \mathrm{Fib}_n\mathrm{Fib}_{m-n-1}) \\ &= \gcd(\mathrm{Fib}_n, \mathrm{Fib}_{n+1}\mathrm{Fib}_{m-n}) \end{aligned}$$

由上文提到的 Fib_{n+1} 与 Fib_n 互质，则有 $\gcd(\mathrm{Fib}_n, \mathrm{Fib}_m) = \gcd(\mathrm{Fib}_n, \mathrm{Fib}_{m-n})$。该式子与辗转相除法相似，得证。

斐波那契数列整除性质：

$$\mathrm{Fib}_n \mid \mathrm{Fib}_m \iff n \le 2 \vee n \mid m$$

斐波那契数列模意义下的循环节

在模意义下，二元组 $(\mathrm{Fib}_n \bmod p, \mathrm{Fib}_{n+1} \bmod p)$ 仅有 p^2 种取值。**因此，当 n 足够大时，必定会出现循环节。**

根据递推式，有 $\mathrm{Fib}_n = \mathrm{Fib}_{n+2} - \mathrm{Fib}_{n+1}$，**也就是说 $\mathrm{Fib}_{n+1}, \mathrm{Fib}_{n+2}$ 的前面必为 Fib_n。**

根据这两个事实我们可以得知，**斐波那契数列在模意义下必有循环节，并且一定是纯循环的（循环节的起点为 $\mathrm{Fib}_0, \mathrm{Fib}_1$）。**

最小循环节（皮萨诺周期）：定义斐波那契数列模 p 意义下的最小循环节 $\pi(p)$ 为最小的正整数 m，使得 $\mathrm{Fib}_m \equiv 0 \pmod p$ 且 $\mathrm{Fib}_{m+1} \equiv 1 \pmod p$。

注意：周期一定是最小循环节的倍数，即

$$\begin{cases} \mathrm{Fib}_k \equiv 0 \pmod p \\ \mathrm{Fib}_{k+1} \equiv 1 \pmod p \end{cases} \iff \pi(p) \mid k$$

很多时候我们只需求出周期即可。

最小循环节的上界：**斐波那契数列 $\bmod m$ 的最小正周期不超过 $6m$，即 $\pi(m) \le 6m$。**

求斐波那契数列循环节（结论法）：

- 质数情景：
  - $\pi(2) = 3$，$\pi(5) = 20$。
  - 当 p 为奇质数且 $p \equiv \pm 1 \pmod 5$ 时，$\pi(p) \mid p - 1$。
  - 当 p 为奇质数且 $p \equiv \pm 2 \pmod 5$ 时，$\pi(p) \mid 2p + 2$。
- 一般情景：
  - 对于质数 p 以及任意正整数 k，有 $\pi(p^k) \mid p^{k-1}\pi(p)$。
  - 对于 $m = \prod p_i^{c_i}$，有 $\pi(m) = \mathrm{lcm}\{\pi(p_i^{c_i})\}$，同时有 $\pi(m) \mid \mathrm{lcm}\{p_i^{c_i-1}\pi(p_i)\}$。

在一般应用中，通常只需要求出一个循环节即可。

对于质数 p，记

$$g(p) = \begin{cases} 3 & p = 2 \\ 20 & p = 5 \\ p - 1 & p \equiv \pm 1 \pmod 5 \\ 2p + 2 & p \equiv \pm 2 \pmod 5 \end{cases}$$

然后将 m 质因数分解成 $\prod p_i^{c_i}$，使用 $\mathrm{lcm}\{p_i^{c_i-1} g(p_i)\}$ 作为循环节即可（注意这并不是最小循环节，这个值在多数情况下已经够用了）。

求斐波那契数列循环节（矩阵 BSGS）：

> **适用场景：最小循环节的上界确定，转移矩阵可逆。**

考虑斐波那契数列的矩阵表示

$$\begin{bmatrix}\mathrm{Fib}_n & \mathrm{Fib}_{n-1}\end{bmatrix} = \begin{bmatrix}1 & 0\end{bmatrix} \times \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-1}$$

记转移矩阵 $A = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$，那么循环节即为满足 $A^m \equiv I \pmod p$ 的正整数 m。

在转移矩阵可逆的情况下，套用 BSGS 的流程：取 S 表示上界的根号，设 $m = iS - j$（其中 $i \geq 1$ 且 $0 \leq j < S$），则有

$$A^{iS} \times A^{-j} \equiv I \pmod p$$

$$A^{iS} \equiv A^{j} \pmod p$$

预处理 $0 \leq j < S$ 的 A^j 存入哈希表，每次在哈希表中查询 A^{iS} 即可。

时间复杂度 $\mathcal{O}(\sqrt{p})$。

求斐波那契数列循环节（生日悖论随机化）：

若存在两个数 $0 \leq i, j \leq 6p$ $(i \neq j)$，满足 $(\mathrm{Fib}_i, \mathrm{Fib}_{i+1}) \equiv (\mathrm{Fib}_j, \mathrm{Fib}_{j+1}) \pmod p$，则 $|i-j|$ 为一个循环节（注意这并不是最小循环节）。

每次新随机一个数 x，将二元组 $(\mathrm{Fib}_x, \mathrm{Fib}_{x+1})$ 存入哈希表中，并查询哈希表中有没有与之相同的 $(\mathrm{Fib}_y, \mathrm{Fib}_{y+1})$。

发现这与生日悖论的形式很像，于是时间复杂度正确。

> 生日悖论：在 $[1, n]$ 中随机选择正整数，选出两个相同数的期望轮数为 $\mathcal{O}(\sqrt{n})$ 级别（近似 $\sqrt{\frac{\pi n}{2}}$）。

实际取数过程可能需要在 $[0, 12p]$ 中随机以保证期望次数。求 Fib_i 需要用到“光速幂”的技巧。

普通生成函数

普通生成函数（OGF）：

$$F(x) = \sum_n a_n x^n$$

OGF 乘法：

$$F(x)G(x) = \sum_n \left(\sum_{i=0}^{n} a_i b_{n-i} \right) x^n$$

OGF 常用于解决无标号的计数问题（例如：优化计数类背包）。

常见序列 OGF 封闭形式：

- $\{1, 1, 1, \dots\} \xrightarrow{\text{OGF}} \frac{1}{1-x}$
- $\{1, p, p^2, \dots\} \xrightarrow{\text{OGF}} \frac{1}{1-px}$
- $\{\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \dots\} \xrightarrow{\text{OGF}} (1+x)^n$
- $\{\binom{n}{0}, \binom{n+1}{1}, \binom{n+2}{2}, \dots\} \xrightarrow{\text{OGF}} \frac{1}{(1-x)^{n+1}}$
- $\{0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots\} \xrightarrow{\text{OGF}} \ln \frac{1}{1-x} = -\ln(1-x)$
- $\{1, 1, \frac{1}{2!}, \frac{1}{3!}, \frac{1}{4!}, \dots\} \xrightarrow{\text{OGF}} e^x$

指数生成函数

指数生成函数（EGF）：

$$F(x) = \sum_n a_n \frac{x^n}{n!}$$

EGF 乘法：

$$F(x)G(x) = \sum_n \left(\sum_{i=0}^{n} \binom{n}{i} a_i b_{n-i} \right) \frac{x^n}{n!}$$

EGF 常用于解决有标号的计数问题。

常见序列 EGF 封闭形式：

- $\{1, 1, 1, \dots\} \xrightarrow{\text{EGF}} e^x$
- $\{1, p, p^2, \dots\} \xrightarrow{\text{EGF}} e^{px}$
- $\{1, n, n^2, n^3, n^4, \dots\} \xrightarrow{\text{EGF}} (1+x)^n$

EGF 中多项式 exp 的组合意义：

回顾一下 exp 的式子（在这里我们默认 $[x^0]f(x) = 0$）

$$\exp f(x) = \sum_{i=0}^{\infty} \frac{f^i(x)}{i!}$$

如果 $f(x)$ 描述的是单个组件的 EGF，那么 $f^k(x)$ 得到的就是由 k 个组件构成的有序序列的 EGF，但是如果我们将这 k 个组件看成一个集合，就要除以 $k!$ 来消除顺序的影响。

所以 $\frac{f^k(x)}{k!}$ 的含义是由 k 个组件构成的无序集合的 **EGF**。因此，$\exp f(x)$ 的含义是由若干个组件构成的无序集合的 **EGF**。

符号化方法

~~待填坑。~~

图论计数

有标号无向连通图计数

算法一：分治 NTT

设 f_i 表示点数为 i 的有标号简单无向连通图个数，设 g_i 表示点数为 i 的有标号简单无向图个数，显然 $g_i = 2^{\binom{i}{2}}$。

考虑容斥。枚举点 1 所在的连通块大小，有递推式

$$f_1 = 1$$

$$f_i = g_i - \sum_{j=1}^{i-1} \binom{i-1}{j-1} f_j g_{i-j}$$

化简式子，得

$$\frac{f_i}{(i-1)!} = \frac{g_i}{(i-1)!} - \sum_{j=1}^{i-1} \frac{f_j}{(j-1)!} \frac{g_{i-j}}{(i-j)!}$$

使用分治 NTT，时间复杂度 $\mathcal{O}(n \log^2 n)$。

> 另一种理解方式：设 $F_i = \frac{f_i}{(i-1)!}, G_i = \frac{g_i}{(i-1)!}, H_i = \frac{g_i}{i!}$，可以将该递推式看成 $G = F \times H$，于是 $F = G \times H^{-1}$。
> 使用多项式求逆，时间复杂度 $\mathcal{O}(n \log n)$。

算法二：多项式 ln

设 $F(x)$ 是 f_i 的 EGF，$G(x)$ 是 g_i 的 EGF。由 EGF 中多项式 exp 的组合意义，有 $G(x) = e^{F(x)}$。所以

$$F(x) = \ln G(x)$$

使用多项式 ln，时间复杂度 $\mathcal{O}(n \log n)$。

0x33 有标号无向连通图计数.cpp

```cpp
// 有标号无向连通图计数
poly GraphCount(int n) {
    std::vector<int> fact(n + 1), facv(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++) {
        fact[i] = 1ll * fact[i - 1] * i % mod;
    }
    facv[n] = qpow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i --) {
        facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
    }

    poly g(n + 1);
    for (int i = 0; i <= n; i ++) {
        g[i] = 1ll * qpow(2, 1ll * i * (i - 1) / 2, mod) * facv[i] % mod;
    }
    poly f = g.ln(n + 1);
    for (int i = 0; i <= n; i ++) {
        mul(f[i], fact[i]);
    }
    return f;
}
```

有标号无向欧拉图计数

欧拉图的条件：所有节点的度数均为偶数，并且连通。

设 f_i 表示点数为 i 的有标号简单无向欧拉图个数，设 g_i 表示点数为 i 的有标号度数均为偶数的简单图个数。

考虑求 g_i，先让前 $i-1$ 个点任意连边，最后所有奇点的度数都通过与 i 连边调节成偶数。于是 $g_i = 2^{\binom{i-1}{2}}$。

剩余部分同“有标号无向连通图计数”，时间复杂度 $\mathcal{O}(n \log n)$。

0x33 有标号无向欧拉图计数.cpp

```cpp
// 有标号无向欧拉图计数
poly GraphCount(int n) {
    std::vector<int> fact(n + 1), facv(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++) {
        fact[i] = 1ll * fact[i - 1] * i % mod;
    }
```

```
    facv[n] = qpow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i --) {
        facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
    }

    poly g(n + 1);
    g[0] = 1;
    for (int i = 1; i <= n; i ++) {
        g[i] = 1ll * qpow(2, 1ll * (i - 1) * (i - 2) / 2, mod) * facv[i] %
        ↪  mod;
    }
    poly f = g.ln(n + 1);
    for (int i = 0; i <= n; i ++) {
        mul(f[i], fact[i]);
    }
    return f;
}
```

有标号无向二分图计数

设 f_i 表示点数为 i 的有标号简单无向二分图个数，设 g_i 表示点数为 i 的对节点进行 2 染色，满足同色节点不存在边的图（二分染色图）个数。

考虑求 g_i，枚举黑色节点个数 j，则有

$$g_i = \sum_{j=0}^{i} \binom{i}{j} 2^{j(i-j)}$$

$$g_i = \sum_{j=0}^{i} \binom{i}{j} 2^{\binom{i}{2}-\binom{j}{2}-\binom{i-j}{2}}$$

$$\frac{g_i}{i! \cdot 2^{\binom{i}{2}}} = \sum_{j=0}^{i} \frac{1}{j! \cdot 2^{\binom{j}{2}}} \cdot \frac{1}{(i-j)! \cdot 2^{\binom{i-j}{2}}}$$

> 这里用到了 $ij = \binom{i+j}{2} - \binom{i}{2} - \binom{j}{2}$。

可以使用卷积 $\mathcal{O}(n \log n)$ 求出 g_i。

对于二分染色图的每一个连通块，确定了最小点的颜色就确定了整个连通块的颜色，而最小点有黑白两种选法。

则二分染色图可以由两个组件构成，第一个组件是最小点颜色为黑色的连通块集合，第二个组件是最小点颜色为白色的连通块集合。而两个组件内部均是二分图。

于是设 $F(x)$ 是 f_i 的 EGF，$G(x)$ 是 g_i 的 EGF，则有

$$F(x) \times F(x) = G(x)$$

$$F(x) = \sqrt{G(x)}$$

使用多项式开方，时间复杂度 $\mathcal{O}(n \log n)$。

0x33 有标号无向二分图计数.cpp

```
// 有标号无向二分图计数
poly GraphCount(int n) {
    std::vector<int> fact(n + 1), facv(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++) {
        fact[i] = 1ll * fact[i - 1] * i % mod;
    }
    facv[n] = qpow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i --) {
        facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
    }

    poly a(n + 1);
    for (int i = 0, i2 = qpow(2, mod - 2, mod); i <= n; i ++) {
        a[i] = 1ll * facv[i] * qpow(i2, 1ll * i * (i - 1) / 2, mod) % mod;
    }
    poly g = (a * a).modxk(n + 1);
    for (int i = 0; i <= n; i ++) {
        mul(g[i], qpow(2, 1ll * i * (i - 1) / 2, mod));
    }
    poly f = g.sqrt(n + 1);
    for (int i = 0; i <= n; i ++) {
        mul(f[i], fact[i]);
    }
    return f;
}
```

有标号 DAG 计数

设 f_i 表示点数为 i 的有标号 **DAG（不保证弱连通）**个数。

将 DAG 视作一个分层图，考虑容斥，钦定拓扑序第一层（零入度点）的个数。

$$f_i = \sum_{j=1}^{i} (-1)^{j+1} \binom{i}{j} 2^{j(i-j)} f_{i-j}$$

至此，可以做到 $\mathcal{O}(n^2)$。

进一步

$$\frac{f_i}{i!\cdot 2^{\binom{i}{2}}}=\sum_{j=1}^{i}\frac{(-1)^{j+1}}{j!\cdot 2^{\binom{j}{2}}}\cdot\frac{f_{i-j}}{(i-j)!\cdot 2^{\binom{i-j}{2}}}$$

设 $F_i=\frac{f_i}{i!\cdot 2^{\binom{i}{2}}},G_j=\frac{(-1)^{j+1}}{j!\cdot 2^{\binom{j}{2}}}$ $(G_0=0)$，则有

$$F=G\times F+1$$

$$F=\frac{1}{1-G}$$

使用多项式求逆，时间复杂度 $\mathcal{O}(n\log n)$。

0x33 有标号 DAG 计数.cpp

```cpp
// 有标号 DAG 计数
poly GraphCount(int n) {
    std::vector<int> fact(n + 1), facv(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++) {
        fact[i] = 1ll * fact[i - 1] * i % mod;
    }
    facv[n] = qpow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i --) {
        facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
    }

    poly g(n + 1);
    g[0] = 0;
    for (int i = 1, i2 = qpow(2, mod - 2, mod); i <= n; i ++) {
        g[i] = 1ll * facv[i] * qpow(i2, 1ll * i * (i - 1) / 2, mod) % mod;
        if ((i + 1) & 1) neg(g[i]);
    }
    poly f = (poly{1} - g).inv(n + 1);
    for (int i = 0; i <= n; i ++) {
        mul(f[i], fact[i]), mul(f[i], qpow(2, 1ll * i * (i - 1) / 2, mod));
    }
    return f;
}
```

有标号弱连通 DAG 计数

设 f_i 表示点数为 i 的有标号 **DAG（不保证弱连通）**个数，设 h_i 表示点数为 i 的有标号弱连通 **DAG** 个数。f_i 的求解详见“有标号 DAG 计数”。

设 $F(x)$ 是 f_i 的 EGF，$H(x)$ 是 h_i 的 EGF。由 EGF 中多项式 exp 的组合意义，有 $F(x)=e^{H(x)}$。所以

$$H(x)=\ln F(x)$$

使用多项式 ln，时间复杂度 $\mathcal{O}(n\log n)$。

0x33 有标号弱连通 DAG 计数.cpp

```cpp
// 有标号弱连通 DAG 计数
poly GraphCount(int n) {
    std::vector<int> fact(n + 1), facv(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++) {
        fact[i] = 1ll * fact[i - 1] * i % mod;
    }
    facv[n] = qpow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i --) {
        facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
    }

    poly g(n + 1);
    g[0] = 0;
    for (int i = 1, i2 = qpow(2, mod - 2, mod); i <= n; i ++) {
        g[i] = 1ll * facv[i] * qpow(i2, 1ll * i * (i - 1) / 2, mod) % mod;
        if ((i + 1) & 1) neg(g[i]);
    }
    poly f = (poly{1} - g).inv(n + 1);
    for (int i = 0; i <= n; i ++) {
        mul(f[i], qpow(2, 1ll * i * (i - 1) / 2, mod));
    }
    f = f.ln(n + 1);
    for (int i = 0; i <= n; i ++) {
        mul(f[i], fact[i]);
    }
    return f;
}
```

有标号强连通图计数

设 f_i 表示点数为 i 的有标号强连通图个数，设 $b_{i,j}$ 表示 i 个点形成 j 个独立的强连通图的个数，设 h_i 表示点数为 i 的任意有向图个数。显然 $h_i=2^{i(i-1)}$。

则有

$$h_i=\sum_{j=1}^{i}\binom{i}{j}h_{i-j}\times 2^{j(i-j)}\times\sum_{k=1}^{j}(-1)^{k+1}b_{j,k}$$

对于任意点数为 i 的有向图，将每个强连通分量缩成一个点，会得到一张 DAG。这其中必定有出度为 0 的点。

钦定 k 个强连通分量出度为 0，这 k 个强连通分量里包含了 j 个点。剩下的 $i-j$ 个点随意连接。

设 $g_j=\sum_{k=1}^{j}(-1)^{k+1}b_{j,k}$，则有

$$g_i=h_i-\sum_{j=1}^{i-1}\binom{i}{j}2^{j(i-j)}h_{i-j}g_j$$

$$\frac{h_i}{i!\cdot 2^{\binom{i}{2}}}=\sum_{j=1}^{i}\frac{g_j}{j!\cdot 2^{\binom{j}{2}}}\cdot\frac{h_{i-j}}{(i-j)!\cdot 2^{\binom{i-j}{2}}}$$

设 $H_i=\frac{h_i}{i!\cdot 2^{\binom{i}{2}}},G_i=\frac{g_i}{i!\cdot 2^{\binom{i}{2}}}$，则有

$$H=G\times H+1$$

$$G=1-\frac{1}{H}$$

至此，可以使用多项式求逆求出 G。

再根据 g 的定义，考虑 g 的转移，有

$$g_i=f_i-\sum_{j=1}^{i-1}\binom{i-1}{j-1}f_jg_{i-j}$$

$$f_i=g_i+\sum_{j=1}^{i-1}\binom{i-1}{j-1}f_jg_{i-j}$$

$$\frac{f_i}{(i-1)!}=\frac{g_i}{(i-1)!}+\sum_{j=1}^{i-1}\frac{f_j}{(j-1)!}\cdot\frac{g_{i-j}}{(i-j)!}$$

设 $F_i=\frac{f_i}{(i-1)!},A_i=\frac{g_i}{(i-1)!},B_i=\frac{g_i}{i!}$，则有

$$F=A+F\times B$$

$$F=\frac{A}{1-B}$$

0x33 有标号强连通图计数.cpp

```cpp
// 有标号强连通图计数
poly GraphCount(int n) {
    std::vector<int> fact(n + 1), facv(n + 1);
    fact[0] = 1;
    for (int i = 1; i <= n; i ++) {
        fact[i] = 1ll * fact[i - 1] * i % mod;
    }
    facv[n] = qpow(fact[n], mod - 2, mod);
    for (int i = n - 1; i >= 0; i --) {
        facv[i] = 1ll * facv[i + 1] * (i + 1) % mod;
    }

    poly h(n + 1);
    h[0] = 1;
    for (int i = 1; i <= n; i ++) {
        h[i] = 1ll * qpow(2, 1ll * i * (i - 1) / 2, mod) * facv[i] % mod;
    }
    poly g2 = poly{1} - h.inv(n + 1);
    for (int i = 1; i <= n; i ++) {
        mul(g2[i], qpow(2, 1ll * i * (i - 1) / 2, mod));
    }
    poly g1 = g2;
    for (int i = 1; i <= n; i ++) {
        mul(g1[i], i);
    }
    poly f = (g1 * (poly{1} - g2).inv(n + 1)).modxk(n + 1);
    for (int i = 1; i <= n; i ++) {
        mul(f[i], fact[i - 1]);
    }
    return f;
}
```

Misc

Raney 引理

Raney 引理：若整数序列 $x_1,\dots,x_m$ 满足 $\sum_{i=1}^{m}x_i=1$，则它的 m 个循环移位中恰好有 1 个满足前缀和均为正数。

唯一合法的循环移位起点，即为最靠后的“前缀和最小点”。

广义 Raney 引理：若整数序列 $x_1,\dots,x_m$ 满足 $x_i\le 1$ 且 $\sum_{i=1}^{m}x_i=n$，则它的 m 个循环移位中恰好有 n 个满足前缀和均为正数。

0x34 数学 初等数论–质数

质数

质数：若一个正整数 n $(n>1)$ 无法被除了 1 和它自身以外的任何正整数整除，则称该数为质数（或素数）。否则称该数为合数。特别地，1 既不是质数也不是合数。

质数分布函数：$\pi(n)$ 表示 $1\sim n$ 中的质数个数。当 n 足够大时，$\pi(n)\sim\frac{n}{\ln n}$。

唯一分解定理：任意一个大于 1 的正整数 n，都可以唯一分解成有限个质数的乘积，记作

$$n=p_1^{c_1}p_2^{c_2}\dots p_t^{c_t}$$

其中 p_i 均为质数且满足 $p_1<p_2<\dots<p_t$，c_i 均为正整数。

质因数的根号分治：一个正整数 n，大于 $\sqrt{n}$ 的质因子最多只有 1 个。

阶乘的质因数分解

对于阶乘 $n!=1\times2\times\dots\times n$，由于每一项均为 $[1,n]$ 中的正整数，故分解出的质因数也均为 $[1,n]$ 中的质数。

筛出 $1\sim n$ 中的所有质数，**对于每个质数 p，考虑 $n!$ 中一共包含了多少个质因子 p。**

共有大约 $\frac{n}{\ln n}$ 个质数，每个质数 p 的求解需要 $\log_p n$ 的开销，故时间复杂度 $\mathcal{O}(n)$。

阶乘 $n!$ 中包含质数 p 的次数：

$$\sum_{i=1}^{\lfloor\log_p n\rfloor}\left\lfloor\frac{n}{p^i}\right\rfloor$$

简单推导：设 c_i 表示 i 包含的质因子 p 个数，则阶乘 $n!$ 包含 p 的个数为 $\sum_{j=1}^{\infty}\sum_{i=1}^{n}j[c_i=j]$，进一步得 $\sum_{j=1}^{\infty}\sum_{i=1}^{n}[c_i\geq j]$，不难发现其中 $\sum_{i=1}^{n}[c_i\geq j]=\left\lfloor\frac{n}{p^j}\right\rfloor$。

阶乘 $n!$ 中包含合数 m 的次数：将 m 唯一分解成 $m=p_1^{c_1}p_2^{c_2}\dots p_t^{c_t}$ 的形式。

记 $v_p(n)=\sum_{i=1}^{\lfloor\log_p n\rfloor}\left\lfloor\frac{n}{p^i}\right\rfloor$，答案为

$$\min_{1\leq i\leq t}\left\lfloor\frac{v_{p_i}(n)}{c_i}\right\rfloor$$

埃氏筛

注意到，任意整数 x $(x>1)$ 的倍数 $2x,3x,\dots$ 都不是质数。

从 2 到 n 依次扫描每一个数 x，若 x 尚未被标记，则说明 x 为质数，并且将其倍数 $2x,3x,\dots,\lfloor\frac{n}{x}\rfloor x$ 均标记为合数。

时间复杂度 $\mathcal{O}(n\log\log n)$。

在开 O2 优化的情况下，`std::bitset` 连续读写效率较高，可以用来优化埃氏筛。

> 埃氏筛的时间复杂度证明：**相当于是证明 $1\sim n$ 中所有质数的倒数之和为 $\mathcal{O}(\log\log n)$ 级别。**
>
> 由质数分布函数，可以推测第 n 个质数的大小为 $\Theta(n\log n)$，于是有
>
> $$\begin{aligned}\sum_{k=1}^{\pi(n)}\frac{1}{p_k}&=\mathcal{O}\left(\sum_{k=2}^{\pi(n)}\frac{1}{k\log k}\right)\\&=\mathcal{O}\left(\int_2^{\pi(n)}\frac{1}{x\log x}\mathrm{dx}\right)\\&=\mathcal{O}(\log\log\pi(n))\\&=\mathcal{O}(\log\log n)\end{aligned}$$

0x34 埃氏筛.cpp

```cpp
// 埃氏筛
std::bitset<MaxV> vis;
void sieve(int n) {
    for (int i = 2; i <= n; i ++) {
        if (vis[i]) {
            continue;
        }
        for (int j = 2; j <= n / i; j ++) {
            vis[i * j] = 1;
        }
    }
}
```

线性筛

为了避免埃氏筛重复标记合数，在线性筛中，我们希望确定一个合数的方式能够唯一。

从 2 到 n 依次扫描每一个数 x，在标记合数时，**我们只在 x 的基础上乘一个质因子 p，并且保证 p 是 xp 的最小质因子。**这样做，每个合数只会被自己的最小质因子标记一次。并且顺带还能筛出每个数的最小质因子。

时间复杂度 $\mathcal{O}(n)$。

由于线性筛每次只扩展一个最小质因子的结构，可以额外处理很多信息（例如筛出积性函数）。

0x34 线性筛.cpp

```cpp
// 线性筛
int primeCount, prime[MaxV], fac[MaxV];
void sieve(int n) {
    for (int i = 2; i <= n; i ++) {
        if (!fac[i]) {
            prime[++ primeCount] = i, fac[i] = i;
        }
        for (int j = 1; j <= primeCount; j ++) {
            if (prime[j] > fac[i] || prime[j] > n / i) break;
            fac[i * prime[j]] = prime[j];
        }
    }
}
```

狄利克雷前缀和

狄利克雷前缀和：已知 $a_1,\dots,a_n$，求出 $b_1,\dots,b_n$，其中 $b_i=\sum_{j|i}a_j$。

记 $j=\prod p_k^{\alpha_k}, i=\prod p_k^{\beta_k}$，则 a_j 对 b_i 有贡献当且仅当：对于所有的 k 均有 $\alpha_k\le\beta_k$。类似地，运用高维前缀和即可。

```cpp
// 注意：要先枚举质数 prime[j]
for (int j = 1; j <= primeCount; j ++) {
    for (int i = 1; i <= n / prime[j]; i ++) {
        f[i * prime[j]] += f[i];
    }
}
```

狄利克雷后缀和：已知 $a_1,\dots,a_n$，求出 $b_1,\dots,b_n$，其中 $b_i=\sum_{i|j}a_j$。

```cpp
// 注意：要先枚举质数 prime[j]
for (int j = 1; j <= primeCount; j ++) {
    for (int i = n / prime[j]; i >= 1; i --) {
        f[i] += f[i * prime[j]];
    }
}
```

狄利克雷倒前缀和：已知 $b_1,\dots,b_n$，反推 $a_1,\dots,a_n$，其中 $b_i=\sum_{j|i}a_j$。

```cpp
// 注意：要先枚举质数 prime[j]
for (int j = 1; j <= primeCount; j ++) {
    for (int i = n / prime[j]; i >= 1; i --) { // 顺序与正变换相反
        f[i * prime[j]] -= f[i];
    }
}
```

狄利克雷倒后缀和：已知 $b_1,\dots,b_n$，反推 $a_1,\dots,a_n$，其中 $b_i=\sum_{i|j}a_j$。

```cpp
// 注意：要先枚举质数 prime[j]
for (int j = 1; j <= primeCount; j ++) {
    for (int i = 1; i <= n / prime[j]; i ++) { // 顺序与正变换相反
        f[i] -= f[i * prime[j]];
    }
}
```

Miller–Rabin 素性测试

回顾一些内容：

费马小定理：设 p 是素数，对于任意整数 a（a 不能是 p 的倍数），满足 $a^{p-1}\equiv 1 \pmod p$。

Fermat 伪素数：若满足 $a^{n-1}\equiv 1 \pmod n$，但 n 不是素数，则称 n 为以 a 为底的 Fermat 伪素数。

对于任何固定的基底 a，都存在无穷多个的 Fermat 伪素数。

Fermat 素性测试：不断地选取在 $[2,n-1]$ 中的基底 a，并检验是否每次都满足 $a^{n-1}\equiv 1 \pmod n$。

即使检查了所有与 n 互质的基底 a，依然无法保证 n 是素数。这样的数 n 称为 Carmichael 数，有无穷多个。

二次探测定理：设 p 是奇素数，则同余方程 $x^2\equiv 1 \pmod p$ 的解为 $x\equiv 1 \pmod p$ 或 $x\equiv p-1 \pmod p$。

Miller-Rabin 素性测试：Miller-Rabin 素性测试，结合了费马小定理和二次探测定理，是一种概率性素性测试（可以放心使用）。

先将 $a^{n-1}\equiv 1 \pmod n$ 中的指数 $n-1$，分解成 $n-1=u\times 2^t$ 的形式。

每次随机一个基底 a，进行如下的判断：

- 先使用快速幂求出 $v=a^u \bmod n$。如果 $v=1$ 或 $v=n-1$，则该轮测试通过。
- 然后对这个值进行 $t-1$ 次平方操作，途中如果该值等于 $n-1$，则该轮测试通过。
- 否则该轮测试不通过。

在不考虑乘法的开销时，对 n 进行 k 轮测试的时间复杂度为 $\mathcal{O}(k\log n)$。

可以证明，奇合数 n $(n>9)$ 通过随机选取的一个基底的概率不超过 $\frac{1}{4}$。**因此随机选取 k 个基底后，仍然误判的概率不超过 $\frac{1}{4^k}$。**随机选取 10 次较为保险。

对于 $[2,2^{64})$ 范围内的数，工业上通常选用 $\{2,325,9375,28178,450775,9780504,1795265022\}$ 作为检验的基底。为了方便也可以选择前 12 个质数作为基底。

Pollard–Rho 质因数分解

Pollard-Rho：用于找出合数 n 的某一个非平凡因数 d（非平凡因数：非 1 且非 n 的因数）。

若一个正整数 n 为合数，那么 n 的最小质因子 p 满足 $p\le\sqrt{n}$。

如果随机生成 $\mathcal{O}(\sqrt{p})$ 个正整数，由生日悖论，可能存在两个正整数 a,b 满足 $a\equiv b\pmod p$，此时 $|a-b|$ 是 p 的倍数，于是查询 $\gcd(|a-b|,n)$ 即可得到 n 的一个因子。

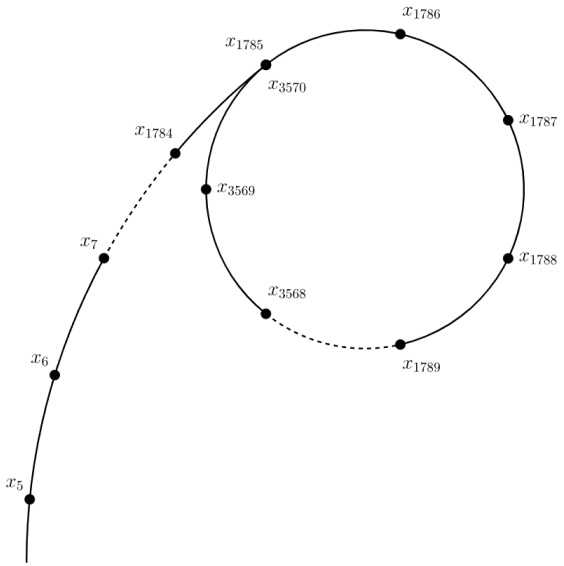
> 生日悖论：在 $[1,n]$ 中随机选择正整数，选出两个相同数的期望轮数为 $\mathcal{O}(\sqrt{n})$ 级别（近似 $\sqrt{\frac{\pi n}{2}}$）。

但这样做，枚举数对的时间复杂度还是太高。

记映射 $f(x)=(x^2+c)\bmod n$，其中 c 是一个随机选取的常数（$c\neq 0,-2$）。Pollard 构造了这样的一个伪随机序列 $\{x_i\}$，满足 $x_i=f(x_{i-1})$。

考虑序列 $\{x_i \bmod p\}$，由于 x_i 仅仅与 x_{i-1} 有关，且 $x_i \bmod p\in[0,p)$，**所以 $\{x_i \bmod p\}$ 是一个混循环序列。**

建图后形似 ρ 形（rho），**由生日悖论可得尾长与环长均为 $\mathcal{O}(\sqrt{p})$ 级别。**



我们发现，**若 $x\equiv y\pmod p$，必有 $f(x)\equiv f(y)\pmod p$。由此可得在模 p 的环上，只要某个相距 d 步的点对满足条件，则所有相距 d 步的点对均满足条件。**所以我们每次最好要检查不同的距离。

Brent 倍增迭代：初始有两个点 a,b，在第 k 轮，a 原地不动，b 不断地向前共计 2^k 步，最后再让 a 直接跳到 b 的位置。b 的每一步移动，我们都计算 $d=\gcd(|a-b|,n)$，当第一次出现 $d>1$ 时

- 若 $1<d<n$，则我们已经找到了一个非平凡因数，直接返回 d。
- 若 $d=n$，则说明我们已经完整地跑完了一个环，继续在环上跑也没有意义。此时我们应该换个映射，或者换一个初始值继续做。

进一步，我们想要降低求解 gcd 的次数。注意到，如果 $\gcd(a,n)>1$，则对于任意正整数 b 均有 $\gcd(ab \bmod n,n)=\gcd(ab,n)>1$。

也就是说，如果 $\gcd\left(\prod|x_i-x_j| \bmod n,n\right)>1$，那么其中必有一对 x_i,x_j 满足 $\gcd(|x_i-x_j|,n)>1$。

于是我们每 k 对 a,b 就计算一次 gcd。时间复杂度 $\mathcal{O}(\sqrt{p}+k^{-1}\sqrt{p}\log n)$，当 k 与 $\log n$ 大致相等时，**可以得到 $\mathcal{O}(\sqrt{p})$ 的理论复杂度。**具体实现中，一般取 $k=128$。

Pollard-Rho 质因数分解：如果要对 n 进行质因数分解，先使用 Miller-Rabin 判断 n 是否是质数，若 n 为合数，则使用 Pollard-Rho 找出 n 的一个非平凡因数 d，得到两个规模分别为 d 和 $\frac{n}{d}$ 的子问题，递归进行求解。

0x34 素性测试与质因数分解（Miller-Rabin + Pollard-Rho）.cpp

```cpp
// 素性测试与质因数分解
template <class T, auto qmul>
struct primeTools {
    constexpr T qpow(T a, T b, T p) {
        T ans = 1;
        for (; b; b >>= 1) {
            if (b & 1) ans = qmul(ans, a, p);
            a = qmul(a, a, p);
        }
        return ans;
    }

    std::mt19937_64 mtrand{std::random_device{}()};
    T rand(T l, T r) {
        std::uniform_int_distribution<T> range(l, r);
        return range(mtrand);
    }

    // 素性测试 Miller-Rabin
    bool isPrime(T n) {
        assert(n > 1);
        if (n % 2 == 0) return n == 2;
        if (n % 3 == 0) return n == 3;

        T t = __builtin_ctzll(n - 1);
        T u = (n - 1) >> t;

        int test = 6;
        while (test --) {
            T a = rand(2, n - 1);
            T v = qpow(a, u, n);
            if (v == 1 || v == n - 1) {
                continue;
            }
            for (int i = 1; i < t; i ++) {
                v = qmul(v, v, n);
                if (v == n - 1) {
                    break;
                }
            }
            if (v != n - 1) {
                return 0;
            }
        }
        return 1;
    }

    T walk(T n) {
        T c = rand(1, n - 1);
        T x = rand(1, n - 1), y, v, d;
        for (int k = 1;; k <<= 1) {
            y = x, v = 1;
            for (int i = 1; i <= k; i ++) {
                x = (qmul(x, x, n) + c) % n;
                v = qmul(v, x - y + n, n);
                if (i % 127 == 0 || i == k) {
                    d = std::gcd(v, n);
                    if (d > 1) return d;
                }
            }
        }
    }
    // 质因数分解 Pollard-Rho
    auto primeFactor(T n) {
        std::map<T, int> buc;
        std::function<void(T)> solve = [&] (T n) {
            if (n == 1) {
                return;
            }
            if (isPrime(n)) {
                buc[n] ++;
                return;
            }
            T d = 0;
            while (d = walk(n), d == n);
            solve(d), solve(n / d);
        };
        solve(n);
        return std::vector<std::pair<T, int>>(buc.begin(), buc.end());
    }
};

// primeTools<int, [] (int a, int b, int p) { return 1ll * a * b % p; }> mrpr;
// primeTools<i64, [] (i64 a, i64 b, i64 p) { return i128(a) * b % p; }> mrpr;
// primeTools<i64, [] (i64 a, i64 b, i64 p) {
//     i64 c = static_cast<long double>(a) * b / p + 1e-8;
//     i64 ans = a * b - c * p;
//     if (ans < 0) ans += p;
```

```
//     if (ans >= p) ans -= p;
//     return ans;
// }> mrpr;
```

0x34 数学 初等数论–约数

约数

整除：设 $d, n \in \mathbb{Z}$ 且 $d \neq 0$，如果 $\exists q \in \mathbb{Z}$ 使得 $qd = n$，那么称 d 整除 n（记作 $d \mid n$）。

整除的简单性质：

- $a \mid b \wedge b \mid c \Longrightarrow a \mid c$。
- $a \mid b \wedge a \mid c \Longrightarrow \forall x, y \in \mathbb{Z}, a \mid (xb + yc)$。
- 设 $m \neq 0$，则 $a \mid b \Longrightarrow ma \mid mb$。
- 设 $\gcd(a, m) = 1$，则 $a \mid mb \Longrightarrow a \mid b$。

约数个数：

- n 的约数个数必定不超过 $2\sqrt{n}$。实际上很难达到该上界，可以简单认为在 $10^6 \sim 10^{18}$ 范围内为 $\mathcal{O}(\sqrt[3]{n})$ 级别。
- $1 \sim n$ 的约数个数总和为 $\mathcal{O}(n \log n)$ 级别。

最大公约数与最小公倍数

最大公约数：最大的公共约数（**G**reatest **C**ommon **D**ivisor，简写 GCD）。

最小公倍数：最小的公共倍数（**L**east **C**ommon **M**ultiple，简写 LCM）。

唯一分解角度下的 gcd 与 lcm：设 $a = p_1^{c_1} p_2^{c_2} \dots$，$b = p_1^{d_1} p_2^{d_2} \dots$，则有

$$\gcd(a, b) = p_1^{\min(c_1, d_1)} p_2^{\min(c_2, d_2)} \dots$$
$$\operatorname{lcm}(a, b) = p_1^{\max(c_1, d_1)} p_2^{\max(c_2, d_2)} \dots$$

在唯一分解角度下，**gcd 相当于是对每个指数取 min，lcm 相当于是对每个指数取 max**。

gcd 的简单性质：

- $\gcd(a_1, \dots, a_n) = \gcd(|a_1|, \dots, |a_n|)$。
- $\gcd(a, 0) = \gcd(a, a) = |a|$。
- $\gcd(bq + r, b) = \gcd(r, b)$。
- 设 $\gcd(a, m) = 1$，则 $\gcd(a, mb) = \gcd(a, b)$。
- $\gcd(ma_1, \dots, ma_n) = |m| \gcd(a_1, \dots, a_n)$。

lcm 的简单性质：

- $\operatorname{lcm}(a_1, \dots, a_n) = \operatorname{lcm}(|a_1|, \dots, |a_n|)$。
- $\operatorname{lcm}(a, 1) = \operatorname{lcm}(a, a) = |a|$。
- $\operatorname{lcm}(ma_1, \dots, ma_n) = |m| \operatorname{lcm}(a_1, \dots, a_n)$。

gcd 和 lcm 的常见恒等式

gcd-lcm 恒等式：

$$\gcd(a, b) \operatorname{lcm}(a, b) = ab$$

请注意！该式子扩展到三元以上几乎不成立！

事实上当 $n \geq 3$ 时，$\gcd(a_1, \dots, a_n) \operatorname{lcm}(a_1, \dots, a_n) = a_1 \dots a_n$ 当且仅当 $a_1, \dots, a_n$ 两两互质。

gcd 和 lcm 的结合律：

$$\gcd(a, b, c) = \gcd(\gcd(a, b), c)$$
$$\operatorname{lcm}(a, b, c) = \operatorname{lcm}(\operatorname{lcm}(a, b), c)$$

该式子扩展到任意多元仍然成立。

于是可以使用恒等式 $\operatorname{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$，用来求解 $\operatorname{lcm}(a, b, c)$，以及 $\operatorname{lcm}(a_1, \dots, a_n)$。

gcd 对乘法的分配律：

$$\gcd(a, bc) = \gcd(a, b) \gcd\left(\frac{a}{\gcd(a, b)}, c\right)$$

根据对乘法的分配律计算 $\gcd(a, b_1 \dots b_n)$。每次处理 b_i 时，计算 $t = \gcd(a, b_i)$，然后将 a 除以 t（相当于是将已计算的质因数贡献去掉），答案乘以 t。

gcd 的差分律：

$$\gcd(a_1, a_2, \dots, a_n) = \gcd(a_1, a_2 - a_1, \dots, a_n - a_{n-1})$$

通过差分律，我们可以维护差分数组的 gcd，以实现“区间加”与“区间 gcd”。

斐波那契数列 gcd 恒等式：

$$\gcd(\mathrm{Fib}_n, \mathrm{Fib}_m) = \mathrm{Fib}_{\gcd(n, m)}$$

幂差 gcd 恒等式：**不妨设 $a \geq b$，当 $\gcd(a, b) = 1$ 时，有**

$$\gcd(a^n - b^n, a^m - b^m) = a^{\gcd(n, m)} - b^{\gcd(n, m)}$$

简单推导：

右边整除左边：可以证明 $x-y \mid x^k-y^k$。

> k 次方差公式：$x^k-y^k=(x-y)(x^{k-1}+x^{k-2}y+\cdots+xy^{k-2}+y^{k-1})$。

对于 n,m 的公约数 d 来说，可以证得 $a^d-b^d \mid a^n-b^n$ 与 $a^d-b^d \mid a^m-b^m$。

于是 $a^d-b^d \mid \gcd(a^n-b^n,a^m-b^m)$。所以

$$a^{\gcd(n,m)}-b^{\gcd(n,m)} \mid \gcd(a^n-b^n,a^m-b^m)$$

左边整除右边：

> “降幂引理”：不妨设 $r \geq s$，当 $\gcd(a,b)=1$ 时，有
>
> $$\gcd(a^r-b^r,a^s-b^s) \mid a^{r-s}-b^{r-s}$$
>
> 证明：设 d 为 a^r-b^r,a^s-b^s 的公约数。
>
> 因为 $(a^r-b^r)-a^{r-s}(a^s-b^s)=b^s(a^{r-s}-b^{r-s})$，可以证得 $d \mid b^s(a^{r-s}-b^{r-s})$；同理可得 $d \mid a^s(a^{r-s}-b^{r-s})$。
>
> 又由于 $\gcd(a,b)=1$，于是 $d \mid a^{r-s}-b^{r-s}$，“降幂引理” 得证。

对 $\gcd(a^n-b^n,a^m-b^m)$ 反复应用 “降幂引理”，过程等价于辗转相除法。所以

$$\gcd(a^n-b^n,a^m-b^m) \mid a^{\gcd(n,m)}-b^{\gcd(n,m)}$$

最大公约数常见求法

已有的轮子：`std::gcd(a, b)`。

求法 1：辗转相除法

（也称之为欧几里得算法）

对于 $\forall a,b \in \mathbb{N}$ 且 $b \neq 0$，有

$$\gcd(a,b)=\gcd(b,a \bmod b)$$

- 当 $a<b$ 时，一次迭代后就会转化为 $a \geq b$。
- 当 $a \geq b$ 时，由于 $a \bmod b \leq \frac{a}{2}$，这样的迭代不超过 $2\log_2 a$ 次。

时间复杂度 $\mathcal{O}(\log \max(a,b))$。

使用辗转相除法求解斐波那契数列相邻两项的 gcd，会让该算法达到最坏复杂度。

```cpp
// 辗转相除法
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

求法 2：更相减损术

对于 $\forall a,b \in \mathbb{N}$，不妨设 $a \geq b$，有

$$\gcd(a,b)=\gcd(a,a-b)=\gcd(b,a-b)$$

$$\gcd(2a,2b)=2\gcd(a,b)$$

- 当 $a=b$ 时，$\gcd(a,b)=a$。
- “两个偶数”：$\gcd(a,b)=2\gcd(\frac{a}{2},\frac{b}{2})$。
- “一奇一偶”：不妨设 a 为偶数，$\gcd(a,b)=\gcd(\frac{a}{2},b)$。
- “两个奇数”：不妨设 $a \geq b$，$\gcd(a,b)=\gcd(a-b,b)$。

当至少有一个偶数的时候，每次迭代至少会将 a,b 之一减半。而两个奇数相减会得到偶数。

时间复杂度 $\mathcal{O}(\log \max(a,b))$。

更相减损术常用于计算高精度 gcd（因为 “高精度取模” 复杂度较高，“高精度减法” 以及 “高精除低精” 复杂度较低）。

0x34 gcd（更相减损术）.cpp

```cpp
// 更相减损术
int gcd(int a, int b) {
    if (a == b) return a;
    if (a == 0 || b == 0) return a ^ b;
    int az = __builtin_ctz(a), bz = __builtin_ctz(b), z = az < bz ? az : bz,
     ↪  v = 0;
    a >>= az, b >>= bz;
    while (b) {
        v = b - a;
        a = a < b ? a : b;
        b = v < 0 ? -v : v;
        if (b) b >>= __builtin_ctz(b);
    }
    return a << z;
}
```

求法 3：递推

初值 $\gcd(i,0)=\gcd(0,i)=i$，递推式 $\gcd(i,j)=\gcd(i-j,j)$。

时间复杂度：预处理 $\mathcal{O}(n^2)$，查询 $\mathcal{O}(1)$。

```
for (int i = 0; i <= n; i ++) {
    g[i][0] = g[0][i] = i;
    for (int j = 1; j <= i; j ++) {
        g[i][j] = g[j][i] = g[i - j][j];
    }
}
```

求法 4：基于值域预处理的快速 gcd

引理：任意正整数 n 一定可以被拆分成三个正整数 a,b,c $(a\le b\le c)$ 的乘积，其中 $a,b\le\sqrt{n}$，且 $c\le\sqrt{n}$ 与 $c\in\mathbb{P}$ 至少有一个成立。

证明 1：假设 $c>\sqrt{n}$ 且 $c\notin\mathbb{P}$。因为 $c>\sqrt{n}$ 且 $abc=n$，所以 $ab\le\sqrt{n}$。由于 c 不是质数，设 $c=xy$，显然 x,y 不可能同时 $>\sqrt{n}$，不妨设 $x\le y$，则必有 $x\le\sqrt{n}$。此时 n 可以被拆分成 x,ab,y 的乘积。不断对 y 进行相同的讨论。

由于数值不断递减，迭代到最后 $c\le\sqrt{n}$ 与 $c\in\mathbb{P}$ 至少有一个成立。

证明 2：考虑归纳。当 $n=1$ 时显然成立。当 $n>1$ 时，设 p 为 n 的最小质因数，设 a',b',c' $(a'\le b'\le c')$ 为 $\frac{n}{p}$ 的一组合法拆分。

- 当 $p\le\sqrt[4]{n}$ 时，由于 $a'\le\sqrt[3]{\frac{n}{p}}$，所以 $pa'\le\sqrt[3]{np^2}\le\sqrt{n}$。此时 pa',b',c' 为一组合法拆分。
- 当 $p>\sqrt[4]{n}$ 时，假设 $a'\ge p$，则 $n=pa'b'c'\ge p^4>n$ 矛盾，故 $a'<p$。由于 p 为 n 的最小质因数，所以 $a'=1$。此时 p,b',c' 为一组合法拆分。

预处理：通过线性筛得出 $1\sim n$ 所有数的最小质因数。然后根据证明 2，递推得出 $1\sim n$ 所有数的合法拆分。对于 $i>1$，设 p 为 i 的最小质因数，**将 $\frac{i}{p}$ 的合法拆分中的最小数乘以 p 即可得到 i 的合法拆分**。最后预处理出所有 $0\le x,y\le\sqrt{n}$ 的 $\gcd(x,y)$。

查询：当询问 $\gcd(p,q)$ 时，根据 gcd 对乘法的分配律，依次考虑 q 的合法拆分 a,b,c，设当前考虑到的数为 x。

- 计算 $t=\gcd(p,x)$，具体地：
  - 若 x 为合数（此时 $x\le\sqrt{n}$），则 $\gcd(p,x)=\gcd(x,p\bmod x)$。使用预处理的 gcd 值回答即可。
  - 若 x 为质数，则当且仅当 p 为 x 的倍数时 $\gcd(p,x)=x$，否则 $\gcd(p,x)=1$。
- 然后将 p 除以 t，答案乘以 t。

时间复杂度：预处理 $\mathcal{O}(n)$，查询 $\mathcal{O}(1)$。

0x34 gcd（基于值域预处理的快速 gcd）.cpp

```cpp
// O(n)-O(1) gcd
struct flashGCD {
    int b;
    std::vector<int> prime, fac;
    std::vector<std::array<int, 3>> s;
    std::vector<std::vector<int>> g;

    flashGCD() {}
    flashGCD(int n) {
        init(n);
    }

    void init(int n) {
        b = sqrt(n);
        prime.clear(), fac.assign(n + 1, 0);
        s.assign(n + 1, {});
        g.assign(b + 1, std::vector<int>(b + 1, 0));

        for (int i = 2; i <= n; i ++) {
            if (!fac[i]) {
                prime.push_back(i), fac[i] = i;
            }
            for (int p : prime) {
                if (p > fac[i] || p > n / i) break;
                fac[i * p] = p;
            }
        }

        s[1] = {1, 1, 1};
        for (int i = 2; i <= n; i ++) {
            s[i] = s[i / fac[i]], s[i][0] *= fac[i];
            if (s[i][0] > s[i][1]) {
                std::swap(s[i][0], s[i][1]);
            }
            if (s[i][1] > s[i][2]) {
                std::swap(s[i][1], s[i][2]);
            }
        }

```

```
        for (int i = 0; i <= b; i ++) {
            g[i][0] = g[0][i] = i;
            for (int j = 1; j <= i; j ++) {
                g[i][j] = g[j][i] = g[i - j][j];
            }
        }
    }

    int gcd(int p, int q) {
        if (p == 0 || q == 0) {
            return p ^ q;
        }
        int ans = 1;
        for (int x : s[q]) {
            int t = x <= b ? g[x][p % x] : (p % x ? 1 : x);
            p /= t, ans *= t;
        }
        return ans;
    }
} fg;
```

类欧几里得算法

类欧几里得算法：用来解决一类形如 $\left\lfloor \frac{ai+b}{c} \right\rfloor$ 结构的求和问题。

基本模型为

$$f(a,b,c,n)=\sum_{i=0}^{n}\left\lfloor \frac{ai+b}{c} \right\rfloor$$

首先，可以转化为 $0 \le a,b < c$ 的情况

$$\begin{aligned}
f(a,b,c,n)&=\sum_{i=0}^{n}\left\lfloor \frac{ai+b}{c} \right\rfloor \\
&=\sum_{i=0}^{n}\left\lfloor \frac{\left(\left\lfloor \frac{a}{c} \right\rfloor c+(a \bmod c)\right)i+\left(\left\lfloor \frac{b}{c} \right\rfloor c+(b \bmod c)\right)}{c} \right\rfloor \\
&=\sum_{i=0}^{n}\left(\left\lfloor \frac{a}{c} \right\rfloor i+\left\lfloor \frac{b}{c} \right\rfloor+\left\lfloor \frac{(a \bmod c)i+(b \bmod c)}{c} \right\rfloor\right) \\
&=\frac{n(n+1)}{2}\left\lfloor \frac{a}{c} \right\rfloor+(n+1)\left\lfloor \frac{b}{c} \right\rfloor+f(a \bmod c,b \bmod c,c,n)
\end{aligned}$$

然后，令 $m=\left\lfloor \frac{an+b}{c} \right\rfloor$

$$\begin{aligned}
f(a,b,c,n)&=\sum_{i=0}^{n}\left\lfloor \frac{ai+b}{c} \right\rfloor \\
&=\sum_{i=0}^{n}\sum_{j=0}^{m-1}\left[j<\left\lfloor \frac{ai+b}{c} \right\rfloor\right] \\
&=\sum_{i=0}^{n}\sum_{j=0}^{m-1}\left[j+1\le\frac{ai+b}{c}\right] \\
&=\sum_{i=0}^{n}\sum_{j=0}^{m-1}[cj+c-b-1<ai] \\
&=\sum_{j=0}^{m-1}\sum_{i=0}^{n}\left[\left\lfloor \frac{cj+c-b-1}{a} \right\rfloor<i\right] \\
&=\sum_{j=0}^{m-1}\left(n-\left\lfloor \frac{cj+c-b-1}{a} \right\rfloor\right) \\
&=nm-f(c,c-b-1,a,m-1)
\end{aligned}$$

结合这两步操作，二元组 $(a,c)\to(c,a \bmod c)$，相当于对 (a,c) 进行辗转相除法，故称之为类欧几里得算法。

时间复杂度 $\mathcal{O}(\log \min(a,c))$。

0x34 类欧几里得算法.cpp

```
// 类欧几里得算法（要求 a, b >= 0 以及 c > 0）
i64 euclid(i64 a, i64 b, i64 c, i64 n) {
    if (a == 0) {
        return (n + 1) * (b / c);
    }
    if (a >= c || b >= c) {
        return euclid(a % c, b % c, c, n) + (n * (n + 1) / 2) * (a / c) + (n
        ↪  + 1) * (b / c);
    }
    i64 m = (a * n + b) / c;
    if (m == 0) {
        return 0;
    }
    return n * m - euclid(c, c - b - 1, a, m - 1);
}
```

```cpp
// 类欧几里得算法
i64 euclid_norm(i64 a, i64 b, i64 c, i64 n) {
    if (c < 0) {
        a = -a, b = -b, c = -c;
    }
    i64 sum = 0;
    if (a < 0) {
        i64 t = (a - c + 1) / c;
        a -= t * c, sum += (n * (n + 1) / 2) * t;
    }
    if (b < 0) {
        i64 t = (b - c + 1) / c;
        b -= t * c, sum += (n + 1) * t;
    }
    return euclid(a, b, c, n) + sum;
}
```

0x34 数学 初等数论–取整

取整

向下取整：$\lfloor x \rfloor$，表示不超过 x 的最大整数（floor）。

向上取整：$\lceil x \rceil$，表示不小于 x 的最小整数（ceiling）。

注意：C++ 中的整数除法结果向零取整，取模结果与被除数同号，与数学中的取整有较大区别。

取整的简单性质：

- $x-1<\lfloor x \rfloor \le x$，$x \le \lceil x \rceil < x+1$。
- $\lfloor -x \rfloor = -\lceil x \rceil$，$\lceil -x \rceil = -\lfloor x \rfloor$。

取整的常见恒等式

上下取整的相互转化：$n,m \in \mathbb{Z}$，$m>0$

$$\left\lceil \frac{n}{m} \right\rceil = \left\lfloor \frac{n+m-1}{m} \right\rfloor = \left\lfloor \frac{n-1}{m} \right\rfloor + 1$$
$$\left\lfloor \frac{n}{m} \right\rfloor = \left\lceil \frac{n-m+1}{m} \right\rceil = \left\lceil \frac{n+1}{m} \right\rceil - 1$$

上下取整的嵌套恒等式：$a,b,c \in \mathbb{N}_+$

$$\left\lfloor \frac{\lfloor \frac{a}{b} \rfloor}{c} \right\rfloor = \left\lfloor \frac{a}{bc} \right\rfloor$$
$$\left\lceil \frac{\lceil \frac{a}{b} \rceil}{c} \right\rceil = \left\lceil \frac{a}{bc} \right\rceil$$

上下取整对加法的恒等式：$A,B,C,D \in \mathbb{N}_+$

$$\left\lfloor \frac{\lfloor \frac{A}{B} \rfloor + C}{D} \right\rfloor = \left\lfloor \frac{A+BC}{BD} \right\rfloor$$
$$\left\lceil \frac{\lceil \frac{A}{B} \rceil + C}{D} \right\rceil = \left\lceil \frac{A+BC}{BD} \right\rceil$$

数论分块

也称之为整除分块。

下取整数论分块

给出一个正整数 n，记 D_n 表示 $\lfloor \frac{n}{d} \rfloor$ 所有可能的取值构成的集合，即 $D_n = \{\lfloor \frac{n}{d} \rfloor \mid 1 \le d \le n\}$。

性质 1：$\lfloor \frac{n}{d} \rfloor$ 的取值不超过 $2\sqrt{n}$ 种，即 $|D_n| \le 2\sqrt{n}$。

简单推导：

- 当 $d \le \sqrt{n}$ 时，因为 d 的取值不超过 $\sqrt{n}$ 种，所以此时 $\lfloor \frac{n}{d} \rfloor$ 的取值不超过 $\sqrt{n}$ 种。
- 当 $d > \sqrt{n}$ 时，因为 $\lfloor \frac{n}{d} \rfloor \le \sqrt{n}$，所以此时 $\lfloor \frac{n}{d} \rfloor$ 的取值不超过 $\sqrt{n}$ 种。

性质 2：若 $m \in D_n$，则 $D_m \subseteq D_n$。

简单推导：设 $m = \lfloor \frac{n}{k} \rfloor$，此时对于任意 $d \in \mathbb{N}_+$

$$\left\lfloor \frac{m}{d} \right\rfloor = \left\lfloor \frac{\lfloor \frac{n}{k} \rfloor}{d} \right\rfloor = \left\lfloor \frac{n}{kd} \right\rfloor \in D_n$$

这意味着，如果我们想要递归地应用数论分块（类似杜教筛），在计算过程中涉及到的取值集合均为 D_n。

下取整数论分块：快速地划分出 $\lfloor \frac{n}{d} \rfloor$ 取值相同的所有区间。

由于函数 $\frac{n}{x}$ 在区间 $(0,+\infty)$ 上单调递减，所以 $\lfloor \frac{n}{x} \rfloor$ 取值相同的所有自变量 x 构成一段连续的区间。并且由上述分析，这样的区间不超过 $2\sqrt{n}$ 个。

对于任意 i $(1 \le i \le n)$，需要找到一个最大的 j $(i \le j \le n)$ 满足 $\lfloor \frac{n}{i} \rfloor = \lfloor \frac{n}{j} \rfloor$。由不等式 $\lfloor \frac{n}{i} \rfloor \le \frac{n}{j}$ 可得 $\boldsymbol{j = \left\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \right\rfloor}$。

根据该公式，我们就可以根据区间的左端点计算出右端点。于是依次遍历所有区间即可。

```
// h 表示 x 的上界
for (int x = 1, nx; x <= h; x = nx + 1) {
    nx = h;
    if (x <= n) {
        chmin(nx, n / (n / x));
    }
    // 区间 [x, nx] 内：下取整的值均为 n/x
}
```

> By the way：数论分块套数论分块的时间复杂度为 $\mathcal{O}(n^{\frac{3}{4}})$。

上取整数论分块

上取整数论分块：快速地找出 $\lceil \frac{n}{d} \rceil$ 取值相同的所有区间。

由恒等式 $\lceil \frac{n}{m} \rceil = \lfloor \frac{n-1}{m} \rfloor + 1$ 可知，n 的上取整数论分块与 $n-1$ 的下取整数论分块，分出的区间是一样的。

于是套用 $n-1$ 的下取整数论分块即可，**注意 $d \ge n$ 的特殊情况**。

```
// h 表示 x 的上界
for (int x = 1, nx; x <= h; x = nx + 1) {
    nx = h;
    if (x < n) {
        chmin(nx, (n - 1) / ((n - 1) / x));
    }
    // 区间 [x, nx] 内：上取整的值均为 (n-1)/x+1
}
```

多维数论分块

二维数论分块：快速地找出 $\lfloor \frac{n}{d} \rfloor, \lfloor \frac{m}{d} \rfloor$ 取值均相同的所有区间。

显然，这样的区间不超过 $2\sqrt{n} + 2\sqrt{m}$ 个。

对于任意 i，需要找到一个最大的 j 满足 $\lfloor \frac{n}{i} \rfloor = \lfloor \frac{n}{j} \rfloor, \lfloor \frac{m}{i} \rfloor = \lfloor \frac{m}{j} \rfloor$。则有 $\boldsymbol{j = \min\left\{ \left\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \right\rfloor, \left\lfloor \frac{m}{\lfloor \frac{m}{i} \rfloor} \right\rfloor \right\}}$。

扩展到多维也是一样。

```
// h 表示 x 的上界
for (int x = 1, nx; x <= h; x = nx + 1) {
    nx = h;
    if (x <= n) {
        chmin(nx, n / (n / x));
    }
    if (x <= m) {
        chmin(nx, m / (m / x));
    }
    // 区间 [x, nx] 内：n, m 对 x 下取整的值分别为 n/x, m/x
}
```

积性函数与狄利克雷卷积

积性函数：若数论函数 $f(n)$，对于**任意互质**正整数 x, y，都有 $f(xy) = f(x)f(y)$，则称 $f(n)$ 为积性函数。

完全积性函数：若数论函数 $f(n)$，对于**任意**正整数 x, y，都有 $f(xy) = f(x)f(y)$，则称 $f(n)$ 为完全积性函数。

积性函数的简单性质：

- 对于任意积性函数 $f(x)$（除了恒为 0 的函数），有 $f(1) = 1$。
- 对于任意积性函数 $f(x), g(x)$，**以下的函数仍为积性函数**

$$f(x^p)$$
$$f^p(x)$$
$$f(x)g(x)$$
$$\sum_{d|x} f(d)g\left(\frac{x}{d}\right)$$

- 对于任意积性函数 $f(x)$，将 x 唯一分解成 $x = p_1^{c_1} p_2^{c_2} \dots p_t^{c_t}$，有 $f(x) = \prod_{i=1}^{t} f(p_i^{c_i})$。

常见积性函数（及性质）

单位函数（完全积性）

$$\epsilon(n) = [n = 1]$$

恒等函数（完全积性）

$$\mathrm{ID}_k(n) = n^k$$

当 $k = 1$ 时，亦可简写为 $\mathrm{ID}(n) = n$。

常数函数（完全积性）

$$1(n) = 1$$

约数函数（积性）

约数函数：

$$\sigma_k(n) = \sum_{d|n} d^k$$

当 $k = 0$ 时，$\sigma_0(n)$ 表示 n 的约数个数函数。

约数个数函数的前缀和：

$$\sum_{i=1}^{n} \sigma_0(i) = \sum_{i=1}^{n} \left\lfloor \frac{n}{i} \right\rfloor$$

乘积的约数个数函数：

$$\sigma_0(ij) = \sum_{x|i} \sum_{y|j} [\gcd(x, y) = 1]$$

$$\sigma_0(ijk) = \sum_{a|i} \sum_{b|j} \sum_{c|k} [\gcd(a, b) = \gcd(b, c) = \gcd(a, c) = 1]$$

简单推导（仅证二维）：从唯一分解的角度，对每个质因子 p 单独考虑。p 在 (x, y) 中的出现情况数，为 i, j 中质因子 p 的个数之和再加 1。将所有质因子 p 的贡献相乘，这正是 $\sigma_0(ij)$ 的表达式。

可以扩展至更多维。

欧拉函数（积性）

欧拉函数：$\varphi(n)$，表示 $1 \sim n$ 中与 n 互质的数的个数。将 n 唯一分解成 $n = p_1^{c_1} p_2^{c_2} \dots p_t^{c_t}$，则欧拉函数为

$$\varphi(n) = n \prod_{i=1}^{t} (1 - \frac{1}{p_i})$$

乘积的欧拉函数：

$$\varphi(ij) = \frac{\varphi(i)\varphi(j)\gcd(i, j)}{\varphi(\gcd(i, j))}$$

简单推导：设 $P(n)$ 表示 n 的质因数集合，则有 $P(ij) = P(i) \cup P(j)$ 与 $P(\gcd(i, j)) = P(i) \cap P(j)$。

由容斥原理得

$$\prod_{p|ij} \left(1 - \frac{1}{p}\right) = \frac{\prod_{p|i} \left(1 - \frac{1}{p}\right) \prod_{p|j} \left(1 - \frac{1}{p}\right)}{\prod_{p|\gcd(i,j)} \left(1 - \frac{1}{p}\right)} \iff \frac{\varphi(ij)}{ij} = \frac{\frac{\varphi(i)}{i} \frac{\varphi(j)}{j}}{\frac{\varphi(\gcd(i,j))}{\gcd(i,j)}}$$

莫比乌斯函数（积性）

莫比乌斯函数：$\mu(n)$，将 n 唯一分解成 $n = p_1^{c_1} p_2^{c_2} \dots p_t^{c_t}$，则莫比乌斯函数有如下的定义

$$\mu(n) = \begin{cases} (-1)^t & \forall i \in [1, t], c_i = 1 \\ 0 & \exists i \in [1, t], c_i > 1 \end{cases}$$

特别地，$\mu(1) = 1$。

莫比乌斯函数的平方：

$$\mu^2(n) = \sum_{d^2|n} \mu(d)$$

简单推导：记 $m = \prod p_i^{\lfloor c_i/2 \rfloor}$，则 $d^2 \mid n \iff d \mid m$，由 $\sum_{d|m} \mu(d) = [m = 1]$ 可证。

线性筛筛积性函数

对于积性函数 $f(x)$，若质数幂的函数值 $f(p^c)$ 便于求出，则 $f(x)$ 的前 n 项函数值可以通过线性筛得出。

具体地，记 low_i 表示：设 i 的最小质因子为 p_1，其次数为 c_1，则 $\text{low}_i = p_1^{c_1}$。

在线性筛的过程中，额外进行如下的维护

- 维护 low_i 数组。
- 每当筛出一个质数 p，就将 $f(p), f(p^2), \dots, f(p^c)$ 全都求出来。
- 其余满足 $i \neq \text{low}_i$ 的 i，使用递推式 $f(i) = f(\text{low}_i) \times f\left(\frac{i}{\text{low}_i}\right)$ 计算函数值。

0x34 线性筛筛积性函数.cpp

```cpp
// 线性筛筛积性函数
int primeCount, prime[MaxV], low[MaxV];
int f[MaxV];
void sieve(int n) {
    f[1] = 1;
    for (int i = 2; i <= n; i ++) {
        if (!low[i]) {
            prime[++ primeCount] = i, low[i] = i;
            for (i64 v = i, c = 1; v <= n; v *= i, c ++) {
```

```
                // 计算 f[v]
            }
        }
        for (int j = 1; j <= primeCount; j ++) {
            if (prime[j] > n / i) break;
            low[i * prime[j]] = i % prime[j] ? prime[j] : low[i] * prime[j];

            if (i % prime[j] == 0) break;
        }
        if (low[i] < i) {
            f[i] = f[low[i]] * f[i / low[i]];
        }
    }
}
```

狄利克雷卷积

狄利克雷卷积：对于两个数论函数 f, g，定义 f, g 的狄利克雷卷积为

$$(f * g)(x) = \sum_{d|x} f(d) g\left(\frac{x}{d}\right)$$

狄利克雷卷积的简单性质：

- 交换律：$f * g = g * f$。
- 结合律：$(f * g) * h = f * (g * h)$。
- 分配律：$f * (g + h) = f * g + f * h$。
- 单位元：$f * \epsilon = \epsilon * f = f$，因此 ϵ 又被称为狄利克雷卷积的单位元。

常见狄利克雷卷积式

式 1

$$\epsilon = \mu * 1 \iff \sum_{d|n} \mu(d) = [n = 1]$$

简单推导：由积性函数的简单性质可知，关于 n 的函数 $f(n) = \sum_{d|n} \mu(d)$ 为积性函数。

- 当 $n = 1$ 时，等式成立。
- 当 $n > 1$ 时，将 n 唯一分解成 $n = p_1^{c_1} p_2^{c_2} \dots p_t^{c_t}$，则有

$$\begin{aligned} f(n) &= \prod_{i=1}^{t} f(p_i^{c_i}) \\ &= \prod_{i=1}^{t} \left(\sum_{j=0}^{c_i} \mu(p_i^j) \right) \\ &= \prod_{i=1}^{t} (\mu(1) + \mu(p_i)) \\ &= 0 \end{aligned}$$

综上，$\sum_{d|n} \mu(d) = [n = 1]$。

式 2

$$\mathrm{ID} = \varphi * 1 \iff \sum_{d|n} \varphi(d) = n$$

简单推导：由积性函数的简单性质可知，关于 n 的函数 $f(n) = \sum_{d|n} \varphi(d)$ 为积性函数。

- 当 $n = 1$ 时，等式成立。
- 当 $n > 1$ 时，将 n 唯一分解成 $n = p_1^{c_1} p_2^{c_2} \dots p_t^{c_t}$，则有

$$\begin{aligned} f(n) &= \prod_{i=1}^{t} f(p_i^{c_i}) \\ &= \prod_{i=1}^{t} \left(\sum_{j=0}^{c_i} \varphi(p_i^j) \right) \\ &= \prod_{i=1}^{t} \left(1 + \sum_{j=1}^{c_i} (p_i^j - p_i^{j-1}) \right) \\ &= \prod_{i=1}^{t} p_i^{c_i} \\ &= n \end{aligned}$$

综上，$\sum_{d|n} \varphi(d) = n$。

式 3

$$\varphi = \mu * \mathrm{ID} \iff \varphi(n) = \sum_{d|n} d\mu\left(\frac{n}{d}\right)$$

简单推导：

$$\begin{aligned}\varphi * 1 = \mathrm{ID} &\Longrightarrow \varphi * 1 * \mu = \mathrm{ID} * \mu \\ &\Longrightarrow \varphi * \epsilon = \mathrm{ID} * \mu \\ &\Longrightarrow \varphi = \mathrm{ID} * \mu\end{aligned}$$

杜教筛

杜教筛：用于在 $\mathcal{O}(n^{\frac{2}{3}})$ 时间内求解一类数论函数 $f(x)$ 的前缀和。

- $f(x)$ 为积性函数。
- 存在辅助函数 $g(x)$，满足函数 $f*g$ 与 g 的前缀和便于求出。

记 $S(n)=\sum_{i=1}^{n} f(i)$。取杜教筛中的辅助函数 $g(x)$，则有

$$\begin{aligned}\sum_{i=1}^{n}(f*g)(i) &= \sum_{i=1}^{n}\sum_{d|i} g(d) f\left(\frac{i}{d}\right) \\ &= \sum_{i=1}^{n} g(i) \sum_{j=1}^{\lfloor \frac{n}{i} \rfloor} f(j) \\ &= \sum_{i=1}^{n} g(i) S\left(\left\lfloor \frac{n}{i} \right\rfloor\right)\end{aligned}$$

移项得

$$g(1)S(n) = \sum_{i=1}^{n}(f*g)(i) - \sum_{i=2}^{n} g(i) S\left(\left\lfloor \frac{n}{i} \right\rfloor\right)$$

此时得到了一个 $S(n)$ 关于 $S\left(\lfloor \frac{n}{i} \rfloor\right)$ 的递推式。**若函数 $f*g$ 与 g 的前缀和便于求出，就可以使用数论分块处理该式。**因此找到一个合适的辅助函数 g 是杜教筛的关键。

杜教筛的时间复杂度分析：

由数论分块的性质，对于 $m \in D_n$，有 $D_m \subseteq D_n$。于是我们需要使用记忆化保证杜教筛的复杂度，使用记忆化以后，只需要对所有的 $k \in D_n$ 计算一次 $S(k)$ 即可。

假设函数 $f*g$ 与 g 的前缀和均可 $\mathcal{O}(1)$ 求出，设计算 $S(n)$ 的时间复杂度为 $T(n)$，则有

$$\begin{aligned}T(n) &= \sum_{k\in D_n} \mathcal{O}(\sqrt{k}) \\ &= \mathcal{O}(\sqrt{n}) + \sum_{i=1}^{\sqrt{n}} \mathcal{O}(\sqrt{i}) + \sum_{i=2}^{\sqrt{n}} \mathcal{O}\left(\sqrt{\frac{n}{i}}\right) \\ &= \mathcal{O}\left(\int_0^{\sqrt{n}} \left(\sqrt{x} + \sqrt{\frac{n}{x}}\right) \mathrm{dx}\right) \\ &= \mathcal{O}(n^{\frac{3}{4}})\end{aligned}$$

进一步优化，我们预处理出数论函数 $f(x)$ 的前 m 项（其中 $m \geq \sqrt{n}$），并预处理出其前缀和。设预处理的时间复杂度为 $T_0(m)$，则有

$$\begin{aligned}T(n) &= T_0(m) + \sum_{k\in D_n, k>m} \mathcal{O}(\sqrt{k}) \\ &= T_0(m) + \sum_{i=1}^{\lfloor \frac{n}{m} \rfloor} \mathcal{O}\left(\sqrt{\frac{n}{i}}\right) \\ &= \mathcal{O}\left(T_0(m) + \int_0^{\frac{n}{m}} \sqrt{\frac{n}{x}} \mathrm{dx}\right) \\ &= \mathcal{O}\left(T_0(m) + \frac{n}{\sqrt{m}}\right)\end{aligned}$$

通常我们使用线性筛预处理 $f(x)$ 的前 m 项，此时 $T_0(m)=\mathcal{O}(m)$。由均值不等式，取 $m=\mathcal{O}(n^{\frac{2}{3}})$ 可得杜教筛的最优时间复杂度 $\mathcal{O}(n^{\frac{2}{3}})$。

总结：预处理出 $f(x)$ 的前 $n^{\frac{2}{3}}$ 项及其前缀和，使用关于 $S\left(\lfloor \frac{n}{i} \rfloor\right)$ 的递推式求解 $S(n)$，并使用 `std::map` 记忆化。

0x34 杜教筛.cpp

```cpp
namespace du {
    const int V = 1e6; // 预处理的范围（需根据实际问题调整）
    const int MaxV = V + 10;

    int primeCount, prime[MaxV], low[MaxV];
    int f[MaxV];

    i64 f1[MaxV];
```

```
    std::map<int, i64> f2;

    void init(int n = V) {
        f[1] = 1;
        for (int i = 2; i <= n; i ++) {
            if (!low[i]) {
                prime[++ primeCount] = i, low[i] = i;
                for (i64 v = i, c = 1; v <= n; v *= i, c ++) {
                    // 计算 f[v]
                }
            }
            for (int j = 1; j <= primeCount; j ++) {
                if (prime[j] > n / i) break;
                low[i * prime[j]] = i % prime[j] ? prime[j] : low[i] *
                 ↪  prime[j];

                if (i % prime[j] == 0) break;
            }
            if (low[i] < i) {
                f[i] = f[low[i]] * f[i / low[i]];
            }
        }

        for (int i = 1; i <= n; i ++) {
            f1[i] = f1[i - 1] + f[i];
        }
    }

    i64 fSum(int n) {
        if (n <= V) return f1[n];
        if (f2.contains(n)) return f2[n];

        i64 ans = 0; // 初始计算函数 f*g 第 1~n 项的和
        for (int x = 2, nx; x <= n; x = nx + 1) {
            nx = n / (n / x);
            i64 sum = 0; // 计算函数 g 第 x~nx 项的和
            ans -= sum * fSum(n / x);
        }
        return f2[n] = ans;
    }
}
```

杜教筛求莫比乌斯函数与欧拉函数前缀和：

给莫比乌斯函数选择的辅助函数为 $1(n)=1$，由于 $\mu*1=\epsilon$，函数 ϵ 与 1 的前缀和均便于求出。代入杜教筛公式得

$$S(n)=1-\sum_{i=2}^{n}S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$$

给欧拉函数选择的辅助函数为 $1(n)=1$，由于 $\varphi*1=\mathrm{ID}$，函数 ID 与 1 的前缀和均便于求出。代入杜教筛公式得

$$S(n)=\frac{n(n+1)}{2}-\sum_{i=2}^{n}S\left(\left\lfloor\frac{n}{i}\right\rfloor\right)$$

求欧拉函数前缀和，更简便的方法是莫比乌斯反演，注意到 $\sum_{i=1}^{n}\varphi(i)=\sum_{1\le i\le j\le n}[\gcd(i,j)=1]$。

而 $\sum_{i=1}^{n}\sum_{j=1}^{n}[\gcd(i,j)=1]=\sum_{d=1}^{n}\mu(d)\left\lfloor\frac{n}{d}\right\rfloor^2$，使用莫比乌斯函数前缀和配合数论分块处理该式即可。

Min_25 筛

~~待填坑。~~

0x34 数学 初等数论–同余

同余

带余除法：对于任意 $n,m\in\mathbb{Z}$ 且 $m\neq 0$，存在唯一的 $q,r\in\mathbb{Z}$，使得 $n=qm+r$ $(0\le r<|m|)$。

当 $m>0$ 时，记 $q=\left\lfloor\frac{n}{m}\right\rfloor$，$r=n \bmod m$。

同余：若 $a \bmod m=b \bmod m$，则称 a,b 模 m 同余，记作 $a\equiv b\pmod m$。

同余的简单性质：

- 若 $a\equiv b\pmod m, c\equiv d\pmod m$，则
  - $a+c\equiv b+d\pmod m$。
  - $ac\equiv bd\pmod m$。
- $ac\equiv bc\pmod m\Longrightarrow a\equiv b\pmod{\frac{m}{\gcd(c,m)}}$。

同余类：集合 $\overline{a}=\{a+km\mid k\in\mathbb{Z}\}$（其中 $a\in[0,m-1]$）中的所有数模 m 同余，余数均为 a，该集合称为一个模 m 的同余类。

（完全）剩余系：m 的完全剩余系包含 m 个元素 $\overline{0},\overline{1},\ldots,\overline{m-1}$。

（简化）剩余系：m 的简化剩余系包含 $\varphi(m)$ 个元素，$1 \sim m$ 中与 m 互质的数代表的同余类构成该剩余系。
剩余系的简单性质：

- 对于 $i = 0, 1, \dots, m-1$，$ia \bmod m$ 两两不同当且仅当 $\gcd(a, m) = 1$。
- 对于与 m 互质的整数 i, j，$ij \bmod m$ 仍然与 m 互质。

费马小定理与欧拉定理

费马小定理：设 p 是质数，对于任意整数 a（a 不能是 p 的倍数），满足 $a^{p-1} \equiv 1 \pmod p$。

> 另一个形式：设 p 是质数。对于任意整数 a，满足 $a^p \equiv a \pmod p$。

简单推导：由于 p 是质数且 $p \nmid a$，对于 $i = 0, 1, \dots, p-1$，$ia \bmod p$ 两两不同。
因此 $\{ia \bmod p\}$ 也构成 $0 \sim p-1$ 的排列。所以

$$\prod_{i=1}^{p-1} (ia \bmod p) = \prod_{i=1}^{p-1} i$$

又由于

$$\prod_{i=1}^{p-1} ia \equiv a^{p-1} \prod_{i=1}^{p-1} i \pmod p$$

所以

$$(a^{p-1} - 1) \prod_{i=1}^{p-1} i \equiv 0 \pmod p$$

由于 $1, 2, \dots, p-1$ 均不是 p 的倍数，所以 $a^{p-1} - 1$ 是 p 的倍数。即 $a^{p-1} \equiv 1 \pmod p$。

> By the way：费马小定理的逆命题并不成立。
> 即使对于所有与 p 互质的整数 a 均满足 $a^{p-1} \equiv 1 \pmod p$，p 也不一定是质数。

欧拉定理：设整数 $m > 0$，对于任意满足 $\gcd(a, m) = 1$ 的整数 a，满足 $a^{\varphi(m)} \equiv 1 \pmod m$。
简单推导：设 R 表示 $1 \sim m$ 中与 m 互质的数构成的集合。已知 $a \in R$，对于所有 R 中的元素 r，$ra \bmod m$ 仍然属于 R 且两两不同。所以

$$\prod_{r \in R} (ra \bmod m) = \prod_{r \in R} r$$

又由于

$$\prod_{r \in R} ra \equiv a^{\varphi(m)} \prod_{r \in R} r \pmod m$$

所以

$$(a^{\varphi(m)} - 1) \prod_{r \in R} r \equiv 0 \pmod m$$

由于每个 $r \in R$ 均与 m 互质，所以 $a^{\varphi(m)} \equiv 1 \pmod m$。
扩展欧拉定理：设整数 $m > 0$，对于任意整数 a, b $(b \geq 0)$，有

$$a^b \equiv \begin{cases} a^{b \bmod \varphi(m)} & \gcd(a, m) = 1 \\ a^b & \gcd(a, m) \neq 1, b < \varphi(m) \\ a^{b \bmod \varphi(m) + \varphi(m)} & \gcd(a, m) \neq 1, b \geq \varphi(m) \end{cases} \pmod m$$

直观理解：**考虑 $a^b \bmod m$ 的循环情况**。当 $\gcd(a, m) = 1$ 时，该循环为纯循环；当 $\gcd(a, m) \neq 1$ 时，该循环为混循环（类似 ρ 形），需要对尾巴和环形分别讨论。
通常使用扩展欧拉定理，以达到降幂的效果。

模意义下逆元

模意义下逆元：对于非零整数 a, m，如果存在整数 b 使得 $ab \equiv 1 \pmod m$，则称 b 为 a 在模 m 意义下的逆元。记作 $a^{-1} \bmod m$。

模质数意义下逆元

当模数 p 为质数时，对于任意整数 a（a 不能是 p 的倍数），有

$$a \times a^{p-2} \equiv 1 \pmod p$$

此时，$a^{p-2} \bmod p$ 即为 a 在模 p 意义下的逆元。

单个逆元的求解

若要求解 a 在模 m 意义下的逆元，相当于是要求解线性同余方程 $ax \equiv 1 \pmod m$，使用 exgcd 即可。
时间复杂度 $\mathcal{O}(\log V)$。

0x34 单个逆元的求解.cpp

```cpp
// 单个逆元的求解
i64 inverse(i64 a, i64 m) {
    auto [d, x, y] = exgcd(a, m);
    // assert(d == 1);
    return (x % m + m) % m;
}
```

多个逆元的求解

若要求解多个整数 $a_1, \dots, a_n$ 在模 m 意义下的逆元，处理出前缀积与前缀积逆元，通过 $a_i^{-1} = S_{i-1}S_i^{-1} \bmod m$ 求出所有数的逆元即可。

时间复杂度 $\mathcal{O}(n)$。

预处理线性逆元

若要求解 $1 \sim n$ 所有整数模质数 p 意义下的逆元，对于 $1 < i < p$，有

$$0 \equiv \left\lfloor \frac{p}{i} \right\rfloor i + (p \bmod i) \pmod p$$

将等式两边同乘以 $i^{-1}(p \bmod i)^{-1}$，得

$$i^{-1} \equiv -\left\lfloor \frac{p}{i} \right\rfloor (p \bmod i)^{-1} \pmod p$$

根据上式递推即可。

时间复杂度 $\mathcal{O}(n)$。

0x34 预处理线性逆元.cpp

```cpp
// 预处理线性逆元
std::vector<int> v;
void linearInverse(int n) {
    v.resize(n + 1);
    v[1] = 1;
    for (int i = 2; i <= n; i ++) {
        v[i] = 1ll * v[mod % i] * (mod - mod / i) % mod;
    }
}
```

线性同余方程

裴蜀定理：设 a, b 是不均为 0 的整数。

- 存在整数 x, y，使得 $ax + by = \gcd(a, b)$。
- 对于任意整数 x, y，均有 $\gcd(a, b) \mid ax + by$。

> 裴蜀定理可以扩展到更多维：设 $a_1, \dots, a_n$ 是不均为 0 的整数。
>
> - 存在整数 $x_1, \dots, x_n$，使得 $\sum_{i=1}^{n} a_i x_i = \gcd(a_1, \dots, a_n)$。
> - 对于任意整数 $x_1, \dots, x_n$，均有 $\gcd(a_1, \dots, a_n) \mid \sum_{i=1}^{n} a_i x_i$。
>
> 可以使用 gcd 的结合律进行归纳证明。

exgcd（扩展欧几里得算法）：用于求出二元一次不定方程 $ax + by = \gcd(a, b)$ 的一组可行解 x, y。

在欧几里得算法的基础上归纳求解：

- 当 $b = 0$ 时，有 $x = 1, y = 0$，使得 $a \cdot 1 + 0 \cdot 0 = \gcd(a, 0)$。
- 当 $b > 0$ 时，由于 $\gcd(a, b) = \gcd(b, a \bmod b)$，假设存在一对 x, y 使得

$$bx + (a \bmod b)y = \gcd(b, a \bmod b)$$

利用 $a \bmod b = a - \left\lfloor \frac{a}{b} \right\rfloor b$，并移项得

$$ay + b\left(x - \left\lfloor \frac{a}{b} \right\rfloor y\right) = \gcd(a, b)$$

此时取 $x' = y, y' = x - \left\lfloor \frac{a}{b} \right\rfloor y$，使得 $ax' + by' = \gcd(a, b)$。

时间复杂度 $\mathcal{O}(\log \max(a, b))$。

可以证明，扩展欧几里得算法求出的可行解 x, y 满足 $|x| \le b, |y| \le a$。

0x34 exgcd（扩展欧几里得算法）.cpp

```cpp
// 扩展欧几里得算法
std::array<i64, 3> exgcd(i64 a, i64 b) {
    if (b == 0) return {a, 1, 0};
    auto [d, x, y] = exgcd(b, a % b);
    return {d, y, x - (a / b) * y};
}
```

二元一次不定方程 $ax + by = c$：

- 有解性：当且仅当 $\gcd(a, b) \mid c$。

- 通解：使用 exgcd 找出方程 $ax+by=\gcd(a,b)$ 的一组特解 x_0,y_0，则有通解

$$x=\frac{c}{\gcd(a,b)}x_0+k\frac{b}{\gcd(a,b)}$$
$$y=\frac{c}{\gcd(a,b)}y_0-k\frac{a}{\gcd(a,b)}$$

线性同余方程 $ax\equiv b\pmod m$：

该方程等价于 $ax-b$ 为 m 的倍数，不妨设为 $-y$ 倍，则原方程转化为 $ax+my=b$。

- 有解性：当且仅当 $\gcd(a,m)\mid b$。
- 通解：使用 exgcd 找出方程 $ax+my=\gcd(a,m)$ 的一组特解 x_0,y_0，则有通解

$$x=\frac{b}{\gcd(a,m)}x_0+k\frac{m}{\gcd(a,m)}$$

> 将线性同余方程中 x 的系数消成 1：
> 对于线性同余方程 $ax\equiv b\pmod m$，有解的充要条件是 $\gcd(a,m)\mid b$。当满足该条件时，方程可以转化为
>
> $$\frac{a}{\gcd(a,m)}x\equiv\frac{b}{\gcd(a,m)}\pmod{\frac{m}{\gcd(a,m)}}$$
>
> 此时 $\gcd\left(\frac{a}{\gcd(a,m)},\frac{m}{\gcd(a,m)}\right)=1$，$x$ 在模 $\frac{m}{\gcd(a,m)}$ 意义下有唯一解 x'。
> 于是线性同余方程 $ax\equiv b\pmod m$ 转化为了 $x\equiv x'\pmod{m'}$ 的形式。该形式对线性同余方程组的求解大有用处。

线性同余方程组

模数两两互质的一元线性同余方程组（中国剩余定理，CRT）：

$$\begin{cases}x\equiv a_1\pmod{m_1}\\x\equiv a_2\pmod{m_2}\\\dots\\x\equiv a_n\pmod{m_n}\end{cases}$$

- 记所有模数的积为 $M=\prod_{i=1}^n m_i$。
- 对于第 i 个方程：
  - 记 $M_i=\frac{M}{m_i}$。
  - 记 V_i 表示 M_i 在模 m_i 意义下的逆元。
- 特解：

$$x_0=\sum_{i=1}^n a_iM_iV_i$$

- 通解：

$$x=x_0+kM$$

0x34 CRT（中国剩余定理）.cpp

```cpp
i64 inverse(i64 a, i64 m) {
    auto [d, x, y] = exgcd(a, m);
    return (x % m + m) % m;
}

using equ = std::pair<i64, i64>; // {a, m} 表示方程 x = a (mod m)

// 中国剩余定理
equ CRT(std::vector<equ> seq) {
    i64 M = 1;
    for (auto [a, m] : seq) {
        M *= m;
    }
    i64 x = 0;
    for (auto [a, m] : seq) {
        i64 Mi = M / m;
        i64 Vi = inverse(Mi, m);
        x = (x + a * Mi * Vi) % M; // 必要时，使用“快速乘”防止爆 long long
    }
    x = (x % M + M) % M;
    return {x, M};
}
```

不保证模数两两互质的一元线性同余方程组（扩展中国剩余定理，exCRT）：

仅考虑两个方程的合并（多个方程的情况，可以依次合并）。设这两个方程为

$$\begin{cases}x\equiv a_1\pmod{m_1}\\x\equiv a_2\pmod{m_2}\end{cases}$$

则

$$x=pm_1+a_1=qm_2+a_2\Longrightarrow m_1p-m_2q=a_2-a_1$$

使用 exgcd 找出方程 $m_1p-m_2q=a_2-a_1$ 的一组特解 p_0,q_0，或判断无解。

此时取 $a' = p_0 m_1 + a_1$ 与 $m' = \operatorname{lcm}(m_1, m_2)$，两个方程合并为

$$x \equiv a' \pmod{m'}$$

0x34 exCRT（扩展中国剩余定理）.cpp

```cpp
// 扩展中国剩余定理
struct equ {
    i64 a, m; // {a, m} 表示方程 x = a (mod m)
    equ() {}
    equ(i64 a, i64 m) : a(a), m(m) {}

    friend equ operator + (equ l, equ r) {
        auto [a1, m1] = l;
        auto [a2, m2] = r;
        if (a1 == -1 || a2 == -1) {
            return {-1, -1};
        }
        auto [d, x, y] = exgcd(m1, m2);
        if ((a2 - a1) % d) {
            return {-1, -1};
        }
        x = x * ((a2 - a1) / d) % (m2 / d); // 必要时，使用“快速乘”防止爆 long long
        if (x < 0) x += (m2 / d);
        i64 m = m1 * (m2 / d);
        i64 a = (x * m1 + a1) % m; // 必要时，使用“快速乘”防止爆 long long
        return {a, m};
    }
};
```

阶与原根

阶：设整数 $m > 0$，满足 $a^x \equiv 1 \pmod{m}$ 的最小正整数 x，称作 a 模 m 的阶。记作 $\operatorname{ord}_m(a)$。

阶的存在条件：当且仅当 $\gcd(a, m) = 1$ 时，a 模 m 的阶存在。

> 必要性：当 $\gcd(a, m) \neq 1$ 时，a 在模 m 意义下的逆元不存在。故必定有 $\gcd(a, m) = 1$。
>
> 充分性：当 $\gcd(a, m) = 1$ 时，由欧拉定理 $a^{\varphi(m)} \equiv 1 \pmod{m}$，故此时阶必定存在，并且 $\operatorname{ord}_m(a) \mid \varphi(m)$。

阶的简单性质：

- 若 $a \perp m$，则 $a, a^2, \dots, a^{\operatorname{ord}_m(a)}$ 模 m 两两不同。
- 若 $a \perp m$，则 $\operatorname{ord}_m(a) \mid \varphi(m)$。
- 若 $a \perp m$，则

$$\operatorname{ord}_m(a^k) = \frac{\operatorname{ord}_m(a)}{\gcd(\operatorname{ord}_m(a), k)}$$

简单推导：$\operatorname{ord}_m(a^k) = \frac{\operatorname{lcm}(\operatorname{ord}_m(a), k)}{k}$。

- 若 $a \perp m, b \perp m$，则

$$\frac{\operatorname{lcm}(\operatorname{ord}_m(a), \operatorname{ord}_m(b))}{\gcd(\operatorname{ord}_m(a), \operatorname{ord}_m(b))} \mid \operatorname{ord}_m(ab) \mid \operatorname{lcm}(\operatorname{ord}_m(a), \operatorname{ord}_m(b))$$

- 若 $a \perp m, b \perp m$，则

$$\operatorname{ord}_m(ab) = \operatorname{ord}_m(a)\operatorname{ord}_m(b) \iff \gcd(\operatorname{ord}_m(a), \operatorname{ord}_m(b)) = 1$$

- 若 $a \perp m, b \perp m$，则总是存在满足 $c \perp m$ 的 c，使得

$$\operatorname{ord}_m(c) = \operatorname{lcm}(\operatorname{ord}_m(a), \operatorname{ord}_m(b))$$

> 简单构造：将 $\operatorname{ord}_m(a), \operatorname{ord}_m(b)$ 唯一分解
>
> $$\operatorname{ord}_m(a) = \prod p_i^{s_i}, \quad \operatorname{ord}_m(b) = \prod p_i^{t_i}$$
>
> 根据 s_i 与 t_i 的大小关系，可以将所有质因子分成两类
>
> $$A = \{i \mid s_i \geq t_i\}, \quad B = \{i \mid s_i < t_i\}$$
>
> 记
>
> $$x_A = \prod_{i \in A} p_i^{s_i}, \quad x_B = \prod_{i \in B} p_i^{s_i}, \quad y_A = \prod_{i \in A} p_i^{t_i}, \quad y_B = \prod_{i \in B} p_i^{t_i}$$
>
> 此时 $\operatorname{ord}_m(a) = x_A x_B, \operatorname{ord}_m(b) = y_A y_B$，可得
>
> $$\operatorname{ord}_m(a^{x_B}) = x_A, \quad \operatorname{ord}_m(b^{y_A}) = y_B$$
>
> 又由于 $x_A \perp y_B$，所以
>
> $$\operatorname{ord}_m(a^{x_B} b^{y_A}) = x_A y_B = \prod p_i^{\max(s_i, t_i)} = \operatorname{lcm}(\operatorname{ord}_m(a), \operatorname{ord}_m(b))$$
>
> 取 $c = a^{x_B} b^{y_A}$ 即可。

原根：设整数 $m > 0$，若 $\gcd(g, m) = 1$ 且 $\operatorname{ord}_m(g) = \varphi(m)$，则称 g 为模 m 的原根。

此时 $g, g^2, \dots, g^{\varphi(m)}$ 构成了 m 的简化剩余系。

原根判定定理：设整数 $m \geq 3$，则 g 是模 m 的原根，当且仅当 $\gcd(g, m) = 1$ 且对于 $\varphi(m)$ 的每个质因子 p，都有

$$g^{\frac{\varphi(m)}{p}} \not\equiv 1 \pmod{m}$$

原根存在定理：m 存在原根，当且仅当 $m = 1, 2, 4, p^c, 2p^c$（其中 p 为奇质数且 $c \geq 1$）。

阶的数量：若 m 存在原根，则对于 $d \mid \varphi(m)$，阶等于 d 的元素恰好有 $\varphi(d)$ 个。

> 简单推导：设 m 的原根为 g，则 m 的简化剩余系中的元素可以写成 $g^k \bmod m$ 的形式，其中 $1 \leq k \leq \varphi(m)$。则
>
> $$\mathrm{ord}_m(g^k) = \frac{\varphi(m)}{\gcd(\varphi(m), k)}$$
>
> 令 $\mathrm{ord}_m(g^k) = d$，则有 $\gcd(\varphi(m), k) = \frac{\varphi(m)}{d}$。
> 整个式子除以 $\frac{\varphi(m)}{d}$ 可得 $\gcd(d, k') = 1$，其中 $1 \leq k' \leq d$，此时 k' 的数量为 $\varphi(d)$。

原根数量：若 m 存在原根，则 m 的原根个数为 $\varphi(\varphi(m))$。

> 相当于是阶等于 $\varphi(m)$ 的元素个数，由上述分析，恰好有 $\varphi(\varphi(m))$ 个。

最小原根的上界：若 m 存在原根，则 m 的最小原根的上界为 $m^{0.25}$ 级别。

原根求解：若 m 存在原根，先求出 $\varphi(m)$，再求出 $\varphi(m)$ 的唯一分解形式。此时就可以根据“原根判定定理”配合快速幂判断一个数是否为 m 的原根。

从小到大暴力寻找原根，即可找到最小原根 g。

当我们已知其中一个原根 g 时，可以根据 g 生成其他所有的原根。对于所有满足 $1 \leq k \leq \varphi(m)$ 且 $\gcd(k, \varphi(m)) = 1$ 的 k，这些 $g^k \bmod m$ 不重不漏地覆盖了所有原根。

0x34 最小原根.cpp

```cpp
// 最小原根
int findPrimitiveRoot(int m) {
    if (m == 1) return 0;
    if (m == 2) return 1;

    int phi = m, x = m; // 若已知 m 为质数，则直接令 phi = m - 1
    for (int i = 2; i * i <= x; i ++) {
        if (x % i == 0) {
            phi = phi / i * (i - 1);
            while (x % i == 0) {
                x /= i;
            }
        }
    }
    if (x > 1) {
        phi = phi / x * (x - 1);
    }

    std::vector<int> pf;
    x = phi;
    for (int i = 2; i * i <= x; i ++) {
        if (x % i == 0) {
            pf.push_back(i);
            while (x % i == 0) {
                x /= i;
            }
        }
    }
    if (x > 1) {
        pf.push_back(x);
    }

    for (int g = 2; g < m; g ++) {
        if (std::gcd(g, m) != 1) { // 若已知 m 为质数，则可以省略该判断
            continue;
        }
        int flag = 1;
        for (int p : pf) {
            if (qpow(g, phi / p, m) == 1) {
                flag = 0;
                break;
            }
        }
        if (flag) {
            return g;
        }
    }
    return -1; // 若 m 较大，建议先判断 m 是否存在原根
}
```

0x34 所有原根.cpp

```cpp
// 所有原根
std::vector<int> allPrimitiveRoot(int m) {
    if (m == 1) return {0};
    if (m == 2) return {1};

    int phi = m, x = m; // 若已知 m 为质数，则直接令 phi = m - 1
    for (int i = 2; i * i <= x; i ++) {
        if (x % i == 0) {
            phi = phi / i * (i - 1);
            while (x % i == 0) {
                x /= i;
            }
        }
    }
    if (x > 1) {
        phi = phi / x * (x - 1);
    }

    std::vector<int> pf;
    x = phi;
    for (int i = 2; i * i <= x; i ++) {
        if (x % i == 0) {
            pf.push_back(i);
            while (x % i == 0) {
                x /= i;
            }
        }
    }
    if (x > 1) {
        pf.push_back(x);
    }

    int g = 2;
    for (; g < m; g ++) {
        if (std::gcd(g, m) != 1) { // 若已知 m 为质数，则可以省略该判断
            continue;
        }
        int flag = 1;
        for (int p : pf) {
            if (qpow(g, phi / p, m) == 1) {
                flag = 0;
                break;
            }
        }
        if (flag) {
            break;
        }
    }
    if (g == m) {
        return {};
    }

    std::vector<int> res;
    x = 1;
    for (int i = 1; i <= phi; i ++) {
        x = 1ll * x * g % m;
        if (std::gcd(i, phi) == 1) {
            res.push_back(x);
        }
    }
    std::sort(res.begin(), res.end());
    return res;
}
```

离散对数

离散对数：设整数 $m > 0$ 存在原根 g，对于 m 的简化剩余系中的元素 a，必存在唯一的整数 $0 \le k < \varphi(m)$ 使得 $g^k \equiv a \pmod m$。

此时称 k 为以 g 为底，模 m 的离散对数。记作 $k = \operatorname{ind}_g(a)$。

离散对数的简单性质：

- $\operatorname{ind}_g(ab) \equiv \operatorname{ind}_g(a) + \operatorname{ind}_g(b) \pmod{\varphi(m)}$。
  说明离散对数可以将“模 m 意义下乘法”转换成“模 $\varphi(m)$ 意义下加法”。
- $\operatorname{ind}_g(a^k) \equiv k \operatorname{ind}_g(a) \pmod{\varphi(m)}$。
- $\operatorname{ind}_{g_1}(a) \equiv \operatorname{ind}_{g_2}(a) \operatorname{ind}_{g_1}(g_2) \pmod{\varphi(m)}$。其中 g_1, g_2 均为 m 的原根。

高次同余方程

固定底数且与模数互质的高次同余方程 $a^x \equiv b \pmod p$（BSGS）：

取 $S = \lceil \sqrt{p} \rceil$，设 $x = iS - j$（其中 $i \ge 1$ 且 $0 \le j < S$），则有

$$a^{iS-j} \equiv b \pmod p$$
$$a^{iS} \times a^{-j} \equiv b \pmod p$$
$$a^{iS} \equiv b \times a^j \pmod p$$

预处理 $0 \le j < S$ 的 $b \times a^j \bmod p$ 存入哈希表，枚举所有 $1 \le i \le S$，在哈希表中查询 $(a^S)^i \bmod p$ 及其对应的 j。如果查询成功则返回 $iS - j$。如果一直查询失败则方程无解。

时间复杂度 $\mathcal{O}(\sqrt{p})$。

0x34 BSGS（大步小步算法）.cpp

```cpp
// 大步小步算法
int BSGS(int a, int b, int p) { // a^x = b (mod p)
    if (b == 1 || p == 1) return 0;
    if (a == 0) return b == 0 ? 1 : -1;

    int S = sqrt(p) + 1;
    std::unordered_map<int, int> buc;

    int w = 1;
    for (int j = 0; j < S; j ++, w = 1ll * w * a % p) {
        int v = 1ll * w * b % p;
        buc[v] = j;
    }
    int x = w;
    for (int i = 1; i <= S; i ++, x = 1ll * x * w % p) {
        if (buc.contains(x)) {
            return i * S - buc[x];
        }
    }
    return -1;
}
```

固定底数且不保证与模数互质的高次同余方程 $a^x \equiv b \pmod p$（exBSGS）：

该方程等价于 $a^x - b$ 为 p 的倍数，不妨设为 $-k$ 倍，则原方程转化为 $a \cdot a^{x-1} + kp = b$。

若 $\gcd(a,p) \nmid b$ 则无解，否则

$$a^{x-1}\frac{a}{\gcd(a,p)} + k\frac{p}{\gcd(a,p)} = \frac{b}{\gcd(a,p)}$$

每次将 $\frac{a}{\gcd(a,p)}$ 提取出来，并将 b,p 除以 $\gcd(a,p)$，就变成了一个规模更小的子问题。

记已经提取出的系数之积为 $r = \prod \frac{a}{\gcd(a,p)}$。在递降的过程中，若 $r \equiv b' \pmod{p'}$，则直接返回答案。

否则一直递降直到 $\gcd(a,p') = 1$，此时得到一个固定底数且与模数互质的高次同余方程 $ra^x \equiv b' \pmod{p'}$，套用 BSGS。

0x34 exBSGS（扩展大步小步算法）.cpp

```cpp
// 扩展大步小步算法
int exBSGS(int a, int b, int p) { // a^x = b (mod p)
    if (b == 1 || p == 1) return 0;
    if (a == 0) return b == 0 ? 1 : -1;

    int d, r = 1, step = 0;
    while (d = std::gcd(a, p), d > 1) {
        if (b % d) {
            return -1;
        }
        b /= d, p /= d, r = 1ll * r * (a / d) % p;
        step ++;
        if (r == b) {
            return step;
        }
    }
    if (p == 1) {
        return step;
    }

    int S = sqrt(p) + 1;
    std::unordered_map<int, int> buc;

    int w = 1;
    for (int j = 0; j < S; j ++, w = 1ll * w * a % p) {
        int v = 1ll * w * b % p;
        buc[v] = j;
    }
    int x = 1ll * r * w % p;
    for (int i = 1; i <= S; i ++, x = 1ll * x * w % p) {
        if (buc.contains(x)) {
            return step + i * S - buc[x];
        }
    }
    return -1;
}
```

固定次数且模数为质数的高次同余方程 $x^a \equiv b \pmod p$：设 p 的原根为 g，记 $x = g^c$（这里 c 是未知数），问题转化为

$$(g^a)^c \equiv b \pmod p$$

记 $v = g^a$，此时得到一个固定底数的高次同余方程 $v^c \equiv b \pmod p$。

- 特解：$x \equiv g^c \pmod p$，使用 BSGS 得到。

- 通解：$x \equiv g^{c+k\frac{p-1}{\gcd(p-1,a)}} \pmod p$，此时 $k=0,1,\dots,\gcd(p-1,a)-1$，其中 $\frac{p-1}{\gcd(p-1,a)}$ 表示 $\mathrm{ord}_p(g^a)$。

注意特判 $b=0$ 的情况。

Misc

平方差

平方差表示：只有 $1,2,4$，以及所有形如 $4k+2$ 的正整数，不能被表示成两个不同正整数的平方差。

平方差表示证明：对于正整数 n，相当于判断是否存在两个不同的正整数 $a,b\ (a>b)$ 使得 $n=a^2-b^2=(a+b)(a-b)$。

设 $x=a+b, y=a-b$，则 $n=xy$。由于 x,y 之间的差为 $2b$，所以 x,y 奇偶性相同。对于同奇偶的两个 $x,y\ (x>y)$，也都可以解出 a,b 的值。

故对于正整数 n，相当于是要找出 n 的两个不同的约数 x,y，且 x,y 奇偶性相同。

- 当 n 为奇数时，取 $x=n,y=1$ 即可。
- 当 n 为偶数时：
  - 当 $n=4k$ 时，取 $x=2k,y=2$ 即可。
  - 当 $n=4k+2$ 时，分解出的两个约数，必为一奇一偶。
- $n=1,2,4$ 为特殊情况，均不能被表示。

0x35 数学 离散数学

0x36 数学 线性代数

0x37 数学 概率论

0x38 数学 博弈论

0x41 数据结构 序列结构

ST 表

0x41 ST 表.cpp

```cpp
// ST 表
template <class T, auto op>
struct SparseTable {
    int t;
    std::vector<std::vector<T>> f;

    SparseTable() {}
    SparseTable(int n, const std::vector<T> &a) {
        init(n, a);
    }

    void init(int n, const std::vector<T> &a) {
        t = std::__lg(n);
        f.assign(t + 1, std::vector<T>(n + 1));
        for (int i = 1; i <= n; i ++) {
            f[0][i] = a[i];
        }
        for (int j = 1; j <= t; j ++) {
            for (int i = 1; i <= n - (1 << j) + 1; i ++) {
                f[j][i] = op(f[j - 1][i], f[j - 1][i + (1 << (j - 1))]);
            }
        }
    }

    T ask(int l, int r) {
        int k = std::__lg(r - l + 1);
        return op(f[k][l], f[k][r - (1 << k) + 1]);
    }
};
```

并查集

0x41 可撤销并查集.cpp

```cpp
// 可撤销并查集
struct DSU {
    std::vector<int> sze;
    std::vector<int> fa;
    std::vector<std::pair<int, int>> his;

    DSU() {}
    DSU(int n) {
        init(n);
    }

    void init(int n) {
        sze.assign(n + 1, 1);
        fa.resize(n + 1);
        std::iota(fa.begin(), fa.end(), 0);
        his.clear();
    }

    int get(int x) {
        return fa[x] == x ? x : get(fa[x]); // 按秩合并不能路径压缩
    }

    bool merge(int x, int y) {
        int p = get(x), q = get(y);
        if (p == q) {
            return 0;
        }
        if (sze[p] > sze[q]) {
            std::swap(p, q);
        }
        his.push_back({p, q});
        sze[q] += sze[p], fa[p] = q;
        return 1;
    }

    int time() {
        return his.size();
    }
    void revert(int tim) {
        while (his.size() > tim) {
            auto [p, q] = his.back(); his.pop_back();
            sze[q] -= sze[p], fa[p] = p;
        }
    }
};
```

线段树

扫描线

势能线段树

区间开平方：一个整数 x 开平方 $\mathcal{O}(\log\log V)$ 次之后就会变成 1。若区间 $\max = 1$ 则 `return`。

区间取模：一个整数 x $(x \geq p)$ 对 p 取模，x 至少变小一半。若区间 $\max < p$ 则 `return`。

区间除法：整除 p 之后，由于 $\lfloor \frac{\max}{p} \rfloor - \lfloor \frac{\min}{p} \rfloor \leq \lceil \frac{\max - \min}{p} \rceil$，区间最大值与最小值的差，多数情况下会减半。

- 若区间 $\max - \min = 0$，则区间加之后 `return`。
- 若区间 $\max - \min = 1$ 且 $\lfloor \frac{\max}{p} \rfloor \neq \lfloor \frac{\min}{p} \rfloor$，则区间加之后 `return`。

区间按位与：一个二进制数 x 与 v 进行按位与，至多进行 $\log V$ 次有效操作。若区间（在二进制下）$\text{bitor} \subseteq v$ 则 `return`。

区间取最值（吉老师线段树）：以区间取最小值为例：

- 在线段树上的每一个节点 p 需要维护：
  - `mx`：区间最大值。
  - `cnt`：区间最大值个数。
  - `md`：区间严格次大值。
- 在区间修改时，若遇到一个“被询问区间完全覆盖的节点”时：
  - 若 $x \geq \text{mx}$，则 `return`。
  - 若 $x \leq \text{md}$，则暴力递归左右儿子修改。
  - 若 $\text{md} < x < \text{mx}$，则有 cnt 个 mx 被修改成 x，打标记后 `return`。
- 时间复杂度 $\mathcal{O}(n \log^2 n)$。

0x41 吉老师线段树.cpp

```cpp
struct Info {
    int mx, md;
    int cnt;
```

```
    Info() {}
    Info(int mx, int md, int cnt) : mx(mx), md(md), cnt(cnt) {}

    friend Info operator + (Info l, Info r) {
        if (l.mx == r.mx) {
            return Info(l.mx, std::max(l.md, r.md), l.cnt + r.cnt);
        } else if (l.mx > r.mx) {
            return Info(l.mx, std::max(l.md, r.mx), l.cnt);
        } else {
            return Info(r.mx, std::max(l.mx, r.md), r.cnt);
        }
    }
};

// 吉老师线段树
namespace SGT {
    struct node {
        int len;
        i64 sum;
        Info info;

        int add; // 加法标记（初始值等于 0）
        int tag; // 区间取 min 标记（初始值等于 +inf，需要根据数据范围修改）
        void mk_add(int x) {
            sum += 1ll * x * len;
            info.mx += x;
            if (info.md != -inf) info.md += x;
            add += x;
            if (tag < inf) tag += x;
        }
        void mk_tag(int x) {
            if (info.mx <= x) return;
            sum -= 1ll * (info.mx - x) * info.cnt;
            info.mx = x;
            tag = x;
        }
    } t[N * 4];

    void upd(int p) {
        t[p].sum = t[p * 2].sum + t[p * 2 + 1].sum;
        t[p].info = t[p * 2].info + t[p * 2 + 1].info;
    }

    void spread(int p) {
        if (t[p].add) {
            t[p * 2].mk_add(t[p].add), t[p * 2 + 1].mk_add(t[p].add);
            t[p].add = 0;
        }
        if (t[p].tag < inf) {
            t[p * 2].mk_tag(t[p].tag), t[p * 2 + 1].mk_tag(t[p].tag);
            t[p].tag = inf;
        }
    }

    void build(int p, int l, int r) {
        t[p].add = 0, t[p].tag = inf, t[p].len = r - l + 1;
        if (l == r) {
            t[p].sum = a[l], t[p].info = Info(a[l], -inf, 1);
            return;
        }
        int mid = (l + r) >> 1;
        build(p * 2, l, mid), build(p * 2 + 1, mid + 1, r);
        upd(p);
    }

    void change_add(int p, int l, int r, int s, int e, int x) {
        if (s <= l && r <= e) {
            t[p].mk_add(x);
            return;
        }
        spread(p);
        int mid = (l + r) >> 1;
        if (s <= mid) {
            change_add(p * 2, l, mid, s, e, x);
        }
        if (mid < e) {
            change_add(p * 2 + 1, mid + 1, r, s, e, x);
        }
        upd(p);
    }

    void change_min(int p, int l, int r, int s, int e, int x) {
        if (s <= l && r <= e && t[p].info.md < x) {
            t[p].mk_tag(x);
            return;
        }
        spread(p);
        int mid = (l + r) >> 1;
        if (s <= mid) {
            change_min(p * 2, l, mid, s, e, x);
        }
        if (mid < e) {
```

```
            change_min(p * 2 + 1, mid + 1, r, s, e, x);
        }
        upd(p);
    }
}
```

笛卡尔树

区间笛卡尔树

0x42 数据结构 树上结构

0x51 动态规划 状态设计

0x52 动态规划 转移优化

0x65 图论 二分图与网络流

最大流

Dinic

0x65 最大流（Dinic）.cpp

```cpp
template <class T>
struct MaxFlow {
    struct Edge {
        int to;
        T cap;
    };

    int n;
    std::vector<Edge> e;
    std::vector<std::vector<int>> g;
    std::vector<int> d, cur;

    MaxFlow() {}
    MaxFlow(int n) {
        init(n);
    }

    void init(int n) {
        this->n = n;
        e.clear();
        g.assign(n + 1, {});
        d.resize(n + 1), cur.resize(n + 1);
    }

    void addEdge(int u, int v, T w) {
        g[u].push_back(e.size());
        e.push_back({v, w});
        g[v].push_back(e.size());
        e.push_back({u, 0});
    }

    bool bfs(int s, int t) {
        d.assign(n + 1, 0);
        std::queue<int> q;
        q.push(s), d[s] = 1;
        while (q.size()) {
            int u = q.front(); q.pop();
            for (int i : g[u]) {
                auto [v, w] = e[i];
                if (w > 0 && !d[v]) {
                    d[v] = d[u] + 1;
                    q.push(v);
                    if (v == t) {
                        return 1;
                    }
                }
            }
        }
        return 0;
    }
    T dfs(int u, int t, T f) {
        if (u == t) {
            return f;
        }
        auto r = f;
        for (int &i = cur[u]; i < g[u].size(); i ++) {
            int j = g[u][i];
            auto [v, w] = e[j];
            if (w > 0 && d[v] == d[u] + 1) {
                auto a = dfs(v, t, std::min(r, w));
                e[j].cap -= a;
                e[j ^ 1].cap += a;
                r -= a;
                if (r == 0) {
```

```
                    return f;
                }
            }
        }
        return f - r;
    }

    T dinic(int s, int t) {
        T ans = 0;
        while (bfs(s, t)) {
            cur.assign(n + 1, 0);
            ans += dfs(s, t, std::numeric_limits<T>::max());
        }
        return ans;
    }
};
```

0x71 计算几何 I

计算几何基础

0x71 计算几何全家桶（i64）.cpp

```cpp
// 二维点
struct point {
    i64 x, y;
    point() {}
    point(i64 x, i64 y) : x(x), y(y) {}
};

point operator + (point a, point b) {
    return point(a.x + b.x, a.y + b.y);
}
point operator - (point a, point b) {
    return point(a.x - b.x, a.y - b.y);
}
point operator * (point a, i64 b) {
    return point(a.x * b, a.y * b);
}

// 叉积
i128 cross(point a, point b) {
    return i128(a.x) * b.y - i128(a.y) * b.x;
}
}
// ab 与 ac 的叉积
// 检测连续三个点的凸性，按 x 递增时 >0 下凸 <0 上凸，按 x 递减时反之
i128 cross(point a, point b, point c) {
    return i128(b.x - a.x) * (c.y - a.y) - i128(c.x - a.x) * (b.y - a.y);
}
```

0x72 计算几何 II

凸壳

0x72 凸壳.cpp

```cpp
// 凸壳（默认：按 x 递增的上凸壳）
auto gethull(std::vector<point> s) {
    std::sort(s.begin(), s.end(), [] (point l, point r) -> bool {
        return l.x != r.x ? l.x < r.x : l.y > r.y; // 若要求下凸壳，改成 l.y < r.y

    });

    std::vector<point> h;
    for (point p : s) {
        if (h.size() && h.back().x == p.x) {
            continue;
        }
        while (h.size() >= 2 && cross(h.rbegin()[1], h.back(), p) >= 0) { // 若要求下凸壳，改成 <=
            h.pop_back();
        }
        h.push_back(p);
    }
    return h;
}
```

凸包

0x72 凸包.cpp

```cpp
// 凸包（默认：逆时针）
auto gethull(std::vector<point> s) {
    std::sort(s.begin(), s.end(), [] (point l, point r) -> bool {
        return l.x != r.x ? l.x < r.x : l.y < r.y;
    });
    s.erase(std::unique(s.begin(), s.end(), [] (point l, point r) -> bool {
        return l.x == r.x && l.y == r.y;
    }), s.end());

    std::vector<point> h;
    for (int i = 0; i < s.size(); i ++) {
        while (h.size() >= 2 && cross(h.rbegin()[1], h.back(), s[i]) <= 0) {
            h.pop_back();
        }
        h.push_back(s[i]);
    }
    for (int i = (int)s.size() - 2, t = h.size(); i >= 0; i --) {
        while (h.size() > t && cross(h.rbegin()[1], h.back(), s[i]) <= 0) {
            h.pop_back();
        }
        h.push_back(s[i]);
    }
    if (h.size() > 1) {
        h.pop_back();
    }
    return h;
}
```

闵可夫斯基和

0x72 闵可夫斯基和.cpp

```cpp
// 闵可夫斯基和（默认：逆时针）
auto minkowski(std::vector<point> a, std::vector<point> b) {
    if (a.size() <= 2 || b.size() <= 2) { // 若 a, b 均为凸壳，可以省去这一段特判
        std::vector<point> c;
        for (point p : a) for (point q : b) c.push_back(p + q);
        return gethull(c);
    }

    std::vector<point> c{a[0] + b[0]};
    a.push_back(a[0]), b.push_back(b[0]); // 若 a, b 均为凸壳，省去此行
    for (int i = 0; i + 1 < a.size(); i ++) {
        a[i] = a[i + 1] - a[i];
    }
    for (int i = 0; i + 1 < b.size(); i ++) {
        b[i] = b[i + 1] - b[i];
    }
    a.pop_back(), b.pop_back();
    c.resize(a.size() + b.size() + 1);
    std::merge(a.begin(), a.end(), b.begin(), b.end(), c.begin() + 1, [](point l, point r) {
        return cross(l, r) > 0; // 若 a, b 均为顺时针凸包（或凸壳），改成 <0
    });
    for (int i = 1; i < c.size(); i ++) {
        c[i] = c[i] + c[i - 1];
    }
    c.pop_back(); // 若 a, b 均为凸壳，省去此行
    return c; // c 可能会共线，但不会重点
}
```