WXS's 算法模板

Version 25w16b, last built at 2025/4/24

WXS's 算法模板

数学

位运算

数的二进制位

- **获取数的某一位**

```
// 获取 a 的第 b 位
int getBit(int a, int b) { return (a >> b) & 1; }
```

- **将数的某一位设为0**

```
// 将 a 的第 b 位设置为 0
int unsetBit(int a, int b) { return a & ~(1 << b); }
```

- **将数的某一位设为1**

```
// 将 a 的第 b 位设置为 1
int setBit(int a, int b) { return a | (1 << b); }
```

- **将数的某一位取反**

```
// 将 a 的第 b 位取反
int flapBit(int a, int b) { return a ^ (1 << b); }
```

内建函数

函数	作用
`int __builtin_ffs(int x)`	返回 x 的二进制末尾最后一个 1 的位置，位置的编号从 1 开始（最低位编号为 1 ）。当 x 为 0 时返回 0
`int __builtin_clz(unsigned int x)`	返回 x 的二进制的前导 0 的个数。当 x 为 0 时，结果未定义
`int __builtin_ctz(unsigned int x)`	返回 x 的二进制末尾连续 0 的个数。当 x 为 0 时，结果未定义
`int __builtin_clrsb(int x)`	当 x 的符号位为 0 时返回 x 的二进制的前导 0 的个数减一，否则返回 x 的二进制的前导 1 的个数减一
`int __builtin_popcount(unsigned int x)`	返回 x 的二进制中 1 的个数
`int __builtin_parity(unsigned int x)`	判断 x 的二进制中 1 的个数的奇偶性

可在函数名末尾添加 `l` 或 `ll` （如 `__builtin_popcountll` ）来使参数类型变为 (`unsigned`) `long` 或 (`unsigned`) `long long` , 返回值仍然是 `int` 类型。

二进制集合操作

模 2 的幂

一个数对 2 的非负整数次幂取模，等价于取二进制下一个数的后若干位，等价于和 $mod-1$ 进行与操作。

```
int modPower2(int x, int p) {
    return x & (p - 1);
}
```

判断一个数是不是的非负整数次幂。当且仅当的二进制表示只有一个时，为的非负整数次幂。

```
bool isPowerOf2(int x) {
    return x > 0 && (x & (x - 1) == 0);
}
```

子集遍历

遍历一个二进制数表示的集合的全部子集，等价于枚举二进制数对应掩码的所有子掩码。

```
for (int s = m; s; s = (s - 1) & m)
    // s 是 m 的一个非空子集
```

快速幂

```
constexpr ll power(ll a, ll b, ll p) {
    ll res = 1;
    for (; b; b /= 2, a = a * a % p) {
        if (b % 2) {
            res = res * a % p;
        }
    }
    return res;
}
```

数论

数论基础

整除

> 设 $a,b\in\mathbf{Z}$，$a\neq 0$。如果 $\exists q\in\mathbf{Z}$，使得 $b=aq$，那么就说 b 可被 a **整除**，记作 $a\mid b$；b 不被 a 整除记作 $a\nmid b$。

整除的性质：

- $a\mid b\iff -a\mid b\iff a\mid -b\iff |a|\mid |b|$
- $a\mid b\wedge b\mid c\implies a\mid c$
- $a\mid b\wedge a\mid c\iff \forall x,y\in\mathbf{Z},a\mid (xb+yc)$
- $a\mid b\wedge b\mid a\implies b=\pm a$

- 设 $m \neq 0$，那么 $a \mid b \iff ma \mid mb$。
- 设 $b \neq 0$，那么 $a \mid b \implies |a| \leq |b|$。
- 设 $a \neq 0, b = qa + c$，那么 $a \mid b \iff a \mid c$。

最大公约数

- **欧几里得算法**：$\gcd(a, b) = \gcd(b, a \bmod b)$

```cpp
int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    return gcd(b, a % b);
}
```

- **更相减损术**：$a > b, \gcd(a, b) = \gcd(a - b, b)$.
- **Stein 算法**

  若 a, b 为偶数，$\gcd(a, b) = \gcd(\frac{a}{2}, \frac{b}{2})$.

  若 a 为偶数，b 为奇数，$\gcd(a, b) = \gcd(\frac{a}{2}, b)$, 同理。

  若 a, b 为奇数，$\gcd(a, b) = \gcd(|a - b|, a)$.

最小公倍数

```cpp
int lcm(int a, int b) {
    return a * b / gcd(a, b);
}
```

C++17 可以使用 `<numeric>` 头中的 `std::gcd` 与 `std::lcm` 来求最大公约数和最小公倍数。

数论函数

定义域为正整数的函数称为数论函数。

- **积性函数**

  若函数 $f(n)$ 满足 $f(1) = 1$，且 $f(xy) = f(x)f(y)$ 对任意互质的 $x, y \in \mathbf{N}^*$ 都成立，则 $f(n)$ 为 **积性函数。**

  若函数 $f(n)$ 满足 $f(1) = 1$ 且 $f(xy) = f(x)f(y)$ 对任意的 $x, y \in \mathbf{N}^*$ 都成立，则 $f(n)$ 为 **完全积性函数。**

  积性函数性质：

  若 $f(x)$ 和 $g(x)$ 均为积性函数，则

$$
\begin{aligned}
h(x) &= f(x^p) \\
h(x) &= f^p(x) \\
h(x) &= f(x)g(x) \\
h(x) &= \sum_{d|x} f(d)g\left(\frac{x}{d}\right)
\end{aligned}
$$

  都为积性函数。

  设正整数 x 的其唯一质因数分解为 $x = \prod p_i^{k_i}$, 若 $F(x)$ 为积性函数，则有 $F(x) = \prod F(p_i^{k_i})$; 若 $F(x)$ 为完全积性函数，则有 $F(x) = \prod F(p_i^{k_i}) = \prod F(p_i)^{k_i}$.

常见积性函数：

除数函数：$\sigma_k(n) = \sum_{d|n} d^k$. $\sigma_0(n)$ 通常简记作 $d(n)$ 或 $\tau(n)$, $\sigma_1(n)$ 通常简记作 $\sigma(n)$.

欧拉函数：$\varphi(n)$.

莫比乌斯函数：$\mu(n)$.

常见完全积性函数：

单位函数：$\varepsilon(n) = [n = 1]$.

恒等函数：$\mathrm{id}_k(n) = n^k$, $\mathrm{id}_1(n)$ 通常简记作 $\mathrm{id}(n)$.

常数函数：$1(n) = 1$.

素数

素数计数函数

用 $\pi(x)$ 表示小于或等于 x 的素数的个数。随着 x 的增大，有这样的近似结果：$\pi(x) \sim \dfrac{x}{\ln(x)}$.

素性测试

- **试除法**，复杂度 $O(n)$.

```cpp
bool isPrime(int x) {
    if (x < 2) {
        return false;
    }
    for(int i = 2; i <= x / i; i++) {
        if (x % i == 0) {
            return false;
        }
    }
    return true;
}
```

- **Miller–Rabin 素性测试**，复杂度 $O(k \log^3 n)$.

  $n = 1e9, O(k \log^3 n) \approx O(7)$

  $n = 1e18, O(k \log^3 n) \approx O(18)$

```cpp
using ull = unsigned long long;
using i128 = __int128;

constexpr ll base32[] { 2, 7, 61 };
constexpr ll base64[] { 2, 325, 9375, 28178, 450775, 9780504, 1795265022
};

// 快速乘 int128版
ll mul(ll a, ll b, ll p) {
    return static_cast<i128>(a) * b % p;
}
// 快速乘 无int128版
ll mul(ll a, ll b, ll p) {
    ll c = (long double)a / p * b;
```

```
    ll res = (ull)a * b - (ull)c * p;
    return (res + p) % p;
}
// 使用快速乘的快速幂
ll power(ll a, ll b, ll p) {
    ll res = 1;
    for (; b; b /= 2, a = mul(a, a, p)) {
        if (b % 2) {
            res = mul(res, a, p);
        }
    }
    return res;
}
bool millerRabin(ll n) {
    if (n < 3 || n % 2 == 0) {
        return n == 2;
    }

    ll u = n - 1, t = 0;
    while (u % 2 == 0) { // 将u处理为奇数
        u /= 2;
        t++;
    }
    // int范围使用base32, k=3; long long范围使用base64, k=7
    for (ll x : base32) {
        ll v = power(x, u, n);
        if (v == 0 || v == 1 || v == n - 1) {
            continue;
        }
        for (int j = 1; j <= t; j++) {
            v = mul(v, v, n);
            if (v == n - 1 && j != t) {
                v = 1;
                break;
            }
            if (v == 1) {
                return false;
            }
        }
        if (v != 1) {
            return false;
        }
    }
    return true;
}
```

素数筛法

- **埃拉托斯特尼筛法**，复杂度 $O(n \log \log n)$.

```
constexpr int N = 1e7;
int p[N], pcnt;
bool pst[N]; // x为素数则 st[x] = 0

void sieve(int n) {
```

```
    for (int i = 2; i <= n; i++) {
        if (pst[i]) {
            continue;
        }
        p[pcnt++] = i;
        for (int j = i + i; j <= n; j += i) {
            pst[j] = true;
        }
    }
}
```

- **线性筛法**，复杂度 $O(n)$.

```
constexpr int N = 1e7;
int p[N], pcnt;
bool pst[N]; // x 为素数则 st[x] = 0

void sieve(int n) {
    for (int i = 2; i <= n; i++) {
        if (!pst[i]) {
            p[pcnt++] = i;
        }
        for (int j = 0; p[j] <= n / i; j++) {
            pst[p[j] * i] = true;
            if (i % p[j] == 0) {
                break;
            }
        }
    }
}
```

分解质因数

- **试除法**分解质因数，从 $[2, \sqrt{n}]$ 进行遍历，复杂度 $O(\sqrt{n})$.

```
vector<int> factor;

void divide(int x) {
    for (int i = 2; i <= x / i; i++) {
        if (x % i == 0) {
            int s = 0;
            while (x % i == 0) {
                x /= i;
                s++;
            }
            factor.push_back(s);
        }
    }
    if (x > 1) {
        factor.push_back(x);
    }
}
```

预处理素数表，可将复杂度将降到 $O(\sqrt{\frac{n}{\ln n}})$.

- **Pollard Rho 算法**

  随机化算法，可以在 $O(\sqrt{p}) = O(n^{1/4})$ 的期望复杂度获得一个非平凡因子（不一定是素因子）。

```cpp
// millerRabin 素性测试略，记得使用base64
// 随机数生成，范围[l,r]
template <class T>
T randint(T l, T r = 0) {
    static mt19937
eng(chrono::steady_clock::now().time_since_epoch().count());
    if (l > r) {
        swap(l, r);
    }
    uniform_int_distribution<T> dis(l, r);
    return dis(eng);
}

ll pollardRho(ll n) {
    if (n == 4) { // 特判 4
        return 2;
    }
    if (millerRabin(n)) { // 特判素数
        return n;
    }

    while (true) {
        ll c = randint<ll>(1, n - 1);
        auto f = [&](ll x) -> ll {
            return (mul(x, x, n) + c) % n;
        };
        ll t = 0, r = 0, p = 1, q = 0;
        do {
            for (int i = 0; i < 128; i++) {
                t = f(t), r = f(f(r));
                if (t == r || (q = mul(p, abs(t - r), n)) == 0) {
                    break;
                }
                p = q;
            }
            ll d = gcd(p, n);
            if (d > 1) {
                return d;
            }
        } while (t != r);
    }
}
```

- 使用 **Pollard Rho** 求最大质因数，复杂度约 $O(n^{1/4})$.

```cpp
ll maxfac = 0;
void maxFactor(ll x) {
    if (x <= maxfac || x < 2) {
        return;
    }
    if (millerRabin(x)) { // x为质数
```

```
        maxfac = max(maxfac, x);
        return;
    }
    ll p = x;
    while (p >= x) {
        p = pollardRho(x);
    }
    while ((x % p) == 0) {
        x /= p;
    }
    maxFactor(x), maxFactor(p); // 继续向下分解x和p
}
```

欧拉函数

$\varphi(n)$ 表述 $[1,n]$ 中与 n 互质的数。

$$\varphi(n)=n\cdot\prod_{i=1}^{s}(1-\frac{1}{p_i})=n\cdot\prod_{i=1}^{s}(\frac{p_i-1}{p_i}),p_i为n的所有质因数,n\in\mathbb{N}_+$$

欧拉函数性质

1. p 是素数时，$\varphi(p)=p-1,\varphi(kp)=k(p-1),\varphi(p^k)=p^k-p^{k-1}$
2. 欧拉函数是积性函数，$\forall a,b\in Z,(a,b)=1$，有 $\varphi(ab)=\varphi(a)\varphi(b)$
3. $\varphi(n)=\prod_{i=1}^{m}\varphi(p_i^{k_i})$
4. 若 p 是素数，$\varphi(i\cdot p)=\begin{cases}\varphi(i)\cdot(p-1) & ,p\nmid i\\ \varphi(i)\cdot p & ,p\mid i\end{cases}$
5. $n=\sum_{d|n}\varphi(d)$

求单个数的欧拉函数值：直接根据定义质因数分解来求，可用 Pollard Rho 算法优化。

```
int eulerPhi(int n) {
    int phi = n;
    for (int i = 2; i <= n / i; i++) {
        if (n % i == 0) {
            phi = phi / i * (i - 1);
            while (n % i == 0) {
                n /= i;
            }
        }
    }
    if (n > 1) {
        phi = phi / n * (n - 1);
    }
    return phi;
}
```

筛法求欧拉函数

在线性筛中，每一个合数都是被最小的质因子筛掉。设 p_1 是 n 的最小质因子，$n'=\frac{n}{p_1}$，那么线性筛的过程中 n 通过 $n'\times p_1$ 筛掉。

如果 $n'\bmod p_1=0$, 那么 n' 包含了 n 的所有质因子。则

$$\varphi(n) = n \times \prod_{i=1}^{s} \frac{p_i - 1}{p_i} = p_1 \times n' \times \prod_{i=1}^{s} \frac{p_i - 1}{p_i} = p_1 \times \varphi(n')$$

如果 $n' \bmod p_1 \neq 0$, 那么 n' 和 p_1 是互质的，则

$$\varphi(n) = \varphi(p_1) \times \varphi(n') = (p_1 - 1) \times \varphi(n')$$

```cpp
constexpr int N = 1e7;
int p[N], pcnt;
int phi[N];
bool pst[N];

void eulerPhi(int n) {
    phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (!pst[i]) {
            p[pcnt++] = i;
            phi[i] = i - 1;
        }
        for (int j = 0; p[j] <= n / i; j++) {
            int t = p[j] * i;
            pst[t] = true;

            if (i % p[j] == 0) {
                phi[t] = phi[i] * p[j];
                break;
            }
            phi[t] = phi[i] * (p[j] - 1);
        }
    }
}
```

欧拉反演

将 $n = \gcd(a, b)$ 代入 $n = \sum_{d|n} \varphi(d)$ 中，可得到

$$\gcd(a, b) = \sum_{d|\gcd(a,b)} \varphi(d) = \sum_{d} [d|a][d|b]\varphi(d)$$

其中 $[\cdot]$ 为 Iverson 括号，只有当命题 P 为真时，$[P]$为1, 否则为0.

可利用所示结论化简一列最大公约数的和。

> **Luogu P2398 GCD SUM**
> 求 $\sum_{i=1}^{n} \sum_{j=1}^{n} \gcd(i, j)$
>
> $$\begin{aligned}\sum_{i=1}^{n} \sum_{j=1}^{n} \gcd(i, j) &= \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{d|gcd(i,j)} \varphi(d) \\ &= \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{d|i,d|j} \varphi(d) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{d=1}^{n} \varphi(d)[d|i][d|j] \\ &= \sum_{d=1}^{n} \varphi(d) \sum_{i=1}^{n} \sum_{j=1}^{n} [d|i][d|j] = \sum_{d=1}^{n} \varphi(d) \lfloor \frac{n}{d} \rfloor^2\end{aligned}$$

逆元

记 a 在模 p 意义下的逆元为 $x = a^{-1}$.

- **快速幂法**：根据费马小定理，得 $x \equiv a^{p-2} \pmod{p}$.
- **扩展欧几里得法**：改写为 $ax + py = 1$

```
int inv(int a, int p) {
    int x, y;
    exgcd(a, p, x, y);
    return x;
}
```

- **线性求任意 n 个数的逆元**

  求 n 个数 $a_1, a_2, \cdots, a_n$ 的逆元，先计算 n 个数的前缀积，记为 s_i, 然后计算 s_n 的逆元，记为 sv_n.

  sv_i 是前 i 个数的积的逆元，当乘上 a_i 时，就会和 a_i 的逆元抵消，于是就得到了 a_1 到 a_{i-1} 的积逆元 sv_{i-1}.

  计算出所有的 sv_i, a_i^{-1} 就可以用 $s_{i-1} \times sv_i$ 求得。

```
constexpr int N = 1e7;
ll a[N], s[N], sv[N], inv[N];

void init(int n) {
    s[0] = 1;
    for (int i = 1; i <= n; i++) {
        s[i] = s[i - 1] * a[i] % P;
    }
    sv[n] = power(s[n], P - 2, P);
    for (int i = n; i; i--) {
        sv[i - 1] = sv[i] * a[i] % P;
    }
    for (int i = 1; i <= n; i++) {
        inv[i] = sv[i] * s[i - 1] % P;
    }
}
```

线性同余方程

形如 $ax \equiv b \pmod{n}$ 的方程称为线性同余方程，a、b 和 n 为给定整数，x 为未知数。需要从区间 $[0, n-1]$ 中求解 x，当解不唯一时需要求出全体解。

- **逆元求解**

  当 a 和 n 互素时，可得到唯一解 $x \equiv ba^{-1} \pmod{n}$.

- **扩展欧几里得算法求解**

  a 和 n 不互素时，将原方程改写为线性不定方程 $ax + nk = b$, x 和 k 是未知数。有整数解的充要条件为 $\gcd(a, n) \mid b$.

  使用扩展欧几里得算法求出一组解 x_0, k_0, 即 $ax_0 + nk_0 = \gcd(a, n)$, 变换为

$$a\frac{b}{\gcd(a,n)}x_0+n\frac{b}{\gcd(a,n)}k_0=b$$

```cpp
// exgcd 略
// 求 a*x % b = c 的特解
bool liEu(int a, int b, int c, int& x, int& y) {
  int d = exgcd(a, b, x, y);
  if (c % d != 0) {
      return false;
  }
  int k = c / d;
  x *= k;
  y *= k;
  return true;
}
// 求 a*x % b = c 的最小正整数解
int maxAns(int a, int b, int c) {
    int x, y;
    liEu(a, b, c, x, y);
    return (x % b + b) % b;
}
```

数论分块

快速计算一些含有除法向下取整的和式，形如 $\sum_{i=1}^{n} f(i)g(\left\lfloor \frac{n}{i} \right\rfloor)$. 当可以在 $O(1)$ 内计算 $f(r)-f(l)$ 或已经预处理出 f 的前缀和时，数论分块就可以在 $O(\sqrt{n})$ 的时间内计算和式的值。

- **引理 1**

$$\forall a,b,c\in\mathbb{Z},\left\lfloor \frac{a}{bc} \right\rfloor=\left\lfloor \frac{\left\lfloor \frac{a}{b} \right\rfloor}{c} \right\rfloor$$

- **引理 2**

$$\forall n\in\mathbb{N}_+,\left|\left\{\left\lfloor \frac{n}{d} \right\rfloor \mid d\in\mathbb{N}_+,d\le n\right\}\right|\le\lfloor 2\sqrt{n}\rfloor$$

- **结论**

  对于常数 n，使得式子

$$\left\lfloor \frac{n}{i} \right\rfloor=\left\lfloor \frac{n}{j} \right\rfloor$$

  成立且满足 $i\le j\le n$ 的 j 值最大为 $\left\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \right\rfloor$，即值 $\left\lfloor \frac{n}{i} \right\rfloor$ 所在块的右端点为 $\left\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \right\rfloor$.

  考虑和式 $\sum_{i=1}^{n} f(i)\left\lfloor \frac{n}{i} \right\rfloor$，先求出 $f(i)$ 的前缀和，然后每次以 $[l,r]=[l,\left\lfloor \frac{n}{\lfloor \frac{n}{i} \rfloor} \right\rfloor]$ 为一块，分块求出贡献累加到结果。

```cpp
int l = 1, r;
ll ans = 0;
while (l <= n) {
    r = n / (n / l); // 计算当前块的右端点
    ans += (s[r] - s[l - 1]) * (n / l); // s[i]为f[i]的前缀和
    r = l + 1;
}
```

> **Luogu P2261**
>
> 给出正整数 n 和 k，计算 $G(n,k)=\sum_{i=1}^{n} k \bmod i$.
>
> $$\begin{aligned} G(n,k) &= \sum_{i=1}^{n}(k-i\lfloor\frac{k}{i}\rfloor) \\ &= \max(0,n-k)\times k+\sum_{i=1}^{min(n,k)}(k-i\lfloor\frac{k}{i}\rfloor) \\ &= \max(0,n-k)\times k+\min(n,k)\times k-\sum_{i=1}^{min(n,k)} i\lfloor\frac{k}{i}\rfloor \end{aligned}$$

莫比乌斯反演

莫比乌斯函数

$$\mu(n)=\begin{cases}1 & n=1\\ 0 & n\text{ 含有平方因子}\\ (-1)^k & k\text{ 为 }n\text{ 的本质不同质因子个数}\end{cases}$$

性质

$$\sum_{d|n}\mu(d)=\begin{cases}1 & n=1\\ 0 & n\neq 1\end{cases}$$

> 将 n 的所有质因子去重，得到 n'. 那么
>
> $$\sum_{d|n}\mu(d)=\sum_{d|n'}\mu(d)=\sum_{i=0}^{k}\binom{k}{i}\cdot(-1)^i=\sum_{i=0}^{k}\binom{k}{i}\cdot(-1)^i\cdot 1^{k-i}=(1+(-1))^k=0^k$$

数论定理

费马小定理

若 p 为素数，$\gcd(a,p)=1$, 则 $a^{p-1}\equiv 1\pmod p$. 等价于对于任意整数 a, 有 $a^p\equiv a\pmod p$.

欧拉定理

若 $\gcd(a,m)=1$, 则 $a^{\varphi(m)}\equiv 1\pmod m$.

扩展欧几里得定理

求方程 $ax+by=gcd(a,b)$ 的一组可行解。

```cpp
int exgcd(int a, int b, int &x, int &y) {
    if (!b) {
        x = 1, y = 0;
        return a;
    }
    int d = exgcd(b, a % b, y, x);
    y -= (a / b) * x;
    return d;
}
```

裴蜀定理

设 a, b 是不全为零的整数，对任意整数 x, y. 满足 $\gcd(a, b) \mid ax + by$, 且存在整数 x, y, 使得 $ax + by = \gcd(a, b)$.

逆定理

设 a, b 是不全为零的整数，若 $d > 0$ 是 a, b 的公因数，且存在整数 x, y, 使得 , 则 $d = \gcd(a, b)$.

特殊地，设 a, b 是不全为零的整数，若存在整数 x, y, 使得 $ax + by = 1$, 则 a, b 互质。

裴蜀定理可以推广到 n 个整数的情形。

中国剩余定理

用于求解如下形式的一元线性同余方程组（其中 $n_1, n_2, \cdots, n_k$ 两两互质）

$$\begin{cases} x & \equiv a_1 \pmod{n_1} \\ x & \equiv a_2 \pmod{n_2} \\ & \vdots \\ x & \equiv a_k \pmod{n_k} \end{cases}$$

记 $N = \prod_{i=1}^{k} n_i, m_i = \frac{N}{n_i}, m_i^{-1}$ 是 m_i 在模 n 意义下的逆元。则方程组在模 n 意义下的唯一解为

$$x = \sum_{i=1}^{k} a_i m_i m_i^{-1} \pmod{n}$$

```cpp
ll CRT(int n, vector<ll>& a, vector<ll>& r) {
    ll res = 0, R = 1;
    for (int i = 0; i < n; i++) {
        R *= r[i];
    }
    for (int i = 0; i < n; i++) {
        ll m = R / r[i], inv, tmp;
        exgcd(m, r[i], inv, tmp); // m * inv % r[i] = 1
        res = (res + a[i] * m * inv % R) % R;
    }
    return (res + R) % R;
}
```

组合数学

排列组合

组合数性质

- $\binom{n}{k}=\frac{n}{k}\binom{n-1}{k-1}$
- $\binom{n}{m}=\binom{n-1}{m}+\binom{n-1}{m-1}$
- $\sum_{k=0}^{n}\binom{n}{k}=2^n, n\in N$
- $\sum_{k=0}^{n}(-1)^k\binom{n}{k}=0, n\in N$
- $\sum_{l=0}^{n}\binom{l}{k}=\binom{n+1}{k+1}, n,k\in N$
- $\sum_{i=0}^{m}\binom{n}{i}\binom{m}{m-i}=\binom{m+n}{m}, n\geq m$
- $\sum_{i=0}^{n}\binom{n}{i}^2=\binom{2n}{n}$
- $\binom{n}{r}\binom{r}{k}=\binom{n}{k}\binom{n-k}{r-k}, n\geq r\geq k, n,r,k\in N$
- $\sum_{k=0}^{r}\binom{m}{k}\binom{n}{r-k}=\binom{n+m}{r}, n,m,r\in N, r\leq \min(m,n)$
- $\sum_{k=0}^{n}\binom{m}{k}\binom{n}{k}=\binom{m+n}{m}, m,n\in N$
- $\sum_{i=0}^{n}i\binom{n}{i}=n2^{n-1}$
- $\sum_{i=0}^{n}\binom{n-i}{i}=F_{n+1}$

插板法

n 个完全相同的元素，将其分为 k 组，保证每组至少有一个元素，有 $\binom{n-1}{k-1}$ 选法。

若每组允许为空，可先借 k 个元素，使每组至少有一个元素，插完板后再拿走，有 $\binom{n+k-1}{k-1}=\binom{n+k-1}{n}$ 选法。

> 本质是求 $x_1+x_2+\cdots+x_k=n$ 的非负整数解的组数

若对于第 i 组，至少要分到 $a_i, \sum a_i\leq n$ 个元素，则先借 $\sum a_i$ 个元素，有 $\binom{n-\sum a_i+k-1}{n-\sum a_i}$ 选法。

> 本质是求 $x_1+x_2+\cdots+x_k=n$ 的解的数目，其中 $x_i\geq a_i$.

$1\sim n$ 这 n 个自然数中选 k 个，这 k 个数中任何两个数都不相邻的组合有 $\binom{n-k+1}{k}$ 种。

二项式定理

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

扩展为多项式的形式

$$(x_1+x_2+\cdots+x_t)^n=\sum_{\text{满足}n_1+\cdots+n_t=n\text{的非负整数解}}\binom{n}{n_1,n_2,\cdots,n_t}x_1^{n_1}x_2^{n_2}\cdots x_t^{n_t}$$

多重集的排列数

多重集是指包含重复元素的广义集合。设 $S=\{n_1\cdot a_1,n_2\cdot a_2,\cdots,n_k\cdot a_k\}$ 表示由 n_1 个 a_1, n_2 个 a_2, ..., n_k 个 a_k 组成的多重集，S 的全排列个数为

$$\binom{n}{n_1,n_2,\cdots,n_t}=\frac{n!}{n_1!,n_2!,\cdots,n_t!}$$

多重集的排列数常被称作 **多重组合数。**

多重集的组合数

从多重集 $S=\{n_1\cdot a_1,\cdots n_k\cdot a_k\}$ 选 r 个元素的多重集组合数，为 $x_1+\cdots+x_k=r$ 的非负整数解数目。

若 $r<\min n_i$, 由插板法知解为 $\binom{r+k-1}{k-1}$.

若无限制条件，则解为

$$\sum_{p=0}^{k}(-1)^p\sum_{A}\binom{k+r-1-\sum_A n_{A_i}-p}{k-1}$$

A 是枚举子集，满足 $|A|=p, A_i<A_{i+1}$.

圆排列

n 个人围成一圈，所有的排列数记为 Q_n^n, 则

$$\mathrm{Q}_n^r=\frac{\mathrm{A}_n^r}{r}=\frac{n!}{r\times(n-r)!}$$

抽屉原理

将 n 个物品划分为 k 组，至少存在一个分组包含大于等于 $\lceil\frac{n}{k}\rceil$ 个物品。

容斥原理

设 U 中元素有 n 种不同的属性，称第 i 种属性为 P_i, 拥有属性 P_i 的元素构成集合 S_i, 则

$$\begin{aligned}\left|\bigcup_{i=1}^{n}S_i\right|&=\sum_i|S_i|-\sum_{i<j}|S_i\cap S_j|+\sum_{i<j<k}|S_i\cap S_j\cap S_k|-\cdots\\&\quad+(-1)^{m-1}\sum_{a_i<a_{i+1}}\left|\bigcap_{i=1}^{m}S_{a_i}\right|+\cdots+(-1)^{n-1}|S_1\cap\cdots\cap S_n|\\&=\sum_{m=1}^{n}(-1)^{m-1}\sum_{a_i<a_{i+1}}\left|\bigcap_{i=1}^{m}S_{a_i}\right|\end{aligned}$$

全集 U 下的集合的交可用全集减去补集的并集求得：

$$\left|\bigcap_{i=1}^{n} S_i\right| = |U| - \left|\bigcup_{i=1}^{n} \overline{S_i}\right|$$

不定方程非负整数解

给出不定方程 $\sum_{i=1}^{n} x_i = m$ 和 n 个限制条件 $x_i \le b_i$，其中 $m, b_i \in \mathbb{N}$. 求方程的非负整数解的个数。

若无 $x_i < b_i$ 限制，插板法即可。

有限制的情况下，尝试抽象出容斥原理的模型：U 对于不定方程的所有解，元素对于方程变量 x_i, 属性 P_i 对于限制条件 $x_i < b_i$.

则非负整数解的个数可表示为 $\left|\bigcap_{i=1}^{n} S_i\right| = |U| - \left|\bigcup_{i=1}^{n} \overline{S_i}\right|$, 其中 $|U|$ 可用组合数计算。

考虑 $\overline{S_{a_i}}$ 的含义，表示同时满足条件 $x_{a_i} \ge b_{a_i} + 1$ 的解的数目。因此这个交集对应的不定方程中，有些变量有下界限制，而有些则没有限制。因为要求的是非负整数解，则直接把所有大于 0 的下界减掉，就可使得所有变量的下界变成 0. 因此 $\left|\bigcap_{a_i<a_{i+1}}^{1\le i\le k} S_{a_i}\right|$ 的不定方程形式为 $\sum_{i=1}^{n} x_i = m - \sum_{i=1}^{k}(b_{a_i}+1)$. 也可使用组合数计算。

> **Luogu P1450 硬币购物**
>
> 4 种面值的硬币，第 i 种的面值是 C_i. n 次询问，每次询问给出每种硬币的数量 D_i 和一个价格 S, 问付款方式。$n \le 10^3, S \le 10^5$.
>
> 转化为求不定方程 $\sum_{i=1}^{4} C_i x_i = S, x_i \le D_i$ 非负整数解的个数。
>
> 先预处理无限背包 f, 每次询问的答案为 $f[s] - sum$. 其中 sum 是不定方程 $\sum_{i=1}^{4} C_i x_i = S - \sum_{i=1}^{k} C_{a_i}(D_{a_i}+1)$ 所有解的个数，也可以使用预处理好的 f 求出。

```
constexpr int S = 1e5 + 5;
ll f[S];
ll c[5], d[5];

void solve() {
    ll s, t;
    for (int i = 1; i <= 4; i++) {
        cin >> d[i];
    }
    cin >> s;

    ll sum = 0;
    for (int i = 1; i < 16; i++) {
        t = s;
        int cnt = 0;
        for (int j = 0; j < 4; j++) { // 枚举所有子集
            if ((i >> j) & 1) {
                cnt++;
                t -= c[j + 1] * (d[j + 1] + 1);
            }
        }
```

```
        if (t >= 0) {
            sum += f[t] * (cnt % 2 ? 1 : -1);
        }
    }
    cout << f[s] - sum << '\n';
}

int main() {
    int t;
    for (int i = 1; i <= 4; i++) {
        cin >> c[i];
    }
    cin >> t;
    // 预处理无限背包
    f[0] = 1;
    for (int i = 1; i <= 4; i++) {
        for (int j = c[i]; j < S; j++) {
            f[j] += f[j - c[i]];
        }
    }
    while (t--) {
        solve();
    }
    return 0;
}
```

求组合数

递推法

复杂度 $O(n^2)$

```
constexpr int N = 1e3;
ll comb[N][N];

void init(int n) {
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= i; j++) {
            if (!j) {
                comb[i][j] = 1;
            } else {
                comn[i][j] = comb[i - 1][j] + comb[i - 1][j - 1];
            }
        }
    }
}
```

预处理逆元

要求模数为素数，复杂度 $O(n)$

```
constexpr int N = 1e7;
ll fac[N], invfac[N];

```

```cpp
// 快速幂略
void init(int n) {
    fac[0] = 1;
    for (int i = 1; i <= n; i++) {
        fac[i] = fac[i - 1] * i % P;
    }
    invfac[n] = power(fac[n], P - 2, P);
    for (int i = n; i; i--) {
        invfac[i - 1] = invfac[i] * i % P;
    }
}

ll comb(int a, int b) {
    return fac[a] * invfac[b] % P * invfac[a - b] % P;
}
```

卢卡斯定理

对于素数 p, 有

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

其中，当 $n < k$ 时，二项式系数 $\binom{n}{k}$ 规定为 0.

复杂度为 $O(f(p) + g(p) \log_p n)$, 其中， $f(p)$ 为预处理组合数的复杂度，$g(p)$ 为单次计算组合数的复杂度。

```cpp
// 快速幂略
// 此处使用定义求组合数，也可使用其他方法
ll comb(int a, int b, int p) {
    if (a < b) {
        return 0;
    }
    ll x = 1, y = 1;
    for (int i = a, j = 1; j <= b; i--, j++) {
        x = x * i % p;
        y = y * i % p;
    }
    return x * power(y, p - 2, p) % p;
}

ll lucas(int a, int b, int p) {
    if (a < p && b < p) {
        return comb(a, b);
    }
    return comb(a % p, b % p) * lucas(a / p, b / p, p) % p;
}
```

卡特兰数

Catalan 数列前 7 项 $H_0 \sim H_6$ 为 $1, 1, 2, 5, 14, 42, 132$. H_{36} 会爆 long long.

H_n 的递推式为

$$H_n = \sum_{i=0}^{n-1} H_i H_{n-i-1} \quad (n \geq 2)$$

$$H_n = \begin{cases} \sum_{i=1}^{n} H_{i-1} H_{n-i} & n \geq 2, n \in \mathbf{N}_+ \\ 1 & n = 0, 1 \end{cases}$$

$$H_n = \frac{H_{n-1}(4n-2)}{n+1}$$

通项公式为

$$H_n = \frac{\binom{2n}{n}}{n+1} (n \geq 2, n \in \mathbf{N}_+)$$

$$H_n = \binom{2n}{n} - \binom{2n}{n-1}$$

应用

- 一个无穷大的栈，进栈序列为 $1, 2, 3, \cdots, n$, 有 H_n 个不同的出栈序列
- 矩阵连乘 $P = a_1 \times a_2 \times a_3 \times \cdots \times a_n$，依据乘法结合律，不改变其顺序，只用括号表示成对的乘积，括号化的方案数
- 对角线不相交的情况下，将一个凸多边形区域分成三角形区域的方法数
- 圆 $2n$ 点用 n 条不相交线段连接的方案数
- 给定 n 对括号，求括号正确配对的字符串数
- 给定 n 个节点，能构成多少不同的二叉搜索树
- n 个 $+1$ 和 n 个 -1 构成 $2n$ 项 $a_1, \cdots, a_{2n}$ 满足前缀和 $a_1 + \cdots + a_k > 0 (k = 1, 2, \cdots 2n)$ 的数列方案数
- $2n$ 人交 5 元，n 人只有 5 元一张，另 n 人只有 10 元一张，只要有 10 元人交钱就有 5 元找零的方案数

路径计数问题

只能向上或向右走的路径称为非降路径。

- 从 $(0, 0)$ 到 (m, n) 的非降路径数 $\binom{n+m}{m}$
- 从 $(0, 0)$ 到 (n, n) 的除端点外不接触直线 $y = x$ 的非降路径数 $2\binom{2n-2}{n-1} - 2\binom{2n-2}{n}$
- 从 $(0, 0)$ 到 (n, n) 的除端点外不穿过直线 $y = x$ 的非降路径数 $\frac{2}{n+1}\binom{2n}{n}$

斯特林数

第一类斯特林数（斯特林轮换数）

将 n 个两两不同的元素，划分为 k 个非空圆排列的数目（如 $[A,B,C]\neq[C,B,A]$）. 记为 $\begin{bmatrix}n\\k\end{bmatrix}$ 或 $s(n,k)$.

递推式为 $\begin{bmatrix}n\\k\end{bmatrix}=\begin{bmatrix}n-1\\k-1\end{bmatrix}+(n-1)\begin{bmatrix}n-1\\k\end{bmatrix},\begin{bmatrix}n\\0\end{bmatrix}=[n=0]$.

第二类斯特林数（斯特林子集数）将 n 个两两不同的元素，划分为 k 个互不区分的非空子集的方案数。记为 $\begin{Bmatrix}n\\k\end{Bmatrix}$ 或 $S(n,k)$.

递推式为 $\begin{Bmatrix}n\\k\end{Bmatrix}=\begin{Bmatrix}n-1\\k-1\end{Bmatrix}+k\begin{Bmatrix}n-1\\k\end{Bmatrix},\begin{Bmatrix}n\\0\end{Bmatrix}=[n=0]$.

通项公式为 $\begin{Bmatrix}n\\m\end{Bmatrix}=\sum_{i=0}^{m}\frac{(-1)^{m-i}i^n}{i!(m-i)!}$.

多项式与生成函数

狄利克雷生成函数

狄利克雷卷积

两个数论函数 $f(x),g(x)$ 的狄利克雷卷积为 $h(x)=\sum_{d|x}f(d)g\left(\frac{x}{d}\right)=\sum_{ab=x}f(a)g(b)$. 简记为 $h=f*g$.

性质:

交换律： $f*g=g*f$。

结合律：$(f*g)*h=f*(g*h)$。

分配律：$(f+g)*h=f*h+g*h$。

等式的性质： $f=g\Leftrightarrow f*h=g*h$，其中数论函数 $h(x)$ 满足 $h(1)\neq 0$.

狄利克雷生成函数

定义一个数论函数 f 的狄利克雷生成函数，简称 DEF 为

$$F(x)=\sum_{i\geq 1}\frac{f(i)}{i^x}$$

如果 f 是积性函数，则

$$F(x)=\prod_{p\in\mathcal{P}rime}\left(1+\frac{f(p)}{p^x}+\frac{f(p^2)}{p^{2x}}+\frac{f(p^3)}{p^{3x}}+\cdots\right)=\prod_{p\in\mathcal{P}rime}\sum_{k\geq 0}\frac{f(p^k)}{p^{kx}}$$

对于两个数论函数 f,g, 其 DGF 之积对应的是两者的狄利克雷卷积的 DGF:

$$F(x)G(x)=\sum_i\sum_j\frac{f(i)g(i)}{(ij)^x}=\sum_i\frac{1}{i^x}\sum_{d|i}f(d)g(\frac{i}{d})$$

计算几何

计算几何基础

点

```cpp
template <class T>
struct Point {
    T x, y;
    Point(const T& x = 0, const T& y = 0) : x(x), y(y) { }

    Point& operator+=(const Point& p) & {
        x += p.x;
        y += p.y;
        return *this;
    }
    Point& operator-=(const Point& p) & {
        x -= p.x;
        y -= p.y;
        return *this;
    }
    Point& operator*=(const T& v) & {
        x *= v;
        y *= v;
        return *this;
    }
    Point& operator/=(const T& v) & {
        x /= v;
        y /= v;
        return *this;
    }
    Point operator-() const {
        return Point(-x, -y);
    }
    friend Point operator+(Point a, const Point& b) {
        return a += b;
    }
    friend Point operator-(Point a, const Point& b) {
        return a -= b;
    }
    friend Point operator*(Point a, const T& b) {
        return a *= b;
    }
    friend Point operator/(Point a, const T& b) {
        return a /= b;
    }
    friend Point operator*(const T& a, Point b) {
        return b *= a;
    }
    friend bool operator<(const Point& a, const Point& b) {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
    }
    friend bool operator==(const Point& a, const Point& b) {
```

```
        return a.x == b.x && a.y == b.y;
    }
    friend istream& operator>>(istream& is, Point& p) {
        return is >> p.x >> p.y;
    }
    friend ostream& operator<<(ostream& os, const Point& p) {
        return os << p.x << ' ' << p.y;
        // return os << "(" << p.x << ", " << p.y << ")";
    }
};

// 两点内积
template <class T>
T dot(const Point<T>& a, const Point<T>& b) {
    return a.x * b.x + a.y * b.y;
}

// 两点外积
template <class T>
T cross(const Point<T>& a, const Point<T>& b) {
    return a.x * b.y - a.y * b.x;
}

// 点模平方
template <class T>
T square(const Point<T>& p) {
    return dot(p, p);
}

// 点模长度
template <class T>
double length(const Point<T>& p) {
    return sqrt(square(p));
}

// 点归一化
template <class T>
Point<T> normalize(const Point<T>& p) {
    return p / length(p);
}

// 两点距离
template <class T>
double distP2P(const Point<T>& a, const Point<T>& b) {
    return length(a - b);
}

// 判断点的方向（上半平面或x轴正方向为1，否则为-1）
template <class T>
int sgn(const Point<T>& a) {
    return a.y > 0 || (a.y == 0 && a.x > 0) ? 1 : -1;
}
```

- **旋转点**

向量绕二维直角坐标系原点**逆时针旋**转 θ, 对应的旋转矩阵用 $R(\theta)$ 表示 。向量发生旋转之前，动坐标系的标准正交基和向量 r 在静止坐标系中表示为

$$i=\begin{pmatrix}1\\0\end{pmatrix}, j=\begin{pmatrix}0\\1\end{pmatrix}, r=xi+yj$$

向量旋转后，动坐标系的标准正交基在静止坐标系中表示为

$$R(\theta)i=R(\theta)\begin{pmatrix}1\\0\end{pmatrix}=\begin{pmatrix}\cos\theta\\\sin\theta\end{pmatrix}, R(\theta)j=R(\theta)\begin{pmatrix}0\\1\end{pmatrix}=\begin{pmatrix}-\sin\theta\\\cos\theta\end{pmatrix}$$

向量 r 旋转后，在静止坐标系可以表示为

$$R(\theta)r=R(\theta)(xi+yj)=x\begin{pmatrix}\cos\theta\\\sin\theta\end{pmatrix}+y\begin{pmatrix}-\sin\theta\\\cos\theta\end{pmatrix}=\begin{pmatrix}x\cos\theta-y\sin\theta\\x\sin\theta+y\cos\theta\end{pmatrix}$$

```cpp
// 将点逆时针旋转
// 输入为弧度制
template <class T>
Point<T> rotate(const Point<T>& a, const double& rad) {
    double s = sin(rad), c = cos(rad);
    return Point(a.x * c - a.y * s, a.x * s + a.y * c);
}
```

线

```cpp
template <class T>
struct Line {
    Point<T> a;
    Point<T> b;
    Line(const Point<T>& a = Point<T>(), const Point<T>& b = Point<T>())
        : a(a)
        , b(b) {
    }
};

// 线段长度
template <class T>
double length(const Line<T>& l) {
    return length(l.a - l.b);
}
```

- **点到直线距离**

  点 P 到直线 AB 的距离 $d=\dfrac{|\overrightarrow{AB}\times\overrightarrow{AP}|}{|AB|}$.

```cpp
// 点到直线距离
template <class T>
double distP2L(const Point<T>& p, const Line<T>& l) {
    return abs(cross(l.a - l.b, l.a - p)) / length(l);
}
```

- **点到线段距离**

```cpp
// 点到线段距离
template <class T>
double distP2S(const Point<T>& p, const Line<T>& l) {
    if (dot(p - l.a, l.b - l.a) < 0) { // 点p在l的起点外侧
        return distP2P(p, l.a);
    }
    if (dot(p - l.b, l.a - l.b) < 0) { // 点p在l的终点外侧
        return distP2P(p, l.b);
    }
    return distP2L(p, l); // 点p在线段l的投影在线段上
}
```

- **点在直线上投影**

点 P 在直线 AB 上的投影 C 满足 $\overrightarrow{OC} = \overrightarrow{OA} + \dfrac{\overrightarrow{AB} \cdot \overrightarrow{AP}}{|AB|^2} \cdot \overrightarrow{AB}$.

```cpp
// 点在直线上投影
template <class T>
Point<T> project(const Point<T>& p, const Line<T>& l) {
    auto v = l.b - l.a; // 求直线方向向量
    return l.a + dot(p - l.a, v) / square(v) * v;
}
```

- **判断直线平行**

两条直线平行则外积为0.

```cpp
// 判断直线是否平行
template <class T>
bool parallel(const Line<T>& l1, const Line<T>& l2) {
    return cross(l1.b - l1.a, l2.b - l2.a) == 0;
}
```

- **判断点是否在直线左侧**

```cpp
// 判断点是否在直线左侧
template <class T>
bool pointOnLineLeft(const Point<T>& p, const Line<T>& l) {
    return cross(l.b - l.a, p - l.a) > 0;
}
```

- **判断点是否在线段上**

```cpp
// 判断点是否在线段上
template <class T>
bool pointOnSegment(const Point<T>& p, const Line<T>& l) {
    return cross(p - l.a, l.b - l.a) == 0
        && min(l.a.x, l.b.x) <= p.x && p.x <= max(l.a.x, l.b.x)
        && min(l.a.y, l.b.y) <= p.y && p.y <= max(l.a.y, l.b.y);
}
```

- **射线法判断点在任意多边形内部**

  先特判点在多边形边或顶点上，再以该点为端点引出一条射线，如果这条射线与多边形有奇数个交点，则该点在多边形内部，否则该点在多边形外部，简记为**奇内偶外**。

```cpp
// 判断点是否在多边形内
template <class T>
bool pointInPolygon(const Point<T>& a, const vector<Point<T>>& p) {
    int n = p.size();
    // 特判边界
    for (int i = 0; i < n; i++) {
        if (pointOnSegment(a, Line(p[i], p[(i + 1) % n]))) {
            return true;
        }
    }

    int t = 0;
    for (int i = 0; i < n; i++) {
        auto u = p[i];
        auto v = p[(i + 1) % n];
        if (u.x < a.x && v.x >= a.x && pointOnLineLeft(a, Line(v, u))) {
            t ^= 1; // 使用异或统计奇偶
        }
        if (u.x >= a.x && v.x < a.x && pointOnLineLeft(a, Line(u, v))) {
            t ^= 1;
        }
    }
    return t == 1;
}
```

- **求直线交点**

  设直线 AB, CD 交于点 P, O 为原点，则

$$\begin{aligned}\overrightarrow{OP} =& \overrightarrow{OA} + \overrightarrow{AP} \\ =& \overrightarrow{OA} + \frac{S_{\triangle ACD}}{S_{\triangle BCD}} \cdot \overrightarrow{AB} \\ =& \overrightarrow{OA} + \frac{\overrightarrow{CA} \times \overrightarrow{CD}}{\overrightarrow{CB} \times \overrightarrow{CD}} \cdot \overrightarrow{AB}\end{aligned}$$

```cpp
// 求直线交点
template <class T>
Point<T> lineIntersection(const Line<T>& l1, const Line<T>& l2) {
    return l1.a + (l1.b - l1.a) * (cross(l2.b - l2.a, l1.a - l2.a)
           / cross(l2.b - l2.a, l1.a - l1.b));
}
```

  求线段交点，需要先判断是否相交。

- **判断线段交点**

  快速排斥实验：规定“一条线段的区域”为以这条线段为对角线的，各边均与某一坐标轴平行的矩形所占的区域，若两条线段没有公共区域，则这两条线段一定不相交。未通过快速排斥实验是两线段无交点的 充分不必要条件。

跨立实验：若两线段 a, b 相交，a 线段的两个端点一定分布在 b 线段所在直线两侧，同理。

```cpp
// 判断线段交点情况
// 不相交 : {0, , }
// 严格相交 : {1, 交点, }
// 重叠 : {2, 重叠端点, 重叠端点}
// 在端点相交 : {3, 相交端点, }
template <class T>
tuple<int, Point<T>, Point<T>> segmentIntersection(const Line<T>& l1, const Line<T>& l2) {
    // 快速排斥实验
    if (max(l1.a.x, l1.b.x) < min(l2.a.x, l2.b.x)) {
        return { 0, Point<T>(), Point<T>() };
    }
    if (min(l1.a.x, l1.b.x) > max(l2.a.x, l2.b.x)) {
        return { 0, Point<T>(), Point<T>() };
    }
    if (max(l1.a.y, l1.b.y) < min(l2.a.y, l2.b.y)) {
        return { 0, Point<T>(), Point<T>() };
    }
    if (min(l1.a.y, l1.b.y) > max(l2.a.y, l2.b.y)) {
        return { 0, Point<T>(), Point<T>() };
    }

    // 跨立实验
    if (parallel(l1, l2)) { // 两线段平行
        if (cross(l1.b - l1.a, l2.a - l1.a) != 0) { // 共线检查
            return { 0, Point<T>(), Point<T>() };
        } else {
            // 计算重叠部分
            auto maxx1 = max(l1.a.x, l1.b.x);
            auto minx1 = min(l1.a.x, l1.b.x);
            auto maxy1 = max(l1.a.y, l1.b.y);
            auto miny1 = min(l1.a.y, l1.b.y);
            auto maxx2 = max(l2.a.x, l2.b.x);
            auto minx2 = min(l2.a.x, l2.b.x);
            auto maxy2 = max(l2.a.y, l2.b.y);
            auto miny2 = min(l2.a.y, l2.b.y);

            Point<T> p1(max(minx1, minx2), max(miny1, miny2));
            Point<T> p2(min(maxx1, maxx2), min(maxy1, maxy2));

            if (!pointOnSegment(p1, l1)) {
                swap(p1.y, p2.y);
            }
            if (p1 == p2) { // 交于一点
                return { 3, p1, p2 };
            } else { // 交于一条线段
                return { 2, p1, p2 };
            }
        }
    }

    // 跨立实验的参数
    auto cp1 = cross(l2.a - l1.a, l2.b - l1.a);
```

```
    auto cp2 = cross(l2.a - l1.b, l2.b - l1.b);
    auto cp3 = cross(l1.a - l2.a, l1.b - l2.a);
    auto cp4 = cross(l1.a - l2.b, l1.b - l2.b);

    // 判断是否相交
    if ((cp1 > 0 && cp2 > 0) || (cp1 < 0 && cp2 < 0)
        || (cp3 > 0 && cp4 > 0) || (cp3 < 0 && cp4 < 0)) {
        return { 0, Point<T>(), Point<T>() };
    }

    // 计算交点
    Point p = lineIntersection(l1, l2);
    if (cp1 != 0 && cp2 != 0 && cp3 != 0 && cp4 != 0) { // 严格相交
        return { 1, p, p };
    } else { // 在端点相交
        return { 3, p, p };
    }
}
```

- **线段到线段距离**

```
template <class T>
double distS2S(const Line<T>& l1, const Line<T>& l2) {
    if (get<0>(segmentIntersection(l1, l2)) != 0) { // 如果相交则距离为0
        return 0.0;
    }
    return min({ distP2S(l1.a, l2), distP2S(l1.b, l2),
                 distP2S(l2.a, l1), distP2S(l2.b, l1) });
}
```

- **判断线段是否在多边形内**

凸包

凸多边形是指所有内角大小都在 $[0, \pi]$ 范围内的简单多边形。

在平面上能包含所有给定点的最小凸多边形叫做凸包。

Andrew 算法

将所有点以横坐标为第一关键字，纵坐标为第二关键字排序。显然排序后最小和最大的元素一定在凸包上。在凸多边形上，从一个点出发逆时针走，轨迹总是**左拐**的，一旦出现右拐，就说明这一段不在凸包上。可用一个单调栈来维护上下凸壳。

求下凸壳时，若将进栈的点 P 和栈顶的两个点 A, B 满足 PB 在 PA 的左侧，即 $\overrightarrow{PA} \times \overrightarrow{PB} \geq 0$，则弹出栈顶 A 并重复检测，直到 $\overrightarrow{PA} \times \overrightarrow{PB} < 0$ 或栈内仅剩一个元素为止。求上凸壳同理。

复杂度为 $O(n \log n)$，n 为待求凸包点集的大小。

```
template <class T>
vector<Point<T>> getHull(vector<Point<T>> p) {
    sort(p.begin(), p.end(), [&](auto a, auto b) {
        return a.x < b.x || (a.x == b.x && a.y < b.y);
```

```
    }); // 若已重载 Point 小于号，直接排序即可
    p.erase(unique(p.begin(), p.end()), p.end());

    if (p.size() <= 1) {
        return p;
    }

    vector<Point<T>> hi, lo;
    for (auto a : p) {
        // 构建上凸壳（要求非左转）
        while (hi.size() > 1
               && cross(a - hi.back(), a - hi[hi.size() - 2]) <= 0) {
            hi.pop_back();
        }
        // 构建下凸壳（要求非右转）
        while (lo.size() > 1
               && cross(a - lo.back(), a - lo[lo.size() - 2]) >= 0) {
            lo.pop_back();
        }
        lo.push_back(a);
        hi.push_back(a);
    }

    // 合并上下凸壳
    lo.pop_back();
    reverse(hi.begin(), hi.end());
    hi.pop_back();
    lo.insert(lo.end(), hi.begin(), hi.end());

    return lo;
}
```

通常情况下不需要保留位于凸包边上的点，因此使用 $\geq$ 和 $\leq$, 可以视情况改为 $>$ 和 $<$.

Graham 扫描法

找到所有点中，纵坐标最小的一个点 P, 则 P 一定在凸包上。将所有的点以相对于点 P 的极角大小为关键字进行排序。

与 Andrew 算法类似，从 P 出发逆时针走，轨迹总是**左拐**的。即对于凸包逆时针方向上任意连续经过的三个点 P_1, P_2, P_3, 满足 $\overrightarrow{P_1P_2} \times \overrightarrow{P_2P_3} \geq 0$.

凸包的闵可夫斯基和

点集 P 和点集 Q 的闵可夫斯基和 $P+Q$ 定义为 $P+Q=\{a+b|a\in P, b\in Q\}$，即把点集 Q 中的每个点看做一个向量，将点集 P 中每个点沿这些向量平移。

若点集 P，Q 为凸集，则其闵可夫斯基和 $P+Q$ 也是凸集。

若点集 P，Q 为凸集，则其闵可夫斯基和 $P+Q$ 的边集是由凸集 P，Q 的边按极角排序后连接的结果。

旋转卡壳

通过枚举凸包上某一条边的同时维护其他需要的点，能够在线性时间内求解如凸包直径、最小矩形覆盖等和凸包性质相关的问题。

求凸包直径

逆时针地遍历凸包上的边，对于每条边都找到离边最远的点。随着边的转动，对应的最远点也在逆时针旋转。因此可在逆时针枚举凸包上的边时，维护一个当前最远点来更新答案。

判断点到边的距离大小时可用叉积分别算出两个三角形的面积来比较，速度更快。

```cpp
template <class T>
double hullDiameter(const vector<Point<T>>& p) {
    int n = p.size();
    if (n <= 1) {
        return 0;
    }
    if (n <= 2) {
        return distP2P(p[0], p[1]);
        // return square(p[0] - p[1]);
    }

    double maxd = 0;
    int j = 1;
    for (int i = 0; i < n; i++) {
        const auto &u = p[i], &v = p[(i + 1) % n];
        while (cross(v - u, p[(j + 1) % n] - u) >= cross(v - u, p[j] - u)) {
            j = (j + 1) % n;
        }
        maxd = max({ maxd, distP2P(u, p[j]), distP2P(v, p[j]) });
        // maxd = max({ maxd, square(u - p[j]), square(v - p[j]) });
    }

    return maxd;
}
```

求最小矩形覆盖

在凸包直径的基础上，同时维护3个点：所枚举的直线对面的点、两个在不同侧面的点。侧面的最优点用点积来比较，即比较在枚举直线上投影的长度，左右两个投影长度相加可以代表矩形的边长。

```cpp
constexpr double dInf = numeric_limits<double>::max();

// 返回最小矩形面积和4个顶点
template <class T>
pair<double, vector<Point<T>>> minRectangleCover(const vector<Point<T>>& p)
{
    int n = p.size();
    if (n <= 2) {
        return { 0, {} };
    }
    vector<Point<T>> res(4);

    double minArea = dInf;
    int l = 1, r = 1, j = 1;
```

```
    for (int i = 0; i < n; i++) {
        const auto &u = p[i], &v = p[(i + 1) % n];
        auto l1 = Line(u, v);
        while (cross(v - u, p[(j + 1) % n] - u) >= cross(v - u, p[j] - u)) {
            j = (j + 1) % n;
        }
        // 寻找侧面点时加上等号，否则遇到垂直边会终止寻找
        while (dot(v - u, p[(r + 1) % n] - u) >= dot(v - u, p[r] - u)) {
            r = (r + 1) % n;
        }
        if (i == 0) {
            l = r;
        }
        while (dot(v - u, p[(l + 1) % n] - u) <= dot(v - u, p[l] - u)) {
            l = (l + 1) % n;
        }

        const auto &a = p[j], &b = p[l], &c = p[r];
        double area = distP2L(a, l1)
                    * distP2P(project(b, l1), project(c, l1));
        if (area < minArea) {
            minArea = area;
            // 计算过点且与枚举直线平行的直线，即在点上加上直线的方向向量，构成另一个点
            auto l2 = Line(a, a + l1.a - l1.b);
            // 保证顶点逆时针排列
            res = {
                project(b, l1), project(c, l1),
                project(c, l2), project(b, l2)
            };
        }
    }

    return { minArea, res };
}
```

杂项

置换和排列

逆序数

一个排列里出现的逆序的总个数，叫做这个置换的逆序数。排列的逆序数是它恢复成正序序列所需要做相邻对换的最少次数，排列的逆序数的奇偶性和相应的置换的奇偶性一致。

- **使用归并排序**

```
// 区间为 [l, r)，注意为引用
ll mergeSort(vector<int>& nums, int l, int r) {
    if (r - l <= 1) {
        return 0;
    }

    ll res = 0;
    int mid = (l + r) / 2;
```

```
    res += mergeSort(nums, l, mid);
    res += mergeSort(nums, mid, r);

    vector<int> tmp(r - l);
    int i = l, j = mid, k = 0;
    while (i < mid && j < r) {
        if (nums[j] < nums[i]) {
            tmp[k] = nums[j++];
            res += mid - i;
        } else {
            tmp[k] = nums[i++];
        }
        k++;
    }

    while (i < mid) {
        tmp[k++] = nums[i++];
    }
    while (j < r) {
        tmp[k++] = nums[j++];
    }
    for (i = l, j = 0; i < r; i++, j++) {
        nums[i] = tmp[j];
    }
    return res;
}
```

- **使用树状数组**

```
ll count(const vector<int>& nums) {
    vector<int> sorted(nums);
    sort(sorted.begin(), sorted.end());
    sorted.erase(unique(sorted.begin(), sorted.end()), sorted.end());

    int n = sorted.size();
    map<int, int> idx;
    for (int i = 0; i < n; i++) {
        idx[sorted[i]] = n - i;
    }

    ll res = 0;
    Fenwick<int> f(n + 1);
    for (int x : nums) {
        int y = idx[x];
        res += f.sum(y);
        f.add(y, 1);
    }
    return res;
}
```

约瑟夫问题

n 个人标号 $0, 1, \cdots, n-1$, 逆时针站一圈。从 0 号开始，每一次从当前的人逆时针数 k 个，然后让这个人出局。问最后剩下的人是谁。

设 $J_{n,k}$ 表示规模分别为 n, k 的约瑟夫问题的答案。有如下递归式

$$J_{n,k} = (J_{n-1,k} + k) \bmod n$$

```cpp
int josephus(int n, int k) {
  int res = 0;
  for (int i = 1; i <= n; i++) {
      res = (res + k) % i;
  }
  return res;
}
```

数据结构

并查集

```cpp
struct DSU {
    std::vector<int> f, siz;

    DSU() { }
    DSU(int n) {
        init(n); // [0, n)
    }

    void init(int n) {
        f.resize(n);
        std::iota(f.begin(), f.end(), 0);
        siz.assign(n, 1);
    }

    int find(int x) {
        while (x != f[x]) {
            x = f[x] = f[f[x]];
        }
        return x;
    }

    bool same(int x, int y) {
        return find(x) == find(y);
    }

    bool merge(int x, int y) {
        x = find(x);
        y = find(y);
        if (x == y) {
            return false;
        }
        siz[x] += siz[y];
```

```
        f[y] = x;
        return true;
    }

    int size(int x) {
        return siz[find(x)];
    }
};
```

ST表

ST 表（稀疏表）是用于解决**可重复贡献问题**的数据结构。

> **可重复贡献问题** 是指对于运算 opt, 满足 $x \operatorname{opt} x = x$, 则对应的区间询问就是一个可重复贡献问题。
>
> 常见的可重复贡献问题有：区间最值、区间按位和、区间按位或、区间GCD等。

ST 表基于倍增思想，可做到 $O(n \log n)$ 预处理，$O(1)$ 询问，不支持修改操作。

以区间最大值为例，令 $f(i,j)$ 表示区间 $[i, i+2^j-1]$ 的最大值，显然 $f(i,0) = a_i$. 状态转移方程为 $f(i,j) = \max(f(i,j-1), f(i+2^{j-1}, j-1))$. 对于每个询问 $[l,r]$，分为 $[l, l+2^s-1]$ 与 $[r-2^s+1, r]$ 两部分，其中 $s = \lfloor \log_2(r-l+1) \rfloor$, 求出两部分的最大值即可。

```
template <typename Info>
struct SparseTable {
    vector<vector<Info>> dp;
    vector<int> log2;
    int n;

    SparseTable() : n(0) {};
    SparseTable(const vector<Info>& arr) {
        n = arr.size();
        init();
        build(arr);
    }

    void init() {
        log2.assign(n + 1, {});
        log2[1] = 0;
        for (int i = 2; i <= n; i++) {
            log2[i] = log2[i >> 1] + 1;
        }
    }

    void build(const vector<Info>& arr) {
        int m = log2[n] + 1;
        dp.assign(n, vector<Info>(m, Info()));

        for (int i = 0; i < n; i++) {
            dp[i][0] = arr[i];
        }

        for (int j = 1; j < m; j++) {
```

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

    // [l, r]
    Info query(int l, int r) {
        int j = log2[r - l + 1];
        return dp[l][j] + dp[r - (1 << j) + 1][j];
    }
};

struct Info {
    int x;
    Info operator+(const Info& o) const {
        return { max(x, o.x) };
    }
    friend istream& operator>>(istream& is, Info& o) {
        return is >> o.x;
    }
    friend ostream& operator<<(ostream& os, const Info& o) {
        return os << o.x;
    }
};
```

树状数组

树状数组支持在 $O(\log n)$ 的复杂度下单点修改和区间查询。

```
template <typename T>
struct Fenwick {
    int n;
    vector<T> a;

    Fenwick(int n_ = 0) {
        init(n_);
    }
    // [0, n)
    void init(int n_) {
        n = n_;
        a.assign(n, T {});
    }

    void add(int x, const T& v) {
        for (int i = x + 1; i <= n; i += i & -i) {
            a[i - 1] = a[i - 1] + v;
        }
    }
    // [0, x)
    T sum(int x) {
        T ans {};
        for (int i = x; i > 0; i -= i & -i) {
```

```
            ans = ans + a[i - 1];
        }
        return ans;
    }
    // [l, r)
    T rangeSum(int l, int r) {
        return sum(r) - sum(l);
    }

    int select(const T& k) {
        int x = 0;
        T cur {};
        for (int i = 1 << __lg(n); i; i /= 2) {
            if (x + i <= n && cur + a[x + i - 1] <= k) {
                x += i;
                cur = cur + a[x - 1];
            }
        }
        return x;
    }
};
```

- **区间加区间和**

  考虑序列 a 的差分数组 d, 其中 $d[i] = a[i] - a[i-1]$. 由于差分数组的前缀和就是原数组，所以 $a_i = \sum_{j=1}^{i} d_j$. 将查询区间和通过差分转化为查询前缀和。那么考虑查询 $a[1 \dots r]$ 的和，即

$$\begin{aligned}\sum_{i=1}^{r} a_i &= \sum_{i=1}^{r} \sum_{j=1}^{i} d_j \\ &= \sum_{i=1}^{r} d_i \times (r - i + 1) \\ &= \sum_{i=1}^{r} d_i \times (r+1) - \sum_{i=1}^{r} d_i \times i\end{aligned}$$

  $\sum_{i=1}^{r} d_i$ 并不能推出 $\sum_{i=1}^{r} d_i \times i$, 所以要用两个树状数组分别维护 d_i 和 $d_i \times i$ 的和信息。

```
template <typename T>
struct RangeFenwick {
    Fenwick<T> f1, f2;

    RangeFenwick(int n = 0)
        : f1(n)
        , f2(n) {
    }
    void init(int n) {
        f1.init(n);
        f2.init(n);
    }
    // [l, r)
    void rangeAdd(int l, int r, const T& v) {
        f1.add(l, v);
        f1.add(r, -v);
        f2.add(l, v * l);
```

```
        f2.add(r, -v * r);
    }
    // [0, x)
    T prefixSum(int x) {
        return f1.sum(x) * x - f2.sum(x);
    }
    // [l, r)
    T rangeSum(int l, int r) {
        return prefixSum(r) - prefixSum(l);
    }
};
```

线段树

```
template <class Info, class Tag>
struct LazySegmentTree {
    int n;
    vector<Info> info;
    vector<Tag> tag;

    LazySegmentTree()
        : n(0) {
    }
    LazySegmentTree(int n, Info v = Info()) {
        init(n, v);
    }
    // [0, n)
    template <class T>
    LazySegmentTree(vector<T> arr) {
        init(arr);
    }

    void init(int n, Info v = Info()) {
        init(vector(n, v));
    }
    template <class T>
    void init(vector<T> arr) {
        n = arr.size();
        info.assign(4 << __lg(n), Info());
        tag.assign(4 << __lg(n), Tag());
        function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                info[p] = arr[l];
                return;
            }
            int m = (l + r) / 2;
            build(2 * p, l, m);
            build(2 * p + 1, m, r);
            pull(p);
        };
        build(1, 0, n);
    }

```

```
    void pull(int p) {
        info[p] = info[2 * p] + info[2 * p + 1];
    }
    void apply(int p, const Tag& v) {
        info[p].apply(v);
        tag[p].apply(v);
    }
    void push(int p) {
        apply(2 * p, tag[p]);
        apply(2 * p + 1, tag[p]);
        tag[p] = Tag();
    }

    void modify(int p, int l, int r, int x, const Info& v) {
        if (r - l == 1) {
            info[p] = v;
            return;
        }
        int m = (l + r) / 2;
        push(p);
        if (x < m) {
            modify(2 * p, l, m, x, v);
        } else {
            modify(2 * p + 1, m, r, x, v);
        }
        pull(p);
    }
    void modify(int p, const Info& v) {
        modify(1, 0, n, p, v);
    }

    Info rangeQuery(int p, int l, int r, int x, int y) {
        if (l >= y || r <= x) {
            return Info();
        }
        if (l >= x && r <= y) {
            return info[p];
        }
        int m = (l + r) / 2;
        push(p);
        return rangeQuery(2 * p, l, m, x, y) + rangeQuery(2 * p + 1, m, r,
x, y);
    }
    // [l, r)
    Info rangeQuery(int l, int r) {
        return rangeQuery(1, 0, n, l, r);
    }

    void rangeApply(int p, int l, int r, int x, int y, const Tag& v) {
        if (l >= y || r <= x) {
            return;
        }
        if (l >= x && r <= y) {
            apply(p, v);
            return;
        }
```

```
        int m = (l + r) / 2;
        push(p);
        rangeApply(2 * p, l, m, x, y, v);
        rangeApply(2 * p + 1, m, r, x, y, v);
        pull(p);
    }
    // [l, r)
    void rangeApply(int l, int r, const Tag& v) {
        return rangeApply(1, 0, n, l, r, v);
    }
};

struct Tag {
    // 储存标记
    void apply(const Tag& t) & {
        // 使用父节点的标记更新子节点的标记
    }
};

struct Info {
    // 储存信息
    void apply(const Tag& t) & {
        // 使用父节点的标记更新子节点的信息
    }
};

Info operator+(const Info& a, const Info& b) {
    // 重载加号规则
}
```

使用注意事项

```
vector<Info> a(n);
// 下标为 [0, n)
for (int i = 0; i < n; i++) {
    cin >> a[i].x; // 直接输入值 x
}
LazySegmentTree<Info, Tag> seg(a);
while (m--) {
    int op, x, y, k;
    cin >> op >> x >> y;
    if (op == 1) {
        cin >> k;
        // 未定义构造函数，不要使用 Tag(k)
        seg.rangeApply(x - 1, y, { k });
    } else {
        // 直接输出值 x
        cout << seg.rangeQuery(x - 1, y).x << '\n';
    }
}
```

字符串

字典树

```
struct Trie {
    static constexpr int ABC = 26, N = 2e5 + 10;
    int son[N][ABC];
    int cnt[N];
    int idx = 0;

    int getIdx(char c) {
        return c - 'a';
    }

    void insert(const string& str) {
        int p = 0;
        for (char ch : str) {
            int u = getIdx(ch);
            if (son[p][u] == 0) {
                son[p][u] = ++idx;
            }
            p = son[p][u];
        }
        cnt[p]++;
    }

    int query(const string& str) {
        int p = 0;
        for (char ch : str) {
            int u = getIdx(ch);
            if (son[p][u] == 0) {
                return 0;
            }
            p = son[p][u];
        }
        return cnt[p];
    }

    void clear() {
        for (int i = 0; i <= idx; i++) {
            fill(son[i], son[i] + ABC, 0);
            cnt[i] = 0;
        }
        idx = 0;
    }
} trie;
```

图论

树

树的直径

- **两次 DFS**

  从任意节点 x 开始进行第一次 DFS，到达距离其最远的节点，记为 z. 再从 z 开始做第二次 DFS，到达距离 z 最远的节点，记为 z', 则 $\delta(z,z')$ 即为树的直径。

- **树形DP**

  记录当 1 为树的根时，每个节点作为子树的根向下所能延伸的最长路径长度 d_1 与次长路径（与最长路径无公共边） d_2，则直径就是所有点的 d_1+d_2 中的最大值。

  树形 DP 可以在存在负权边的情况下求解出树的直径。

```cpp
int dfs(int u, int fa) {
    int d1 = -inf, d2 = -inf;
    for (auto [v, w] : adj[u]) {
        if (v == fa) {
            continue;
        }
        int t = dfs(v, u) + w;
        if (t > d1) {
            d2 = d1, d1 = t;
        } else if (t > d2) {
            d2 = t;
        }
    }
    d = max(d, d1 + d2);
    return max(d1, 0);
}
```

树的中心

如果节点 x 作为根节点时，从 x 出发的最长链最短，那么称 x 为这棵树的中心。

性质

- 树的中心不一唯一，但最多有 2 个，且这两个中心是相邻的，位于树的直径上。
- 当树的中心为根节点时，到达直径端点的两条链分别为最长链和次长链。
- 在两棵树间连一条边以合并为一棵树时，连接两棵树的中心可以使新树的直径最小。

维护 $d1_x$ 和 $d2_x$ 表示节点 x 子树内的最长链和次长链。维护 up_x 表示节点 x 子树外的最长链，该链必定经过 x 的父节点。找到点 x 使得 $\max(d1_x, up_x)$ 最小，那么 x 即为树的中心。

```cpp
void dfsd(int u, int fa) {
    for (auto [v, w] : adj[u]) {
        if (v == fa) {
            continue;
        }
        dfsd(v, u);
        if (d1[v] + w > d1[u]) {
```

```
            d2[u] = d1[u], d1[u] = d1[v] + w;
        } else if (d1[v] + w > d2[u]) {
            d2[u] = d1[v] + w;
        }
    }
}

void dfsu(int u, int fa) {
    for (auto [v, w] : adj[u]) {
        if (v == fa) {
            continue;
        }
        if (d1[v] + w != d1[u]) {
            // u 子树的最长链不在 v 子树里
            up[v] = max(up[u], d1[u]) + w;
        } else {
            up[v] = max(up[u], d2[u]) + w;
        }
        dfsu(v, u);
    }
}

void treeCenter() {
    dfsd(1, 0);
    dfsu(1, 0);
    int minl = inf;
    for (int i = 1; i <= n; i++) {
        minl = min(minl, max(up[i], d1[i]));
    }
    for (int i = 1; i <= n; i++) {
        if (minl == max(up[i], d1[i])) {
            cout << i << ' ';
        }
    }
}
```

树的重心

删除节点 x 将树分成多棵子树，统计所有子树节点数并记录最大值，使得该最大值最小的 x 为这棵树的重心。

性质

- 树的重心不一唯一，但最多有 2 个，且这两个重心是相邻的。
- 以树的重心为根时，所有子树的大小都不超过整棵树大小的一半。
- 树中所有点到某个点的距离和中，到重心的距离和是最小的。
- 把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上。

用DFS 中计算每个子树的大小，记录向下的子树的最大大小，利用总点数 - 当前子树的大小得到向上的子树的大小，根据依据定义找重心即可。

```
// siz 为子树的大小，w 为子树的最大大小，core 为重心
// 对于有权树，a 为节点的权值，sum 为 a 的和
```

```cpp
// 对于无权树，只需将 a[u] 改为 1, sum 改为 n
void dfs(int u, int fa) {
    siz[u] = a[u];
    for (int v : adj[u]) {
        if (v == fa) {
            continue;
        }
        dfs(v, u);
        siz[u] += siz[v];
        w[u] = max(w[u], siz[v]);
    }
    w[u] = max(w[u], sum - siz[u]);
    if (w[u] <= sum / 2) {
        core.push_back(u);
    }
}
```

最近公共祖先

```cpp
struct LCA {
    int n, logN;
    vector<vector<int>> adj, up;
    vector<int> depth;

    LCA(int n)
        : n(n) {
        logN = log2(n) + 1;
        adj.resize(n);
        up.assign(n, vector<int>(logN, -1));
        depth.resize(n);
    }

    void add(int u, int v) {
        adj[u].push_back(v), adj[v].push_back(u);
    }

    void dfs(int u, int parent) {
        up[u][0] = parent;
        for (int i = 1; i < logN; ++i) {
            if (up[u][i - 1] != -1) {
                up[u][i] = up[up[u][i - 1]][i - 1];
            }
        }
        for (int v : adj[u]) {
            if (v != parent) {
                depth[v] = depth[u] + 1;
                dfs(v, u);
            }
        }
    }

    void work(int root) {
        depth[root] = 0;
```

```
        dfs(root, -1);
    }

    int getLCA(int u, int v) {
        if (depth[u] < depth[v]) {
            swap(u, v);
        }
        // 将 u 提升到与 v 同一深度
        for (int i = logN - 1; i >= 0; --i) {
            if (depth[u] - (1 << i) >= depth[v]) {
                u = up[u][i];
            }
        }
        if (u == v) {
            return u;
        }
        // 同时提升 u 和 v
        for (int i = logN - 1; i >= 0; --i) {
            if (up[u][i] != up[v][i]) {
                u = up[u][i];
                v = up[v][i];
            }
        }
        return up[u][0];
    }
};
```

拓扑排序

拓扑排序用于解决给一个有向无环图的所有节点排序。

构造拓扑排序的步骤如下：从图中选择一个入度为零的点，输出该顶点并从图中删除此顶点及其所有的出边。重复上面两步，直到所有顶点都输出，则拓扑排序完成。否则说明图是有环图，无法完成拓扑排序。

```
bool topsort() {
    vector<int> list;
    queue<int> q;

    for (int i = 1; i <= n; i++) {
        if (in[i] == 0) {
            q.push(i);
        }
    }

    while (q.size()) {
        int u = q.front();
        q.pop();
        list.push_back(u);

        for (int v : adj[u]) {
            if (--in[v] == 0) {
                q.push(v);
```

```
            }
        }
    }
    if (list.size() == n) {
        return true;
    }
    return false;
}
```

> 有时候要求得到唯一确定排序时（Luogu P1347 排序），可记录拓扑排序层数的最大值，当最大值为 n 时才存在排序。

最短路

Dijkstra 算法

用于求解非负权图上单源最短路径，复杂度 $O(m \log m)$.

Floyd 算法

用于求解非负权图上全源最短路径，复杂度 $O(n^3)$.

Bellman–Ford 算法

Bellman–Ford 算法不断尝试对图上每一条边进行松弛。每进行一轮循环，就对图上所有的边都尝试进行一次松弛操作，当一次循环中没有成功的松弛操作时，算法停止。若最短路存在，由于一次松弛操作会使最短路的边数至少 $+1$, 而最短路的边数最多为 $n-1$, 因此最多执行 $n-1$ 轮松弛操作。如果第 n 轮循环时仍然存在能松弛的边，说明从起点出发，能够抵达一个负环。复杂度为 $O(nm)$.

```
bool bellmanFord(int n, int s) {
    dist[s] = 0;
    bool relaxed; // 是否发生松弛操作
    for (int i = 1; i <= n; i++) {
        relaxed = false;
        for (auto [u, v, w] : edge) {
            if (dist[u] == inf) {
                continue;
            }
            if (dist[v] > dist[u] + w) {
                dist[v] = dist[u] + w;
                relaxed = true;
            }
        }
        if (!relaxed) {
            break;
        }
    }
    return relaxed;
}
```

> 如果要判断整个图上是否存在负环，应建立一个超级源点，向图上每个节点连一条权值为 0 的边，然后以超级源点为起点执行 Bellman–Ford 算法，注意松弛操作要进行 $n+1$ 次。

杂项

输入输出

int128 库函数自定义

```cpp
using i128 = __int128;

i128 toi128(const string& s) {
    i128 n = 0;
    int i = 0, neg = 0;
    if (s[i] == '-') {
        neg = 1;
        i = 1;
    }
    for (; i < s.size(); i++) {
        n = n * 10 + (s[i] - '0');
    }
    return neg ? -n : n;
}

ostream& operator<<(ostream& os, i128 n) {
    if (n < 0) {
        os << '-';
        n = -n;
    }
    if (n > 9) {
        os << (n / 10);
    }
    os << (char)(n % 10 + '0');
    return os;
}

istream& operator>>(istream& is, i128& n) {
    string s;
    is >> s;
    n = toi128(s);
    return is;
}

i128 sqrt(i128 n) {
    i128 lo = 0, hi = 1e16;
    while (lo < hi) {
        i128 x = (lo + hi + 1) / 2;
        if (x * x <= n) {
            lo = x;
        } else {
            hi = x - 1;
        }
    }
    return lo;
}

i128 gcd(i128 a, i128 b) {
    while (b) {
```

```
        a %= b;
        swap(a, b);
    }
    return a;
}
```

浮点数四舍五入

`std::fixed` 指示输出流在格式化浮点数时采用固定的定点表示法，即总是包含小数点以及后面的小数部分。`std::setprecision` 用于设置输出流中浮点数的精度，即小数部分显示的位数。

```
double round(double x, int n) {
    double k = pow(10.0, n);
    return round(x * k + (x < 0 ? -0.5 : 0.5)) / k;
}
```

```
// 保留小数点后10位
double x = numbers::pi;
cout << fixed << setprecision(10) << round(x, 10) << '\n';
```

> c++20 可用 `std::format` 自带四舍五入

format

```
// 保留小数点后10位，自带四舍五入
cout << format("{:.10f}", x) << '\n';
```

STL

vector

`vector` 里面存放两个变量 `size` （数组实际**长度**大小）和 `capacity` （数组分配的内存空间**容量**大小）。

当 `size` 大于 `capacity` 时，vector会自动进行扩容。扩容规则为：重新开辟新的内存空间（大小为原来的 `capacity` 的2倍），原来的 `vector` 中存储内容先复制到新的地址空间中，然后销毁原来的地址空间。

方法	复杂度	含义
`size()`	$O(1)$	实际数据个数（unsigned类型）
`begin()`	$O(1)$	返回首元素的迭代器
`end()`	$O(1)$	返回最后一个元素**后一个位置**的迭代器
`front()`	$O(1)$	返回容器中的第一个数据
`back()`	$O(1)$	返回容器中的最后一个数据

方法	复杂度	含义
`at()`		返回下标为`idx`的数据，会进行边界检查
`push_back(ele)`	$O(1)$	在尾部加一个数据
`pop_back()`	$O(1)$	删除最后一个数据
`emplace_back(ele)`	$O(1)$	在尾部加一个数据
`insert(pos, x)`	$O(N)$	向任意迭代器`pos`插入元素`x`
`erase(first, last)`	$O(N)$	删除`[first, last)`的所有元素
`clear()`	$O(N)$	清除容器中的所有元素，会将`size`设为`0`
`resize(n, val)`		修改`size`为`n`并将数值赋为`val`，默认赋值为`0`
`assign(n, val)`		清除容器并将`n`个`val`值拷贝到`c`数组中，会将`size`设为`n`
`c.reserve(n)`		为数组提前分配`n`的内存大小，即将`capacity`设为`n`

map

方法	复杂度	含义
`size()`	$O(1)$	返回映射的对数
`begin()`	$O(1)$	返回指向首元素的迭代器
`end()`	$O(1)$	返回指向最后一个元素**后一个位置**迭代器
`rbegin()`	$O(1)$	返回指向最后一个元素的迭代器
`rend()`	$O(1)$	返回指向第一个元素**前一个位置**的逆向迭代器
`insert()`	$O(\log N)$	插入元素，构造键值对
`count(key)`	$O(\log N)$	查看元素是否存在，存在返回`1`，不存在返回`0`
`find(key)`	$O(\log N)$	返回键为`key`的映射的迭代器 当数据存在时，返回数据所在位置的迭代器，数据不存在时，返回`end()`
`erase(it)`	$O(\log N)$	删除迭代器对应的键值对
`erase(key)`	$O(\log N)$	根据映射的键删除对应的键值对
`erase(first, last)`	$O(last - first)$	删除左闭右开区间迭代器对应的键值对
`clear()`	$O(N)$	清空所有元素

方法	复杂度	含义
`lower_bound()`		返回键值大于等于 `k` 的第一个键值对的迭代器
`upper_bound()`		返回键值大于 `k` 的第一个键值对的迭代器

set

方法	复杂度	含义
`size()`	$O(1)$	返回容器中的元素个数
`begin()`	$O(1)$	返回首元素的迭代器
`end()`	$O(1)$	返回最后一个元素**后一个位置**迭代器
`rbegin()`	$O(1)$	返回指向最后一个元素的逆向迭代器
`rend()`	$O(1)$	返回指向第一个元素**前一个位置**的逆向迭代器
`insert()`	$O(\log N)$	插入元素
`count(key)`	$O(\log N)$	查看元素是否存在，存在返回 `1`，不存在返回 `0`
`find(key)`	$O(\log N)$	返回键为 `key` 的映射的迭代器 当数据存在时，返回数据所在位置的迭代器，数据不存在时，返回 `end()`
`erase(it)`	$O(\log N)$	删除迭代器对应的值
`erase(key)`	$O(\log N)$	删除对应的值
`erase(first, last)`	$O(last - first)$	删除左闭右开区间迭代器对应的值
`clear()`	$O(N)$	清空所有元素
`lower_bound(x)`		返回大于等于 `k` 的第一个元素的迭代器
`upper_bound(x)`		返回大于 `k` 的第一个元素的迭代器

重写set排序规则

- 普通函数指针

```
bool cmp(const int& x, const int& y) {
    return x > y;
}
set<int, bool(*)(const int& x, const int& y)> s(cmp);
set<int, decltype(&cmp)> s(cmp);
```

也可用 lambda 书写

```cpp
set<int, function<bool(int, int)>> s([&](int x, int y) {
    return x > y;
});
```

- 函数对象

```cpp
struct Cmp {
    bool operator()(const int& x, const int& y) const {
        return x < y;
    }
};
set<int, Cmp> s;
```

- 库函数

```cpp
set<int, greater<int>> s;
```

multiset

元素可以重复，且元素有序。

进行删除操作时，要明确删除目标。删除多个元素，使用 `erase(val)` 方法会删除掉所有与 `val` 相等的元素。需要删除一个元素时，需要使用 `erase(find(val))`.

STL函数

- **max_element / min_element**

  返回 $[beg, end)$ 中最大 / 最小元素对应的迭代器。

- **nth_element(beg, mid, end)**

  将 $[beg, end)$ 中的内容重新分配顺序，小于（等于）mid 的元素在 $[beg, mid)$，大于（等于）mid 的元素都在 (mid, end). 复杂度为 $O(n)$.

- **next_permutation(beg, ed)**

  将 $[beg, end)$ 更改为下一个排列，复杂度 $O(n)$, 注意要先排序。

- **count(beg, end, val)**

  返回 $[beg, end)$ 等于 val 的元素的个数。

- **shuffle(beg, end, gen)**

  打乱 $[beg, end)$ 的顺序，`gen` 是一个随机数生成器。

- **iota(beg, end, val)**

  将 $[beg, end)$ 中的元素依次赋值为 $val, val + 1, val + 2, \cdots$.

- **accumulate(beg, end, val)**

  将 $[beg, end)$ 中所有元素与初始值 val 相加，返回这个和。注意返回值与 val 类型一致。

- **lower_bound(beg, end, val, cmp)**

  查找 $[beg, end)$ 第一个大于等于 val 的元素，返回地址。

  自定义比较函数 `bool cmp(element, val)`，会返回 $[beg, end)$ 中第一个使 `cmp` 为 **false** 的数。

```cpp
vector<pair<int, int>> vec { { 1, 10 }, { 2, 20 }, { 3, 30 } };
auto it = lower_bound(vec.begin(), vec.end(), 2, [](pair<int, int>& p,
int val) {
    return val > p.first;
});
cout << it->second << '\n'; // 等价于 (*it).second, 输出 20
```

- **upper_bound(beg, end, val, cmp)**

  查找 $[beg, end)$ 第一个大于 val 的元素，返回地址。

  自定义比较函数 `bool cmp(value, element)`，会返回 $[beg, end)$ 中第一个使 `cmp` 为 **true** 的数。

```cpp
vector<pair<int, int>> vec { { 1, 10 }, { 2, 20 }, { 3, 30 } };
auto it = upper_bound(vec.begin(), vec.end(), 2, [](int val, pair<int,
int>& p) {
    return val < p.first;
});
cout << it->second << '\n'; // 等价于 (*it).second, 输出 30
```

- **is_sorted(beg, end)**

  判断序列是否为升序。