MTF 莫号模板库

1 杂项

1.1 约定

1. C++ 版本：
   1. 最低支持版本：c++17；
   2. 默认版本：gnu++17；
2. 格式：
   1. 缩进宽度为 4 个空格；
   2. 大括号换行；
   3. 一元运算符不空格，其他运算符空一格；
3. 命名：
   1. 遵循 GoLang 的命名规范；
4. 其他：
   1. 使用邻接表存图；
   2. 使用 0-indexed 的左闭右开区间；
   3. 使用解绑同步流的 std::cin 和 std::cout 输入输出；
   4. 使用 using namespace std;；
   5. 使用局部变量而非全局变量，局部 lambda 而非全局函数；
   6. 使用 emplace 而非 push，集合查重使用 count；
   7. 若键值较小，使用 vector 而非 map；不要求顺序的情况下可以用 unordered_ 系列，但在 Codeforces 上记得使用随机模数；
   8. 每个模板自包含：自带所需类型别名与 include，独立作用域、零前置依赖，抄哪段就只用哪段（验证器对每个模板做独立编译 门禁）；任意组合可拼入同一编译单元（合并编译门禁保证）；

1.2 初始代码

```cpp
#include <bits/stdc++.h>
using namespace std;

int T{0};
void solve() {}

int main() {
    cin.tie(nullptr)->sync_with_stdio(false);
    if (!T)
        cin >> T;
    while (T--)
        solve();
}
```

1.3 类型定义

```cpp
using i8 = signed char;
using u8 = unsigned char;
using i32 = signed;
using u32 = unsigned;
using i64 = int64_t;
using u64 = uint64_t;
using i128 = __int128;
using u128 = unsigned __int128;
```

1.4 随机数生成

```cpp
using u64 = uint64_t;

// std::mt19937
mt19937 rng(std::chrono::steady_clock::now()
                .time_since_epoch()
                .count());
// splitmix64
struct custom_hash {
    static auto splitmix64(u64 x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    auto operator()(u64 x) const {
        static const u64 SEED =
            std::chrono::high_resolution_clock::
                now()
                    .time_since_epoch()
                    .count();
        return splitmix64(x + SEED);
    }
};
```

1.5 二分答案

```cpp
int l, r, ans;
bool check(int);
while (l <= r) {
    int mid = l + (r - l) / 2;
    if (check(mid)) {
        ans = mid;
        // l 和 r 取决于方向
        l = mid + 1;
    } else {
        r = mid - 1;
    }
}
```

1.6 Lambda 递归

```cpp
std::vector<std::vector<int>> adj;
auto dfs1 = [&](auto&& self, int x,
                int p) -> void {
    for (auto y : adj[x]) {
        if (y == p)
            continue;
        self(self, y, x);
    }
```

```
};
dfs1(dfs1, 0, 0);
// C++17
std::function<void(int, int)> dfs2 =
    [&](int x, int p) {
        for (auto y : adj[x]) {
            if (y == p)
                continue;
            dfs2(y, x);
        }
    };
```

1.7 高维前缀和 (SOS DP)

就地计算所有集合的子集和 / 超集和，逐位转移。

- 复杂度 O(n 2^n)（逐集合枚举子集是 3^n）；
- f.size() 必须是 2 的幂；变换按位独立，位枚举顺序任意；
- inverse = true 做逆变换（把 += 换成 -=），与正变换互逆；
- AND 卷积：两数组分别做超集和，逐点相乘，再做一次逆超集和；OR 卷积同理换成子集和；
- 也可对某一维只做部分位，处理“固定若干位、其余任意”的统计。

```
template <class T>
void SubsetSum(std::vector<T>& f, bool inverse = false) {
    int n = __builtin_ctzll(f.size());
    for (int i = 0; i < n; i++)
        for (int s = 0; s < (int)f.size(); s++)
            if (s >> i & 1) {
                if (inverse)
                    f[s] -= f[s ^ 1 << i];
                else
                    f[s] += f[s ^ 1 << i];
            }
}

template <class T>
void SupersetSum(std::vector<T>& f, bool inverse = false) {
    int n = __builtin_ctzll(f.size());
    for (int i = 0; i < n; i++)
        for (int s = 0; s < (int)f.size(); s++)
            if (not(s >> i & 1)) {
                if (inverse)
                    f[s] -= f[s | 1 << i];
                else
                    f[s] += f[s | 1 << i];
            }
}
```

2 数学

2.1 线性筛

初始化 [2, N] 内质数和非质数标记。

- pri 存储所有质数；
- np[x] 表示 x 是否不是质数；
- 固定上界作为模板参数，使用 Sieve<N> sieve; 初始化；
- 复杂度 O(N)。

```
template <int N>
struct Sieve {
    std::vector<int> pri;
    bool np[N + 1]{};

    Sieve() {
        np[0] = np[1] = true;
        for (int i = 2; i <= N; i++) {
            if (!np[i])
                pri.emplace_back(i);
            for (int p : pri) {
                if (i * p > N)
                    break;
                np[i * p] = true;
                if (i % p == 0)
                    break;
            }
        }
    }
};
```

2.2 因数/约数分解

Factor(n) 返回质因数分解，Divisor(n) 返回大于 1 且小于等于 n 的约数。

- 适合 int 范围内的试除；
- 复杂度 O(sqrt n)。

```
inline auto Factor(int n) {
    std::vector<std::pair<int, int>> res;
    for (int i = 2; i * i <= n; i++) {
        int cur = 0;
        while (n % i == 0) {
            cur++;
            n /= i;
        }
        if (cur != 0)
            res.emplace_back(i, cur);
    }
    if (n > 1)
        res.emplace_back(n, 1);
    return res;
}

inline auto Divisor(int n) {
    std::vector<int> res;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            res.emplace_back(i);
            if (i * i != n)
                res.emplace_back(n / i);
        }
    }
    std::sort(res.begin(), res.end());
    return res;
}
```

2.3 扩展欧几里得

返回 gcd(a, b)，并求出 ax + by = gcd(a, b) 的一组解。

- 参数可以是整数类型；
- 复杂度 O(log min(a, b))。

```
template <class T>
inline T ExGCD(T a, T b, T& x, T& y) {
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    auto d = ExGCD(b, a % b, y, x);
    y -= a / b * x;
    return d;
}
```

2.4 快速幂

计算 a^b mod p。

- b >= 0；
- 复杂度 O(log b)。

```
using i64 = int64_t;

inline auto PowMod(i64 a, i64 b, int p) {
    i64 res = 1;
    while (b) {
        if (b & 1)
            res = res * a % p;
        b = b >> 1;
        a = a * a % p;
    }
    return res;
}
```

2.5 组合数

基于 ModInt 预处理阶乘和逆阶乘。

- Z 需要支持乘法、除法；
- C(n, k) 返回组合数；
- 预处理 O(n)，单次查询 O(1)。

```
template <class Z>
struct Comb {
    std::vector<Z> fac, ifac;

    Comb(int n) : fac(n + 1), ifac(n + 1) {
        fac[0] = 1;
        for (int i = 1; i <= n; i++)
            fac[i] = fac[i - 1] * i;
        ifac[n] = Z(1) / fac[n];
        for (int i = n; i >= 1; i--)
```

```
            ifac[i - 1] = ifac[i] * i;
    }

    auto C(int n, int k) const {
        if (k < 0 or k > n)
            return Z{};
        return fac[n] * ifac[k] * ifac[n - k];
    }

    auto A(int n, int k) const {
        if (k < 0 or k > n)
            return Z{};
        return fac[n] * ifac[n - k];
    }
};
```

2.6 线性基

维护异或线性空间。

- 默认处理 u64，最高位为 63；
- Insert(x) 返回 x 是否使秩增加；
- Contains(x) 判断 x 能否由当前线性基异或得到；
- MaxXor(x) 返回 x 与线性空间中某个元素异或后的最大值。

```
using u64 = uint64_t;

template <class T = u64, int LOG = 63>
struct LinearBasis {
    std::array<T, LOG + 1> p{};
    int rank = 0;

    bool Insert(T x) {
        for (int i = LOG; i >= 0; i--) {
            if (((x >> i) & 1) == 0)
                continue;
            if (!p[i]) {
                p[i] = x;
                rank++;
                return true;
            }
            x ^= p[i];
        }
        return false;
    }

    bool Contains(T x) const {
        for (int i = LOG; i >= 0; i--) {
            if (((x >> i) & 1) == 0)
                continue;
            if (!p[i])
                return false;
            x ^= p[i];
        }
        return true;
    }

    T MaxXor(T x = 0) const {
        for (int i = LOG; i >= 0; i--)
            if ((x ^ p[i]) > x)
                x ^= p[i];
        return x;
    }

    std::vector<T> Basis() const {
        std::vector<T> res;
        for (int i = 0; i <= LOG; i++)
            if (p[i])
                res.emplace_back(p[i]);
        return res;
    }
};
```

2.7 矩阵快速幂

定长静态矩阵乘法与快速幂，转置 + 分块延迟取模压常数。

- 按题目修改 MOD 与 N；要求 MOD < 2^30，否则 16 组乘积的 u64 累加会溢出；
- 用法：输入写进 a，设好 n，调用 Pow(k)，结果在 b；
- Pow(0) 返回单位矩阵；b 每次调用时重新初始化，可重复调用；
- 注意 Pow 会破坏 a（变为 a 的若干次平方）；
- 复杂度 O(n^3 log k)；mul 先把右操作数转置成按行访问，再以 16 列为块累加进 u64、块末取一次模；
- 线性递推加速：m 阶递推压成 m x m 转移矩阵的幂。

```
using u32 = unsigned;
using u64 = uint64_t;
using i64 = int64_t;

namespace MatrixOps {
    constexpr int MOD = 998244353;
    constexpr int N = 200;
    u32 a[N][N], b[N][N];
    int n;

    void mul(const u32 A[N][N], const u32 B[N][N],
             u32 C[N][N]) {
        static u32 bt[N][N];
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                bt[j][i] = B[i][j];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                u64 res = 0;
                int k = 0;
                for (; k + 15 < n; k += 16) {
                    u64 sum = 0;
                    for (int d = 0; d < 16; d++)
                        sum += u64(A[i][k + d]) * bt[j][k + d];
                    res += sum % MOD;
                }
                u64 sum = 0;
                for (; k < n; k++)
                    sum += u64(A[i][k]) * bt[j][k];
                res += sum % MOD;
                C[i][j] = res % MOD;
            }
        }
    }

    void Pow(i64 y) {
        static u32 t[N][N];
        for (int i = 0; i < n; i++) {
            std::fill(b[i], b[i] + n, 0u);
            b[i][i] = 1;
        }
        while (y) {
            if (y & 1) {
                mul(a, b, t);
                std::memcpy(b, t, sizeof(b));
            }
            y >>= 1;
            if (!y)
                break;
            mul(a, a, t);
            std::memcpy(a, t, sizeof(a));
        }
    }
}
```

3 数据结构

3.1 并查集 (DSU)

维护无向连通性和连通块大小，Merge(x, y) 返回是否真的合并成功。

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

    DSU(int n) : f(n), sz(n, 1) {
        std::iota(f.begin(), f.end(), 0);
    }

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

    auto Merge(int x, int y) {
        x = Find(x), y = Find(y);
        if (x == y)
            return false;
        if (sz[x] < sz[y])
            std::swap(x, y);
        sz[x] += sz[y];
        f[y] = x;
        return true;
    }

    auto Size(int x) { return sz[Find(x)]; }
};
```

3.2 树状数组

维护单点加、前缀和、区间和。

- 使用 1-indexed；
- Sum(l, r) 查询闭区间 [l, r]；
- 单次操作复杂度 O(log n)。

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

    Fenwick(int n) : n(n), a(n + 1) {}

    void Add(int x, T v) {
        for (; x <= n; x += x & -x)
            a[x] += v;
    }

    auto sum(int x) {
        T res = {};
        for (; x; x -= x & -x)
            res += a[x];
        return res;
    }

    auto Sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }
};
```

3.3 线段树

维护单点修改、区间查询。

- 单次操作复杂度 O(log n)。

关于 S 的约束：

- 存在满足封闭性结合律的 operator+ 运算；
- 存在单位元 {}（具有默认构造函数，且满足和单位元运算后不改变原值）

关于初始区间：

- 使用随机访问迭代器；
- 若元素类型 ≠ S，则必须保证 S 存在对应构造函数。

```
template <class S>
struct SegTree {

    int n;
    std::vector<S> tr;

    SegTree(int n, const S& e) {
        build(n, std::vector<S>(n, e));
    }

    template <class It>
    SegTree(It l, It r) {
        build(r - l, l);
    }

    template <class Arr>
    void build(int m, const Arr& arr) {
        for (n = 1; n < m; n <<= 1)
            ;
        tr.resize(n << 1);
        for (int i = 0; i < m; i++)
            tr[i + n] = arr[i];
        for (int i = n - 1; i >= 1; i--)
            pull(i);
    }

    void pull(int k) {
        tr[k] = tr[k << 1] + tr[k << 1 | 1];
    }

    void Set(int p, const S& x) {
        p += n;
        tr[p] = x;
        for (p >>= 1; p; p >>= 1)
            pull(p);
    }

    auto Get(int p) { return tr[p + n]; }

    auto Query(int l, int r) {
        l += n, r += n;
        S sml{}, smr{};
        while (l < r) {
            if (l & 1)
                sml = sml + tr[l++];
            if (r & 1)
                smr = tr[--r] + smr;
            l >>= 1;
            r >>= 1;
        }
        return sml + smr;
    }
};
```

S 的经典实例——维护最大子段和（Kadane 合并）。空状态用哨兵 -1e18 而 sum、len 取 0，恰好构成单位元：

```
using i64 = int64_t;

struct Kadane {
    i64 len{};
    i64 sum{}, ans = -1e18;
    i64 pre = -1e18, suf = -1e18;

    Kadane() = default;
    Kadane(i64 v)
        : len(1), sum(v), ans(v), pre(v), suf(v) {}

    friend Kadane operator+(const Kadane& l, const Kadane& r) {
        Kadane res;
        res.len = l.len + r.len;
        res.sum = l.sum + r.sum;
        res.pre = std::max(l.pre, l.sum + r.pre);
        res.suf = std::max(r.suf, l.suf + r.sum);
        res.ans = std::max({l.ans, r.ans, l.suf + r.pre});
        return res;
    }
};
```

3.4 懒标记线段树

使用非递归的实现方式。

- 单次操作复杂度 O(log n)。

关于 F 的约束：

- 存在满足封闭性的 operator+= 运算；
- 存在一个恒等映射 {}（默认构造函数）。

关于 S 的约束：

- 存在满足封闭性结合律的 operator+ 运算；
- 存在单位元 {}（具有默认构造函数，且满足和单位元运算后不改变原值）
- 存在 operator*= 运算满足将映射 F 应用于 S 返回一个 S，并且满足分配律。

关于初始区间：

- 使用随机访问迭代器；
- 若元素类型 ≠ S，则必须保证 S 存在对应构造函数。

```
template <class S, class F>
struct LazySegTree {

    int n, h;
    std::vector<S> tr;
    std::vector<F> lz;

    LazySegTree(int n, const S& e) {
        build(n, std::vector<S>(n, e));
    }

    template <class It>
    LazySegTree(It l, It r) {
        build(r - l, l);
    }

    template <class Arr>
    void build(int m, const Arr& arr) {
        for (n = 1; n < m; n <<= 1)
            ;
        h = __builtin_ctz(n);
        tr.resize(n << 1);
        lz.resize(n);
        for (int i = 0; i < m; i++)
            tr[i + n] = arr[i];
        for (int i = n - 1; i >= 1; i--)
            pull(i);
    }

    void apply(int k, const F& f) {
        tr[k] *= f;
        if (k < n)
```

```
            lz[k] += f;
    }

    void pull(int k) {
        tr[k] = tr[k << 1] + tr[k << 1 | 1];
    }

    void push(int k) {
        apply(k << 1, lz[k]);
        apply(k << 1 | 1, lz[k]);
        lz[k] = {};
    }

    void Set(int p, const S& x) {
        p += n;
        for (int i = h; i >= 1; i--)
            push(p >> i);
        tr[p] = x;
        for (int i = 1; i <= h; i++)
            pull(p >> i);
    }

    auto Get(int p) {
        p += n;
        for (int i = h; i >= 1; i--)
            push(p >> i);
        return tr[p];
    }

    void Update(int l, int r, const F& f) {
        l += n, r += n;
        for (int i = h; i >= 1; i--) {
            if ((l & ((1 << i) - 1)) != 0)
                push(l >> i);
            if ((r & ((1 << i) - 1)) != 0)
                push((r - 1) >> i);
        }
        {
            int l_ = l, r_ = r;
            while (l < r) {
                if (l & 1)
                    apply(l++, f);
                if (r & 1)
                    apply(--r, f);
                l >>= 1;
                r >>= 1;
            }
            l = l_;
            r = r_;
        }
        for (int i = 1; i <= h; i++) {
            if ((l & ((1 << i) - 1)) != 0)
                pull(l >> i);
            if ((r & ((1 << i) - 1)) != 0)
                pull((r - 1) >> i);
        }
    }

    auto Query(int l, int r) {
        l += n, r += n;
        for (int i = h; i >= 1; i--) {
            if ((l & ((1 << i) - 1)) != 0)
                push(l >> i);
            if ((r & ((1 << i) - 1)) != 0)
                push((r - 1) >> i);
        }
        S sml{}, smr{};
        while (l < r) {
            if (l & 1)
                sml = sml + tr[l++];
            if (r & 1)
                smr = tr[--r] + smr;
            l >>= 1;
            r >>= 1;
        }
        return sml + smr;
    }
};
```

3.5 异或 Trie

维护非负整数可重集，支持插入、删除和查询与 x 异或的最大值。

- 默认考虑 [30, 0] 位；
- 删除前需要保证元素存在；
- 查询前需要保证集合非空；
- 单次操作复杂度均为 O(31)。

```
struct XorTrie {
    std::vector<std::array<int, 2>> ch;
    std::vector<int> cnt;
    int tot = 1;

    XorTrie(int n) : ch(n * 32), cnt(n * 32) {}

    void Insert(int x) {
        int u = 1;
        cnt[u]++;
        for (int i = 30; i >= 0; i--) {
            int c = (x >> i) & 1;
            if (not ch[u][c]) {
                ch[u][c] = ++tot;
            }
            u = ch[u][c];
            cnt[u]++;
        }
    }

    void Erase(int x) {
        int u = 1;
        cnt[u]--;
        for (int i = 30; i >= 0; i--) {
            int c = (x >> i) & 1;
            u = ch[u][c];
            cnt[u]--;
        }
    }

    auto Query(int x) {
        int res = 0;
        int u = 1;
        for (int i = 30; i >= 0; i--) {
            int c = (x >> i) & 1;
            int v = ch[u][c ^ 1];
            if (v and cnt[v] > 0) {
                u = v;
                res |= (1 << i);
            } else {
                u = ch[u][c];
            }
        }
        return res;
    }
};
```

3.6 ST 表

维护静态区间查询。

- 查询区间为左闭右开 [l, r)；
- 运算需要满足结合律和幂等性，例如 min、max、gcd；
- 预处理复杂度 O(n log n)，单次查询 O(1)。

```
template <class T>
struct SparseTable {
    using Op = std::function<T(const T&, const T&)>;

    int n = 0;
    Op op;
    std::vector<int> lg;
    std::vector<std::vector<T>> st;

    SparseTable() = default;

    SparseTable(const std::vector<T>& a, Op op)
        : op(std::move(op)) {
        Build(a.begin(), a.end());
    }

    template <class It>
    SparseTable(It l, It r, Op op)
        : op(std::move(op)) {
        Build(l, r);
    }

    template <class It>
    void Build(It l, It r) {
        n = int(r - l);
        lg.assign(n + 1, 0);
        for (int i = 2; i <= n; i++)
            lg[i] = lg[i >> 1] + 1;
        st.assign(lg[n] + 1, std::vector<T>(n));
        for (int i = 0; i < n; i++)
            st[0][i] = *(l + i);
        for (int k = 1; k < int(st.size()); k++) {
            int len = 1 << k;
            for (int i = 0; i + len <= n; i++) {
                st[k][i] =
                    op(st[k - 1][i],
                       st[k - 1][i + (len >> 1)]);
            }
        }
```

```
    }

    T Query(int l, int r) const {
        assert(0 <= l and l < r and r <= n);
        int k = lg[r - l];
        return op(st[k][l], st[k][r - (1 << k)]);
    }
};
```

3.7 可并堆

pb_ds 配对堆，免手写左偏树。

- 需要额外引入 <ext/pb_ds/priority_queue.hpp>；
- 默认大根堆，小根堆传 std::greater<T>；
- push 返回 point_iterator 句柄，句柄在 join 之后仍然有效，可用于 modify(it, v) 与 erase(it)；
- a.join(b) 把 b 并入 a 并清空 b，均摊 O(1)； pop 均摊 O(log n)；
- 按集合合并时常配 DSU：以 DSU 的 sz 决定 join 方向，堆下标始终用 Find 后的代表元。

```
#include <ext/pb_ds/priority_queue.hpp>

template <class T, class Cmp = std::less<T>>
using MeldHeap = __gnu_pbds::priority_queue<
    T, Cmp, __gnu_pbds::pairing_heap_tag>;

// MeldHeap<int, std::greater<int>> h;  小根堆
// auto it = h.push(x);                 稳定句柄
// h.modify(it, v), h.erase(it);
// a.join(b);                           b 被清空
```

3.8 势能线段树

处理不满足结合律、但会让元素快速收敛的区间修改（区间开方、区间取模等）：额外维护区间最大值，整段已收敛则直接跳过。

- 模板实现区间开方 + 区间求和（洛谷 P4145）： mx <= 1 的子树开方不变，整棵剪掉；
- 每个元素开方 O(log log V) 次后收敛到 1，总复杂度 O((n + q log n) log log V) 量级；
- 区间取模变体：改判 mx < x 则跳过，叶子执行 a_i %= x （每次有效取模至少减半，势能同理）；
- 修改必须递归到叶子逐个执行，不能打懒标记—— 剪枝条件是正确性的全部来源。

```
using i64 = int64_t;

struct PotSegTree {
    struct Node {
        i64 sum = 0, mx = 0;
    };

    int n;
    std::vector<Node> tr;

    template <class It>
    PotSegTree(It first, It last)
        : n(last - first), tr(4 * n) {
        build(first, 1, 0, n - 1);
    }

    template <class It>
    void build(It a, int p, int l, int r) {
        if (l == r) {
            tr[p] = {a[l], a[l]};
            return;
        }
        int mid = (l + r) / 2;
        build(a, 2 * p, l, mid);
        build(a, 2 * p + 1, mid + 1, r);
        pull(p);
    }

    void pull(int p) {
        tr[p].sum = tr[2 * p].sum + tr[2 * p + 1].sum;
        tr[p].mx = std::max(tr[2 * p].mx, tr[2 * p + 1].mx);
    }

    void Sqrt(int l, int r) { sqrt(l, r, 1, 0, n - 1); }

    void sqrt(int ql, int qr, int p, int l, int r) {
        if (qr < l or r < ql or tr[p].mx <= 1)
            return;
        if (l == r) {
            tr[p].sum = tr[p].mx = i64(sqrtl(tr[p].sum));
            return;
        }
        int mid = (l + r) / 2;
        sqrt(ql, qr, 2 * p, l, mid);
        sqrt(ql, qr, 2 * p + 1, mid + 1, r);
        pull(p);
    }

    i64 Query(int l, int r) {
        return query(l, r, 1, 0, n - 1);
    }

    i64 query(int ql, int qr, int p, int l, int r) {
        if (qr < l or r < ql)
            return 0;
        if (ql <= l and r <= qr)
            return tr[p].sum;
        int mid = (l + r) / 2;
        return query(ql, qr, 2 * p, l, mid) +
               query(ql, qr, 2 * p + 1, mid + 1, r);
    }
};
```

3.9 可持久化权值线段树

主席树：按前缀建版本的权值线段树，两版本相减得到 任意区间的值域信息。

- 值域须先离散化到 [1, n]；构造后按原数组顺序逐个 Add，版本 i 对应前缀 [1, i]；
- Kth(l, r, k)：下标区间 [l, r]（1-indexed）第 k 小的 离散化值；Count(l, r, x, y)：区间内值落在 [x, y] 的个数；
- 单次操作 O(log n)；节点数约 (插入次数) x (log n + 1)， 大数据先 tr.reserve 防反复扩容；
- 版本 0 是空树（节点 0 自环为空儿子），无需特判。

```
struct HJT {
    struct Node {
        int l = 0, r = 0, cnt = 0;
    };

    int n;
    std::vector<Node> tr;
    std::vector<int> root;

    HJT(int n) : n(n), tr(1), root(1, 0) {}

    void Add(int x) {
        root.push_back(add(root.back(), 1, n, x));
    }

    int add(int v, int l, int r, int x) {
        int u = tr.size();
        tr.push_back(tr[v]);
        tr[u].cnt++;
        if (l == r)
            return u;
        int mid = (l + r) / 2;
        if (x <= mid)
            tr[u].l = add(tr[v].l, l, mid, x);
        else
            tr[u].r = add(tr[v].r, mid + 1, r, x);
        return u;
    }

    int Kth(int ql, int qr, int k) {
        return kth(root[ql - 1], root[qr], 1, n, k);
    }

    int kth(int u, int v, int l, int r, int k) {
        if (l == r)
            return l;
        int mid = (l + r) / 2;
        int res = tr[tr[v].l].cnt - tr[tr[u].l].cnt;
        if (k <= res)
            return kth(tr[u].l, tr[v].l, l, mid, k);
        return kth(tr[u].r, tr[v].r, mid + 1, r, k - res);
    }

    int Count(int ql, int qr, int x, int y) {
        return count(root[ql - 1], root[qr], 1, n, x, y);
    }

    int count(int u, int v, int l, int r, int x, int y) {
        if (x <= l and r <= y)
            return tr[v].cnt - tr[u].cnt;
        int mid = (l + r) / 2;
        int res = 0;
        if (x <= mid)
```

```
            res += count(tr[u].l, tr[v].l, l, mid, x, y);
        if (y > mid)
            res += count(tr[u].r, tr[v].r, mid + 1, r, x, y);
        return res;
    }
};
```

4 图论

4.1 单源最短路

适用于非负边权图。

- adj[u] 存储 (v, w);
- 默认点编号为 1..n;
- 复杂度 O((n + m) log n)。

```
using i64 = int64_t;
constexpr i64 INF = 4'000'000'000'000'000'000LL;

template <class Adj>
auto dijkstra(const Adj& adj, int n, int s) {
    vector<i64> dis(n + 1, INF);
    vector<bool> vis(n + 1, false);
    priority_queue<pair<i64, int>> pq;
    dis[s] = 0;
    pq.emplace(0, s);
    while (!pq.empty()) {
        auto [_, u] = pq.top();
        pq.pop();
        if (vis[u])
            continue;
        vis[u] = true;
        for (auto [v, w] : adj[u]) {
            if (dis[u] + w < dis[v]) {
                dis[v] = dis[u] + w;
                pq.emplace(-dis[v], v);
            }
        }
    }
    return dis;
}
```

4.2 分层图最短路

适用于非负边权图上最多使用 k 次特殊操作。

P4568 的建模方式：

- dist[u][i] 表示到达点 u，且已经用了 i 次免费机会的最小代价;
- 走普通边：(u, i) -> (v, i)，边权为 w;
- 若 i < k，则可以免费走这条边：(u, i) -> (v, i + 1)，边权为 0;
- 答案为 min(dist[t][0..k])。

复杂度 O((n k + m k) log(n k))。

```
template <class Adj>
auto LayeredDijkstra(const Adj& adj, int n, int s,
                     int k) {
    using i64 = int64_t;
    constexpr i64 INF = 4'000'000'000'000'000'000LL;
    vector dist(n + 1, vector<i64>(k + 1, INF));
    priority_queue<tuple<i64, int, int>> pq;
    dist[s][0] = 0;
    pq.emplace(0, s, 0);
    while (!pq.empty()) {
        auto [d, u, used] = pq.top();
        pq.pop();
        d = -d;
        if (d != dist[u][used])
            continue;
        for (auto [v, w] : adj[u]) {
            if (dist[u][used] + w < dist[v][used]) {
                dist[v][used] = dist[u][used] + w;
                pq.emplace(-dist[v][used], v, used);
            }
            if (used < k and
                dist[u][used] < dist[v][used + 1]) {
                dist[v][used + 1] = dist[u][used];
                pq.emplace(-dist[v][used + 1], v,
                           used + 1);
            }
        }
    }
    return dist;
}
```

4.3 差分约束

Bellman-Ford 判负环并求一组可行解。

- 约束形如 x[v] <= x[u] + w;
- edges 存储 (u, v, w);
- 无解返回空数组。

```
using i64 = int64_t;

auto DifferenceConstraints(
    int n, const vector<array<int, 3>>& edges) {
    vector<i64> d(n + 1);
    for (int i = 1; i <= n; i++) {
        bool changed = false;
        for (auto [u, v, w] : edges) {
            if (d[v] > d[u] + w) {
                d[v] = d[u] + w;
                changed = true;
            }
        }
        if (!changed)
            break;
        if (i == n)
            return vector<i64>{};
    }
    return d;
}
```

4.4 拓扑排序

Kahn 算法。

- 默认点编号 1..n;
- 若返回数量小于 n，则图中有环;
- 复杂度 O(n + m)。

```
auto TopoSort(const vector<vector<int>>& adj,
              int n) {
    vector<int> indeg(n + 1), res;
    queue<int> q;
    for (int u = 1; u <= n; u++)
        for (int v : adj[u])
            indeg[v]++;
    for (int i = 1; i <= n; i++)
        if (!indeg[i])
            q.emplace(i);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        res.emplace_back(u);
        for (int v : adj[u])
            if (--indeg[v] == 0)
                q.emplace(v);
    }
    return res;
}
```

4.5 强连通分量 (SCC)

Tarjan 求有向图强连通分量。

- 默认点编号 1..n;
- id[u] 为 u 所在 SCC 编号;
- 复杂度 O(n + m)。

```
struct SCC {
    int n, now = 0, cnt = 0;
    vector<vector<int>> adj;
    vector<int> dfn, low, stk, ins, id;

    SCC(int n)
        : n(n), adj(n + 1), dfn(n + 1), low(n + 1),
          ins(n + 1), id(n + 1) {}

    void AddEdge(int u, int v) {
        adj[u].emplace_back(v);
    }

    void tarjan(int u) {
        dfn[u] = low[u] = ++now;
        stk.emplace_back(u);
        ins[u] = true;
        for (int v : adj[u]) {
            if (!dfn[v]) {
                tarjan(v);
                low[u] = min(low[u], low[v]);
            } else if (ins[v]) {
```

```
                low[u] = min(low[u], dfn[v]);
            }
        }
        if (dfn[u] == low[u]) {
            cnt++;
            while (true) {
                int x = stk.back();
                stk.pop_back();
                ins[x] = false;
                id[x] = cnt;
                if (x == u)
                    break;
            }
        }
    }

    std::pair<std::vector<int>, int> Work() {
        for (int i = 1; i <= n; i++)
            if (!dfn[i])
                tarjan(i);
        return {id, cnt};
    }
};
```

4.6 割点

Tarjan 求无向图割点。

- 默认点编号 1..n；
- 返回所有割点；
- 复杂度 O(n + m)。

```
auto CutVertex(const vector<vector<int>>& adj,
               int n) {
    vector<int> dfn(n + 1), low(n + 1), cut(n + 1),
        res;
    int now = 0;
    auto dfs = [&](auto&& self, int u,
                   int p) -> void {
        dfn[u] = low[u] = ++now;
        int child = 0;
        for (int v : adj[u]) {
            if (!dfn[v]) {
                child++;
                self(self, v, u);
                low[u] = min(low[u], low[v]);
                if (p != 0 and low[v] >= dfn[u])
                    cut[u] = true;
            } else if (v != p) {
                low[u] = min(low[u], dfn[v]);
            }
        }
        if (p == 0 and child >= 2)
            cut[u] = true;
    };
    for (int i = 1; i <= n; i++)
        if (!dfn[i])
            dfs(dfs, i, 0);
    for (int i = 1; i <= n; i++)
        if (cut[i])
            res.emplace_back(i);
    return res;
}
```

4.7 Dinic 最大流

分层图 + 当前弧优化求最大流，链式前向星扁平存边。

- 默认点编号 1..n；
- AddEdge(u, v, c) 添加一条容量为 c 的有向边，返回全局边编号（正反两条成对，id ^ 1 即反向边）；
- Flow(id) 返回该边的实际流量；
- BFS 在弹出深度不小于 dep[t] 的节点时截断：层图到 t 所在层仍然完整，复杂度证明不受影响；
- 一般图复杂度 O(n^2m)，二分图等特殊图更快。

```
using i64 = int64_t;

struct Dinic {
    int n;
    vector<int> to, nxt, head, dep, cur, que;
    vector<i64> cap;

    Dinic(int n)
        : n(n), head(n + 1, -1), dep(n + 1), cur(n + 1) {}

    int AddEdge(int u, int v, i64 c) {
        int id = to.size();
        to.push_back(v), nxt.push_back(head[u]);
        cap.push_back(c);
        head[u] = id;
        to.push_back(u), nxt.push_back(head[v]);
        cap.push_back(0);
        head[v] = id + 1;
        return id;
    }

    i64 Flow(int id) const { return cap[id ^ 1]; }

    i64 MaxFlow(int s, int t) {
        i64 flow = 0;
        constexpr i64 INF = numeric_limits<i64>::max() / 4;
        while (bfs(s, t)) {
            copy(head.begin(), head.end(), cur.begin());
            while (i64 f = dfs(s, t, INF))
                flow += f;
        }
        return flow;
    }

    bool bfs(int s, int t) {
        fill(dep.begin(), dep.end(), -1);
        que.clear();
        que.push_back(s);
        dep[s] = 0;
        for (size_t i = 0; i < que.size(); i++) {
            int u = que[i];
            if (dep[t] != -1 and dep[u] >= dep[t])
                break;
            for (int e = head[u]; e != -1; e = nxt[e]) {
                if (cap[e] > 0 and dep[to[e]] == -1) {
                    dep[to[e]] = dep[u] + 1;
                    que.push_back(to[e]);
                }
            }
        }
        return dep[t] != -1;
    }

    i64 dfs(int u, int t, i64 f) {
        if (u == t or f == 0)
            return f;
        for (int& e = cur[u]; e != -1; e = nxt[e]) {
            int v = to[e];
            if (cap[e] <= 0 or dep[v] != dep[u] + 1)
                continue;
            i64 w = dfs(v, t, min(f, cap[e]));
            if (w == 0)
                continue;
            cap[e] -= w;
            cap[e ^ 1] += w;
            return w;
        }
        return 0;
    }
};
```

4.8 二分图最大匹配

匈牙利算法。

- 左部点 1..n，右部点 1..m；
- adj[u] 存储左部点 u 能匹配的右部点；
- 复杂度 O(nm)，稀疏图通常够用。

```
auto BipartiteMatching(
    const vector<vector<int>>& adj, int n, int m) {
    vector<int> mt(m + 1), vis(m + 1);
    int ans = 0, stamp = 0;
    auto dfs = [&](auto&& self,
                   int u) -> bool {
        for (int v : adj[u]) {
            if (vis[v] == stamp)
                continue;
            vis[v] = stamp;
            if (!mt[v] or self(self, mt[v])) {
                mt[v] = u;
                return true;
            }
        }
        return false;
    };
    for (int i = 1; i <= n; i++) {
        stamp++;
        ans += dfs(dfs, i);
    }
    return ans;
}
```

4.9 Hopcroft-Karp

二分图最大匹配。

- 左部点 1..n，右部点 1..m；
- AddEdge(u, v) 添加一条左部 u 到右部 v 的边；
- matchL[u] 是左部点 u 匹配到的右部点；
- 复杂度 O(E \sqrt V)。

```
struct HopcroftKarp {
    int n, m;
    vector<vector<int>> adj;
    vector<int> matchL, matchR, dis;

    HopcroftKarp(int n, int m)
        : n(n), m(m), adj(n + 1), matchL(n + 1),
          matchR(m + 1), dis(n + 1) {}

    void AddEdge(int u, int v) {
        adj[u].emplace_back(v);
    }

    int Work() {
        int ans = 0;
        while (bfs()) {
            for (int u = 1; u <= n; u++)
                if (!matchL[u] and dfs(u))
                    ans++;
        }
        return ans;
    }

    bool bfs() {
        queue<int> q;
        fill(dis.begin(), dis.end(), -1);
        for (int u = 1; u <= n; u++) {
            if (!matchL[u]) {
                dis[u] = 0;
                q.emplace(u);
            }
        }
        bool found = false;
        while (!q.empty()) {
            int u = q.front();
            q.pop();
            for (int v : adj[u]) {
                int x = matchR[v];
                if (!x) {
                    found = true;
                } else if (dis[x] == -1) {
                    dis[x] = dis[u] + 1;
                    q.emplace(x);
                }
            }
        }
        return found;
    }

    bool dfs(int u) {
        for (int v : adj[u]) {
            int x = matchR[v];
            if (!x or (dis[x] == dis[u] + 1 and dfs(x))) {
                matchL[u] = v;
                matchR[v] = u;
                return true;
            }
        }
        dis[u] = -1;
        return false;
    }
};
```

5 树上问题

5.1 LCA

倍增求树上最近公共祖先。

- 默认点编号 1..n；
- adj 是无向树；
- Get(u, v) 返回 u 和 v 的 LCA；
- Dis(u, v) 返回 u 和 v 的距离；
- Kth(u, v, k) 返回从 u 到 v 路径上的第 k 个点，k 从 0 开始；
- Component(u, v) 返回删掉点 u 后 v 所在连通块的大小，要求 u 和 v 相邻；
- 预处理复杂度 O(n log n)，单次查询 O(log n)。

```
struct LCA {
    int n, LOG;
    vector<int> dep, siz;
    vector<vector<int>> up;

    LCA(const vector<vector<int>>& adj, int root = 1) {
        n = adj.size() - 1;
        LOG = __lg(n) + 1;
        dep.assign(n + 1, 0);
        siz.assign(n + 1, 1);
        up.assign(LOG, vector<int>(n + 1, root));

        auto dfs = [&](auto&& self, int u,
                       int p) -> void {
            up[0][u] = p;
            for (int i = 1; i < LOG; i++)
                up[i][u] = up[i - 1][up[i - 1][u]];
            for (int v : adj[u]) {
                if (v == p)
                    continue;
                dep[v] = dep[u] + 1;
                self(self, v, u);
                siz[u] += siz[v];
            }
        };
        dfs(dfs, root, root);
    }

    int Get(int u, int v) const {
        if (dep[u] < dep[v])
            swap(u, v);
        u = jump(u, dep[u] - dep[v]);
        if (u == v)
            return u;
        for (int i = LOG - 1; i >= 0; i--) {
            if (up[i][u] != up[i][v]) {
                u = up[i][u];
                v = up[i][v];
            }
        }
        return up[0][u];
    }

    int Dis(int u, int v) const {
        int g = Get(u, v);
        return dep[u] + dep[v] - 2 * dep[g];
    }

    int Kth(int u, int v, int k) const {
        int g = Get(u, v);
        int du = dep[u] - dep[g];
        int d = du + dep[v] - dep[g];
        if (k <= du)
            return jump(u, k);
        return jump(v, d - k);
    }

    int Component(int u, int v) const {
        if (up[0][v] == u)
            return siz[v];
        return n - siz[u];
    }

    int jump(int u, int k) const {
        for (int i = 0; i < LOG; i++)
            if (k >> i & 1)
                u = up[i][u];
        return u;
    }
};
```

5.2 树上差分

对树上路径做批量加法，再一次 DFS 汇总。

- 默认点编号 1..n；
- AddVertexPath(u, v, w) 给路径上的点加 w；
- AddEdgePath(u, v, w) 给路径上的边加 w；
- Work() 返回汇总后的差分值。对于边差分，边权存放在子节点上。

```
using i64 = int64_t;

struct TreeDifference {
    int n, LOG;
    vector<vector<int>> adj, up;
    vector<int> dep;
    vector<i64> diff;

    TreeDifference(const vector<vector<int>>& adj,
                   int root = 1)
        : n(adj.size() - 1), adj(adj),
```

```
          LOG(__lg(n) + 1), up(LOG, vector<int>(n + 1, root)),
          dep(n + 1), diff(n + 1) {
        auto dfs = [&](auto&& self, int u,
                       int p) -> void {
            up[0][u] = p;
            for (int i = 1; i < LOG; i++)
                up[i][u] = up[i - 1][up[i - 1][u]];
            for (int v : adj[u]) {
                if (v == p)
                    continue;
                dep[v] = dep[u] + 1;
                self(self, v, u);
            }
        };
        dfs(dfs, root, root);
    }

    void AddVertexPath(int u, int v,
                       i64 w = 1) {
        int g = lca(u, v);
        diff[u] += w;
        diff[v] += w;
        diff[g] -= w;
        if (up[0][g] != g)
            diff[up[0][g]] -= w;
    }

    void AddEdgePath(int u, int v,
                     i64 w = 1) {
        int g = lca(u, v);
        diff[u] += w;
        diff[v] += w;
        diff[g] -= 2 * w;
    }

    vector<i64> Work(int root = 1) {
        auto res = diff;
        auto dfs = [&](auto&& self, int u,
                       int p) -> void {
            for (int v : adj[u]) {
                if (v == p)
                    continue;
                self(self, v, u);
                res[u] += res[v];
            }
        };
        dfs(dfs, root, root);
        return res;
    }

    int jump(int u, int k) const {
        for (int i = 0; i < LOG; i++)
            if (k >> i & 1)
                u = up[i][u];
        return u;
    }

    int lca(int u, int v) const {
        if (dep[u] < dep[v])
            swap(u, v);
        u = jump(u, dep[u] - dep[v]);
        if (u == v)
            return u;
        for (int i = LOG - 1; i >= 0; i--) {
            if (up[i][u] != up[i][v]) {
                u = up[i][u];
                v = up[i][v];
            }
        }
        return up[0][u];
    }
};
```

5.3 Kruskal 重构树

把“按边权阈值连通”转成树上问题：按序加边，每次合并 新建内部节点记录边权，叶子 1..n 是原图点。

- 传入的边序自己定：按 w 升序建树，两点 LCA 的 val 是路径最大边权的最小值；按降序建树则是 路径最小边权的最大值（NOIP 货车运输）；
- 内部节点沿根方向 val 单调，“与 u 在阈值 w 内连通的 点集”是 u 某个祖先的整棵子树，可配倍增在祖先链上二分；
- 节点总数至多 2n - 1；图不连通时是森林， 查询前用 Find 判连通；
- 两点瓶颈查询：配本章 LCA 模板在重构树上求 val[lca(u, v)]。

```
struct KruskalTree {
    int n, tot;
    std::vector<int> f, val;
    std::vector<std::array<int, 2>> son;

    // es 中每条边为 {w, u, v}，按调用方给定的顺序依次合并
    KruskalTree(int n, const std::vector<std::array<int, 3>>&
es)
        : n(n), tot(n), f(2 * n), val(2 * n), son(2 * n) {
        std::iota(f.begin(), f.end(), 0);
        for (auto [w, u, v] : es) {
            int x = Find(u), y = Find(v);
            if (x == y)
                continue;
            ++tot;
            val[tot] = w;
            son[tot] = {x, y};
            f[x] = f[y] = tot;
        }
    }

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

6 字符串

6.1 KMP

求前缀函数，支持模式串匹配。

- Kmp(s)[i] 表示 s[0..i) 的 border 长度；
- 匹配复杂度 O(n + m)。

```
auto Kmp(const string& s) {
    int n = s.size();
    vector<int> f(n + 1);
    for (int i = 1, j = 0; i < n; i++) {
        while (j > 0 and s[i] != s[j])
            j = f[j];
        j += (s[i] == s[j]);
        f[i + 1] = j;
    }
    return f;
}
```

7 多项式与卷积

7.1 FFT

复数 FFT 求整数多项式卷积。

- 适合普通整数卷积；
- 返回长度为 a.size() + b.size() - 1 的结果；
- 复杂度 O(n log n)；
- 依赖 double 精度：需保证 n * max|a| * max|b| 不超过约 1e15，更大范围改用拆系数或 NTT。

```
using i64 = int64_t;
using comp = complex<double>;
const double PI = acos(-1);

void Fft(vector<comp>& a, bool inv) {
    int n = a.size();
    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;
        if (i < j)
            swap(a[i], a[j]);
    }
    for (int len = 2; len <= n; len <<= 1) {
        double ang = 2 * PI / len * (inv ? -1 : 1);
        comp wlen(cos(ang), sin(ang));
        for (int i = 0; i < n; i += len) {
            comp w(1);
            for (int j = 0; j < len / 2; j++) {
                comp u = a[i + j];
                comp v = a[i + j + len / 2] * w;
                a[i + j] = u + v;
                a[i + j + len / 2] = u - v;
                w *= wlen;
            }
        }
    }
    if (inv)
```

```
        for (auto& x : a)
            x /= n;
}

auto Convolution(const vector<i64>& a,
                 const vector<i64>& b) {
    if (a.empty() or b.empty())
        return vector<i64>{};
    int need = a.size() + b.size() - 1;
    int n = 1;
    while (n < need)
        n <<= 1;
    vector<comp> fa(a.begin(), a.end()),
        fb(b.begin(), b.end());
    fa.resize(n);
    fb.resize(n);
    Fft(fa, false);
    Fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    Fft(fa, true);
    vector<i64> res(need);
    for (int i = 0; i < need; i++)
        res[i] = llround(fa[i].real());
    return res;
}
```