基本算法

二分

求最小值，将区间划分为 [l, mid - 1] 和 [mid, r]，此时求 mid 相当于向下取整 (l + r) >> 1

求最大值，将区间划分为 [l, mid] 和 [mid + 1, r]，此时求 mid 相当于向上取整 (l + r + 1) >> 1

check 返回 true 表示：mid 满足题意，若当前求最大值，需要考虑提升下界 `l = mid`（得寸进尺），若当前求最小值，需要考虑降低上界 `r = mid`（方便找更小）

以下下标均从 1 开始

l + r >> 1 可优化为 l + ((r - l) >> 1);

求最小值

```
while(l < r) {
    int mid = l + r >> 1;
    if(check(mid)) r = mid;
    else l = mid + 1;
}
l
```

求最大值

```
while(l < r) {
    int mid = (l + r + 1) >> 1;
    if(check(mid)) l = mid;
    else r = mid - 1;
}
r
```

数据结构

树状数组

```cpp
/**
 * 树状数组（Fenwick Tree/BIT）实现区间最值
 * - 核心思想：利用二进制索引特性，每个节点存储特定区间的统计值
 * - 适用场景：单点更新 + 区间查询（最值/求和）
 * - 时间复杂度：O(n log n) 建树，O(log n) 更新/查询
 *
 * 模板说明：
 * 1. 修改合并方式：将 std::max 改为其他操作（如加法、最小值）
 * 2. 修改数据存储：a[] 存储原始数据，tree[] 存储区间统计值
 * 3. 注意最值操作的局限性：无法高效处理区间覆盖更新
 */

const int N = 2e5 + 10;
/**
 * 初始化 memset(tree, 0, sizeof tree);
 */
int n, m, a[N], tree[N];

int lowbit(int x) {
    return x & -x;
}

/**
 * 单点更新：更新所有包含x的区间
 * 初始化建树时 update(i, a[i]);
 */
void update(int x, int val) {
    for (int i = x; i <= n; i += lowbit(i)) {
        tree[x] = val;
        for (int i = 1; i < lowbit(x); i <<= 1) {
            tree[x] = std::max(tree[x], tree[x - i]);
        }
    }
}

/**
 * 区间查询：从右往左合并区间
 */
int query(int l, int r) {
    int ans = 0;
    while (r >= l) {
        ans = std::max(ans, a[r]); // 先比较原始数据
        --r;                       // 移动到下一个待处理位置
        // 如果剩余区间可以跳跃到前一个块
        while (r - lowbit(r) >= l) {
            ans = std::max(ans, tree[r]);
            r -= lowbit(r);
        }
    }
```

```
    return ans;
}
```

线段树

```
const int N = 2e5 + 5;
i64 a[N];

struct Node {
    int l, r;
    i64 max_val; // 根据需求修改val的类型和合并方式
} tree[N << 2];

inline int left_son(int x) { return x << 1; }
inline int right_son(int x) { return x << 1 | 1; }

/**
 * 合并左右子节点，示例最大值
 */
void push_up(int root) {
    tree[root].max_val = std::max(tree[left_son(root)].max_val,
tree[right_son(root)].max_val);
}

/**
 * 建树
 * build(1, 1, n);
 */
void build(int root, int l, int r) {
    tree[root].l = l;
    tree[root].r = r;
    if (l == r) {
        tree[root].max_val = a[l]; // 初始化叶子节点
        return;
    }
    int mid = (l + r) >> 1;
    build(left_son(root), l, mid);
    build(right_son(root), mid + 1, r);
    push_up(root);
}

/**
 * 单点更新
 */
void update(int root, int pos, i64 val) {
    if (tree[root].l == tree[root].r) {
        tree[root].max_val = val;
        return;
    }
    int mid = (tree[root].l + tree[root].r) >> 1;
    if (pos <= mid) {
        update(left_son(root), pos, val);
    } else {
        update(right_son(root), pos, val);
    }
```

```
    push_up(root);
}

/**
 * 区间查询
 */
i64 query(int root, int L, int R) {
    if (L <= tree[root].l && tree[root].r <= R) {
        return tree[root].max_val;
    }
    int mid = (tree[root].l + tree[root].r) >> 1;
    i64 max_val = 0; // 初始值需根据操作类型设定，如求和为0，最大为-INF等
    if (L <= mid) max_val = std::max(max_val, query(left_son(root), L, R));
    if (R > mid) max_val = std::max(max_val, query(right_son(root), L, R));
    return max_val;
}
```

珂朵莉树（Chtholly Tree）

珂朵莉树的适用范围是有 **区间赋值** 操作且 **数据随机** 的题目，时间复杂度比线段树低

ST 表

写 `st` 表大概率要 **关闭同步流**

```
/**
 * st表用于解决可重复贡献问题
 * 设有一个二元运算 op(x, y) ，满足 op(a, a) = a，则称op为可重复贡献的
 * 显然最大值、最小值、最大公因数、最小公倍数、按位或、按位与都符合这个条件
 *
 * std::__lg(x) 函数时间复杂度为 O(1)
 */

const int N = 5e4 + 3;
// 原数组a，最大值st表Max，最小值st表Min
// 这里为了内存局部性将较小者作为行数
int a[N], Max[std::__lg(N) + 1][N], Min[std::__lg(N) + 1][N];

/**
 * 初始化st表
 * st[i][j]: 表示闭区间 [i, i + 2^j - 1] 的操作结果
 * 显然 st[i][0] = ai
 *
 * st[i][j] = op( st[i][j - 1], st[i + 2^(j-1)][i - 1] )
 */
void init(int n) {
    for (int i = 1; i <= n; ++i) {
        Max[0][i] = a[i];
        Min[0][i] = a[i];
    }
    for (int i = 1; i <= std::__lg(n); ++i) {
        for (int j = 1; j + (1 << i) - 1 <= n; ++j) {
```

```
            Max[i][j] = std::max(Max[i - 1][j], Max[i - 1][j + (1 << (i - 1))]);
            Min[i][j] = std::min(Min[i - 1][j], Min[i - 1][j + (1 << (i - 1))]);
        }
    }
}

/**
 * 区间查询
 * s = floor(log_2(r - l + 1))
 * 将区间分为两部分 [l, l + 2^s - 1] 与 [r - 2^s + 1, r]
 * op两部分结果
 */
int query_max(int l, int r) {
    int s = std::__lg(r - l + 1);
    return std::max(Max[s][l], Max[s][r - (1 << s) + 1]);
}

int query_min(int l, int r) {
    int s = std::__lg(r - l + 1);
    return std::min(Min[s][l], Min[s][r - (1 << s) + 1]);
}
```

```
/**
 * 特别地，可动态追加的st表
 * 用于求解数组初始为空不断追加的问题
 * 与init系出同源
 */
void insert(int val) {
    len++;
    st[0][len] = val;
    for (int s = 1; (1 << s) <= len; s++) {
        int start = len - (1 << s) + 1;
        if (start < 1) break;
        st[s][start] = std::max(st[s - 1][start], st[s - 1][start + (1 << (s - 1))]);
    }
}
```

better

```
/**
 * @see https://codeforces.com/problemset/problem/622/C
 */
#include <bits/stdc++.h>

template <typename T>
struct St {
    int n;
    std::vector<T> a;                   // 下标从1开始
    std::vector<std::vector<T>> st;     // ST表
    const std::function<T(T, T)> cmp; // 比较规则

```

```cpp
    St(int n, const std::vector<T>& _a, std::function<T(T, T)> _cmp) :
            n(n), a(_a), cmp(_cmp) {
        assert(a.size() > n);

        const int LG = std::__lg(n);
        st.assign(LG + 1, std::vector<T>(n + 1));

        for (int i = 1; i <= n; i++) {
            st[0][i] = i;
        }

        for (int j = 1; j <= LG; j++) {
            for (int i = 1; i <= n - (1 << j) + 1; i++) {
                st[j][i] = cmp(st[j - 1][i], st[j - 1][i + (1 << j - 1)]);
            }
        }
    }

    /**
     * 区间查询
     */
    T query(int l, int r) {
        int k = std::__lg(r - l + 1);
        return cmp(st[k][l], st[k][r - (1 << k) + 1]);
    }
};

void solve() {
    int n, m;
    std::cin >> n >> m;
    std::vector<int> a(n + 1);
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
    }

    auto rmqMin = St<int>(n, a, [&](int i, int j) {
        return a[i] < a[j] ? i : j;
    });
    auto rmqMax = St<int>(n, a, [&](int i, int j) {
        return a[i] > a[j] ? i : j;
    });

    while (m--) {
        int l, r, x;
        std::cin >> l >> r >> x;

        int minidx = rmqMin.query(l, r), maxidx = rmqMax.query(l, r);
        if (a[minidx] == a[maxidx] && a[minidx] == x) {
            std::cout << -1 << std::endl;
        } else {
            std::cout << (a[minidx] != x ? minidx : maxidx) << std::endl;
        }
    }
}

int main() {
```

```
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    solve();
}
```

对顶堆

1. **动态求中位数**：
   - **大根堆** 保存较小的一半元素，**小根堆** 保存较大的一半。
   - 插入元素时调整堆的平衡，中位数由堆顶决定。
2. **实时 Top-K 查询**：
   - **小根堆** 保存当前最大的 `K` 个元素，堆顶是第 `K` 大的元素。
   - 新元素若大于堆顶则替换，并调整堆。
3. **流式数据的分位数统计**：
   - 通过调整两个堆的比例，快速查询任意分位数（如 90% 分位）。

```
class SORTracker {
public:
    struct Node {
        std::string name;
        int score;

        bool operator<(const Node& other) const {
            if (score != other.score) return score < other.score;
            return name > other.name;
        }

        bool operator>(const Node& other) const {
            if (score != other.score) return score > other.score;
            return name < other.name;
        }
    };

    int cur;
    std::priority_queue<Node> ge;                                    // 大根堆
    std::priority_queue<Node, std::vector<Node>, std::greater<>> le; // 小根堆

    SORTracker() :
            cur{1} {}

    void add(std::string name, int score) {
        Node n{name, score};
        if (le.empty() || le.top() < n) {
            le.push(n);
        } else {
            ge.push(n);
        }
```

```
        while (le.size() > cur) {
            ge.push(le.top());
            le.pop();
        }
        while (le.size() < cur && !ge.empty()) {
            le.push(ge.top());
            ge.pop();
        }
    }

    std::string get() {
        auto res = le.top().name;
        ++cur;
        while (le.size() < cur && !ge.empty()) {
            le.push(ge.top());
            ge.pop();
        }
        return res;
    }
};
```

```
std::priority_queue<int> a;
std::priority_queue<int, std::vector<int>, std::greater<>> b;

for(int i = 1; i <= n; ++i) {
    int x;
    std::cin >> x;
    if(b.empty() || b.top() < x) {
        b.push(x);
    } else {
        a.push(x);
    }
    int k = std::max(1, i * w / 100);
    while(b.size() > k) {
        a.push(b.top());
        b.pop();
    }
    while(b.size() < k) {
        b.push(a.top());
        a.pop();
    }
    std::cout << b.top() << ' ';
}
```

并查集

```
/**
 * @see https://codeforces.com/edu/course/2/lesson/7/1/practice/contest/289390/problem/B
 * 下标从 1 开始
 * 查询集合最小值、最大值、元素个数
 */
```

```cpp
#include <bits/stdc++.h>

const int N = 3e5 + 7;

struct Info {
    int max, min, cnt;
} info[N];

int fa[N];

void init(int n) {
    for (int i = 1; i <= n; ++i) {
        fa[i] = i;
        info[i] = {i, i, 1};
    }
}

int query(int x) {
    while (x != fa[x]) {
        x = fa[x] = fa[fa[x]];
    }
    return x;
}

bool merge(int x, int y) {
    int fx = query(x);
    int fy = query(y);
    if (fx == fy) {
        return false;
    }
    fa[fy] = fx;
    info[fx].max = std::max(info[fx].max, info[fy].max);
    info[fx].min = std::min(info[fx].min, info[fy].min);
    info[fx].cnt += info[fy].cnt;
    return true;
}

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

int main() {
    int n, m;
    std::cin >> n >> m;
    init(n);
    while (m--) {
        std::string op;
        int u, v;
        std::cin >> op;
        if (op == "get") {
            std::cin >> v;
            int fv = query(v);
            std::cout << info[fv].min << ' ' << info[fv].max << ' ' <<
info[fv].cnt << std::endl;
        } else {
            std::cin >> u >> v;
```

```
                merge(u, v);
            }
        }
    }
```

单调栈

- 从栈底到栈顶严格单调递减，可以找到当前栈顶对应元素在原序列中右边最近的比该元素大的元素
- 单调栈中可以存下标，视题目而定

```
while(!st.empty() && st.top() <= x) {
    st.pop();
}
```

平衡二叉树

替罪羊树

`Treap`

伸展树

树链剖分

图论

建图

邻接表

```cpp
const int N = 1e6 + 5;
std::vector<std::pair<int, int>> adj[N]; // adj[u]: (目标节点v, 边权w)

void addEdge(int u, int v, int w) {
    adj[u].push_back({v, w});
}
```

链式前向星

```cpp
const int N = 1e5 + 3;

int head[N], tot;
struct node {
    int to, next, w;
} edge[N << 1];

void add_edge(int u, int v, int w) {
    edge[++tot] = {v, head[u], w};
    head[u] = tot;
}

for(int i = head[u]; i; i = edge[u].next) {
    int v = edge[i].to;
    int w = edge[i].w;
    // ...
}
```

最短路径

结点 N、边 M	边权值	选用算法	数据结构
n < 200	允许有负	Floyd	邻接矩阵
n×m < 107	允许有负	Bellman-Ford	邻接表
更大	有负	SPFA	邻接表、前向星
更大	无负数	Dijkstra	邻接表、前向星

Floyd

所有点对最短路径

O(n^3)

利用 Floyd 算法很容易判断负圈，因为 `graph[i][i]` 是 i 到外面绕一圈回来的最小路径，若小于 0，说明存在负圈。可以置 `graph[i][i]=0`，并在 `floyd()` 中判断是否存在某个 `graph[i][i]<0`，若存在则说明该图中有负圈。

```cpp
constexpr int INF = 0x3f3f3f3f;

std::vector dis(n, std::vector<Node>(n, {INF, 0})); // 一般存int即权重，根据题意换成结构体
for (int i = 0; i < n; ++i) {
    dis[i][i] = {0, 0};
}

for (int k = 0; k < n; ++k) {
    for (int u = 0; u < n; ++u) {
        if(dis[u][k].w == INF) continue;
        for (int v = 0; v < n; ++v) {
            dis[u][v] = min(dis[u][v], dis[u][k] + dis[k][v]);
        }
    }
}
```

Dijkstra

单源最短路径

C++ 17 版

```cpp
constexpr int INF = 0x3f3f3f3f;

std::vector<std::vector<std::pair<int, i64>>> adj(n);
adj[u].push_back({v, w});
adj[v].push_back({u, w});

auto dijkstra = [&](int s, int t) {
    std::vector<int> dis(n, INF);
    std::priority_queue<std::pair<int, int>, std::vector<std::pair<int, int>>, std::greater<>> pq;
    pq.emplace(0, s);

    while(!pq.empty()) {
        auto[d, u] = pq.top();
        pq.pop();

        if(dis[u] != INF) {
            continue;
        }
        dis[u] = d;

        for(const auto&[v, w] : adj[u]) {
            pq.emplace(d + w, v);
        }
    }
    return dis[t];
};
```

C++ 11 版

```cpp
int dijkstra(int s, int t) {
    std::vector<int> dis(n, INF); // dis[i]: s到i的最短路径
    std::priority_queue<pii, std::vector<pii>, std::greater<pii>> pq; // (距离, 节点)
    pq.emplace(0, s);

    while (!pq.empty()) {
        auto x = pq.top(); // 取出当前距离起点最近的节点
        auto d = x.first;
        auto u = x.second;
        pq.pop();

        if (dis[u] != INF) continue;
        dis[u] = d; // 记录该节点的最终最短距离
        if (u == t) break;

        for (const auto& it : adj[u]) {
            auto v = it.first;
            auto w = it.second;
            pq.emplace(d + w, v); // 这里允许队列中存在多个v的不同距离值
        }
    }
    return dis[t];
}
```

字符串

字典树（`Trie`）

查询前缀个数

```cpp
const int N = 3e6 + 10;
const int ALPHABET = 62; // 数字 + 大小写字母，只有小写字母就设为26

int trie[N][ALPHABET];
int cnt[N]; // 前缀计数器
int tot;    // 当前节点总数

int get_idx(char c) {
    if (isdigit(c)) return c - '0';
    if (isupper(c)) return 10 + (c - 'A');
    return 36 + (c - 'a');
}

void init_trie() {
    memset(trie[0], 0, sizeof(trie[0]));
    tot = 0;
}

int create_node() {
    ++tot;
    memset(trie[tot], 0, sizeof(trie[tot]));
    cnt[tot] = 0;
    return tot;
}

void insert(const std::string& s) {
    int p = 0;
    for (char c : s) {
        int idx = get_idx(c);
        if (!trie[p][idx]) {
            trie[p][idx] = create_node();
        }
        p = trie[p][idx];
        ++cnt[p];
    }
}

int query(const std::string& s) {
    int p = 0;
    for (char c : s) {
        int idx = get_idx(c);
        if (!trie[p][idx]) return 0;
        p = trie[p][idx];
    }
    return cnt[p];
}
```

01 字典树

KMP 算法

```cpp
/**
 * 获取next数组
 * @par t 模式串
 * @return next数组
 * @note 时间复杂度 O(m)
 */
std::vector<int> get_next(const std::string& t) {
    int m = t.size();
    std::vector<int> next(m, 0);
    for (int i = 1, j = 0; i < m; ++i) { // j为模式串中已匹配的前缀长度
        while (j > 0 && t[i] != t[j]) {
            j = next[j - 1];
        }
        j += (t[i] == t[j]);
        next[i] = j;
    }
    return next;
}

/**
 * tmp算法匹配模式串
 * @par s 文本串，长度为n
 * @par t 模式串，长度为m
 * @return t在s中出现的所有位置（起始索引），若不存在，则返回空列表
 * @note 时间复杂度 O(n + m)
 */
std::vector<int> kmp(const std::string& s, const std::string& t) {
    std::vector<int> res;
    if (t.empty()) return res;
    std::vector<int> next = get_next(t);
    int n = s.size(), m = t.size();
    for (int i = 0, j = 0; i < n; ++i) { // j为模式串中已匹配的前缀长度
        while (j > 0 && s[i] != t[j]) {
            j = next[j - 1];
        }
        j += (s[i] == t[j]);
        // 模式串匹配完
        if (j == m) {
            res.push_back(i - m + 1);
            j = next[j - 1];
        }
    }
    return res;
}
```

Manacher 算法

```cpp
/**
 * r[i]表示以t中第i个字符为中心的最长回文半径（包括自身）
 * 求出来的 max(r[i] - 1) for i in r 即为最长回文子串的长度
 */
std::vector<int> manacher(const std::string& s) {
    std::string t = "#";
    for (auto c : s) {
        t += c;
        t += '#';
    }
    int n = t.size();
    std::vector<int> r(n);
    for (int i = 0, j = 0; i < n; ++i) {
        if (2 * j - i >= 0 && j + r[j] > i) {
            r[i] = std::min(r[2 * j - i], j + r[j] - i);
        }
        while (i - r[i] >= 0 && i + r[i] < n && t[i - r[i]] == t[i + r[i]])
{
            r[i] += 1;
        }
        if (i + r[i] > j + r[j]) {
            j = i;
        }
    }
    return r;
}
```

后缀数组

字符串哈希

杂项

基本 `cf` 模板

```cpp
#include <bits/stdc++.h>

using i64 = long long;

void solve() {

}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int t;
    std::cin >> t;

    while(t--) {
        solve();
    }

    return 0;
}
```

快读

```cpp
inline int read() {
    int res = 0, fu = 1;
    char c = getchar();
    while (c < '0' || c > '9') {
        if (c == '-') fu = -1;
        c = getchar();
    }
    while (c >= '0' && c <= '9') {
        res = (res << 1) + (res << 3) + c - '0';
        c = getchar();
    }
    return res;
}
```

交互

交互的本质是二分

CF1807E

```cpp
/**
 * @see https://www.luogu.com.cn/problem/CF1807E
 */
```

```cpp
#include <bits/stdc++.h>

using i64 = long long;
const int N = 2e5 + 3;
int n, a[N], pre[N];

int ask(int l, int r) {
    std::cout << "? " << r - l + 1 << ' ';
    for(int i = l; i <= r; ++i) {
        std::cout << i << ' ';
    }
    std::cout << std::endl;
    std::cout.flush();
    int x;
    std::cin >> x;
    return x;
}

void ret(int x) {
    std::cout << "! " << x << std::endl;
}

bool check(int l, int mid) {
    return ask(l, mid) != pre[mid] - pre[l - 1]; // 询问真实的和计算虚假的是否相等
}

void solve() {
    std::cin >> n;
    for(int i = 1; i <= n; ++i) {
        std::cin >> a[i];
        pre[i] = pre[i - 1] + a[i];
    }

    int l = 1, r = n;
    while(l < r) {
        int mid = (l + r) >> 1;
        if(check(l, mid)) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }
    ret(l);
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);

    int t;
    std::cin >> t;

    while(t--) {
        solve();
```

```
59        }
60
61        return 0;
62    }
```