栈

`#include<stack>`

普通栈

基本操作

- `s.push(val)`：在栈顶添加一个元素。
- `s.pop()`：移除栈顶元素。
- `s.top()`：返回栈顶元素的引用，但不移除它。
- `s.empty()`：检查栈是否为空。
- `s.size()`：返回栈中元素的数量。
- `s.emplace(val)`：在栈顶添加一个原地构造的元素。

无`clear()`函数。

对于`int`、`double`等内置数据类型而言，`push()`和`emplace()`是相同的。

对于自定义数据类型而言，使用`push()`前必须先将要添加的对象构造出来（实例化），而使用`emplace()`既可以填入实例化的对象也可以填入对象的构造参数，并且更节省内存。

```
std::stack<T> s;
// 你能使用
s.push({1, 2});
s.emplace({1, 2});
s.emplace(1, 2);
// 而不能使用
s.push(1, 2);
```

此外，`std::stack`还提供了一些运算符。较为常用的是使用赋值运算符 = 为 stack 赋值，示例：

注意事项

- stack 不提供直接访问栈中元素的方法，只能通过`top()`访问栈顶元素。
- 尝试在空栈上调用`top()`或`pop()`将导致未定义行为。

单调栈

单调栈即满足单调性的栈结构。

过程 & 模板

将一个元素插入单调栈时，为了维护栈的单调性，需要在保证将该元素插入到栈顶后整个栈满足单调性的前提下弹出最少的元素。

不使用STL实现 使用数组进行模拟。

时间复杂度 $O(n)$，空间复杂度 $O(n)$。

```
int num[MAXN];
```

```cpp
int stack[MAXN];
int p;

int main()
{
    /* 输入或其他部分 */

    for (int i = 1; i <= n; i++) {
        while (p && num[stack[p]] <= num[i]) p--;

        /* 统计答案部分 */

        stack[++p] = i;
    }
}
```

- `stack[]`：以数组形式实现单调栈。
- `p`：保存栈顶下标。
- `while (p &&...)`：避免`p`变为负数，导致下标越界。`p = 0`表示栈空。
- `while (...&& num[stack[p]] <= num[i])`：如果栈顶对应元素**小于等于**当前待插入元素，栈顶下标`-1`，表示**弹出栈顶**，直到**栈顶元素大于当前待插入元素**，以保持栈内对应元素**单调下降**。可更改为：
  - `num[stack[p]] < num[i]`：保持栈内对应元素**单调不增**。
  - `num[stack[p]] >= num[i]`：保持栈内对应元素**单调上升**。
  - `num[stack[p]] > num[i]`：保持栈内对应元素**单调不减**。
  - 可更换至其他**cmp函数**，自定义单调栈的单调性。
- `stack[++p] = i`：栈内保存对应元素在`num[]`数组的下标。该代码表示插入该元素对应的下标。

例题

$LuoguP5788$ 【模板】单调栈

> 给出项数为 n 的整数数列 $a_{1..n}$。
> 定义函数 $f(i)$ 代表数列中第 i 个元素之后第一个大于 a_i 的元素的下标，即 $f(i)=min_{i<j\leq n,a_j>a_i}\{j\}$。若不存在，则 $f(i)=0$。
> 试求出 $f(1\ldots n)$。

实现 1

由 **第 i 个元素之后第一个大于 a_i 的元素的下标**，可想到使用 **单调栈** 时 `while` 内对于对应元素大小的判断。

由于**单调栈内对应元素单调不增**，故某个元素弹出时和它进行比较的待插入元素即第一个大于它的元素。

由于其被弹出，后续操作不会再影响其答案。

当所有元素处理完后，因为**没有元素使它们被弹出**，所以留在单调栈内的元素不存在第一个大于其的元素的下标。

参考代码：

```
int n;
int num[MAXN];
int stack[MAXN], p;
int ans[MAXN];
int main()
{
    std::cin >> n;
    p = 0;
    for(int i = 1; i <= n; i++)
    {
        std::cin >> num[i];
        while (p && num[stack[p]] < num[i]) {
            ans[stack[p]] = i;
            p--;
        }
        stack[++p] = i;
    }
    return 0;
}
```

实现 2

面对这样的题面，我们可以把它转化一下：**有 n 个人，每个人都向右看，求每个人看到的第一个人。**

通过观察，对于在 i 前面的人来说，如果比 i 高或者一样高，他会看向 i 后面更高的人。如果比 i 矮，他会看向 i 。所以从 i 到 j （比 i 更高的人）之间的点是无效的，将其排除掉后可见单调性。

故可用单调栈维护。

因为是每个人往右看到的第一个人，所以单调性体现在大于 i 的部分，所以从右往左扫。弹完单调栈内比当前待插入 i 对应元素小或相等的元素后，栈顶对应元素就是比当前元素大的第一个元素，统计对应答案。

因为大小相同取最左边的下标，而单调栈弹掉的元素都是在更右边的，栈顶维护的是更小的下标，故此时单调栈 **单调不增** 或者 **单调递减** 都可以，用代码实现即可。

参考代码：

```
int n;
int num[MAXN];
int stack[maxn], p;
int ans[maxn];
int main()
{
    std::cin >> n;
    for(int i = 1; i <= n; i++)
        std::cin >> num[i];
    for(int i = n; i >= 1; i--) {
        while (p && num[stack[p]] <= num[i]) p--;
        ans[i] = stack[p];
        stack[++p] = i;
    }
    return 0;
}
```

另外，单调栈也可以用于离线解决 **RMQ 问题。**

我们可以把所有询问按右端点排序，然后每次在序列上从左往右扫描到当前询问的右端点处，并把扫描到的元素插入到单调栈中。这样，每次回答询问时，单调栈中存储的值都是位置 $\leq r$ 的、可能成为答案的决策点，并且这些元素满足单调性质。

此时，单调栈上第一个位置 $\geq l$ 的元素就是当前询问的答案，这个过程可以用二分查找实现。使用单调栈解决 RMQ 问题的时间复杂度为 $O(q \log q + q \log n)$，空间复杂度为 $O(n)$。

队列

普通队列

`#include<queue>`

基本操作

- `q.push(val)`：在队尾添加一个元素。
- `q.pop()`：移除队首元素。
- `q.front()`：返回队首元素的引用。
- `q.back()`：返回队尾元素的引用。
- `q.empty()`：检查队列是否为空。
- `q.size()`：返回队列中元素的数量。
- `q.emplace(val)`：在队尾添加一个原地构造的元素。

无`clear()`函数。

对于自定义数据类型而言，使用`push()`前必须先将要添加的对象构造出来（实例化），而使用`emplace()`既可以填入实例化的对象也可以填入对象的构造参数，并且更节省内存。

```
std::queue<T> q;
// 你能使用
q.push({1, 2});
q.emplace({1, 2});
q.emplace(1, 2);
// 而不能使用
q.push(1, 2);
```

此外，`std::queue`还提供了一些运算符。较为常用的是使用赋值运算符 = 为 queue 赋值，示例：

常用于BFS、SPFA等算法中。

双端队列

`#include<deque>`

基本操作

- `q.push_back(val)`：在队尾添加一个元素。
- `q.push_front(val)`：在队首添加一个元素。
- `q.emplace_back(val)`：在队尾添加一个原地构造的元素。
- `q.emplace_front(val)`：在队首添加一个原地构造元素。
- `q.pop_back()`：移除队尾元素。
- `q.pop_front()`：移除队首元素。
- `q.front()`：返回队首元素的引用。
- `q.back()`：返回队尾元素的引用。
- `q.empty()`：检查队列是否为空。
- `q.size()`：返回队列中元素的数量。
- `q.erase(it)`：删除指定位置或指定范围的元素（传入一个或两个迭代器）。
- `q.insert(it, val)`：在指定位置添加一个元素（传入迭代器和数值）
- `q.emplace(it, val)`：在指定位置添加一个原地构造的元素（传入迭代器和数值）。
- `q.clear()`：清除双端队列内所有元素。

有`clear()`函数。

此外，deque 还提供了一些运算符。其中较为常用的有：

- 使用赋值运算符 `=` 为 `std::deque` 赋值，类似 `std::queue`。
- 使用 `[]` 访问元素，类似 `std::vector`。

优先队列

`#include<queue>`

和`std::queue`不同的地方在于我们可以自定义其中数据的优先级, 让优先级高的排在队列前面,优先出队。

参数

模板申明带3个参数：
`std::priority_queue<Type, Container, Functional>`
其中`Type`为数据类型， `Container`为保存数据的容器， `Functional`为元素比较方式。
`Container`必须是用数组实现的容器，比如`std::vector`,`std::deque`等等，但不能用 `list`。

原始为大根堆，可重载运算符或添加比较方式变为小根堆。

重载运算符方式：

```cpp
// 变为小根堆
struct Node {
    int val;
    bool operator < (const Node &b) const {
        return val > b.val;
    }
}
std::priority_queue<Node> q;
```

添加比较方式：

```cpp
// 升序队列，小顶堆
std::priority_queue<int, std::vector<int>, std::greater<int>> q;
```

```cpp
// 降序队列，大顶堆
std::priority_queue<int, std::vector<int>, std::less<int>> q;
```

greater和less是std实现的两个仿函数（就是使一个类的使用看上去像一个函数。其实现就是类中实现一个`operator()`，这个类就有了类似函数的行为，就是一个仿函数类了）

```cpp
struct cmp {
    bool operator()(int a, int b) { // 默认是less
        return a > b;
    }
};
// 自写仿函数
std::priority_queue<int, std::vector<int>, cmp> q;
```

基本操作

- `q.push(val)`：在队尾添加一个元素。
- `q.pop()`：移除队首元素。
- `q.top()`：返回队首元素的引用。
- `q.empty()`：检查队列是否为空。
- `q.size()`：返回队列中元素的数量。
- `q.erase(it)`：删除指定位置的元素（传入迭代器）
- `q.emplace(val)`：在队尾添加一个原地构造的元素。

无`clear()`函数。

此外，`std::priority_queue` 还提供了一些运算符。其中较为常用的有使用赋值运算符 = 为 `std::priority_queue` 赋值，类似 `std::queue`。

单调队列

要求的是每连续的 k 个数中的最大（最小）值，很明显，当一个数进入所要 "寻找" 最大值的范围中时，若这个数比其前面（先进队）的数要大，显然，前面的数会比这个数先出队且不再可能是最大值。

也就是说——当满足以上条件时，可将前面的数 "弹出"，再将该数真正 `push` 进队尾。

这就相当于维护了一个递减的队列，符合单调队列的定义，减少了重复的比较次数，不仅如此，由于维护出的队伍是查询范围内的且是递减的，队头必定是该查询区域内的最大（最小）值，因此输出时只需输出队头即可。

显而易见的是，在这样的算法中，每个数只要进队与出队各一次，因此时间复杂度被降到了 $O(n)$。

而由于查询区间长度是固定的，超出查询空间的值再大也不能输出，因此还需要比较队头和当前下标的差值。如果差值大于查询的区间长度，弹出越界的队头。

过程

不使用STL实现 使用数组模拟。

时间复杂度 $O(n)$，空间复杂度 $O(n)$。

```
int num[MAXN];
int q[MAXN];
int l = 1, r = 0;

int main()
{
    /* 输入或其他部分 */

    for (int i = 1; i <= n; i++) {

        while (l <= r && q[l] < i - k + 1) l++;

        while (l <= r && num[q[r]] <= num[i]) r--;

        q[++r] = i;

        /* 统计答案部分 */

    }
}
```

- `q[]`：以数组形式实现单调队列。
- `l, r`：保存队首和队尾 有效区间在`[l, r]`。
- `while (l <= r &&...)`：避免`r < l - 1`产生，$r = l - 1$时表示队列空，不能再弹出元素。
- `while (...&& q[l] < i - k + 1)`：如果队首存储的下标已经超出当前所统计的范围 k，则此答案无用，需要弹出。
- `while (...&& num[q[r]] <= num[i])`：如果队尾对应元素**小于等于**当前待插入元素，`l++`，表示**弹出队尾**，直到**队尾对应元素大于当前待插入元素**，以保持队列内对应元素**单调下降**。可更改为：
  - `num[q[r]] < num[i]`：保持队列内对应元素**单调不增**。
  - `num[q[r]] >= num[i]`：保持队列内对应元素**单调上升**。
  - `num[q[r]] > num[i]`：保持队列内对应元素**单调不减**。
  - 可更换至其他**cmp函数**，自定义单调单列的单调性。
- `q[++r] = i`：队列内保存对应元素在`num[]`数组的下标。该代码表示插入该元素对应的下标。

例题

$LuoguP2698\ \ FlowerpotS$

> 给出 N 滴水的坐标，y 表示水滴的高度，x 表示它下落到 x 轴的位置。每滴水以每秒 1 个单位长度的速度下落。你需要把花盆放在 x 轴上的某个位置，使得从被花盆接着的第 1 滴水开始，到被花盆接着的最后 1 滴水结束，之间的时间差至少为 D。 我们认为，只要水滴落到 x 轴上，与花盆的边沿对齐，就认为被接住。给出 N 滴水的坐标和 D 的大小，请算出最小的花盆的宽度 W 。

实现

将所有水滴按照 x 坐标排序之后，题意可以转化为求一个 x 坐标差最小的区间使得这个区间内 y 坐标的最大值和最小值之差至少为 D 。我们发现这道题与一个区间内的最大值最小值有关，但是这道题区间的大小不确定，而且区间大小本身还是我们要求的答案。

我们可以使用一个递增，一个递减两个单调队列在 R 不断后移时维护 $[L,R]$ 内的最大值和最小值，不过此时我们发现，如果 L 固定，那么 $[L,R]$ 内的最大值只会越来越大，最小值只会越来越小，所以设 $f(R)=max[L,R]-min[L,R]$ ，则 $f(R)$ 是个关于 R 的递增函数，故 $f(R)\geq D => f(r)\geq D, R\leq r\leq N$。这说明对于每个固定的 L，向右第一个满足条件的 R 就是最优答案。 所以我们整体求解的过程就是，先固定 L ，从前往后移动 R ，使用两个单调队列维护 $[L,R]$ 的最值。当找到了第一个满足条件的 R ，就更新答案并将 L 也向后移动。随着 L 向后移动，两个单调队列都需及时弹出队头。这样，直到 R 移到最后，每个元素依然是各进出队列一次，保证了 $O(n)$ 的时间复杂度。

参考代码：

```
typedef long long ll;
int mxq[N], mnq[N];
int D, ans, n, hx, rx, hn, rn;

struct la {
    int x, y;
    bool operator<(const la &y) const {
        return x < y.x;
    }
}a[N];

int main()
{
    scanf("%d%d", &n, &D);
    for (int i = 1; i <= n; ++i) {
        scanf("%d%d", &a[i].x, &a[i].y);
    }
    std::sort(a + 1, a + n + 1);
    hx = hn = 1;
    rx = rn = 0;
    ans = 2e9;
    int L = 1;
    for (int i = 1; i <= n; ++i) {
        while (hx <= rx && a[mxq[rx]].y < a[i].y) rx--;
        mxq[++rx] = i;
        while (hn <= rn && a[mnq[rn]].y > a[i].y) rn--;
        mnq[++rn] = i;
        while (L <= i && a[mxq[hx]].y - a[mnq[hn]].y >= D) {
            ans = min(ans, a[i].x - a[L].x);
```

```
            L++;
            while (hx <= rx && mxq[hx] < L) hx++;
            while (hn <= rn && mnq[hn] < L) hn++;
        }
    }
    if (ans < 2e9)
        printf("%d\n", ans);
    else
        puts("-1");
    return 0;
}
```

链表

链表是一种用于存储数据的数据结构，通过如链条一般的指针来连接元素。它的特点是插入与删除数据十分方便，但寻找与读取数据的表现欠佳。

复杂度

能方便地删除、插入数据，操作次数是 $O(1)$。

寻找、读取数据的效率不如数组高，在随机访问数据中的操作次数是 $O(n)$ 。

单向链表

```
// 定义链表
struct Node {
    int value;
    Node *next;
};

// 初始化链表
Node *initialize() {
    Node *head = new Node;
    head->next = NULL;
    return head;
}

// 插入数据（在Node *p的后面）
void insertNode(int val, Node *p) {
    Node *node = new Node;
    node->value = val;
    node->next = p->next;
    p->next = node;
}

// 删除数据
void deleteNode(Node *p) {
    // 从链表中删除 p 时，将 p 的下一个结点 p->next 的值覆盖给 p 即可.
    // 与此同时更新 p 的下下个结点。
    p->value = p->next->value;
    Node *t = p->next;
    p->next = p->next->next;
```

```
    delete t;
}
```

双向链表

```
// 定义链表
struct Node {
    int value;
    Node *prev, *next;
}
Node *head, *tail;

// 初始化链表
void initialize() {
    head = new Node;
    tail = new Node;
    head->next = tail;
    head->prev = nullptr;
    tail->next = nullptr;
    tail->prev = head;
}

// 插入数据（在Node *p的后面）
void insertNode(int val, Node *p) {
    Node *node = new Node;
    node->value = val;
    p->next->prev = node;
    node->next = p->next;
    p->next = node;
    node->prev = p;
}

// 删除数据
void deleteNode(Node *p) {
    p->next->prev = p->prev;
    p->prev->next = p->next;
    delete p;
}

// 清除链表
void clear() {
    while (head != tail) {
        // 不断将head的下一个节点赋给head 此时原来的head节点变成head->prev
        // 删除head->prev节点即可
        head = head->next;
        delete head->prev;
    }
    // 最后删掉tail即可
    delete tail;
}
```

哈希表

一种以「key-value」形式存储数据的数据结构。

普通哈希

要让键值对应到内存中的位置，就要为键值计算索引，也就是计算这个数据应该放到哪里。这个根据键值计算索引的函数就叫做哈希函数，也称散列函数。能为 `key` 计算索引之后，我们就可以知道每个键值对应的值 `value` 应该放在哪里了。假设我们用数组 `a` 存放数据，哈希函数是 f，那键值对 $(key, value)$ 就应该放在 $a[f(key)]$ 上。不论键值是什么类型，范围有多大，$f(key)$ 都是在可接受范围内的整数，可以作为数组的下标。

当键值的范围比较小的时候，可以直接把键值作为数组的下标，但当键值的范围比较大，比如以 10^9 范围内的整数作为键值的时候，就需要用到哈希表。一般把键值模一个较大的质数作为索引，也就是取 $f(x) = x(mod M)$ 作为哈希函数。

另一种比较常见的情况是 `key` 为字符串的情况，由于不支持以字符串作为数组下标，并且将字符串转化成数字存储也可以避免多次进行字符串比较。所以在 OI 中，一般不直接把字符串作为键值，而是先算出字符串的哈希值，再把其哈希值作为键值插入到哈希表里。关于字符串的哈希值，我们一般采用进制的思想，将字符串想象成一个 127 进制的数。那么，对于每一个长度为 n 的字符串 s，就有：

$$x = s_0 \cdot 127^0 + s_1 \cdot 127^1 + s_2 \cdot 127^2 + \ldots + s_n \cdot 127^n$$

我们可以将得到的 x 对 2^{64}（即 `unsigned long long` 的最大值）取模。这样 `unsigned long long` 的自然溢出就等价于取模操作了。可以使操作更加方便。

这种方法虽然简单，但并不是完美的。可以构造数据使这种方法发生冲突（即两个字符串的 x 对 2^{64} 取模后的结果相同）。

实际上，常常会出现两个不同的键值，他们用哈希函数计算出来的索引是相同的。这时候就需要一些方法来处理冲突。

双哈希

选取两个大质数 a, b。当且仅当两个键值的哈希值对 a 和对 b 取模都相等时，我们才认为这两个键值相等。这样可以大大降低哈希冲突的概率。常用于字符串。

开散列法

开散列法是在每个存放数据的地方开一个链表，如果有多个键值索引到同一个地方，只用把他们都放到那个位置的链表里就行了。

查询的时候需要把对应位置的链表整个扫一遍，对其中的每个数据比较其键值与查询的键值是否一致。如果索引的范围是 $[1, M]$，哈希表的大小为 N，那么一次插入/查询需要进行期望 $O(\frac{N}{M})$ 次比较。

闭散列法

闭散列方法把所有记录直接存储在散列表中，如果发生冲突则根据某种方式继续进行探查。

比如线性探查法：如果在 d 处发生冲突，就依次检查 `d + 1`，`d + 2`.....

map

`#include<map>`

一种关联容器，用于存储键值对（key-value pairs）。

`std::map` 容器中的元素按照键的顺序自动排序，适合需要快速查找和有序数据的场景。

基本操作

- `mp.begin()`：返回头部迭代器。
- `mp.end()`：返回末尾迭代器。
- `mp.count()`：返回指定元素出现的次数。
- `mp.empty()`：检查容器是否为空。
- `mp.erase(it / key)`：删除迭代器it或者键key对应的元素 返回被删除元素的下一个元素的迭代器。
- `mp.insert(pair<T, T> / std::map<T, T>::value_type(key, val) / {key, val})`：向容器中插入元素。
- `mp.emplace(key, val)`：向容器中插入原地构造的元素。
- `mp.find(val)`：寻找值为val的元素，返回该元素的迭代器，否则返回`map.end()`。
- `mp.lower_bound(val)`：返回键值 $\geq val$ 的第一个元素的迭代器
- `mp.upper_bound(val)`：返回键值 $> val$ 给定元素的第一个元素的迭代器
- `mp.clear()`：清除容器内所有元素

`std::map`基于红黑树结构实现。红黑树具有自动排序的功能，因此它使得map也具有按键（key）排序的功能。

对`std::map`增、删、改、查的复杂度都为 $O(logn)$，复杂度即红黑树的高度。

unordered_map

`#include<unordered_map>`

`std::unordered_map` 不保证元素的排序，但通常提供更快的查找速度。

`std::unordered_map` 是一个关联容器，使用哈希表来存储元素，这使得它在查找、插入和删除操作中具有平均常数时间复杂度。

基本操作

- `ump.begin()`：返回头部迭代器。
- `ump.end()`：返回末尾迭代器。
- `ump.count()`：返回指定元素出现的次数。
- `ump.empty()`：检查容器是否为空。
- `ump.erase(it / key)`：删除迭代器it或者键key对应的元素 返回被删除元素的下一个元素的迭代器。
- `ump.insert(pair<T, T> / std::map<T, T>::value_type(key, val) / {key, val})`：向容器中插入元素。
- `ump.emplace(key, val)`：向容器中插入原地构造的元素。
- `ump.emplace_hint(it, key, val)`：向容器中的指定位置插入原地构造的元素。
- `ump.find(val)`：寻找值为val的元素，返回该元素的迭代器，否则返回`map.end()`。

- `ump.lower_bound(val)`：返回键值 $\geq val$ 的第一个元素的迭代器
- `ump.upper_bound(val)`：返回键值 $> val$ 给定元素的第一个元素的迭代器
- `ump.merge(ump2)`：将ump2合并到ump中，ump2清空。键值冲突时，ump2保留冲突的键值对，ump内键对应的值无影响。
- `ump.clear()`：清除容器内所有元素

`std::unordered map`基于哈希表实现。

对于插入、删除和查找操作的平均时间复杂度为 $O(1)$，最坏情况下的时间复杂度为 $O(n)$。

注意事项

- `std::unordered_map` 不保证元素的顺序，因此元素的迭代顺序可能在不同的运行中不同。
- 哈希表的性能依赖于良好的哈希函数，以避免过多的哈希冲突。
- 与 `std::map` 相比，`std::unordered_map` 在元素数量较少时可能占用更多的内存。
- 仍可能构造出相应数据卡掉`std::unordered_map`，改进方法如下。

改进方法

```
// 无万能头时需加入头文件<chrono>
#include <chrono>
struct custom_hash {
    static uint64_t splitmix64(uint64_t x) {
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }
    size_t operator () (uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
std::chrono::steady_clock::now().time_since_epoch().count(); // 时间戳
        return splitmix64(x + FIXED_RANDOM);
    }
};

// 通过伪类自定义哈希函数 然后定义unordered_map
std::unordered_map<long long, int, custom_hash> safe_map;
gp_hash_table<long long, int, custom_hash> safe_hash_table;
```

并查集

并查集是一种用于管理元素所属集合的数据结构，实现为一个森林，其中每棵树表示一个集合，树中的节点表示对应集合中的元素。

并查集能在一张无向图中维护节点之间的连通性，擅长动态维护许多具有传递性的关系。

基础操作

初始化

（一定要记得）

```
int fa[MAXN];

for (int i = 1; i <= n; i++) {
    fa[i] = i;
}
```

查询

```
int find(x) {
    if (x == fa[x]) return x;
    return find(fa[x]);
}
```

路径压缩

由于路径压缩单次合并可能造成大量修改，有时路径压缩并不适合使用。

例如，在可持久化并查集、线段树分治 + 并查集中，一般使用只启发式合并的并查集。

```
void prework() {
    int find(x) {
    if (x == fa[x]) return x;
    return fa[x] = find(fa[x]);
}
```

合并

```
void merge(int x, int y) {
    int fx = find(x), fy = find(y);
    if (fx != fy) fa[fx] = fy;
}
```

启发式合并（按秩合并）

如果我们将一棵点数与深度都较小的集合树连接到一棵更大的集合树下，显然相比于另一种连接方案，接下来执行查找操作的用时更小（也会带来更优的最坏时间复杂度）。

当然，我们不总能遇到恰好如上所述的集合——点数与深度都更小。鉴于点数与深度这两个特征都很容易维护，我们常常从中择一，作为估价函数。而无论选择哪一个，时间复杂度都为 $O(\alpha(N))$ 。

```
int fa[MAXN], rank[MAXN]; // rank[i]表示i子树的深度

void prework() {
    for (int i = 1; i <= n; i++) {
        fa[i] = i;
        rank[i] = 0; // 初始化为1也可以
```

```
    }

void merge(int x, int y) {
    int fx = find(x), fy = find(y);
    if (fx == fy) return;
    if (rank[fx] < rank[fy]) std::swap(fx, fy); // 确保小树合并到大树里
    fa[fy] = fx; // 合并
    if (rank[fx] == rank[fy]) rank[fx]++; // 处理两秩相同的情况 此时fx的秩会加一
}
```

在算法竞赛的实际代码中，即便不使用启发式合并，代码也往往能够在规定时间内完成任务。不使用启发式合并、只使用路径压缩的最坏时间复杂度是 $O(\log N)$ ，在平均情况下，时间复杂度依然是 $O(\alpha(N))$ 。

如果只使用启发式合并，而不使用路径压缩，时间复杂度为 $O(\log N)$ 。

只含查询与删除的并查集模板

```
int fa[MAXN], rank[MAXN]; // rank表示树深 也可替换为树大小

void prework() {
    for (int i = 1; i <= n; i++) {
        fa[i] = i;
        rank[i] = 0; // 初始化为1也可以
    }
}

int find(int x) {
    if (x == fa[x]) return x;
    return fa[x] = find(fa[x]);
}

void merge(int x, int y) {
    int fx = find(x), fy = find(y);
    if (fx == fy) return;
    if (rank[fx] < rank[fy]) std::swap(fx, fy);
    fa[fy] = fx;
    if (rank[fx] == rank[fy]) rank[fx]++;
}
```

删除

常规的并查集并不能实现这一操作，因此，我们需要另一种写法。

常规并查集的初始化的方法是把每个节点的父节点初始化成自己，`fa[i] = i` 。此时，i 点有了两重身份：一个节点的名字，集合祖宗的名字。

删除操作的关键就是只改变这一个点的祖宗，而不改变这个点所在集合的祖宗。

```
struct dsu {
    std::vector<int> fa, size;

    explicit dsu(int SIZE) : fa(SIZE * 2), size(SIZE * 2, 1), index(SIZE * 2) {
        iota(fa.begin(), fa.begin() + SIZE, SIZE);
        iota(fa.begin() + SIZE, fa.end(), SIZE);
```

```
    }

    void erase(int x) {
        int fx = find(x);
        --size[fx];
        fa[x] = x; // 重新将该点连向自己
    }
};
```

这样做，其他原来跟2同一祖宗的节点的祖宗并没有改变，他们还是连在**节点原来所连的虚点**上，从而实现了删除操作。

移动

```
void dsu::move(int x, int y) {
    int fx = find(x), fy = find(y);
    if (fx == fy) return;
    fa[x] = fy;
    --size[fx], ++size[fy];
}
```

带删除和移动的总模板

```
struct dsu {
    std::vector<int> fa, size, sum;

    explicit dsu(int SIZE) : fa(SIZE * 2), size(SIZE * 2, 1), sum(SIZE * 2) {
        // size 与 sum 的前半段其实没有使用，只是为了让下标计算更简单
        iota(fa.begin(), fa.begin() + SIZE, SIZE);
        iota(fa.begin() + SIZE, fa.end(), SIZE);
        iota(sum.begin() + SIZE, sum.end(), 0);
    }

    int find(int x) {
        if (x == fa[x]) return x;
        return fa[x] = find(fa[x]);
    }

    void merge(int x, int y) {
        int fx = find(x), fy = find(y);
        if (fx == fy) return;
        if (size[fx] < size[fy]) std::swap(fx, fy);
        fa[fy] = fx;
        size[fx] += size[fy];
        sum[fx] += sum[fy];
    }

    void erase(int x) {
        int fx = find(fx);
        --size[fx]; sum[fx] -= x;
        fa[x] = x;
    }

    void move(int x, int y) {
```

```
        int fx = find(x), fy = find(y);
        if (fx == fy) return;
        fa[x] = fy;
        --size[fx], ++size[fy];
        sum[fx] -= x, sum[fy] += x;
    }
};
```

“边带权”的并查集

并查集实际上是由若干棵树构成的森林，我们可以在树中的每条边上记录一个权值，即维护一个数组`d`，用`d[x]`保存节点x到父节点fa[x]之间的边权。路径压缩后，每个访问过的节点都会直接指向树根，如果我们同时更新这些节点的`d`值，就可以利用路径压缩的过程来统计每个节点到树根之间的路径上的一些信息。

```
int fa[MAXN], d[MAXN];

int find(int x) {
    if (x == fa[x]) return x;
    int root = find(fa[x]);
    d[x] += d[fa[x]]; // 根据实际问题需要来修改
    return fa[x] = root;
}

void merge(int x, int y) {
    int fx = find(x), fy = find(y);
    /* 更新边权部分 根据实际问题需要来添加*/
    fa[fx] = fy; size[fy] += size[fx];
}
```

“扩展域”的并查集

在某些问题中，“传递关系”不止一种，并且这些“传递关系”能够互相导出。此时可以使用“扩展域”或者“边带权”的并查集来解决。

$POJ1733$ Parity Game

> Alice 和 Bob 在玩一个游戏：他写一个由 0 和 1 组成的序列。
>
> Alice 选其中的一段（比如第 3 位到第 5 位），问他这段里面有奇数个 1 还是偶数个 1。Bob 回答你的问题，然后 Alice 继续问。
>
> Alice 要检查 Bob 的答案，指出在 Bob 的第几个回答一定有问题。

实现

如果我们用`sum`数组表示序列的前缀和，那么在每个回答中：

1. S[l ~ r] 有偶数个1，等价于 sum[l - 1] 与 sum[r] 的奇偶性相同；
2. S[l ~ r] 有奇数个1，等价于 sum[l - 1] 与 sum[r] 的奇偶性不同。

除了使用明显的“边带权”的并查集，本题还可以使用“扩展域”的并查集。

把每个变量 x 拆成两个节点 x_{odd} 和 x_{even} ，其中 x_{odd} 表示 sum[x] 是奇数，x_{even} 表示 sum[x] 是偶数。我们也经常把这两个节点称为 x 的“奇数域”和“偶数域”。

本题需要离散化，令离散化后 $l-1$ 和 r 的值分别是 x 和 y 。

1. 若 $ans=0$ 则合并 x_{odd} 和 y_{odd} ， x_{even} 和 y_{even} 。
2. 若 $ans=1$ 则合并 x_{odd} 和 y_{even} ， x_{even} 和 y_{odd} 。

上述合并同时还维护了关系的传递性。试想，在处理完 $(x,y,0)$ 和 $(y,z,1)$ 两个回答后， (x,z) 的关系也就已知了。这种做法就相当于在无向图上维护节点之间的连通情况，只是拓展了多个域来应对多种传递关系。

处理每个问题前还需要检查是否存在矛盾。

参考代码：

```
int fa[MAXN << 1]; // 两倍的扩展域 需要开两倍空间

int find(int x) {
    if (fa[x] == x) return x;
    return fa[x] = find(fa[x]);
}

int main() {
    read_discrete();
    for (int i = 1; i <= 2 * n; i++) {
        // 扩展域 所以1 ~ 2 * n
        fa[i] = i;
    }
    for (int i = 1; i <= m; i++) {
        int x = get(query[i].l - 1), y = get(query[i].r);

        int x_odd = x, y_odd = y;
        int x_even = x + n, y_even = y + n;
        int fxodd = find(x_odd), fyodd = find(y_odd);
        int fxeven = find(x_even), fyeven = find(y_even);

        if (query[i].ans == 0) {
            // 因为合并时两个域都合并，均存在传递关系，故判断其中一个域即可
            if (fxodd == fyeven) {
                std::cout << i - 1 << '\n';
                return 0;
            }
            fa[fxodd] = fyodd;
            fa[fxeven] = fyeven;
        } else {
            if (fxodd == fyodd) {
                std::cout << i - 1 << '\n';
                return 0;
            }
            fa[fxodd] = fyeven;
            fa[fxeven] = fyodd;
        }
    }
```

```
    std::cout << m << '\n';
}
```

$LuoguP2024$ [NOI2001] 食物链

> 动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B，B 吃 C，C 吃 A。
>
> 现有 N 个动物，以 $1\sim N$ 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。
>
> 有人用两种说法对这 N 个动物所构成的食物链关系进行描述：
>
> - 第一种说法是 1 X Y，表示 X 和 Y 是同类。
> - 第二种说法是2 X Y，表示 X 吃 Y。
>
> 此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
>
> - 当前的话与前面的某些真的话冲突，就是假话；
> - 当前的话中 X 或 Y 比 N 大，就是假话；
> - 当前的话表示 X 吃 X，就是假话。
>
> 你的任务是根据给定的 N 和 K 句话，输出假话的总数。

由题目可知可以把每个动物拆成三个节点，同类域 x_{self} 、捕食域 x_{eat} 、天敌域 x_{enemy}，分别表示 x 的同类、 x 的捕食、 x 的天敌。

X 吃 Y 还会得到一个隐藏信息，Y 的捕食为 X 的天敌。

参考代码：

```
int n,k;
int fa[MAXN * 3];
int ans;
bool judge(int x, int y) {
    if(x > n || y > n) return 1;
    return 0;
}

int find(int x) {
    if(x == fa[x]) return x;
    return fa[x] = find(fa[x]);
}

int main() {
    std::cin >> n >> k;
    for(int i = 1; i <= 3 * n; i++) fa[i] = i;
    for(int i = 1; i <= k; i++) {
        int opt, x, y;
        std::cin >> opt >> x >> y;
        if(judge(x,y) || (opt == 2 && x == y)) {
            ans++;
            continue;
        }
        int fx_self = find(x), fx_hunt = find(x + n), fx_enemy=find(x + 2 * n);
        int fy_self = find(y), fy_hunt = find(y + n), fy_enemy=find(y + 2 * n);
```

```
        if(opt == 1) {
            if(fx_hunt == fy_self || fx_self == fy_hunt) ans++;
            else fa[fx_self] = fy_self, fa[fx_hunt] = fy_hunt, fa[fx_enemy] =
fy_enemy;
        } else if(opt == 2) {
            if(fx_self == fy_self || fx_self == fy_hunt) ans++;
            else fa[fx_self] = fy_enemy, fa[fx_hunt]=fy_self, fa[fx_enemy] =
fy_hunt;
        }
    }
    std::cout << ans;
    return 0;
}
```

本题还可以用边带权（权值为0，1，2）解决。

边带权和扩展域通常可以相互转化。

可持久化并查集

```
struct Tree {
    int ls, rs;
    int fa, dep;
}t[N << 6];
int idx, root[N << 6];

int build(int l, int r) {
    int p = ++idx;
    if (l == r) {
        t[p].fa = l;
        return p;
    }
    int mid = l + r >> 1;
    t[p].ls = build(l, mid);
    t[p].rs = build(mid + 1, r);
    return p;
}
int merge(int now, int l, int r, int x, int v) {
    int p = ++idx;
    t[p] = t[now];
    if (l == r) {
        t[p].fa = v;
        return p;
    }
    int mid = l + r >> 1;
    if (x <= mid) t[p].ls = merge(t[now].ls, l, mid, x, v);
    else t[p].rs = merge(t[now].rs, mid + 1, r, x, v);
    return p;
}
int update(int now, int l, int r, int x) {
    // 需要在修改深度时创建新副本
    // 不然会修改到历史版本的dep，导致按秩合并的时间复杂度退化。
    int p = ++idx;
```

```
    t[p] = t[now];
    if (l == r) {
        t[p].dep++;
        return p;
    }
    int mid = l + r >> 1;
    if (x <= mid) t[p].ls = update(t[now].ls, l, mid, x);
    else t[p].rs = update(t[now].rs, mid + 1, r, x);
    return p;
}
int query(int p, int l, int r, int x) {
    if (l == r) return p;
    int mid = l + r >> 1;
    if (x <= mid) return query(t[p].ls, l, mid, x);
    else return query(t[p].rs, mid + 1, r, x);
}
int find(int p, int x) {
    int now = query(p, 1, n, x);
    if (t[now].fa == x) return now;
    return find(p, t[now].fa);
}

int main() {
    std::ios::sync_with_stdio(0);
    std::cin.tie(0), std::cout.tie(0);
    std::cin >> n >> m;
    root[0] = build(1, n);
    for (int i = 1; i <= m; i++) {
        int opt, x, y;
        std::cin >> opt;
        if (opt == 1) {
            std::cin >> x >> y;
            int fx = find(root[i - 1], x), fy = find(root[i - 1], y);
            if (t[fx].fa != t[fy].fa) {
                if (t[fx].dep > t[fy].dep) std::swap(fx, fy);
                root[i] = merge(root[i - 1], 1, n, t[fx].fa, t[fy].fa);
                if (t[fx].dep == t[fy].dep) root[i] = update(root[i], 1, n,
t[fy].fa);
            } else root[i] = root[i - 1];
        } else if (opt == 2) {
            std::cin >> x;
            root[i] = root[x];
        } else if (opt == 3) {
            std::cin >> x >> y;
            root[i] = root[i - 1];
            int fx = find(root[i], x), fy = find(root[i], y);
            if (t[fx].fa == t[fy].fa) std::cout << "1\n";
            else std::cout << "0\n";
        }
    }
    return 0;
}
```

堆

堆是一棵树，其每个节点都有一个键值，且每个节点的键值都大于等于 / 小于等于其父亲的键值。

每个节点的键值都大于等于其父亲键值的堆叫做小根堆，否则叫做大根堆。

`std::priority_queue` 是一个大根堆。

支持的操作

堆主要支持的操作有：**插入一个数**、**查询最值**、**删除最值**、**合并两个堆**、**减小一个元素的值**。

分类

操作\数据结构	二叉堆	配对堆	左偏树	二项堆	斐波那契堆
插入	$O(\log N)$	$O(1)$	$O(\log N)$	$O(\log N)$	$O(1)$
寻找最值	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
删除最值	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$
合并	$O(N)$	$O(1)$	$O(\log N)$	$O(\log N)$	$O(1)$
减小一个元素的值	$O(\log N)$	最大 $O(2^{2\sqrt{\log \log N}})$ 均摊 $O(\log N)$	$O(\log N)$	$O(\log N)$	$O(1)$
支持可持久化	√	×	√	√	×

习惯上，不加限定提到「堆」时往往都指二叉堆。

pb_ds库的堆

pb_ds 库封装了很多数据结构，其中就有堆。

```
#include <ext/pb_ds/priority_queue.hpp>
__gnu_pbds::priority_queue<T, Compare, Tag> q;
```

模板形参

- T: 储存的元素类型
- Compare: 提供严格的弱序比较类型
- Tag: 是 __gnu_pbds 提供的不同的六种堆，Tag 参数默认是 pairing_heap_tag 六种分别是：
  - `priority_queue_tag`：**基本堆结构**
  - `pairing_heap_tag`：**配对堆** 官方文档认为在非原生元素（如自定义结构体/std :: string/pair）中，配对堆表现最好

- `binary_heap_tag`：**二叉堆** 官方文档认为在原生元素中二叉堆表现最好，不过笔者测试的表现并没有那么好
- `binomial_heap_tag`：**二项堆** 二项堆在合并操作的表现要优于二叉堆，但是其取堆顶元素操作的复杂度比二叉堆高
- `rc_binomial_heap_tag`：**冗余计数二项堆**
- `thin_heap_tag`：除了合并的复杂度都和 **斐波那契堆**一样的一个堆

注意：

`binary_heap_tag` 的实际运行速度极慢，不如 `std` 和**手写堆**

pb_ds库的堆常数极大，建议手写。

构造方式

```
__gnu_pbds ::priority_queue<int>
__gnu_pbds::priority_queue<int, greater<int>>
__gnu_pbds ::priority_queue<int, greater<int>, pairing_heap_tag>
__gnu_pbds ::priority_queue<int>::point_iterator id; // 点类型迭代器
// 在 modify 和 push 的时候都会返回一个 point_iterator，下文会详细的讲使用方法
id = q.push(1);
```

成员函数

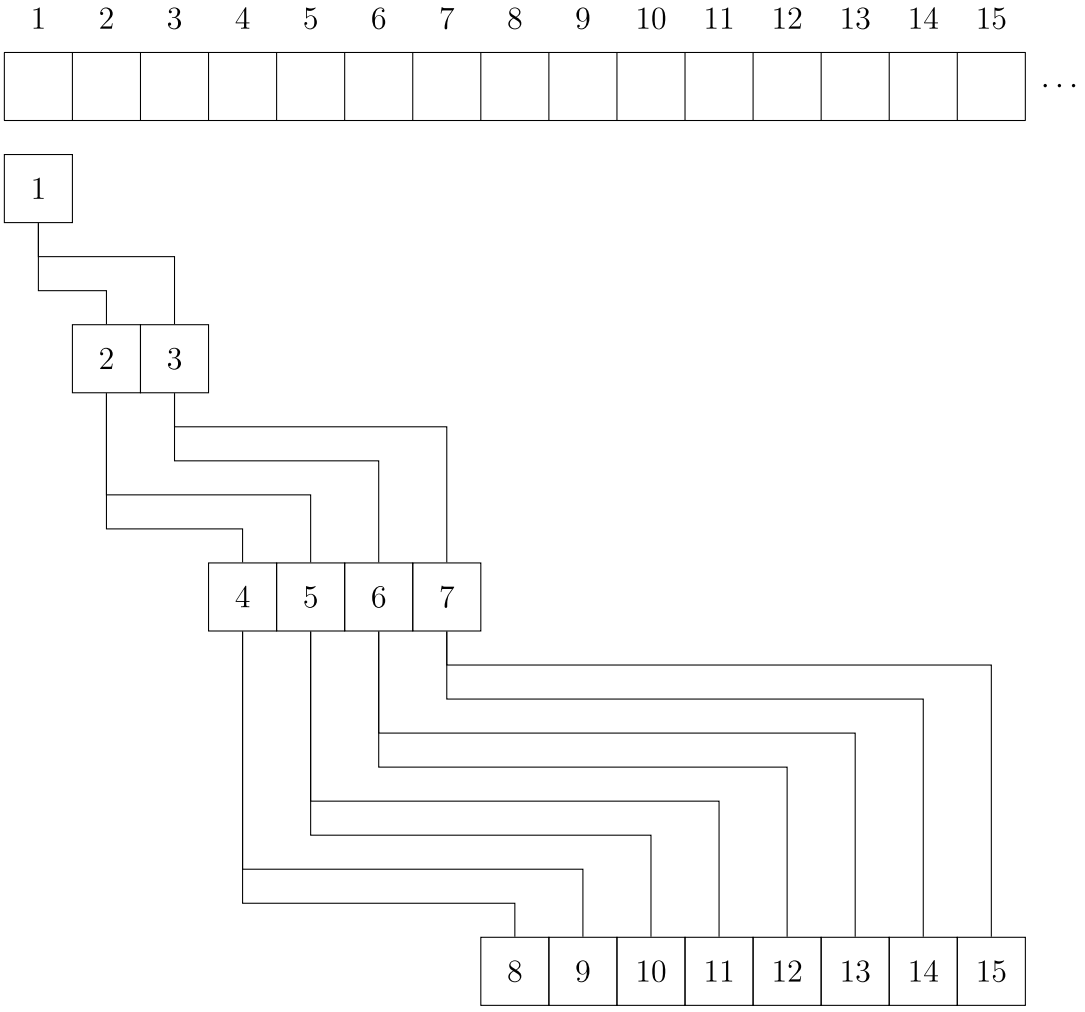
- `q.push()`：向堆中压入一个元素，返回该元素位置的迭代器。
- `q.pop()`：将堆顶元素弹出。
- `q.top()`：返回堆顶元素。
- `q.size()`：返回元素个数。
- `q.empty()`：返回是否非空。
- `q.modify(point_iterator, const key)`：把迭代器位置的 key 修改为传入的 key，并对底层储存结构进行排序。
- `q.erase(point_iterator)`：把迭代器位置的键值从堆中擦除。
- `q.join(__gnu_pbds :: priority_queue &other)`：把 other 合并到 *this 并把 other 清空。

时间复杂度

	push	pop	modify	erase	join
pairing_heap_tag（配对堆）	$O(1)$	均摊 $O(\log N)$ 最坏 $O(N)$	均摊 $O(\log N)$ 最坏 $O(N)$	均摊 $O(\log N)$ 最坏 $O(N)$	$O(1)$
binary_heap_tag（二叉堆）	均摊 $O(\log N)$ 最坏 $O(N)$	均摊 $O(\log N)$ 最坏 $O(N)$	$O(N)$	$O(N)$	$O(N)$
binomial_heap_tag（二项堆）	均摊 $O(1)$ 最坏 $O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$
rc_binomial_heap_tag（冗余计数二项堆）	$O(1)$	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(\log N)$

	push	pop	modify	erase	join
thin_heap_tag（斐波那契堆）	$O(1)$	均摊 $O(\log N)$ 最坏 $O(N)$	均摊 $O(1)$ 最坏 $O(\log N)$	均摊 $O(\log N)$ 最坏 $O(N)$	$O(N)$

二叉堆



堆性质很弱，二叉堆并不是唯一的。因此建堆能达到 $O(N)$ 级别 。

大根堆

```
int heap[MAXN], n;

// 插入带有权值val的新节点在堆尾 然后向上调整 直到满足堆性质
// 时间复杂度 O(logN)
void up(int p) {
    while (p > 1) {
        if (heap[p] > heap[p / 2]) {
            std::swap(heap[p], heap[p / 2]);
            p /= 2;
        } else break;
    }
}
void insert(int val) {
```

```
    heap[++n] = val;
    up(n);
}

// 返回堆顶 为最大值
// 时间复杂度 O(1)
int getTop() {
    return heap[1];
}

// 移除堆顶
// 考虑插入操作的逆过程，设法将根结点移到最后一个结点，然后直接删掉。
// 新的根结点可能不满足堆性质 需要向下调整
// 时间复杂度 O(log N)
void down(int p) {
    int s = p << 1;
    while (s <= n) {
        if (s < n && heap[s] < heap[s + 1]) s++;
        if (heap[s] > heap[p]) {
            std::swap(heap[s], heap[p]);
            p = s, s = p << 1;
        } else break;
    }
}
void extract() {
    heap[1] = heap[n--];
    down(1);
}

// 删除下标p位置的节点
// 此时既有可能需要向下调整 也有可能需要向上调整 分别检查和处理
// 时间复杂度 O(logN)
void remove(int k) {
    heap[k] = heap[n--];
    up(k), down(k);
}

// 修改某个节点的权值 然后分别向上向下调整
// 时间复杂度 O(logN)
void modify(int k, int v) {
    heap[k] = v;
    up(k), down(k);
}

// 建堆方法1：使用 decreasekey（即，向上调整）
// 从根开始，按 BFS 序进行。
// 对于第 k 层的结点，向上调整的复杂度为 O(k) 而不是 O(logN)。
// 时间复杂度 O(log1 + log2 + ··· + logN) = O(logN)
void build_heap_1() {
    // 数据已在heap[]中
    for (int i = 1; i <= n; i++) up(i);
}

// 建堆方法2：使用向下调整
// 从叶子开始，逐个向下调整
// 注意到向下调整的复杂度为 O(logN - k)
```

```
// 另外注意到叶节点其实无需调整，因此可从序列约 n / 2 的位置开始调整
// 可减少部分常数但不影响复杂度。
// 时间复杂度 O(N)
void build_heap_2() {
    // 数据已在heap[]中
    for (int i = n; i >= 1; i--) down(i);
}
```

小根堆

```
// 根据大根堆改变大小判断条件即可得到
// 时间复杂度和大根堆相同
int heap[MAXN], n;

void up(int p) {
    while (p > 1) {
        if (heap[p] < heap[p / 2]) {
            std::swap(heap[p], heap[p / 2]);
            p /= 2;
        } else break;
    }
}
void insert(int val) {
    heap[++n] = val;
    up(n);
}

int getTop() {
    return heap[1];
}

void down(int p) {
    int s = p << 1;
    while (s <= n) {
        if (s < n && heap[s] > heap[s + 1]) s++;
        if (heap[s] < heap[p]) {
            std::swap(heap[s], heap[p]);
            p = s, s = p << 1;
        } else break;
    }
}
void extract() {
    heap[1] = heap[n--];
    down(1);
}

void remove(int k) {
    heap[k] = heap[n--];
    up(k), down(k);
}

void modify(int k, int v) {
    heap[k] = v;
    up(k), down(k);
}
```

```
void build_heap_1() {
    // 数据已在heap[]中
    for (int i = 1; i <= n; i++) up(i);
}

void build_heap_2() {
    // 数据已在heap[]中
    for (int i = n; i >= 1; i--) down(i);
}
```

由于pb_ds库中的二叉堆的实际运行速度极慢，请使用 `std` 空间的 `priority_queue` 或者 **手写。**

对顶堆（大根堆小根堆的应用）

动态维护一个序列上第 k 大的数，k 值可能会发生变化。

对于此类问题，我们可以使用 对顶堆 这一技巧予以解决（可以避免写权值线段树或 BST 带来的繁琐）。

对顶堆由一个大根堆与一个小根堆组成，小根堆维护大值即前 k 大的值（包含第 k 个），大根堆维护小值即比第 k 大数小的其他数。

这两个堆构成的数据结构支持以下操作：

- 维护：当小根堆的大小小于 k 时，不断将大根堆堆顶元素取出并插入小根堆，直到小根堆的大小等于 k；当小根堆的大小大于 k 时，不断将小根堆堆顶元素取出并插入大根堆，直到小根堆的大小等于 k；
- 插入元素：若插入的元素大于等于小根堆堆顶元素，则将其插入小根堆，否则将其插入大根堆，然后维护对顶堆；
- 查询第 k 大元素：小根堆堆顶元素即为所求；
- 删除第 k 大元素：删除小根堆堆顶元素，然后维护对顶堆；
- k 值 $+1/-1$：根据新的 k 值直接维护对顶堆。

显然，查询第 k 大元素的时间复杂度是 $O(1)$ 的。由于插入、删除或调整 k 值后，小根堆的大小与期望的 k 值最多相差 1，故每次维护最多只需对大根堆与小根堆中的元素进行一次调整，因此，这些操作的时间复杂度都是 $O(logN)$ 的。

通常使用 `std::priority_queue` 实现。

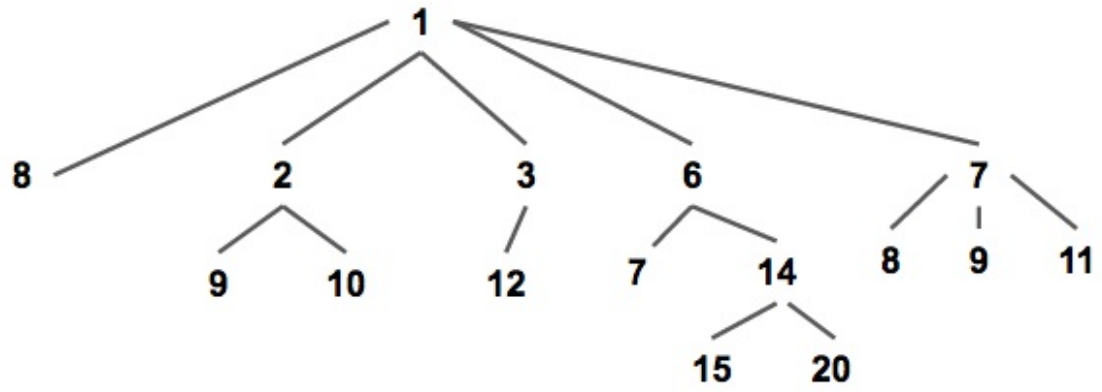
```
![pairingheap1](C:\Users\fzy\Desktop\板子\pairingheap1.jpg)int main() {
    int t, x;
    std::cin >> t;
    while (t--) {
        // 大根堆，维护前一半元素（存小值）
        priority_queue<int, vector<int>, less<int> > a;
        // 小根堆，维护后一半元素（存大值）
        priority_queue<int, vector<int>, greater<int> > b;
        while (std::cin >> x && x) {
            // 若为查询并删除操作，输出并删除大根堆堆顶元素
            // 因为这题要求输出中位数中较小者（偶数个数字会存在两个中位数候选）
            // 这个和上面的第k大讲解有稍许出入，但如果理解了上面的，这个稍微变通
下便可理清
            if (x == -1) {
                std::cout << a.top();
                a.pop();
```

```
            }
            // 若为插入操作，根据大根堆堆顶的元素值，选择合适的堆进行插入
            else {
                if (a.empty() || x <= a.top()) a.push(x);
                else b.push(x);
            }
            // 对对顶堆进行调整
            if (a.size() > (a.size() + b.size() + 1) / 2) {
                b.push(a.top());
                a.pop();
            } else if (a.size() < (a.size() + b.size() + 1) / 2) {
                a.push(b.top());
                b.pop();
            }
        }
    }
    return 0;
}
```

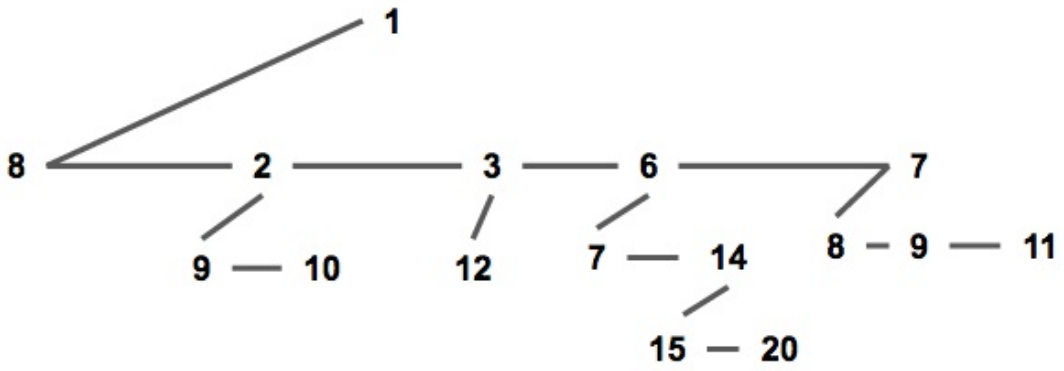
配对堆

配对堆有速度快和结构简单的优势，但由于其为基于势能分析的均摊复杂度，无法可持久化。

配对堆是一棵满足堆性质的带权多叉树，即每个节点的权值都小于或等于他的所有儿子（以小根堆为例）。



通常我们使用儿子 - 兄弟表示法储存一个配对堆，一个节点的所有儿子节点形成一个单向链表。每个节点储存第一个儿子的指针，即链表的头节点；和他的右兄弟的指针。




```
struct Node {
    T v;  // T为权值类型
    int id;
    Node *child, *sibling;
    // child 指向该节点第一个儿子，sibling 指向该节点的下一个兄弟。
    // 若该节点没有儿子/下个兄弟则指针指向 nullptr。
    Node* father;
    // 父指针，若该节点为根节点则指向空节点 nullptr
};
```

从定义可以发现，和其他常见的堆结构相比，配对堆不维护任何额外的树大小，深度，排名等信息（二叉堆也不维护额外信息，但它是通过维持一个严格的完全二叉树结构来保证操作的复杂度），且任何一个满足堆性质的树都是一个合法的配对堆。

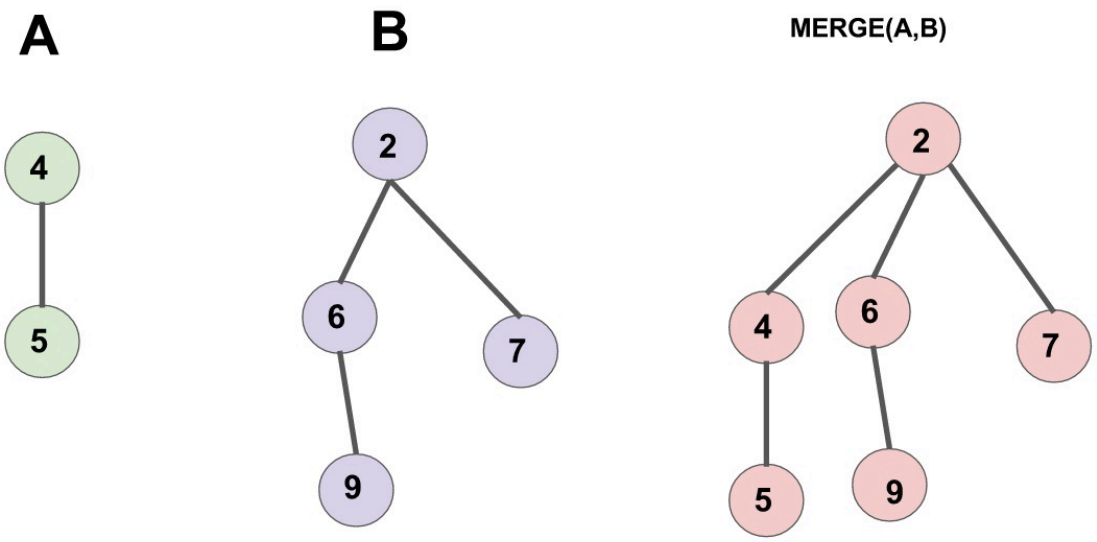
配对堆通过一套精心设计的操作顺序来保证它的总复杂度。

查询最小值

即根节点

```
int getTop() {
    return root->v;
}
```

合并



需要注意的是，一个节点的儿子链表是按插入时间排序的，即最右边的节点最早成为父节点的儿子，最左边的节点最近成为父节点的儿子。

```
Node* meld(Node* x, Node* y) {
    // 若有一个为空则直接返回另一个
    if (x == nullptr) return y;
    if (y == nullptr) return x;
    // swap后x为权值小的堆，y为权值大的堆
    // 若要成大根堆则变为 x->v < y->v
    if (x->v > y->v) std::swap(x, y);
    // 维护父指针
    if (x->child != nullptr) {
        x->child->father = y;
    }
    y->father = x;
    // 将y设为x的儿子
    y->sibling = x->child;
    x->child = y;
    return x;  // 新的根节点为 x
}
```

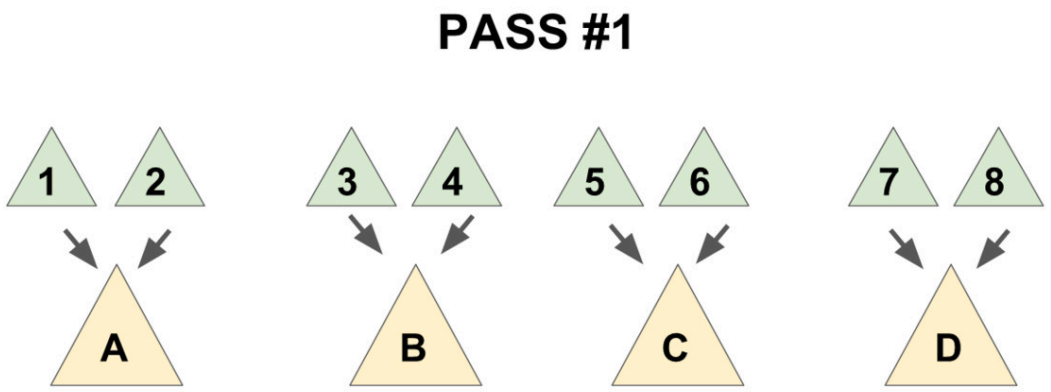
插入

插把新元素视为一个新的配对堆和原堆合并即可。

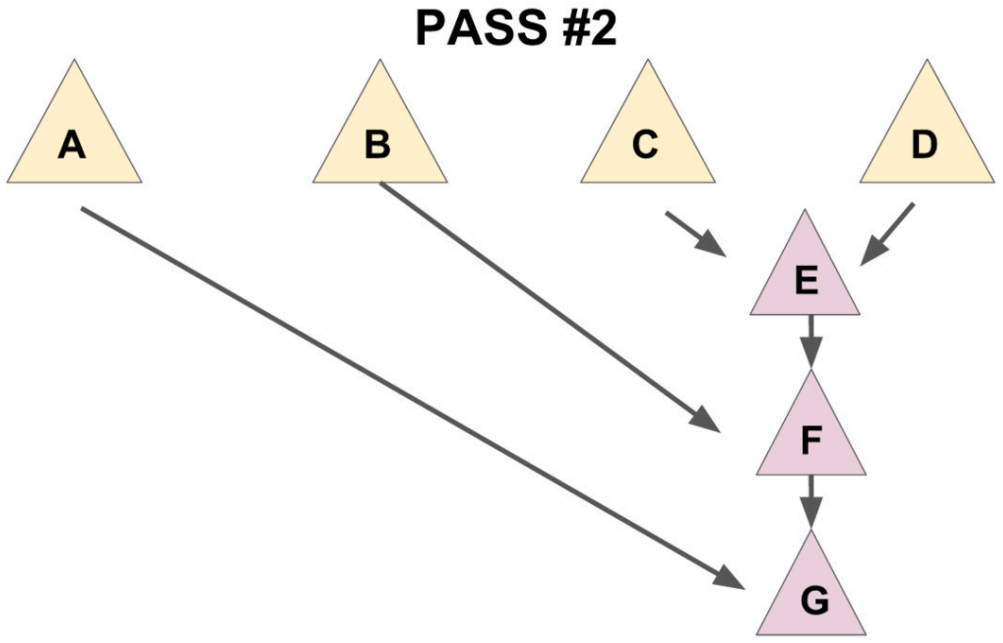
删除最值

为了保证总的均摊复杂度，需要使用一个「两步走」的合并方法：

- 把儿子们两两配成一对，用 `meld` 操作把被配成同一对的两个儿子合并到一起



- 将新产生的堆 **从右往左**（即老的儿子到新的儿子的方向）挨个合并在一起




```
Node* merges(Node* x) {
    if (x == nullptr) return nullptr;  // 如果该树为空
    x->father = nullptr;
    if (x->sibling == nullptr) // 如果该树没有下一个兄弟，就不需要合并了，
return。
    Node* y = x->sibling;                // y 为 x 的下一个兄弟
    Node* c = y->sibling;                // c 是再下一个兄弟
    y->father = nullptr;                 // 维护父指针
    x->sibling = y->sibling = nullptr;   // 拆散
    return meld(merges(c), meld(x, y));  // 核心部分
}
Node* delete_min(Node* x) {
    Node* t = merges(x->child);
    delete x;  // 如果需要内存回收
    return t;
}
```

减小一个元素的值

```
// root为堆的根，x为要操作的节点，v为新的权值，调用时需保证 v <= x->v
// 返回值为新的根节点
Node *decrease_key(Node *root, Node *x, LL v) {
    x->v = v;                  // 更新权值
    if (x == root) return x;  // 如果 x 为根，则直接返回
    // 把x从fa的子节点中剖出去，这里要分x的位置讨论一下。
    if (x->father->child == x) {
        x->father->child = x->sibling;
    } else {
        x->father->sibling = x->sibling;
    }
    if (x->sibling != nullptr) {
        x->sibling->father = x->father;
    }
    x->sibling = nullptr;
    x->father = nullptr;
    return meld(root, x);  // 重新合并 x 和根节点
}
```

配对堆总模板

```
struct Node {
    T v;  // T为权值类型
    int id;
    Node *child, *sibling;
    // child 指向该节点第一个儿子，sibling 指向该节点的下一个兄弟。
    // 若该节点没有儿子/下个兄弟则指针指向 nullptr。
    Node* father;
    // 父指针，若该节点为根节点则指向空节点 nullptr
};
// 查询最值
int getTop() {
    return root->v;
}
// 合并
Node* meld(Node* x, Node* y) {
    // 若有一个为空则直接返回另一个
    if (x == nullptr) return y;
    if (y == nullptr) return x;
    // swap后x为权值小的堆，y为权值大的堆
    // 添加(x->v > y->v && x > y)会使得在删除时删除优先输入的
    // 若要成大根堆则变为 x->v < y->v
    if (x->v > y->v || (x->v > y->v && x > y)) std::swap(x, y);
    // 维护父指针
    if (x->child != nullptr) {
        x->child->father = y;
    }
    y->father = x;
    // 将y设为x的儿子
    y->sibling = x->child;
    x->child = y;
    return x;  // 新的根节点为 x
```

```
}
// 删除最值
Node* merges(Node* x) {
    if (x == nullptr) return nullptr;  // 如果该树为空
    x->father = nullptr;
    if (x->sibling == nullptr) return x;// 如果该树没有下一个兄弟，就不需要合并
了，return。
    Node* y = x->sibling;                // y 为 x 的下一个兄弟
    Node* c = y->sibling;                // c 是再下一个兄弟
    y->father = nullptr;                 // 维护父指针
    x->sibling = y->sibling = nullptr;   // 拆散
    return meld(merges(c), meld(x, y));  // 核心部分
}
Node* delete_min(Node* x) {
    Node* t = merges(x->child);
    return t;
}
// 更新一个元素的值
Node* decrease_key(Node *root, Node *x, LL v) {
    x->v = v;                   // 更新权值
    // 因为此处是小根堆 且减小堆顶最小值不会影响堆的结构，则如果 x 为根，则直接返
回
    if (x == root) return x;
    // 把x从fa的子节点中剖出去，这里要分x的位置讨论一下。
    if (x->father->child == x) {
        x->father->child = x->sibling;
    } else {
        x->father->sibling = x->sibling;
    }
    if (x->sibling != nullptr) {
        x->sibling->father = x->father;
    }
    x->sibling = nullptr;
    x->father = nullptr;
    return meld(root, x);  // 重新合并 x 和根节点
}

//插入
void insert(int v, int id) {
    Node* p = new Node;
    p->v = v;
    p->id = id
    meld(root, p); // p与root合并
}
```

配对堆无decrease_key总模板（不需要维护father）（小根堆）

```
struct Node {
    T v;  // T为权值类型
    Node *child, *sibling;
};
Node* meld(Node* x, Node* y) {
    // 若有一个为空则直接返回另一个
    if (x == nullptr) return y;
    if (y == nullptr) return x;
```

```
    if (x->v > y->v) std::swap(x, y);
    // 将y设为x的儿子
    y->sibling = x->child;
    x->child = y;
    return x; // 新的根节点
}
Node* merges(Node* x) {
    if (x == nullptr || x->sibling == nullptr)
        return x;  // 如果该树为空或他没有下一个兄弟，就不需要合并了，return。
    Node* y = x->sibling;                // y 为 x 的下一个兄弟
    Node* c = y->sibling;                // c 是再下一个兄弟
    x->sibling = y->sibling = nullptr;   // 拆散
    return meld(merges(c), meld(x, y));  // 核心部分
}
Node* delete_min(Node* x) {
    Node* t = merges(x->child);
    return t;
}
```

配对堆无decrease_key总模板（不需要维护father）（大根堆）

```
struct Node {
    T v;  // T为权值类型
    Node *child, *sibling;
};
Node* meld(Node* x, Node* y) {
    // 若有一个为空则直接返回另一个
    if (x == nullptr) return y;
    if (y == nullptr) return x;
    if (x->v < y->v) std::swap(x, y);
    // 将y设为x的儿子
    y->sibling = x->child;
    x->child = y;
    return x; // 新的根节点
}
Node* merges(Node* x) {
    if (x == nullptr || x->sibling == nullptr)
        return x;  // 如果该树为空或他没有下一个兄弟，就不需要合并了，return。
    Node* y = x->sibling;                // y 为 x 的下一个兄弟
    Node* c = y->sibling;                // c 是再下一个兄弟
    x->sibling = y->sibling = nullptr;   // 拆散
    return meld(merges(c), meld(x, y));  // 核心部分
}
Node* delete_min(Node* x) {
    Node* t = merges(x->child);
    return t;
}
```

pb_ds库的配对堆模板

```
// 为了更好的阅读体验，定义宏如下
#define pair_heap __gnu_pbds::priority_queue<int, std::greater<int>,
pairing_heap_tag>
```

```
pair_heap q1;  // 大根堆，配对堆
pair_heap q2;
pair_heap::point_iterator id;  // 一个迭代器

int main() {
    id = q1.push(1);
    // 堆中元素 :  [1];
    for (int i = 2; i <= 5; i++) q1.push(i);
    // 堆中元素 :  [1, 2, 3, 4, 5];
    std::cout << q1.top() << '\n';
    // 输出结果 : 5;
    q1.pop();
    // 堆中元素 : [1, 2, 3, 4];
    id = q1.push(10);
    // 堆中元素 : [1, 2, 3, 4, 10];
    q1.modify(id, 1);
    // 堆中元素 :  [1, 1, 2, 3, 4];
    std::cout << q1.top() << '\n';
    // 输出结果 : 4;
    q1.pop();
    // 堆中元素 : [1, 1, 2, 3];
    id = q1.push(7);
    // 堆中元素 : [1, 1, 2, 3, 7];
    q1.erase(id);
    // 堆中元素 : [1, 1, 2, 3];
    q2.push(1), q2.push(3), q2.push(5);
    // q1中元素 : [1, 1, 2, 3], q2中元素 : [1, 3, 5];
    q2.join(q1);
    // q1中无元素, q2中元素 : [1, 1, 1, 2, 3, 3, 5];
}
```

例题

$LuoguP3377$【模板】左偏树/可并堆

> 如题，一开始有 n 个小根堆，每个堆包含且仅包含一个数。接下来需要支持两种操作：
>
> `1 x y`：将第 x 个数和第 y 个数所在的小根堆合并（若第 x 或第 y 个数已经被删除或第 x 和第 y 个数在同一个堆内，则无视此操作）。
>
> `2 x`：输出第 x 个数所在的堆最小数，并将这个最小数删除（若有多个最小数，优先删除先输入的；若第 x 个数已经被删除，则输出 -1 并无视删除操作）。

参考代码：

```
#include <bits/stdc++.h>
constexpr int N = 1e5 + 5;
struct Node {
    int v;
    int id;
    Node *child, *sibling;
}node[N];
bool b[N];
Node* heap[N];
Node* meld(Node* x, Node* y) {
    if (x == nullptr) return y;
```

```
    if (y == nullptr) return x;
    if (x->v > y->v || (x->v == y-> v && x->id > y->id)) std::swap(x, y);
    y->sibling = x->child;
    x->child = y;
    return x;
}
Node* merges(Node* x) {
    if (x == nullptr || x->sibling == nullptr)
        return x;
    Node* y = x->sibling;
    Node* c = y->sibling;
    x->sibling = y->sibling = nullptr;
    return meld(merges(c), meld(x, y));
}
Node* delete_min(Node* x) {
    Node* t = merges(x->child);
    return t;
}

int fa[N];
int find(int x) {
    if (x == fa[x]) return x;
    return fa[x] = find(fa[x]);
}
bool uni(int x, int y) {
    if (x == y) return false;
    fa[y] = x;
    return true;
}

int main()
{
    int n, m;
    std::cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        fa[i] = i;
    }
    for (int i = 1; i <= n; i++) {
        int v;
        std::cin >> v;
        node[i] = {v, i, nullptr, nullptr};
        heap[i] = &node[i];
    }
    for (int i = 1; i <= m; i++) {
        int opt, x, y;
        std::cin >> opt;
        if (opt == 1) {
            std::cin >> x >> y;
            int fx = find(x), fy = find(y);
            if (b[x] || b[y] || !uni(fx, fy)) continue;
            heap[fx] = meld(heap[fx], heap[fy]);
        } else if (opt == 2) {
            std::cin >> x;
            int fx = find(x);
            if (b[x]) {
                std::cout << "-1\n";
```

```
                continue;
            }
            std::cout << heap[fx]->v << '\n';
            b[heap[fx]->id] = true;
            heap[fx] = delete_min(heap[fx]);
        }
    }
    return 0;
}
```

左偏树

左偏树 与 配对堆 一样，是一种 **可并堆**，具有堆的性质，并且可以快速合并。

对于一棵二叉树，我们定义 **外节点** 为左儿子或右儿子为空的节点，定义**一个外节点的** $dist$ **为** 1，一个不是外节点的**节点** $dist$ **为其到子树中最近的外节点的距离加一。空节点的** $dist$ **为** 0 。

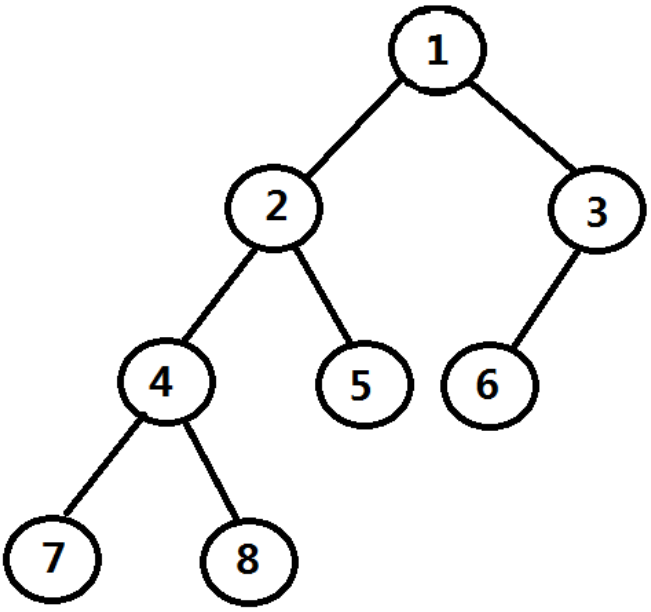
换一种说法。设当前结点为 p，则 $dist_p$ **为子树中深度最小的空结点的到** p **的距离。**

注：$dist$ 又称 npl 或 $s-value$

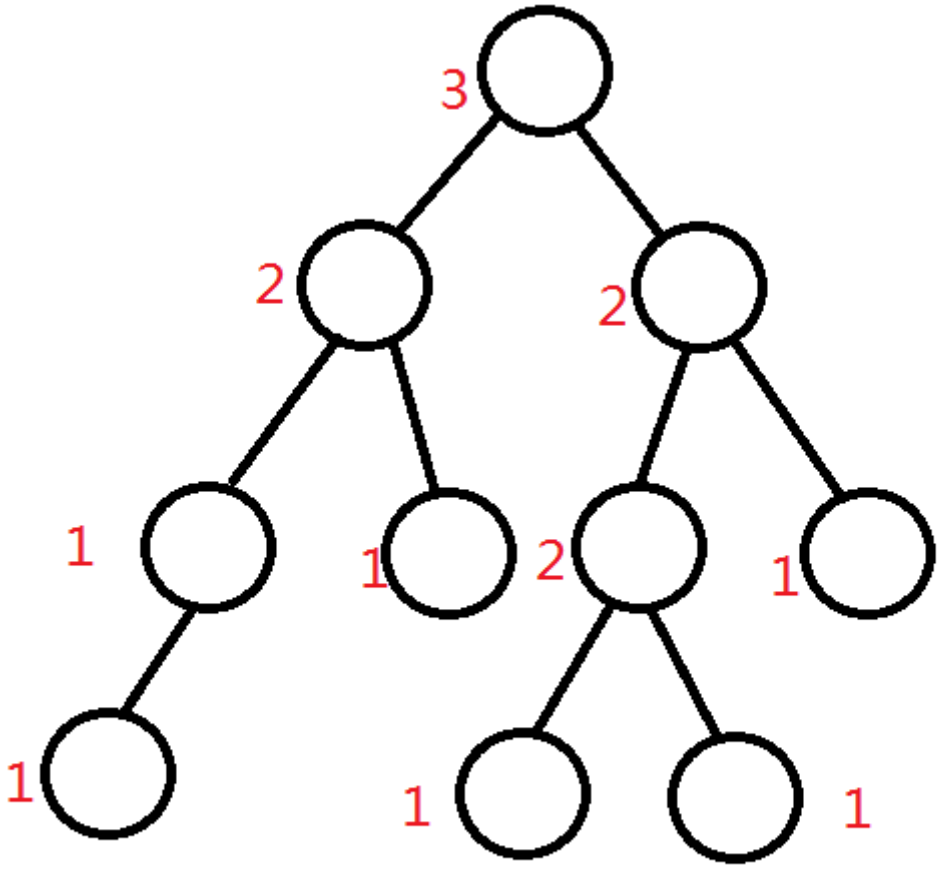
定义

每个结点的左儿子 $dist$ **都不小于它右儿子的** $dist$

以下两棵都是左偏树：



洛谷@qkhm



洛谷@qkhm

由左偏性质还可以延伸出一条很重要的性质： **一个结点的 $dist$ 能且只能被它两个子结点中 $dist$ 小的更新**，而右儿子的 $dist$ 永远比左儿子小，所以可以**直接使用右儿子的 $dist$ 来更新当前结点的 $**dist$ 。

得到公式 $dist_p = dist_r + 1$

初始化

```
struct Heap{
    #define l(p) tr[p].lt
    #define r(p) tr[p].rt
    #define val(p) tr[p].val
    #define dst(p) tr[p].dist
    #define fa(p) tr[p].fa
    int fa, lt, rt, val, dist;
}t[MAXN];
```

查询

类似并查集，只需要使用一个 fa 数组来存储每个结点所在子树的根结点就好了。

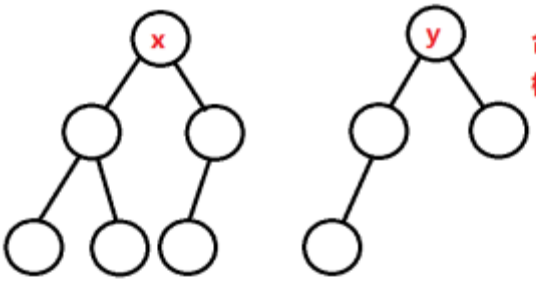
```
int find(int x) {
    return fa(x) == x ? x : fa(x) = find(fa(x));
}
```

合并

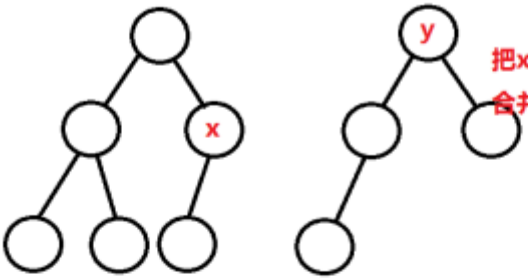
左偏树的合并代码分为 3 部分（以小根堆为例）。

1. 把根结点权值小的堆（根为 x）作为基础，它的所有左儿子不做出改变，而把另一个堆（根为 y）和 x 的右儿子继续递归合并。
2. 递归到 x, y 中的一个为空结点时结束，返回不为空的结点编号
3. 更新 $dist$ 值,且把不满足左偏性质的左右儿子互换。

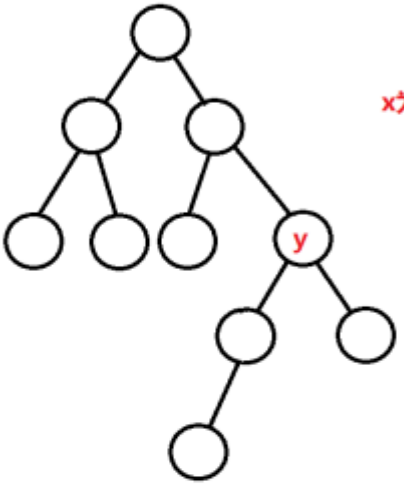
图例：



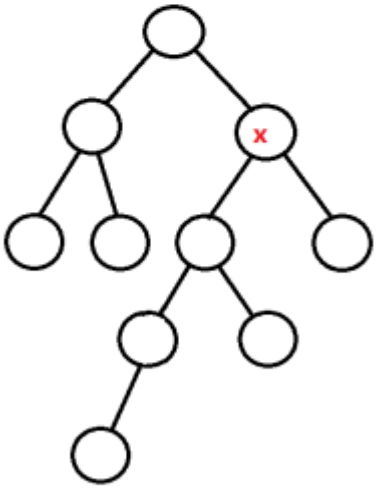
合并以x，y为根的两个堆



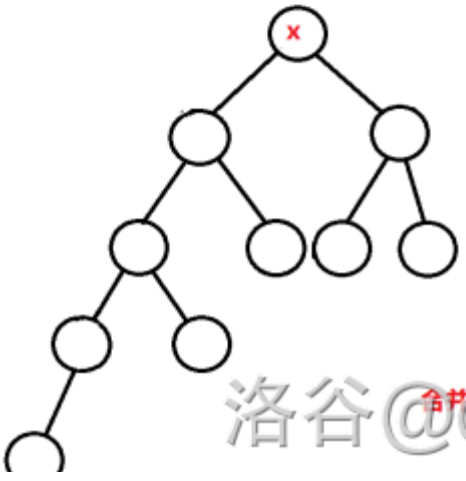
把x的右子树与y合并



x为空，当前位置为y



维护左偏树性质，交换两个儿子



不满足，继续交换

合并成功（撒花）

为方便查询（减小复杂度），还需要添加路径压缩。

```
int merge(int x, int y) {
    if(!x || !y) return x | y;
    if(val(x) > val(y)) std::swap(x, y);
    r(x) = merge(r(x), y);
    if(dst(l(x)) < dst(r(x))) std::swap(l(x), r(x));
    dst(x) = dst(r(x)) + 1;
    fa(l(x)) = fa(r(x)) = fa(x) = x;
    return x;
}
```

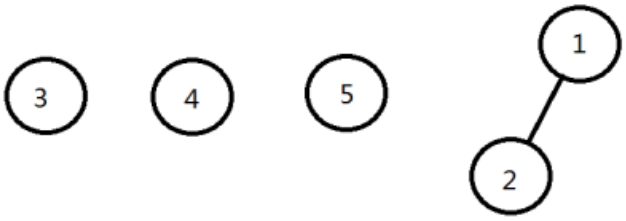
插入

把问题转换成把一个堆和一个只有单个结点的堆合并即可。

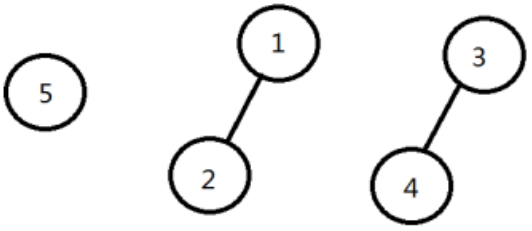
建堆

和建普通堆一样，一般通过队列实现，每次把队首的两个堆取出来，合并后扔到队尾，直到只剩下一个堆。

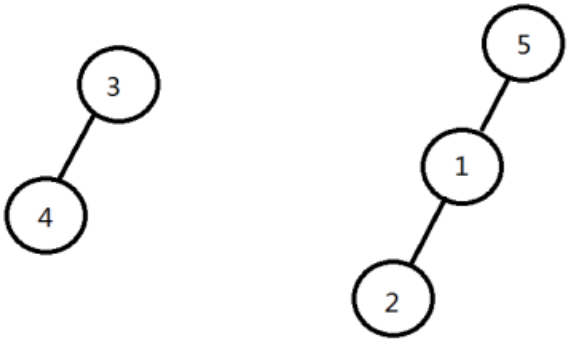
有序排队



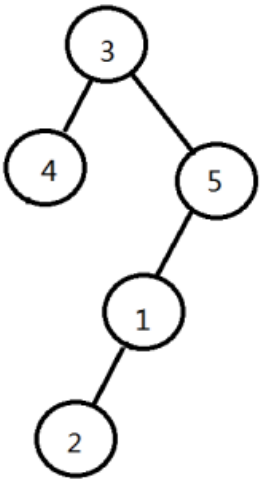
1,2合并，放到队尾



3,4合并，放到队尾



5和1,2合并，继续放到队尾



3，4和5,1,2合并，建堆完成（撒花）

时间复杂度 $O(N)$

```
int build(){
    queue <int> q;
    for(int i = 1;i <= n;++ i) q.push(i);
    while(q.size() > 1){
        int x = q.front();q.pop();
        int y = q.front();q.pop();
        q.push(merge(x, y));//取出前两个合并
    }
    return q.front();//返回合并出的根结点
}
```

弹出堆顶

把根结点的左右儿子合并，再维护一下并查集。

```
int pop(int x) {
    val(x) = -1;
    fa(l(x)) = l(x), fa(r(x)) = r(x);
    fa(x) = merge(l(x), r(x));
    return 0;
}
```

这里把根结点的权值定为 `-1` 来表示它被删掉了，同时维护左右子树的 `fa`。

只改变了根结点附近的 fa，还有一些结点仍然认被删结点 x 为老大，所以让 x 认新合并出的根结点为老大，这样后面的结点自然也就认新根结点为老大了（老大的老大还是老大）。避免了并查集产生混乱。

总模板（小根堆）

```
struct Heap{
    #define l(p) t[p].lt
    #define r(p) t[p].rt
    #define val(p) t[p].val
    #define dst(p) t[p].dist
    #define fa(p) t[p].fa
    int fa, lt, rt, val, dist;
}t[N];

int find(int x) {
    return fa(x) == x ? x : fa(x) = find(fa(x));
}

int merge(int x, int y) {
    if(!x || !y) return x | y;
    //  若要大根堆则改成val(x) < val(y)
    if(val(x) > val(y)) std::swap(x, y);
    r(x) = merge(r(x), y);
    if(dst(l(x)) < dst(r(x))) std::swap(l(x), r(x));
    dst(x) = dst(r(x)) + 1;
    fa(l(x)) = fa(r(x)) = fa(x) = x;
    return x;
}
```

```
int pop(int x) {
    val(x) = -1;
    fa(l(x)) = l(x), fa(r(x)) = r(x);
    fa(x) = merge(l(x), r(x));
    return 0;
}

int build(){
    std::queue<int> q;
    for(int i = 1; i <= n; i++) q.push(i);
    while(q.size() > 1){
        int x = q.front(); q.pop();
        int y = q.front(); q.pop();
        q.push(merge(x, y));//取出前两个合并
    }
    return q.front();//返回合并出的根结点
}
```

总模板（大根堆）

```
struct Heap{
    #define l(p) t[p].lt
    #define r(p) t[p].rt
    #define val(p) t[p].val
    #define dst(p) t[p].dist
    #define fa(p) t[p].fa
    int fa, lt, rt, val, dist;
}t[N];

int find(int x) {
    return fa(x) == x ? x : fa(x) = find(fa(x));
}

int merge(int x, int y) {
    if(!x || !y) return x | y;
    if(val(x) < val(y)) std::swap(x, y);
    r(x) = merge(r(x), y);
    if(dst(l(x)) < dst(r(x))) std::swap(l(x), r(x));
    dst(x) = dst(r(x)) + 1;
    fa(l(x)) = fa(r(x)) = fa(x) = x;
    return x;
}

int pop(int x) {
    val(x) = -1;
    fa(l(x)) = l(x), fa(r(x)) = r(x);
    fa(x) = merge(l(x), r(x));
    return 0;
}

int build(){
    std::queue<int> q;
    for(int i = 1; i <= n; i++) q.push(i);
    while(q.size() > 1){
        int x = q.front(); q.pop();
```

```
        int y = q.front(); q.pop();
        q.push(merge(x, y));//取出前两个合并
    }
    return q.front();//返回合并出的根结点
}
```

无相关pb_ds库

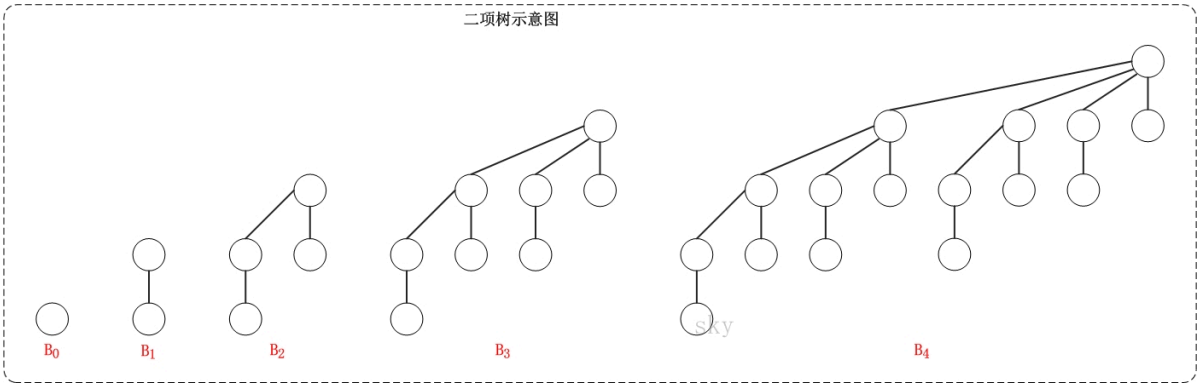
二项堆

二项树

定义

二项树是一种递归定义的有序树。它的递归定义如下：

1. 二项树 B_0 只有一个结点；
2. 二项树 B_k 由两棵二项树 B_{k-1} 组成的，其中一棵树是另一棵树根的最左孩子。



性质

1. B_k 共有 2^k 个节点。

   如上图所示，B_0 有 $2^0=1$ 节点，B_2 有 $2^1=2$个节点，B_2 有 $2^2=4$ 个节点，...

2. B_k 的高度为 k 。

   如上图所示，B_0 的高度为 0 ，B_1 的高度为 1 ，B_2 的高度为 2，...

3. B_k 在深度 i 处恰好有 C_k^i 个节点，其中 $i=0,1,2,\ldots,k$。

   B_4 中深度为 0 的节点 $C_4^0=1$

   B_4 中深度为 1 的节点 $C_4^1=4$

   B_4 中深度为 2 的节点 $C_4^2=6$

   B_4 中深度为 3 的节点 $C_4^3=4$

   B_4 中深度为 4 的节点 $C_4^4=1$

   合计得到 B_4 的节点分布是 $(1,4,6,4,1)$ 。

4. 根的度数为 k，它大于任何其它节点的度数。

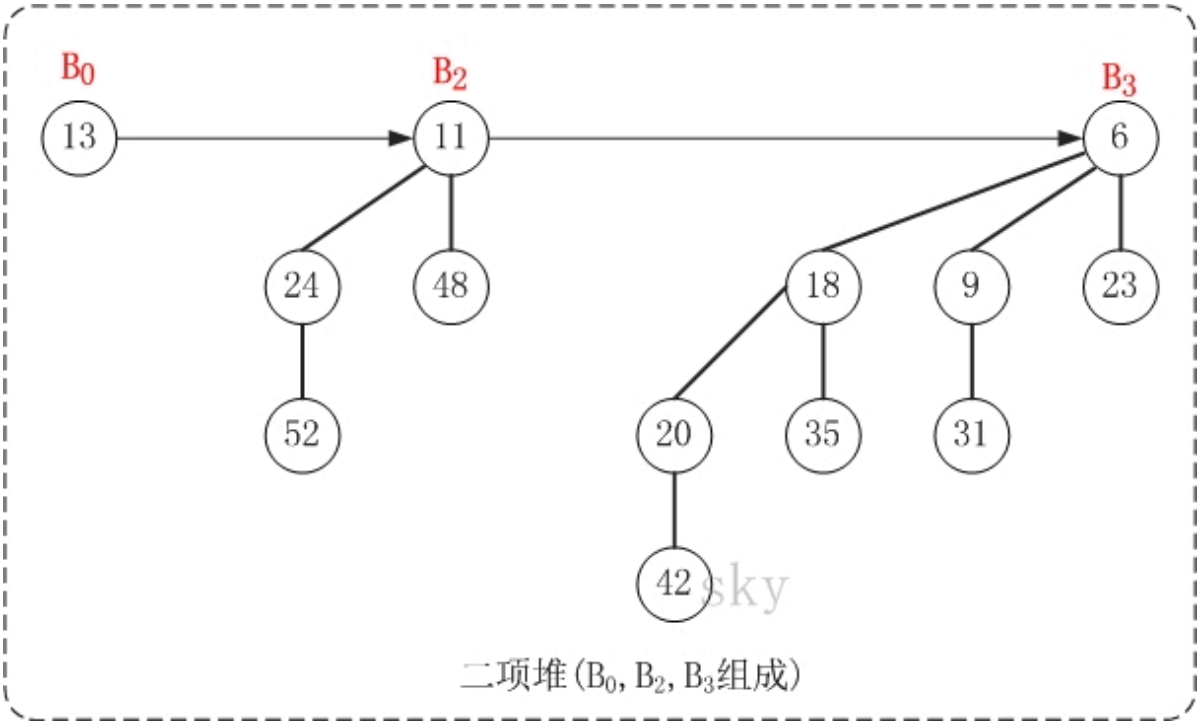
   节点的度数指该结点拥有的子树的数目。

二项堆定义

二项堆是满足以下性质的二项树的集合：

1. 每棵二项树都满足最小堆性质。即，父节点的关键字 $\leq$ 它的孩子的关键字。
2. 不能有两棵或以上的二项树具有相同的度数（包括度数为0）。换句话说，具有度数 k 的二项树有 0 个或 1 个。

由二项树 B_0、B_2 和 B_3 组成的二项堆



二项堆的第 1 个性质保证了二项堆的最小节点就是某个二项树的根节点。

第 2 个性质则说明结点数为 n 的二项堆最多只有 $\log n + 1$ 棵二项树。

实际上，将包含 n 个节点的二项堆，表示成若干个 2 的指数和（或者转换成二进制），则每一个 2 个指数都对应一棵二项树。

例如，13（二进制是1101）的2个指数和为 $13 = 2^3 + 2^2 + 2^0$，因此具有 13 个节点的二项堆由度数为 $3, 2, 0$ 的三棵二项树组成。

pb_ds库的二项堆模板

```
#define binomial_heap __gnu_pbds::priority_queue<int, std::great<int>,
binomial_heap_tag>

binomial_heap q1;  // 大根堆，配对堆
binomial_heap q2;
binomial_heap::point_iterator id;  // 一个迭代器

int main() {
    id = q1.push(1);
    // 堆中元素 :  [1];
    for (int i = 2; i <= 5; i++) q1.push(i);
    // 堆中元素 :  [1, 2, 3, 4, 5];
    std::cout << q1.top() << '\n';
    // 输出结果 : 5;
```

```
    q1.pop();
    // 堆中元素 : [1, 2, 3, 4];
    id = q1.push(10);
    // 堆中元素 : [1, 2, 3, 4, 10];
    q1.modify(id, 1);
    // 堆中元素 :  [1, 1, 2, 3, 4];
    std::cout << q1.top() << '\n';
    // 输出结果 : 4;
    q1.pop();
    // 堆中元素 : [1, 1, 2, 3];
    id = q1.push(7);
    // 堆中元素 : [1, 1, 2, 3, 7];
    q1.erase(id);
    // 堆中元素 : [1, 1, 2, 3];
    q2.push(1), q2.push(3), q2.push(5);
    // q1中元素 : [1, 1, 2, 3], q2中元素 : [1, 3, 5];
    q2.join(q1);
    // q1中无元素, q2中元素 : [1, 1, 1, 2, 3, 3, 5];
}
```

ST表

ST 表（Sparse Table，稀疏表）是用于解决 **可重复贡献问题** 的数据结构。

> 什么是可重复贡献问题?
>
> **可重复贡献问题** 是指对于运算 opt，满足 $x\ opt\ x = x$，则对应的区间询问就是一个可重复贡献问题。例如，最大值有 $max(x, x) = x$，最大公约数有 $gcd(x, x) = x$。所以 RMQ 和区间 GCD 就是一个可重复贡献问题。
>
> 比如区间和就不具有这个性质，如果求区间和的时候采用的预处理区间重叠了，则会导致重叠部分被计算两次。
>
> 另外，opt 还必须满足结合律才能使用 ST 表求解。

模板题

给定 n 个数，有 m 个询问，对于每个询问，你需要回答区间 $[l, r]$ 中的最大值。

过程

ST 表基于 **倍增** 思想，可以做到 $O(n \log n)$ 预处理， $O(1)$ 回答每个询问。但是不支持修改操作。

基于倍增思想，可以发现，如果按照一般的倍增流程，每次跳 2^i 步的话，询问时的复杂度仍旧是 $O(\log N)$，并没有比线段树更优，反而预处理一步还比线段树慢。

我们发现 $max(x, x) = x$，也就是说，区间最大值是一个具有「可重复贡献」性质的问题。即使用来求解的预处理区间有重叠部分，只要这些区间的并是所求的区间，最终计算出的答案就是正确的。

可以发现我们能**使用至多两个预处理过的区间**来覆盖询问区间，也就是说询问时的时间复杂度可以被降至 $O(1)$，在处理有大量询问的题目时十分有效。

预处理 $O(N \log N)$，查询 $O(1)$。

其他信息的维护

除 RMQ 以外，还有其它的「可重复贡献问题」。例如「区间按位与」、「区间按位或」、「区间 GCD」，ST 表都能高效地解决。

需要注意的是，对于「区间 GCD」，ST 表的查询复杂度并没有比线段树更优（令值域为 w，ST 表的查询复杂度为 $O(\log w)$，而线段树为 $O(\log N + \log w)$，且值域一般是大于 n 的），但是 ST 表的预处理复杂度也没有比线段树更劣，而编程复杂度方面 ST 表比线段树简单很多。

如果分析一下，「可重复贡献问题」一般都带有某种类似 RMQ 的成分。例如「区间按位与」就是每一位取最小值，而「区间 GCD」则是每一个质因数的指数取最小值。

具体实现

令 $f(i,j)$ 表示区间 $[i, i+2^j-1]$ 的最大值。

预处理部分：

$f(i,0) = a_i$，有 $f(i,j) = max(f(i,j-1), f(i+2^{j-1}, j-1))$。

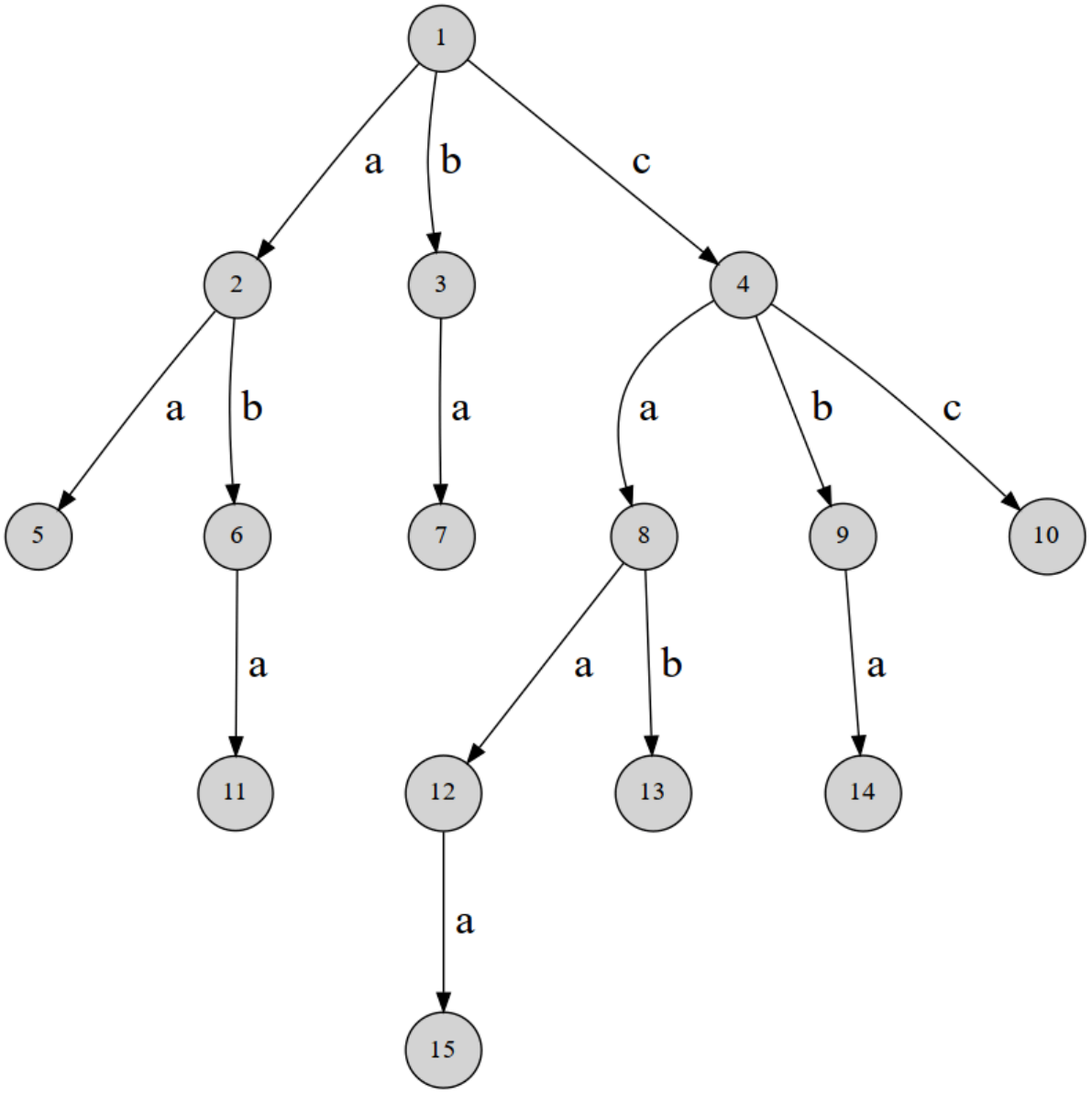
查询部分：

对于每个询问 $[l,r]$，我们把它分成两部分 $[l, l+2^s-1]$ 和 $[r-2^s+1, r]$，为保证 $l+2^s-1 \le r-2^s+1$，$s = \lfloor \log_2(r-l+1) \rfloor$。两部分的结果的最大值就是回答。

模板

```
int logn[MAXN];
int t; // t = log2(n) + 1;
int f[N][25];

void prework() {
    logn[1] = 0;
    logn[2] = 1;
    for (int i = 3; i < MAXN; i++) {
        logn[i] = logn[i / 2] +1;
    }

    t = logn[n] + 1;

    for (int j = 1; j <= t; j++) {
        for (int i = 1; i + (1 << j) - 1 <= n; i++) {
            f[i][j] = std::max(f[i][j - 1], f[i + (1 << (j - 1))][j - 1]);
        }
    }
}

int query(int l, int r) {
    int s = logn[r - l + 1];
    return std::max(f[l][s], f[r - (1 << s) + 1][s]);
}
```

Trie



Trie

这棵字典树用边来代表字母，而从根结点到树上某一结点的路径就代表了一个字符串。举个例子 $1 \to 4 \to 8 \to 12$，表示的就是字符串 `caa`。

trie 的结构非常好懂，我们用 `trie[x][c]` 表示结点 x 的 c 字符指向的下一个结点，或着说是结点 x 代表的字符串后面添加一个字符 c 形成的字符串的结点。（c 的取值范围和字符集大小有关，不一定是 $0-26$。）

基本插入 / 查询模板

```
int trie[MAXN][26], ncnt;
bool end[MAXN];

void insert(char* str) {
    int len = strlen(str), now = 0; // 根节点为0
    for (int i = 0; i < len; i++) {
        int d = str[i] - 'a';
        // 如果没有出边 则添加节点
        if (!trie[now][d]) trie[now][d] = ++ncnt;
```

```
        now = trie[now][d];
    }
    // 更新末节点信息
    end[now] = p;
}

bool find(char* str) {
    int len = strlen(str), now = 0;
    for (int i = 0; i < len; i++) {
        int d = str[i] - 'a';
        now = trie[now][d];
        if (!now) return 0;
    }
    return end[now];
}
```

下面将介绍其高级应用。

01-Trie

可以用来维护异或最大值，同时也多出现在位运算的场合，尤其是按位贪心等技巧，需要在 01-Trie 上二分等，需要掌握。

维护异或极值

01-Trie 是指字符集为 $\{0,1\}$ 的 Trie。

需要按值从高位到低位建立 trie。

具体看例题

例题

$Luogu\ P4551$ 最长异或路径

> 给定一棵 n 个点的带权树，结点下标从 1 开始到 n 。寻找树中找两个结点，求最长的异或路径。
>
> 异或路径指的是指两个结点之间唯一路径上的所有边权的异或。

实现

容易发现，两点之间的异或路径等于根节点分别到两个节点的异或路径的异或和。$D[x]$ 表示根节点到 x 的路径上所有边权的 xor 值，可以通过 DFS 解决。

剩下需要的就是寻找一个让某值的异或和最大的值。

01-Trie可以很好的满足这个需求。把每个值看成二进制，每位则由0或1构成。对于一个值，若某一位是0，为使异或和最大，需要寻找1；若某一位是1，则要寻找0。当然，需要存在上述的0或1供我们选择，若不存在，只能选择存在的0/1数。

根据贪心，我们肯定首先要满足高位的“0对1，1对0”需求，所以需要按值从低位到高位建立 01-Trie。

剩下的就转化为Trie的经典操作。

参考代码

```
int n;
int hd[MAXN], ver[MAXN << 1], nxt[MAXN << 1], edge[MAXN << 1], tot;
void add(int u, int v, int w)
{
    ver[++tot]= v;
    nxt[tot] = hd[u], hd[u] = tot;;
    edge[tot] = w;
}

int sum[MAXN];
void dfs(int x, int fa)
{
    for(int i = hd[x]; i; i = nxt[i])
    {
        int y = ver[i], e = edge[i];
        if (y == fa) continue;
        sum[y] = sum[x] ^ e;
        dfs(y, x);
    }
}

int trie[MAX_TRIE][2];
int ncnt;
int thenum[MAX_TRIE];
int ans;
void insert(int num)
{
    int now = 0;
    for(int i = 31; i >= 0; i--)
    {
        int bit = (num >> i) & 1;
        if (!trie[now][bit]) trie[now][bit] = ++ncnt;
        now = trie[now][bit];
    }
    thenum[now] = num;
}
int search(int num)
{
    int now = 0;
    for(int i = 31; i >= 0; i--)
    {
        int bit = (num >> i) & 1;
        // 寻找每一位取反的数 若没有只能选择相同的数
        if(trie[now][bit ^ 1]) now = trie[now][bit ^ 1];
        else now = trie[now][bit];
    }
    return thenum[now];
}
int main()
{
    std::cin >> n;
    for(int i = 1; i <= n - 1; i++)
    {
```

```
        int u, v, w;
        std::cin >> u >> v >> w;
        add(u, v, w), add(v, u, w);
    }
    dfs(1, -1);
    for(int i = 1; i <= n; i++) {
        // 根据二进制将每个数插入到01-Trie中
        insert(sum[i]);
    }
    for(int i = 1; i <= n; i++) {
        ans = std::max(ans, sum[i] ^ search(sum[i]));
    }
    std::cout << ans;
    return 0;
}
```

维护异或和

01-Trie 也可以用来维护一些数字的异或和，支持修改（删除 + 重新插入），和全局加一（即：让其所维护所有数值递增 1，本质上是一种特殊的修改操作）。

如果要维护异或和，需要按值从低位到高位建立 01-Trie。

插入

如果要维护异或和，我们 只需要 知道某一位上 `0` 和 `1` 个数的 **奇偶性** 即可，也就是对于数字 1 来说，当且仅当这一位上数字 1 的个数为奇数时，这一位上的数字才是 1，请时刻记住这段文字：**如果只是维护异或和，我们只需要知道某一位上 1 的数量即可，而不需要知道 Trie 到底维护了哪些数字。**

对于每一个节点，我们需要记录以下三个量：

- `trie[o][0/1]` 指节点 o 的两个儿子，`trie[o][0]` 指下一位是 0，同理 `trie[o][1]` 指下一位是 1。
- `w[o]` 指节点 o 到其父亲节点这条边上数值的数量（权值）。每插入一个数字 x，x 二进制拆分后在 trie 上 路径的权值都会 +1。
- `xorv[o]` 指以 o 为根的子树维护的异或和。

删除

插入和删除的代码非常相似。

```
// 这里的 MAXH 指 trie 的深度，也就是强制让每一个叶子节点到根的距离为 MAXH。
// 强制插入 MAXH 位，目的是为了便于全局 +1 时处理进位。
const int MAXH = 21;
int trie[MAXN * (MAXH + 1)][2], w[MAXN * (MAXH + 1)], xorsum[MAXN * (MAXH + 1)];
int tot = 0;

int mknode() {
    ++tot;
    trie[tot][0] = trie[tot][1] = w[tot] = xorsum[tot] = 0;
    return tot;
}

void update(int x) {
    w[x] = xorsum[x] = 0;
```

```
    if (trie[x][0]) {
        w[x] += w[trie[x][0]];
        xorsum[x] ^= xorsum[trie[x][0]] << 1;
    }
    if (trie[x][1]) {
        w[x] += w[trie[x][1]];
        xorsum[x] ^= (xorsum[trie[x][1]] << 1) | (w[trie[x][1]] & 1);
    }
    // w[x] = w[x] & 1;
    // 只需知道奇偶性即可，不需要具体的值。当然这句话删掉也可以，因为上文就只利用了他的奇偶性。
}

void insert(int &x, int v, int dep) {
    if (!x) x = mknode();
    if (dep > MAXH) return (void)(w[x]++);
    insert(trie[x][v & 1], v >> 1, dep + 1);
    update(x);
}

void erase(int x, int v, int dep) {
    if (dep > 20) return (void)(w[x]--);
    erase(trie[x][v & 1], v >> 1, dep + 1);
    update(x);
}
```

- 我们强制插入 MAXH 位，目的是为了便于全局 +1 时处理进位。例如：如果原数字是 3（11），递增之后变成 4（100），如果当初插入 3 时只插入了 2 位，那这里的进位就没了。
- 插入和删除，只需要修改叶子节点的 `w[]` 即可，在回溯的过程中一路维护即可。

全局加1

指让这棵 Trie 中所有的数值 +1。

形式化的讲，设 Trie 中维护的数值有 $V_1, V_2, V_3, \dots V_n$，全局加1后，其中维护的值应该变成 $V_1+1, V_2+1, V_3+1, \dots V_n+1$。

过程

我们只需要从低位到高位开始找第一个出现的 0，把它变成 1，然后这个位置后面的 1 都变成 0 即可。

下面给出几个例子感受一下：（括号内的数字表示其对应的十进制数字）。

> 1000(8) + 1 = 1001(9)；
> 10011(19) + 1 = 10100(20)；
> 11111(31) + 1 = 100000(32)；
> 10101(21) + 1 = 10110(22)；
> 1000000000111111(16447) + 1 = 1000000001000000(16448)

```
// 当为空时，addall没有任何效果
void addall(int x) {
    // 所有该位为1的数变成0 所有该位为0的数变成1 故交换即可
    std::swap(trie[x][0], trie[x][1]);
    // trie[x][0] 表示交换前下一位是1
    // 因为进位 如果存在1 则下一位以及下一位的后继还需要变化
    // 如果只有0 则将0变成1即可 后续没有进位
    if (trie[x][0]) addall(trie[x][0]);
    maintain(x);
}
```

对应 trie 的操作，其实就是交换其左右儿子，顺着 **交换后** 的 0 边往下递归操作即可。

总模板

```
constepxr int MAXN = 5e5 + 5;
constexpr int MAXH = 25;
int trie[MAXN * MAXH][2], w[MAXN * MAXH], xorsum[MAXN * MAXH];
int ncnt;
int mknode()
{
    ++ncnt;
    trie[ncnt][0] = trie[ncnt][1] = w[ncnt] = xorsum[ncnt] = 0;
    return ncnt;
}
void update(int x)
{
    w[x] = xorsum[x] = 0;
    if (trie[x][0]) {
        w[x] += w[trie[x][0]];
        xorsum[x] ^= (xorsum[trie[x][0]] << 1);
    }
    if (trie[x][1]) {
        w[x] += w[trie[x][1]];
        xorsum[x] ^= (xorsum[trie[x][1]] << 1) | (w[trie[x][1]] & 1);
    }
}
void insert(int& x, int v, int dep)
{
    if (!x) x = mknode();
    if (dep > 20) {
        w[x]++;
        return;
    }
    insert(trie[x][v & 1], v >> 1, dep + 1);
    update(x);
}
void erase(int x, int v, int dep)
{
    if (dep > 20) {
        w[x]--;
        return;
    }
    erase(trie[x][v & 1], v >> 1, dep + 1);
    update(x);
```

```
}
void addall(int x)
{
    std::swap(trie[x][0], trie[x][1]);
    if (trie[x][0]) addall(trie[x][0]);
    update(x);
}
```

例题

$Luogu\ P6018$ **[Ynoi2010] Fusion Tree**

> 给你一棵 n 个结点的树，每个结点有权值。m 次操作。 需要支持以下操作。
>
> 将树上与一个节点 x 距离为 1 的节点上的权值 。这里树上两点间的距离定义为从一点出发到另外一点的最短路径上边的条数。
> 在一个节点 x 上的权值 $-v$ 。
> 询问树上与一个节点 x 距离为 1 的所有节点上的权值的异或和。

trick：可以使用在每一个结点上设置懒标记来标记儿子的权值的增加量。这样就不需要在增加结点 x 的权值时，将 x的每个儿子都遍历一次，并且结点 x 的父亲的值可以通过 $fa[fa[x]]$ 的懒标记来记录到。

参考代码

```
constexpr int N = 5e5 + 5;
int hd[N], nxt[N << 1], ver[N << 1], tot;
void add(int x, int y) {
    ver[++tot] = y;
    nxt[tot] = hd[x], hd[x] = tot;
}

constexpr int MAXH = 25;
int trie[N * MAXH][2], w[N * MAXH], xorsum[N * MAXH];
int ncnt;
int mknode() {
    ++ncnt;
    trie[ncnt][0] = trie[ncnt][1] = w[ncnt] = 0;
    return ncnt;
}
void update(int x) {
    w[x] = xorsum[x] = 0;
    if (trie[x][0]) {
        w[x] += w[trie[x][0]];
        xorsum[x] ^= (xorsum[trie[x][0]] << 1);
    }
    if (trie[x][1]) {
        w[x] += w[trie[x][1]];
        xorsum[x] ^= (xorsum[trie[x][1]] << 1) | (w[trie[x][1]] & 1);
    }
}
void insert(int& x, int v, int dep) {
    if (!x) x = mknode();
    if (dep > 20) {
        w[x]++; return;
    }
    insert(trie[x][v & 1], v >> 1, dep + 1);
```

```
    update(x);
}
void erase(int x, int v, int dep) {
    if (dep > 20) {
        w[x]--; return;
    }
    erase(trie[x][v & 1], v >> 1, dep + 1);
    update(x);
}
void addall(int x) {
    std::swap(trie[x][0], trie[x][1]);
    if (trie[x][0]) addall(trie[x][0]);
    update(x);
}

int f[N], root[N], val[N], lazy[N];
void dfs0(int x, int fa) {
    f[x] = fa;
    for (int i = hd[x]; i; i = nxt[i]) {
        int y = ver[i];
        if (y == fa) continue;
        dfs0(y, x);
    }
}
int main() {
    int n, m;
    std::cin >> n >> m;
    for (int i = 1; i <= n - 1; i++) {
        int x, y;
        std::cin >> x >> y;
        add(x, y), add(y, x);
    }
    dfs0(1, -1);
    for (int i = 1; i <= n; i++) {
        std::cin >> val[i];
        if (f[i] == -1) continue;
        insert(root[f[i]], val[i], 0);
    }
    while (m--) {
        int opt, x, v;
        std::cin >> opt >> x;
        if (opt == 1) {
            lazy[x]++;
            if (x != 1) {
                if (f[f[x]] != -1) erase(root[f[f[x]]], lazy[f[f[x]]] +
val[f[x]], 0);
                val[f[x]]++;
                if (f[f[x]] != -1) insert(root[f[f[x]]], lazy[f[f[x]]] +
val[f[x]], 0);
            }
            addall(root[x]);
        } else if (opt == 2) {
            std::cin >> v;
            if (x != 1) erase(root[f[x]], lazy[f[x]] + val[x], 0);
            val[x] -= v;
            if (x != 1) insert(root[f[x]], lazy[f[x]] + val[x], 0);
```

```
        } else if (opt == 3) {
            int ans = xorsum[root[x]];
            ans ^= lazy[f[f[x]]] + val[f[x]];
            std::cout << ans << '\n';
        }
    }
    return 0;
}
```

合并Trie

合并 Trie 和合并线段树的思路非常相似，可以依照「合并线段树」来合并 Trie。

过程

考虑 int merge(int a, int b) 函数，这个函数传入两个 Trie 树位于**同一相对位置的结点编号**，然后合并完成后**返回合并完成的结点编号。**

分三种情况：

- 如果 a 没有这个位置上的结点，新合并的结点就是 b
- 如果 b 没有这个位置上的结点，新合并的结点就是 a
- 如果 a,b 都存在，那就把 b 的信息合并到 a 上，新合并的结点就是 a，然后递归操作处理 a 的左右儿子。

提示：如果需要的合并是将 a，b 合并到一棵新树上，这里可以**新建结点**，然后合并到这个新结点上，这里的代码实现仅仅是将 b 的信息合并到 a 上。

```
int merge(int a, int b) {
    if (!a) return b;  // 如果 a 没有这个位置上的结点，返回 b
    if (!b) return a;  // 如果 b 没有这个位置上的结点，返回 a
    /*
        如果 `a`, `b` 都存在，
        那就把 `b` 的信息合并到 `a` 上。
    */
    w[a] = w[a] + w[b];
    xorsum[a] ^= xorsum[b];
    /*
        不要使用 update()，
        update() 是合并a的两个儿子的信息
        而这里需要 a b 两个节点进行信息合并
    */
    // 此处针对01-Trie
    trie[a][0] = merge(trie[a][0], trie[b][0]);
    trie[a][1] = merge(trie[a][1], trie[b][1]);
    return a;
}
```

例题

$Luogu\ P6623$ 【省选联考2020A卷】树

> 给定一棵 n 个结点的有根树，结点从 1 开始编号，根结点为 1 号结点，每个结点有一个正整数权值 v_i 。 设 x 号结点的子树内（包含 x 自身）的所有结点编号为 $c_1, c_2, c_3, \ldots, c_k$，定义 x 的价值为：
>
> $$val(x) = (v_{c1} + d(c_1, x)) \oplus (v_{c2} + d(c_2, x)) \oplus \ldots \oplus (v_{ck} + d(c_k, x))$$
>
> 其中 $d(x, y)$ 表示树上 x 号结点与 y 号结点间唯一简单路径所包含的边数，$d(x, x) = 0$ 。 $\oplus$ 表示异或运算。 请你求出 $\sum_{i=1}^{n} val(i)$ 的结果。

考虑每个结点对其所有祖先的贡献。 每个结点建立 trie，初始先只存这个结点的权值，然后从底向上合并每个儿子结点上的 Trie，然后再全局加一，完成后统计答案。

参考代码

```
typedef long long ll;
constexpr int N = 6e5 + 5;
int val[N];

int hd[N], nxt[N << 1], ver[N << 1], tot;
void add(int x, int y) {
    ver[++tot] = y;
    nxt[tot] = hd[x], hd[x] = tot;
}

constexpr int MAXH = 25;
int trie[N * MAXH][2], w[N * MAXH], xorsum[N * MAXH];
int ncnt;
int mknode() {
    ++ncnt;
    trie[ncnt][0] = trie[ncnt][1] = w[ncnt] = xorsum[ncnt] = 0;
    return ncnt;
}
void update(int x) {
    w[x] = xorsum[x] = 0;
    if (trie[x][0]) {
        w[x] += w[trie[x][0]];
        xorsum[x] ^= xorsum[trie[x][0]] << 1;
    }
    if (trie[x][1]) {
        w[x] += w[trie[x][1]];
        xorsum[x] ^= (xorsum[trie[x][1]] << 1) | (w[trie[x][1]] & 1);
    }
}
void insert(int& x, int v, int dep) {
    if (!x) x = mknode();
    if (dep > 23) {
        w[x]++; return;
    }
    insert(trie[x][v & 1], v >> 1, dep + 1);
    update(x);
}
void erase(int x, int v, int dep) {
    if (dep > 23) {
```

```
        w[x]--; return;
    }
    erase(trie[x][v & 1], v >> 1, dep + 1);
    update(x);
}
void addall(int x) {
    std::swap(trie[x][0], trie[x][1]);
    if (trie[x][0]) addall(trie[x][0]);
    update(x);
}
int merge(int a, int b) {
    if (!a) return b;
    if (!b) return a;
    w[a] = w[a] + w[b];
    xorsum[a] ^= xorsum[b];
    trie[a][0] = merge(trie[a][0], trie[b][0]);
    trie[a][1] = merge(trie[a][1], trie[b][1]);
    return a;
}

int root[N];
ll ans;;
void dfs(int x, int fa) {
    for (int i = hd[x]; i; i = nxt[i]) {
        int y = ver[i];
        if (y == fa) continue;
        dfs(y, x);
        root[x] = merge(root[x], root[y]);
    }
    addall(root[x]);
    insert(root[x], val[x], 0);
    ans += 1ll * xorsum[root[x]];
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 1; i <= n; i++) {
        std::cin >> val[i];
    }
    for (int i = 2; i <= n; i++) {
        int p;
        std::cin >> p;
        add(p, i), add(i, p);
    }
    dfs(1, -1);
    std::cout << ans;
    return 0;
}
```

可持久化Trie

可持久化 Trie 的方式和可持久化线段树的方式是相似的，即每次只修改被添加或值被修改的节点，而保留没有被改动的节点，在上一个版本的基础上连边，使最后每个版本的 Trie 树的根遍历所能分离出的 Trie 树都是完整且包含全部信息的。

大部分的可持久化 Trie 题中，Trie 都是以 **01-Trie** 的形式出现的。

插入

插入数字的01-Trie

```
int trie[MAXN][2], ncnt, w[MAXN];

void insert(int x, int last, int v) {
    for (int i = 28; i >= 0; i--) {
        // 新的节点加上1 方便后续查询
        w[x] = w[last] + 1;
        int bit = (v >> i) & 1;
        if (!trie[x][bit]) trie[x][bit] = ++ncnt;
        trie[x][bit ^ 1] = trie[last][bit ^ 1];
        x = trie[x][bit], last = trie[last][bit];
    }
    w[x] = w[last] + 1;
}
```

插入字符串的普通Trie

```
int trie[MAXN][26], ncnt, w[MAXN];

void insert(int x, int last, char* str) {
    int len = strlen(str);
    for (int i = 0; i < len; i++) {
        // 新的节点加上1 方便后续查询
        w[x] = w[last] + 1;
        int d = str[i] - 'a';
        if (!trie[x][d]) trie[x][d] = ++ncnt;
        for (int j = 0; j < 26; j++) {
            if (j == d) continue;
            trie[x][j] = trie[last][j];
        }
        x = trie[x][bit], last = trie[last][bit];
    }
    w[x] = w[last] + 1;
}
```

查询

```
// 以01-Trie为例
int query(int l, int r, int v) {
    int ret = 0;
    for (int i = 28; i >= 0; i--) {
        int bit = (v >> i) & 1;
```

```
        // 如果两次相减等于0 相当于[l, r]的Trie内，该节点不存在 不能遍历
        // 相减大于0，则表示该节点存在
        if (w[trie[r][bit ^ 1]] - w[trie[l][bit ^ 1]] > 0) {
            r = trie[r][bit ^ 1], l = trie[l][bit ^ 1];
            ret += (1 << i);
        } else {
            r = trie[r][bit], l = trie[l][bit];
        }
    }
    return ret;
}
```

例题

$Luogu\ P4735$ 最大异或和

> 对一个长度为 n 的数组 a 维护以下操作：
>
> 1. 在数组的末尾添加一个数 x，数组的长度 n 自增 1。
> 2. 给出查询区间 $[l,r]$ 和一个值 k，求当 $l \leq p \leq r$ 时，$k \oplus \bigoplus_{i=p}^{n} a_i$ 的最大值。

类似于可持久化线段树的思路，考虑每次的查询都查询整个区间。我们只需把这个区间建一棵 Trie 树，将这个区间中的每个树都加入这棵 Trie 中，查询的时候，尽量往与当前位不相同的地方跳。

查询区间，只需要利用前缀和和差分的思想，用两棵前缀 Trie 树（也就是按顺序添加数的两个历史版本）相减即得到该区间的 Trie 树。再利用动态开点的思想，不添加没有计算过的点，以减少空间占用。

参考代码

```
constexpr int N = 6e5 + 5;
int a[N], s[N];

int root[N];
int trie[N * 28][2], w[N * 28], ncnt;
void insert(int x, int last, int v) {
    for (int i = 28; i >= 0; i--) {
        w[x] = w[last] + 1;
        int bit = (v >> i) & 1;
        if (!trie[x][bit]) trie[x][bit] = ++ncnt;
        trie[x][bit ^ 1] = trie[last][bit ^ 1];
        x = trie[x][bit], last = trie[last][bit];
    }
    w[x] = w[last] + 1;
}
int query(int l, int r, int v) {
    int ret = 0;
    for (int i = 28; i >= 0; i--) {
        int bit = (v >> i) & 1;
        if (w[trie[r][bit ^ 1]] - w[trie[l][bit ^ 1]] > 0) {
            r = trie[r][bit ^ 1], l = trie[l][bit ^ 1];
            ret += (1 << i);
        } else {
            r = trie[r][bit], l = trie[l][bit];
        }
    }
    return ret;
```

```
}

int main() {
    int n, m;
    std::cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
    }
    for (int i = 1; i <= n; i++) {
        s[i] = s[i - 1] ^ a[i];
    }
    for (int i = 1; i <= n; i++) {
        root[i] = ++ncnt;
        insert(root[i], root[i - 1], s[i]);
    }
    while (m--) {
        char opt; int l, r, x;
        std::cin >> opt;
        if (opt == 'A') {
            n++;
            std::cin >> a[n];
            s[n] = s[n - 1] ^ a[n];
            root[n] = ++ncnt;
            insert(root[n], root[n - 1], s[n]);
        } else if (opt == 'Q') {
            std::cin >> l >> r >> x;
            l--, r--;
            if (l == 0) {
                std::cout << std::max(s[n] ^ x, query(root[0], root[r], s[n] ^
x)) << '\n';
            } else std::cout << query(root[l - 1], root[r], s[n] ^ x) << '\n';
        }
    }
    return 0;
}
```

AC自动机

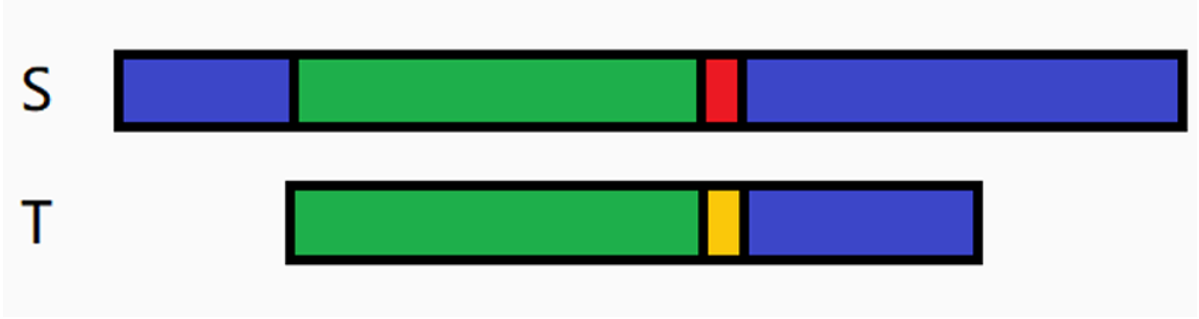
前置知识

KMP

在 $O(N)$ 时间内实现匹配两个字符串的算法。

匹配：给定你两个字符串 S 和 T ，需要 T 在 S 中所有出现的位置。

如果现在我们知道，至少这两个串的绿色部分是匹配的：



我们考虑在绿色部分中找到一个最长的前缀，也就是蓝色部分，满足蓝色部分和后半段部分一致。

换句话说 蓝色部分 是 绿色部分 的所有前缀中满足 **前缀 = 后缀** 的最长前缀。

接着我们直接挪动 T 串向前匹配：

容易发现，由于青色部分与后半段是一致的，因此到失配位置（红色）之前是都能匹配上的，只需要从红色向后匹配即可。

KMP 的跳法不会漏掉任何匹配。

失配数组`next[]`的求法

```
// 字符串 a 自我匹配
int next[MAXN];
int n; // 匹配串长度

next[1] = 0;
for (int i = 2, j = 0; i <= n; i++) {
    while (j > 0 && a[i] != a[j + 1]) j = next[j];
    if (a[i] == a[j + 1]) j++;
    next[i] = j;
}
```

匹配

```
int next[MAXN], f[MAXN];
int m; // 被匹配串长度

for (int i = 1, j = 0; i <= m; i++) {
    while (j > 0 && (j == n || b[i] != a[j + 1])) j = next[j];
    if (b[i] == a[j + 1]) j++;
    // 如果存在f[i] == n 表示T在S里匹配成功，即存在
    f[i] = j;
}
```

引入

> 给定 n 个模式串 $T_1, T_2, \ldots, T_n$ 和一个文本串 S
>
> 求每个模式串 T_i 在 S 中出现的次数。
>
> 比 KMP 更多的匹配。

AC 自动机是一种支持**多模式串匹配**的数据结构。

所谓多模式串匹配，就是用 $O(|S|)$ 的时间求出若干模式串 $T_1, T_2, \ldots, T_n$ 在 S 中出现了几次，出现在哪儿等等。

但是要比 KMP 的空间复杂度略大。

倘若直接用 KMP 做，时间复杂度为 $O(N|S|)$，不够优秀。

Trie 是一种很好的加载多个字符串的数据结构。如果我们给 Trie 上的每个结点加上一个指针，指向**最长的与后缀相同的前缀**，就能像 KMP 一样匹配。

过程

原理

我们设 $str(p)$ 表示 p 代表的字符串，$len(p)$ 表示 p 代表的字符串的长度。

我们设一个节点 p 的 $fail[p]$ 是在 Trie 树上满足 $str(r)[1, len(r)] = str(p)[len(p) - len(r) + 1, len(p)]$ 中最大的那个。

显然 $str(p)$ 的每一个后缀 $suffix(i)$，若存在 r 满足 $str(r) = suffix(i)$，那么 r 是唯一的，因此 $fail[p]$ 也是唯一的。

假设 $str(p) = aabbccdd$，有 $str(r_1) = bccdd$，$str(r_2) = ccdd$，$str(r_3) = cdd$。可以发现 r_1，r_2，r_3 都满足上述条件，但是 $fail[p] = r_1$，因为 $len(r_1)$ 最大。

求 `fail[]` 数组

和 KMP 的 `next[]` 数组一样，我们可以用类似跳法求出 `fail[]` 数组。只是线性在序列上跳变成了用 BFS 在 Trie 树上跳。

考虑一个节点 p，现在要算 $fail[p]$。p 的父亲为 f，$str[p] = str[f] + c$，并且此时所有 $len(s) < len(p)$ 的结点 s 的 $fail[s]$ 都已经求出。

现在我们考虑 $fail[f]$，如果存在 $son[fail[f]][c]$，那么有 $fail[p] = son[fail[f]][c]$。

否则我们令 $q = fail[f]$ 继续沿着 $fail$ 向上跳，直到达到根或者存在 $son[q][c]$，此时将 $fail[p]$ 设置为 $son[q][c]$。

证明类似 KMP。

匹配

类似 KMP 的匹配思路。

若在一个节点 p 没有找到出边 S_i，我们就跳回 $fail[p]$ 继续尝试找出边 S_i，直到跳回根。如果走到一个被标记为末尾的节点，记录一个匹配。

但是——

当你走到一个 $str(p) = aabb$ 的点时，需要注意的是，此时若 Trie 树上也存在一个点 q 使得 $str(q) = abb$ 并且 q 是终止节点，那么这个终止节点不会被标记。因为你走到 p 的路上是不会经过 q 的。

注意到 不断跳 $fail[p]$ 一定能到达这类 q。所以可以在维护 $fail[p]$ 数组时，就进行**"结束标记"**下传：如果 p 是终止节点，那么 $fail[p]$ 也是终止节点（伪终止结点）。

优化1

$O(|S|)$ 的时间复杂度是基于"每次匹配，都只会耗去 $O(1)$ 的时间"。如果有大量的 `a` 和一个 `aaaaa...aa` 是模式串，去匹配 `aaaaa...aa`，那么每个 `a` 都会将所有的 `a` 匹配一遍，给出现次数 +1，该次移动就会产生 $O(a的个数)$ 的计算。

瓶颈是需要优化遍历末尾标记的次数。

每一个点只有一个 $fail$ 指针，根代表空串没有 $fail$（或者说根的 $fail$ 就是自己）。

那么如果我们给每个点都建一条 "从 $fail$ 到自己的边"，最后会构成一棵**树**。 就叫 **fail 树**。

那么就很简单了，我们知道如果一个点被匹配到了，那么它在 fail 树上 的所有祖先都会被匹配到。

我们记录每个点被经过的次数（被匹配到的次数），然后在 fail 树上 DFS 一遍，统计每个点子树经过次数和，就是这个点被匹配到的次数。

优化2

在实际的应用中，我们也常常将 fail 树上祖先的儿子合并到当前节点上，具体来说：

如果结点 p 不存在边 c，那么 $son[p][c]$ 会被记为 $son[fail[p]][c]$。

此时我们建立的 AC 自动机是可以做到：

1. 任何一个串都可以在这个 AC 自动机上走一遍
2. 如果给定的串中包含模式串，那么会在遍历时走到一个终止节点 能通过跳 fail 到终止节点的点上

这样就不用不断跳 fail 链，也优化了时间复杂度。

参考代码

```
struct Node {
    int son[26], fail, cnt;
    std::vector<int> ed;
}tr[MAXN];
int ncnt, ans[MAXN];
std::vector<int> E[MAXN];

// 插入每个模式串 和 Trie 插入的方式相同
void insert(char* str, const int id) {
```

```
    int len = strlen(str), now = 0;
    for (int i = 0; i < len; i++) {
        int d = str[i] - 'a';
        if (!tr[now].son[d]) tr[now].son[d] = ++ncnt;
        now = tr[now].son[d];
    }
    tr[now].ed.push_back(id);
}

// 求fail[]数组 并且构建 fail 树
void get_fail() {
    static std::queue<int> q;
    for (int i = 0; i < 26; i++) {
        if (tr[0].son[i]) {
            q.push(tr[0].son[i]);
        }
    }
    while (!q.empty()) {
        int now = q.front(); q.pop();
        for (int i = 0; i < 26; i++) {
            int p = tr[now].son[i];
            if (p) {
                tr[p].fail = tr[tr[now].fail].son[i];
                q.push(p);
            } else tr[p].son[i] = tr[tr[p].fail].son[i];
        }
    }

    // 建fail树
    for (int i = 1; i <= ncnt; i++) {
        E[tr[i].fail].push_back(i);
    }
}
int dfs(int x) {
    int sum = tr[x].cnt;
    for (int y : E[x]) {
        sum += dfs(y);
    }
    for (int id : tr[x].ed) {
        ans[id] += sum;
    }
    return sum;
}
int main() {
    int n;
    static char s[MAXN];
    std::cin >> n;
    for (int i = 1; i <= n; i++) {
        std::cin >> s;
        insert(s, i);
    }
    get_fail();

    std::cin >> s;
    int len = strlen(s), now = 0;
    for (int i = 0; i < len; i++) {
```

```
        int d = s[i] - 'a';
        now = tr[now].son[d];
        // 经过的每个节点都cnt++ 最后统计答案的时候只有终止节点会被统计到
        tr[now].cnt++;
    }

    dfs(0);
    for (int i = 1; i <= n; i++) {
        std::cout << ans[i] << '\n';
    }
    return 0;
}
```

树状数组（BIT/Fenwick）

树状数组是一种支持 **单点修改** 和 **区间查询** 的，代码量小的数据结构。

普通树状数组维护的信息及运算要满足 **结合律** 且 **可差分**，如加法（和）、乘法（积）、异或等。

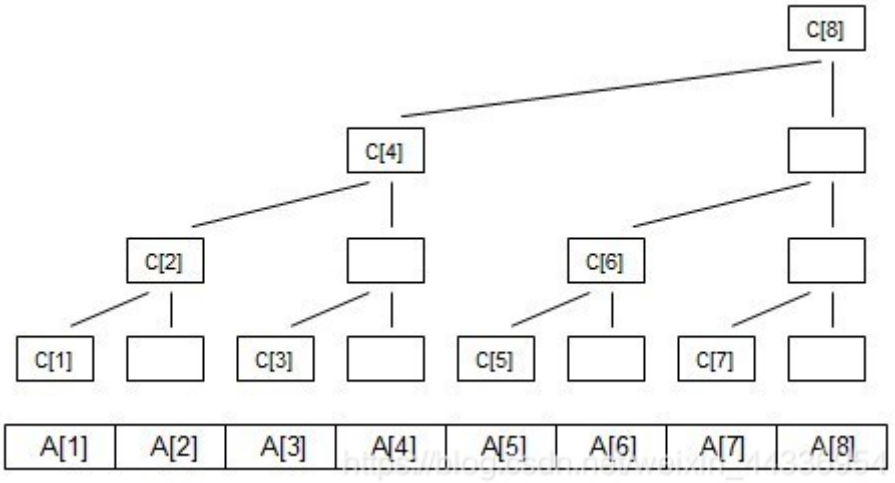
- 结合律：$(x \circ y) \circ z = x \circ (y \circ z)$，其中 $\circ$ 是一个二元运算符；
- 可差分：具有逆运算的运算，即已知 $x \circ y$ 和 x 可以求出 y 。

注意：

- 模意义下的乘法若要可差分，需保证每个数都存在逆元（模数为质数时一定存在）；
- 例如 gcd ， max 这些信息不可差分，所以不能用普通树状数组处理，但是：
  - 使用两个树状数组可以用于处理区间最值；
  - 具有支持不可差分信息查询的，时间复杂度为 $O(\log N)$ 的拓展树状数组。

在差分数组和辅助数组的帮助下，树状数组还可解决更强的 **区间增加** 问题。

原理



有：

- $C[1] = A[1]$;
- $C[2] = A[1] + A[2]$;
- $C[3] = A[3]$;

- $C[4]=A[1]+A[2]+A[3]+A[4]$;
- $C[5]=A[5]$;
- $C[6]=A[5]+A[6]$;
- $C[7]=A[7]$;
- $C[8]=A[1]+A[2]+A[3]+A[4]+A[5]+A[6]+A[7]+A[8]$;

$C[x]$ 保存序列 A $[x-lowbit(x)+1,x]$ 中所有数的和。

普通树状数组

模板

只能维护下标 $[1,n]$ 的数据。

如果下标到 0，需要将每个下标+1，作映射后再用树状数组。

```
int c[MAXN];
// 单点增加
void add(int x, int y) {
    // lowbit(x) = x & -x; 即最低位的 1 和后面的 0 构成的数
    // lowbit(01011000) = 00001000
    for (; x <= n; x += x & -x) c[x] += y;
}
// 区间查询 1 ~ x
int ask(int x) {
    int ans = 0;
    for (; x; x -= x & -x) ans += c[x];
    return ans;
}
```

因为下标范围原因，根据权值建立树状数组时通常需要 **离散化** 。

例题

$Luogu\ P10589$ 楼兰图腾

> 平面上有 N 个点，每个点的横、纵坐标的范围都是 $1\sim N$，任意两个点的横、纵坐标都不相同。
>
> 若三个点 $(x_1,y_1),(x_2,y_2),(x_3,y_3)$ 满足 $x_1<x_2<x_3,y_1>y_2$ 并且 $y_3>y_2$，则称这三个点构成"V"字图腾。
>
> 若三个点 $(x_1,y_1),(x_2,y_2),(x_3,y_3)$ 满足 $x_1<x_2<x_3,y_1<y_2$ 并且 $y_3<y_2$，则称这三个点构成"^"字图腾。
>
> 求平面上"V"和"^"字图腾的个数。

有题目明显看出是考察"逆序对"，而通过树状数组可以很简洁的求出逆序对个数。

不过要注意，求逆序对是树状数组的下标是权值，如果权值超过数组大小可承受的范围并且可以映射到较小的区间，则需要离散化操作。

该题需要正着求一次逆序对，也需要逆着求一次逆序对，然后计算出"^"字图腾；

求"V"字图腾则需要正着求一次正序对，逆着求一次正序对。

参考代码

```
int n;
int y[MAXN];

// 离散化操作
int disc[MAXN], tot;
void discrete() {
    std::sort(disc + 1,disc + 1 + tot);
    tot = std::unique(disc + 1,disc + 1 + tot) - disc - 1;
}
int get(int x) {
    return std::lower_bound(disc+1,disc+1+tot,x)-disc;
}

int prefix1[MAXN], prefix2[MAXN];//prefix1 正序 prefix2 逆序
void add(int a, int b, int array[]) {
    for(; a <= n; a += a & -a) array[a] += b;
}
int ask(int x, int array[]) {
    int ans = 0;
    for(; x; x -= x & -x) ans += array[x];
    return ans;
}

int up1[MAXN], down1[MAXN];//^
int down2[MAXN], up2[MAXN];//v
long long ans1,ans2;

int main() {
    std::cin >> n;
    for(int i = 1; i <= n; i++) {
        std::cin >> y[i];
        disc[++tot] = y[i];
    }
    discrete();
    for(int i = 1; i <= n; i++) {
        int x = get(y[i]);
        // 当前左边已经遍历了的并且比当前数值更小的数
        up1[i] = ask(x - 1, prefix1);
        // 当前左边已经遍历了的并且比当前数值更大的数
        // 即 已经遍历了的数 - 1（当前） - 已经遍历了的不大于当前数值的数
        down2[i] = i - 1 - ask(x, prefix1);
        add(x, 1, prefix1);
    }
    for(int i = n; i >= 1; i--) {
        int x = get(y[i]);
        // 当前右边已经遍历了的并且比当前数值更小的数
        down1[i] = ask(x - 1, prefix2);
        // 当前右边已经遍历了的并且比当前数值更大的数
        // 即 已经遍历了的数 - 1（当前） - 已经遍历了的不大于当前数值的数
        up2[i] = n - i - ask(x, prefix2);
        add(x, 1, prefix2);
    }
    for(int i = 1; i <= n; i++) {
```

```
        ans1 += 1ll * up1[i] * down1[i];
        ans2 += 1ll * up2[i] * down2[i];
    }
    std::cout << ans2 << ' ' << ans1;
    return 0;
}
```

区间增加的树状数组

原理

该问题可以使用两个树状数组维护差分数组解决。

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 $a_i = \sum_{j=1}^{i} d_j$。

一样地，我们考虑将查询区间和通过差分转化为查询前缀和。那么考虑查询 $a[1...r]$ 的和，即 $\sum_{i=1}^{r} a_i$，进行推导有：

$$\sum_{i=1}^{r} a_i = \sum_{i=1}^{r} \sum_{j=1}^{i} d_j$$

观察式子可以发现，每个 d_j 总共被加了 $r - j + 1$ 次。则：

$$\sum_{i=1}^{r} \sum_{j=1}^{i} d_j$$

$$= \sum_{i=1}^{r} d_i \times (r - i + 1)$$

$$= \sum_{i=1}^{r} (r+1) \times d_i - \sum_{i=1}^{r} d_i \times i$$

$\sum_{i=1}^{r} d_i$ 不能推出 $\sum_{i=1}^{r} d_i \times i$ 的值，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**。

- 对于维护 d_i 的树状数组，对 l 单点加 d，$r+1$ 单点加 $-d$；
- 对于维护 $i \times d_i$ 的树状数组，对 l 单点加 $l \times d$，$r+1$ 单点加 $-(r+1) \times d$

模板

```
typedef long long ll;
int a[MAXN];
ll t[2][MAXN], sum[MAXN];

void add(int k, int x, int y) {
    for (; x <= n; x += x & -x) t[k][x] += y;
}

ll ask(int k, int x) {
    ll ans = 0;
    for (; x; x -= x & -x) ans += t[k][x];
    return ans;
}
```

```cpp
int main() {
    int n, m;
    std::cin >> n >> m;

    // 求原数组前缀和
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
        sum[i] = sum[i - 1] + a[i];
    }

    while (m--) {
        int opt, l, r, d;
        std::cin >> opt;
        // opt为1代表区间加d
        if (opt == 1) {
            std::cin >> l >> r >> d;
            add(0, l, d), add(0, r + 1, -d);
            add(1, l, l * d), add(1, r + 1, -(r + 1) * d);
        }
        // opt为2代表查询区间和
        else if (opt == 2) {
            ll ans = (sum[r] + (r + 1) * ask(0, r) - ask(1, r))
                    -(sum[l - 1] + l * ask(0, l - 1) - ask(1, l - 1));
            std::cout << ans << '\n';
        }
    }
    return 0;
}
```

二维树状数组

原理

二维树状数组，也被称作树状数组套树状数组，用来维护二维数组上的**单点修改**和**前缀信息**问题。

我们用 $c[x][y]$ 表示 $a[x-lowbit(x)+1][y-lowbit(y)+1]\dots a[x][y]$ 矩阵总信息，即 $a[x][y]$ 为右下角，高 $lowbit(x)$ ，宽 $lowbit(y)$ 的矩阵的总信息。

单点修改

```cpp
void add(int x, int y, int v) {
    // 注意这里必须得建循环变量，不能像一维数组一样直接 while (x <= n) 了
    for (int i = x; i <= n; i += i & -i) {
        for (int j = y; j <= m; j += j & -j) {
            c[i][j] += v;
        }
    }
}
```

查询子矩阵和

```
int sum(int x, int y) {
    int res = 0;
    for (int i = x; i > 0; i -= lowbit(i)) {
        for (int j = y; j > 0; j -= lowbit(j)) {
            res += c[i][j];
        }
    }
    return res;
}

int ask(int x1, int y1, int x2, int y2) {
    // 查询子矩阵和
    return sum(x2, y2) - sum(x2, y1 - 1) - sum(x1 - 1, y2) + sum(x1 - 1, y1 - 1);
}
```

子矩阵修改

原理

同样考虑维护差分数组。

二维数组的差分：

$$d[i][j] = a[i][j] - a[i-1][j] - a[i][j-1] + a[i-1][j-1]$$

> 定义的原因：
>
> 理想规定状态下，在差分矩阵上做二维前缀和应该得到原矩阵，因为这是一对逆运算。
>
> 二维前缀和：
>
> $s(i,j) = s(i-1,j) + s(i,j-1) - s(i-1,j-1) + a(i,j)$
>
> 类比：
>
> $a(i,j) = a(i-1,j) + a(i,j-1) - a(i-1,j-1) + d(i,j)$
>
> 则 $d(i,j) = a(i,j) - a(i-1,j) - a(i,j-1) + a(i-1,j-1)$

这样一来，对左上角 (x_1, y_1)，右下角 (x_2, y_2) 的子矩阵区间加 v，相当于在差分数组上，对 $d[x_1][y_1]$ 和 $d[x_2+1][y_2+1]$ 分别单点加 v，对 $d[x_2+1][y_1]$ 和 $d[x_1][y_2+1]$ 分别单点加 $-v$。

现在考虑查询子矩阵和：

对于点 (x, y)，它的二维前缀和可以表示为：

$$\sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{h=1}^{i}\sum_{k=1}^{j} d(h,k)$$

原因就是差分的前缀和的前缀和就是原本的前缀和。

和一维树状数组的「区间加区间和」问题类似，统计 $d(h,k)$ 的出现次数，为 $(x-h+1) \times (y-k+1)$。

接着推导：

$$\sum_{i=1}^{x}\sum_{j=1}^{y}\sum_{h=1}^{i}\sum_{k=1}^{j}d(h,k)$$

$$=\sum_{i=1}^{x}\sum_{j=1}^{y}d(i,j)\times(x-i+1)\times(y-j+1)$$

$$=\sum_{i=1}^{x}\sum_{j=1}^{y}d(i,j)\times(xy+x+y+1)-d(i,j)\times i\times(y+1)-d(i,j)\times j\times(x+1)+d(i,j)\times i\times j$$

所以我们需维护四个树状数组，分别维护 $d(i,j)$，$d(i,j)\times i$，$d(i,j)\times j$，$d(i,j)\times i\times j$

模板

```
typedef long long ll;
ll t1[N][N], t2[N][N], t3[N][N], t4[N][N];

void add(ll x, ll y, ll z) {
    for (int i = x; i <= n; i += lowbit(i)) {
        for (int j = y; j <= m; j += lowbit(j)) {
            t1[i][j] += z;
            t2[i][j] += z * x;  // 注意是 z * x 后面同理
            t3[i][j] += z * y;
            t4[i][j] += z * x * y;
        }
    }
}

void range_add(ll xa, ll ya, ll xb, ll yb, ll z) {
    //(xa, ya) 到 (xb, yb) 子矩阵
    add(xa, ya, z);
    add(xa, yb + 1, -z);
    add(xb + 1, ya, -z);
    add(xb + 1, yb + 1, z);
}

ll ask(ll x, ll y) {
    ll res = 0;
    for (int i = x; i; i -= lowbit(i)) {
        for (int j = y; j; j -= lowbit(j)) {
            res += (x + 1) * (y + 1) * t1[i][j] - (y + 1) * t2[i][j] - (x + 1) *
t3[i][j] + t4[i][j];
        }
    }
    return res;
}

ll range_ask(ll xa, ll ya, ll xb, ll yb) {
    return ask(xb, yb) - ask(xb, ya - 1) - ask(xa - 1, yb) + ask(xa - 1, ya - 1);
}
```

权值树状数组+倍增

单点修改，查询全局第 k 小

朴素（树状数组+二分）

第 k 大问题可以通过简单计算转化为第 k 小问题。

该问题可离散化，如果原序列 a 值域过大，离散化后再建立权值数组 b 。注意，还要把单点修改中的涉及到的值也一起离散化，不能只离散化原数组 a 中的元素。

对于单点修改，只需将对原数列的单点修改转化为对权值数组的单点修改即可。具体来说，原数组 $a[x]$ 从 y 修改为 z ，转化为对权值数组 t 的单点修改就是 $t[y]$ 单点减 1 ， $t[z]$ 单点加 1 。

对于查询第 k 小，考虑二分 x ，查询权值树状数组中 $[1,x]$ 的前缀和，找到 x_0 使得 $[1,x_0]$ 的前缀和 $<k$ 而 $[1,x_0+1]$ 的前缀和 $\geq k$ ，则第 k 大的数是 x_0+1 （这里认为 $[1,0]$ 的前缀和是0）

时间复杂度 $O(\log^2 N)$ 。事实上还可以优化。

优化（树状数组+倍增）

考虑用倍增替代二分。

设 $x=0$，$sum=0$ ，枚举 i 从 $\log_2 n$ 降为0：

- 查询权值数组中 $[x+1...x+2^i]$ 的区间和 t 。
- 如果 $sum+t<k$ ，扩展成功，$x=x+2^i$ ， $sum=sum+t$；否则扩展失败，不操作。
- 这样得到的 x 是满足 $[1...x]$ 前缀和 $<k$ 的最大值，所以最终 $x+1$ 就是答案。

看起来这种方法时间效率没有任何改善，但事实上，查询 $[x+1,x+2^i]$ 的区间和只需访问 $c[x+2^i]$ 的值即可。

时间复杂度降低为 $O(\log N)$

模板

```
int kth(int k) {
    int sum = 0, x = 0;
    for (int i = log2(n); ~i; i--) {
        x += 1 << i;                    // 尝试扩展
        if (x >= n || sum + t[x] >= k)  // 如果扩展失败
            x -= 1 << i;
        else
            sum += t[x];
    }
    return x + 1;
}
```

Tricks

$O(N)$ 建树

一般来说，对于一个序列 a，对其中每一个值进行单点增加进行树状数组构建，时间复杂度为 $O(N \log N)$。

方法一

每一个节点的值是由所有与自己直接相连的儿子的值求和得到的。因此可以倒着考虑贡献，即每次确定完儿子的值后，用自己的值更新自己的直接父亲。

```
void init() {
    for (int i = 1; i <= n; i++) {
        t[i] += a[i];
        int j = i + lowbit(i);
        if (j <= n) t[j] += t[i];
    }
}
```

方法二

前面讲到 $c[i]$ 表示的区间是 $[i - lowbit(i) + 1, i]$，那么我们可以先预处理一个 sum 前缀和数组，再计算 c 数组。

```
void init() {
    for (int i = 1; i <= n; i++) {
        t[i] = sum[i] - sum[i - lowbit(i)];
    }
}
```

时间戳优化

对付多组数据很常见的技巧。若每次输入新数据都暴力清空树状数组，就可能会造成超时。因此使用 tag 标记，存储当前节点上次使用时间（即最近一次是被第几组数据使用）。每次操作时判断这个位置 tag 中的时间和当前时间是否相同，就可以判断这个位置应该是 0 还是数组内的值。

```
int tag[MAXN], t[MAXN], Tag;

void reset() { ++Tag; }

void add(int x, int v) {
    for (; x <= n; x += x & -x) {
        if (tag[x] != Tag) t[x] = 0;
        t[x] += v, tag[x] = Tag;
    }
}

int getsum(int k) {
    int ret = 0;
    for (; x; x -= x & -x) {
        if (tag[k] == Tag) ret += t[k];
    }
```

```
    return ret;
}
```

线段树

基本线段树

```
typedef long long ll;
int n,m;
ll num[MAXN];
struct Tree {
    int l, r;
    ll sum, add; // add为懒标记
}t[MAXN << 2];
// 信息上传
void pushup(int p) {
    t[p].sum = t[p << 1].sum + t[p << 1 | 1].sum;
}
// 建树
void build(int p, int l, int r) {
    t[p].l = l,t[p].r = r;
    if (l == r) {
        t[p].sum = num[l];
        return;
    }
    int mid = l + r >> 1;
    build(p << 1, l, mid);
    build(p << 1 | 1, mid + 1, r);
    pushup(p);
}
// 懒标记下传
void pushdown(int p) {
    if (t[p].add) {
        t[p << 1].sum += t[p].add * (t[p << 1].r - t[p << 1].l + 1);
        t[p << 1 | 1].sum += t[p].add * (t[p << 1 | 1].r - t[p << 1 | 1].l + 1);
        t[p << 1].add += t[p].add;
        t[p << 1 | 1].add += t[p].add;
        t[p].add = 0; // 已经下传 当前懒标记清零
    }
}
// 区间询问
ll query(int p, int l, int r) {
    if (t[p].l >= l && t[p].r <= r) {
        return t[p].sum;
    }
    pushdown(p);
    int mid = t[p].l + t[p].r >> 1;
    ll sum = 0;
    if (l <= mid) sum += query(p << 1, l, r);
    if (r > mid + 1) sum += query(p << 1 | 1, l, r);
    return sum;
}
```

```
// 区间修改
void modify(int p, int l, int r, ll k) {
    if (t[p].l >= l && t[p].r <= r) {
        t[p].sum += (t[p].r - t[p].l + 1) * k;
        t[p].add += k;
        return;
    }
    pushdown(p);
    int mid = t[p].l + t[p].r >> 1;
    if (l <= mid) modify(p << 1, l, r, k);
    if (r > mid) modify(p << 1 | 1, l, r, k);
    pushup(p);
}
int main() {
    std::cin >> n >> m;
    for(int i = 1; i <= n; i++) {
        std::cin >> num[i];
    }
    build(1, 1, n);
    for(int i = 1; i <= m; i++) {
        int opt, x, y; ll k;
        std::cin >> opt >> x >> y;
        if (opt == 1) {
            std::cin >> k;
            add(1,x,y,k);
        } else if (opt == 2) {
            std::cout << ask(1, x, y) << '\n';
        }
    }
    return 0;
}
```

动态开点线段树

有时候由于空间原因，不能开满整个线段树，这个时候就需要动态开点线段树。

这个时候 p 对应左子树右子树不再是 $p << 1$ 和 $p << 1 | 1$，而是像 Trie 一样，遇到空节点再创建一个新节点。

并且结构体内需要存储左子树和右子树对应的节点。

```
int n, q;
struct Tree {
    int l, r; // 保存左右子树对应的节点
    int lazy, sum;
    // 如果需要初始化懒标记为 -1
    Tree() {
        lazy = -1;
    }
}t[N << 6]; // 通常需要开45倍以上的空间
int idx; // 节点
int root;
// 信息上传
void pushup(int p) {
```

```
    t[p].sum = t[t[p].l].sum + t[t[p].r].sum;
}
// 懒标记下传
void pushdown(int p, int l, int r) {
    if (t[p].lazy != -1) {
        int mid = l + r >> 1;
        if (!t[p].l) t[p].l = ++idx;
        if (!t[p].r) t[p].r = ++idx;
        int ls = t[p].l, rs = t[p].r;
        t[ls].lazy = t[rs].lazy = t[p].lazy;
        t[ls].sum = t[p].lazy * (mid - l + 1);
        t[rs].sum = t[p].lazy * (r - mid);
        t[p].lazy = -1;
    }
}
// 区间修改
// l r 表示当前子树的范围
// ql qr 表示需要修改的范围
void modify(int& p, int l, int r, int ql,int qr, int k) {
    if(!p) p = ++idx;
    if(ql <= l && r <= qr) {
        t[p].lazy = k;
        t[p].sum = k * (r - l + 1);
        return;
    }
    pushdown(p, l, r);
    int mid = l + r >> 1;
    if (ql <= mid) modify(t[p].l, l, mid, ql, qr, k);
    if (qr > mid) modify(t[p].r, mid + 1, r, ql, qr, k);
    pushup(p);
}
int main() {
    std::cin >> n >> q;
    modify(root, 1, n, 1, n, 1);
    for (int i = 1; i <= q; i++) {
        int l, r, opt;
        std::cin >> l >> r >> opt;
        if (opt == 1) modify(root, 1, n, l, r, 0);
        else if (opt == 2) modify(root, 1, n, l, r, 1);
        std::cout << t[root].sum << '\n';
    }
    return 0;
}
```

可持久化线段树 / 主席树

基础模板（没有懒标记）

```
int n,m;
int num[N];
int root[N];
int idx;
struct Tree {
```

```
    int ls, rs;
    int val;
}t[N << 5];
int build(int l, int r) {
    int p = ++idx;
    if (l == r) {
        t[p].val = num[l];
        return p;
    }
    int mid = l + r >> 1;
    t[p].ls = build(l, mid);
    t[p].rs = build(mid + 1, r);
    return p;
}
int modify(int now,int l,int r,int x,int v) {
    int p = ++idx;
    t[p] = t[now];
    if (l == r) {
        t[p].val = v;
        return p;
    }
    int mid = l + r >> 1;
    if (x <= mid) t[p].ls = modify(t[now].ls, l, mid, x, v);
    else t[p].rs = modify(t[now].rs, mid + 1, r, x, v);
    return p;
}
int query(int p, int l, int r, int x) {
    if (l == r) return t[p].val;
    int mid = l + r >> 1;
    if (x <= mid) return query(t[p].ls, l, mid, x);
    else return query(t[p].rs, mid + 1, r, x);
}
int main() {
    std::cin >> n >> m;
    for(int i = 1; i <= n; i++) {
        std::cin >> num[i];
    }
    root[0] = build(1, n);
    for(int i = 1; i <= m; i++) {
        int v,opt,loc;
        std::cin >> v >> opt >> loc;
        if (opt == 1) {
            int val;
            std::cin >> val;
            root[i] = modify(root[v], 1, n, loc, val);
        }
        else if (opt == 2) {
            std::cout << query(root[v], 1, n, loc) << '\n';
            root[i] = root[v];
        }
    }
    return 0;
}
```

可持久化权值线段树（查询区间第k小）

如果是可持久化权值线段树可以不用写 `build()`

```
int n,m;
int num[N];

int disc[N], tot;
void discrete() {
    std::sort(disc + 1, disc + 1 + tot);
    tot = std::unique(disc + 1, disc + 1 + tot) - disc - 1;
}
int get(int x) {
    return std::lower_bound(disc + 1, disc + 1 + tot, x) - disc;
}

struct Tree
{
    int ls,rs;
    int sum;
}t[N*32];
int idx, root[N];
void pushup(int p) {
    t[p].sum = t[t[p].ls].sum + t[t[p].rs].sum;
}
int build(int l,int r) {
    int p = ++idx;
    if(l == r) return p;
    int mid = l + r >> 1;
    t[p].ls = build(l, mid);
    t[p].rs = build(mid + 1, r);
    return p;
}
int modify(int now, int l, int r, int x, int v) {
    int p = ++idx;
    t[p] = t[now];
    if(l == r) {
        t[p].sum += v;
        return p;
    }
    int mid = l + r >> 1;
    if(x <= mid) t[p].ls = modify(t[now].ls, l, mid, x, v);
    else t[p].rs = modify(t[now].rs, mid + 1, r, x, v);
    pushup(p);
    return p;
}
int query(int p, int q, int l, int r, int k) {
    if(l == r) return l;
    int mid = l + r >> 1;
    int cnt = t[t[q].ls].sum - t[t[p].ls].sum;
    if(cnt >= k) return query(t[p].ls, t[q].ls, l, mid, k);
    else return query(t[p].rs, t[q].rs, mid + 1, r, k - cnt);
}
int main() {
    std::cin >> n >> m;
```

```
    for(int i = 1; i <= n; i++) std::cin >> num[i];
    for(int i = 1; i <= n; i++) disc[++tot] = num[i];
    discrete();
    // 可以不用写
    // root[0]=build(1,tot);
    for(int i = 1; i <= n; i++)
    {
        int x = get(num[i]);
        root[i] = modify(root[i - 1], 1, tot, x, 1);
    }
    for(int i = 1; i <= m; i++)
    {
        int l, r, k;
        std::cin >> l >> r >> k;
        std::cout << disc[query(root[l - 1], root[r], 1, tot, k)]<<'\n';
    }
    return 0;
}
```

带懒标记的可持久化线段树（标记永久化）

回想我们之前处理线段树的方法。

我们在遍历到叶子后，要pushup一层层把值上传更新。

在有区间修改的操作时，我们需要加上pushdown操作，每次查询和修改都要把影响下推。

但是在主席树中，我们有一个特点，就是共用节点。

`modify` 部分：

```
//标记直接打上，子节点打上标记后的影响也直接计算，相当于直接pushup了
int modify(int now, int l, int r, int ql, int qr, int d) {
    int p = ++idx;
    t[p] = t[now];
    //要把子节点标记的影响算进来，有点类似pushup。这里要在if之前，可以自己想一
下。
    t[p].sum += (min(r, qr) - max(l, ql) + 1) * d;
    if(l >= ql && r <= qr){
        t[p].add += d; //直接累加标记
        return;
    }
    int mid = l + r >> 1;
    if(ql <= mid) modify(t[pre].l, l, mid, ql, qr, d);
    if(qr > mid) modify(t[pre].r, mid + 1, r, ql, qr, d);
}
```

`query` 部分：

```
//关键就是，把一路上的标记的影响累加下来，最后一起计算就可以了。
//相当于pushdown一样
int query(int p, int l, int r, int L, int R, int add) {
    if(ql <= l && r <= qr){
        return t[p].sum + (r - l + 1) * add;
    }
    int ans = 0;
    add += t[p].add; //这里不用放在if上面
    int mid = l + r >> 1;
    if(L <= mid) ans += query(t[p].l, l, mid, ql, qr, add);
    if(R > mid) ans += query(t[p].r, mid + 1, r, ql, qr, add);
    return ans;
}
```

模板

```
using ll = long long;
int a[N];

struct Tree {
    int ls, rs;
    ll sum, add;
}t[N << 6];
int root[N], idx;

int build(int l, int r) {
    int p = ++idx;
    if (l == r) {
        t[p].sum = a[l];
        return p;
    }
    int mid = l + r >> 1;
    t[p].ls = build(l, mid);
    t[p].rs = build(mid + 1, r);
    t[p].sum = t[t[p].ls].sum + t[t[p].rs].sum;
    return p;
}
int modify(int now, int l, int r, int ql, int qr, ll d) {
    int p = ++idx;
    t[p] = t[now];
    t[p].sum += (std::min(r, qr) - std::max(l, ql) + 1) * d;
    if (ql <= l && r <= qr) {
        t[p].add += d;
        return p;
    }
    int mid = l + r >> 1;
    if (ql <= mid) t[p].ls = modify(t[now].ls, l, mid, ql, qr, d);
    if (qr > mid) t[p].rs = modify(t[now].rs, mid + 1, r, ql, qr, d);
    return p;
}
ll query(int p, int l, int r, int ql, int qr, ll add) {
    if (ql <= l && r <= qr) {
```

```
        return t[p].sum + (r - l + 1) * add;
    }
    ll ans = 0;
    int mid = l + r >> 1;
    if (ql <= mid) ans += query(t[p].ls, l, mid, ql, qr, add + t[p].add);
    if (qr > mid) ans += query(t[p].rs, mid + 1, r, ql, qr, add + t[p].add);
    return ans;
}
int main() {
    int n, m;
    std::cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
    }
    root[0] = build(1, n);
    int tag = 0;
    while (m--) {
        char opt;
        int l, r, d, t;
        std::cin >> opt;
        if (opt == 'C') {
            std::cin >> l >> r >> d;
            ++tag;
            root[tag] = modify(root[tag - 1], 1, n, l, r, d);
        } else if (opt == 'Q') {
            std::cin >> l >> r;
            std::cout << query(root[tag], 1, n, l, r, 0) << '\n';
        } else if (opt == 'H') {
            std::cin >> l >> r >> t;
            std::cout << query(root[t], 1, n, l, r, 0) << '\n';
        } else if (opt == 'B') {
            std::cin >> t;
            tag = t;
        }
    }
    return 0;
}
```

线段树合并

过程

顾名思义，线段树合并是指建立一棵新的线段树，这棵线段树的每个节点都是两棵原线段树对应节点合并后的结果。它常常被用于维护树上或是图上的信息。

显然，我们不可能真的每次建满一颗新的线段树，因此我们需要使用动态开点线段树。

假设两颗线段树为 A 和 B，我们从 1 号节点开始递归合并。

递归到某个节点时，如果 A 树或者 B 树上的对应节点为空，直接返回另一个树上对应节点，这里运用了动态开点线段树的特性。

如果递归到叶子节点，我们合并两棵树上的对应节点。

最后，根据子节点更新当前节点并且返回。

时间复杂度 $O(N \log N)$

> 对于两颗满的线段树，合并操作的复杂度是 $O(N \log N)$ 的。
>
> 但实际情况下使用的常常是权值线段树，总点数和 N 的规模相差并不大。并且合并时一般不会重复地合并某个线段树，所以我们最终增加的点数大致是 $N \log N$ 级别的。这样，总的复杂度就是 $O(N \log N)$ 级别的。当然，在一些情况下，可并堆可能是更好的选择。

实现

不开新节点（可能出现重复利用某棵子树时因已被更新导致出错的问题）

```
int merge(int a, int b, int l, int r) {
    if (!a || !b) return a + b;
    if (l == r) {
        // do something...
        return a;
    }
    int mid = l + r >> 1;
    tr[a].ls = merge(tr[a].ls, tr[b].ls, l, mid);
    tr[a].rs = merge(tr[a].rs, tr[b].rs, mid + 1, r);
    pushup(a);
    return a;
}
```

开新节点

假设 a 和 b 合并 b 树上对应某个区间的点接在 a 上，变成了 a 树上对应这个区间的点，后来 a 再合并一棵树 c ， c 上对应这个区间的点合并时就会改变原本在 b 上的点，那dfs结束之后已经不是我们需要的当时的那棵 b 树了

```
int merge(int a, int b, int l, int r) {
    if (!a || ! b) return a + b;
    int cur = ++idx;
    if (l == r) {
        t[cur].sum = t[a].sum + t[b].sum;
        return cur;
    }
    int mid = l + r >> 1;
    t[cur].ls = merge(t[a].ls, t[b].ls, l, mid);
    t[cur].rs = merge(t[a].rs, t[b].rs, mid + 1, r);
    pushup(cur);
    return cur;
}
```

线段树分裂

过程

线段树分裂实质上是线段树合并的逆过程。线段树分裂只适用于有序的序列，无序的序列是没有意义的，常用在动态开点的权值线段树。

注意当分裂和合并都存在时，我们在合并的时候必须回收节点，以避免分裂时会可能出现节点重复占用的问题。

从一颗区间为 $[1,N]$ 的线段树中分裂出 $[l,r]$，建一颗新的树：

- 从 1 号结点开始递归分裂，当节点不存在或者代表的区间 $[s,t]$ 与 $[l,r]$ 没有交集时直接回溯。
- 当 $[s,t]$ 与 $[l,r]$ 有交集时需要开一个新结点。
- 当 $[s,t]$ 包含于 $[l,r]$ 时，需要将当前结点直接接到新的树下面，并把旧边断开。

时间复杂度为 $O(\log N)$。

```
int New() {
    return cnt ? rub[cnt--] : ++idx;
}
void Del(int &p) {
    ls[p] = rs[p] = sum[p] = 0;
    rub[++cnt] = p;
    p = 0;
}

void split(int &p, int &q, int l, int r, int s, int t) {
    if (r < ql || qr < l) return;
    if (!p) return;
    if (ql <= l && r <= qr) {
        q = p;
        p = 0;
        return;
    }
    if (!q) q = New();
    int mid = l + r >> 1;
    if (ql <= mid) split(t[p].ls, t[q].ls, l, mid, ql, qr);
    if (mid < qr) split(t[p].rs, t[q].rs, mid + 1, r, ql, qr);
    pushup(p), pushup(q);
}
```

模板（包含线段树合并）

```
using ll = long long;
int n, m;
int a[N];

struct Tree {
    int ls, rs;
    ll sum;
}t[N << 5];
int root[N << 2], idx = 1;
```

```
int rub[N << 5], cnt, tot;
int New() {
    return cnt ? rub[cnt--] : ++tot;
}
void Del(int& p) {
    t[p].ls = t[p].rs = t[p].sum = 0;
    rub[++cnt] = p;
    p = 0;
}
void pushup(int p) {
    t[p].sum = t[t[p].ls].sum + t[t[p].rs].sum;
}
void build(int& p, int l, int r) {
    if (!p) p = New();
    if (l == r) {
        t[p].sum = a[l];
        return;
    }
    int mid = l + r >> 1;
    build(t[p].ls, l, mid);
    build(t[p].rs, mid + 1, r);
    pushup(p);
}
void modify(int& p, int l, int r, int x, int v) {
    if (!p) p = New();
    if (l == r) {
        t[p].sum += v;
        return;
    }
    int mid = l + r >> 1;
    if (x <= mid) modify(t[p].ls, l, mid, x, v);
    else modify(t[p].rs, mid + 1, r, x, v);
    pushup(p);
}
int merge(int a, int b, int l, int r) {
    if (!a || !b) return a + b;
    if (l == r) {
        t[a].sum += t[b].sum;
        Del(b);
        return a;
    }
    int mid = l + r >> 1;
    t[a].ls = merge(t[a].ls, t[b].ls, l, mid);
    t[a].rs = merge(t[a].rs, t[b].rs, mid + 1, r);
    Del(b);
    pushup(a);
    return a;
}
void split(int& p, int& q, int l, int r, int ql, int qr) {
    if (!p) return;
    if (ql <= l && r <= qr) {
        q = p;
        p = 0;
        return;
    }
    if (!q) q = New();
```

```
    int mid = l + r >> 1;
    if (ql <= mid) split(t[p].ls, t[q].ls, l, mid, ql, qr);
    if (mid < qr) split(t[p].rs, t[q].rs, mid + 1, r, ql, qr);
    pushup(p), pushup(q);
}
ll query(int p, int l, int r, int ql, int qr) {
    if (!p) return 0;
    if (ql <= l && r <= qr) return t[p].sum;
    int mid = l + r >> 1;
    ll ans = 0;
    if (ql <= mid) ans += query(t[p].ls, l, mid, ql, qr);
    if (qr > mid) ans += query(t[p].rs, mid + 1, r, ql, qr);
    return ans;
}
int kth(int p, int l, int r, int k) {
    if (l == r) return l;
    int mid = l + r >> 1;
    ll left = t[t[p].ls].sum;
    if (left >= k) return kth(t[p].ls, l, mid, k);
    else return kth(t[p].rs, mid + 1, r, k - left);
}
int main() {
    std::cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
    }
    build(root[1], 1, n);
    while (m--) {
        int opt, x, y, p, q, k;
        std::cin >> opt;
        if (opt == 0) {
            std::cin >> p >> x >> y;
            split(root[p], root[++idx], 1, n, x, y);
        } else if (opt == 1) {
            std::cin >> p >> q;
            root[p] = merge(root[p], root[q], 1, n);
        } else if (opt == 2) {
            std::cin >> p >> x >> q;
            modify(root[p], 1, n, q, x);
        } else if (opt == 3) {
            std::cin >> p >> x >> y;
            std::cout << query(root[p], 1, n, x, y) << '\n';
        } else if (opt == 4) {
            std::cin >> p >> k;
            if (t[root[p]].sum < k) std::cout << "-1\n";
            else std::cout << kth(root[p], 1, n, k) << '\n';
        }
    }
    return 0;
}
```

线段树套线段树

常用作在线处理多维度信息。

例题

Luogu 3810 **陌上花开**

> 有 n 个元素，第 i 个元素有 a_i, b_i, c_i 三个属性，设 $f(i)$ 表示满足 $a_j \leq a_i$ 且 $b_j \leq b_i$ 且 $c_j \leq c_i$ 且 $j \neq i$ 的 j 的数量
>
> 对于 $d \in [0, n)$，求 $f(i) = d$ 的数量。

明显的多维度，可以使用线段树套线段树完成。

参考代码 & 模板

```
int n, k;
struct Val {
    int a,b,c;
    bool operator < (const Val &nxt) const {
        return a < nxt.a;
    }
}e[N];
struct SegmentTree
{
    int l, r;
    int sum;
}t1[N << 2], t2[N * 80];
int _1st_tot, _2nd_tot;
int _1st_root, _2nd_root[N * 80];
void _2nd_modify(int &p, int l, int r, int x)
{
    if (!p) p = ++_2nd_tot;
    t2[p].sum += 1;
    if (l == r) return;
    int mid = l + r >> 1;
    if (x <= mid) _2nd_modify(t2[p].l, l, mid, x);
    else _2nd_modify(t2[p].r, mid + 1, r, x);
}
void _1st_modify(int &p, int l, int r, int x1, int x2)
{
    if (!p) p = ++_1st_tot;
    _2nd_modify(_2nd_root[p], 1, k, x2);
    if(l == r) return;
    int mid = l + r >> 1;
    if (x1 <= mid) _1st_modify(t1[p].l, l, mid, x1, x2);
    else _1st_modify(t1[p].r, mid + 1, r, x1, x2);
}
int _2nd_query(int p, int l, int r, int ql, int qr)
{
    if (!p) return 0;
    if (ql <= l && r <= qr) return t2[p].sum;
    int mid = l + r >> 1, res = 0;
    if (ql <= mid) res += _2nd_query(t2[p].l, l, mid, ql, qr);
    if (qr > mid) res += _2nd_query(t2[p].r, mid + 1, r, ql, qr);
```

```
    return res;
}
int _1st_query(int p, int l, int r, int ql, int qr, int x)
{
    if (!p) return 0;
    if (ql <= l && r <= qr) return _2nd_query(_2nd_root[p], 1, k, 1, x);
    int mid = l + r >> 1, res = 0;
    if (ql <= mid) res += _1st_query(t1[p].l, l, mid, ql, qr, x);
    if (qr > mid) res += _1st_query(t1[p].r, mid + 1, r, ql, qr, x);
    return res;
}
int ans[N];
int main()
{
    std::cin >> n >> k;
    for(int i = 1; i <= n; i++) {
        std::cin >> e[i].a >> e[i].b >> e[i].c;
    }
    std::sort(e + 1, e + 1 + n);
    int last = 1;
    for(int i = 1; i <= n; i++) {
        _1st_modify(_1st_root, 1, k, e[i].b, e[i].c);
        if(e[i+1].a != e[i].a) {
            for(int j = last; j <= i; j++)
                ans[_1st_query(_1st_root, 1, k, 1, e[j].b, e[j].c) - 1]++;
            last = i + 1;
        }
    }
    for(int i = 0; i < n; i++) std::cout << ans[i] << '\n';
    return 0;
}
```

树状数组套权值线段树

静态区间 k 小值（POJ 2104 K-th Number） 的问题可以用 **可持久化权值线段树** 在 $O(N \log N)$ 的时间复杂度内解决。

如果还要求支持一种操作：单点修改某一位上的值，就需要用到 **树状数组套权值线段树** 。

不使用树状数组套权值线段树，修改的时间复杂度是 $O(N \log N)$ ，查询的时间复杂度是 $O(\log N)$ 。

其中修改的时间复杂度不可承受。

使用树状数组套权值线段树，修改的时间复杂度是 $O(\log^2 N)$ ，查询的时间复杂度是 $O(\log^2 N)$ 。

总时间复杂度 $O(N \log^2 N)$ ，总空间复杂度 $O(N \log^2 N)$ 。

例题

$Luogu\ P3380$ **树套树**

> 维护一个有序数列，其中需要提供以下操作：
>
> 1. 查询 k 在区间内的排名
> 2. 查询区间内排名为 k 的值

3. 修改某一位置上的数值
4. 查询 k 在区间内的前驱（前驱定义为严格小于 x，且最大的数，若不存在输出 -2147483647）
5. 查询 k 在区间内的后继（后驱定义为严格大于 x，且最小的数，若不存在输出 2147483647）

参考代码 & 模板

```
int n, m;
int a[N];
struct Query {
    int opt;
    int l,r;
    int k;
}q[N];
int disc[N * 10], tot;
void discrete() {
    std::sort(disc + 1,disc + 1 + tot);
    tot = std::unique(disc + 1,disc + 1 + tot) - disc - 1;
}
int get(int x) {
    return std::lower_bound(disc + 1, disc + 1 + tot, x) - disc;
}
struct Tree {
    int l,r;
    int sum;
}t[N * 600];
int idx, root[N];
void pushup(int p) {
    t[p].sum = t[t[p].l].sum + t[t[p].r].sum;
}
void change(int &p, int l, int r, int x, int v) {
    if (!p) p = ++idx;
    if (l == r) {t[p].sum += v; return;}
    int mid = l + r >> 1;
    if (x <= mid) change(t[p].l, l, mid, x, v);
    else change(t[p].r, mid + 1, r, x, v);
    pushup(p);
}
void change(int x, int y) {
    int now = get(a[x]);
    for (; x <= n; x += x & -x) change(root[x], 1, tot, now, y);
}
int n1, n2, rt1[N], rt2[N];
int query2(int l, int r, int x) {
    if (l == r) return 0;
    int sum = 0, mid = l + r >> 1;
    if (x <= mid) {
        for (int i = 1; i <= n1; i++) rt1[i] = t[rt1[i]].l;
        for (int i = 1; i <= n2; i++) rt2[i] = t[rt2[i]].l;
        return query2(l, mid, x);
    }
    else {
        for (int i = 1; i <= n1; i++) sum -= t[t[rt1[i]].l].sum;
```

```
        for (int i = 1; i <= n2; i++) sum += t[t[rt2[i]].l].sum;
        for (int i = 1; i <= n1; i++) rt1[i] = t[rt1[i]].r;
        for (int i = 1; i <= n2; i++) rt2[i] = t[rt2[i]].r;
        return sum + query2(mid + 1, r, x);
    }
}
int query1(int x, int y, int k) {
    n1 = n2 = 0;
    for (; x; x -= x & -x) rt1[++n1] = root[x];
    for (; y; y -= y & -y) rt2[++n2] = root[y];
    return query2(1, tot, k) + 1;
}
int ask2(int l, int r, int x) {
    if (l == r) return l;
    int s = 0,mid = l + r >> 1;
    for (int i = 1; i <= n1; i++) s -= t[t[rt1[i]].l].sum;
    for (int i = 1; i <= n2; i++) s += t[t[rt2[i]].l].sum;
    if (s >= x) {
        for (int i = 1; i <= n1; i++) rt1[i] = t[rt1[i]].l;
        for (int i = 1; i <= n2; i++) rt2[i] = t[rt2[i]].l;
        return ask2(l, mid, x);
    }
    else {
        for (int i = 1; i <= n1; i++) rt1[i] = t[rt1[i]].r;
        for (int i = 1; i <= n2; i++) rt2[i] = t[rt2[i]].r;
        return ask2(mid + 1, r, x - s);
    }
}
int ask1(int x, int y, int k) {
    n1 = n2 = 0;
    for (; x; x -= x & -x) rt1[++n1] = root[x];
    for (; y; y -= y & -y) rt2[++n2] = root[y];
    return ask2(1, tot, k);
}
int main() {
    std::cin >> n >> m;
    for (int i = 1; i <= n; i++) std::cin >> a[i];
    for (int i = 1; i <= n; i++) disc[++tot] = a[i];
    for (int i = 1; i <= m; i++) {
        std::cin >> q[i].opt >> q[i].l >> q[i].r;
        if (q[i].opt != 3) std::cin >> q[i].k;
        else disc[++tot] = q[i].r;
        if (q[i].opt == 4 || q[i].opt == 5) disc[++tot] = q[i].k;
    }
    discrete();
    for (int i = 1; i <= n; i++) change(i, 1);
    for (int i = 1; i <= m; i++) {
        int opt, l, r, k;
        opt = q[i].opt, l = q[i].l, r = q[i].r, k = q[i].k;
        if (opt == 1) {
            int now = get(k);
            std::cout << query1(l - 1, r, now)<<'\n';
        }
        else if (opt == 2) {
            std::cout << disc[ask1(l - 1, r, k)]<<'\n';
        }
```

```
        else if (opt == 3) {
            change(l, -1);
            a[l]=r;
            change(l, 1);
        }
        else if (opt == 4) {
            int now = get(k);
            int rnk = query1(l - 1, r, now);
            if(rnk == 1) std::cout << "-2147483647\n";
            else std::cout << disc[ask1(l - 1, r, rnk - 1)]<<'\n';
        }
        else if (opt == 5) {
            int now = get(k);
            if(now == tot) {std::cout << "2147483647\n"; continue;}
            int rnk = query1(l - 1, r, now + 1) - 1;
            if(rnk == r - l + 1) std::cout << "2147483647\n";
            else std::cout << disc[ask1(l - 1, r, rnk + 1)] << '\n';
        }
    }
    return 0;
}
```

扫描线

扫描线一般运用在图形上面，它和它的字面意思十分相似，就是一条线在整个图上扫来扫去，它一般被用来解决图形面积，周长，以及二维数点等问题。

面积问题

```
using ll = long long;
struct SegmentTree {
    int l, r;
    ll sum, val;
}t[N << 4];
struct Segment {
    int l, r, h, val;
    bool operator < (const Segment &k) const
    {
        return h < k.h;
    }
}seg[N << 3];
int n;
ll x[N << 3];
ll ans;
void build(int p, int l, int r) {
    t[p].l = l, t[p].r = r;
    t[p].sum = 0, t[p].val = 0;
    if(l == r) return;
    int mid = l + r >> 1;
    build(p << 1, l, mid);
    build(p << 1 | 1, mid + 1, r);
}
void pushup(int p) {
```

```
    int l = t[p].l, r = t[p].r;
    if(t[p].sum) t[p].val = x[r+1] - x[l];
    else t[p].val = t[p << 1].val + t[p << 1 | 1].val;
}
void update(int p,int l,int r, int val) {
    int ll = t[p].l,rr = t[p].r;
    if(x[rr+1] <= l || x[ll] >= r) return;
    if(x[rr+1] <= r && x[ll] >= l)
    {
        t[p].sum += val;
        pushup(p);
        return;
    }
    int mid = l + r >> 1;
    update(p << 1, l, r, val);
    update(p << 1 | 1, l, r, val);
    pushup(p);
}
int main() {
    std::cin >> n;
    for(int i = 1; i <= n; i++) {
        int x1, y1, x2, y2;
        std::cin >> x1 >> y1 >> x2 >> y2;
        x[2 * i - 1] = x1; x[2 * i] = x2;
        seg[2 * i - 1] = (Segment){x1, x2, y1, 1};
        seg[2 * i] = (Segment){x1, x2, y2, -1};
    }
    n *= 2;
    std::sort(seg + 1, seg + n + 1);
    std::sort(x + 1, x + n + 1);
    int cnt = std::unique(x + 1, x + n + 1) - x - 1;
    build(1, 1, cnt - 1);
    for(int i = 1; i < n; i++) {
        update(1, seg[i].l, seg[i].r, seg[i].val);
        ans += t[1].val * (seg[i + 1].h - seg[i].h);
    }
    std::cout << ans;
    return 0;
}
```

周长问题

```
#include <bits/stdc++.h>
const int N = 5e3 + 5;
typedef long long ll;
int n;
ll ans;
struct Line {
    int x1, x2, y;
    int is;
    bool operator < (const Line & b) const {
        return (y == b.y) ? is > b.is : y < b.y;
    }
}line1[N << 1], line2[N << 1];
int rowx[N << 1], rowy[N << 1], totx, toty;
```

```
void add(int x1, int x2, int y, int is, int it, Line* line) {
    line[it] = {x1, x2, y, is};
}
void discrete(int* row, int& tot) {
    std::sort(row + 1, row + 1 + n);
    tot = std::unique(row + 1, row + 1 + n) - row - 1;
}
int get(int x, int* row, int tot) {
    return std::lower_bound(row + 1, row + 1 + tot, x) - row;
}
struct SegmentTree {
    int l, r;
    int cnt, sum;
}t1[N << 3], t2[N << 3];
void update(SegmentTree* t, int p, int* row) {
    if(t[p].cnt) {
        t[p].sum = row[t[p].r + 1] - row[t[p].l];
    }
    else {
        t[p].sum = t[p << 1].sum + t[p << 1 | 1].sum;
    }
}
void build(SegmentTree* t, int p, int l, int r) {
    t[p].l = l, t[p].r = r;
    if(l == r) return;
    int mid = l + r >> 1;
    build(t, p << 1, l, mid);
    build(t, p << 1 | 1, mid + 1, r);
}
void change(SegmentTree* t, int p, int l, int r, int v, int* row) {
    if(l <= t[p].l && r >= t[p].r) {
        t[p].cnt += v;
        update(t, p, row);
        return;
    }
    int mid = t[p].l + t[p].r >> 1;
    if(l <= mid) change(t, p << 1, l, r, v, row);
    if(r > mid) change(t, p << 1 | 1, l, r, v, row);
    update(t, p, row);
}
int main() {
    std::cin >> n;
    for (int i = 1; i <= n; i++) {
        int x1, y1, x2, y2;
        std::cin >> x1 >> y1 >> x2 >> y2;
        add(x1, x2, y1, 1, 2 * i - 1, line1);
        add(x1, x2, y2, -1, 2 * i, line1);
        add(y1, y2, x1, 1, 2 * i - 1, line2);
        add(y1, y2, x2, -1, 2 * i, line2);
        rowx[2 * i - 1] = x1, rowx[2 * i] = x2;
        rowy[2 * i - 1] = y1, rowy[2 * i] = y2;
    }
    n <<= 1;
    std::sort(line1 + 1, line1 + 1 + n);
    std::sort(line2 + 1, line2 + 1 + n);
    discrete(rowx, totx), discrete(rowy, toty);
```

```
    build(t1, 1, 1, totx - 1), build(t2, 1, 1, toty - 1);
    ll tmp1 = 0, tmp2 = 0;
    for (int i = 1; i <= n; i++) {
        int l1 = get(line1[i].x1, rowx, totx), r1 = get(line1[i].x2, rowx, totx);
        int l2 = get(line2[i].x1, rowy, toty), r2 = get(line2[i].x2, rowy, toty);
        change(t1, 1, l1, r1 - 1, line1[i].is, rowx), change(t2, 1, l2, r2 - 1,
line2[i].is, rowy);
        ans += abs(t1[1].sum - tmp1) + abs(t2[1].sum - tmp2);
        tmp1 = t1[1].sum, tmp2 = t2[1].sum;
    }
    std::cout << ans;
    return 0;
}
```

分块

在维护较为复杂的信息（尤其是不满足区间可加、可减性的信息）时，树状数组和线段树显得吃力。这个时候用 **分块** 可以简单、直观的解决。

分块的基本思想是通过适当的划分，预处理一部分信息并保存下来，用空间换取时间，达到时空平衡。

维护数据的分块

例题 1 （区间修改 区间求和）

> 给定长度为 N 的数列 A，然后输入 Q 行操作指令。
>
> 第一类指令形如 "$C\ l\ r\ d$"，表示把数列中第 $l-r$ 个数都加 d。
>
> 第一类指令形如 "$Q\ l\ r$"，表示询问数列中第 $l-r$ 个数的和。

把数列 A 分成若干个长度不超过 $\lfloor\sqrt{N}\rfloor$ 的段，其中第 i 段左端点为 $(i-1)\lfloor\sqrt{N}\rfloor+1$，右端点为 $min(i\lfloor\sqrt{N}\rfloor,N)$。

另外，预处理出数组 sum，其中 $sum[i]$ 表示第 i 段的区间和。设 $add[i]$ 表示第 i 段的“增量标记”，起初 $add[i]=0$。

对于指令 "$C\ l\ r\ d$":

1. 若 l r 同时处于第 i 段内，直接把 $A[l],A[l+1],\ldots,A[r]$ 都加 d，同时令 $sum[i]+=d*(r-l+1)$。
2. 否则，设 l 处于第 p 段，r 处于第 q 段。
    1. 对于 $i\in[p+1,q-1]$，令 $add[i]+=d$。
    2. 对于开头、结尾不足一整段的两部分，按照第一种情况朴素更新。

对于指令 "$Q\ l\ r$":

1. 若 l r 同时处于第 i 段内，则 $A[l]+A[l+1]+\ldots+A[r]+(r-l+1)*add[i]$ 就是答案。
2. 否则，设 l 处于第 p 段，r 处于第 q 段。
    1. 对于 $i\in[p+1,q-1]$，令 $ans+=sum[i]+add[i]*len[i]$，$len[i]$ 表示第 i 段的长度。
    2. 对于开头、结尾不足一整段的两部分，按照第一种情况朴素累加。

为什么是分成 $\lfloor\sqrt{N}\rfloor$ 段。

若每段有 s 个元素，对于查询：当 $l\ r$ 在同一块内，直接暴力求和即可，因为块长为 s，最坏时间复杂度是 $O(s)$；当 $l\ r$ 不在同一块内，根据计算方法可知最坏时间复杂度是 $O(N/s+s)$。修改同理。

由均值不等式可知 $N/s+s$ 最小时，$s=\sqrt{N}$。

参考代码 & 模板

```
using ll = long long;
int n, m;
int a[N];
int B, L[N], R[N]; // 块数 每块的左边界 每块的右边界
int pos[N]; // 所在的块编号
ll sum[N], add[N]; // 当前块内元素的总和 当前块的懒标记

void change(int l, int r, int d) {
    // l 所在块 r 所在块
    int p = pos[l], q = pos[r];
    // 若处在相同块 朴素处理
    if (p == q) {
        for (int i = l; i <= r; i++) a[i] += d;
        sum[p] += d * (r - l + 1);
    }
    // 若不在相同块 大段维护 局部朴素
    else {
        for (int i = p + 1; i <= q - 1; i++) add[i] += d;
        for (int i = l; i <= R[p]; i++) a[i] += d;
        for (int i = L[q]; i <= r; i++) a[i] += d;
        sum[p] += d * (R[p] - l + 1);
        sum[q] += d * (r - L[q] + 1);
    }
}
ll query(int l, int r) {
    // l 所在块 r 所在块
    int p = pos[l], q = pos[r];
    ll ans = 0;
    // 若处在相同块 朴素处理
    if (p == q) {
        for (int i = l; i <= r; i++) ans += a[i];
        // 当前块的懒标记 add[] 也要加上
        ans += add[p] * (r - l + 1);
    }
    // 若不在相同块 大段维护 局部朴素
    else {
        for (int i = p + 1; i <= q - 1; i++) ans += sum[i] + add[i] * (R[i] -
L[i] + 1);
        for (int i = l; i <= R[p]; i++) ans += a[i];
        ans += add[p] * (R[p] - l + 1);
        for (int i = L[q]; i <= r; i++) ans += a[i];
        ans += add[q] * (r - L[q] + 1);
    }
    return ans;
}
int main() {
    std::cin >> n >> m;
```

```
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
    }
    B = sqrt(n);
    for (int i = 1; i <= B; i++) {
        L[i] = (i - 1) * B + 1;
        R[i] = i * B;
    }
    // 如果尾部还有未维护到的区域
    if (R[B] < n) {
        B++;
        L[B] = R[B - 1] + 1, R[B] = n;
    }
    for (int i = 1; i <= B; i++) {
        for (int j = L[i]; j <= R[i]; j++) {
            pos[j] = i;
            sum[i] += a[j];
        }
    }
    while (m--) {
        int opt;
        int l, r, d;
        std::cin >> opt;
        if (opt == 1) {
            std::cin >> l >> r >> d;
            change(l, r, d);
        } else if (opt == 2) {
            std::cin >> l >> r;
            std::cout << query(l, r) << '\n';
        }
    }
    return 0;
}
```

例题2（维护众数）

$Luogu\ P4168$ **蒲公英**

> 在乡下的小路旁种着许多蒲公英，而我们的问题正是与这些蒲公英有关。
>
> 为了简化起见，我们把所有的蒲公英看成一个长度为 n 的序列 $a_1, a_2, \ldots, a_n$，其中 a_i 为一个正整数，表示第 i 棵蒲公英的种类编号。
>
> 而每次询问一个区间 $[l, r]$ 你需要回答区间里出现次数最多的是哪种蒲公英，如果有若干种蒲公英出现次数相同，则输出种类编号最小的那个。
>
> 注意，你的算法必须是在线的。

本题是经典的在线求区间众数的问题，因为众数不具有“区间可加性”（已知序列 a 中区间 $[x, y]$ 的众数和区间 $[y+1, z]$ 的众数，不能直接得到区间 $[x, z]$ 的众数），所以用树状数组和线段树维护非常困难。故常用分块。

这个时候把序列 a 分成 T 块，每块的长度就是 $L = N/T$ 。

同样，我们把 $[l, r]$ 分成三部分。

而序列 a 在区间 $[l, r]$ 中的众数只可能来自于以下两种情况：

1. 区间 $[L[p+1], R[q-1]]$ 的众数。
2. 出现在 $[l, L)$ 和 $(R, r]$ 之间的数。

方法一

预处理出所有以“段边界”为端点的区间 $[L, R]$ 中每个数出现的次数，以及区间众数，一共有 $O(T^2)$ 个，并且保存每个数出现的次数需要长度为 $O(N)$ 的数组 （记为 $cnt_{L,R}$）

对于每个询问中的 $[l, L)$ 与 $(R, r]$，可以通过朴素扫描，在数组 $cnt_{L,R}$ 的基础上累加次数，从而更新答案。回答询问后再进行一次朴素扫描，把数组复原。

这个算法的时间为 $O(NT^2 + MN/T)$，空间为 $O(NT^2)$。设 N M 相同数量级，由均值不等式得 $T = \sqrt[3]{N}$，此时时间复杂度为 $O(N^{\frac{5}{3}})$。

通常需要根据时间复杂度来指定块数/块长。

```
int n, m;
int a[N], b[N];
// 离散化
int disc[N], tot;
void discrete() {
    std::sort(disc + 1, disc + 1 + tot);
    tot = std::unique(disc + 1, disc + 1 + tot) - disc - 1;
}
int get(int x) {
    return std::lower_bound(disc + 1, disc + 1 + tot, x) - disc;
}

int f[42][42], d[42][42];
int c[42][42][N];
int l[N], r[N];
int pos[N];
int cnt,num;
int ans;
int B,size;
int L,R;
void update(int j) {
    c[L][R][b[j]]++;
    if (c[L][R][b[j]] > cnt || c[L][R][b[j]] == cnt && b[j] < num)
        cnt = c[L][R][b[j]], num = b[j];
}

int main() {
    std::cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        std::cin >> a[i];
        disc[++tot] = a[i];
    }
    discrete();
    for (int i = 1; i <= n; i++) {
        b[i] = get(a[i]);
    }
    // 计算块数
    B = (int)pow(n, 1.0 / 3.0);
    // 如果满足大于0块 则计算每块块长
```

```
    if (B) size = n / B;
    // 计算左右边界
    for (int i = 1; i <= B; i++) {
        l[i] = (i - 1) * size + 1;
        r[i] = i * size;
    }
    // 处理末尾区域
    if (r[B] < n) l[++B] = r[B] + 1, r[B] = n;
    // 计算所处块
    for (int i = 1; i <= B; i++) {
        for (int j = l[i]; j <= r[i]; j++) {
            pos[j] = i;
        }
    }
    // 计算大区间众数
    // c[i][j][num] 表示数字 num 在 i 到 j 块里出现的次数
    // f[i][j] 表示 i 到 j 块里出先最多数字的出现次数 d[i][j] 表示该数字
    for (int i = 1; i <= B; i++) {
        for (int j = i; j <= B; j++) {
            for (int k = l[i]; k <= r[j]; k++) c[i][j][b[k]]++;
            for (int k = 1; k <= tot; k++) {
                if (f[i][j] < c[i][j][k] || f[i][j] == c[i][j][k] && d[i][j] > k)
                    f[i][j] = c[i][j][k], d[i][j] = k;
            }
        }
    }
    // 询问
    for (int i = 1; i <= m; i++) {
        int x, y, ll, rr;
        std::cin >> x >> y;
        x = (x + ans - 1) % n + 1; y = (y + ans - 1) % n + 1;
        if (x > y) std::swap(x, y);
        ll = pos[x], rr = pos[y];
        if (ll + 1 <= rr - 1) L = ll + 1, R = rr - 1;
        else L = R = 0;
        cnt = f[L][R], num = d[L][R];
        if(ll == rr) {
            for (int j = x; j <= y; j++)
                update(j);
            // 恢复原状
            for (int j = x; j <= y; j++)
                c[L][R][b[j]]--;
        }
        else {
            for (int j = x;j <= r[ll]; j++)
                update(j);
            for (int j = l[rr]; j <= y; j++)
                update(j);
            // 恢复原状
            for (int j = x; j <= r[ll]; j++)
                c[L][R][b[j]]--;
            for (int j = l[rr]; j <= y; j++)
                c[L][R][b[j]]--;
        }
        ans = disc[num];
        std::cout << ans << '\n';
```

```
    }
    return 0;
}
```

方法二

在预处理时，只保存所有以“段边界”为端点的区间 $[L,R]$ 的众数。另外，对每个数值建立一个 STL vector，按顺序保存该数值在序列 a 中每次出现的位置。

对于每个询问，扫描 $[l,L)$ 和 $(R,r]$ 中的每个数 x，在对应的 vector 里二分查找即可得到 x 在 $[l,r]$ 中出现的次数，从而更新答案。

这个算法的时间复杂度为 $O(NT+MN/T*\log N)$，空间为 $O(T^2)$。应取 $T=\sqrt{N\log N}$，时间复杂度则为 $O(N\sqrt{N\log N})$。

维护询问的分块（莫队）（离线算法）

普通莫队

时间复杂度一般为 $O(N\log N)$

例题

$Luogu\ P2709$ **小B的询问**

> 小B 有一个长为 n 的整数序列 a，值域为 $[1,k]$。
>
> 他一共有 m 个询问，每个询问给定一个区间 $[l,r]$，求 $\sum_{i=1}^{k}c_i^2$
>
> 其中 c_i 表示数字 i 在 $[l,r]$ 中的出现次数。

参考代码 & 模板

```
int n, m, k;
int B;
int a[N], c[N];
int sum;
int ans[N];
struct Query
{
    int l, r, id;
    bool operator < (const Query &b) const {
        // 不在同一块内 按左端点排序
        if(l / B != b.l / B) return l < b.l;
        // 在同一块内，根据块编号的奇偶按右端点排序
        // 奇数块按右端点从小到大 偶数块按右端点从大到小
        return ((l / B) & 1) ? (r < b.r) : (r > b.r);
    }
}q[N];
void add(int x) {
    sum -= c[x] * c[x];
    c[x]++;
    sum += c[x] * c[x];
}
void del(int x) {
```

```
    sum -= c[x]*c[x];
    c[x]--;
    sum += c[x]*c[x];
}
int main() {
    std::cin >> n >> m >> k;
    B = sqrt(n);
    for (int i = 1; i <= n; i++) std::cin >> a[i];
    for (int i = 1; i <= m; i++) {
        std::cin >> q[i].l >> q[i].r;
        q[i].id = i;
    }
    std::sort(q + 1,q + 1 + m);
    for (int i = 1, l = 1, r = 0; i <= m; i++) {
        // 需要严格按照以下顺序
        while (l > q[i].l) add(a[--l]);
        while (r < q[i].r) add(a[++r]);
        while (l < q[i].l) del(a[l++]);
        while (r > q[i].r) del(a[r--]);
        ans[q[i].id] = sum;
    }
    for(int i = 1; i <= m; i++) std::cout << ans[i] << '\n';
    return 0;
}
```

带修改莫队

例题

$Luogu\ P1903$ 数颜色/维护队列

> 墨墨购买了一套 N 支彩色画笔（其中有些颜色可能相同），摆成一排，你需要回答墨墨的提问。墨墨会向你发布如下指令：
>
> Q L R 代表询问你从第 L 支画笔到第 R 支画笔中共有几种不同颜色的画笔
>
> R P C 把第 P 支画笔替换为颜色 C 。

参考代码 & 模板

推荐以 $N^{\frac{2}{3}}$ 为块长。

```
int n, m;
int B;
int c[N];
int cnt[N],sum;
struct Query {
    int l, r, tim, id;
    bool operator < (const Query &b) const {
        if(l/B != b.l/B) return l < b.l;
        if(r/B != b.r/B) return r < b.r;
        return tim < b.tim;
    }
}q[N];
int qi, qtim;
struct Operation {
```

```
    int pos,x;
}p[N];
int ans[N];
void add(int x) {
    if (!cnt[x]) sum++;
    cnt[x]++;
}
void del(int x) {
    cnt[x]--;
    if (!cnt[x]) sum--;
}
int main() {
    std::cin >> n >> m;
    B = pow(n, 0.666);
    for(int i = 1; i <= n; i++) {
        std::cin >> c[i];
    }
    for(int i = 1; i <= m; i++) {
        char opt;int l, r;
        std::cin >> opt >> l >> r;
        if(opt == 'Q') {
            q[++qi]={l, r, qtim, qi};
        }
        else if(opt == 'R'){
            p[++qtim] = {l,r};
        }
    }
    std::sort(q + 1, q + 1 + qi);
    for (int i = 1, l = 1, r = 0, x = 0; i <= qi; i++) {
        while (l > q[i].l) add(c[--l]);
        while (r < q[i].r) add(c[++r]);
        while (l < q[i].l) del(c[l++]);
        while (r > q[i].r) del(c[r--]);
        while (x < q[i].tim) {
            int pos = p[++x].pos;
            if (l <= pos && pos <= r) del(c[pos]), add(p[x].x);
            std::swap(c[pos], p[x].x);
        }
        while(x > q[i].tim) {
            int pos = p[x].pos;
            if(l <= pos && pos <= r) del(c[pos]), add(p[x].x);
            std::swap(c[pos], p[x--].x);
        }
        ans[q[i].id] = sum;
    }
    for (int i = 1;i <= qi; i++) std::cout << ans[i] << '\n';
    return 0;
}
```

树上莫队

例题

$Luogu\ P4074$ 糖果公园

> 糖果公园的结构十分奇特，它由 n 个游览点构成，每个游览点都有一个糖果发放处，我们可以依次将游览点编号为 1 至 n 。有 $n-1$ 条双向道路连接着这些游览点，并且整个糖果公园都是连通的，即从任何一个游览点出发都可以通过这些道路到达公园里的所有其它游览点。
>
> 糖果公园所发放的糖果种类非常丰富，总共有 m 种，它们的编号依次为 1 至 m 。每一个糖果发放处都只发放某种特定的糖果，我们用 C_i 来表示 i 号游览点的糖果。
>
> 来到公园里游玩的游客都不喜欢走回头路，他们总是从某个特定的游览点出发前往另一个特定的游览点，并游览途中的景点，这条路线一定是唯一的。他们经过每个游览点，都可以品尝到一颗对应种类的糖果。
>
> 大家对不同类型糖果的喜爱程度都不尽相同。 根据游客们的反馈打分，我们得到了糖果的美味指数， 第 i 种糖果的美味指数为 V_i 。另外，如果一位游客反复地品尝同一种类的糖果，他肯定会觉得有一些腻。根据量化统计，我们得到了游客第 i 次品尝某类糖果的新奇指数 W_i 。如果一位游客第 i 次品尝第 j 种糖果，那么他的愉悦指数 H 将会增加对应的美味指数与新奇指数的乘积，即 $V_j \times W_i$。这位游客游览公园的愉悦指数最终将是这些乘积的和。
>
> 当然，公园中每个糖果发放点所发放的糖果种类不一定是一成不变的。有时，一些糖果点所发放的糖果种类可能会更改（也只会是 m 种中的一种），这样的目的是能够让游客们总是感受到惊喜。
>
> 糖果公园的工作人员小 A 接到了一个任务，那就是根据公园最近的数据统计出每位游客游玩公园的愉悦指数。

参考代码 & 模板（带修改）

树上莫队基于欧拉序。

```
using ll = long long;
int n, m, q;
int B;
int v[N], w[N], c[N];
int hd[N], ver[N << 1], nxt[N << 1], tot;
void add(int x,int y) {
    ver[++tot] = y;
    nxt[tot] = hd[x], hd[x] = tot;
}
int son[N], size[N], fa[N], dep[N];
int in[N], out[N], rnk[N << 1], tim, top[N];
void dfs1(int x) {
    size[x] = 1;
    for (int i = hd[x]; i; i = nxt[i]) {
        int y = ver[i];
        if (y == fa[x]) continue;
        fa[y] = x;
        dep[y] = dep[x] + 1;
        dfs1(y);
        size[x] += size[y];
        if (size[y] > size[son[x]]) son[x] = y;
    }
}
```

```
void dfs2(int x,int t) {
    in[x] = ++tim; rnk[tim] = x; top[x] = t;
    if(son[x]) dfs2(son[x], t);
    for(int i = hd[x]; i; i = nxt[i]) {
        int y = ver[i];
        if(y == fa[x]) continue;
        if(y != son[x]) dfs2(y, y);
    }
    out[x] = ++tim; rnk[tim] = x;
}
int lca(int x, int y) {
    while(top[x] != top[y]) {
        if(dep[top[x]] < dep[top[y]]) std::swap(x, y);
        x = fa[top[x]];
    }
    return dep[x] < dep[y] ? x : y;
}
struct Query {
    int l, r, tim, id, lca;
    bool operator < (const Query &b) const {
        if(l / B != b.l / B) return l < b.l;
        if(r / B != b.r / B) return r < b.r;
        return tim < b.tim;
    }
}Q[N];
int qi, qtim;
struct Modify
{
    int tim, pos, val;
}P[N];
ll sum;
int cnt[N];
bool vis[N];
void add(int x)
{
    vis[x] ^= 1;
    if (vis[x]) sum += 1ll * w[++cnt[c[x]]] * v[c[x]];
    else sum -= 1ll * w[cnt[c[x]]--] * v[c[x]];
}
ll ans[N];
int main()
{
    std::cin >> n >> m >> q;
    B = pow(2 * n, 0.666);
    for (int i = 1; i <= m; i++) std::cin >> v[i];
    for (int i = 1; i <= n; i++) std::cin >> w[i];
    for (int i = 1; i <= n - 1; i++) {
        int x, y;
        std::cin >> x >> y;
        add(x, y); add(y, x);
    }
    dfs1(1); dfs2(1, 1);
    for (int i = 1; i <= n; i++) std::cin >> c[i];
    for (int i = 1; i <= q; i++) {
        int opt, x, y;
        std::cin >> opt >> x >> y;
```

```
        if (!opt) {
            P[++qtim] = {qtim, x, y};
        }
        else if (opt) {
            int LCA = lca(x, y);
            if(in[x] > in[y]) std::swap(x, y);
            if(LCA == x) {
                Q[++qi] = {in[x], in[y], qtim, qi, 0};
            }
            else {
                Q[++qi] = {out[x], in[y], qtim, qi, LCA};
            }
        }
    }
    std::sort(Q + 1, Q + 1 + qi);
    for (int i = 1,l = 1, r = 0, t = 0; i <= qi; i++) {
        while (l > Q[i].l) add(rnk[--l]);
        while (r < Q[i].r) add(rnk[++r]);
        while (l < Q[i].l) add(rnk[l++]);
        while (r > Q[i].r) add(rnk[r--]);
        while (t < Q[i].tim) {
            int pos = P[++t].pos;
            if (vis[pos]) {
                add(pos);
                std::swap(P[t].val, c[pos]);
                add(pos);
            }
            else std::swap(P[t].val, c[pos]);
        }
        while(t > Q[i].tim) {
            int pos = P[t].pos;
            if (vis[pos]) {
                add(pos);
                std::swap(P[t--].val, c[pos]);
                add(pos);
            }
            else std::swap(P[t--].val, c[pos]);
        }
        ans[Q[i].id] = sum;
        // 补上两点最近公共祖先的贡献
        if(Q[i].lca) ans[Q[i].id] += 1ll * w[cnt[c[Q[i].lca]] + 1] *
v[c[Q[i].lca]];
    }
    for (int i = 1; i <= qi; i++) std::cout << ans[i] << '\n';
    return 0;
}
```

回滚莫队

例题

$AT\_joisc2014\_c$ 歴史の研究

> 日记中记录了连续 N 天发生的事件，大约每天发生一件。
>
> 事件有种类之分。第 i 天发生的事件的种类用一个整数 X_i 表示，X_i 越大，事件的规模就越大。

> JOI 教授决定用如下的方法分析这些日记：
>
> 选择日记中连续的几天 $[L, R]$ 作为分析的时间段；
>
> 定义事件 A 的重要度 W_A 为 $A \times T_A$ ，其中 T_A 为该事件在区间 $[L, R]$ 中出现的次数。
>
> 现在，您需要帮助教授求出所有事件中重要度最大的事件是哪个，**并输出其重要度**。
>
> 多组询问。

过程

在这个问题中，在增加的过程中更新答案是很好实现的，但是**在删除的过程中更新答案**是不好实现的。

因为如果增加会影响答案，那么新答案必定是刚刚增加的数字的重要度，而如果删除过后区间重要度最大的数字改变，我们很难确定新的重要度最大的数字是哪一个。

所以，普通的莫队很难解决这个问题。

使用回滚莫队 / 不删除莫队可以解决。

- 对原序列进行分块，对询问按以左端点所属块编号升序为第一关键字，右端点升序为第二关键字的方式排序。
- 按顺序处理询问：
  - 如果询问左端点所属块 B 和上一个询问左端点所属块的不同，那么将莫队区间的左端点初始化为 B 的右端点加 1，将莫队区间的右端点初始化为 B 的右端点；
  - 如果询问的左右端点所属的块相同，那么直接扫描区间回答询问；
  - 如果询问的左右端点所属的块不同：
    - 如果询问的右端点大于莫队区间的右端点，那么不断扩展右端点直至莫队区间的右端点等于询问的右端点；
    - 不断扩展莫队区间的左端点直至莫队区间的左端点等于询问的左端点；
    - 回答询问；
    - 撤销莫队区间左端点的改动，使莫队区间的左端点回滚到 B 的右端点加 1 。

参考代码 & 模板

```
#include<bits/stdc++.h>
const int N(2e5+5);
int n;
int B, num;
int a[N], block[N];
int disc[N],tot;
void discrete() {
    std::sort(disc + 1, disc + 1 + tot);
    tot = std::unique(disc + 1, disc + 1 + tot) - disc - 1;
}
int get(int x) {
    return std::lower_bound(disc + 1, disc + 1 + tot, x) - disc;
}
int m;
struct Query {
    int l, r, id;
    bool operator < (const Query &b) const {
        if(block[l] != block[b.l]) return l < b.l;
```

```
        return r < b.r;
    }
}q[N];
int pos[N], p[N], mx_pos[N], back_pos[N];
int res, last;
int ans[N];
int cal(int l, int r) {
    int ans = 0;
    for (int i = l; i <= r; i++) {
        p[a[i]] = 0;
    }
    for (int i = l; i <= r; i++) {
        if (!p[a[i]]) p[a[i]] = i;
        else ans = std::max(ans, i - p[a[i]]);
    }
    return ans;
}
void add_r(int x, int p) {
    if (!pos[x]) back_pos[x] = mx_pos[x] = pos[x] = p;
    else {
        res = std::max(res, p - pos[x]);
        back_pos[x] = mx_pos[x] = p;
    }
}
void add_l(int x, int p) {
    if (!mx_pos[x]) mx_pos[x] = p;
    else res = std::max(res, mx_pos[x] - p);
}
void del(int x) {
    mx_pos[x]=back_pos[x];
}
int main() {
    std::cin >> n;
    B = sqrt(n);
    for(int i = 1; i <= n; i++) {
        std::cin >> a[i];
        block[i] = (i - 1) / B + 1;
    }
    num = block[n];
    for(int i = 1; i <= n; i++) {
        disc[++tot] = a[i];
    }
    discrete();
    for(int i = 1; i <= n; i++) {
        a[i] = get(a[i]);
    }
    std::cin >> m;
    for(int i = 1; i <= m; i++) {
        std::cin >> q[i].l >> q[i].r;
        q[i].id = i;
    }
    std::sort(q + 1, q + 1 + m);
    for (int i = 1, x = 1; i <= num; i++) {
        last = res = 0;
        for (int j = 1; j <= n; j++) back_pos[j] = mx_pos[j] = pos[j] = 0;
        int R = std::min(i * B, n), l = R + 1, r = R;
```

```
        for (; block[q[x].l] == i; x++) {
            if (block[q[x].l] == block[q[x].r]) {
                ans[q[x].id] = cal(q[x].l, q[x].r);
                continue;
            }
            while (r < q[x].r) add_r(a[++r], r);
            last = res;
            while (l > q[x].l) add_l(a[--l], l);
            ans[q[x].id] = res;
            while (l <= R) del(a[l++]);
            res = last;
        }
    }
    for (int i = 1; i <= m; i++) std::cout << ans[i] << '\n';
    return 0;
}
```

K-D Tree

k-D Tree(KDT , k-Dimension Tree) 是一种可以 高效处理 k 维空间信息 的数据结构。

在结点数 n 远大于 2^k 时，应用 k-D Tree 的时间效率很好。

在算法竞赛的题目中，一般有 $k = 2$ 。分析时间复杂度时，将认为 k 是常数。

构建 k-D Tree 时间复杂度是 $O(N \log N)$ 的。

例题

$Luogu\ P1429$ **平面最近点对**

> 给定平面上 n 个点，找出其中的一对点的距离，使得在这 n 个点的所有点对中，该距离为所有点对中最小的。

参考代码 & 模板

使用二进制分组 / 插入重构建树

时间复杂度 建树均摊 $O(N \log^2 N)$ 查询 $O(N^{1-\frac{1}{k}})$ 。

```
constexpr int N(2e5+5);
constexpr int LG(20);
constexpr double INF(2e18);
int n;
struct Point {
    double xy[2];
    int lc,rc;
    double L[2],R[2];
}t[N];
int b[N], cnt;
int rt[LG];
double ans = INF;
void pushup(int p)
{
```

```
    for (int k : {0, 1}) {
        t[p].L[k] = t[p].R[k] = t[p].xy[k];
        if (t[p].lc) {
            t[p].L[k] = std::min(t[p].L[k], t[t[p].lc].L[k]);
            t[p].R[k] = std::max(t[p].R[k], t[t[p].lc].R[k]);
        }
        if (t[p].rc) {
            t[p].L[k] = std::min(t[p].L[k], t[t[p].rc].L[k]);
            t[p].R[k] = std::max(t[p].R[k], t[t[p].rc].R[k]);
        }
    }
}
int build(int l, int r, int dep = 0) {
    int mid = l + r >> 1;
    std::nth_element(b+l, b+mid, b+r+1, [dep](int x, int y){return t[x].xy[dep] <
t[y].xy[dep];});
    int x = b[mid];
    if(l < mid) t[x].lc = build(l, mid - 1, dep ^ 1);
    if(r > mid) t[x].rc = build(mid+1, r, dep ^ 1);
    pushup(x);
    return x;
}
void append(int &p) {
    if (!p) return;
    b[++cnt] = p;
    append(t[p].lc); append(t[p].rc);
    p = 0;
}
double pw(double x) {
    return x * x;
}
double dis(int p) {
    double ret = 0;
    for (int k : {0, 1}) {
        ret += pw(t[p].xy[k] - t[0].xy[k]);
    }
    return ret;
}
double dis_area(int p) {
    if (!p) return INF;
    double ret = 0;
    for (int k : {0,1}) {
        ret += pw(std::max(t[p].L[k] - t[0].xy[k], 0.0)) + pw(std::max(t[0].xy[k]
- t[p].R[k], 0.0));
    }
    return ret;
}
void query(int p) {
    if (!p) return;
    ans = std::min(ans, dis(p));
    double dl = dis_area(t[p].lc), dr = dis_area(t[p].rc);
    if (dl < dr) {
        if(dl < ans) query(t[p].lc);
        if(dr < ans) query(t[p].rc);
    }
    else {
```

```
        if(dr < ans) query(t[p].rc);
        if(dl < ans) query(t[p].lc);
    }
}
int main() {
    std::cin >> n;
    for (int i = 1; i <= n; i++) {
        std::cin >> t[i].xy[0] >> t[i].xy[1];
        t[0] = {{t[i].xy[0],t[i].xy[1]}};
        for (int j = 0; j < LG; j++) query(rt[j]);
        b[cnt = 1] = i;
        for (int sz = 0; ; sz++) {
            if (!rt[sz]) {rt[sz] = build(1, cnt); break;}
            else append(rt[sz]);
        }
    }
    std::cout << std::fixed << std::setprecision(4) << sqrt(ans);
    return 0;
}
```

点分治

普通点分治（无修改操作）

```
const int N = 1e4+5;
const int INF = 1e9;
using ll = long long;
int n, m;
int hd[N], ver[N << 1], nxt[N << 1], edge[N << 1];
int tot;
bool vis[N];
void add(int x, int y, int e) {
    ver[++tot] = y;
    nxt[tot] = hd[x], hd[x] = tot;
    edge[tot] = e;
}
int sze[N], root;
int allpoint, minsze = INF;
void getroot(int x, int fa) {
    sze[x] = 1;
    int nowsze = 0;
    for(int i = hd[x]; i; i = nxt[i]) {
        int y = ver[i];
        if(y == fa || vis[y]) continue;
        getroot(y, x);
        sze[x] += sze[y];
        nowsze = std::max(nowsze, sze[y]);
    }
    nowsze = std::max(nowsze, allpoint-sze[x]);
    if(nowsze < minsze) root = x ,minsze = nowsze;
}
int q[N];
```

```
ll ans[N];
int dis[N], l, r, cnt;
int que[N];
void getdis(int x, int fa) {
    que[++cnt] = dis[x];
    for (int i = hd[x]; i; i = nxt[i])
    {
        int y = ver[i], e = edge[i];
        if (y == fa || vis[y]) continue;
        dis[y] = dis[x] + e;
        getdis(y, x);
    }
}
void cal(int x, int e, int c) {
    dis[x] = e; cnt = 0;
    getdis(x, 0);
    std::sort(que + 1, que + 1 + cnt);
    for (int i = 1; i <= m; i++) {
        l = 1, r = cnt;
        while (l < r) {
            if (que[l] + que[r] < q[i]) l++;
            else if (que[l] + que[r] > q[i]) r--;
            else ans[i] += 1ll * c, l++, r--;
        }
    }
}
void dfs(int x) {
    cal(x, 0, 1); vis[x] = 1;
    for (int i = hd[x]; i; i = nxt[i]) {
        int y = ver[i], e = edge[i];
        if (vis[y]) continue;
        cal(y, e, -1); allpoint = sze[y];
        minsze = INF; getroot(y, x);
        dfs(root);
    }
}
int main() {
    std::cin >> n >> m;
    for (int i = 1; i <= n - 1; i++) {
        int u, v, w;
        std::cin >> u >> v >> w;
        add(u, v, w); add(v, u, w);
    }
    for (int i = 1; i <= m; i++) std::cin >> q[i];
    allpoint = n;
    getroot(1, 0);
    dfs(root);
    for (int i = 1; i <= m; i++) {
        if (ans[i]) std::cout<<"AYE\n";
        else std::cout<<"NAY\n";
    }
    return 0;
}
```

动态点分治（带修改操作）

```
const int N = 1e5 + 5;
int n, m, t;
int w[N];
int hd[N], ver[N << 1], nxt[N << 1];
int tot;
void add(int x, int y) {
    ver[++tot] = y;
    nxt[tot] = hd[x], hd[x] = tot;
}

struct LCA {
    int dep[N],fa[N][20];
    void dfs(int x, int f) {
        for (int i = hd[x]; i; i = nxt[i]) {
            int y = ver[i];
            if (y == f) continue;
            dep[y] = dep[x] + 1;
            fa[y][0] = x;
            for (int j = 1; j <= t; j++)
                fa[y][j] = fa[fa[y][j - 1]][j - 1];
            dfs(y, x);
        }
    }
    int lca(int x, int y) {
        if (dep[x] > dep[y]) std::swap(x, y);
        for (int i = t; i >= 0; i--)
            if (dep[fa[y][i]] >= dep[x]) y = fa[y][i];
        if (x == y) return x;
        for (int i = t; i >= 0; i--)
            if (fa[x][i] != fa[y][i]) x = fa[x][i], y = fa[y][i];
        return fa[x][0];
    }
    int getdis(int x, int y) {
        return dep[x] + dep[y] - 2 * dep[lca(x, y)];
    }
}lca;

struct SegmentTree {
    int idx, root[N], sum[N * 40], lc[N * 40], rc[N * 40];
    void pushup(int p) {
        sum[p] = sum[lc[p]] + sum[rc[p]];
    }
    void change(int &p, int l, int r, int x, int v) {
        if (!p) p = ++idx;
        if (l == r) {sum[p]+=v;return;}
        int mid = l + r >> 1;
        if (x <= mid) change(lc[p], l, mid, x, v);
        else change(rc[p], mid + 1, r, x, v);
        pushup(p);
    }
    int query(int p,int l,int r,int ql,int qr) {
        if (!p) return 0;
        if (ql <= l && r <= qr) return sum[p];
```

```
        int mid = l + r >> 1, res = 0;
        if (ql <= mid) res += query(lc[p], l, mid, ql, qr);
        if (qr > mid) res += query(rc[p], mid + 1, r, ql, qr);
        return res;
    }
}sg, ch;

struct PointTree {
    int root, size[N], mxp[N], allpoint;
    bool vis[N];
    void findwc(int x, int fa) {
        size[x] = 1, mxp[x] = 0;
        for (int i = hd[x]; i; i = nxt[i]) {
            int y = ver[i];
            if (y == fa || vis[y]) continue;
            findwc(y, x);
            size[x] += size[y];
            mxp[x] = std::max(mxp[x], size[y]);
        }
        mxp[x] = std::max(mxp[x], allpoint - size[x]);
        if(mxp[x] < mxp[root]) root = x;
    }
    void getroot(int x, int sz) {
        mxp[root = 0] = allpoint = sz;
        findwc(x, -1);
        findwc(root, -1);
    }
    int fa[N], dep[N], dis[N][20];
    void build_sg(int x, int fa, int wc, int d) {
        sg.change(sg.root[wc], 0, n, d, w[x]);
        for (int i = hd[x]; i; i = nxt[i]) {
            int y = ver[i];
            if (y == fa || vis[y]) continue;
            build_sg(y, x, wc, d + 1);
        }
    }
    void build_ch(int x,int fa,int wc,int d) {
        ch.change(ch.root[wc], 0, n, d, w[x]);
        for (int i = hd[x]; i; i = nxt[i]) {
            int y = ver[i];
            if (y == fa || vis[y]) continue;
            build_ch(y, x, wc, d + 1);
        }
    }
    void build(int x) {
        vis[x] = 1;
        build_sg(x, -1, x, 0);
        for (int i = hd[x]; i; i = nxt[i]) {
            int y = ver[i];
            if (vis[y]) continue;
            getroot(y, size[y]);
            build_ch(y, x, root, 1);
            fa[root] = x, dep[root] = dep[x] + 1;
            build(root);
        }
    }
```

```
    void init() {
        getroot(1, n);
        build(root);
        lca.dfs(1, -1);
        for(int i = 1;i <= n; i++)
            for(int j = i; j; j = fa[j])
                dis[i][dep[i] - dep[j]] = lca.getdis(i, j);
    }
    void change(int x, int y) {
        sg.change(sg.root[x], 0, n, 0, y - w[x]);
        for(int i = x; fa[i]; i = fa[i]) {
            int d = dis[x][dep[x] - dep[fa[i]]];
            sg.change(sg.root[fa[i]], 0, n, d, y - w[x]);
            ch.change(ch.root[i], 0, n, d, y - w[x]);
        }
    }
    int query(int x,int y) {
        int ans = sg.query(sg.root[x], 0, n, 0, y);
        for(int i = x; fa[i]; i = fa[i]) {
            int d = dis[x][dep[x] - dep[fa[i]]];
            ans += sg.query(sg.root[fa[i]], 0, n, 0, y - d);
            ans -= ch.query(ch.root[i], 0, n, 0, y - d);
        }
        return ans;
    }
}pt;

int main() {
    std::cin >> n >> m;
    t = (int)(log(n)/log(2)) + 1;
    for (int i = 1; i <= n; i++) std::cin >> w[i];
    for (int i = 1; i <= n - 1; i++) {
        int u, v;
        std::cin >> u >> v;
        add(u, v); add(v, u);
    }
    pt.init();
    int last = 0;
    for (int i = 1; i <= m; i++) {
        int opt, x, y;
        std::cin >> opt >> x >> y;
        x ^= last,y ^= last;
        if (!opt) {
            last = pt.query(x, y);
            std::cout << last << '\n';
        }
        else if(opt) pt.change(x, y), w[x] = y;
    }
    return 0;
}
```

高精度模板

```
#include <bits/stdc++.h>
const double PI = acos(-1.0);
struct Complex
{
    double x, y;
    Complex(double _x = 0.0, double _y = 0.0) {
        x = _x; y = _y;
    }
    Complex operator+ (const Complex& b) const {
        return Complex(x + b.x, y + b.y);
    }
    Complex operator- (const Complex& b) const {
        return Complex(x - b.x, y - b.y);
    }
    Complex operator* (const Complex& b) const {
        return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
    }
};
void change(Complex y[], int len)
{
    for (int i = 1, j = len / 2; i < len - 1; i++) {
        if (i < j) std::swap(y[i], y[j]);
        int k = len / 2;
        while (j >= k) {
            j = j - k;
            k = k / 2;
        }
        if (j < k) j += k;
    }
}
void fft(Complex y[], int len, int on)
{
    change(y, len);
    for (int h = 2; h <= len; h <<= 1) {
        Complex wn(cos(on * 2 * PI /h), sin(on * 2 * PI / h));
        for (int j = 0; j < len; j += h) {
            Complex w(1, 0);
            for (int k = j; k < j + h / 2; k++) {
                Complex u = y[k];
                Complex t = w * y[k + h / 2];
                y[k] = u + t;
                y[k + h / 2] = u - t;
                w = w * wn;
            }
        }
    }
    if (on == -1) {
        for (int i = 0; i < len; i++) {
            y[i].x /= len;
        }
    }
}
```

```
class BigInt
{
#define Value(x, nega) ((nega) ? -(x) : (x))
#define At(vec, index) ((index) < vec.size() ? vec[(index)] : 0)
    static int absComp(const BigInt& lhs, const BigInt& rhs) {
        if (lhs.size() != rhs.size()) {
            return lhs.size() < rhs.size() ? -1 : 1;
        }
        for (int i = lhs.size() - 1; i >= 0; i--) {
            if (lhs[i] != rhs[i]) {
                return lhs[i] < rhs[i] ? -1 : 1;
            }
        }
        return 0;
    }
    using ll = long long;
    const static int Exp = 9;
    const static ll MOD = 1000000000;
    mutable std::vector<ll> val;
    mutable bool nega = false;
    void trim() const {
        while (val.size() && val.back() == 0) {
            val.pop_back();
        }
        if (val.empty()) {
            nega = false;
        }
    }
    int size() const {
        return val.size();
    }
    ll& operator[] (int index) const{
        return val[index];
    }
    ll& back() const {
        return val.back();
    }
    BigInt(int size, bool nega) : val(size), nega(nega) {}
    BigInt(const std::vector<ll>& val, bool nega) : val(val), nega(nega) {}

public:
    friend std::ostream& operator<< (std::ostream& os, const BigInt& n) {
        if (n.size()) {
            if (n.nega) {
                std::cout.put('-');
            }
            for (int i = n.size() - 1; i >= 0; i--) {
                if (i == n.size() - 1) {
                    std::cout << n[i];
                }
                else {
                    std::cout << std::setw(n.Exp) << std::setfill('0') << n[i];
                }
            }
        }
        else {
```

```
            std::cout.put('0');
        }
        return os;
    }
    friend BigInt operator+ (const BigInt& lhs, const BigInt& rhs) {
        BigInt ret(lhs);
        return ret += rhs;
    }
    friend BigInt operator- (const BigInt& lhs, const BigInt& rhs) {
        BigInt ret(lhs);
        return ret -= rhs;
    }
    BigInt(ll x = 0) {
        if (x < 0) {
            x = -x;
            nega = true;
        }
        while (x >= MOD) {
            val.push_back(x % MOD), x /= MOD;
        }
        if (x) {
            val.push_back(x);
        }
    }
    BigInt(const char* s) {
        int bound = 0, pos;
        if (s[0] == '-') {
            nega = true, bound = 1;
        }
        ll cur = 0, pow = 1;
        for (pos = strlen(s) - 1; pos >= Exp + bound - 1; pos -= Exp,
val.push_back(cur), cur = 0, pow = 1){
            for (int i = pos; i > pos - Exp; i--) {
                cur += (s[i] - '0') * pow;
                pow *= 10;
            }
        }
        for (cur = 0, pow = 1; pos >= bound; pos--) {
            cur += (s[pos] - '0') * pow;
            pow *= 10;
        }
        if (cur) {
            val.push_back(cur);
        }
    }
    BigInt& operator= (const char* s) {
        BigInt n(s);
        *this = n;
        return n;
    }
    BigInt& operator= (const ll x) {
        BigInt n(x);
        *this = n;
        return n;
    }
    friend std::istream& operator>> (std::istream& is, BigInt& n) {
```

```
        std::string s;
        is >> s;
        n = (char*)s.data();
        return is;
    }
    BigInt& operator+= (const BigInt& rhs) {
        const int cap = std::max(size(), rhs.size()) + 1;
        val.resize(cap);
        int carry = 0;
        for (int i = 0; i < cap - 1; i++) {
            val[i] = Value(val[i], nega) + Value(At(rhs, i), rhs.nega) + carry;
            carry = 0;
            if (val[i] >= MOD) {
                val[i] -= MOD;
                carry = 1;
            }
            else if (val[i] < 0) {
                val[i] += MOD;
                carry = -1;
            }
        }
        if ((val.back() = carry) == -1) {
            nega = true, val.pop_back();
            bool tailZero = true;
            for (int i = 0; i < cap - 1; i++) {
                if (tailZero && val[i]) {
                    val[i] = MOD - val[i], tailZero = false;
                }
                else {
                    val[i] = MOD - 1 - val[i];
                }
            }
        }
        trim();
        return *this;
    }
    friend BigInt operator- (const BigInt& rhs) {
        BigInt ret(rhs);
        ret.nega ^= 1;
        return ret;
    }
    BigInt operator-= (const BigInt& rhs) {
        rhs.nega ^= 1;
        *this += rhs;
        rhs.nega ^= 1;
        return *this;
    }
    friend BigInt operator* (const BigInt& lhs, const BigInt& rhs) {
        int len = 1;
        BigInt L = lhs, R = rhs;
        L.nega = lhs.nega ^ rhs.nega;
        while (len < 2 * lhs.size() || len < 2 * rhs.size()) len <<= 1;
        L.val.resize(len), R.val.resize(len);
        Complex x1[len], x2[len];
        for (int i = 0; i < len; i++) {
            Complex nx(L[i], 0.0), ny(R[i], 0.0);
```

```
            x1[i] = nx;
            x2[i] = ny;
        }
        fft(x1, len, 1);
        fft(x2, len, 1);
        for (int i = 0; i < len; i++) {
            x1[i] = x1[i] * x2[i];
        }
        fft(x1, len, -1);
        for (int i = 0; i < len; i++) {
            L[i] = int(x1[i].x + 0.5);
        }
        for (int i = 0; i < len; i++) {
            L[i + 1] += L[i] / MOD;
            L[i] %= MOD;
        }
        L.trim();
        return L;
    }
    friend BigInt operator* (const BigInt& lhs, const ll& x) {
        BigInt ret(lhs);
        bool negat = (x < 0);
        ll xx = (negat) ? -x : x;
        ret.nega ^= negat;
        ret.val.push_back(0);
        ret.val.push_back(0);
        for (int i = 0; i < ret.size(); i++) {
            ret[i] *= xx;
        }
        for (int i = 0; i < ret.size(); i++) {
            ret[i + 1] = ret[i] / MOD;
            ret[i] %= MOD;
        }
        ret.trim();
        return ret;
    }
    BigInt& operator*= (const BigInt &rhs) {
        return *this = *this * rhs;
    }
    BigInt& operator*= (const ll& x) {
        return *this = *this * x;
    }
    friend BigInt operator/ (const BigInt& lhs, const BigInt& rhs) {
        static std::vector<BigInt> powTwo{BigInt(1)};
        static std::vector<BigInt> estimate;
        estimate.clear();
        if (absComp(lhs, rhs) < 0) {
            return BigInt();
        }
        BigInt cur = rhs;
        int cmp;
        while ((cmp = absComp(cur, lhs)) <= 0) {
            estimate.push_back(cur);
            cur += cur;
            if (estimate.size() >= powTwo.size()) {
                powTwo.push_back(powTwo.back() + powTwo.back());
```

```
            }
        }
        if (cmp == 0) {
            return BigInt(powTwo.back().val, lhs.nega ^ rhs.nega);
        }
        BigInt ret = powTwo[estimate.size() - 1];
        cur = estimate[estimate.size() - 1];
        for (int i = estimate.size(); i >= 0 && cmp != 0; i--) {
            if ((cmp = absComp(cur + estimate[i], lhs)) <= 0) {
                cur += estimate[i];
                ret += powTwo[i];
            }
        }
        ret.nega = lhs.nega ^ rhs.nega;
        return ret;
    }
    friend BigInt operator/ (const BigInt& num,const ll& x) {
        bool negat = (x < 0);
        ll xx = (negat) ? -x : x;
        BigInt ret;
        ll k = 0;
        ret.val.resize(num.size());
        ret.nega = (num.nega ^ negat);
        for(int i = num.size() - 1 ;i >= 0; i--){
            ret[i] = (k * MOD + num[i]) / xx;
            k = (k * MOD + num[i]) % xx;
        }
        ret.trim();
        return ret;
    }
    bool operator== (const BigInt& rhs) const {
        return nega == rhs.nega && val == rhs.val;
    }
    bool operator!= (const BigInt& rhs) const {
        return nega != rhs.nega || val != rhs.val;
    }
    bool operator>= (const BigInt& rhs) const {
        return !(*this < rhs);
    }
    bool operator> (const BigInt& rhs) const {
        return !(*this <= rhs);
    }
    bool operator<= (const BigInt& rhs) const {
        if (nega && !rhs.nega) {
            return true;
        }
        if (!nega && rhs.nega) {
            return false;
        }
        int cmp = absComp(*this, rhs);
        return nega ? cmp >= 0 : cmp <= 0;
    }
    bool operator< (const BigInt& rhs) const {
        if (nega && !rhs.nega) {
            return true;
        }
```

```
        if (!nega && rhs.nega) {
            return false;
        }
        return (absComp(*this, rhs) < 0) ^ nega;
    }
    void swap(const BigInt& rhs) const {
        std::swap(val, rhs.val);
        std::swap(nega, rhs.nega);
    }
};
BigInt ba, bb;
int main() {
    std::cin >> ba >> bb;
    std::cout << ba + bb;
    return 0;
}
```