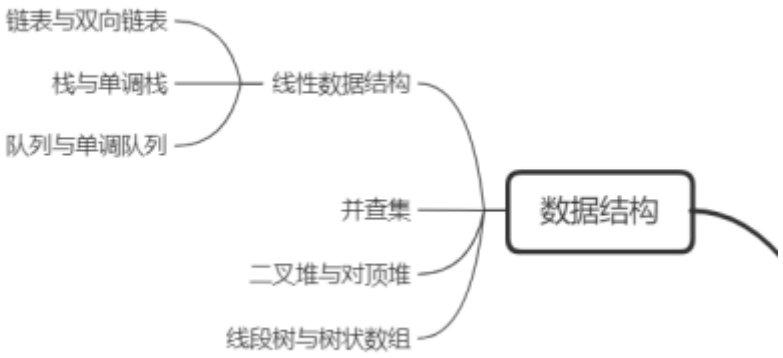
蓝桥杯十天冲刺省一



Day-5 数据结构

队列

队列的概念

队列是一种操作受到限制的线性表，用于模拟类似生活中的排队类型的数据结构，它支持在一端插入元素，在另外一端删除元素。

模拟现实生活中的排队过程。若干对象排成一条线性的队伍，等待某种操作（结账、处理银行业务、看病打针等等）。只有队首才能进行某种操作。队首对象处理完后，排在队首后面的人成为新的队首，然后才可以进行某种操作。

因此，队列有下列几种概念：

（1）队首：队列中的第一个元素

（2）队尾：队列中的最后一个元素

（3）队列长度/大小：队列中的元素个数

（4）队列判空：判断队列是否为空

（5）取队首：取得队首元素的值，注意，队列为空时，取队首操作是非法的

（6）入队：向队列尾部添加一个元素

（7）出队：队首元素移出队列，注意，队列为空时，出队操作是非法的

队列的操作

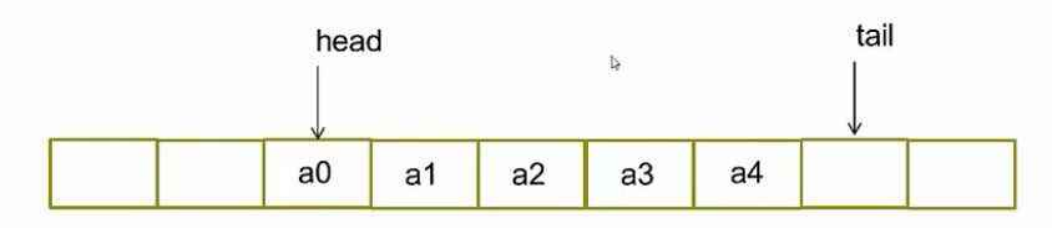
1. 加入队列：有一个人也想去排队，就要加入进去，并且排到队列的尾部，也就是当前队列的最后一个人的后面。（插队？不允许的，非法操作！）
2. 出队操作：队首元素离开队列

3. 求队列大小：队列中元素的个数

4. 判断队列为空：

5. 获取队首元素

6. 获取队尾元素



队列实现代码

- 数组模拟

```
int q[1000];              // 定义一个最大历史容量是 1000 的队列
int head=0, tail=0;       // 定义队列头 队列尾，初始值依个人习惯而设

int len = tail - head;    // 获取队列的长度
if(head == tail)          // 判断队列为空
int h = q[head];          // 取队首
q[tail++] = t;            // 将 t 入队
head++;                   // 队首出队，前提是队列不为空
```

- C++自带queue

```
queue<int> q;             // 定义一个空队列
int len = q.size();       // 获取队列的长度
if(q.empty())             // 判断队列为空
int h = q.front();        // 取队首
int t = q.back();         // 取队尾
q.push(t);                // 将 t 入队
q.pop();                  // 队首出队，前提是队列不为空
......                    // 更多 STL 队列的使用方法可自行百度
```

队列

- 手写队列

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

char q[1010]; // 队列数组。
int hh = 1, tt = 0; // 队列头和尾指针。

// 检查队列是否为空。
bool q_empty()
{
    return hh > tt;
}

// 入队操作。
void q_push(char x)
{
    tt++;
    q[tt] = x;
}

// 出队操作。
void q_pop()
{
    hh++;
}

// 获取队首元素。
char q_front()
{
    return q[hh];
}

// 主函数。
int main()
{
    int n;
```

```
    cin >> n; // 输入队列的大小。
    for(int i = 1; i <= n; i++)
    {
        char c;
        cin >> c; // 输入要入队的字符。
        q_push(c);
    }

    // 执行操作直到队列为空。
    while(!q_empty())
    {
        cout << q_front(); // 输出队首元素。
        q_push(q_front()); // 将队首元素再次入队。
        q_pop(); // 出队两次以丢弃队首元素。
        q_pop();
    }
}
```

- queue

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    queue<char> q;//声明队列q
    int n;
    cin >> n;
    for(int i = 1; i <= n; i++)
    {
        char a;
        cin >> a;
        q.push(a);
    }

    while(!q.empty())
    {
        cout << q.front();
        q.push(q.front());
        q.pop();
```

```
            q.pop();
        }
    }
```

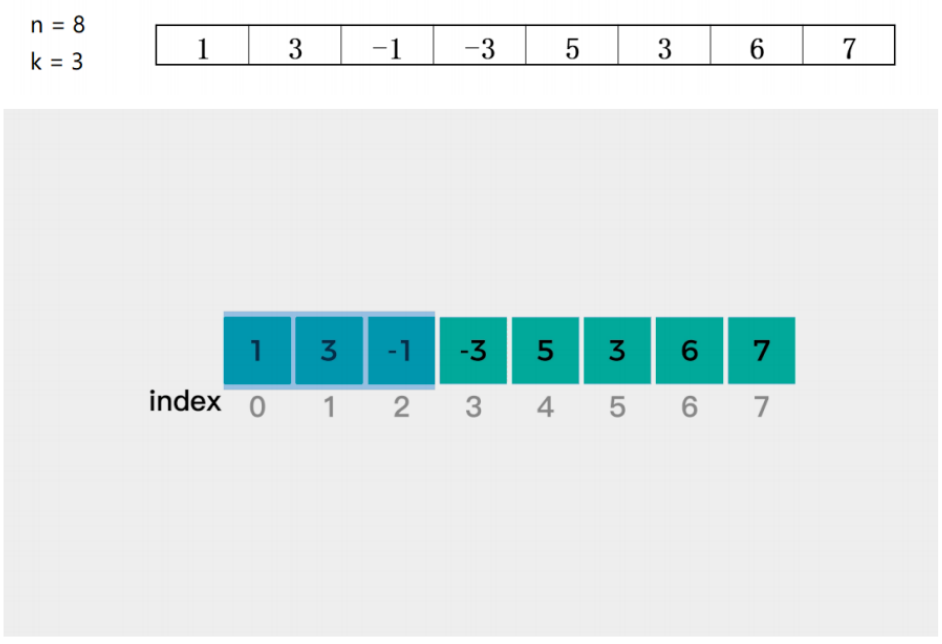
单调队列

单调队列，即单调递减或单调递增的队列。队列中元素之间的关系具有单调性，而且，队首和队尾都可以进行出队操作，只有队尾可以进行入队操作。

单调队列的使用频率不高，但在有些程序中会有非同寻常的作用。

问题引入

给定一个长度为 $(n(n \leq 10^7)$ 的数组 $A[]$ 和长度为 $k(k \leq 10^5)$ 的窗口，这个窗口从数组左端向右端滑动，求每向前滑动一次后，窗口内元素的最小值，并输出它们。例如：

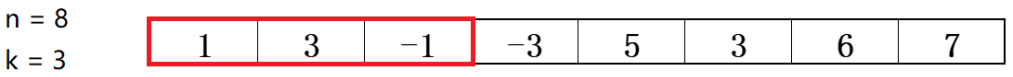


这个问题如果要暴力枚举的话，代码如下时间复杂度为$O(nk)$

```
for(int i = 1; i <= n-k+1; i++)
{
    int minn = INT_MAX;
    for(int j = i; j < i+k; j++)
        minn = min(minn, a[j]);
    cout << minn << " ";
}
```

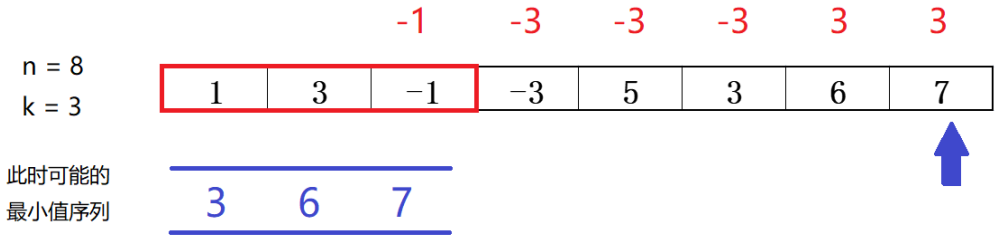
时间复杂度 $O(nk)$，太高，如何优化?

考虑尽可能排除掉不可能是答案的值，第一层循环不能降低了，只能考虑降低第二层循环的次数。比如下图中，当处理到 -1 的时候，很明显，前面的 1 和 3 不会成为当前及之后的窗口中的最小值（因为 -1 比它们更靠后），可以直接排除掉了。



更一般地，如果 $A[i]$ 和 $A[j]$ 处在同一个窗口内，并且 $i < j$ 且 $A[i] \geq A[j]$，那么，必有 $A[i]$ 不可能是当前窗口内的最小值，也不可能是后续窗口内的最小值。即 A[i] 是一个无用的数。

总结一下，我们发现，当处理 $A[j]$ 时，$A[j]$ 之前的所有比 $A[j]$ 大的元素都可以排除，不用考虑。由此，我们可以得出上面例子中的各个滑动窗口内的最小值如下：



最后，当滑动到 7 时，此时可能的解有三个：$3, 6, 7$。元素 -1 在处理到元素 3 时被排除掉，元素 -3 在处理元素 6 时被排除掉。

对于上面这种既有序，又单调的序列，我们称之为“单调队列”。

单调队列性质

单调队列中元素之间的关系具有单调性，分为 **单调递增队列** 和 **单调递减队列**。

1. 队列中的元素在原来的列表中的位置是由前往后的(随着循环顺序入队)。
2. 队列中的队首元素和队尾元素的距离应不超过问题中规定的区间长度（**区间性**）
3. 队列中元素的大小是单调递增或递减的（**单调性**）。

因此，

1. 为了保证队列的区间性，那么我们在将元素入队之前，应该将队列中会破坏区间性的（队首）元素删除掉。
2. 为了保证队列的单调性，那么我们在将元素入队之前，应该将队列中会破坏单调性的（队尾）元素删除掉。

代码实现

- 手写队列queue

```
int q[N], head = 0, tail = 0;
for (int i = 0; i < n; i ++ )
{
    while (head < tail && check_out(q[head], i)) head++;  // 队头滑出窗口区域就弹出队首
    while (head < tail && check(q[tail-1], i)) tail--;    // 队尾不满足单调性就弹出队尾
    q[tail++] = i;                                        // 当前元素入队
}
```

- 双端队列deque

```
#include <deque>          // 头文件
deque<int> dq;            // 定义
dq.push_back(t);          // 在队尾插入元素 t
dq.push_front(t);         // 在队首插入元素 t
dq.pop_back();            // 删除队尾元素
```

```
dq.pop_front();             // 删除队首元素
dq.front() = t;             // 获取队首元素的引用，并修改为 t
dq.back() = t;              // 获取队尾元素的引用，并修改为 t
dq[pos] = t;                // 获取下标元素的引用，并修改为 t
int len = dq.size();        // 获取队列大小
dq.clear();                 // 清空队列
bool flag = dq.empty();     // 判空

// 遍历输出队列
for(int i = 0; i < dq.size(); i++)
    cout << dq[i] << ' ';

// 双端队列实现单调递增队列
deque<int> q;  // 队列保存下标即可
for(int i = 1; i <= n; i++) {
    while(!q.empty() && q.front()< i-len+1) q.pop_front(); //
队首元素已经出了长为 len 的窗口，需要出队
    while(!q.empty() && a[q.back()] >= a[i]) q.pop_back();  //
队列不空，且队尾元素大于当前元素，不满足单调递增
    q.push_back(i);  // 当前元素（下标）入队
}
```

单调队列

```
#include<bits/stdc++.h> // 万能头文件
using namespace std;

// 结构体定义
struct node{
    int v;    // 值
    int pos;  // 位置
};

deque<node> que;  // 双端队列
int n, a[10005];  // 变量声明

int main()
{
    cin >> n; // 输入队列长度
```

```
    for(int i = 1; i <= n; i++)
        cin >> a[i]; // 输入队列元素
    for(int i = 1; i <= n; i++) {
        while(!que.empty() && que.back().v <= a[i])
que.pop_back(); // 维护队列单调性
        que.push_back((node){a[i], i}); // 入队
    }

    // 输出队列元素
    for(int i = 0; i < que.size(); i++)
        cout << que[i].v << ' ';
    return 0;
}
```

```
#include<bits/stdc++.h> // 万能头文件
using namespace std;

int a[10005], dq[100005]; // 声明数组

int main()
{
    int n;
    cin >> n; // 输入数组长度
    for(int i = 1; i <= n; i++)
    {
        cin >> a[i]; // 输入数组元素
    }
    int l = 1, r = 0; // 初始化左右指针
    for(int i = 1; i <= n; i++)
    {
        while(l <= r && a[i] >= dq[r])
        {
            r--; // 维护双端队列单调性
        }
        dq[++r] = a[i]; // 入队
    }
    cout << dq[l] << " " << dq[r]; // 输出双端队列的首尾元素
}
```

栈

栈的概念

栈是一种操作受限的线性表，可以看作是一个特殊的数组，数组只能在末尾增加元素或删除元素。这样的一种数据结构就叫做栈。栈有如下几个重要的概念：

栈的大小：数组中的元素的个数
栈顶元素：数组中的最后一个元素
栈为空：数组中没有任何元素
进栈：又称入栈，向数组末尾添加一个元素
出栈：如果数组不为空，就可以把数组中的最后一个元素（栈顶元素）删除

栈的实现

- 手写栈

```
int stk[100], top = 0;  // 定义一个栈，大小为100
int size = top - 0;  // 栈内元素个数
if(top == 0)         // 判空，表示栈为空
int t = stk[top];    // 取栈顶，前置条件：栈不为空top > 0
stk[++top] = t;      // 入栈，将 t 入栈
top--;               // 出栈，前置条件：栈不为空  top > 0
```

- stack

```
stack<int> stk;      // 定义一个整型栈，大小为100
stk.size();          // 栈内元素个数
if(stk.empty())      // 判空，表示栈为空
int t = stk.top();   // 取栈顶，前置条件：栈不为空top > 0
stk.push(t);         // 入栈，将 t 入栈
stk.pop();             // 出栈，前置条件：栈不为空  top > 0
```

算式求值

- 手写栈

```
/******author:ssxyz******/
#include <bits/stdc++.h>
using namespace std;

int s[100010]; // 数字栈
int p = 0; // 栈顶下标

int main() {
    int n, ans = 0;
    char c;
    cin >> n;
    s[++p] = n; //第一个输入数字入栈
    while (cin >> c >> n) {
        if (c == '*')  //是乘法，就取出栈顶乘n之后放入栈顶
            s[p] = n * s[p] % 10000;
        else //是加法，直接入栈即可
            s[++p] = n;
    }
    while (p) { // 栈不为空，求一次和，出一次栈
        ans += s[p--];
        ans %= 10000;
    }
    cout << ans;
}
```

- stack

```
#include <bits/stdc++.h>
using namespace std;
stack<int> s;

int main() {
    int n, ans = 0;
    char c;
    cin >> n;
```

```
    s.push(n);  //第一个输入数字入栈
    while (cin >> c >> n) {
        if (c == '*') {
            int t = s.top();           //获取栈顶元素
            s.pop();                   //出栈
            s.push(t * n % 10000);     //将乘法的结果入栈
        } else  //是加法，直接入栈即可
            s.push(n);
    }
    while (!s.empty()) { // 栈不为空，求一次和，出一次栈
        ans += s.top();
        s.pop();
        ans %= 10000;
    }
    cout << ans;
}
```

单调栈

在一个线性数据结构中，为任意一个元素找左边和右边第一个比自己大/小的位置，要求 $O(n)$ 的复杂度

对于这类问题，基本解法很容易想到 $O(n^2)$ 的解法，不过这种解法明显太基础了，有优化的空间。单调栈就可以优化基本解法的时间复杂度。

时间复杂度

单调栈是$O(n)$的。

> *时间复杂度的证明：所有的元素只会进栈一次，而且一旦出栈后就不会再进来了。因此总的进出次数最多是$2n$次，因此时间复杂度为$O(n)$。*

单调栈的性质

单调栈里的元素具有单调性:

1. 单调递减栈=>向栈生长的地方单调递减;
2. 单调递增栈=>向栈生长的地方单调递增

元素加入栈前，会在栈顶端把破坏栈单调性的元素都删除!

使用单调栈可以找到元素向左遍历第一个比他小的元素，也可以找到元素向左遍历第一个比他大的元素。

一般解题思路

1. 栈中不存放值，而是存放下标；
2. 在尝试去将一个元素加入栈前，先对它和栈顶元素进行比较，并按照单调性要求，把破坏栈单调性的栈顶元素都删除，直至满足单调性。

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

int n, x;
int stk[100005], top; // 定义栈和栈顶指针

int main()
{
    cin >> n; // 输入查询次数
    for(int i = 1; i <= n; i++) // 循环进行查询
    {
        cin >> x; // 输入当前查询的值
        while(top > 0 && stk[top] >= x) top--; // 将栈顶大于当前查询值的元素弹出
        if(top > 0) cout << stk[top] << ' '; // 输出当前查询值的区间最小值
        else cout << "-1 "; // 如果栈为空，表示当前查询值为最小值
        stk[++top] = x; // 将当前查询值入栈
    }
    return 0;
}
```

接雨水

考虑在用单调递减栈去遍历元素时，对于第 i 个元素 $h[i]$，有以下两个性质：

- 如果有栈顶元素（记为 t）且 $h[t] < h[i]$，即需要出栈以维护单调递减性，那么 i 一定是 t 右侧第一个比它大的元素。（证明：如果不是，那么 t 必然已经被 $t \sim i$ 之间的某个元素入栈时弹出过了）

- 在栈内，如果 t 前面还有一个元素，记为 p，那么它一定是 t 前面比 t 大的第一个元素，这个根据单调递减栈的性质可知。

根据上面这两个性质，我们可以知道，在 $h[p] > h[t] < h[i]$，这里能储存一些水，其中高度在 $h[t]$ 以上部分的水的面积是 $(i-p-1) \times (\min(h[i], h[p]) - h[t])$。

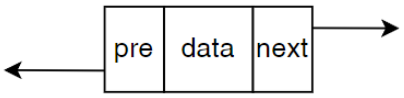
如此累加即可。

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

const int N = 1e5+5;
long long n, a[N], ans; // 定义变量和数组

int main()
{
    cin >> n; // 输入数组长度
    stack<int> st; // 定义栈，用于存储数组下标

    for(int i = 1; i <= n; i++) {
        cin >> a[i]; // 输入数组元素
        while(!st.empty() && a[st.top()] <= a[i]) { // 当栈不为空且当前元素大于栈顶元素时
            int t = st.top(); // 获取栈顶元素
            st.pop(); // 弹出栈顶元素
            if(!st.empty()) { // 如果栈不为空
                int w = i - st.top() - 1; // 计算宽度
                int h = min(a[i], a[st.top()]) - a[t]; // 计算高度
                ans += 1ll*w*h; // 更新答案
            }
        }
        st.push(i); // 当前元素入栈
    }

    cout << ans; // 输出结果
    return 0;
}
```

链表

定义

前面讲到，链表在内存当中存储时，每个元素的位置都是分散的。那么，怎么将体现或者让其组成一个线性的数据结构呢？（双向）链表通过在每个结点（元素）上额外定义两个指针变量，去分别指向当前结点的上一个结点和下一个结点来实现，如下图所示。



结点之间通过指针，实现了相互的连接，进而可以互相访问：

- 通过往前（左）指的指针 `pre`，可以不断地在链表上向左遍历；
- 通过往后（右）指的指针 `next`，可以不断地在链表上向右遍历；

在代码实现上，我们这里只讲解链表的数组模拟实现方式。

结点定义如下：

```
// 结构体写法
struct Node {
    int data;   // 结点信息，类型不一定是 int，有些问题中下标就是结点信息，进而不做定义
    int pre;    // 左指针，指向前一个元素
    int next;   // 右指针，指向后一个元素
};
Node lst[N];    // 定义多个结点，用于构造成链表
// 多数组写法
int data[N], pre[N], nxt[N];    // 不建议用 next 作为变量名，可能会和某些头文件有冲突
```

链表的构建

对于数组模拟出的链表，我们可以定义两个结点去表示链表头左边的空结点和链表尾右边的空结点（尽管这个结点实际上并不存在）。一般地，我们定义 0 是左侧空结点，极大的数字，比如 $N-1$ 是右侧的空结点：

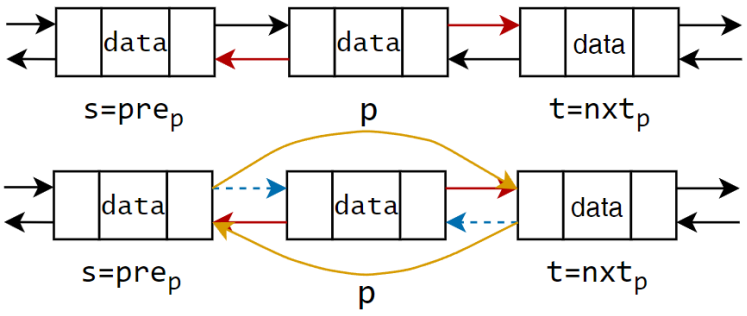
```
const int N = 1e5+5;
// 多数组写法
int data[N], pre[N], nxt[N];
// 初始化一个空的链表
nxt[0] = N-1;    // 0: 链表头
pre[N-1] = 0;    // N-1: 链表尾
```

此时，我们可以把 0 号下标表示的结点叫做链表头，把 $N-1$ 号下标表示的结点叫做链表尾。做了这个定义之后，我们可以很清晰地表示一个空链表——**当链表头直接和链表尾连接时，表示链表是空链表。**

而且，这么定义还有另外一个好处，就是我们可以**安全地**操作任何链表结点的插入和删除操作（接下来将讲到）。

链表的删除

假设我们要删除结点 p，如下图所示，那么要如何操作呢？



容易想到，我们直接让 p 的前一个结点和 p 的后一个结点直接连接就可以了。代码实现如下：

```
// 删除结点 p
void del(int p) {
    int s = pre[p], t = nxt[p];  // 先保存好 p 的前一个结点 和 后一个结点
    nxt[s] = t;   // s 的后面是 t
    pre[t] = s;   // t 的前面是 s
}
```

时间复杂度：$O(1)$

值得注意的是，在删除结点 p 后，结点 p 的两个指针仍旧分别指向 s 和 t，结点仍旧占用一个元素的空间，并没有真的从内存中删除掉，这是数组模拟链表的一个弊端。当操作中存在大量插入和删除操作时，空间是存在一定的浪费的（和数组模拟的队列类似，结点出队后，该结点所在位置就再也不会访问，但是仍占据存储空间）。

一个解决办法是采用”垃圾回收机制“，额外定义一个容器去保存已经被删除的结点（可以再次利用的结点）编号，当需要新建结点时，优先从该容器中取出可以使用的结点编号。

链表的插入

假设我们要在结点 p 后面插入结点 q，那么要如何操作呢？

容易想到，我们先让 p 和 q 连接，然后让 q 和 p 原来的后继结点连接即可。

如果结点 q 是新建的结点，就涉及到链表结点的申请了。类似队列，我们可以额外定义一个变量 cnt 表示当前已经使用到的最大的链表结点编号。然后将其插入到要插入的结点后面。

代码实现如下：

```
// 新建结点，其值为 x
data[++cnt] = x;
pre[cnt] = nxt[cnt] = 0;    // 清空指针，一般不需要
// 将结点 q 插入到 p 后面
void ins(int p, int q) {
    int t = nxt[p];
    nxt[p] = q;
    pre[q] = p;
    nxt[q] = t;
    pre[t] = q;
}
```

时间复杂度：$O(1)$

链表的遍历

链表的遍历和数组类似，从第一个开始，不断往后即可。但是由于其下标并不代表其在链表中的顺序，因此不能直接用 `i++/i--` 来实现切换到下一个/上一个结点。而是通过 pre/nxt 数组来实现。

示例代码如下：

```
// 从前往后遍历链表
int p = nxt[0];    // 0 的后面的结点才是链表中的第一个结点，即真正的链表头
while(p != N-1) {  // 只要结点没到 N-1（链表尾的标志）
    // 处理当前结点，比如输出
    cout << data[p] << " ";
    ......
    p = nxt[p];    // 切换到下一个结点
}
// 从后往前遍历链表
int p = pre[N-1];    // N-1 的前面的结点才是链表中的最后一个结点，即真正的链表尾
while(p != 0) {  // 只要结点没到 0（链表头的标志）
    // 处理当前结点，比如输出
    cout << data[p] << " ";
    .......
    p = pre[p];    // 切换到前一个结点
```

```
}
```

时间复杂度：$O(n)$

找第 k 个元素

这是链表所有操作中，和数组相比最”不好“的一种操作。在数组里我们可以很快知道，访问 $a[k]$ 即可，时间复杂度是 $O(1)$ 的。而在链表中，你并不能很快地知道第 k 个元素到底在内存的哪个位置，因此，我们不得不从链表头开始，逐个遍历到第 k 个，因此时间复杂度是 $O(k)$ 的。

示例代码如下：

```
// 找链表中第 k 个元素，如果找不到，返回 N-1
int kth() {
    int p = 0, num = 0;
    while(num < k && p != N-1) {
        p = nxt[p];
        num++;
    }
    return p;
}
```

时间复杂度：$O(k)$

STL List

基本语法

在C++中，可以使用 `std::list` 来创建一个双向链表数据结构。以下是 `std::list` 的一些基本用法：

1. **包含头文件**： 在使用 `std::list` 之前，首先要包含相应的头文件：

   ```
   cppCopy code
   #include <list>
   ```

2. **声明一个list**：

```
cppCopy code
std::list<int> myList; // 声明一个存储整数的list
```

3. **插入元素**：

```
cppCopy codemyList.push_back(5); // 在列表末尾添加元素5
myList.push_front(10); // 在列表开头添加元素10
```

4. **访问元素**：

```
cppCopy code// 使用迭代器访问元素
for (std::list<int>::iterator it = myList.begin(); it !=
myList.end(); ++it) {
    std::cout << *it << " ";
}
```

5. **删除元素**：

```
cppCopy codemyList.pop_back(); // 删除列表末尾的元素
myList.pop_front(); // 删除列表开头的元素
```

6. **获取列表大小**：

```
cppCopy code
int size = myList.size(); // 获取列表的大小
```

7. **清空列表**：

```
cppCopy code
myList.clear(); // 清空列表
```

8. **其他操作**：
    - `insert()`：在指定位置插入元素
    - `erase()`：删除指定位置或者元素值的元素
    - `sort()`：对列表进行排序
    - `merge()`：合并两个有序列表
    - `reverse()`：反转列表等。
9. **迭代器**：`std::list` 支持双向迭代器，可以使用迭代器来访问列表中的元素。迭代器用于在容器中遍历元素。

迭代器O(1)删除一个元素

当你使用迭代器来遍历 `std::list` 并删除元素时，你需要注意迭代器失效的问题。因为删除元素会改变列表的结构，可能会使得之前获取的迭代器无效。为了避免这种情况，你可以使用迭代器的 `erase()` 函数。

下面是一个详细的讲解：

假设你有一个 `std::list<int>` 的列表，并且想要删除其中一些元素。你可以使用迭代器来遍历列表，并在遍历的过程中删除元素。但是直接调用 `erase()` 函数可能会导致迭代器失效，因此你需要使用迭代器的返回值来更新迭代器。

```
cppCopy code#include <iostream>
#include <list>

int main() {
    std::list<int> myList = {1, 2, 3, 4, 5};

    // 使用迭代器遍历列表
    for (std::list<int>::iterator it = myList.begin(); it != 
myList.end(); /* 不需要it++在这里 */) {
        if (*it % 2 == 0) {
            // 删除偶数元素
            it = myList.erase(it); // erase()函数返回指向下一个元
素的迭代器
        } else {
            // 只有在不删除元素时才递增迭代器
            ++it;
        }
    }

    // 打印列表
    for (int num : myList) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

在这个示例中，我们遍历 `myList` 的所有元素，并删除偶数元素。注意在删除元素时，我们使用 `erase()` 函数，它返回指向下一个元素的迭代器，这样可以避免迭代器失效。

需要注意的是，在我们的循环中，我们没有使用 `it++` 来增加迭代器。这是因为 `erase()` 函数已经在迭代器内部进行了递增操作。如果我们手动递增迭代器，可能会导致跳过某些元素，或者在删除元素后出现未定义行为。

通过这种方式，我们可以安全地在使用迭代器遍历 `std::list` 并删除元素时避免迭代器失效的问题。

士兵

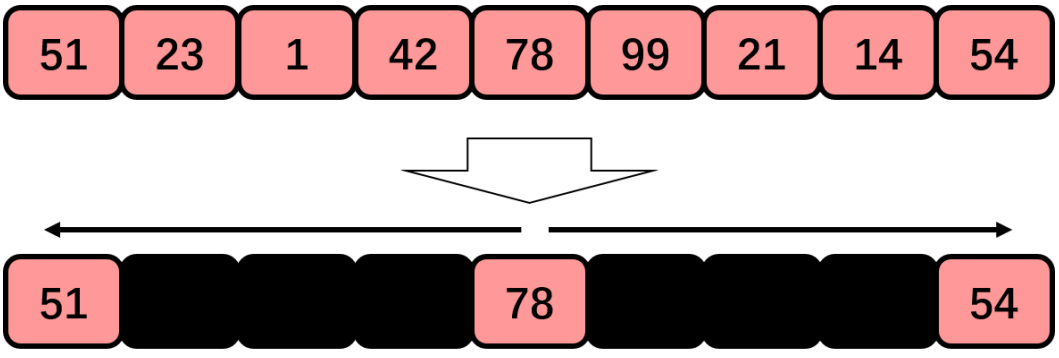
问题1：如何优化查询删除效率？

回答1：用模拟链表方法。

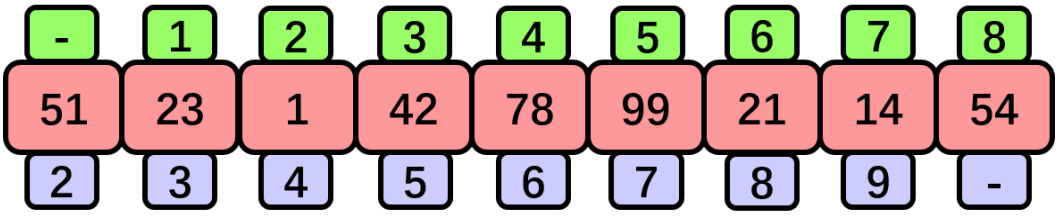
什么是链表？在初赛课程中我们也已经介绍过，它是一种数据结构，在这我只使用了它的一小部分特性，**即使不懂链表的同学也能搞懂**。

回答1修改版：用辅助数组维护左右相邻值的位置。

先回顾为什么我们的暴力这么慢，显然是因为我们删除的时候直接标记了一下就不管了，结果找寻找左右相邻值的时候经过大量被"删除"的位置，查找效率是 $O(n)$，例如：

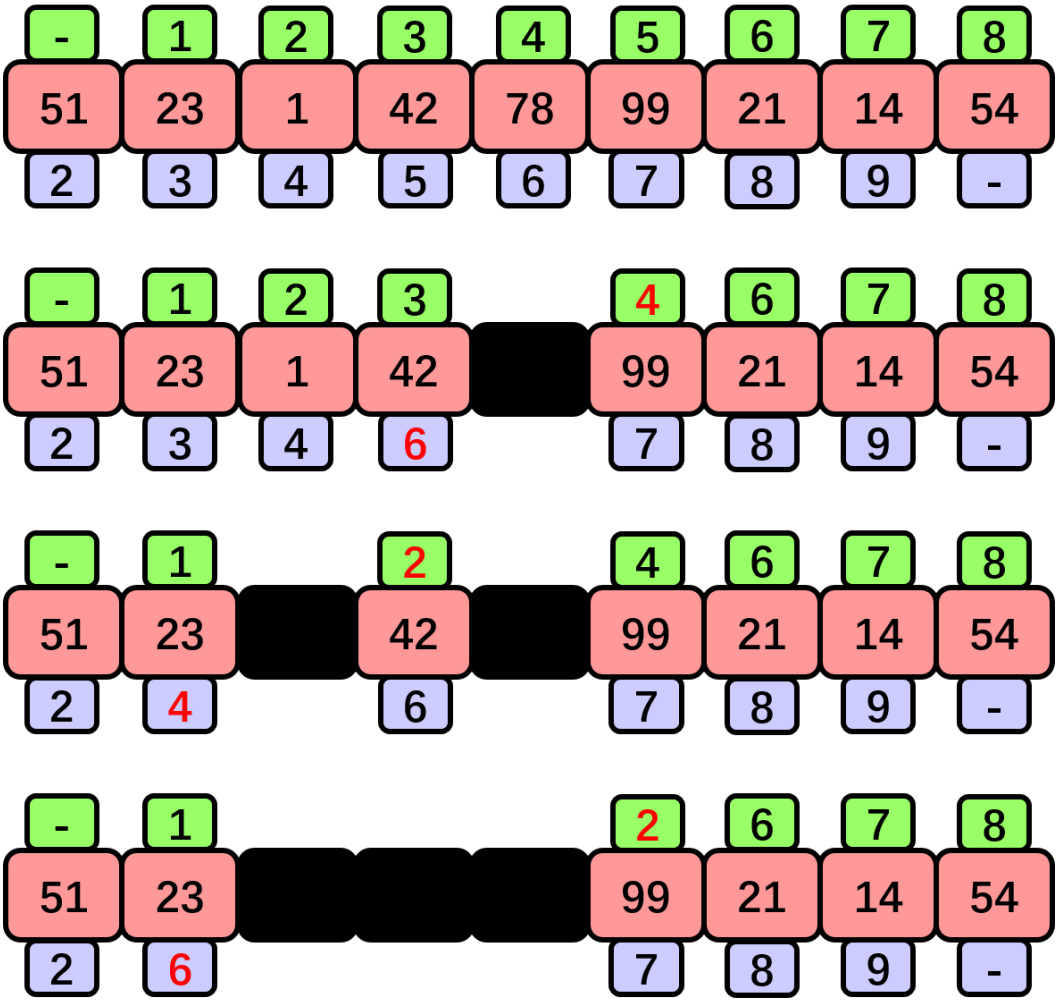


现在我们给数组的每一个数字按上两个辅助变量，专门用来记录左边和右边相邻值在什么位置



其中绿色变量 $L[x]$ 表示 x 位置左边相邻位置的下标，蓝色变量表示 $R[x]$ 表示 x 位置数字右边相邻位置的下标，这个我们一开始是可以处理好的。

然后对于每一次删除的值 x，我们可以 **快速找到** 它的位置（就是 x）与它的 **左右相邻位置**（$L[x]$ 与 $R[x]$）。至于删除 x，我们发现受影响只有 $L[x]$ 与 $R[x]$ 这两个位置（它们的 $L[]$ 与 $R[]$ 需要更新），那么我们更新它们就可以了，举个例子：



用了这个方法的效率又如何？不难发现，定位，查找，删除的时间效率都是 $O(1)$，那么一次询问的总体时间效率就是 $O(1)$，优化成功。

至于这个东西具体怎么写，其实很简单，直接代码附上。

```
#include <bits/stdc++.h>
using namespace std;
struct node {
    int l, r;  //l左伙伴下标，r右伙伴下标
} s[1000010];
```

```
int main() {
    ios::sync_with_stdio(0), cin.tie(0);  //加速cin和cout
    int n, m, t;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) {
        s[i].l = i - 1;
        s[i].r = i + 1;
    }
    while (m--) {
        cin >> t;
        if (s[t].l == 0) cout << "* "; //左边是0伙伴
        else cout << s[t].l << " ";
        if (s[t].r == n + 1) cout << "*\n"; //右边是n+1伙伴
        else cout << s[t].r << "\n";
        s[s[t].l].r = s[t].r; //t的左伙伴的右边是t的右伙伴
        s[s[t].r].l = s[t].l; //t的右伙伴的左边是t的左伙伴
    }
}
```

整数删除（蓝桥杯C/C++2023B组省赛）

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

using ll = long long;
#define pli pair<ll, int>
#define ft first
#define sd second

const int N = 5e5 + 100;

ll n, k, l[N], r[N] = { 1 }, a[N]; // 定义变量和数组

priority_queue<pli, vector<pli>, greater<pli>> heap; // 小顶堆，用于存储元素

int main() {
    scanf("%lld%lld", &n, &k); // 输入 n 和 k
```

```
    for (int i = 1; i <= n; i++) {
        scanf("%lld", &a[i]); // 输入数组 a 的值
        heap.push({ a[i], i }); // 将值和索引入小顶堆
        l[i] = i - 1; // 初始化左相邻节点
        r[i] = i + 1; // 初始化右相邻节点
    }
    while (k) {
        ll val = heap.top().ft; // 获取堆顶的值
        ll pos = heap.top().sd; // 获取堆顶值对应的索引
        heap.pop(); // 弹出堆顶元素

        if (val != a[pos]) continue; // 如果当前值与数组中的值不
相等，说明已经被更新过，跳过本次循环
        else k--; // 否则，减少 k 的值

        if (l[pos] >= 1) { // 如果左相邻节点存在
            a[l[pos]] += val; // 更新左相邻节点的值
            heap.push({ a[l[pos]], l[pos] }); // 将更新后的值和
索引入小顶堆
        }
        if (r[pos] <= n) { // 如果右相邻节点存在
            a[r[pos]] += val; // 更新右相邻节点的值
            heap.push({ a[r[pos]], r[pos] }); // 将更新后的值和
索引入小顶堆
        }

        l[r[pos]] = l[pos]; // 更新右相邻节点的左相邻节点
        r[l[pos]] = r[pos]; // 更新左相邻节点的右相邻节点
    }
    for (int i = r[0]; i <= n; i = r[i]) // 输出结果
        printf("%lld ", a[i]);
}
```

并查集

集合与并查集的概念

集合是由一个或多个确定的元素所构成的整体。集合中的元素有如下 三个特征:

1. 确定性:一个元素要么属于集合，要么不属于集合。
2. 互异性:集合中的元素互不相同。
3. 无序性:集合中的元素没有先后顺序。

并查集是一个可以维护集合的数据结构，它能高效支持集合的基本操作:

1. 合并两个集合。
2. 查询两个指定元素是否属于同一个集合。

需要注意的是，由于计算机存储结构的限制，并查集维护的集合是离散意义下的集合，而不是广义的集合。集合中的元素是有限的。

并查集的操作

因为一个元素只可能属于一个集合，所以我们可以为每一个集合选取一个代表元。于是查询两个元素是否属于同一个集合实际上就是询问两个元素所在集合的代表元是否相同。这个询问的时间复杂度可以利用数组标记优化为 $O(1)$。

但是合并两个集合时需要改变其中一个集合中所有元素的代表元，时间复杂度仍然非常高，如何优化呢?

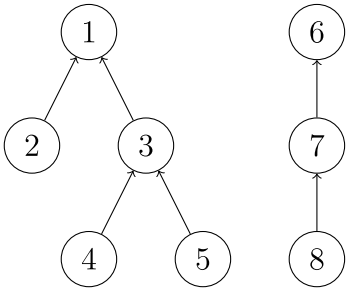
注意到，合并操作的时间复杂度远高于查询操作的复杂度，这启发我们通过一定的方式，提高查询操作的复杂度，降低合并操作的复杂度。

我们并不需要 $O(1)$ 知道每个元素所属集合的代表元，这启发我们用森林来维护代表元。用森林中的一棵树代表一个集合，树根为对应集合的代表元。

这样，对于每棵树上的元素，查询其代表元时，时间复杂度与树的高度成正比。

对两个集合进行合并操作时，只需将其中一个集合的代表元（树根）指向另一个集合（树）的代表元即可。时间复杂度也与树的高度成正比。

这就是并查集。



并查集是一种树形的数据结构，顾名思义，它用于处理一些不交集的 **合并** 及 **查询** 问题。它支持两种操作：

- 查找（Find）：确定某个元素处于哪个子集；
- 合并（Union）：将两个子集合并成一个集合。

定义

```
int fa[N];       // fa[i]: i 所属的集合（i 的祖先节点）
```

初始化（一定不要忘）

```
void makeSet(int size) {
  for (int i = 0; i < size; i++) fa[i] = i;  // i就在它本身的集合里
  return;
}
```

查找

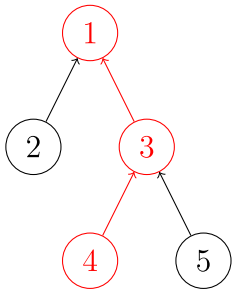
通俗地讲一个故事：几个家族进行宴会，但是家族普遍长寿，所以人数众多。由于长时间的分离以及年龄的增长，这些人逐渐忘掉了自己的亲人，只记得自己的爸爸是谁了，而最长者（称为「祖先」）的父亲已经去世，他只知道自己是祖先。为了确定自己是哪个家族，他们想出了一个办法，只要问自己的爸爸是不是祖先，一层一层的向上问，直到问到祖先。如果要判断两人是否在同一家族，只要看两人的祖先是不是同一人就可以了。

在这样的思想下，并查集的查找算法诞生了。

此处给出一种 C++ 的参考实现：

```
int fa[MAXN];  // 记录某个人的爸爸是谁，特别规定，祖先的爸爸是他自
己
int find(int x) {
  // 寻找 x 的祖先
  if (fa[x] == x)  // 如果 x 是祖先则返回
    return x;
  else
    return find(fa[x]);  // 如果不是则 x 的爸爸问 x 的爷爷
}
```

显然这样最终会返回 x 的祖先。并且上述查询操作的代码的时间复杂度取决于每个集合对应的树的高度。



合并

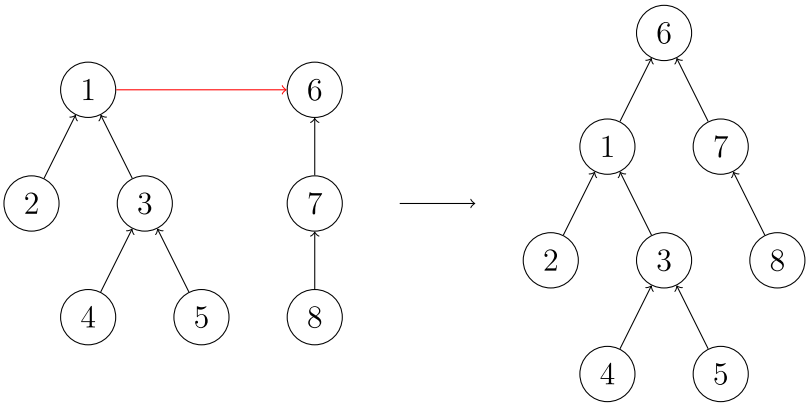
宴会上，一个家族的祖先突然对另一个家族说：我们两个家族交情这么好，不如合成一家好了。另一个家族也欣然接受了。

我们之前说过，并不在意祖先究竟是谁，所以只要其中一个祖先变成另一个祖先的儿子就可以了。

此处给出一种 C++ 的参考实现：

```
void unionSet(int x, int y) {
  // x 与 y 所在家族合并
  x = find(x);
  y = find(y);
  fa[x] = y;  // 把 x 的祖先变成 y 的祖先的儿子
}
```

可以看出，合并操作的时间复杂度取决于查找操作的时间复杂度，也就是每个集合对应的树的高度。



时间复杂度优化

不幸的是，上面代码看似优秀，但实际上，在最坏情况下时间复杂度仍然很高——因为森林中树的深度可能比较大。如果我们不停地从一个深度比较大的点向上寻找代表元，时间复杂度就令人难以接受。

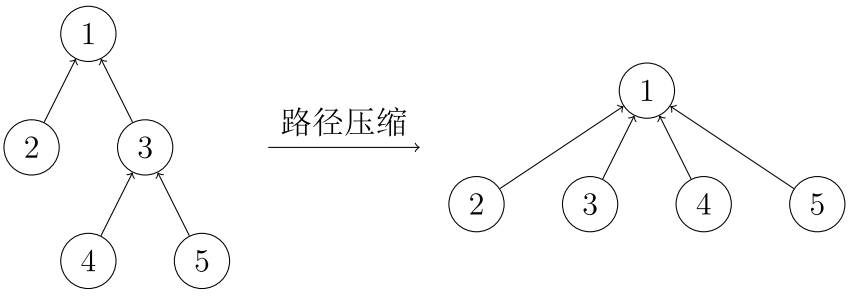
下面我们来讲讲上述操作的优化操作。

查找的优化——路径压缩

这样的确可以达成目的，但是显然效率实在太低。为什么呢？因为我们使用了太多没用的信息，我的祖先是谁与我父亲是谁没什么关系，这样一层一层找太浪费时间，不如我直接当祖先的儿子，问一次就可以出结果了。甚至祖先是谁都无所谓，只要这个人可以代表我们家族就能得到想要的效果。 **把在路径上的每个节点都直接连接到根上** ，这就是路径压缩。

此处给出一种 C++ 的参考实现：

```
int find(int x) {
  if (x == fa[x])  return fa[x];// 如果 x 是祖先(根)则返回
  return fa[x] = find(fa[x]);  // 查找 x 的祖先直到找到代表,于是顺手路径压缩
}
```



家族血统

```
#include <cstdio>
#include <iostream>
#include <map>

using namespace std;

map<string, string> fa; // 用于存储关系的映射，其中 fa[儿子] = 父亲

// 查找一个人的最终父亲
string find(string x) {
    // 如果 x 的父亲不是它自己，则递归查找直到找到最终父亲
    if (fa[x] != x)
        fa[x] = find(fa[x]);
    return fa[x];
}

string father, son;

int main() {
    char q;
    // 持续读取输入，直到遇到 '$'
    while ((cin >> q) && q != '$') {
        if (q == '#') {
            // 如果输入是 '#'，则读取父亲的名字
            cin >> father;
            // 如果父亲还不在映射中，则将其添加，并将其指向自己
            if (fa[father] == "")
                fa[father] = father;
        } else if (q == '+') {
            // 如果输入是 '+'，则读取儿子的名字，并将其父亲设为之前读取的父亲
            cin >> son;
            fa[son] = father;
```

```
        } else {
            // 如果输入既不是 '#' 也不是 '+'，则是一个查询，读取儿子的名字
            cin >> son;
            // 输出儿子的名字和他的最终父亲
            cout << son << ' ' << find(son) << '\n';
        }
    }
    return 0;
}
```

带权并查集

> *带权并查集的核心能力就是维护多个元素之间的连通以及偏移关系，甚至可以维护多个偏移关系。*
>
> *而偏移量可以理解为当前结点到根结点的距离之和。其核心运算是向量运算。*

普通的并查集仅仅记录的是集合的关系，这个关系无非是同属一个集合或者是不在一个集合。

带权并查集不仅记录集合的关系，还记录着**集合内元素的关系**或者说是集合内**元素连接线的权值**。这个权值会在查询（路径压缩）和合并时进行更新。

普通并查集本质是不带权值的图，而带权并查集则是带权的图

考虑到权值就会有以下问题：

1. 每个节点都记录的是与根节点之间的权值 val[]，那么在 Find 的路径压缩过程中，权值也应该做相应的更新，因为在路径压缩之前，每个节点都是与其父节点链接着，那个 Value 自然也是与其父节点之间的权值。
2. 在两个并查集做合并的时候，权值也要做相应的更新，因为两个并查集的根节点不同。

向量偏移法

对于集合里的**任意两个元素** x, y 而言，它们之间必定存在着某种联系，因为**并查集中的元素均是有联系的**（这点是并查集的实质，要深刻理解），否则也不会被合并到当前集合中，那么我们就把这 2 个元素之间的关系量转化为一个**偏移量**。

路径压缩

只需要在原有代码基础上做一点修改即可。

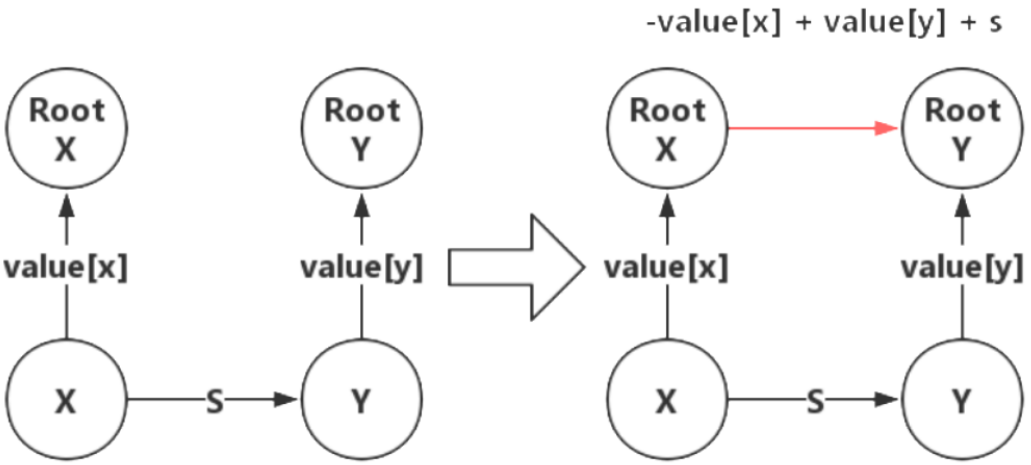
```
int fa[N];   // fa[i]: i 的父结点
int val[N];  // val[i]: 结点 i 到其所在集合的根节点的路径权值和
int find(int x) {
    if(x == fa[x]) return x;
    int k = find(fa[x]);  // 记录原父节点编号
    val[x] += val[fa[x]];
    return fa[x] = k;
}
```

因为在路径压缩后**父节点直接变为根节点**，此时父节点的权值已经是父节点到根节点的权值了，将当前节点的权值加上原本父节点的权值，就得到当前节点到根节点的权值。

合并

已知 x，y 根节点分别为 $xRoot$，$yRoot$，如果有了 x、y 之间的关系，合并如果不考虑权值直接修改 $father$ 就行了，但是现在是带权并查集，必须得求出 $xRoot$ 与 $yRoot$ 这条边的权值是多少，很显然 x 到 $yRoot$ 两条路径的权值之和应该相同，就不难得出下面代码所表达的更新式（但是需要注意并不是每个问题都是这样更新的，有时候可能会做取模之类的操作，这一点在之后的例题中可以体现）。

> *注意：这里的路径权值“和”在有些问题中可能会变成“乘、异或”等。*




```
void merge(int x, int y, int s) {
    int xRoot = find(x);
    int yRoot = find(y);
    if (xRoot != yRoot) {
        parent[xRoot] = yRoot;
        value[xRoot] = -value[x] + value[y] + s;
    }
}
```

网络分析（蓝桥杯C/C++2020B组省赛第一场）

```
#include <iostream>
#include <vector>
#include <cstring>

using namespace std;

const int N = 1e4 + 10, M = 1e5 + 10;

int n, m;

int p[N], val[N];

// 查找并压缩路径
int find(int x) {
```

```
    if (p[x] == x || p[p[x]] == p[x]) // 如果 x 的父亲是自己，或者 x 的祖父等于父亲，直接返回 x
        return p[x];

    int r = find(p[x]); // 递归查找根节点
    val[x] += val[p[x]]; // 更新值
    p[x] = r; // 路径压缩

    return r;
}

int main() {
    cin >> n >> m;

    for (int i = 1; i <= n; i++) {
        p[i] = i, val[i] = 0; // 初始化每个节点的父亲为自己，值为0
    }

    while (m--) {
        int op, a, b;

        cin >> op >> a >> b;

        int fa = find(a), fb = find(b); // 查找 a 和 b 的根节点

        if (op == 1) { // 合并操作
            if (fa != fb) { // 如果 a 和 b 不在同一个集合中
                p[fa] = fb; // 合并集合
                val[fa] -= val[fb]; // 更新值
            }
        } else { // 修改值操作
            val[fa] += b; // 修改 a 所在集合的值
        }
    }

    // 输出每个节点的最终值
    for (int i = 1; i <= n; i++) {
        int root = find(i); // 找到当前节点所在集合的根节点
```

```
        int finalValue = val[i] + (root == i ? 0 : val[root]);
// 计算最终值
        cout << finalValue;
        if (i != n) // 如果不是最后一个节点，则输出空格
            cout << ' ';
        else // 如果是最后一个节点，则输出换行
            cout << '\n';
    }

    return 0;
}
```

二叉堆与对顶堆

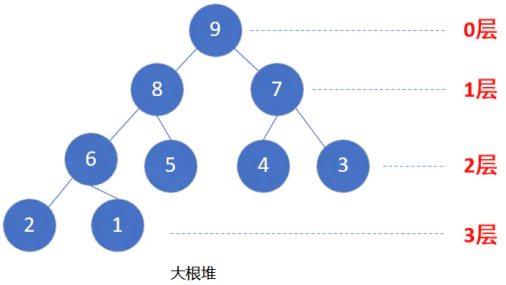
二叉堆的概念

二叉堆是一种特殊的堆，并且是一棵完全二叉树，每个结点中存有一个元素。二叉堆有两种：

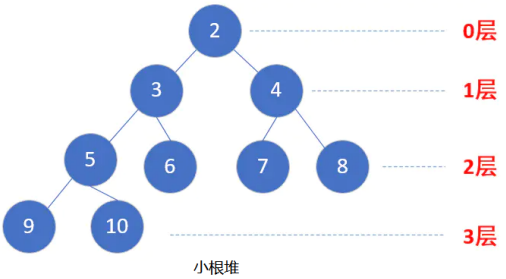
（1）大根堆（最大堆）：任何一个父节点的值，都**大于等于**它左右孩子节点的值；

（2）小根堆（最小堆）：任何一个父节点的值，都**小于等于**它左右孩子节点的值；

二叉堆的根节点叫做 **堆顶**，最后一个元素叫 **堆尾**。



大根堆



小根堆

二叉堆的实现

其实就是用我们之前 day1 讲义中的 `priority_queue` 还请同学们仔细回顾一下之前内容，为了节省讲义篇幅这里不再多赘述。我们重点在这里要讲一个动态维护第 k 大数的问题

排序查询（动态维护第k大数）

- 当执行完 3 操作后，2 操作中的输出**开头元素**就是**输出 A 中的最小值**。
- 在执行任意次数的 1 或 2 操作后，只要不执行 3 操作，那么 1 操作所添加的元素仍是按原添加顺序放入 A 中的，不是有序的。

于是，我们可以考虑将放入 A 中的数据分成两部分：

- 一部分是排序好的，用优先队列（小根堆）去维护；
- 另一部分是放入了，但是还是按原顺序排列的，没参与排序的，用队列维护；
- 注意，用优先队列（小根堆）维护的元素出现时间早于队列维护的部分的出现时间。

这样：

- 当执行 1 操作时，新元素入队列；
- 执行 2 操作时，先去堆中查询最小值，也即堆首，如果堆空了，去队列中查询队列的开头元素，也即队首。
- 执行 3 操作时，我们把队列中元素移入堆中即可。

如此，时间复杂度为 $O(n+q\log n)$。

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

int n, op, t;
priority_queue<int, vector<int>, greater<int> > p; // 小顶堆，用于存储元素
queue<int> q; // 队列，用于存储元素

int main() {
```

```
    cin.tie(0); // 解除 cin 和 cout 的绑定，提高输入输出效率
    cin >> n; // 输入操作次数

    for(int i = 1; i <= n; i++) { // 循环进行 n 次操作
        cin >> op; // 输入操作类型
        if(op == 1) { // 如果操作类型为 1，表示插入元素到队列中
            cin >> t; // 输入要插入的元素
            q.push(t); // 将元素插入队列
        } else if(op == 2) { // 如果操作类型为 2，表示弹出一个元素
            if(!p.empty()) { // 如果小顶堆不为空，说明之前进行过插入操作
                cout << p.top() << '\n'; // 输出小顶堆的顶部元素（最小值）
                p.pop(); // 弹出小顶堆的顶部元素
            } else { // 如果小顶堆为空，说明之前只进行过插入队列的操作
                cout << q.front() << '\n'; // 输出队列的首元素
                q.pop(); // 弹出队列的首元素
            }
        } else { // 如果操作类型不是 1 或 2，表示将队列中的所有元素移动到小顶堆中
            while(!q.empty()) { // 当队列不为空时，将队列中的元素依次插入到小顶堆中
                p.push(q.front()); // 将队列的首元素插入到小顶堆中
                q.pop(); // 弹出队列的首元素
            }
        }
    }

    return 0;
}
```

树状数组与线段树

树状数组

基本思想

> *二进制唯一分解 + 前缀和*

根据任意正整数关于 2 的不重复次幂的唯一分解性质，若一个正整数 x 的二进制表示为 10101，其中等于 1 的位是 $0, 2, 4$，则正整数 x 可以被“二进制分解”成 $2^4 + 2^2 + 2^0$。进一步地，区间 $[1, x]$ 可以分成 $O(\log x)$ 个小区间:

- 长度为 2^4 的小区间 $[1, 2^4]$。
- 长度为 2^2 的小区间 $[2^4 + 1, 2^4 + 2^2]$。
- 长度为 2^0 的小区间 $[2^4 + 2^2 + 1, 2^4 + 2^2 + 2^2 + 2^0]$。

树状数组就是一种基于上述思想的数据结构，其基本用途是 **维护序列的前缀和**。对于区间 $[1, x]$，树状数组将其分为 $\log x$ 个子区间，从而满足快速询问区间和。

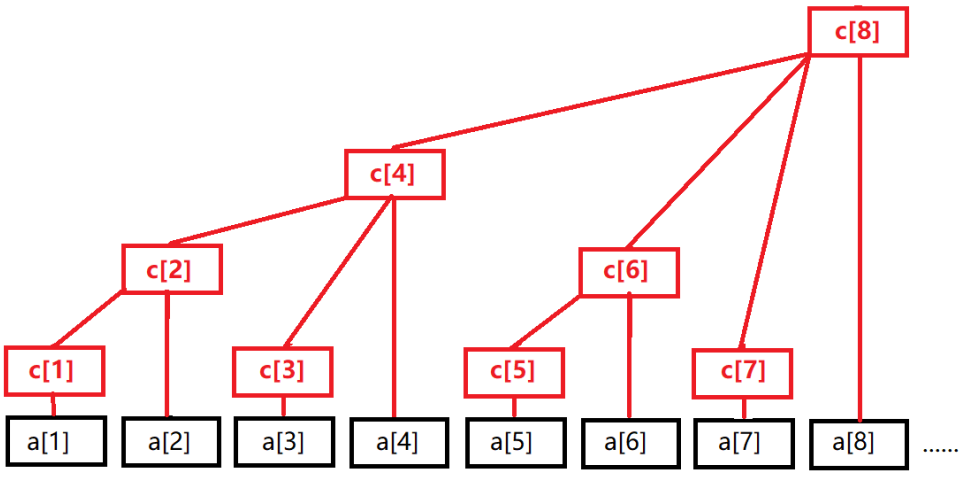
基本操作

由上文可知，这些子区间的共同特点是：若区间结尾为 R，则区间长度就等于R的“二进制分 解”下最小的 2 的次幂，我们设为 `lowbit(R)` 。

对于给定的序列 $a[]$，我们建立一个数组 c，其中 $c[x]$ 保存 **序列 $a[]$ 的区间 $[x - lowbit(x) + 1, x]$ 中所有数的和**，即:

$c[x] = \sum_{i=x-lowbit(x)+1}^{x} a[i]$

事实上，数组 c 可以看作一个如下图所示的 **树形结构**，图中最下边一行是 n 个叶结点($n = 8$)，代表数值 $a[1 \sim n]$。(如果 n 不是 2 的整次幂，那么树状数组就是一个具有同样性质的 **森林结构**)



该结构满足以下性质:

1. 每个内部结点 $c[x]$ 保存以它为根的子树中所有叶结点的和，$c[x]=\sum_{i=x-lowbit(x)+1}^{x}a[i]$。
2. 每个内部结点 $c[x]$ 的子结点个数等于 $lowbit(x)$ 的位数。
3. 除树根外，每个内部结点 $c[x]$ 的 **父结点** 是 $c[x+lowbit(x)]$。
4. 树的深度为 $O(\log N)$。

1. 求 $lowbit(n)$

`lowbit` 操作是树状数组的最基础的操作，也是为什么我们说树状数组中利用了二进制唯一分解的原因（实现上同时利用了负数的补码存储机制）。

$lowbit(n)$ 的含义是：取出非负整数 n 在二进制表示下最低位的 1 以及它后边的 0 构成的数值（如果记为第 k 位，那么就是求 2^k），其代码实现如下：

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

具体过程与原理分析如下：

设 $n>0$，n 的第 k 位是 1，第 $0\sim k-1$ 位都是 0。

为了实现 `lowbit` 运算，先把 n 取反，此时第 k 位变为 0，第 $0\sim k-1$ 位都是 1。再令 $n=n+1$，此时因为进位，第 k 位变为 1，第 $0\sim k-1$ 位都是 0。在上面的取反加 1 操作后，n 的第 $k+1$ 到最高位恰好与原来相反，所以 $n\&(\sim n+1)$ 仅有第 k 位为 1，其余位都是 0。而在 **补码** 表示

下，$-n = n-1$，因此 `lowbit(n)=n&(~n+1)=n&(-n)` 。

2. 对某个元素进行加法操作

树状数组支持单点增加，意思是给序列中的某个数 $a[x]$ 加上 y，同时正确维护序列的前缀和。

根据上面给出的树形结构和它的性质，只有结点 $c[x]$ 及其 **所有祖先结点** 保存的“区间和”包含 $a[x]$，而任意一个结点的祖先至多只有 $\log n$ 个，我们逐一对它们的数组 c 值进行更新即可。

下面的代码在 $O(\log n)$ 时间内执行单点增加操作。

```
void add(int x, int y){
    while(x<=n) {
        c[x] += y;
        x+=lowbit(x);
    }
}
```

特别说明： 对于 c 数组的初始化，为了简便，一般将 c 数组初始化为 0，并且在输入 $a[i]$ 的时候执行上面的 `add(i, a[i])` 操作，以此来实现 c 数组的初始化。

3. 查询前缀和

树状数组支持查询前缀和，即序列 a 第 $1 \sim x$ 个数的和。按照我们刚才提出的方法，应该对 x 进行二进制拆分，求出 x 的二进制表示中每个等于 1 的位，把 $[1,x]$ 分成 $O(\log n)$ 个小区间，而每个小区间的区间和都已经保存在数组 c 中。

下面的代码在 $O(\log n)$ 时间复杂度下查询前缀和：

```
int sum(int x) {
    int res = 0;
    while(x) {
        res += c[x];
        x -= lowbit(x);
    }
    return res;
}
```

4. 查询区间和

调用以上的查询前缀和操作，就可以在 $O(\log n)$ 时间复杂度下查询区间和：

```
// 求 a[x~y] 的和
int res = sum(y) - sum(x-1);
```

注意事项： 树状数组的下标不能从 0 开始使用，因为 $lowbit(0) = 0$，无法处理 0，会导致程序死循环。

5. 区间修改，单点查询

> *区间修改：把 $x \sim y$ 区间内的所有值全部加上 k 或者减去 k。*
>
> *单点查询：询问某个点的值。*

这种时候该怎么做呢。如果用上面的树状数组，就必须把 $x \sim y$ 区间内每个值都更新，这样的时间复杂度肯定是不行的。所以这个时候，就不能再用数据的值建树了，这里我们引入差分， 利用差分建树。**差分可以将区间修改转化为（两个）单点修改，将单点查询转化为查询前缀和**。举例如下：

假设我们规定 $a[0] = 0$;

则有 $a[i] = \sum_{j=1}^{i} D[j], (D[j] = a[j] - a[j-1])$ 即前面 i 项的差值和，这个有什么用呢？例如对于下面这个数组

- a[] = 1 2 3 5 6 9
- D[] = 1 1 1 2 1 3

如果我们把 $[2, 5]$ 区间内值加上 2，则变成了

- a[] = 1 4 5 7 8 9
- D[] = 1 3 1 2 1 1

发现了没有，当某个区间 $[x, y]$ 值改变了，区间内的差值是不变的，只有 $D[x]$ 和 $D[y+1]$ 的值发生改变。

这样，原来要更新一个区间的值变成了只需要更新两个点，如此转化之后，就好处理了。

所以我们就可以利用这个性质对 $D[]$ 数组建立树状数组。

```cpp
const int N = 1e6 + 5;
long long n, q, a[N], c[N];

long long lowbit(int x){
    return x&-x;
}
void add(int x, int y) {   // D[x] += y
    while(x<=n) {
        c[x] += y;
        x += lowbit(x);
    }
}
long long sum(int x) {   // 求D[1 ~ i]的和，即 a[i] 值
    long long res = 0;
    while(x) {
        res += c[x];
        x -= lowbit(x);
    }
    return res;
}
int main(){
    ios::sync_with_stdio(0);
    cin.tie(0);
    cin >> n >> q;
    for(int i = 1; i <= n; i++) {
        cin >> a[i];
        add(i, a[i]-a[i-1]);  //输入初值的时候，也相当于更新了值，同时构建的是差分数组上的树状数组
    }
    while(q--){
        int op, x, y, z;
        cin >> op >> x;
        if(op == 1) {
            cin >> y >> z;
            add(x, z);       //a[x] - a[x-1] 增加 k
            add(y+1, -z);    //a[y+1] - a[y] 减少 k
        } else {
            cout << sum(x) << '\n';   // 差分数组上的前缀和就是原数组 a[x]
```

```
        }
    }
}
```

6. 区间修改，区间查询

上面我们说的用差值建树状数组，得到的是某个点的值，那如果我们既要区间更新，又要区间查询怎么办。这里我们还是利用差分，由差分数组的定义可知：

$$
\begin{aligned}
sum_{i=1}^{n} a[i] &= \sum_{i=1}^{n}\sum_{j=1}^{i} D[j] \\
&= D[1] + (D[1] + D[2]) + \ldots + (D[1] + D[2] + \ldots + D[n]) \\
&= n * D[1] + (n-1) * D[2] + \ldots + D[n] \\
&= n * (D[1] + D[2] + \ldots + D[n]) - (0 * D[1] + 1 * D[2] + \ldots + (n-1) * D[n]) \\
&= n * \sum_{i=1}^{n} D[i] - \sum_{i=1}^{n} (i-1) * D[i]
\end{aligned}
$$

同单点修改，我们只需要开两个树状数组，一个维护 $D[i]$，一个维护 $(i-1) * D[i]$ 就行了。

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

const int N = 1e6+5;
int n, q, a[N];
long long c1[N];   // 维护 (D[1] + D[2] + ... + D[n])
long long c2[N];   // 维护 (0*D[1] + 1*D[2] + ... + (n-1)*D[n])

// 获取 x 的最低位 1 的位置
int lowbit(int x) {
    return x & -x;
}

// 在树状数组中进行单点更新
void add(int x, int y) {
    long long p = x;  // 因为 x 不变，所以要先保存
    while(x <= n) {
        c1[x] += y;
        c2[x] += (p - 1) * y;
```

```
        x += lowbit(x);
    }
}

// 计算前缀和
long long sum(int x) {
    long long res = 0, p = x;
    while(x) {
        res += p * c1[x] - c2[x];
        x -= lowbit(x);
    }
    return res;
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    // 输入数组长度和查询次数
    cin >> n >> q;

    // 输入数组元素
    for(int i = 1; i <= n; i++) {
        cin >> a[i];
        add(i, a[i] - a[i - 1]);
    }

    while(q--) {
        int op, l, r, x;
        cin >> op >> l >> r;
        if(op == 1) {
            cin >> x;
            add(l, x);
            add(r + 1, -x);
        } else {
            cout << sum(r) - sum(l - 1) << '\n';
        }
    }
```

```
    return 0;
}
```

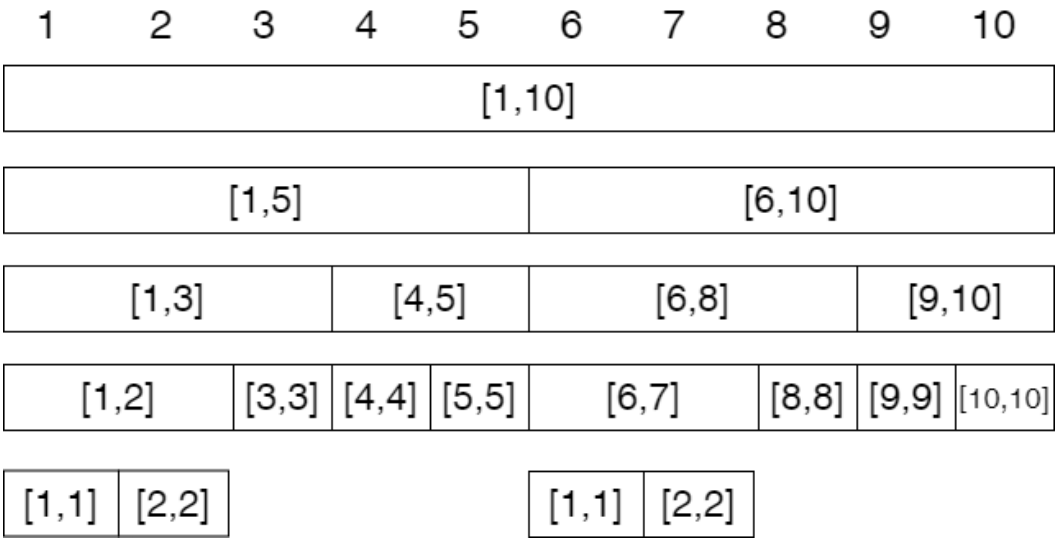
线段树

概念引入

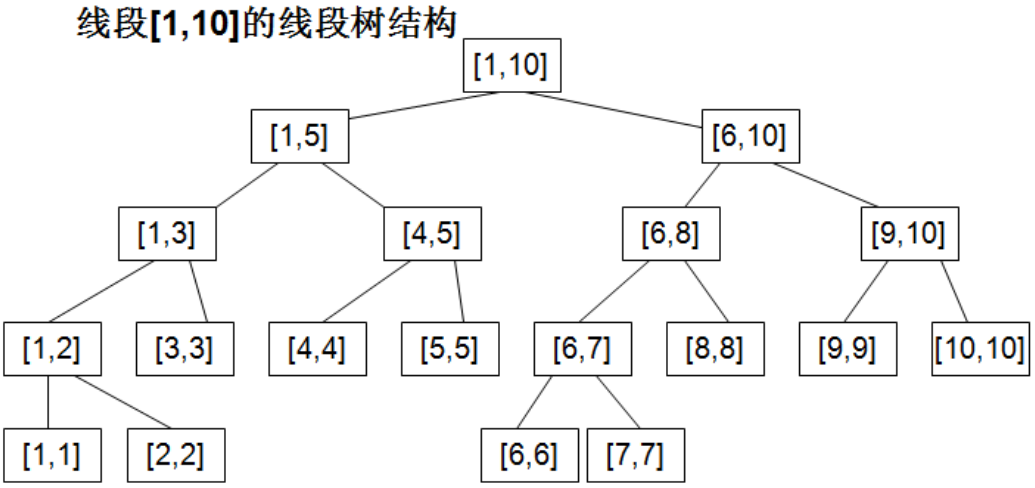
概括地说，线段树是一种基于 **分治思想** 的 **二叉树** 结构，用于在区间上进行信息统计。与按照二进制位（2 的次幂）进行区间划分的树状数组相比，线段树是一种更加通用的数据结构，功能比树状数组更强大：

1. 线段树的每个节点都代表一个 **区间/线段**（每个节点都维护对应子树的一些信息）。
2. 线段树具有唯一的根节点，代表的区间是整个统计范围，如 $[1, N]$。
3. 线段树的叶节点都代表一个长度为 1 的元区间 $[x, x]$。
4. 对于 **每个内部节点 $[l, r]$，它的左子节点是 $[l, mid]$，右子节点是 $[mid + 1, r]$，其中 $mid = (l + r) >> 1$**
5. 线段树是一棵二叉树，一个区间每次折一半往下分，包含 n 个元素的线段树，最多分 $\log n$ 次就到达最低层。需要查找一个点或者区间的时候，顺着结点往下找，**最多 $\log n$ 次就能找到**。
6. 结点所表示的“线段”的值，可以是区间和、最值或者其他根据题目灵活定义的值。

比如说一个长度为 10 的线段，我们可以表示成这样：



上面例子中，对应的线段树结构如下图所示：



上图展示了一棵线段树。可以发现，除去树的最后一层，整棵线段树一定是一棵 **完全二叉树**，树的深度为 $O(\log N)$。因此，我们可以按照完全二叉树的存储方法来保存节点间的关系：

1. 根节点编号为 1。
2. 编号为 x 的节点的左子节点编号为 $2x$，右子节点编号为 $2x+1$。

这样一来，我们就能简单地使用一个结构体数组来保存线段树。当然，树的最后一层节点在数组中保存的位置不是连续的，直接空出数组中多余的位置即可。

在理想情况下，N 个叶节点的满二叉树有 $N+N/2+N/4+\cdots+2+1=2N-1$ 个节点。因为在上述存储方式下，最后还有一层可能产生了空余，所以保存线段树的数组长度要 **不小于** $4N$ 才能保证不会越界。

考查每个线段 $[L,R]$，L 是左端，R 是右端：

1. $L=R$，说明这个结点只有一个元素，它是一个叶子结点。
2. $L<R$，说明这个结点代表的不只一个点，那么它有两个儿子，左儿子区间是 $[L,Mid]$，右儿子区间是 $[Mid+1,R]$，其中 $Mid=(L+R)/2$。例如：$L=1,R=5,Mid=3$，左儿子是 $[1,3]$，右儿子是 $[4,5]$。

线段树的构建

线段树是除了最后一层之外，其他层都是满的一种树，这点和堆（完全二叉树）是一样的，因此可以使用(静态)一维数组来保存整棵树。

对于编号是 x 的节点，和其相关的节点关系如下：

$$\text{编号是}x\begin{cases}\text{父亲节点：}\lfloor\frac{x}{2}\rfloor\text{,或写为}x>>1\\ \text{左儿子：}2x\text{,或写为}x<<1\\ \text{右儿子：}2x+1\text{,或写为}x<<1|1\end{cases}$$

> *PS:上述两种写法都一样，表面上第二种速度最快，但现代编译器会将除二和加一操作优化为右式，所以编译后速度一样。（不过第二种看着更高级）*

编码时，可以定义标准的二叉树数据结构；在竞赛中一般用静态数组实现的满二叉树，虽然比较浪费空间，但是编码稍微简单一点。

下面给的代码，都用静态分配的 $tree[]$。父结点和子结点之间的访问非常简单，缺点是最后一行有大量结点被浪费。

```
// 定义根结点是tree[1]，即编号为1的结点是根
// 第一种方法：定义二叉树数据结构
struct{
    int l, r, data;          // 用tree[i].data记录线段i的最值或区间和
} tree[N*4];                 // 分配静态数组，开4倍大

// 第二种方法：直接用数组表示二叉树，更节省空间，在竞赛中一般用此方法
int tree[N*4];               // 用tree[i]记录线段i的最值或区间和
// 结点p是父结点，ls(p)是左儿子，rs(p)是右儿子
inline int ls(int x){ return x<<1;  }     // 左儿子 l，编号是 p*2
inline int rs(int x){ return x<<1|1;}     // 右儿子 r，编号是 p*2+1
```

线段树的基本用途是对序列进行维护，支持查询与修改指令。给定一个长度为 N 的序列 A，我们可以在区间 $[1,N]$ 上建立一棵线段树，每个叶节点 $[i,i]$ 保存 $A[i]$ 的值。线段树的二叉树结构可以很方便地 **从下往上** 传递信息。以区间最大值问题为例，记 $dat(l,r)$ 等于 $\max_{l\le i\le r}\{A[i]\}$，显然 $dat(l,r)=max(dat(l,mid),dat(mid+1,r))$。

下面这段代码建立了一棵线段树并在每个节点上保存了对应区间的最大值。

```
inline void push_up(int p){                         // 从下往上传递区间值
    // tree[p] = tree[p<<1] + tree[p<<1|1];      // 求区间和
    tree[p] = max(tree[p<<1], tree[p<<1|1]);     // 求区间最大值
}
void build(int p,int l,int r){              // 结点编号 p 指向区间[l, r]
    if(l==r) {                                      // 最底层的叶子，存叶子的值
        tree[p]=a[l];
        return;
    }
    int mid = (l + r) >> 1;                         // 分治：折半
    build(p<<1, l, mid);                            // 递归左儿子
    build(p<<1|1, mid+1, r);                        // 递归右儿子
    push_up(p);                                     // 从下往上传递区间值
}
// 建树的入口
build(1, 1, n);
```

线段树的单点修改

操作：形如 `C x k`，把区间的第 x 位加上 k

这个代码比较好写，**从根节点开始**，看这个 x 是在左子树还是在右子树，递归查找，直到找到该叶子节点。需要注意的是在返回的时候，**最后不能漏掉 push_up 操作，来更新所有受到该叶子节点影响的父节点**。

```
// 单点修改，x 位增加 k
void modify(int x, int k, int p, int l, int r)  // l: p 对应区间的左端点， r: p 对应区间的右端点
{
    if(l == r) {   // 叶子节点
        tree[p] += k;
        return;
    }
```

```
    int mid = (l + r) >> 1;
    if(x <= mid)  modify(x, k, p*2, l, mid);   // 在左子树
    else  modify(x, k, p*2+1, mid+1, r);         // 在右子树
    push_up(p);    // 从下往上传递区间值
}

// 调用入口
modify(x, k, 1, 1, n);
```

线段树的区间查询

操作：形如 `Q l r`，查询区间 $[l,r]$ 上的信息（区间和、最大值、最小值等）。

根据分治的思想，某一个区间上的和目标区间的交集和的计算方式为：

[交集和 = 左区间上的交集部分和 + 右区间上的交集部分和]

具体实现起来，总体思路为：

1. **如果这个区间被完全包括在目标区间里面，直接返回这个区间的值**
2. **如果这个区间的左儿子和目标区间有交集，那么搜索左儿子**
3. **如果这个区间的右儿子和目标区间有交集，那么搜索右儿子**

以求解区间和为例，实现的示例代码如下：

```
// 区间查询（区间和）
int query(int ql, int qr, int p, int l, int r)
{
    if(ql <= l && r <= qr)   return tree[p];   // 当前区间完全包含于查询的区间，直接返回该区间值

    int ans = 0;
    int mid = (l + r) >> 1;
    if(ql <= mid)                                  // 如果这个区间的左儿子和目标区间又交集，那么搜索左儿子
        ans += query(ql, qr, p<<1, l, mid);
    if(mid < qr)                                    // 如果这个区间的右儿子和目标区间又交集，那么搜索右儿子
        ans += query(ql, qr, p<<1|1, mid+1, r);
```

```
    return ans;
}
// 求区间 [L,R] 的和的调用方式：query(L, R, 1, 1, n)

// 求区间和 —— 结构体示例写法
int query(int i,int l,int r){
    int s = 0;
    if(tree[i].l>=l && tree[i].r<=r)             // 如果这个区间被完全包括在目标区间里面，直接返回这个区间的值
        return tree[i].sum;
    int s=0;
    if(tree[i*2].r>=l)  s+=query(i*2,l,r);       // 如果这个区间的左儿子和目标区间又交集，那么搜索左儿子
    if(tree[i*2+1].l<=r)  s+=query(i*2+1,l,r);   // 如果这个区间的右儿子和目标区间又交集，那么搜索右儿子
    return s;
}
```

以查询区间 $[l,r]$ 的和为例，给出代码。查询递归到某个结点 p（p 表示的区间是 $[pl,pr]$）时，有 3 种情况：

- $[l,r]$ 完全覆盖了 $[pl,pr]$，即 $l \le pl \le pr \le r$，直接返回节点 p 存储的值即可。
- $[l,r]$ 与 $[pl,pr]$ 不相交，即 $l > pr$ 或者 $r < pl$，退出（**这种情况在代码里不存在，因为初始区间 [1,n] 肯定包含目标区间**）。
- $[l,r]$ 与 $[pl,pr]$ 部分重叠，分别搜左右子结点。$l < pr$，继续递归左子结点，例如查询区间 $[4,9]$，与第 2 个结点 $[1,5]$ 有重叠，因为 $4 < 5$。$R > pl$，继续递归右子结点，例如 $[4,9]$ 与第 3 个结点 $[6,10]$ 有重叠，因为 $9 > 6$。

这种查询过程会把询问的区间 $[l,r]$ 在线段树上划分成 $O(\log N)$ 个节点，然后再求解区间信息。

区间修改，单点查询

区间修改和单点查询，我们的思路就变为：如果把这个区间加上 k，相当于把这个区间涂上一个 k 的标记，然后单点查询的时候，就从上跑到下，把沿路的标记加起来就好。因此，**这类操作里，不需要 push_up 操作！建树时也是如此！**

这里面给区间贴标记的方式与上面的区间查找类似，原则还是那三条，只不过第一条： **如果这个区间被完全包括在目标区间里面，直接返回这个区间的值** 变为了 **如果这个区间被完全包括在目标区间里面，将这个区间增加k**。

区间修改代码：

```
// 区间修改：[ql, qr] 增加 k
void modify(int ql, int qr, int p,int l,int r, int k){
    if(ql <= l && r <= qr)   // 如果这个区间被完全包括在目标区间里面，将这个区间标记k
    {
        tr[p] += k;
        return ;
    }
    int mid = (l + r) >> 1;
    if(ql <= mid)    // 包含部分左区间
        modify(ql, qr, p*2, l, mid, k);
    if(qr > mid)     // 包含部分右区间
        modify(ql, qr, p*2+1, mid+1, r, k);
    // 注意，无 push_up 操作
}
```

单点查询，查找区间的第 x 位。这个更好实现，就是 x 哪往哪跑，把路径上所有的值加上就好了：

```
int query(int x, int p, int l, int r){
    int ans = 0;
    ans += tr[p];
    if(l == r)
        return ans;
    int mid = (l + r )>>1;
    if(x <= mid) ans += query(x, p << 1, l, mid);
    else ans += query(x, p<<1|1, mid+1, r);
    return ans;
}
```

这样的线段树，除了求和，还可以求区间最小最大值，还可以区间染色。

线段树的区间修改 + 区间查询

本节介绍线段树的核心技术“**lazy-tag**”，并给出“区间修改+区间查询”问题的模板。

在树状数组这一节中，已经指出区间修改比单点修改复杂很多。最普通区间修改，例如对一个数列的 $[l,r]$ 区间内每个元素统一加上 d，如果在线段树上，一个个地修改这些元素，那么 m 次区间修改的复杂度是 $O(mn\log n)$ 的。

试想，如果我们在一次修改指令中发现节点 p 代表的区间 $[pl,pr]$ 被修改区间 $[l,r]$ 完全覆盖，并且逐一更新了子树 p 中的所有节点，但是在之后的查询指令中却根本没有用到 $[l,r]$ 的子区间作为候选答案，那么更新 p 的整棵子树就是徒劳的。

换言之，我们在执行修改指令时，同样可以在 $l\le pl\le pr\le r$ 的情况下立即返回，只不过在回溯之前向节点 p 增加一个标记，标识“该节点曾经被修改，但其子节点尚未被更新”。

如果在后续的指令中，需要从节点 p 向下递归，我们再检查 p 是否具有标记。若有标记,就根据标记信息更新 p 的两个子节点，同时为 p 的两个子节点增加该标记，然后清除 p 的标记（标记往下传）。这个标记，就被称为 `lazy-tag` 。

lazy-tag

lazy-tag（又称懒惰标记，或者延迟标记）。当修改的是一个线段区间时，就只对这个线段区间进行整体上的修改，其内部每个元素的内容先不做修改，**只有当这个线段区间的一致性被破坏时**，才把变化值传递给下一层的子区间。每次区间修改的复杂度是 $O(\log n)$ 的，一共 m 次操作，总复杂度是 $O(m\log n)$ 的。区间 i 的 $lazy$ 操作，用 $tag[i]$ 记录。

下面举例说明区间修改函数 $update()$ 的具体步骤。例如把 $[4,9]$ 区间内的每个元素加 3，执行步骤是：

（1）左子树递归到结点 5，即区间 $[4,5]$，完全包含在 $[4,9]$ 内，打标记 $tag[5]=3$，更新 $tree[5]$ 为 20，不再继续深入；

（2）左子树递归返回，更新 $tree[2]$ 为 30；

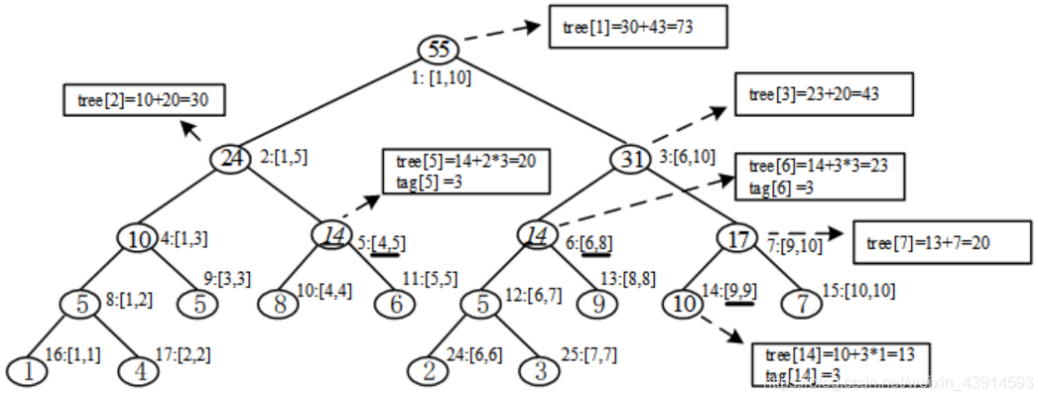
（3）右子树递归到结点 6，即区间 $[6,8]$，完全包含在 $[4,9]$ 内，打标记 $tag[6]=3$，更新 $tree[6]$ 为 23。

（4）右子树递归到结点 14，即区间 $[9,9]$，打标记 $tag[14]=3$，更新 $tree[14]=13$；

（5）右子树递归返回，更新 $tree[7]=20$；继续返回，更新 $tree[3]=43$；

（6）返回到根节点，更新 $tree[1]=73$。

详情见下图。



push down

push_down()函数。在进行多次区间修改时，一个结点需要记录多个区间修改。而这些区间修改往往有冲突，例如做 2 次区间修改，一次是 $[4,9]$，一次是 $[5,8]$，它们都会影响 $5:[4,5]$ 这个结点。第一次修改 $[4,9]$ 覆盖了结点 5，用 $tag[5]$ 做了记录；而第二次修改 $[5,8]$ 不能覆盖结点 5，需要再向下搜到结点 $11:[5,5]$，从而破坏了 $tag[5]$，此时原 $tag[5]$ 记录的区间统一修改就不得不往它的子结点传递和执行了，传递后 $tag[5]$ 失去了意义，需要清空。所以 $lazy-tag$ 的主要操作是 **解决多次区间修改**，用 `push_down()` 函数完成。它首先检查结点 p 的 $tag[p]$，如果有值，说明前面做区间修改时给 p 打了 tag 标记，接下来就把 $tag[p]$ 传给左右子树，然后把 $tag[p]$ 清零。

`push_down()` 函数不仅在“区间修改”中用到，在“区间查询”中同样用到。

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long
const int N = 1e6 + 10;
ll a[N];      //记录数列的元素，从a[1]开始
ll tree[N << 2]; //tree[i]: 第i个结点的值，表示一个线段区间的值，例如最值、区间和
```

```
ll tag[N << 2]; //tag[i]: 第i个结点的lazy-tag，统一记录这个区间的修改

void push_up(ll p) {             //从下往上传递区间值
    tree[p] = tree[p*2] + tree[p*2+1];
    // 本题是区间和。如果求最小值，改为: tree[p] = min(tree[ls(p)], tree[rs(p)]);
}
void build(ll p, ll l, ll r) {   // 建树。p是结点编号，它指向区间[pl, pr]
    tag[p] = 0;                  // lazy-tag标记
    if (l == r) {
        tree[p] = a[l];    // 最底层的叶子，赋值
        return;
    }
    ll mid = (l + r) >> 1;       // 分治: 折半
    build(p*2, l, mid);          // 左儿子
    build(p*2+1, mid + 1, r);    // 右儿子
    push_up(p);                  // 从下往上传递区间值
}
inline void addtag(ll p, ll l, ll r, ll d) {  // 给结点p打tag标记，并更新tree
    tag[p] += d;                    // 打上tag标记
    tree[p] += d * (r - l + 1);     // 计算新的tree
}
inline void push_down(ll p, ll l, ll r) {  //不能覆盖时，把tag传给子树
    if (tag[p]) {                     //有tag标记，这是以前做区间修改时留下的
        ll mid = (l + r) >> 1;
        addtag(p*2, l, mid, tag[p]); //把tag标记传给左子树
        addtag(p*2+1, mid + 1, r, tag[p]); //把tag标记传给右子树
        tag[p] = 0;                       //p自己的tag被传走了，归0
    }
}
void modify(ll ql, ll qr, ll p, ll l, ll r, ll d) { //区间修改: 把[L, R]内每个元素加上d
```

```
    if (ql <= l && r <= qr) { // 完全覆盖，直接返回这个结点，它的子树不用再深入了
        addtag(p, l, r, d);   // 给结点p打tag标记，下一次区间修改到p这个结点时会用到
        return;
    }
    push_down(p, l, r);                     // 如果不能覆盖，把tag传给子树
    ll mid = (l + r) >> 1;
    if (ql <= mid) modify(ql, qr, p*2, l, mid, d); //递归左子树
    if (qr > mid)  modify(ql, qr, p*2+1, mid + 1, r, d); //递归右子树
    push_up(p);                                 //更新
}
ll query(ll ql, ll qr, ll p, ll l, ll r) {
    //查询区间[L,R]；p是当前结点（线段）的编号，[pl,pr]是结点p表示的线段区间
    if (ql <= l && r <= qr) return tree[p];    // 完全覆盖，直接返回
    push_down(p, l, r);                     // 不能覆盖，把tag传给子树
    ll res = 0;
    ll mid = (l + r) >> 1;
    if (ql <= mid) res += query(ql, qr, p*2, l, mid);       //左子节点有重叠
    if (qr > mid)  res += query(ql, qr, p*2+1, mid + 1, r); //右子节点有重叠
    return res;
}
int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    ll n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i];
    build(1, 1, n);                 //建树
    while (m--) {
        ll op, l, r, x;
        cin >> op;
```

```
        if (op == 1) {                  //区间修改：把[L,R]的每个元素加上x
            cin >> l >> r >> x;
            modify(l, r, 1, 1, n, x);
        } else {                        //区间询问：[L,R]的区间和
            cin >> l >> r;
            cout << query(l, r, 1, 1, n) << '\n';
        }
    }
    return 0;
}
```

单点修改区间查询

树状数组

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

const int N = 1e6+5;
long long n, q, a[N], c[N]; // 定义变量和数组：n 为数组长度，q 为查询次数，a 为输入数组，c 为树状数组

// 获取 x 的最低位 1 的位置
int lowbit(int x){
    return x&(-x);
}

// 在树状数组中进行单点更新
void add(int x, int y) {
    while(x <= n) {
        c[x] += y;
        x += lowbit(x);
    }
}

// 计算前缀和
long long sum(int x) {
```

```
    long long res = 0;
    while(x) {
        res += c[x];
        x -= lowbit(x);
    }
    return res;
}

int main()
{
    ios::sync_with_stdio(0); // 关闭输入输出流的同步，提高读入效率
    cin.tie(0); // 解除 cin 和 cout 的绑定，提高读入效率

    cin >> n >> q; // 输入数组长度和查询次数
    for(int i = 1; i <= n; i++) {
        cin >> a[i]; // 输入数组元素
        add(i, a[i]); // 在树状数组中进行单点更新
    }

    // 处理查询
    for(int i = 1, op, x, y; i <= q; i++) {
        cin >> op >> x >> y; // 输入操作类型、区间端点
        if(op == 1) {
            add(x, y); // 更新操作：将第 x 个元素增加 y
        } else {
            cout << sum(y) - sum(x-1) << '\n'; // 查询操作：输出
区间 [x, y] 的和
        }
    }

    return 0;
}
```

线段树

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

const int MAXN = 4000010;
```

```cpp
long long a[MAXN];

struct SegmentTree {
    long long l, r, n;
} tree[MAXN];

// 向上更新节点信息
void push_up(long long p) {
    tree[p].n = tree[2 * p].n + tree[2 * p + 1].n;
}

// 建立线段树
void build(long long p, long long l, long long r) {
    tree[p].l = l;
    tree[p].r = r;
    if (l == r) {
        tree[p].n = a[l];
        return;
    }
    long long mid = (l + r) >> 1;
    build(p << 1, l, mid);
    build(p << 1 | 1, mid + 1, r);
    push_up(p);
}

// 查询区间和
long long query(long long p, long long l, long long r) {
    long long s = 0;
    if (tree[p].l >= l && tree[p].r <= r)
        return tree[p].n;
    long long mid = (tree[p].r + tree[p].l) >> 1;
    if (l <= mid)
        s += query(p << 1, l, r);
    if (r > mid)
        s += query(p << 1 | 1, l, r);
    return s;
}
```

```
// 修改节点值
void modify(long long p, long long l, long long k) {
    if (tree[p].l == tree[p].r) {
        tree[p].n += k;
        return;
    }
    long long mid = (tree[p].l + tree[p].r) >> 1;
    if (l <= mid)
        modify(p * 2, l, k);
    else
        modify(2 * p + 1, l, k);
    push_up(p);
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    long long i, n, k, t, l, r, x;
    cin >> n >> k;
    for (i = 1; i <= n; i++) {
        cin >> a[i];
    }
    build(1, 1, n);
    for (i = 1; i <= k; i++) {
        cin >> t >> l;
        if (t == 1) {
            cin >> x;
            modify(1, l, x);
        } else {
            cin >> r;
            cout << query(1, l, r) << '\n';
        }
    }
    return 0;
}
```

区间修改单点查询

树状数组

```
#include <bits/stdc++.h>
using namespace std; // 使用 std 命名空间

const int N = 1000010;

long long t[5 * N];
int n, m;

// 获取 x 的最低位 1 的位置
int lowbit(int x){
    return x & (-x);
}

// 在树状数组中进行单点更新
void modify(int x, int y){
    for(int i = x; i <= n; i += lowbit(i)){
        t[i] += y;
    }
}

// 查询前缀和
long long query(int x){
    long long ans = 0;
    for(int i = x; i; i -= lowbit(i)){
        ans += t[i];
    }
    return ans;
}

int main(){
    ios::sync_with_stdio(false); // 关闭输入输出流的同步，提高读入效率
    cin.tie(nullptr); // 解除 cin 和 cout 的绑定，提高读入效率
    cout.tie(nullptr); // 解除 cin 和 cout 的绑定，提高读入效率
```

```
    cin >> n >> m;
    int last = 0;
    for(int i = 1; i <= n; i ++ ){
        int x;
        cin >> x;
        modify(i, x - last); // 在树状数组中进行单点更新
        last = x;
    }
    while(m -- ){
        int a, l, r, x;
        cin >> a;
        if(a == 1){
            cin >> l >> r >> x;
            modify(l, x); // 更新区间 [l, r] 的值为 x
            modify(r + 1, -x); // 恢复 r+1 处的值为原来的值
        } else {
            cin >> x;
            cout << query(x) << '\n'; // 查询前缀和
        }
    }
    return 0;
}
```

线段树

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

const int N = 1e6+5;
int n, q, op, l, r, x;
long long t[N<<2], a[N];

// 建立线段树
void build(int p, int l, int r) {
    if (l == r) {
        t[p] = a[l];
        return;
    }
```

```
    int mid = (l+r) >> 1;
    build(p*2, l, mid);
    build(p*2+1, mid+1, r);
    // 此处可以添加其他初始化操作
}

// 查询区间和
long long query(int p, int l, int r, int k) {
    long long ans = 0;
    ans += t[p];
    if (l == r) return ans;
    int mid = (l+r) >> 1;
    if (k <= mid) ans += query(p*2, l, mid, k);
    else ans += query(p*2+1, mid+1, r, k);
    return ans;
}

// 区间修改
void modify(int ql, int qr, int p, int l, int r, int k) {
    if (ql <= l && r <= qr) {
        t[p] += k;
        return;
    }
    int mid = (l+r) >> 1;
    if (ql <= mid) modify(ql, qr, p*2, l, mid, k);
    if (qr > mid) modify(ql, qr, p*2+1, mid+1, r, k);
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    cin >> n >> q;
    for (int i = 1; i <= n; i++) cin >> a[i];
    build(1, 1, n);
    while (q--) {
        cin >> op >> l;
        if (op == 1) {
            cin >> r >> x;
```

```
            modify(l, r, 1, 1, n, x); // 更新区间 [l, r] 的值为
x
        } else {
            cout << query(1, 1, n, l) << '\n'; // 查询位置 l 的
值
        }
    }
    return 0;
}
```

区间修改区间查询

树状数组

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

const int N = 1e6+5;
int n, q, a[N];
long long c1[N];   // 维护 (D[1] + D[2] + ... + D[n])
long long c2[N];   // 维护 (0*D[1] + 1*D[2] + ... + (n-1)*D[n])

// 获取 x 的最低位 1 的位置
int lowbit(int x) {
    return x & -x;
}

// 在树状数组中进行单点更新
void add(int x, int y) {
    long long p = x;  // 因为 x 不变，所以要先保存
    while(x <= n) {
        c1[x] += y;
        c2[x] += (p - 1) * y;
        x += lowbit(x);
    }
}

// 计算前缀和
```

```
long long sum(int x) {
    long long res = 0, p = x;
    while(x) {
        res += p * c1[x] - c2[x];
        x -= lowbit(x);
    }
    return res;
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    // 输入数组长度和查询次数
    cin >> n >> q;

    // 输入数组元素
    for(int i = 1; i <= n; i++) {
        cin >> a[i];
        add(i, a[i] - a[i - 1]);
    }

    while(q--) {
        int op, l, r, x;
        cin >> op >> l >> r;
        if(op == 1) {
            cin >> x;
            add(l, x);
            add(r + 1, -x);
        } else {
            cout << sum(r) - sum(l - 1) << '\n';
        }
    }

    return 0;
}
```

线段树

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long
const int N = 1e6 + 10;
ll a[N];      //记录数列的元素，从a[1]开始
ll tree[N << 2]; //tree[i]: 第i个结点的值，表示一个线段区间的值，例如最值、区间和
ll tag[N << 2]; //tag[i]: 第i个结点的lazy-tag，统一记录这个区间的修改

void push_up(ll p) {             //从下往上传递区间值
    tree[p] = tree[p*2] + tree[p*2+1];
    // 本题是区间和。如果求最小值，改为: tree[p] = min(tree[ls(p)], tree[rs(p)]);
}
void build(ll p, ll l, ll r) {   // 建树。p是结点编号，它指向区间[pl, pr]
    tag[p] = 0;                  // lazy-tag标记
    if (l == r) {
        tree[p] = a[l];    // 最底层的叶子，赋值
        return;
    }
    ll mid = (l + r) >> 1;       // 分治: 折半
    build(p*2, l, mid);          // 左儿子
    build(p*2+1, mid + 1, r);    // 右儿子
    push_up(p);                  // 从下往上传递区间值
}
inline void addtag(ll p, ll l, ll r, ll d) {  // 给结点p打tag标记，并更新tree
    tag[p] += d;                    // 打上tag标记
    tree[p] += d * (r - l + 1);     // 计算新的tree
}
inline void push_down(ll p, ll l, ll r) {  //不能覆盖时，把tag传给子树
    if (tag[p]) {                     //有tag标记，这是以前做区间修改时留下的
        ll mid = (l + r) >> 1;
```

```
        addtag(p*2, l, mid, tag[p]); //把tag标记传给左子树
        addtag(p*2+1, mid + 1, r, tag[p]); //把tag标记传给右子树
        tag[p] = 0;                          //p自己的tag被传走了，归
0
    }
}
void modify(ll ql, ll qr, ll p, ll l, ll r, ll d) { //区间修改:
把[L, R]内每个元素加上d
    if (ql <= l && r <= qr) { // 完全覆盖，直接返回这个结点，它的
子树不用再深入了
        addtag(p, l, r, d);   // 给结点p打tag标记，下一次区间修改
到p这个结点时会用到
        return;
    }
    push_down(p, l, r);                      // 如果不能覆盖，把
tag传给子树
    ll mid = (l + r) >> 1;
    if (ql <= mid) modify(ql, qr, p*2, l, mid, d); //递归左子树
    if (qr > mid)  modify(ql, qr, p*2+1, mid + 1, r, d); //递归
右子树
    push_up(p);                                 //更新
}
ll query(ll ql, ll qr, ll p, ll l, ll r) {
    //查询区间[L,R]；p是当前结点（线段）的编号，[pl,pr]是结点p表示
的线段区间
    if (ql <= l && r <= qr) return tree[p];    // 完全覆盖，直接
返回
    push_down(p, l, r);                      // 不能覆盖，把tag
传给子树
    ll res = 0;
    ll mid = (l + r) >> 1;
    if (ql <= mid) res += query(ql, qr, p*2, l, mid);       //
左子节点有重叠
    if (qr > mid)  res += query(ql, qr, p*2+1, mid + 1, r); //
右子节点有重叠
    return res;
}
int main() {
    ios::sync_with_stdio(0);
```

```
    cin.tie(0);
    ll n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) cin >> a[i];
    build(1, 1, n);                    //建树
    while (m--) {
        ll op, l, r, x;
        cin >> op;
        if (op == 1) {                 //区间修改：把[L,R]的每个元素加上x
            cin >> l >> r >> x;
            modify(l, r, 1, 1, n, x);
        } else {                       //区间询问：[L,R]的区间和
            cin >> l >> r;
            cout << query(l, r, 1, 1, n) << '\n';
        }
    }
    return 0;
}
```

求逆序对

树状数组求逆序对

任意给定一个集合 a，如果用 $t[val]$ 保存数值 val 在集合 a 中出现的次数，那么数组 t 在 $[l,r]$ 上的区间和（即 $\sum_{i=1}^{r} t[i]$）就表示集合 a 中范围在 $[l,r]$ 内的数有多少个。

我们可以在集合 a 的数值范围上建立一个树状数组，来维护 t 的前缀和。这样即使在集合 a 中插入或删除一个数，也可以高效地进行统计。

我们前面已经学习过了逆序对问题以及使用归并排序的解法。对于一个序列 a，若 $i<j$ 且 $a[i]>a[j]$，则称 $a[i]$ 与 $a[j]$ 构成逆序对。按照上述思路，利用树状数组也可以求出一个序列的逆序对个数:

1. 在序列 a 的数值范围上建立树状数组，初始化为全零。
2. 倒序扫描给定的序列 a，对于每个数 $a[i]$:
   (1) 在树状数组中查询前缀和 $[1,a[i]-1]$，累加到答案 ans 中。
   (2) 执行“单点增加”操作，即把位置 $a[i]$ 上的数加 1(相当于 $t[a[i]]++$)，同时正确维护 t 的前缀和。这表示数值 $a[t]$ 又出现了 1 次。

3. ans 即为所求。

```
for(int i = n; i; i--) {
    ans += sum(a[i]-1);
    add(a[i], 1);
}
```

在这个算法中，因为倒序扫描，“已经出现过的数”就是在 $a[i]$ 后边的数，所以我们通过树状数组查询的内容就是“每个 $a[i]$ 后边有多少个比它小”。每次查询的结果之和当然就是逆序对个数。时间复杂度为 $O((N+M)\log M)$，M 为数值范围的大小。

当数值范围较大时，当然可以先进行离散化，再用树状数组进行计算。不过因为离散化本身就要通过排序来实现，所以在这种情况下就不如直接用归并排序来计算逆序对数了。

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

const int N = 1e6 + 5;
int n, t, b[N], c[N], cnt;
long long ans;
pair<int, int> a[N];

// 获取 x 的最低位 1 的位置
int lowbit(int x){
    return x & -x;
}

// 在树状数组中进行单点更新
void add(int x, int y) {
    while(x <= n) {
        c[x] += y;
        x += lowbit(x);
    }
}

// 计算前缀和
int sum(int x) {
    int res = 0;
```

```
    while(x) {
        res += c[x];
        x -= lowbit(x);
    }
    return res;
}

int main(){
    cin >> n;
    for(int i = 1; i <= n; i++) {
        cin >> a[i].first;
        a[i].second = i;
    }
    sort(a + 1, a + n + 1);

    // b[i]: 原数组元素 a[i] 是原数组去重离散化后第 b[i] 大的数字
    cnt = 1;
    b[a[1].second] = cnt;
    for(int i = 2; i <= n; i++) {
        if(a[i].first != a[i - 1].first)
            cnt++;
        b[a[i].second] = cnt;
    }

    for(int i = n; i; i--) {  // 倒序扫描
        ans += sum(b[i] - 1); // 比 a[i] 小的数字的数量
        add(b[i], 1);
    }

    cout << ans;
    return 0;
}
```

归并排序求逆序对

划重点： **归并排序的交换次数就是数组的逆序对个数**。

归并排序是将数列 $a[l,r]$ 分成两半 $a[l,mid]$ 和 $a[mid+1,r]$ 分别进行归并排序，然后再将这两半合并起来。在合并的过程中（设 $l <= i <= mid$，$mid+1 <= j <= r$），当 $a[i] <= a[j]$ 时，并不产生逆序数；当 $a[i] > a[j]$ 时，在前半部分中比 $a[i]$ 大的数都比 $a[j]$ 大，将 $a[j]$ 放在 $a[i]$ 前面的话，逆序数要加上 $mid+1-i$。因此，可以在归并排序中的合并过程中计算逆序数。

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

int a[200005], g[200005], n;
long long ans;

// 归并排序的合并操作
void merge(int l, int r) {
    int mid = (l + r) >> 1;
    int pl = l, pr = mid + 1, pg = l; // 分别指向左半部分、右半部分和临时数组
    while (pl <= mid && pr <= r) {
        if (a[pl] <= a[pr])
            g[pg++] = a[pl++];
        else {
            g[pg++] = a[pr++];
            // a[pl] > a[pr]，说明 a[pl] 到 a[mid] 之间的所有元素都大于 a[pr]，则逆序对数为 mid - pl + 1
            ans += mid - pl + 1;
        }
    }
    while (pl <= mid) g[pg++] = a[pl++]; // 左边还剩元素，但右边不剩，无法构成逆序对
    while (pr <= r) g[pg++] = a[pr++];   // 右边剩，但都比左边大
    for (int i = l; i <= r; i++) a[i] = g[i]; // 将临时数组的值复制回原数组
}

// 归并排序
void merge_sort(int l, int r) {
    if (l >= r)
        return;
```

```
    int mid = (l + r) >> 1;
    merge_sort(l, mid); // 左半部分递归排序
    merge_sort(mid + 1, r); // 右半部分递归排序
    merge(l, r); // 合并两个有序数组
}

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> a[i];
    merge_sort(1, n); // 对整个数组进行归并排序
    cout << ans; // 输出逆序对数
    return 0;
}
```

广义逆序对

对于任意的$i < j$和$f(i) > f(j)$其中f代表的是一种广义的规则

小红与数组（淘天集团2024技术笔试）

```
#include <iostream>
#include <unordered_map>
#include <vector>

const int N = 200010;
long long a[N]; // 输入数组
long long b[N]; // 辅助数组
long long tree[N * 2]; // 树状数组
long long c[N]; // 记录数组
long long ans = 0; // 答案

// 解决函数
void solve(int n);
// 更新函数
void update(long long x, long long k);
// 求和函数
long long sum(long long x);
// 计算低位
long long lowbit(long long x);
```

```cpp
int main() {
    int n;
    std::cin >> n;
    solve(n);
    return 0;
}

void solve(int n) {
    std::unordered_map<long long, long long> bk; // 记录a[i]出现的次数
    std::unordered_map<long long, long long> mk; // 记录a[i]从末尾开始出现的次数

    // 输入数组a，同时统计a[i]出现的次数，记录在bk中
    for (int i = 1; i <= n; ++i) {
        std::cin >> a[i];
        bk[a[i]]++;
        b[i] = bk[a[i]]; // b[i]表示a[i]出现的次数
    }

    // 从末尾开始遍历数组a，同时统计a[i]从末尾开始出现的次数，记录在mk中
    for (int i = n; i >= 1; --i) {
        mk[a[i]]++;
        c[i] = mk[a[i]]; // c[i]表示a[i]从末尾开始出现的次数
        update(c[i], 1); // 更新树状数组
    }

    // 从头开始遍历数组a，计算答案
    for (int i = 1; i <= n; ++i) {
        update(c[i], -1); // 在树状数组中减去c[i]位置的值
        ans += sum(b[i] - 1); // 计算比b[i]-1小的值的总和，即在树状数组中查询小于b[i]-1的数的个数
    }

    std::cout << ans << std::endl; // 输出答案
}
```

```
// 计算低位
long long lowbit(long long x) {
    return x & (-x);
}

// 更新函数：从位置x开始，每次增加k
void update(long long x, long long k) {
    for (long long i = x; i <= N; i += lowbit(i))
        tree[i] += k;
}

// 求和函数：返回从位置1到位置x的所有值的和
long long sum(long long x) {
    long long ans = 0;
    for (long long i = x; i > 0; i -= lowbit(i))
        ans += tree[i];
    return ans;
}
```

```
import java.util.*;

public class Main {
    static final int N = 200010; // 最大长度
    static long[] a = new long[N]; // 输入数组
    static long[] b = new long[N]; // 辅助数组，记录每个数字出现的次数
    static long[] tree = new long[N * 2]; // 树状数组
    static long[] c = new long[N]; // 记录数组，记录从末尾开始每个数字出现的次数
    static long ans = 0; // 答案

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt(); // 输入数组的长度
        solve(scanner, n); // 调用解决函数
    }

    // 解决函数
```

```
    static void solve(Scanner scanner, int n) {
        Map<Long, Long> bk = new HashMap<>(); // 用于记录a[i]出现的次数
        Map<Long, Long> mk = new HashMap<>(); // 用于记录a[i]从末尾开始出现的次数
        for (int i = 1; i <= n; i++) {
            a[i] = scanner.nextLong(); // 输入数组a
            bk.put(a[i], bk.getOrDefault(a[i], 0L) + 1); // 记录a[i]出现的次数
            b[i] = bk.get(a[i]); // 记录数组b
        }
        for (int i = n; i >= 1; i--) {
            mk.put(a[i], mk.getOrDefault(a[i], 0L) + 1); // 记录a[i]从末尾开始出现的次数
            c[i] = mk.get(a[i]); // 记录数组c
            update(c[i], 1); // 更新树状数组
        }
        for (int i = 1; i <= n; i++) {
            update(c[i], -1); // 在树状数组中减去c[i]位置的值
            ans += sum(b[i] - 1); // 计算比b[i]-1小的值的总和，即在树状数组中查询小于b[i]-1的数的个数
        }
        System.out.println(ans); // 输出答案
    }

    // 计算最低位
    static long lowbit(long x) {
        return x & (-x);
    }

    // 更新函数：从位置x开始，每次增加k
    static void update(long x, long k) {
        for (long i = x; i <= N; i += lowbit(i))
            tree[(int) i] += k;
    }

    // 求和函数：返回从位置1到位置x的所有值的和
    static long sum(long x) {
        long ans = 0;
```

```
        for (long i = x; i > 0; i -= lowbit(i))
            ans += tree[(int) i];
        return ans;
    }
}
```