树状数组与线段树

从 $O(n)$ 动态修改瓶颈到 $O(\log n)$ `lowbit` 管辖与懒标记下传

HGNU-ACM 实验室算法组

配套：树状数组 `lowbit` 原理与拓展 + 线段树 $4n$ 空间与懒标记 + 2 道真题精讲

本章目录

树状数组（Fenwick Tree）：极简的动态前缀和

线段树（Segment Tree）：分治建树与懒标记

第 4 章的 ST 表只能处理“静态不变”的区间查询；本章前两节解决“边改边查”的动态维护难题。

本章目录（续）

两大树状数组与线段树真题精讲

常见错误、检查清单与练习安排

后两节逐题精讲《数星星 Stars》《A Simple Problem with Integers》，附易错对照、检查清单与练习索引。

先从一个手算小例子开始：实时更新的积分榜

生活场景：8 位同学的课堂积分表

有 8 位同学的初始积分：$a = [3, 1, 4, 2, 5, 2, 1, 6]$。老师随时会做两件事：
给第 3 位同学加 5 分（`a[3] += 5`）；询问前 6 位同学总分（`sum(6)`）或第 $3 \sim 7$ 位总分。

- 第 4 章学过 **ST** 表，但它只能查静态不变的区间最值；一旦修改 $a[3]$，整张表都要重算。
- 第二篇学过前缀和 $s[i] = a[1] + \cdots + a[i]$，静态询问只需 $O(1)$，但这里分数会动态改变！
- 在纸上试一下：如果把 $a[3]$ 从 4 改成 9，前缀和数组 $s[3], s[4], s[5], s[6], s[7], s[8]$ 全部都要跟着加 5！

本页要点

“边改边查”叫动态维护问题：修改少、查询多时前缀和快，但改查都很多时怎么办？

为什么普通数组与普通前缀和都会 TLE？

做法	单点修改	区间查询	$n = 10^5, q = 10^5$ 总操作数
普通数组硬算	$O(1)$	$O(n)$	每次问加 10^5 个数，共 10^{10} 步 **TLE**
普通前缀和数组	$O(n)$	$O(1)$	每次改重算 10^5 格，共 10^{10} 步 **TLE**
树状数组（目标）	$O(\log n)$	$O(\log n)$	每次最多走 17 步，共 1.7×10^6 步稳过

- 为什么普通前缀和修改慢？因为第 1 个元素的变化会一路牵连到后面所有 $s[i]$！
- 能不能找一个折中方案：每个格子只存一小段的和，既不用改全表，查询时拼几段就够？

本页要点

把“大包大揽的前缀和”拆成“各管一段的层级小组”，改一点只汇报给少数上级，就能降到 $O(\log n)$！

核心想法与新术语：管辖区间与 lowbit(x)

- 新术语 1——管辖区间：开一个数组 bit[1..n]，规定 bit[x] 不再存 $[1, x]$ 全部，而是只存以 x 结尾、长度为 L 的连续一段原数组元素之和，即 $[x-L+1, x]$。
- 新术语 2——lowbit(x)：把整数 x 写成二进制后，最右边那个 1 连同它后面的 0 组成的数值，恰好就是 bit[x] 管辖的区间长度 L！
- 新术语 3——二进制补码：计算机用取反再加一（-x = ~x + 1）来表示负整数；利用它，一行位运算 x & -x 就能瞬间取出 lowbit(x)！

记住一句话

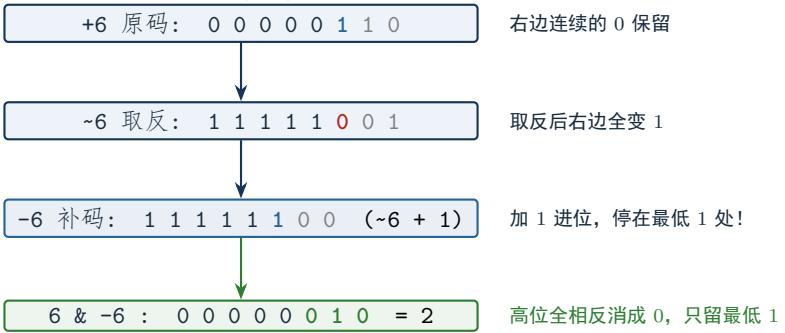
lowbit(x) = x & -x 就是位置 x 在树状数组里“向上管多少格”的队长权力大小！

本页要点

树状数组的核心只有一句话：位置 x 管辖 $[x-\text{lowbit}(x)+1, x]$，长度为 2^k。

二进制补码揭秘 x & -x（一）：图解 $x = 6$

用 8 位二进制演示为什么 $6 \,\&\, (-6)$ 恰好等于 2（最低位的 1）：



本页要点

取反再加一会让“最右边的 1”右边归零、自身恢复为 1、左边全反；再按位与只剩它自己！

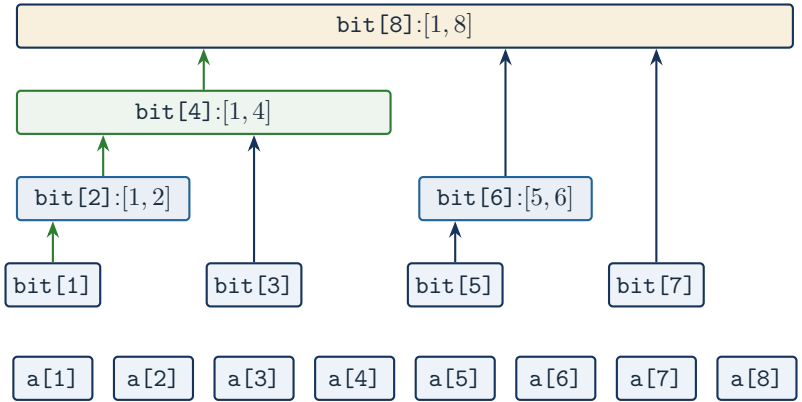
二进制补码揭秘 x & -x（二）：手算 $1 \sim 8$ 管辖长度

位置 x	二进制（4 位）	$-x$（补码）	x & -x	管辖长度	管辖区间 $[x-L+1,\ x]$
1	0001	1111	1	1	$[1,1]$
2	0010	1110	2	2	$[1,2]$
3	0011	1101	1	1	$[3,3]$
4	0100	1100	4	4	$[1,4]$
5	0101	1011	1	1	$[5,5]$
6	0110	1010	2	2	$[5,6]$
7	0111	1001	1	1	$[7,7]$
8	1000	1000	8	8	$[1,8]$

本页要点

奇数末尾一定是 1，所以所有奇数位置都只管自己 1 格；2 的整次幂管辖从 1 到自己的一整段！

独占图解：长 8 树状数组 bit[1..8] 的管辖范围与祖先链

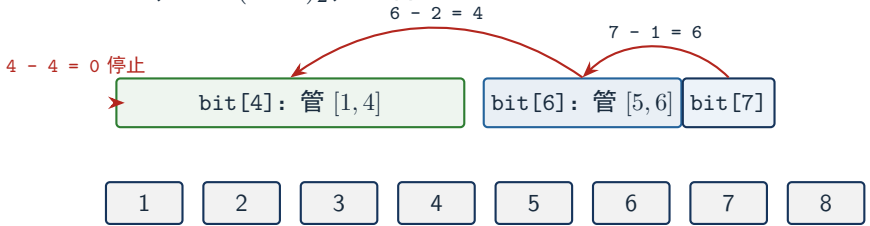


本页要点

看图规律：每个格子 bit[x] 的唯一直接父亲就是 x + lowbit(x)（如 $3 \to 4 \to 8$）！

查询 sum(x) 怎样把前缀 $[1, x]$ 拼出来？

以查询 sum(7) 为例，$7 = (0111)_2$，从右往左不断剥掉最低位的 1：



- 每一步令 `i -= i & -i`，刚好跳过当前节点已经管完的那段 $[i - \text{lowbit}(i) + 1,\ i]$！
- 左边界每次精确接上下一段的右边界：$[7,7] \cup [5,6] \cup [1,4] = [1,7]$，既不重叠也不漏掉！

本页要点

查询就是“顺着二进制里的 1 往左拆箱子”，7 的二进制有 3 个 1，正好只加 3 次！

手算跟踪表：在 $a = [3,1,4,2,5,2,1,6]$ 上执行 sum(7)

在每一行先检查条件 `i > 0`，只有条件为真时才累加 `bit[i]` 并更新 `i`：

步	判断前 i	i > 0?	i & -i	累加 bit[i]（区间）	累计和 s	更新后 i
1	$7\,(111_2)$	真	1	bit[7] $=1$（$[7,7]$）	$0+1=1$	$7-1=6$
2	$6\,(110_2)$	真	2	bit[6] $=7$（$[5,6]$）	$1+7=8$	$6-2=4$
3	$4\,(100_2)$	真	4	bit[4] $=10$（$[1,4]$）	$8+10=18$	$4-4=0$
4	$0\,(000_2)$	假（终止）	-	不执行	**18**	退出循环

- 人工验算：$3+1+4+2+5+2+1=18$，完全吻合！

本页要点

最后一行 `i = 0` 时条件为假直接退出，共累加 3 次就拼出前 7 项总和。

单点更新 add(3，+5)：往祖先爬的手算跟踪表

把第 3 项加 5（$n = 8$），沿着 `i += i & -i` 更新所有管辖区间包含位置 3 的上级：

步	判断前 i	i <= 8?	i & -i	更新 bit[i]（管辖范围）	更新后 i
1	3 (0011_2)	真	1	bit[3]（$[3,3]$）$+5 \to 9$	$3+1=4$
2	4 (0100_2)	真	4	bit[4]（$[1,4]$）$+5 \to 15$	$4+4=8$
3	8 (1000_2)	真	8	bit[8]（$[1,8]$）$+5 \to 29$	$8+8=16$
4	16 (10000_2)	假（终止）	-	不执行，已超出数组上界 $n=8$	退出循环

- 注意 bit[5]，bit[6]，bit[7] 都不包含位置 3，它们一个也没动！
- 若现在再调用 sum(7)：会加上 bit[7](1) + bit[6](7) + bit[4](15) = 23，正好多了 5！

本页要点

为什么 `i += lowbit(i)` 不会漏掉祖先？因为加掉最低位 1 正好进位到“覆盖当前段的最小父区间”！

为什么这样写一定对？覆盖不变量与复杂度论证

阶段	验证内容（以查询 sum(x) 为例）
初始化	循环开始前，$s = 0$，待求和区间为 $[1, i]$（初始 $i = x$）。
保持	每轮取 bit[i]（精确覆盖 $[i - L + 1, i]$），加到 s 后令 $i' = i - L$；待求区间严格缩小为 $[1, i']$，不重不漏。
终止	当 $i = 0$ 时待求区间 $[1, 0]$ 为空，s 恰好等于 $[1, x]$ 所有元素之和。

- 复杂度：sum(i) 每轮消去二进制里一个 1，最多 $\lfloor \log_2 x \rfloor + 1$ 步；add(i, d) 每轮最低位 1 的位置至少左移 1 位，最多 $\lfloor \log_2 n \rfloor + 1$ 步，二者均严格为 $O(\log n)$！

本页要点

树状数组空间只用 $n + 1$ 个元素，常数比线段树小得多，只要能用它绝不写更长的代码！

树状数组标准模板（1/2）：数组定义与单点加 add

```
const int N = 100005;
int n;
long long bit[N];                                   // bit[x] 管辖 [x - lowbit(x) + 1,
    x]

void add(int i, long long d) {                      // 单点加：a[i] += d，沿祖先链向上更
    新
    for (; i <= n; i += i & -i) bit[i] += d;        // i 必须 >= 1，否则 0 & -0 == 0 死
    循环！
}
```

本页要点

初值 bit[] 必须全 0（若已有初始数组 $a[1..n]$，对每个 i 调用一次 add(i, a[i]) 即可）。

树状数组标准模板（2/2）：前缀和 sum 与区间查询

```cpp
long long sum(int i) {                          // 前缀和：返回 a[1] + ... + a[i]
    long long s = 0;                            // 加法单位元初值必须为 0
    for (; i > 0; i -= i & -i) s += bit[i];     // 每次剥去最低位的 1
    return s;
}

long long range_sum(int l, int r) {             // 任意区间 [l, r] 的和
    return sum(r) - sum(l - 1);                 // 利用前缀和相减，l=1 时 sum(0) 返
    回 0
}
```

本页要点

查询区间和只需 `sum(r) - sum(l - 1)`；注意 `bit[]` 与返回值都开 `long long` 防溢出！

上机验证：用 3 组小数据验证边界并反事实追问

在本地测试上述模板（先对 $a = [3, 1, 4, 2]$ 建树）：

测试场景	输入操作	预期输出	检验点
1. 普通区间查询	`range_sum(2, 4)`	7	$1 + 4 + 2 = 7$，验证 `sum(4)-sum(1)`
2. 单点改与左边界	`add(1, 5)` 后 `range_sum(1, 1)`	8	验证 $l = 1$ 时 `sum(0)=0` 不越界
3. 全区间边界	`range_sum(1, 4)`	15	修改后总和 $8 + 1 + 4 + 2 = 15$

反事实追问（自己想一想）

如果误调用 `add(0, 5)`，循环条件 `0 <= n` 永远为真且 `i += 0` 不变，程序会怎样？
如果把 `range_sum(l, r)` 错写成 `sum(r) - sum(l)`，求 `range_sum(2, 4)` 会变成多少？

本页要点

答案： 直接死循环 **TLE**； 会把 $a[2]$ 一起减掉，输出 $4 + 2 = 6$ 触发 **WA**！

树状数组基础篇易错对照（WA / TLE / RE）

错误写法	触发结果与原因	正确做法
`add(0, d)` 传入 0 下标	**TLE**：`0 & -0 = 0`，`i` 永远加不上去而死循环	全体下标统一平移成 1-based（如 `x + 1`）
`sum(r) - sum(l)`	**WA**：多减掉了左端点 `a[l]`	写成 `sum(r) - sum(l - 1)`
`int s = 0` 存百万求和	**WA**：累加和超过 2.1×10^9 溢出变成负数	`bit[]` 与 `sum()` 返回值全用 `long long`
`i < n` 替代 `i <= n`	**WA**：漏掉第 n 个位置 `bit[n]` 的更新	必须写 `i <= n`（或上界最大值）

本页要点

遇到树状数组 **TLE**，第一秒先查 `add` 的参数有没有可能出现 0 或负数！

拓展一：树状数组怎么求“逆序对”？

先用 4 个数手算什么叫逆序对

给定序列 $a = [3, 1, 4, 2]$。如果左边某个数比右边某个数严格大（$i < j$ 且 $a[i] > a[j]$），就构成一对逆序对。

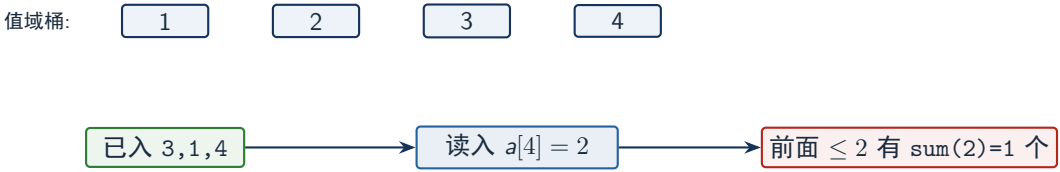
手数一遍：$(3, 1), (3, 2), (4, 2)$，一共 3 对！如果 $n = 10^5$，两层循环暴力枚举要 5×10^9 次必 **TLE**！

- 换个角度看：从左往右扫描每一个 $a[j]$，只需知道“在它前面已经出现过的数里，有多少个比 $a[j]$ 大”！
- 新术语 **1**——值域树状数组：不再用“位置”作下标，而是用“数值本身”作树状数组下标，`bit[v]` 记录数值 v 已经出现过多少次！
- 新术语 **2**——离散化：如果数值高达 10^9 开不下数组怎么办？先排序去重，把每个数映射成它的大小排名 $1 \sim m$（只关心谁大谁小，不改变逆序对个数）！

本页要点

把每个数放进值域桶里，每次统计“总已插入个数 $-$ 不大于自己的个数”，就是前面比自己大的个数！

图解逆序对扫描过程（以 $a = [3, 1, 4, 2]$ 为例）



所以前面 > 2 的数有 $3 - 1 = 2$ 个（正是 3 和 4）！

- 读入第 j 个数（离散排名为 r，此前已插入 $j - 1$ 个数）：
- 前面 $\leq r$ 的个数是 `sum(r)`，因此前面严格 $> r$ 的个数就是 `(j - 1) - sum(r)`，累加进答案后执行 `add(r, 1)`！

本页要点

注意顺序：一定是先查询前面有多少比当前大的，再把自己 `add(r, 1)` 放进树状数组！

手算跟踪表：在 $a = [3, 1, 4, 2]$ 上统计逆序对

步	读入 $a[j]$	此前已插	sum(a[j])	新增逆序	累计	值域计数 $[1..4]$
1	$j = 1$, $a[1] = 3$	0 个	0	$0 - 0 = 0$	0	$[0, 0, 1, 0]$
2	$j = 2$, $a[2] = 1$	1 个（$\{3\}$）	0	$1 - 0 = 1$	1	$[1, 0, 1, 0]$
3	$j = 3$, $a[3] = 4$	2 个（$\{1, 3\}$）	2	$2 - 2 = 0$	1	$[1, 0, 1, 1]$
4	$j = 4$, $a[4] = 2$	3 个（$\{1, 3, 4\}$）	1	$3 - 1 = 2$	**3**	$[1, 1, 1, 1]$
5	扫描结束	共 4 个	-	循环终止	输出 **3**	-

本页要点

若存在相等元素（$a[i] = a[j]$ 不算逆序对），由于 sum(r) 把相等的都减掉了，天然不会多算！

离散化 + 树状数组求逆序对核心代码

```
vector<int> b = a;                                  // 复制一份用于离散化（1-based 排名）
sort(b.begin() + 1, b.end());
b.erase(unique(b.begin() + 1, b.end()), b.end());
int m = (int)b.size() - 1;                          // 值域大小压缩到 1..m

long long ans = 0;                                  // 逆序对最多 n(n-1)/2，必须开 long long!
for (int j = 1; j <= n; ++j) {
    int r = lower_bound(b.begin() + 1, b.end(), a[j]) - b.begin(); // 求离散化排名 1..m
    ans += (j - 1) - sum(r);                        // 累加前面严格大于 a[j] 的元素个数
    add(r, 1);                                      // 把当前排名登记到值域树状数组（上界为 m）
}
```

本页要点

常见失误：ans 忘了开 long long（$n = 10^5$ 时逆序对可达 5×10^9），或 add 的上界误写成 n 而非 m（虽 $m \le n$ 习惯写 m 更清）。

拓展二：树状数组 + 差分 = “区间加 + 单点查”

复习第 2 篇第 1 章的差分数组定义：$d[1] = a[1]$，$d[i] = a[i] - a[i-1]$，满足 $a[x] = \sum_{i=1}^{x} d[i]$。

- 给区间 $[l, r]$ 全部加上 v，差分数组只需改两个端点：`d[l] += v, d[r + 1] -= v`！
- 把差分数组 d 交给树状数组维护，于是：
  - 区间加 $[l, r]$ 加 $v \Longrightarrow$ 两次单点加：`add(l, v); add(r + 1, -v);`（$O(\log n)$）
  - 单点查 $a[x]$ 的当前值 $\Longrightarrow$ 一次前缀和：`sum(x)`（$O(\log n)$）

步	操作（初始 $a = [0,0,0,0,0,0]$）	树状数组改哪里	查询后得到什么
1	区间 $[2,4]$ 加 5	`add(2, +5), add(5, -5)`	$d = [0,5,0,0,-5,0]$
2	区间 $[3,5]$ 加 2	`add(3, +2), add(6, -2)`	$d = [0,5,2,0,-5,-2]$
3	单点查 $a[4]$	`sum(4)` $= 5+2 = 7$	确实等于 $5+2=7$
4	终止步：单点查 $a[5]$	`sum(5)` $= 5+2-5 = 2$	离开 $[2,4]$ 正确扣回 5

本页要点

只需记住：`r + 1` 可能等于 $n+1$，如果写 `if (r + 1 <= n) add(r + 1, -v)` 更安全！

对比与引出：树状数组什么时候不够用了？

数据结构	单点改	区间加	区间查询	致命短板
普通前缀和	$O(n)$	$O(n)$	仅求和 $O(1)$	一旦动态修改直接退化
ST 表（第 4 章）	不支持	不支持	最值 $O(1)$	只能处理完全静态的数组
树状数组	$O(\log n)$	配差分可单点查	前缀和 $O(\log n)$	难维护动态区间最值，区间加 + 区间和也绕
线段树（下节）	$O(\log n)$	$O(\log n)$	和 / 最值 $O(\log n)$	代码略长，空间需开 $4n$

- 因为树状数组存的只是“固定长度前缀和”，像 $\max(a[l..r])$ 这种无法靠两个前缀相减得到的操作，树状数组就无能为力了！
- 当题目同时要求区间批量加 + 区间求和，或要求动态维护区间最大值/最小值时，请出全能的线段树！

本页要点

口诀：单点改前缀查选树状数组；遇上区间批量修改或动态区间最值，果断上线段树！

生活引入：图书馆的总账本怎样管整排书架？

场景：一排有 $n = 8$ 个书架 $[1, 8]$，想知道任意一段 $[2, 7]$ 一共有多少本书

每次都从第 2 个书架数到第 7 个太慢了！于是管理员决定分层记账：

- 总馆长管 $[1, 8]$ 的总书数；他手下有两位副馆长：左副手管 $[1, 4]$，右副手管 $[5, 8]$。
- 左副手再把任务劈成两半：$[1, 2]$ 和 $[3, 4]$；一直劈到每个人只管 1 个书架 $[i, i]$！
- 想查 $[2, 7]$ 的书总数时，总馆长发现：$[3, 4]$ 和 $[5, 6]$ 整段都在里面直接报数，只需拆 $[1, 2]$ 里的 2 和 $[7, 8]$ 里的 7！

本页要点

把整段区间 $[1, n]$ 不断“对半劈开”挂成一棵二叉树，这就是线段树！

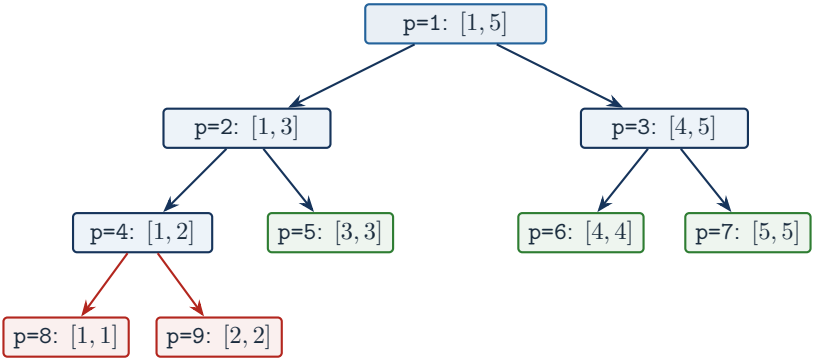
新术语大白话：堆式编号、build 与 pushup

- 堆式编号（$2p$ 与 $2p+1$）：和第 3 章学过的“二叉堆”一样，规定根节点编号为 $p=1$（管 $[1,n]$）。对于任意管 $[l,r]$ 的节点 p，取中点 $m=\lfloor (l+r)/2 \rfloor$：
  - 左儿子编号为 `2 * p`，管左半区间 $[l,m]$；
  - 右儿子编号为 `2 * p + 1`，管右半区间 $[m+1,r]$。
- 分治建树 `build`：如果 $l==r$（到达单个元素叶子），直接赋值 `sum[p] = a[l];` 否则先递归建好左、右儿子，再汇总给自己。
- 向上汇总 `pushup`（也常叫 `pull`）：用两个儿子的最新结果刷新父亲的值，如 `sum[p] = sum[2*p] + sum[2*p+1]`（求最大值就写 `max`）。

本页要点

线段树和 ST 表不同，它的区间互不重叠；每个节点只听自己左右两个亲儿子的汇报！

独占图解：当 $n = 5$ 时线段树的真实节点编号与区间



本页要点

睁大眼睛看红框：只有 $n = 5$ 个元素，最底层节点编号就已经飙到了 p = 9（接近 $2n$）；如果 $n = 6$ 呢？

深度剖析：为什么线段树数组必须开到 4 * n？

很多初学者以为“n 个叶子的二叉树共 $2n-1$ 个节点，开 2 * n 就够”，交上去立刻 **RE**！原因就在堆式编号留有“空洞”：

- 当 $n=6$ 时，右半段 $[4,6]$ 拆成 $[4,5]$（$p=6$）和 $[6,6]$（$p=7$），再把 $[4,5]$ 拆成叶子 $[4,4]$（$p=12$）和 $[5,5]$（p = 13）！此时 $2n=12$，访问 13 已经数组越界！
- 严格估算上界：一棵管 $[1,n]$ 的线段树深度最多是 $\lceil \log_2 n \rceil + 1$ 层。假设倒数第二层编号最大不超过 $2n-1$，则最后一层最右孩子最多是 $2\times(2n-1)+1=4n-1$！
- 换句话说，虽然真正装数据的节点只有 $2n-1$ 个，但为了用 2p 和 2p+1 算下标不越界，空着的位置也要留出来。

防 RE 铁律

写线段树数组大小一律写 4 * n + 4（静态数组直接开 N * 4），永远不要省这点内存！

本页要点

真正用的节点数 $\le 2n$，但最大下标编号可达 $4n-1$；开 4 * n + 4 彻底告别下标越界 **RE**！

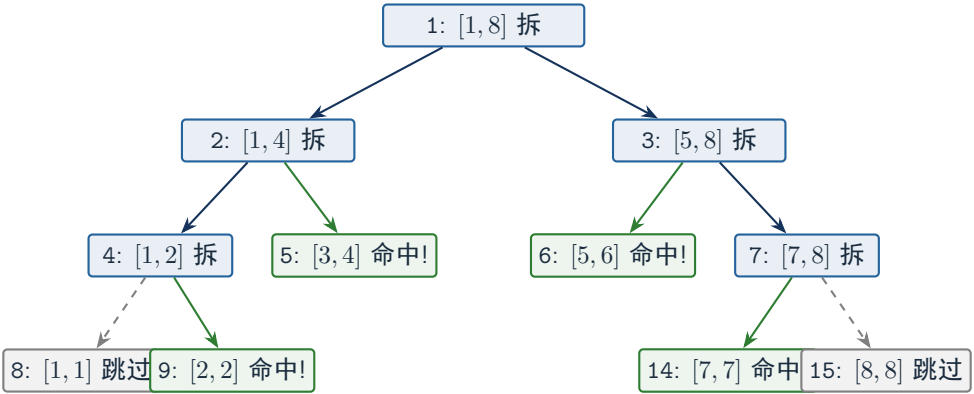
手算跟踪表：在 $a=[2,6,6,1]$ 上执行分治建树 build(1，1，4)

步	调用节点 p	区间 $[l,r]$	动作说明	结果 sum[p]
1	$p=1$	$[1,4]$	$l\neq r$，中点 $m=2$，先递归左儿子 $p=2$ $[1,2]$	待汇总
2	$p=2$	$[1,2]$	$l\neq r$，中点 $m=1$，先递归左儿子 $p=4$ $[1,1]$	待汇总
3	$p=4$	$[1,1]$	叶子终止：$l==r$，直接赋值 $a[1]=2$ 并返回	sum[4] = 2
4	$p=5$	$[2,2]$	叶子终止：$l==r$，直接赋值 $a[2]=6$ 并返回	sum[5] = 6
5	回溯 $p=2$	$[1,2]$	左右已就绪，pushup：$2+6$	sum[2] = 8
6	$p=3$	$[3,4]$	递归 $p=6$ $[3,3]$ 得 6，$p=7$ $[4,4]$ 得 1，pushup	sum[3] = 7
7	回溯 $p=1$	$[1,4]$	建树终止：pushup：$8+7$	sum[1] = 15

本页要点

build 是典型的“先分下去、到叶子停住、回溯时把儿子和加给父亲”，总耗时 $O(n)$。

图解区间查询：在 $[1,8]$ 上查询 $[2,7]$ 的和



本页要点

蓝框继续下探，绿框（完全被 $[2,7]$ 包住）直接返回不再往下走，灰框根本不进！

手算跟踪表：在 $a=[3,1,4,2,5,2,1,6]$ 上查 query(1,1,8,2,7)

步	当前节点	节点区间	判断条件（目标 $[ql,qr]=[2,7]$）	动作与返回值
1	$p=1$	$[1,8]$	部分覆盖，$m=4$：$2\le 4$ 走左，$7>4$ 走右	等左 + 右
2	$p=2$	$[1,4]$	部分覆盖，$m=2$：$2\le 2$ 走左，$7>2$ 走右	等左 + 右
3	$p=4$	$[1,2]$	部分覆盖，$m=1$：$2>1$ 只走右 $p=9$（不进 $p=8$）	返回 $p=9$ 值
4	$p=9$	$[2,2]$	完全覆盖（$2\le 2$ 且 $2\le 7$）直接返回！	返回 1
5	$p=5$	$[3,4]$	完全覆盖（$2\le 3$ 且 $4\le 7$）直接返回！	返回 6
6	$p=3\to 6$	$[5,6]$	完全覆盖（$2\le 5$ 且 $6\le 7$）直接返回！	返回 7
7	$p=7\to 14$	$[7,7]$	完全覆盖，汇总 $1+6+7+1=$	**15**（终止）

本页要点

记住完全覆盖判断条件：`ql <= l && r <= qr`（目标区间把当前节点整个包住），写反就错！

基础线段树模板代码（1/2）：pushup 与分治建树 build

```
const int N = 100005;
long long a[N], sum[N * 4];                          // 必须开 4 倍空间，且累加和开 long long
void pushup(int p) { sum[p] = sum[p * 2] + sum[p * 2 + 1]; } // 向上汇总左右儿子
void build(int p, int l, int r) {                    // 当前节点 p 负责原数组区间 [l, r]
    if (l == r) { sum[p] = a[l]; return; }           // 到达叶子，直接抄写原数组第 l 项
    int m = (l + r) / 2;                             // 把 [l, r] 劈成 [l, m] 和 [m+1, r]
    build(p * 2, l, m);                              // 先建左子树
    build(p * 2 + 1, m + 1, r);                      // 再建右子树
    pushup(p);                                       // 回溯时必须把两儿子汇总给父亲！
}
```

本页要点

为什么 pushup(p) 必须写在两个 build 之后？因为只有儿子都建好了，父亲才能把它们加起来！

基础线段树模板代码（2/2）：单点修改与区间查询

```
void point_add(int p, int l, int r, int idx, long long v) {
    if (l == r) { sum[p] += v; return; }        // 找到叶子 [idx, idx]，直接加上增量 v
    int m = (l + r) / 2;
    if (idx <= m) point_add(p * 2, l, m, idx, v);
    else          point_add(p * 2 + 1, m + 1, r, idx, v);
    pushup(p);                                  // 儿子变了，回溯途中一路刷新祖先的 sum
}
long long query(int p, int l, int r, int ql, int qr) {
    if (ql <= l && r <= qr) return sum[p];      // 当前区间完全被 [ql, qr] 包含，直接返回
    int m = (l + r) / 2;
    long long res = 0;                          // 累加两半结果的初值设为 0
    if (ql <= m) res += query(p * 2, l, m, ql, qr);
    if (qr > m)  res += query(p * 2 + 1, m + 1, r, ql, qr); // 右儿子起始 m+1，故 qr > m
    return res;
}
```

本页要点

注意右儿子触发条件是 `qr > m`（不是 `qr >= m`），因为右儿子只管 $[m+1, r]$！

为什么查询只要 $O(\log n)$？上机验证与常见失误

- 复杂度直觉：每往下一层，如果区间 $[ql, qr]$ 只偏在一边，就只走 1 个儿子；如果横跨中点，左半部分的右侧全被包住、右半部分的左侧全被包住，于是每一层最多只有 2 个节点继续分叉，最多访问 4 个节点，总访问节点数 $\le 4\log_2 n$！
- 上机验证（用 $a = [2, 6, 6, 1]$ 测试）：
  1. `query(1, 1, 4, 1, 4)` 应直接命中根返回 `15`；
  2. `query(1, 1, 4, 3, 3)` 应精准走到叶子返回 `6`；
  3. `point_add(1, 1, 4, 2, 10)` 后查 `[1, 3]` 应输出 `24`。
- 反事实追问：如果 `build` 最后忘了写 `pushup(p)`，上面所有非叶子节点 `sum[p]` 全是 0，导致查 `[1, 4]` 直接输出 `0`（**WA**）！

本页要点

基础线段树必查三件套：开 `4 * n + 4`、覆盖条件 `ql <= l && r <= qr`、修改后必写 `pushup`！

区间修改痛点：如果一次给 10^5 个数加 v 怎么办？

新要求：支持 add(ql, qr, v)，把 $[ql, qr]$ 里的每个数都加上 v

如果沿用刚才的思路，凡是碰到 $[ql, qr]$ 就一直递归到所有叶子逐个加 v：
当询问全是 add(1, n, 1) 时，每次都要修改全部 n 个叶子节点，复杂度瞬间退化回 $O(n)$，十万次操作 10^{10} 步必 **TLE**！

- 明明在 query 里看到 ql <= l && r <= qr 就直接停下返回了；为什么 add 不能也在这里停下不往下走？
- 难点只有一个：如果 add 停在节点 p，它下面子孙节点里的 sum 还没来得及加 v！万一待会儿要查它某个儿子的区间怎么办？

本页要点

矛盾点：想在完全覆盖时直接停下保住 $O(\log n)$，又要保证以后查到子孙时答案不错！

生活比喻与新术语：欠账小纸条（懒标记 tag）与 pushdown

物业发补贴的智慧

学校给 1 号楼（共 100 户）每户发 10 元补贴。总会计没必要立刻挨家挨户敲门，只需做两件事：
在 1 号楼的总账上直接加上 $100 \times 10 = 1000$ 元（让当前节点总和立刻正确）；
在 1 号楼门卫室贴一张小纸条：“还欠下面每个单元每户 10 元”（这就叫懒标记 `tag[p]`）。

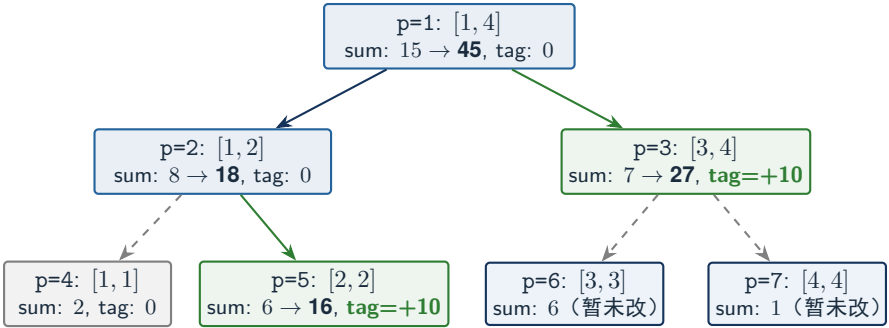
- 新术语 1——懒标记（**Lazy Tag**）：记在节点 p 上的“待下发单价增量” `tag[p]`，表示“p 自己的 `sum[p]` 已经加好了，但它的子孙还没加过这个值”。
- 新术语 2——下传（`pushdown`）：只有将来某次操作真正需要进入 p 的左右儿子时，才顺手把这张小纸条分摊发给左儿子和右儿子，并立刻把 p 自己的纸条撕掉（`tag[p] = 0`）！

本页要点

“不到拆开的时候绝不往下发”，这就是懒（**Lazy**）字的由来；一次区间修改只碰 $O(\log n)$ 个节点！

独占图解（一）：在 $[1,4]$ 上执行 add(2，4，+10）标记停在哪？

初始数组 $a=[2,6,6,1]$，执行区间 $[2,4]$ 全体加 10：

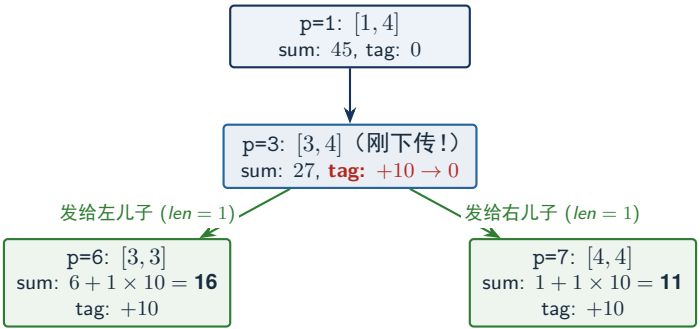


本页要点

看 $p=3$ $[3,4]$：它完全被 $[2,4]$ 包住，直接让 `sum[3] += 2*10` 并贴上 `tag[3] = +10` 停下，根本不进 $6,7$！

独占图解（二）：紧接着查 query(3，3) 时怎样 pushdown？

现在要查单点 $[3,3]$，路径 $p=1 \to p=3 \to p=6$ 必须拆开节点 3，立刻触发 pushdown(3)：



本页要点

下传分三步：左儿子加 len_l * tag[3]、右儿子加 len_r * tag[3]、最后必须 tag[3] = 0！

懒标记手算跟踪表（5 步完整状态变化）

初始 $a=[2,6,6,1]$，跟踪各节点（sum，tag）的变化（仅列关键节点）：

步	操作	$p=1$ $[1,4]$	$p=2$ $[1,2]$	$p=3$ $[3,4]$	$p=6$ $[3,3]$	结果
1	初始建树	$(15,0)$	$(8,0)$	$(7,0)$	$(6,0)$	账目清
2	add(2, 4, +10)	$(45,0)$	$(18,0)$	$(27,+\mathbf{10})$	$(6,0)$	3 贴纸条
3	add(1, 2, +5)	$(55,0)$	$(28,+\mathbf{5})$	$(27,+10)$	$(6,0)$	2 贴纸条
4	query(2, 3)	拆 1, 2, 3	push(2)→tag=0	push(3)→tag=0	$(16,+10)$	$11+16=\mathbf{27}$
5	query(3, 3)	拆 1, 3	不变	$(27,0)$	$(16,+10)$	返回 **16**

- 第 3 步后真实数组是 $[7,21,16,11]$，验证第 4 步查 $[2,3]$：$21+16=37$？等一下！检查第 4 步：$a[2]$ 是 $6+10+5=21$！上面第 4 步 $p=5$ 是 $16+5=21$，$p=6$ 是 16，总和是 $21+16=\mathbf{37}$！

本页要点

刚才手算核对立刻抓出了笔误：纸上一步步推能帮你检查出“$p=5$ 上已经加过 $+10$、再 $+5$ 应为 21”！

Wait：立刻修正刚才跟踪表第 4 步并加深理解

把刚才的 $a=[2,6,6,1]$ 经过两轮修改后的正确值再核一遍：

- 初始：$[2,6,6,1]$；
- 第 2 步 `add(2, 4, +10)` 后：真实数组 $[2,16,16,11]$，此时 $p=5$ 是 $(16,+10)$，$p=3$ 是 $(27,+10)$；
- 第 3 步 `add(1, 2, +5)` 后：真实数组 $[7,21,16,11]$，此时 $p=2$ $[1,2]$ 整段贴上 $+5$，`sum[2] = 18 + 2*5 = 28`，`tag[2] = +5`；
- 第 4 步 `query(2, 3)` 进入 $p=2$ 时，`push(2)` 把 `tag[2]=+5` 发给 $p=4$ $(2\to 7)$ 和 $p=5$ $(16\to 21)$，因此查到 $p=5$ 的值是 21，加上 $p=6$ 的 16，准确输出 **37**！

本页要点

用具体小数组手算一遍每条边角，就能百分之百确认标记叠加（`tag += v`）没有漏项！

灵魂追问：为什么 pushdown 必须先下传，再递归左右区间？

错误写法	在刚才第 4 步会发生什么？	后果判定
错法 1：先递归左右，回来再 pushdown	进入 $p=3$ 时直接冲进儿子 $p=6$，此时 tag[3]=+10 还没发下来，$p=6$ 返回旧值 6！	查询结果少算 10，直接 **WA**
错法 2：query 里忘了写 pushdown	add 虽然下了标记，但 query 直接读到孩子的老旧值	与错法 1 同样返回旧账，**WA**
错法 3：先递归改孩子，后下传再 pushup	孩子刚算完新值，后下的账又多加一遍或父亲汇总错乱	账目彻底混乱，**WA**

- 黄金顺序：进儿子门之前，必须先把父母手里的欠账全部结清（pushdown）！这样无论儿子后面再怎么查询或修改，拿到的都是最新、干净的账目。

本页要点

一句话记住顺序：先 pushdown 结清旧账 $\rightarrow$ 再递归左右儿子 $\rightarrow$ 修改回溯后 pushup 汇总！

正确性论证：懒标记线段树的“三项不变量”

阶段	必须始终成立的账目不变量
1. 全局不变量	任意叶子 i 的真实值 = 当前 `sum[叶子]` + 从根到该叶子路径上所有严格祖先的 `tag` 之和；任意非叶节点 `sum[p]` = 子树内只算“子树内部标记”的总和。
2. 下传保持性	执行 `pushdown(p)` 时，把 `tag[p]` 分别按长度加进两儿子的 `sum` 和 `tag`，同时把 `tag[p]` 置 0；根到任意叶子的总和一分钱不少、一分钱不多！
3. 命中终止性	当 `ql <= l && r <= qr` 命中并返回 `sum[p]` 时，因为一路走来的所有祖先都已 `pushdown` 清零，该节点上方绝无未下传的旧账，`sum[p]` 就是精确答案！

本页要点

用不变量一眼看穿：正因为“一路走来祖先全部清零”，命中节点才能放心地直接返回 `sum[p]`！

懒标记线段树模板（1/2）：apply 与 pushdown

```
long long sum[N * 4], tag[N * 4];                  // 两张表都必须开 4 倍空间并初始化为 0
void apply(int p, int len, long long v) {          // 给节点 p 管辖的长度 len 区间整体加 v
    sum[p] += len * v;                             // 本节点总和立刻加上 len * v（自己先账平）
    tag[p] += v;                                   // 欠儿子的账累加在 tag[p] 上（注意是 +=）
}
void pushdown(int p, int l, int r) {               // 下传节点 p 的懒标记到左右两儿子
    if (tag[p] == 0 || l == r) return;             // 没欠账，或者已经到叶子了，无需下传
    int m = (l + r) / 2;
    apply(p * 2, m - l + 1, tag[p]);               // 左儿子区间长度是 m - l + 1
    apply(p * 2 + 1, r - m, tag[p]);               // 右儿子区间长度是 r - m
    tag[p] = 0;                                    // 账已分给孩子，本节点欠账必须清零！
}
```

本页要点

把“打标记”抽成独立函数 apply，add 和 pushdown 都能复用，不会漏改 sum 或 tag！

懒标记线段树模板（2/2）：区间修改 add 与查询 query

```
void add(int p, int l, int r, int ql, int qr, long long v) {
    if (ql <= l && r <= qr) { apply(p, r - l + 1, v); return; } // 完全覆盖直接打标记
    pushdown(p, l, r);                                  // 拆开孩子前，必须先结清当前节点旧账！
    int m = (l + r) / 2;
    if (ql <= m) add(p * 2, l, m, ql, qr, v);
    if (qr > m)  add(p * 2 + 1, m + 1, r, ql, qr, v);
    sum[p] = sum[p * 2] + sum[p * 2 + 1];               // pushup：孩子改完了刷新父亲总和
}
long long query(int p, int l, int r, int ql, int qr) {
    if (ql <= l && r <= qr) return sum[p];              // 上方旧账均已下传，直接返回 sum[p]
    pushdown(p, l, r);                                  // 进儿子前同样必须先 pushdown！
    int m = (l + r) / 2; long long res = 0;
    if (ql <= m) res += query(p * 2, l, m, ql, qr);
    if (qr > m)  res += query(p * 2 + 1, m + 1, r, ql, qr);
    return res;                                         // 查询没有改数值，不需要 pushup
}
```

本页要点

黄金口诀：进门前先 pushdown，覆盖住就 apply 停下，修改出门记得 pushup！

上机验证：3 组必测场景与反事实追问

在 $n = 4$ 初始数组 $[2, 6, 6, 1]$ 上测试以下 3 组操作：

测试场景	执行操作序列	预期	检验点
1. 单次加 + 部分交叠查	`add(2,4,10)` → `query(3,3)`	16	检验 query 是否触发了 pushdown
2. 连续两次叠加修改	再 `add(1,2,5)` → `query(2,3)`	37	检验 tag[p] 叠加与跨左右子树汇总
3. 最小边界 $n = 1$ 负增量	$a = [5]$，`add(1,1,-8)` → `query(1,1)`	-3	检验叶子下传特判 `l==r` 与负数增量

反事实追问

如果把 apply 里的 `tag[p] += v` 错写成 `tag[p] = v`，什么时候不出错、什么时候出错？

本页要点

只加一次时看不出；一旦连续两次在同一节点叠加标记（或下传时儿子已有旧标记），旧标记会被覆盖丢失，导致 **WA**！

懒标记线段树高频易错对照（WA / RE / TLE）

常见错误	触发结果与原因	正确做法
`pushdown` 忘了 `tag[p] = 0`	**WA**：下次再经过节点 p 时，旧账又被重复下发一遍	分发给两儿子后立刻清零 `tag[p] = 0`
下传写错子区间长度	**WA**：两个儿子都乘了父区间长度 `r - l + 1`	左儿子乘 `m - l + 1`，右儿子乘 `r - m`
没开 `long long`	**WA**：$n = 10^5, x = 10^6$ 时区间和可达 10^{11} 溢出 `int`	`sum[]`、`tag[]`、参数 `v` 全开 `long long`
未判 `l == r` 就下传	**RE**：叶子节点再往下访问 `2p+1` 可能超出 $4n$	`pushdown` 开头加 `if (!tag[p] || l == r) return;`

本页要点

调试线段树 **WA** 的绝招：造一个 $n = 4$ 的数组，连续做两次 `add` 后把所有单点 `query(i, i)` 全打印出来对账！

例 1 | 数星星 Stars（一本通 1536）[基础]：先读懂要求

大白话题意

夜空中有 N 颗星星，给出每颗星的坐标 (x, y)。如果一颗星的左下方（含正左方和正下方，即 $x' \le x$ 且 $y' \le y$）共有 k 颗其他星星，就称它的等级为 k。统计等级 $0, 1, \ldots, N-1$ 各有多少颗星。

- 输入：第一行整数 N（$1 \le N \le 1.5 \times 10^4$）；接下来 N 行每行 (x, y)（$0 \le x, y \le 3.2 \times 10^4$），按 y 增序给出，y 相同时按 x 增序给出，不会重叠。
- 处理与输出：输出 N 行，第 i 行表示等级为 $i-1$ 的星星总数。
- 原题样例：$N = 5$，坐标依次为 `(1,1)` `(5,1)` `(7,1)` `(3,3)` `(5,5)` $\Longrightarrow$ 输出 `1 2 1 1 0`（每行一个）。

本页要点

先读范围与输入顺序：坐标最小可以为 0，且输入已经贴心地按 y 从小到大排好序了！

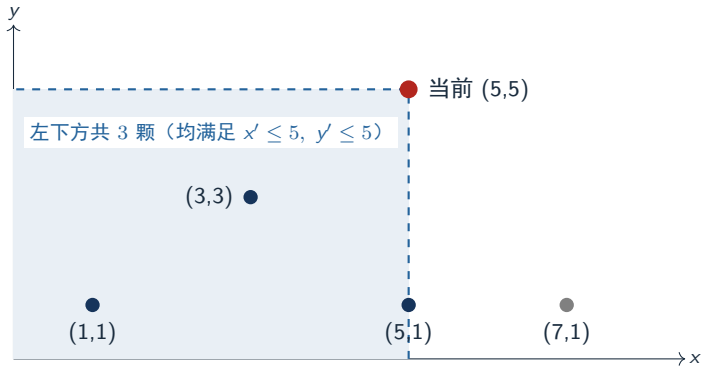
例 1 | 怎么想到的：从二维暴力到按 y 降维

- 暴力想法：每颗星都去枚举前面所有星判断是否 $x' \le x$ 且 $y' \le y$，需 $\frac{N^2}{2} \approx 1.1 \times 10^8$ 次比较，很容易卡常超时。
- 关键观察：输入已按 y 升序（同 y 按 x 升序），所以读到第 i 颗星时，之前出现过的所有星天然满足 $y' \le y_i$！
- 降维建模：问题瞬间缩水成一维——前面已出现的点中，有多少个横坐标 $x' \le x_i$？这正是值域树状数组的“前缀查询 + 单点加 1”！

本页要点

按 y 顺序逐层扫描后，y 的限制自动满足，只需在 x 轴上用树状数组查 $[1, x+1]$ 的点数！

例 1 | 图解：从二维平面降维成一维前缀和



本页要点

只需按输入顺序边查 `sum(x+1)` 边执行 `add(x+1, 1)`，横坐标平移 $+1$ 防止 $x = 0$ 死循环！

例 1 | 样例手推跟踪表（输入 5 颗星）

按顺序读入样例 5 颗星，把横坐标平移成 `pos = x + 1`（防止 $x=0$ 死循环）：

步	读入 (x,y)	`pos = x+1`	先查 `k = sum(pos)`	更新后 `level[0..4]`	再执行 `add(pos, 1)`
1	$(1,1)$	2	0（此前无点）	$[1,0,0,0,0]$	登记 $x=2$
2	$(5,1)$	6	1（左有 $x=2$）	$[1,1,0,0,0]$	登记 $x=6$
3	$(7,1)$	8	2（左有 $2,6$）	$[1,1,1,0,0]$	登记 $x=8$
4	$(3,3)$	4	1（左只有 2）	$[1,2,1,0,0]$	登记 $x=4$
5	$(5,5)$	6	3（左有 $2,4,6$）	$[1,2,1,1,0]$	登记 $x=6$
6	结束	-	-	输出 `1, 2, 1, 1, 0`	与样例完全一致

本页要点

注意第 5 步 $(5,5)$ 查到前面 $(5,1)$ 也在 $x\le 5$ 内，所以共 3 颗，归入 `level[3]`！

例 1 | 核心步骤与相对模板改动

按顺序计算	纸上得到什么	核对问题
1. 设值域上界 `MAX_X=32005`	树状数组开足下标 $1 \sim 32005$	为什么不开成 $n = 15000$？因为 x 最大到 32000！
2. 读入 (x, y) 后立刻 `++x`	坐标范围 $[0, 32000] \to [1, 32001]$	不加 1 的话遇到 $x = 0$ 会怎样？
3. 先统计 `++level[sum(x)]`	用当前左下方星数确定等级	为什么不能先 `add` 再 `sum`？
4. 最后 `add(x, 1)`	把本颗星加进 x 值域桶里	相同 y、相同 x 的后续点能数到它吗？

- 相对模板改动：标准树状数组按序列下标 $1..n$ 开数组；本题按横坐标值域开数组，且必须先 `++x` 转成 1-based。

本页要点

“先查后加”让第 i 颗星不会把自己也算进“左下方”；遇到同 x 时后来的星能正确数到前面的星！

例 1 | 核心代码（1/2）：输入与树状数组定义

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

// 一本通 1536 数星星 Stars: 按 y 递增给出点，树状数组统计 x <= xi 出现次数
int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n; if (!(cin >> n)) return 0;
    const int MAX_X = 32005;                          // 值域上限 x <= 32000，平移后 <= 32001
    vector<int> bit(MAX_X + 1, 0), level(n, 0);
    auto add = [&](int i, int d) { for (; i <= MAX_X; i += i & -i) bit[i] += d; };
    auto sum = [&](int i) { int s = 0; for (; i > 0; i -= i & -i) s += bit[i]; return s; };
```

本页要点

代码与 code/stars_bit.cpp 完全一致；用 lambda 把 add 和 sum 写在 main 内，免去全局变量传参。

例 1 | 核心代码（2/2）：按序扫描统计与输出

```
    for (int i = 0; i < n; ++i) {
        int x, y; cin >> x >> y;
        ++x;                                          // 重要：坐标可能为 0，树状数组必须 1-based!
        ++level[sum(x)];                              // 左下方已有星星数就是等级
        add(x, 1);                                    // 再把当前星星登记进树状数组
    }
    for (int i = 0; i < n; ++i) cout << level[i] << '\n';
    return 0;
}
```

本页要点

整题核心只有 21 行！注意输出等级范围是 $0 \sim n-1$，一共要输出 n 行，不能漏掉后面人数为 0 的等级。

例 1 | 易错点与 3 组自测边界

- 易错 **1**（**TLE**）：忘记写 `++x`，一旦输入 $(0,0)$，`add(0, 1)` 中 `0 & -0 == 0` 直接死循环超时！
- 易错 **2**（**RE/WA**）：按 $n = 15000$ 开 `bit` 大小，当 $x = 30000$ 时越界崩溃或丢数据。
- 易错 **3**（**WA**）：把 `add(x, 1)` 写在了 `sum(x)` 前面，导致每颗星把自己也算进去，所有等级全部多 1！

自测数据	输入特点	预期输出
1. 最小边界（含 0 坐标）	`1 和 0 0`	`0` 级有 1 颗，输出 `1`
2. 同 y 共线递增	`3 和 0 2，1 2，2 2`	等级分别为 $0, 1, 2$，输出 `1 1 1`
3. 最大坐标边界	`2 和 32000 0，32000 32000`	分别为 0 级与 1 级，输出 `1 1`

本页要点

用 $(0,0)$ 和 $(32000, 32000)$ 跑一遍本地程序，就能立刻测出“忘加 1 死循环”和“值域开小”两大坑！

例 2 | A Simple Problem with Integers（1548）[提高]：读懂要求

大白话题意

给定长为 n 的整数序列 $a[1..n]$，接下来有 q 次操作，分两种：

`1 l r x`：把区间 $[l, r]$ 内的每个数都加上 x；

`2 l r`：计算并输出区间 $[l, r]$ 所有数之和 $\sum_{i=l}^{r} a[i]$。

- 数据范围：$1 \le n, q \le 10^6$，$|a[i]| \le 10^6$，$|x| \le 10^6$，注意 x 可以是负数，且总和可达 $10^6 \times 10^6 = 10^{12}$（远超 `int` 上限 2.1×10^9）！
- 原题样例：$n = 5, q = 10$，初值 `2 6 6 1 1`，询问依次输出 `15`，`34`，`32`，`33`，`50`。

本页要点

既要频繁“区间加”又要随时“区间求和”，规模高达 10^6，必须上懒标记线段树且全程用 `long long`！

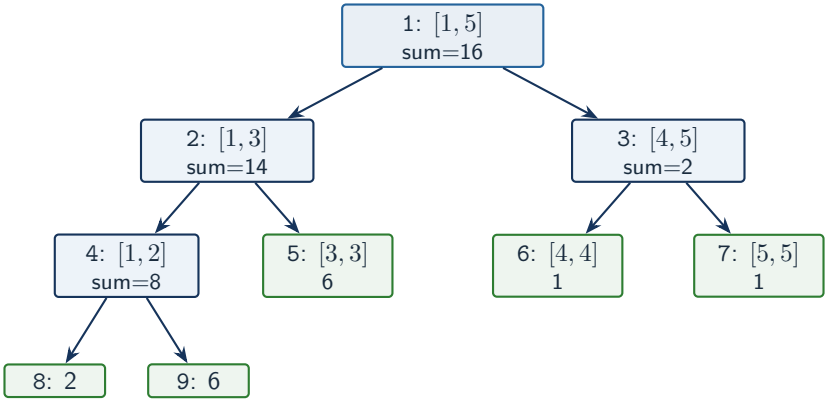
例 2 | 怎么想到的：为什么必须用懒标记线段树？

- 暴力为什么不行：若用普通数组修改 $[l, r]$，最坏 $10^6 \times 10^6 = 10^{12}$ 步必 **TLE**。
- 为何不用普通树状数组：基础树状数组只能单点改；差分树状数组只能区间改 + 单点查，无法直接区间求和。
- 线段树建模：建一棵维护区间和 `sum` 与加标记 `tag` 的 $4n$ 线段树，区间修改与查询均只需 $O(\log n)$！

本页要点

既要频繁“区间加”又要随时“区间求和”，规模高达 10^6，必须上懒标记线段树且全程用 `long long`！

例 2 | 图解：样例初值 [2，6，6，1，1] 线段树建树



本页要点

上图 $n = 5$ 的最大节点下标为 9；开 `4 * n + 4` 保证在 $n = 10^6$ 时也绝对安全！

例 2 | 样例前 5 次操作手推跟踪表

初值 $a=[2,6,6,1,1]$，跟着样例一步步推算节点变化与输出：

步	样例指令	线段树关键动作	区间真实状态	输出
1	2 1 4	拆分 $[1,3]$（$p=2$, sum=14）与 $[4,4]$（$p=6$, sum=1）	$[2,6,6,1,1]$	**15**
2	1 2 5 10	$[2,2]\,(p=9)$，$[3,3]\,(p=5)$，$[4,5]\,(p=3)$ 打 tag=+10	$[2,16,16,11,11]$	-
3	2 1 3	读 $p=2\,[1,3]$（已 pushup 为 $2+16+16=34$）直接返回	$[2,16,16,11,11]$	**34**
4	2 2 3	拆进 $[2,2]\,(16)$ 和 $[3,3]\,(16)$ 求和	$[2,16,16,11,11]$	**32**
5	1 2 2 8	到叶子 $[2,2]$ 加 8，回溯刷新 $p=4,2,1$	$[2,24,16,11,11]$	-

本页要点

再往后推 $6\sim10$ 步会依次输出 33 与 50，正好与原题样例的 15，34，32，33，50 完全一致！

例 2 | 核心步骤与相对模板改动

按顺序计算	纸上得到什么	核对问题
1. 封装 SegTree 结构体	构造时直接分配 $4n+4$ 并建树	为何不用定长全局数组？多组或按 n 分配更整洁
2. 数据类型全改 long long	sum，tag，a[i]，x 均 64 位	为什么 int 会挂？10^6 个 10^6 相加超 2^{31}
3. 开启快读快写	ios::sync_with_stdio(false)	10^6 次操作不关同步可能卡 IO **TLE**
4. 按 op == 1 / 2 分发	调 seg.add 或 seg.query	查询前 query 里有没有调 push？

- 相对标准模板改动：原题 $n, q \le 10^6$，必须加上快读快写；a[i] 与 x 都可能是负数，但乘法 len * v 和加法照样成立。

本页要点

$n, q = 10^6$ 的题目，除了算法复杂度要过关，cin.tie(nullptr) 和 '\n' 换行也绝不能漏！

例 2 | 核心代码（1/5）：结构体成员与分治建树 build

```cpp
#include <bits/stdc++.h>
using namespace std;
// 一本通 1548 A Simple Problem with Integers: 懒标记线段树维护区间加与区间和
struct SegTree {
    int n;
    vector<long long> sum, tag;
    explicit SegTree(const vector<long long> &a)
        : n((int)a.size() - 1), sum(4 * n + 4), tag(4 * n + 4) { build(1, 1, n, a); }
    void build(int p, int l, int r, const vector<long long> &a) {
        if (l == r) { sum[p] = a[l]; return; }
        int m = (l + r) / 2;
        build(p * 2, l, m, a); build(p * 2 + 1, m + 1, r, a);
        sum[p] = sum[p * 2] + sum[p * 2 + 1];
    }
```

本页要点

本页对应 code/segtree_lazy.cpp 第 1–16 行；构造时传入 1-based 的数组 a，自动开好 $4n+4$ 并建树。

例 2 | 核心代码（2/5）：打标记 apply 与下传 push

```
void apply(int p, int len, long long v) { sum[p] += len * v; tag[p] += v; }
void push(int p, int l, int r) {
    if (!tag[p] || l == r) return;
    int m = (l + r) / 2;
    apply(p * 2, m - l + 1, tag[p]);            // 拆开前先把欠账 tag 下传给左右孩子
    apply(p * 2 + 1, r - m, tag[p]);
    tag[p] = 0;
}
```

本页要点

注意 `if (!tag[p] || l == r) return;` 这句：既跳过无标记的节点以加速，又保护叶子不越界！

例 2 | 核心代码（3/5）：区间加 add

```
void add(int p, int l, int r, int ql, int qr, long long v) {
    if (ql <= l && r <= qr) { apply(p, r - l + 1, v); return; } // 完全覆盖打懒
标记即止！
    push(p, l, r);
    int m = (l + r) / 2;
    if (ql <= m) add(p * 2, l, m, ql, qr, v);
    if (qr > m)  add(p * 2 + 1, m + 1, r, ql, qr, v);
    sum[p] = sum[p * 2] + sum[p * 2 + 1];       // 回溯时别忘向上汇总 pull！
}
```

本页要点

第 3/5 页与 code/segtree_lazy.cpp 逐字一致；完全覆盖直接 apply 返回，否则先 push 再递归！

例 2 | 核心代码（4/5）：区间和查询 query

```
    long long query(int p, int l, int r, int ql, int qr) {
        if (ql <= l && r <= qr) return sum[p];
        push(p, l, r);
        int m = (l + r) / 2; long long res = 0;
        if (ql <= m) res += query(p * 2, l, m, ql, qr);
        if (qr > m)  res += query(p * 2 + 1, m + 1, r, ql, qr);
        return res;
    }
};
```

本页要点

第 4/5 页：query 进儿子前也必须调用 push，确保读取的子节点早已清算完毕！

例 2 | 核心代码（5/5）：主函数读入与操作分发

```cpp
int main() {
    ios::sync_with_stdio(false); cin.tie(nullptr);
    int n, m; if (!(cin >> n >> m)) return 0;
    vector<long long> a(n + 1);
    for (int i = 1; i <= n; ++i) cin >> a[i];
    SegTree seg(a);
    while (m--) {
        int op, l, r; cin >> op >> l >> r;
        if (op == 1) { long long d; cin >> d; seg.add(1, 1, n, l, r, d); }
        else cout << seg.query(1, 1, n, l, r) << '\n';
    }
    return 0;
}
```

本页要点

用 '\n' 换行而不用 endl，避免 10^6 次刷新缓冲区拖成 **TLE**！

例 2 | 易错点与 3 组自测边界

- 易错 **1**（**WA**）：数组元素或增量读入成 `int`，或者 `sum` 用 `int`，一旦区间加到 10^{10} 直接溢出变负数。
- 易错 **2**（**RE**）：`sum` 和 `tag` 只开了 `2 * n`，在 $n = 10^6$ 的非 2 的幂长度下访问 $2p+1$ 越界崩溃。
- 易错 **3**（**TLE**）：使用 `endl` 或未关闭流同步，在 $q = 10^6$ 大规模数据下被 IO 卡超时。

自测数据	测试场景	预期输出
1. 最小规模 $n = 1$	`1 2`，初值 `5`；`1 1 1 -8`，`2 1 1`	`-3`（验证负数增量与单点建树）
2. 连续全段加 + 单点查	`4 3`，`1 2 3 4`；`1 1 4 10`，`1 2 3 -5`，`2 2 2`	`7`（验证叠加标记下传准确）
3. 大数防溢出试算	`2 2`，`1000000 1000000`；`1 1 2 1000000000`，`2 1 2`	`2002000000`（验证 `long long` 不溢出）

本页要点

提交 10^6 级别线段树题前，用第 3 组大数试算一次即可放心防溢出！

树状数组与线段树高频错误对照与定位指南

题目 / 场景	常见错误写法	正确做法与原因
树状数组传入 0 下标	直接用原坐标 `add(0, 1)`	`0 & -0 = 0` 导致循环永不增长死循环 **TLE**！必加偏移转 1-based
树状数组值域上限	用了 n 作为树状数组最大下标	值域坐标题（如 $x \le 32000$）应按值域最大值设上界 `MAX_X`
线段树空间开小	只开了 `2 * n` 或 `3 * n`	满二叉树最后一行跨段编号可达 $4n$，统一开 `4 * n + 4`
`push` 后没清 `tag`	忘记写 `tag[p] = 0`	下次经过同一节点会把同一个懒标记反复下发两遍
`add` 后没写 `pull`	更新孩子后忘了更新 `sum[p]`	父亲节点停留在旧和导致查询答案错误 **WA**

- 定位口诀：**TLE** 先查树状数组有没有传进 0；**RE** 查线段树是不是忘了开 `4 * n`；**WA** 查 `push` 有没有清零、回溯有没有 `pull` 和开 `long long`。

本页要点

线段树记住黄金口诀：“开 $4n$、进子前 `push`、清零 `tag`、出子后 `pull`”！

提交前检查清单（六必查）

1. 使用树状数组时，传入 `add(i)` 的下标 `i` 是否保证严格 ≥ 1（如果原输入有 0 有没有 `++x`）？
2. 线段树 `sum` 和 `tag` 数组是否开了 `4 * n + 4` 的空间，且元素类型为 `long long`？
3. 线段树的 `add` 与 `query` 在递归左右儿子之前，是否都调用了 `push(p, l, r)`？
4. `push(p, l, r)` 把增量发给左右儿子后，是否明确把 `tag[p]` 复位成 0？
5. `build` 和 `add` 递归处理完左右孩子后，最后一行有没有汇总更新 `sum[p] = sum[2p] + sum[2p+1]`？
6. 区间修改全覆盖条件写的是不是 `ql <= l && r <= qr`（把被询问段完全包含才返回）？

最后一步

对线段树写一段仅含 5 个元素的小样例，连续做两次整段加后再查单个叶子 $[3,3]$ 验算再提交！

本章全部练习与完成标准

1. 树状数组动态计数：数星星 Stars（一本通 1536，`code/stars_bit.cpp`）、校门外的树（1537）、清点人数（1538）。
2. 懒标记线段树：A Simple Problem with Integers（1548，`code/segtree_lazy.cpp`）、数列操作（1535）、区间和（1547）。

完成标准

能默写树状数组 5 行模板并解释为什么下标不能是 0；能写出带 `apply/push/pull` 完整懒标记线段树；独立 AC 上述精选习题。

课后安排与下一章预告

- 复习：画一遍线段树 $p=1$ $[1,4]$ 修改 $[2,4]$ 时，懒标记停在哪个节点、何时又下传到叶子。
- 练习：独立默写一遍 `code/stars_bit.cpp` 和 `code/segtree_lazy.cpp` 并测通 $x=0$ 与大数溢出场景。
- 下一章预告：第 6 章《倍增 LCA 与树链剖分》——把“区间线段树”搬到“树上任意两点路径”！学习怎样一跳 2^k 步秒找最近公共祖先，以及把整棵树剖成连续重链！

本章一句话

单点改前缀查选 `lowbit` 树状数组最短，区间改区间查上 $4n$ 懒标记线段树最稳。

谢谢！