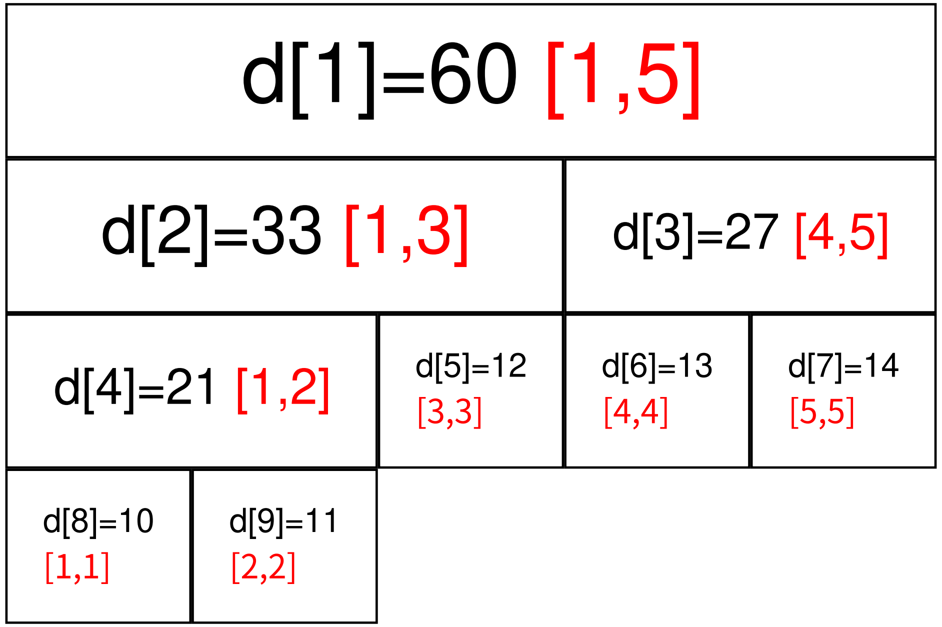
线段树

2024 年 7 月

feecle8146

认识线段树　　支持单点修改区间求和的线段树

下图是一棵维护了序列 [10,11,12,13,14] 的线段树，d 数组表示区间和，红字表示该结点代表的区间。
斜体部分也称为 pushup。

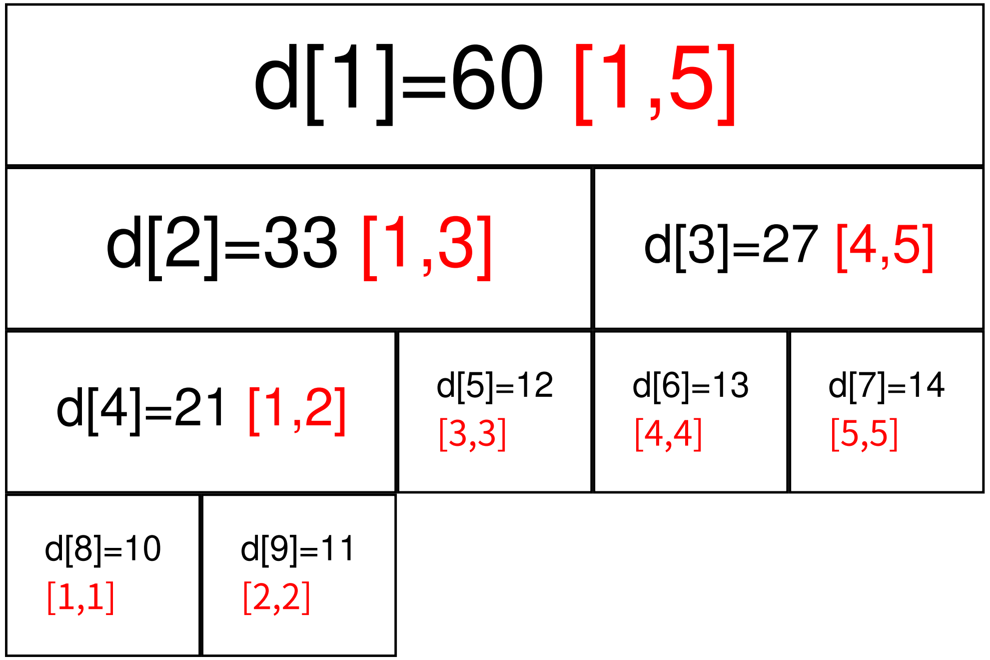



```
int sum[4 * N + 5];
void Build(int p, int l, int r) {
    if (l == r) return sum[p] = a[l], void();
    int mid = (l + r) >> 1;
    Build(p * 2, l, mid);
    Build(p * 2 + 1, mid + 1, r);
    sum[p] = sum[p * 2] + sum[p * 2 + 1];
}

Build(1, 1, n);
```

认识线段树

支持单点修改区间求和的线段树



基本性质：

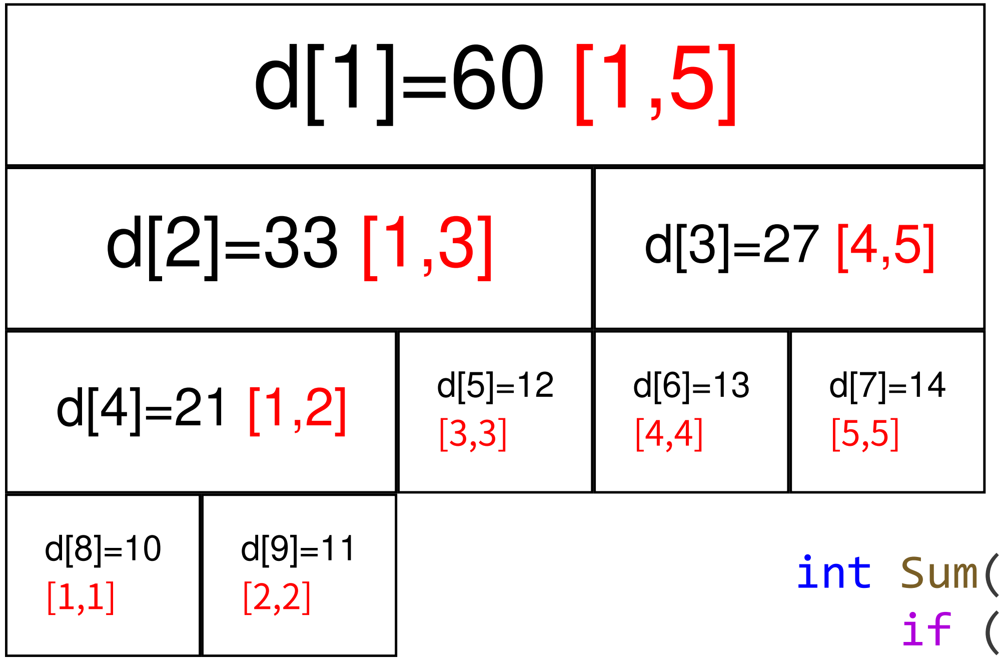
1. 根结点的编号为 1，i 号点的儿子为 $2i, 2i+1$。
2. 共有 $\log_2 n$ 层，第 i 层有 $O(2^i)$ 个结点。
3. 同一层不同结点覆盖的下标不相交。

小练习：试分析下面代码的时间复杂度。

```
void Build(int p, int l, int r) {
    sum[p] = 0;
    for (int i = l; i <= r; i++) sum[p] += a[i];
    if (l == r) return;
    int mid = (l + r) >> 1;
    Build(p * 2, l, mid);
    Build(p * 2 + 1, mid + 1, r);
}

Build(1, 1, n);
```

认识线段树　　支持单点修改区间求和的线段树



区间求和

在线段树上把给定区间拆成个数最少的小区间：例如，$[3,5] = [3,3] + [4,5]$。

思考：下面函数的返回值具体意义是什么？

返回 $[l_p, r_p] \cap [x, y]$ 的和。

```
int Sum(int p, int l, int r, int x, int y) {
    if (x <= l && r <= y) return sum[p];
    int mid = (l + r) >> 1, res = 0;
    if (x <= mid) res += Sum(p * 2, l, mid, x, y);
    if (mid < y) res += Sum(p * 2 + 1, mid + 1, r, x, y);
    return res;
}
```

认识线段树　　支持单点修改区间求和的线段树

区间求和的时间复杂度

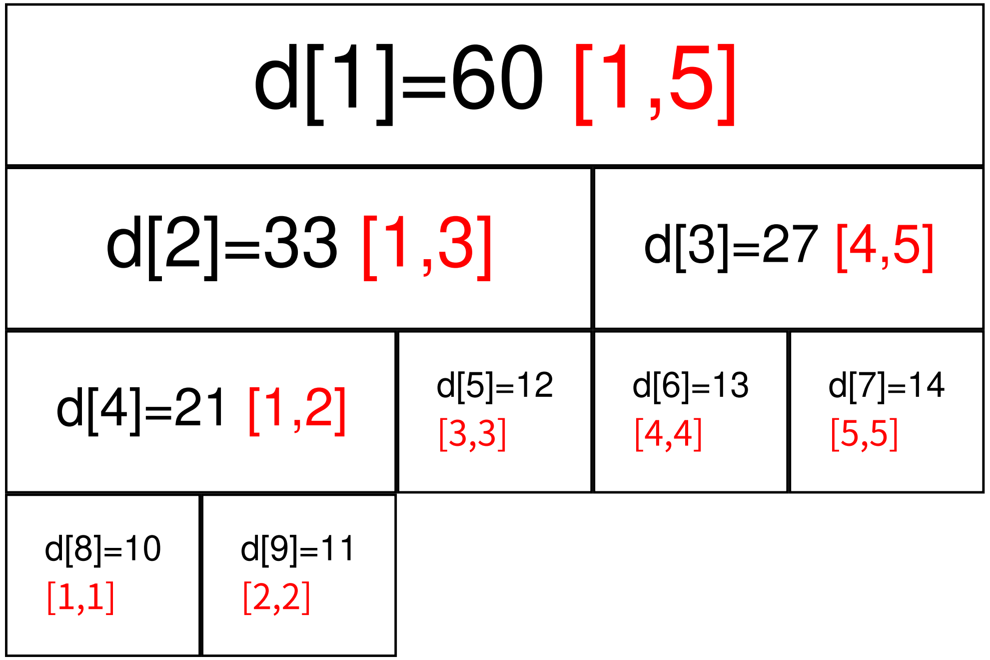
拆分出的区间可以分为“左”“右”两部分，两部分均包含线段树上深度严格递增的若干个结点，且后一个点是前一个点的兄弟的子结点。

按照斜体两部分第一次同时满足的结点分开考虑，其祖先和子树内都访问了 $O(\log n)$ 个点，总共访问结点数量也就不超过 $O(\log n)$。

```
int Sum(int p, int l, int r, int x, int y) {
    if (x <= l && r <= y) return sum[p];
    int mid = (l + r) >> 1, res = 0;
    if (x <= mid) res += Sum(p * 2, l, mid, x, y);
    if (mid < y) res += Sum(p * 2 + 1, mid + 1, r, x, y);
    return res;
}
```

认识线段树

支持单点修改区间求和的线段树



单点修改

在左图的线段树中，假设要将 a_2 修改为 12，则哪些结点的值会受到影响？

事实上，只有 2 对应的叶子到根链上的结点的值需要更新。

若要将 a_x 加上 y，只需把 x 对应叶子到根链上的结点里保存的和加上 y，就能保证线段树的正确性。

循环不变式

上面的解释看上去很清晰，但仍然有含糊其辞的地方。“线段树的正确性”能否更具体地表述？

可以将其拆解为：

1. 查询结果的正确性：查询返回的结果总正确。
2. 修改操作的正确性：每次修改完成后，$\forall p, sum[p] = \sum_{i=l_p}^{r_p} a_i$ 都成立。

在单修区查线段树上，这两点都比较显然，毋需多言。不过，对于类似的命题，有一个专有名词来称呼：循环不变式。

当数据结构变得更复杂时，思考“这个数据结构的循环不变式***具体***是什么”有助于理清思路。

认识线段树　　支持单点修改区间求和的线段树

```
void Upd(int p, int l, int r, int x, int y) {
    // a[x] += y, with increment
    sum[p] += y;
    if (l == r) return;
    int mid = (l + r) >> 1;
    if (x <= mid) Upd(p * 2, l, mid, x, y);
    else Upd(p * 2 + 1, mid + 1, r, x, y);
}
```

单点修改

有两种写法：算变化量，或者每次都重新 pushup。前者常数可能小一些，但适用范围更窄；后者反之。

```
void pushup(int p) { sum[p] = sum[p * 2] + sum[p * 2 + 1]; }

void Upd(int p, int l, int r, int x, int y) {
    // a[x] += y, with pushup
    if (l == r) return sum[p] += y, void();
    int mid = (l + r) >> 1;
    if (x <= mid) Upd(p * 2, l, mid, x, y);
    else Upd(p * 2 + 1, mid + 1, r, x, y);
    pushup(p);
}
```

只要没有特殊要求，通常采用后者。

练习题：洛谷 P3374

认识线段树

支持区间加区间求和的线段树

懒标记

懒标记的基本思路是，只有当需要用到某个值时，才把修改真正作用在这个值上。

操作 $[l,r]$ “直接用到”的结点就是构成 $[l,r]$ 的最小覆盖的结点；而找到这些结点的过程又用到了这些结点的祖先结点。

查询/修改 [3,5]，如左图。

无论查询还是修改，用到的结点上都不该有标记了。

认识线段树　　　　支持区间加区间求和的线段树

懒标记

懒标记的基本思路是，只有当需要用到某个值时，才把修改真正作用在这个值上。
循环不变式：任意操作后，总有 $\forall p$，

$$\sum_{i=l_p}^{r_p} a_{i\,(\text{真实值})} = sum_p + (r_p - l_p + 1) \times \sum_{q\text{是}p\text{的祖先},q\neq p} tag_q$$

一般，同一个结点上维护的两个值最好不要相互影响，所以 sum_p 和 tag_p 不应同时出现在上式中。

理解上式，你就掌握了调试线段树代码的根本方法：将你的代码的输出结果和上式对比，看什么时候不满足了。

认识线段树 支持区间加区间求和的线段树

懒标记

目标： 无论查询还是修改，用到的结点上都不该有标记了。
循环不变式： 任意操作后，总有 $\forall p$，

$$\sum_{i=l_p}^{r_p} a_{i\,(\text{真实值})} = sum_p + (r_p - l_p + 1) \times \sum_{q\text{是}p\text{的祖先},q\neq p} tag_q$$

对 $[l,r]$ 进行任意操作时，首先在线段树上定位出区间，并进行适当的标记下传。

请你根据目标和循环不变式，回答以下问题：
1. 标记下传的具体含义是？
2. 同一操作时，不同结点的标记下传有没有顺序要求？
3. 下传完标记后，查询和修改应分别进行什么后续操作？

认识线段树

支持区间加区间求和的线段树

懒标记

$$\sum_{i=l_p}^{r_p} a_{i\,(\text{真实值})} = sum_p + (r_p - l_p + 1) \times \sum_{q\text{是}p\text{的祖先},q\neq p} tag_q$$

问题 1:

```
void Tag(int p, int z) { tag[p] += z, sum[p] += (R[p] - L[p] + 1) * z; }
void pushdown(int p) {
    if (tag[p]) {
        Tag(p * 2, tag[p]);
        Tag(p * 2 + 1, tag[p]);
        tag[p] = 0;
    }
}
```

思考：为了维持循环不变式，pushdown(p) 后需不需要 pushup(p)?

不需要

请你根据目标和循环不变式，回答以下问题：

1. 标记下传的具体含义是？
2. 同一操作时，不同结点的标记下传有没有顺序要求？
3. 下传完标记后，查询和修改应分别进行什么后续操作？

认识线段树

支持区间加区间求和的线段树

$$\sum_{i=l_p}^{r_p} a_{i\,(\text{真实值})} = sum_p + (r_p - l_p + 1) \times \sum_{q\text{是}p\text{的祖先},q\neq p} tag_q$$

懒标记

问题 2：应由浅到深下传标记。

问题 3：
查询操作，由于用到的结点上都没有标记，所以直接返回所有拆分出来结点的 sum_p 之和。
修改操作，只需对拆分出来的结点都执行 Tag，然后往上 pushup。

思考：查询操作需不需要调用 pushup？
不需要

请你根据目标和循环不变式，回答以下问题：
1. 标记下传的具体含义是？
2. 同一操作时，不同结点的标记下传有没有顺序要求？
3. 下传完标记后，查询和修改应分别进行什么后续操作？

认识线段树

支持区间加区间求和的线段树

$$\sum_{i=l_p}^{r_p} a_{i\,(\text{真实值})} = sum_p + (r_p - l_p + 1) \times \sum_{q\text{是}p\text{的祖先},q\neq p} tag_q$$

懒标记

问题 2：应由浅到深下传标记。

问题 3：
查询操作，由于用到的结点上都没有标记，所以直接返回所有拆分出来结点的 sum_p 之和。
修改操作，只需对拆分出来的结点都执行 Tag，然后往上 pushup。

```
void Upd(int p, int l, int r, int x, int y, int z) {
    if (x <= l && r <= y) return Tag(p, z);
    pushdown(p);
    int mid = (l + r) >> 1;
    if (x <= mid) Upd(p * 2, l, mid, x, y, z);
    if (mid < y) Upd(p * 2 + 1, mid + 1, r, x, y, z);
    pushup(p);
}
```

认识线段树

支持区间加区间求和的线段树

$$\sum_{i=l_p}^{r_p} a_{i\,(\text{真实值})} = sum_p + (r_p - l_p + 1) \times \sum_{q\text{是}p\text{的祖先},q\neq p} tag_q$$

懒标记

问题 2：应由浅到深下传标记。

问题 3：
查询操作，由于用到的结点上都没有标记，所以直接返回所有拆分出来结点的 sum_p 之和。
修改操作，只需对拆分出来的结点都执行 Tag，然后往上 pushup。

```
int Sum(int p, int l, int r, int x, int y) {
    if (x <= l && r <= y) return sum[p];
    pushdown(p);
    int mid = (l + r) >> 1, res = 0;
    if (x <= mid) res += Sum(p * 2, l, mid, x, y);
    if (mid < y) res += Sum(p * 2 + 1, mid + 1, r, x, y);
    return res;
}
```

思考：叶子结点会不会被 pushdown？
不会

认识线段树

支持区间加区间求和的线段树

$$\sum_{i=l_p}^{r_p} a_{i\,(\text{真实值})} = sum_p + (r_p - l_p + 1) \times \sum_{q\text{是}p\text{的祖先},q\neq p} tag_q$$

小结

理解一个算法的最好方式是

1. *具体地写出它满足的循环不变式*
2. 手动在小数据上模拟算法的运行流程

既要有感性的总体认知，也要有理性的具体分析。

通过今天的学习，你应当掌握：
1. 线段树区间查询时间复杂度为何正确
2. 查询区间时，用到了哪些点
3. 懒标记的循环不变式

认识线段树　　支持区间加乘区间求和的线段树

设计标记

现在在之前的操作基础上，加入区间乘操作。

为了使线段树同时支持两种操作，有两种修改方法：

1. 打两个标记 $addtag$ 和 $multag$。
2. 分析标记性质，只打一种标记。

我们将采用第二种，只打一种标记。第一种的劣势是，同一个结点上不同标记存在顺序问题，当标记变得更复杂时难以弄清。

同时，第二种维护方法是普适的：只要一个问题能用线段树解决，就能只用一个标记解决。

认识线段树

支持区间加乘区间求和的线段树

设计标记

对 x 依次执行若干个 $x \to x+a$、$x \to bx$ 的操作后，最终的值总是什么形式？

总可以写作 $x \to Ax + B$。

- $x \to x + a$：$B \to B + a$。
- $x \to bx$：$A \to bA$，$B \to bB$。

思考：tag 数组初值应该是什么？

(1,0)

因此，可以设计 tag 数组为一个结构体，其中包含两个数 A, B。$tag_p = (A, B)$ 表示 $[l_p, r_p]$ 内的所有数 x 都应当被修改为 $Ax + B$。

加 a 操作等价于 $(1, a)$；乘 b 操作等价于 $(b, 0)$。

标记的复合封闭性

“执行若干操作后，最终的值总是某形式”有一个专有名词来称呼：标记在复合意义下是封闭的。之所以叫做复合，是因为标记 tag 可以看作值 x 到值 y 的函数：x 在打了 tag 后变成 y，就说 $tag(x)=y$。而 x 打了 tag_1 之后再打 tag_2，其实就是 $x \rightarrow tag_2(tag_1(x))$。

我们设计线段树上的标记时，复合封闭性是必须满足的条件。

通常用 $\circ$ 符号表示标记的复合：$(f \circ g)(x) = f(g(x))$。

设计满足复合封闭性标记的方法，就是看 x 在经过若干操作后有没有什么普遍的形式。

认识线段树　　支持区间加乘区间求和的线段树

设计标记

懒标记的基本思路是，只有当需要用到某个值时，才把修改真正作用在这个值上。

目标： 无论查询还是修改，用到的结点上都不该有标记了。

思考：之前的 Upd 和 Sum 函数需要进行怎样的修改？

```
void Upd(int p, int l, int r, int x, int y, Tag z) {
    if (x <= l && r <= y) return maketag(p, z);
    pushdown(p);
    int mid = (l + r) >> 1;
    if (x <= mid) Upd(p * 2, l, mid, x, y, z);
    if (mid < y) Upd(p * 2 + 1, mid + 1, r, x, y, z);
    pushup(p);
} // Upd 只需把 z 改成标记结构体!
```

```
int Sum(int p, int l, int r, int x, int y) {
    if (x <= l && r <= y) return sum[p];
    pushdown(p);
    int mid = (l + r) >> 1, res = 0;
    if (x <= mid) res += Sum(p * 2, l, mid, x, y);
    if (mid < y) res += Sum(p * 2 + 1, mid + 1, r, x, y);
    return res;
} // Sum 完全不需要修改!
```

认识线段树　　支持区间加乘区间求和的线段树

下传标记

思考：下传标记的过程和之前相比有什么变化?

1. 需要注意标记的顺序问题，父亲的标记更新，儿子已有的标记更老。
2. 标记应当清空为 (1,0)。

```
struct Tag {
    int A, B; // x -> Ax + B
};

Tag tag[4 * N + 5];

Tag operator+(const Tag &x, const Tag &y) {
    // 先 x 后 y
    return (Tag){x.A * y.A, x.B * y.A + y.B};
}
```

```
void maketag(int p, Tag z) {
    tag[p] = tag[p] + z;
    sum[p] = sum[p] * z.A + z.B * (R[p] - L[p] + 1);
}
// 注意调用 + 的先后顺序

void pushdown(int p) {
    if (tag[p].A != 1 || tag[p].B != 0) {
        maketag(p * 2, tag[p]);
        maketag(p * 2 + 1, tag[p]);
        tag[p] = (Tag){1, 0};
    }
}
```

认识线段树　　支持区间加乘区间求和的线段树

标记有顺序的循环不变式

请你写出区间加乘区间求和的线段树的循环不变式，想一想为什么它总是正确的。

$$\sum_{i=l_p}^{r_p} a_{i\,(\text{真实值})}$$

$= sum_p$ 在被祖先标记按照由深到浅(也就是由旧到新)的顺序作用后的值

一般地，把 sum 换成别的信息，这样的循环不变式也成立。

试看看！

经典问题 无来源

给定一个数组，支持区间加、区间取 max、求区间最大值。

$$n, q \le 200000$$

请你设计标记，并说明 maketag 和 pushdown 函数的写法。

标记：$x \to \max(x + a, b)$

试看看！

经典问题 无来源

给定一个数组，支持区间加、区间赋值、求区间最大值、求区间和。

$$n, q \leq 200000$$

请你设计标记，并说明 maketag 和 pushdown 函数的写法。

标记：维护 $(flag, val)$。若 $flag = 0$，表示 $x \to x + val$；若 $flag = 1$，表示 $x \to val$。

认识线段树

支持区间最大子段和的线段树

最大子段和 SP1043 / P4513

给定一个数组，支持单点修改，求区间最大子段和。

$n, q \leq 200000$

本题中需要设计 pushup 的信息。若仅仅维护 ans_p 表示 $[l_p, r_p]$ 的最大子段和，则无法 pushup，因为新的最大子段和可能跨过区间中点。

但如果跨过区间中点，前半部分和后半部分就独立了，分别取最大后缀和最大前缀即可。因此，还得维护最大后缀和最大前缀。

为了求最大后缀、最大前缀，还需要维护……

认识线段树　　支持区间最大子段和的线段树

最大子段和　　SP1043 / P4513

给定一个数组，支持单点修改，求区间最大子段和。

$n, q \leq 200000$

对每个结点维护 (sum, lmx, rmx, ans) 分别表示和、最大前缀、最大后缀、最大子段和，合并时分类讨论。为了避免空信息参与合并，在查询时需要注意代码细节，如右图。

```
Info Query(int p, int l, int r, int x, int y) {
    if (x <= l && r <= y) return t[p];
    int mid = (l + r) >> 1;
    if (y <= mid) return Query(p * 2, l, mid, x, y);
    if (mid < x) return Query(p * 2 + 1, mid + 1, r, x, y);
    return Query(p * 2, l, mid, x, y) + Query(p * 2 + 1, mid + 1, r, x, y); // 不用 res，避免赋初值
}
```

本题有什么启示？

信息的可合并性

“能够 pushup”有一个专有名词来称呼：信息是可合并的。

如果只维护题目要求的信息 a 不可合并，不妨想想：如果已知 a_l 和 a_r，还需要知道什么（叫做 b）才能算出 a_p。

进一步，如果 (a,b) 还是不能算出 b，想想：如果已知 b_l 和 b_r，还需要知道什么才能算出 b_p……

依次类推，直到可合并。如果无论多少信息都不可合并，就需要反思一下这道题能否直接用线段树解决。