线段树

快速入门不求甚解的

Koshiba / 2021.06

线段树的诱人特性

- 线段树是一种**数据结构**，它可以有效地回答对数组的**范围查询**，例如以 **O(log n)** 寻找连续数组元素 a[l…r] 的**和**，或在 a[l…r] 中找到**最大**、**最小元素**等。
- 线段树允许修改数组，如替换其中的一个元素，甚至改变整个子段的元素
- 线段树还可以很容易地被推广到更大的维度，例如通过二维的片段树，可以以 $O(\log^2 n)$ 时间回答一个给定矩阵的某个子矩形上的总和或最小值查询。
- 线段树只需要**线性的内存**：标准的片段树使用4n顶点来处理大小为n的数组。
- 总的来说，线段树是一个非常灵活的数据结构，用它可以解决大量的问题。此外，线段树还可以应用更复杂的操作，回答更复杂的查询。

要学好，先思考

- 为什么可以用 O(log n) 的时间查询到 n 个连续元素的和呢?
  - 计算 n 个元素的和，时间必然为 O(n)
  - 因此线段树这种数据结构必然维护了某些性质和信息，使得查询时可以直接使用
- O(log n) 很暧昧，让人想浮想联翩
  - 分治法
  - 平衡的 x 叉树上的各种查询
- 线段树的核心正是分治法和平衡 x 叉树

跑个题：分治法

- 分治，即分而治之：其过程除分和治外
  - 最重要的是没提到的合并操作
- 分治法举例：归并排序
  - 分——将一组 n 个元素一分为二组，二分为四组……直至 n 组
  - 治：~~没得治子。~~不需要治
  - 合并：小的元素排前面，大的排后面，二组合成新的一组
- 总结一下其实就是

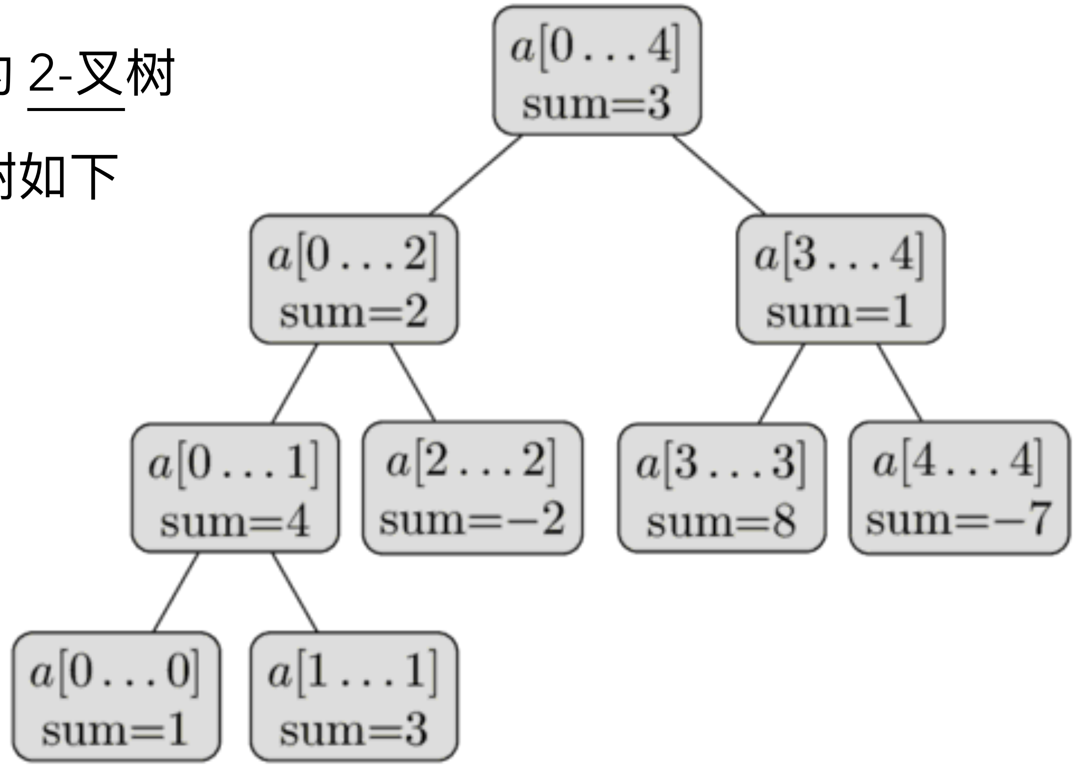
$$
\begin{aligned}
h([x]) &= \overset{\text{治}}{f(x)} \\
h(xs \underset{\text{分}}{+\!\!+} ys) &= h(xs) \overset{\text{合}}{\odot} h(ys)
\end{aligned}
$$

回归主题：线段树

- 简单起见，我们考虑线段树的最简单形式。
- 任务：给定一个数组 a[0…n-1]，必须能够找到索引 l 和 r 之间的所有元素的和，即 $\Sigma_{i=l}^{r} a[i]$。且可以修改数组的元素。两种操作需要在 O(log n) 内完成。
- S1 用分治法建立线段树：
  - 拆解 a[0…n-1] 为 a[0…n/2] 和 a[n/2 + 1… n-1]
  - 计算 a[0…n/2] 和 a[n/2 + 1… n-1] 和，并存储
  - 当然了，上面那一步也是要先拆后求的（拆到叶子，即单个元素再求）

S2 观察树的结构

- 使用分治法，我们得到了一棵很<u>平衡</u>的 <u>2-叉</u>树
- 假设数组为 a = [1,3,-2,8,-7]，得到的树如下
- 一些事实
  - 树高 log n
  - 节点最多 4n
  - 假设 merge 是 O(1)，
    那么建树是 O(n)



S3 试试查询

- 很多 segment 的和都预先算好了，因此查询 a[l...r] 的和才可以快上加快
- 假设我们现在位于存储了 a[tl...tr] 的和的节点

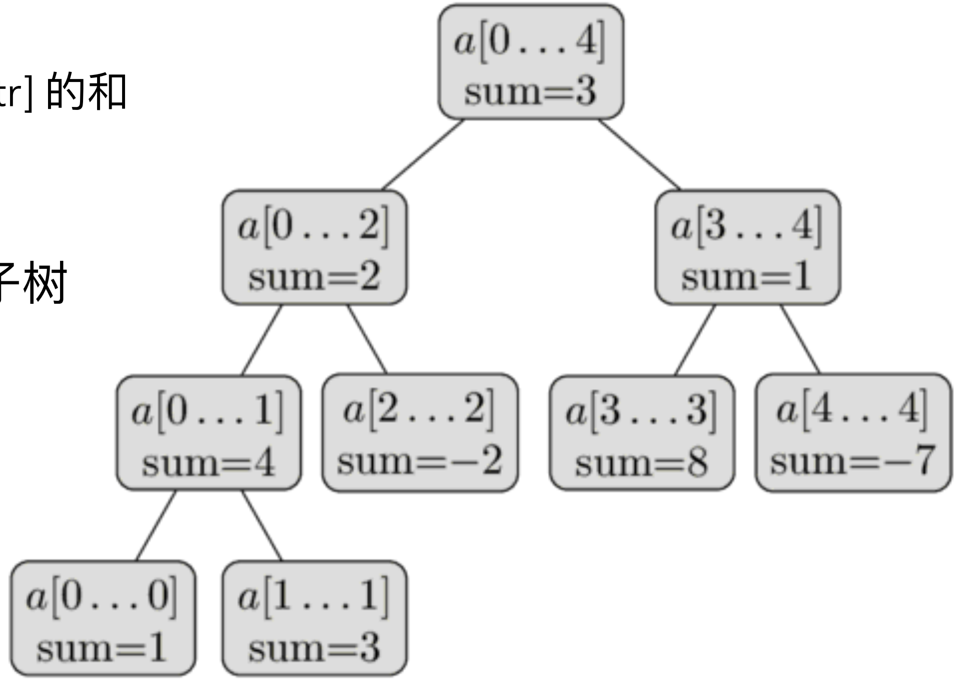
  当前节点存储了a[tl...tr] 的和，设 $tm = (tl + tr)/2$

  则左子树存储了 a[tl… tm] 的和，右子树存储了 a[tm + 1...tr] 的和

  - Case 1: $l = tl, r = tr$，存储的和就是我们要的
  - Case 2: $tr \leq tm$ 或 $tl \geq tm$，递归查询左子树或右子树
  - Case 3: $tl \leq l < tm < r \leq tr$,

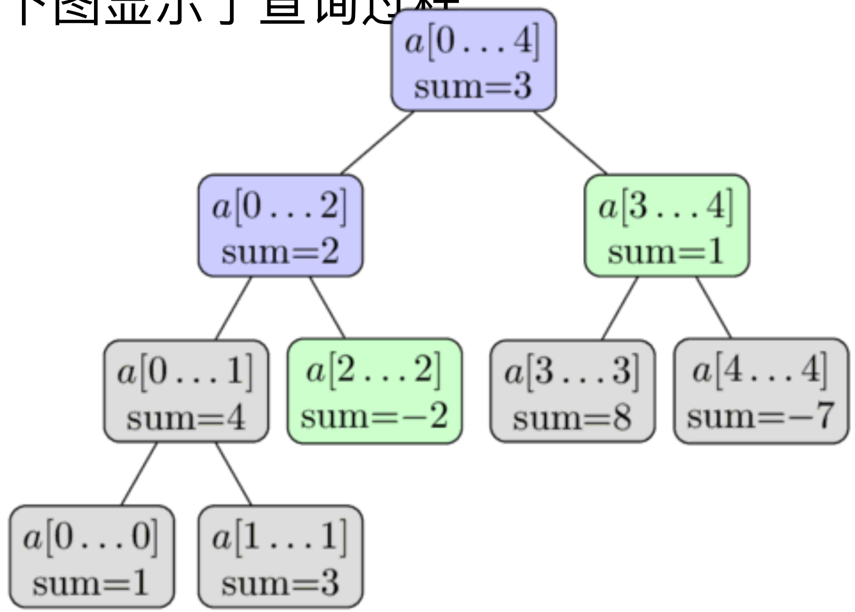
    左右子树都得查，并合并结果

    - a[l...tm] 的结果由左子树得到
    - a[tm + 1...r] 的结果由右子树得到



S3 试试查询

- 假设数组为 $a = [1,3,-2,8,-7]$，查询为$\Sigma_{i=2}^{4} a[i]$。下图显示了查询过程
  - 被访问的节点被标为蓝色、绿色
  - 直接使用其存储的和的节点为绿色



- 速度为什么是 O(log n) 呢?
  - 树高 O(log n)
  - 每一层不会访问超过 4 个节点
- 假设当前层访问的节点不超过 4 个
  - 若当前层访问的节点为1-2个，显然下一层访问不超过 4 个节点
  - 若当前层访问的节点为 3-4 个，则当前层的中间 1-2 个节点不会生成递归调用，会立即返回。因为我们查询的是连续子元素，只有最左和最右的节点会产生递归调用（上一页 case 3）

S3.5 查询速度

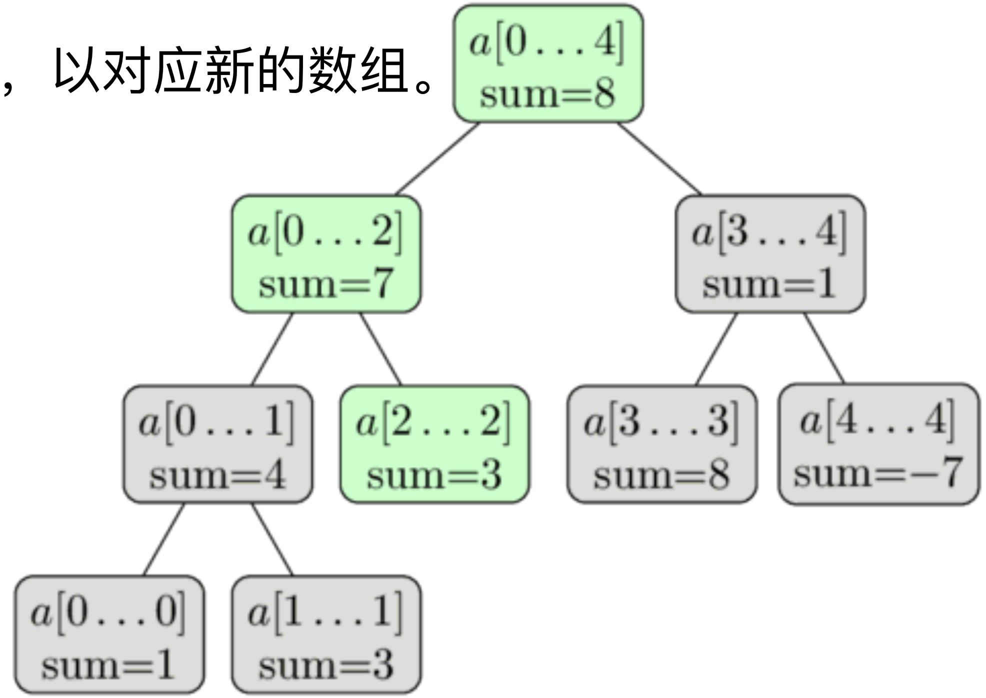
- Case 3: $tl \le l < tm < r \le tr$,
  左右子树都得查，并合并结果
  - a[l...tm] 的结果由左子树得到
  - a[tm + 1...r] 的结果由右子树得到

- 速度为什么是 O(log n) 呢?
  - 树高 O(log n)
  - 每一层不会访问超过 4 个节点

- 假设当前层访问的节点不超过 4 个
  - 若当前层访问的节点为1-2个，显然下一层访问不超过 4 个节点
  - 若当前层访问的节点为 3-4 个，则当前层的中间 1-2 个节点不会生成递归调用，会立即返回。因为我们查询的是连续子元素，只有最左和最右的节点会产生递归调用（case 3）
  - 例如 $l < (tl + tm)/2$ 时，下一层中，对右子树的调用将直接返回，对左子树的调用才有新的递归调用；$l \ge (tl + tm)/2$ 时，下一层只有一个对右子树的递归调用

S4 试试更新 & 更新速度

- 假设数组（还）是$a = [1,3,-2,8,-7]$，我们要做更新$a[2] = 3$。
- 那么跟新完后我们要更新整棵树存储的信息，以对应新的数组。
  - 学过堆的都知道怎么做吧
- 因为每一层最多只改一个 node，
  时间复杂度 O(log n)
- 图示：绿色的节点存储的信息被更新



线段树的实现

- 虽然我们前面一口一个数组，但显然线段树只代表了一种思想。
- 我们完全可以用链表来存储（经典树状结构），时间复杂度不变
  - 不讲了
- 这里看一下用数组怎么存（其实也就是堆，不讲了）
  - 给定数组 a，为查询连续元素的和制作对应的线段树
  - v 是当前 node（的下标）
  - tl 和 tr 是当前 segment 的左端点和右端点

```
int n, t[4*INPUT_N];

void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = t[v*2] + t[v*2+1];
    }
}
```

线段树的实现

- 查询代码

```
int sum(int v, int tl, int tr, int l, int r) {
    if (l > r) {return 0;}
    if (l == tl && r == tr) { return t[v]; }
    int tm = (tl + tr) / 2;
    return sum(v*2, tl, tm, l, min(r, tm))
           + sum(v*2+1, tm+1, tr, max(l, tm+1), r);
}
```

- 更新代码

```
void update(int v, int tl, int tr, int pos, int new_val) {
    if (tl == tr) {
        t[v] = new_val;
    } else {
        int tm = (tl + tr) / 2;
        if (pos <= tm) { update(v*2, tl, tm, pos, new_val); }
        else { update(v*2+1, tm+1, tr, pos, new_val); }
        t[v] = t[v*2] + t[v*2+1];
    }
}
```

扩展话题

数组存储优化

- 当数组长度 n 满足 $n = 2^k + 1, k \in \mathbb{N}$ 时，需要使用 4n 的存储空间
- 因为最后一层的 2n 的存储空间只存了 1 个叶子
- 因此 2n 的空间就够了

- 下标计算
  - 设在线段树中，当前节点的下标是 v，该节点负责存储 [l,r] 之间的某信息（如和）。
  - 设 $mid = (l + r)/2$
  - 左子树下标是 v+1，负责存储 [l,mid] 的某信息，共有 $2 * (mid - l + 1) - 1$ 个节点
  - 所以右子树可以从 $v + 2 * (mid - l + 1)$ 开始

一些常见的查询

- 区间和
- 区间的最大、最小元素
- 区间的最大元素和出现次数
- 计算区间的最大公约数、最小公倍数
- 统计区间中 0 的个数、查找第 k 个 0
- 给定 x，求解最小下标 i，使得数组的前 i 个元素的和不小于 x
- 给定 x，查找区间内第一个超过 x 的元素的下标
- 给定区间 [l,r]，找到子区间 [l′, r′] 使得和最大（滑动窗口也可）
- 还有好多，看这里吧：https://cp-algorithms.com/data_structures/segment_tree.html

线段树的区间修改

- 要求在修改区间时把所有包含在区间 [l,r] 中的节点都遍历、修改一次，时间复杂度顶不住。我们这里引入 lazy propogation。
- 简单的思想：在尽量少的节点记录新值并适当做标记
    - 例如要把 a[l…r] 的值都加上 x，则只在 query 到的每个节点处记录”+ x“ 这个信息。树高 log n，这样最多也只会记录 log n 次。如果下次 query 的时候查到了 a[l…r] 中的数据，则顺道在查询的时候利用上 “+ x” 就行了。
    - 例如要把 a[l…r] 都赋值成 p，设当前节点为 v，则 v 下的每个子节点都要被赋值成 p 时，直接在 v 上做一个标记 p 并跳过对子节点的递归处理。该标记表示从节点 v 开始，所有子子孙孙节点的信息都作废。因此复杂度也是 log n。之后要把 a[l…r] 中的某些元素赋值成其他值时，需要将路径上有标记的节点”下推“。例如一开始 a[0…n-1] = p，则根节点被标记为 p。后来 a[0… n/2] = q，则需要把标记 p 下推到两棵子树，最终只有右子树的标记还是 p。
    - ……

LeetCode 题目

- TBD