cloud.tencent.com

归并排序的正确理解方式及运用 - 腾讯云开发者社区-腾讯云

labuladong

7-9 minutes

一直都有很多读者说，想让我用 框架思维 讲一讲基本的排序算法，我觉得确实得讲讲，毕竟学习任何东西都讲求一个融会贯通，只有对其本质进行比较深刻的理解，才能运用自如。

本文就先讲归并排序，给一套代码模板，然后讲讲它在算法问题中的应用。阅读本文前我希望你读过前文 手把手刷二叉树（纲领篇）。

我在 手把手刷二叉树（第一期） 讲二叉树的时候，提了一嘴归并排序，说归并排序就是二叉树的后序遍历，当时就有很多读者留言说醍醐灌顶。

知道为什么很多读者遇到递归相关的算法就觉得烧脑吗？因为还处在「看山是山，看水是水」的阶段。

就说归并排序吧，如果给你看代码，让你脑补一下归并排序的过程，你脑子里会出现什么场景？

这是一个数组排序算法，所以你脑补一个数组的 GIF，在那一个个交换元素？如果是这样的话，那格局就低了。

但如果你脑海中浮现出的是一棵二叉树，甚至浮现出二叉树后序遍历的场景，那格局就高了，大概率掌握了 框架思维，用这种抽象能

力学习算法就省劲多了。

那么，归并排序明明就是一个数组算法，和二叉树有什么关系？接下来我就具体讲讲。

算法思路

就这么说吧，所有递归的算法，你甭管它是干什么的，本质上都是在遍历一棵（递归）树，然后在节点（前中后序位置）上执行代码，你要写递归算法，本质上就是要告诉每个节点需要做什么。

你看归并排序的代码框架：

```
// 定义：排序 nums[lo..hi]
void sort(int[] nums, int lo, int hi) {
    if (lo == hi) {
        return;
    }
    int mid = (lo + hi) / 2;
    // 利用定义，排序 nums[lo..mid]
    sort(nums, lo, mid);
    // 利用定义，排序 nums[mid+1..hi]
    sort(nums, mid + 1, hi);

    /****** 后序位置 ******/
    // 此时两部分子数组已经被排好序
    // 合并两个有序数组，使 nums[lo..hi] 有序
    merge(nums, lo, mid, hi);
    /**********************/
}

// 将有序数组 nums[lo..mid] 和有序数组 nums[mid+1..hi]
```

```
// 合并为有序数组 nums[lo..hi]
void merge(int[] nums, int lo, int mid, int hi);
```

看这个框架，也就明白那句经典的总结：归并排序就是先把左半边数组排好序，再把右半边数组排好序，然后把两半数组合并。

上述代码和二叉树的后序遍历很像：

```
/* 二叉树遍历框架 */
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    traverse(root.left);
    traverse(root.right);
    /****** 后序位置 ******/
    print(root.val);
    /**********************/
}
```

再进一步，你联想一下求二叉树的最大深度的算法代码：

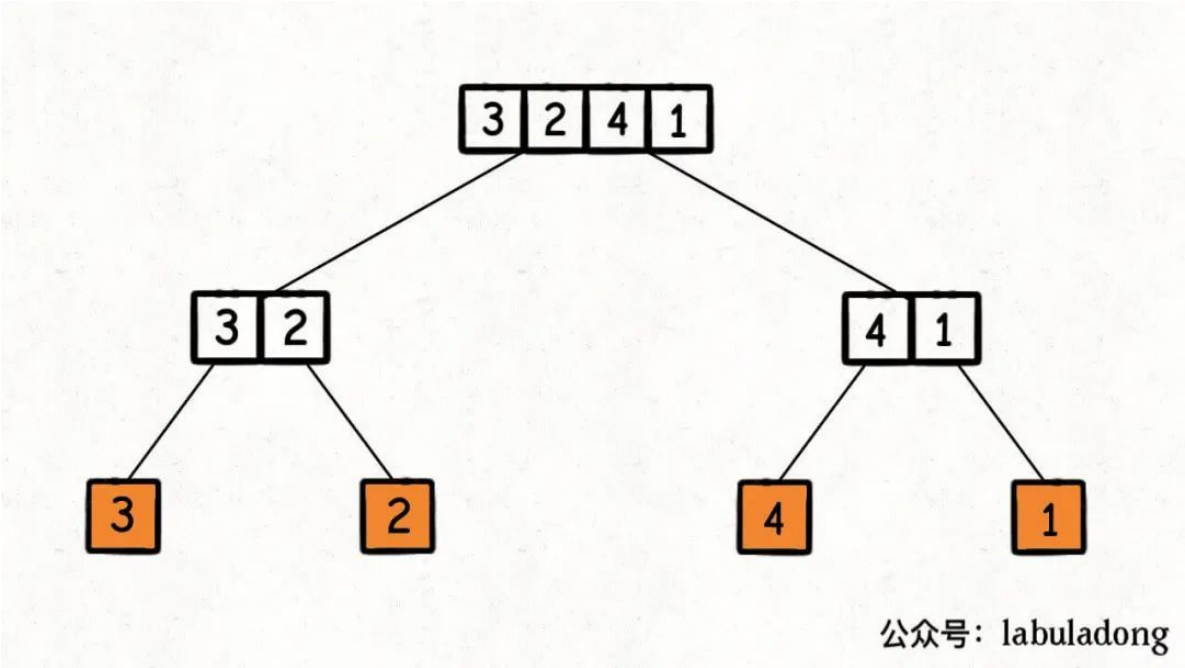
```
// 定义：输入根节点，返回这棵二叉树的最大深度
int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    // 利用定义，计算左右子树的最大深度
    int leftMax = maxDepth(root.left);
    int rightMax = maxDepth(root.right);
    // 整棵树的最大深度等于左右子树的最大深度取最大值，
    // 然后再加上根节点自己
    int res = Math.max(leftMax, rightMax) + 1;
```

```
    return res;
}
```

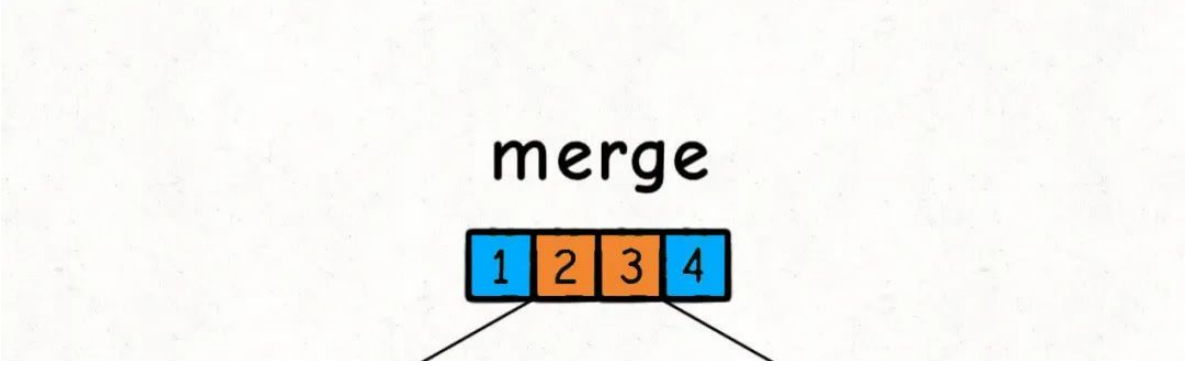
是不是更像了？

前文 手把手刷二叉树（纲领篇） 说二叉树问题可以分为两类思路，一类是遍历一遍二叉树的思路，另一类是分解问题的思路，根据上述类比，显然归并排序利用的是分解问题的思路（分治算法）。

归并排序的过程可以在逻辑上抽象成一棵二叉树，树上的每个节点的值可以认为是`nums[lo..hi]`，叶子节点的值就是数组中的单个元素：

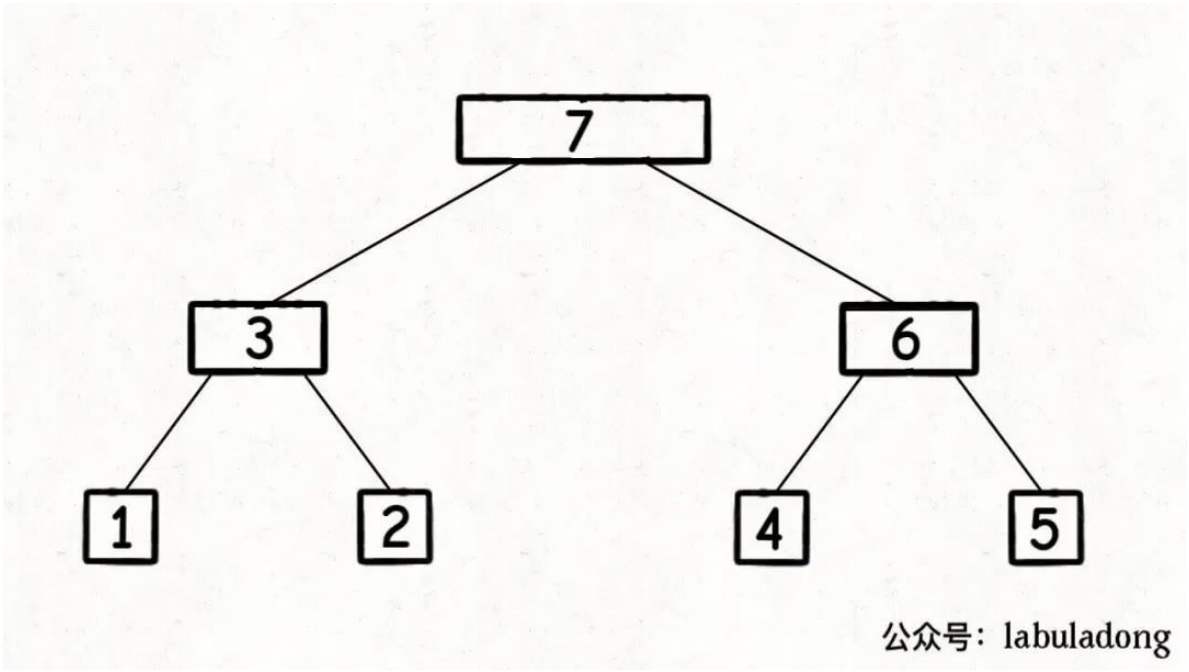


然后，在每个节点的后序位置（左右子节点已经被排好序）的时候执行`merge`函数，合并两个子节点上的子数组：

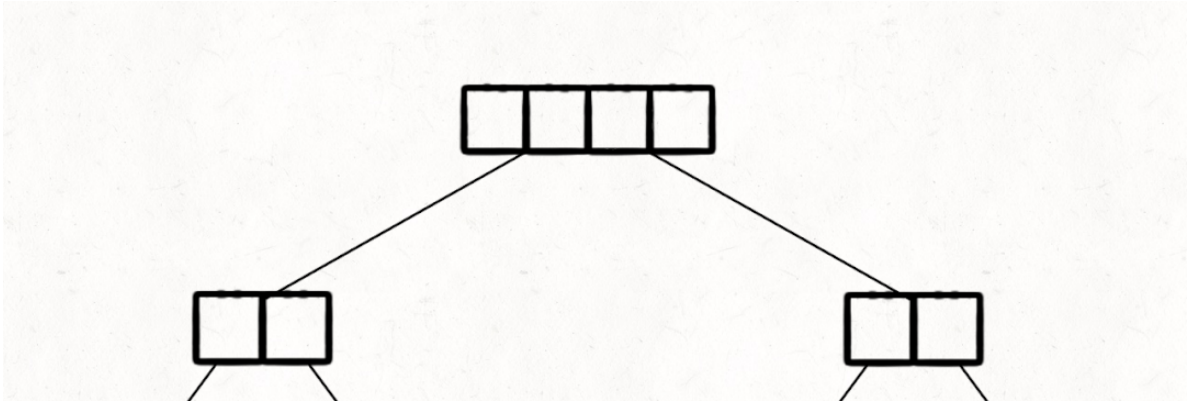


这个merge操作会在二叉树的每个节点上都执行一遍，执行顺序是二叉树后序遍历的顺序。

后序遍历二叉树大家应该已经烂熟于心了，就是下图这个遍历顺序：



结合上述基本分析，我们把nums[lo..hi]理解成二叉树的节点，srot函数理解成二叉树的遍历函数，整个归并排序的执行过程就是以下 GIF 描述的这样：



这样，归并排序的核心思路就分析完了，接下来只要把思路翻译成代码就行。

代码实现及分析

只要拥有了正确的思维方式，理解算法思路是不困难的，但把思路实现成代码，也很考验一个人的编程能力。

毕竟算法的时间复杂度只是一个理论上的衡量标准，而算法的实际运行效率要考虑的因素更多，比如应该避免内存的频繁分配释放，代码逻辑应尽可能简洁等等。

经过对比，《算法 4》中给出的归并排序代码兼具了简洁和高效的特点，所以我们可以参考书中给出的代码作为归并算法模板：

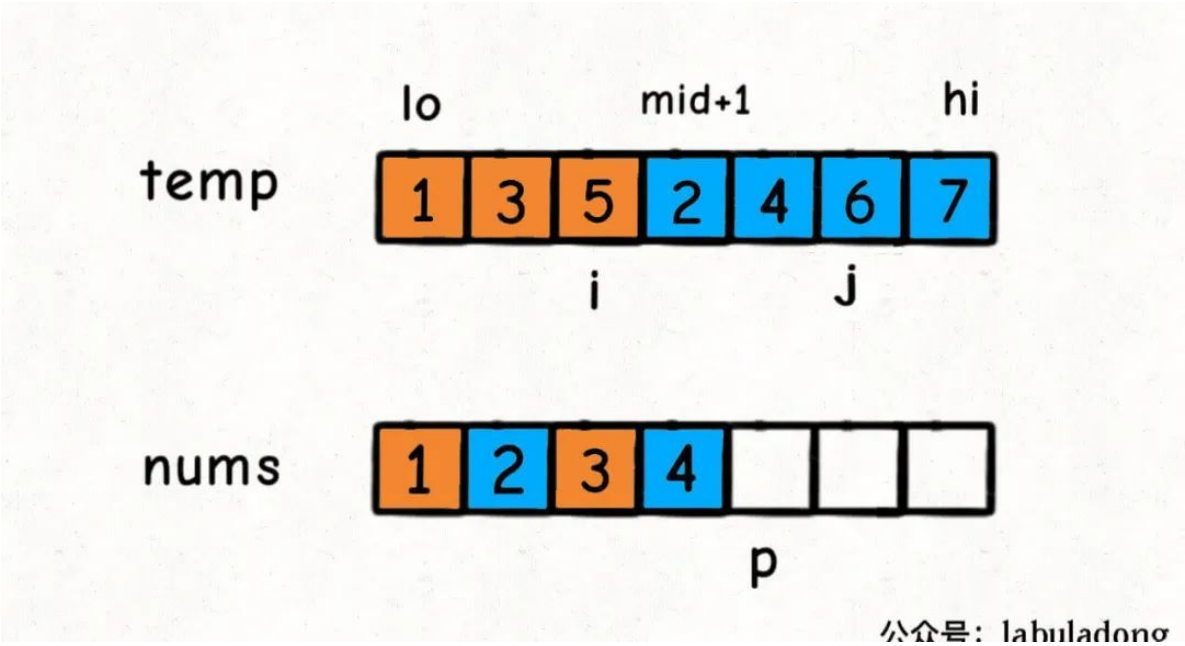
```
class Merge {

    // 用于辅助合并有序数组
    private static int[] temp;

    public static void sort(int[] nums) {
        // 先给辅助数组开辟内存空间
        temp = new int[nums.length];
        // 排序整个数组（原地修改）
        sort(nums, 0, nums.length - 1);
    }

    // 定义：将子数组 nums[lo..hi] 进行排序
```

```
    private static void sort(int[] nums, int lo, int
hi) {
        if (lo == hi) {
            // 单个元素不用排序
            return;
        }
        // 这样写是为了防止溢出，效果等同于 (hi + lo) /
2
        int mid = lo + (hi - lo) / 2;
        // 先对左半部分数组 nums[lo..mid] 排序
        sort(nums, lo, mid);
        // 再对右半部分数组 nums[mid+1..hi] 排序
        sort(nums, mid + 1, hi);
        // 将两部分有序数组合并成一个有序数组
        merge(nums, lo, mid, hi);
    }

    // 将 nums[lo..mid] 和 nums[mid+1..hi] 这两个有序数
组合并成一个有序数组
    private static void merge(int[] nums, int lo, int
mid, int hi) {
        // 先把 nums[lo..hi] 复制到辅助数组中
        // 以便合并后的结果能够直接存入 nums
        for (int i = lo; i <= hi; i++) {
            temp[i] = nums[i];
        }

        // 数组双指针技巧，合并两个有序数组
        int i = lo, j = mid + 1;
        for (int p = lo; p <= hi; p++) {
```

```
            if (i == mid + 1) {
                // 左半边数组已全部被合并
                nums[p] = temp[j++];
            } else if (j == hi + 1) {
                // 右半边数组已全部被合并
                nums[p] = temp[i++];
            } else if (temp[i] > temp[j]) {
                nums[p] = temp[j++];
            } else {
                nums[p] = temp[i++];
            }
        }
    }
}
```

有了之前的铺垫，这里只需要着重讲一下这个merge函数。

sort函数对nums[lo..mid]和nums[mid+1..hi]递归排序完成之后，我们没有办法原地把它俩合并，所以需要 copy 到temp数组里面，然后通过类似于前文 单链表的六大技巧 中合并有序链表的双指针技巧将nums[lo..hi]合并成一个有序数组：



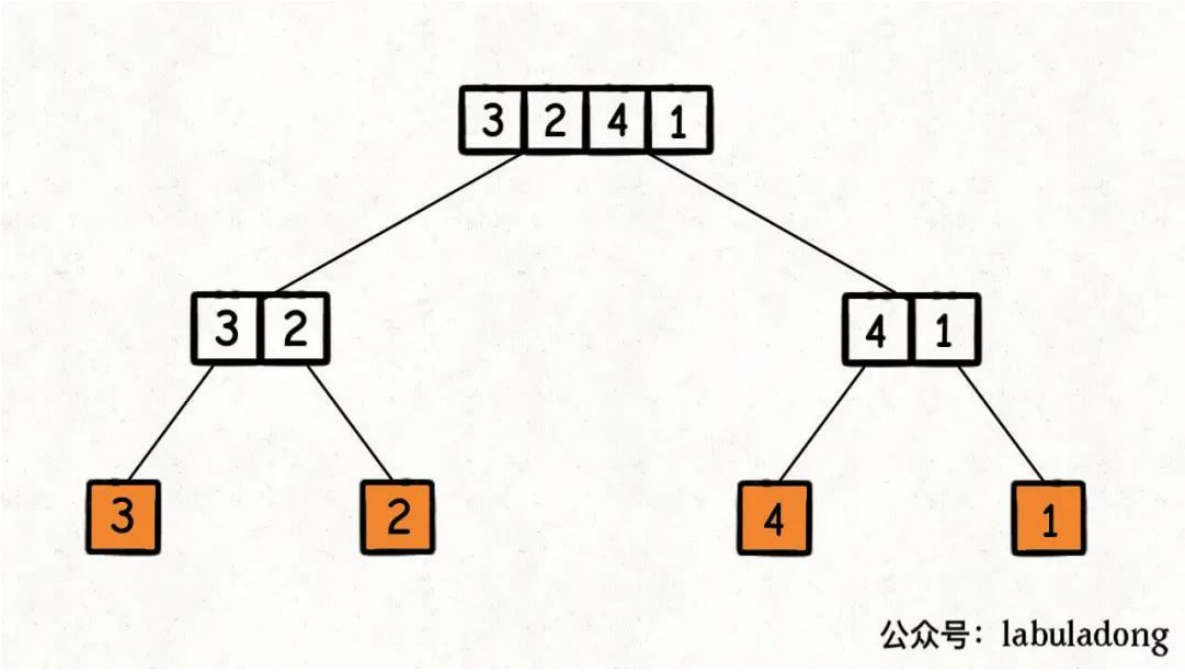
注意我们不是在merge函数执行的时候 new 辅助数组，而是提前把temp辅助数组 new 出来了，这样就避免了在递归中频繁分配和释放内存可能产生的性能问题。

再说一下归并排序的时间复杂度，虽然大伙儿应该都知道是O(NlogN)，但不见得所有人都知道这个复杂度怎么算出来的。

前文 动态规划详解 说过递归算法的复杂度计算，就是子问题个数 x 解决一个子问题的复杂度。

对于归并排序来说，时间复杂度显然集中在merge函数遍历nums[lo..hi]的过程，但每次merge输入的lo和hi都不同，所以不容易直观地看出时间复杂度。

merge函数到底执行了多少次？每次执行的时间复杂度是多少？总的时间复杂度是多少？这就要结合之前画的这幅图来看：



执行的次数是二叉树节点的个数，每次执行的复杂度就是每个节点代表的子数组的长度，所以总的时间复杂度就是整棵树中「数组元素」的个数。

所以从整体上看，这个二叉树的高度是logN，其中每一层的元素个

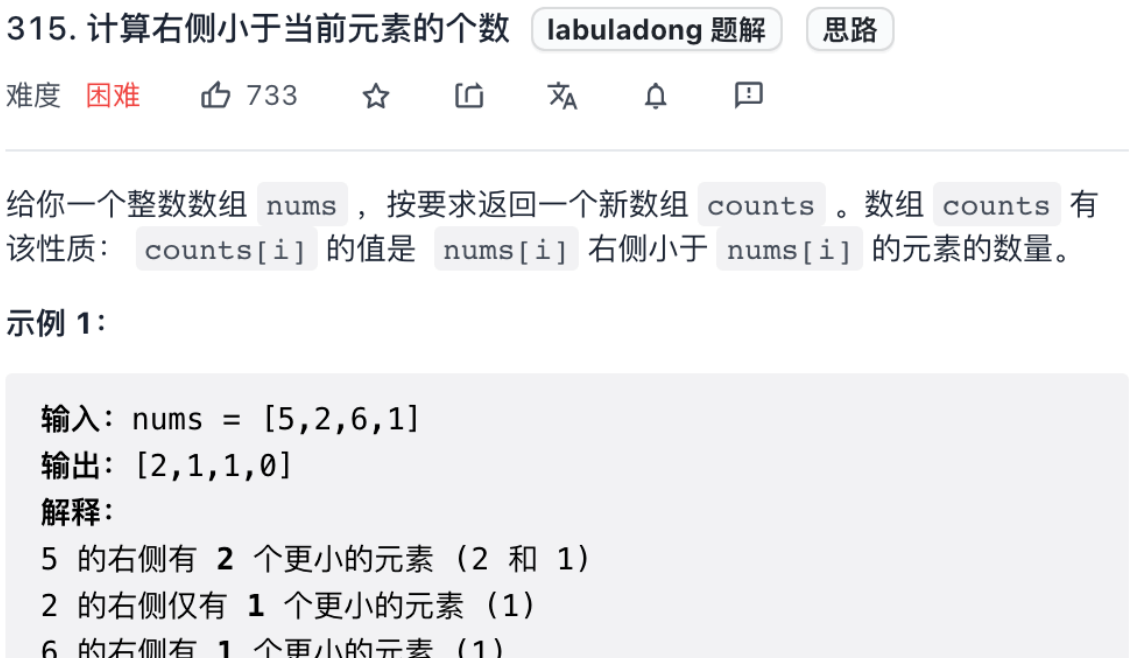
数就是原数组的长度N，所以总的时间复杂度就是O(NlogN)。

力扣第 912 题「排序数组」就是让你对数组进行排序，我们可以直接套用归并排序代码模板：

```
class Solution {
    public int[] sortArray(int[] nums) {
        // 归并排序对数组进行原地排序
        Merge.sort(nums);
        return nums;
    }
}

class Merge {
    // 见上文
}
```

其他应用

除了最基本的排序问题，归并排序还可以用来解决力扣第 315 题「计算右侧小于当前元素的个数」：



315. 计算右侧小于当前元素的个数 labuladong 题解 思路

难度 困难 733

给你一个整数数组 `nums` ，按要求返回一个新数组 `counts` 。数组 `counts` 有该性质： `counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

示例 1：

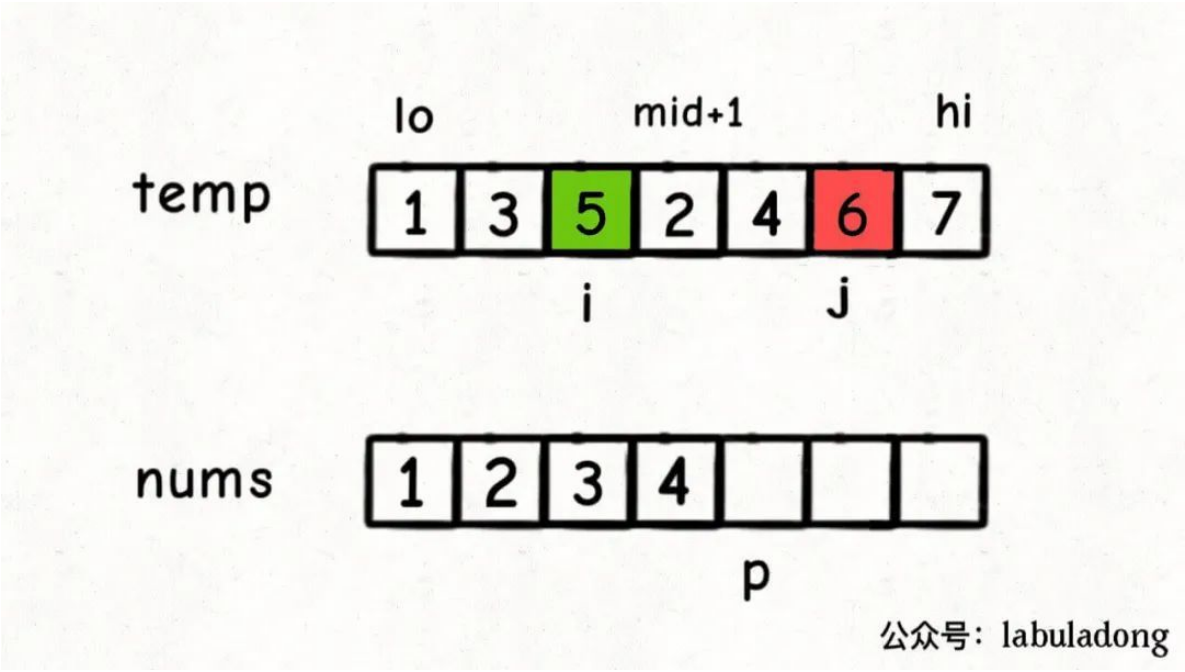
```
输入：nums = [5,2,6,1]
输出：[2,1,1,0]
解释：
5 的右侧有 2 个更小的元素 (2 和 1)
2 的右侧仅有 1 个更小的元素 (1)
6 的右侧有 1 个更小的元素 (1)
```

```
1 的右侧有 0 个更小的元素
```

拍脑袋的暴力解法就不说了，嵌套 for 循环，平方级别的复杂度。

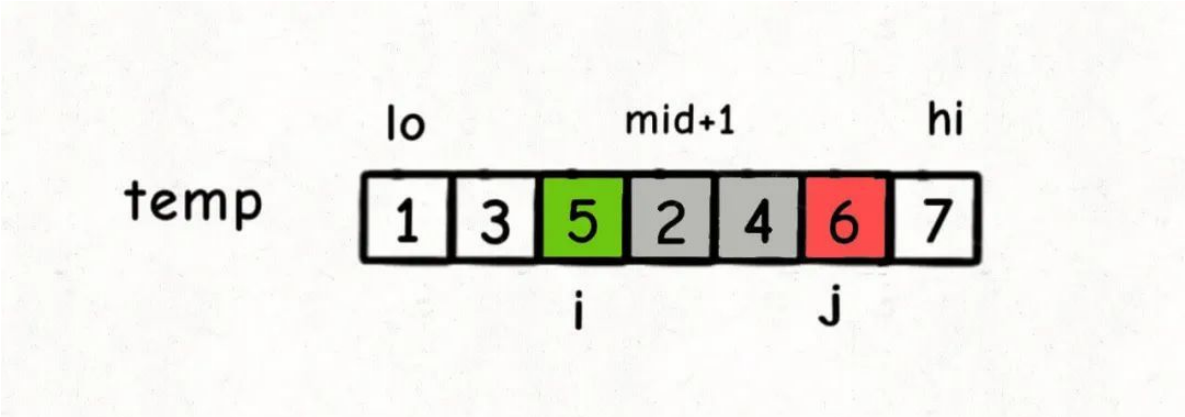
这题和归并排序什么关系呢，主要在merge函数，我们在合并两个有序数组的时候，其实是可以知道一个数字x后边有多少个数字比x小的。

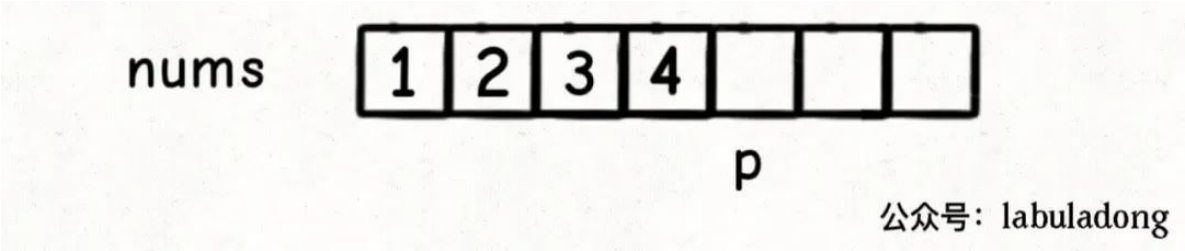
具体来说，比如这个场景：



这时候我们应该把temp[i]放到nums[p]上，因为temp[i] < temp[j]。

但就在这个场景下，我们还可以知道一个信息：5 后面比 5 小的元素个数就是j和mid + 1之间的元素个数，即 2 个。





换句话说，在对nuns[lo..hi]合并的过程中，每当执行nums[p] = temp[i]时，就可以确定temp[i]这个元素后面比它小的元素个数为j - mid - 1。

当然，nums[lo..hi]本身也只是一个子数组，这个子数组之后还会被执行merge，其中元素的位置还是会改变。但这是其他递归节点需要考虑的问题，我们只要在merge函数中做一些手脚，就可以让每个递归节点叠加每次merge时记录的结果。

发现了这个规律后，我们只要在merge中添加两行代码即可解决这个问题，看解法代码：

```
class Solution {
    private class Pair {
        int val, id;
        Pair(int val, int id) {
            // 记录数组的元素值
            this.val = val;
            // 记录元素在数组中的原始索引
            this.id = id;
        }
    }

    // 归并排序所用的辅助数组
    private Pair[] temp;
    // 记录每个元素后面比自己小的元素个数
    private int[] count;
```

```
    // 主函数
    public List<Integer> countSmaller(int[] nums) {
        int n = nums.length;
        count = new int[n];
        temp = new Pair[n];
        Pair[] arr = new Pair[n];
        // 记录元素原始的索引位置，以便在 count 数组中更新结果
        for (int i = 0; i < n; i++)
            arr[i] = new Pair(nums[i], i);

        // 执行归并排序，本题结果被记录在 count 数组中
        sort(arr, 0, n - 1);

        List<Integer> res = new LinkedList<>();
        for (int c : count) res.add(c);
        return res;
    }

    // 归并排序
    private void sort(Pair[] arr, int lo, int hi) {
        if (lo == hi) return;
        int mid = lo + (hi - lo) / 2;
        sort(arr, lo, mid);
        sort(arr, mid + 1, hi);
        merge(arr, lo, mid, hi);
    }

    // 合并两个有序数组
```

```
    private void merge(Pair[] arr, int lo, int mid,
int hi) {
        for (int i = lo; i <= hi; i++) {
            temp[i] = arr[i];
        }

        int i = lo, j = mid + 1;
        for (int p = lo; p <= hi; p++) {
            if (i == mid + 1) {
                arr[p] = temp[j++];
            } else if (j == hi + 1) {
                arr[p] = temp[i++];
                // 更新 count 数组
                count[arr[p].id] += j - mid - 1;
            } else if (temp[i].val > temp[j].val) {
                arr[p] = temp[j++];
            } else {
                arr[p] = temp[i++];
                // 更新 count 数组
                count[arr[p].id] += j - mid - 1;
            }
        }
    }
}
```

因为在排序过程中，每个元素的索引位置会不断改变，所以我们用一个Pair类封装每个元素及其在原始数组nums中的索引，以便count数组记录每个元素之后小于它的元素个数。

你现在回头体会下我在本文开头说那句话：

所有递归的算法，本质上都是在遍历一棵（递归）树，然后在节点

（前中后序位置）上执行代码。你要写递归算法，本质上就是要告诉每个节点需要做什么。

有没有品出点味道?

最后总结一下吧，本文从二叉树的角度讲了归并排序的核心思路和代码实现，同时讲了一道归并排序相关的算法题。这道算法题其实就是归并排序算法逻辑中夹杂一点私货，但仍然属于比较难的，你可能需要亲自做一遍才能理解。

那我最后留一个思考题吧，下一篇文章我会讲快速排序，你是否能够尝试着从二叉树的角度去理解快速排序？如果让你用一句话总结快速排序的逻辑，你怎么描述?

好了，答案下篇文章揭晓。

文章分享自微信公众号: