快速排序的正确理解方式及运用

912. 排序数组（Medium）

215. 数组中的第 K 个最大元素（Medium）

前文 归并排序算法详解 通过二叉树的视角描述了归并排序的算法原理以及应用，很多读者大呼精妙，那我就趁热打铁，今天继续用二叉树的视角讲一讲快速排序算法的原理以及运用。

快速排序算法思路

首先我们看一下快速排序的代码框架：

```
void sort(int[] nums, int lo, int hi) {
    if (lo >= hi) {
        return;
    }
    // 对 nums[lo..hi] 进行切分
    // 使得 nums[lo..p-1] <= nums[p] < nums[p+1..hi]
    int p = partition(nums, lo, hi);
    // 去左右子数组进行切分
    sort(nums, lo, p - 1);
    sort(nums, p + 1, hi);
}
```

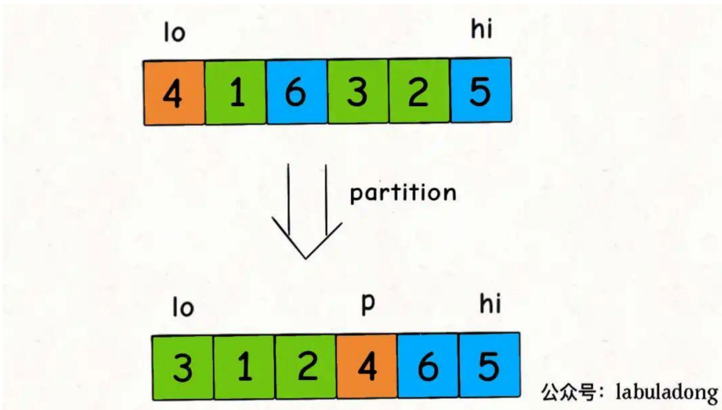
其实你对比之后可以发现，快速排序就是一个二叉树的前序遍历：

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

另外，前文 归并排序详解 用一句话总结了归并排序：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。

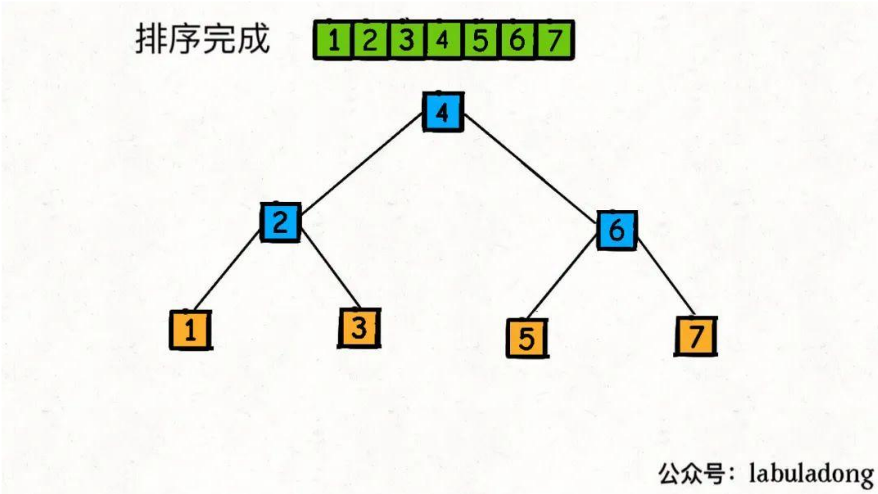
同时我提了一个问题，让你一句话总结快速排序，这里说一下我的答案：
快速排序是先将一个元素排好序，然后再将剩下的元素排好序。

快速排序的核心无疑是 **partition** 函数，**partition** 函数的作用是在 **nums[lo..hi]** 中寻找一个分界点 **p**，通过交换元素使得 **nums[lo..p-1]** 都小于等于 **nums[p]**，且 **nums[p+1..hi]** 都大于 **nums[p]**：



一个元素左边的元素都比它小，右边的元素都比它大，啥意思？不就是它自己已经被放到正确的位置上了吗？
所以 **partition** 函数干的事情，其实就是把 **nums[p]** 这个元素排好序了。
一个元素被排好序了，然后呢？你再把剩下的元素排好序不就得了。
剩下的元素有哪些？左边一坨，右边一坨，去吧，对子数组进行递归，用 partition 函数把剩下的元素也排好序。
从二叉树的视角，我们可以把子数组 nums[lo..hi] 理解成二叉树节点上的值，srot 函数理解成二叉树的遍历函数。
参照二叉树的前序遍历顺序，快速排序的运行过程如下 GIF：

你注意最后形成的这棵二叉树是什么？是一棵二叉搜索树：



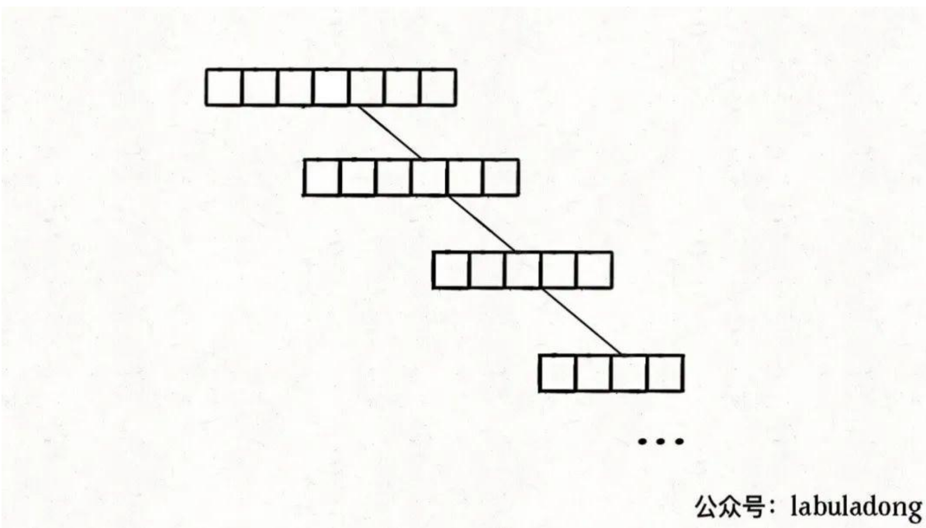
这应该不难理解吧，因为 partition 函数每次都将数组切分成左小右大两部分，恰好和二叉搜索树左小右大的特性吻合。

你甚至可以这样理解：快速排序的过程是一个构造二叉搜索树的过程。

但谈到二叉搜索树的构造，那就不得不说二叉搜索树不平衡的极端情况，极端情况下二叉搜索树会退化成一个链表，导致操作效率大幅降低。

快速排序的过程中也有类似的情况，比如我画的图中每次 partition 函数选出的分界点都能把 nums[lo..hi] 平分成两半，但现实中你不见得运气这么好。

如果你每次运气都特别背，有一边的元素特别少的话，这样会导致二叉树生长不平衡：



这样的话，时间复杂度会大幅上升，后面分析时间复杂度的时候再细说。

我们为了避免出现这种极端情况，需要引入随机性。

常见的方式是在进行排序之前对整个数组执行 [洗牌算法] 进行打乱，或者在 partition函数中随机选择数组元素作为分界点，本文会使用前者。

快速排序代码实现

明白了上述概念，直接看快速排序的代码实现：

```java
class Quick {

    public static void sort(int[] nums) {
        // 为了避免出现耗时的极端情况，先随机打乱
        shuffle(nums);
        // 排序整个数组（原地修改）
        sort(nums, 0, nums.length - 1);
    }

    private static void sort(int[] nums, int lo, int hi) {
        if (lo >= hi) {
            return;
        }
        // 对 nums[lo..hi] 进行切分
        // 使得 nums[lo..p-1] <= nums[p] < nums[p+1..hi]
        int p = partition(nums, lo, hi);

        sort(nums, lo, p - 1);
        sort(nums, p + 1, hi);
    }

    // 对 nums[lo..hi] 进行切分
    private static int partition(int[] nums, int lo, int hi) {
        int pivot = nums[lo];
        // 关于区间的边界控制需格外小心，稍有不慎就会出错
        // 我这里把 i, j 定义为开区间，同时定义：
        // [lo, i) <= pivot；(j, hi] > pivot
        // 之后都要正确维护这个边界区间的定义
        int i = lo + 1, j = hi;
        // 当 i > j 时结束循环，以保证区间 [lo, hi] 都被覆盖
        while (i <= j) {
            while (i < hi && nums[i] <= pivot) {
                i++;
                // 此 while 结束时恰好 nums[i] > pivot
            }
            while (j > lo && nums[j] > pivot) {
                j--;
                // 此 while 结束时恰好 nums[j] <= pivot
            }
            // 此时 [lo, i) <= pivot && (j, hi] > pivot

            if (i >= j) {
```

```java
                break;
            }
            swap(nums, i, j);
        }
        // 将 pivot 放到合适的位置，即 pivot 左边元素较小，右边元素较大
        swap(nums, lo, j);
        return j;
    }

    // 洗牌算法，将输入的数组随机打乱
    private static void shuffle(int[] nums) {
        Random rand = new Random();
        int n = nums.length;
        for (int i = 0 ; i < n; i++) {
            // 生成 [i, n - 1] 的随机数
            int r = i + rand.nextInt(n - i);
            swap(nums, i, r);
        }
    }

    // 原地交换数组中的两个元素
    private static void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

这里啰嗦一下核心函数 partition 的实现，正如前文 二分搜索框架详解 所说，想要正确寻找切分点非常考验你对边界条件的控制，稍有差错就会产生错误的结果。

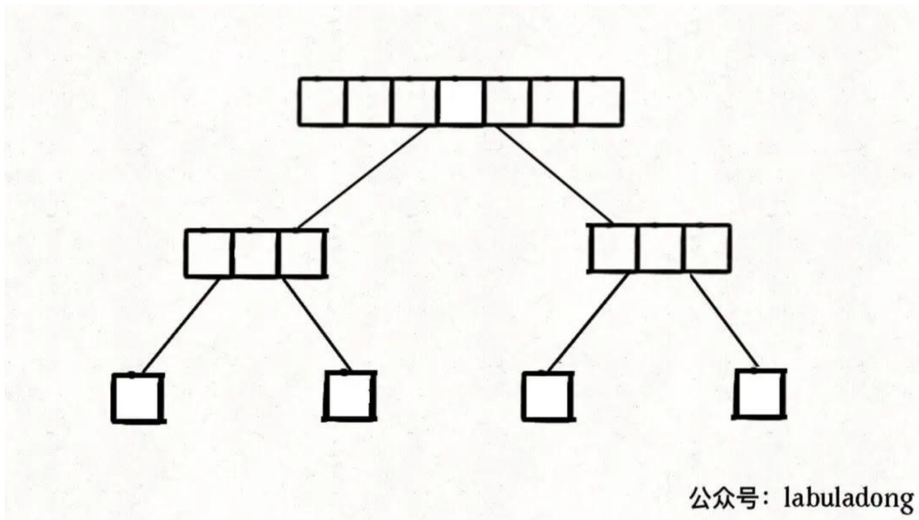
处理边界细节的一个技巧就是，你要明确每个变量的定义以及区间的开闭情况。具体的细节看代码注释，建议自己动手实践。

接下来分析一下快速排序的时间复杂度。

显然，快速排序的时间复杂度主要消耗在 partition 函数上，因为这个函数中存在循环。

所以 partition 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？

和归并排序类似，需要结合之前画的这幅图来从整体上分析：

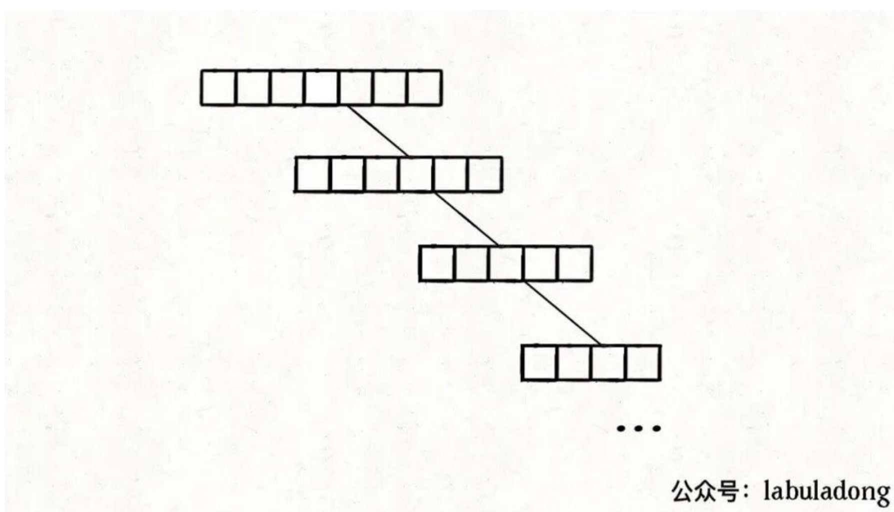


partition 执行的次数是二叉树节点的个数，每次执行的复杂度就是每个节点代表的子数组 nums[lo..hi] 的长度，所以总的时间复杂度就是整棵树中「数组元素」的个数。

假设数组元素个数为 N，那么二叉树每一层的元素个数之和就是 O(N)；分界点分布均匀的理想情况下，树的层数为 O(logN)，所以理想的总时间复杂度为 O(NlogN)。

由于快速排序没有使用任何辅助数组，所以空间复杂度就是递归堆栈的深度，也就是树高 O(logN)。

当然，我们之前说过快速排序的效率存在一定随机性，如果每次 partition 切分的结果都极不均匀：



快速排序就退化成选择排序了，树高为 O(N)，每层节点的元素个数从 N 开始递减，总的时间复杂度为：

N + (N - 1) + (N - 2) + ... + 1 = O(N^2)

所以我们说，快速排序理想情况的时间复杂度是 O(NlogN)，空间复杂度 O(logN)，极端情况下的最坏时间复杂度是 O(N^2)，空间复杂度是 O(N)。

不过大家放心，经过随机化的 partition 函数很难出现极端情况，所以快速排序的效率还是非常高的。

还有一点需要注意的是，快速排序是「不稳定排序」，与之相对的，前文讲的[归并排序](#) 是「稳定排序」。

对于序列中的相同元素，如果排序之后它们的相对位置没有发生改变，则称该排序算法为「稳定排序」, 反之则为「不稳定排序」。
如果单单排序 int 数组，那么稳定性没有什么意义。但如果排序一些结构比较复杂的数据，那么稳定性排序就有更大的优势了。
比如说你有若干订单数据，已经按照订单号排好序了，现在你想对订单的交易日期再进行排序：
如果用稳定排序算法（比如归并排序），那么这些订单不仅按照交易日期排好了序，而且相同交易日期的订单的订单号依然是有序的。
但如果你用不稳定排序算法（比如快速排序），那么虽然排序结果会按照交易日期排好序，但相同交易日期的订单的订单号会丧失有序性。
在实际工程中我们经常会将一个复杂对象的某一个字段作为排序的 key，所以应该关注编程语言提供的 API 底层使用的到底是什么排序算法，是稳定的还是不稳定的，这很可能影响到代码执行的效率甚至正确性。
说了这么多，快速排序算法应该算是讲明白了，力扣第 912 题「排序数组」就是让你对数组进行排序, 我们可以直接套用快速排序的代码模板：

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

class Quick {
    // 见上文
}
```

快速选择算法

不仅快速排序算法本身很有意思，而且它还有一些有趣的变体，最有名的就是快速选择算法（Quick Select)。
力扣第 215 题「数组中的第 K 个最大元素」就是一道类似的题目，函数签名如下：
int findKthLargest(int[] nums, int k);
题目要求我们寻找第 k 个最大的元素，稍微有点绕，意思是去寻找 nums 数组降序排列后排名第 k 的那个元素。
比如输入 nums = [2,1,5,4], k = 2，算法应该返回 4，因为 4 是 nums 中第 2 个最大的元素。
这种问题有两种解法，一种是二叉堆（优先队列）的解法，另一种就是快速选择算法，我们分别来看。
二叉堆的解法比较简单，但时间复杂度稍高，直接看代码好了：

```
int findKthLargest(int[] nums, int k) {
    // 小顶堆，堆顶是最小元素
    PriorityQueue<Integer>
        pq = new PriorityQueue<>();
```

```java
    for (int e : nums) {
        // 每个元素都要过一遍二叉堆
        pq.offer(e);
        // 堆中元素多于 k 个时，删除堆顶元素
        if (pq.size() > k) {
            pq.poll();
        }
    }
    // pq 中剩下的是 nums 中 k 个最大元素,
    // 堆顶是最小的那个，即第 k 个最大元素
    return pq.peek();
}
```

二叉堆（优先队列）是一种能够自动排序的数据结构，我们前文[手把手实现二叉堆数据结构](#) 实现过这种结构，我就默认大家熟悉它的特性了。

核心思路就是把小顶堆 pq 理解成一个筛子，较大的元素会沉淀下去，较小的元素会浮上来；当堆大小超过 k 的时候，我们就删掉堆顶的元素，因为这些元素比较小，而我们想要的是前 k 个最大元素嘛。

当 nums 中的所有元素都过了一遍之后，筛子里面留下的就是最大的 k 个元素，而堆顶元素是堆中最小的元素，也就是「第 k 个最大的元素」。

思路很简单吧，唯一注意的是，Java 的 PriorityQueue 默认实现是小顶堆，有的语言的优先队列可能默认是大顶堆，可能需要做一些调整。

二叉堆插入和删除的时间复杂度和堆中的元素个数有关，在这里我们堆的大小不会超过 k，所以插入和删除元素的复杂度是 O(logk)，再套一层 for 循环，假设数组元素总数为 N，总的时间复杂度就是 O(Nlogk)。

这个解法的空间复杂度很显然就是二叉堆的大小，为 O(k)。

快速选择算法是快速排序的变体，效率更高，面试中如果能够写出快速选择算法，肯定是加分项。

首先，题目问「第 k 个最大的元素」，相当于数组升序排序后「排名第 n - k 的元素」，为了方便表述，后文另 k' = n - k。

如何知道「排名第 k' 的元素」呢？其实在快速排序算法 partition 函数执行的过程中就可以略见一二。

我们刚说了，partition 函数会将 nums[p] 排到正确的位置，使得 nums[lo..p-1] < nums[p] < nums[p+1..hi]：

这时候，虽然还没有把整个数组排好序，但我们已经让 nums[p] 左边的元素都比 nums[p] 小了，也就知道 nums[p] 的排名了。

那么我们可以把 p 和 k' 进行比较，如果 p < k' 说明第 k' 大的元素在 nums[p+1..hi] 中，如果 p > k' 说明第 k' 大的元素在 nums[lo..p-1] 中。

进一步，去 nums[p+1..hi] 或者 nums[lo..p-1] 这两个子数组中执行 partition 函数，就可以进一步缩小排在第 k' 的元素的范围，最终找到目标元素。

这样就可以写出解法代码：

```java
int findKthLargest(int[] nums, int k) {
```

```java
    // 首先随机打乱数组
    shuffle(nums);
    int lo = 0, hi = nums.length - 1;
    // 转化成「排名第 k 的元素」
    k = nums.length - k;
    while (lo <= hi) {
        // 在 nums[lo..hi] 中选一个分界点
        int p = partition(nums, lo, hi);
        if (p < k) {
            // 第 k 大的元素在 nums[p+1..hi] 中
            lo = p + 1;
        } else if (p > k) {
            // 第 k 大的元素在 nums[lo..p-1] 中
            hi = p - 1;
        } else {
            // 找到第 k 大元素
            return nums[p];
        }
    }
    return -1;
}


// 对 nums[lo..hi] 进行切分
int partition(int[] nums, int lo, int hi) {
    // 见前文
}

// 洗牌算法，将输入的数组随机打乱
void shuffle(int[] nums) {
    // 见前文
}

// 原地交换数组中的两个元素
void swap(int[] nums, int i, int j) {
    // 见前文
}
```

这个代码框架其实非常像我们前文 二分搜索框架 的代码，这也是这个算法高效的原因，但是时间复杂度为什么是 O(N) 呢？
显然，这个算法的时间复杂度也主要集中在 partition 函数上，我们需要估算 partition 函数执行了多少次，每次执行的时间复杂度是多少。

最好情况下，每次 partition 函数切分出的 p 都恰好是正中间索引 (lo + hi) / 2（二分），且每次切分之后会到左边或者右边的子数组继续进行切分，那么 partition 函数执行的次数是 logN，每次输入的数组大小缩短一半。
所以总的时间复杂度为：
等差数列
N + N/2 + N/4 + N/8 + ... + 1 = 2N = O(N)
当然，类似快速排序，快速选择算法中的 partition 函数也可能出现极端情况，最坏情况下 p 一直都是 lo + 1 或者一直都是 hi - 1，这样的话时间复杂度就退化为 O(N^2) 了：
N + (N - 1) + (N - 2) + ... + 1 = O(N^2)
这也是我们在代码中使用 shuffle 函数的原因，通过引入随机性来避免极端情况的出现，让算法的效率保持在比较高的水平。随机化之后的快速选择算法的复杂度可以认为是 O(N)。
到这里，快速排序算法和快速选择算法就讲完了，从二叉树的视角来理解思路应该是不难的，但 partition 函数对细节的把控需要你多花心思去理解和记忆。
最后留一个问题吧，比较一下快速排序和前文讲的 归并排序 并且可以说说你的理解：为什么快速排序是不稳定排序，而归并排序是稳定排序呢？