0215. 数组中的第K个最大元素

ITCharge 大约 6 分钟

- 标签：数组、分治、快速排序、排序、堆（优先队列）
- 难度：中等

题目链接

- 0215. 数组中的第K个最大元素 - 力扣

题目大意

描述：给定一个未排序的整数数组 $nums$ 和一个整数 k。

要求：返回数组中第 k 个最大的元素。

说明：

- 要求使用时间复杂度为 $O(n)$ 的算　　决此问题。
- $1 \leq k \leq nums.length \leq 10^5$。
- $-10^4 \leq nums[i] \leq 10^4$。

示例：

- 示例 1：

```py
输入：[3,2,1,5,6,4], k = 2
输出：5
```

- 示例 2：

```py
输入：[3,2,3,1,2,4,5,5,6], k = 4
输出：4
```

解题思路

很不错的一道题，面试常考。

直接可以想到的思路是：排序后输出数组上对应第 k 位大的数。所以问题关键在于排序方法的复杂度。

冒泡排序、选择排序、插入排序时间复杂度 $O(n^2)$ 太高了，很容易超时。

可考虑堆排序、归并排序、快速排序。

这道题的要求是找到第 k 大的元素，使用归并排序只有到最后排序完毕才能返回第 k 大的数。而堆排序每次排序之后，就会确定一个元素的准确排名，同理快速排序也是如此。

思路 1：堆排序

升序堆排序的思路如下：

1. 将无序序列构造成第 1 个大顶堆（初始堆），使得 n 个元素的最大值处于序列的第 1 个位置。
2. **调整堆**：交换序列的第 1 个元素（最大值元素）与第 n 个元素的位置。将序列前 $n-1$ 个元素组成的子序列调整成一个新的大顶堆，使得 $n-1$ 个元素的最大值处于序列第 1 个位置，从而得到第 2 个最大值元素。
3. **调整堆**：交换子序列的第 1 个元素（最大值元素）与第 $n-1$ 个元素的位置。将序列前 $n-2$ 个元素组成的子序列调整成一个新的大顶堆，使得 $n-2$ 个元素的最大值处于序列第 1 个位置，从而得到第 3 个最大值元素。
4. 依次类推，不断交换子序列的第 1 个元素（最大值元素）与当前子序列最后一个元素位置，并将其调整成新的大顶堆。直到获取第 k 个最大值元素为止。

思路 1：代码

```py
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # 调整为大顶堆
        def heapify(nums, index, end):
            left = index * 2 + 1
            right = left + 1
            while left <= end:
                # 当前节点为非叶子节点
                max_index = index
                if nums[left] > nums[max_index]:
                    max_index = left
                if right <= end and nums[right] > nums[max_index]:
                    max_index = right
                if index == max_index:
                    # 如果不用交换，则说明已经交换结束
                    break
                nums[index], nums[max_index] = nums[max_index], nums[index]
                # 继续调整子树
                index = max_index
                left = index * 2 + 1
                right = left + 1

        # 初始化大顶堆
        def buildMaxHeap(nums):
            size = len(nums)
            # (size-2) // 2 是最后一个非叶节点，叶节点不用调整
            for i in range((size - 2) // 2, -1, -1):
                heapify(nums, i, size - 1)
            return nums

        buildMaxHeap(nums)
        size = len(nums)
        for i in range(k-1):
            nums[0], nums[size-i-1] = nums[size-i-1], nums[0]
            heapify(nums, 0, size-i-2)
        return nums[0]
```

思路 1：复杂度分析

- **时间复杂度**：$O(n \times \log n)$。
- **空间复杂度**：$O(1)$。

思路 2：快速排序

使用快速排序在每次调整时，都会确定一个元素的最终位置，且以该元素为界限，将数组分成了左右两个子数组，左子数组中的元素都比该元素小，右子树组中的元素都比该元素大。

这样，只要某次划分的元素恰好是第 k 个下标就找到了答案。并且我们只需关注第 k 个最大元素所在区间的排序情况，与第 k 个最大元素无关的区间排序都可以忽略。这样进一步减少了执行步骤。

思路 2：代码

```py
import random

class Solution:
    # 随机哨兵划分：从 nums[low: high + 1] 中随机挑选一个基准数，并进行移位排序
    def randomPartition(self, nums: [int], low: int, high: int) -> int:
        # 随机挑选一个基准数
        i = random.randint(low, high)
        # 将基准数与最低位互换
        nums[i], nums[low] = nums[low], nums[i]
        # 以最低位为基准数，然后将数组中比基准数大的元素移动到基准数右侧，比他小的元素移动到基准数左侧。最后将基准数放到正确位置上
        return self.partition(nums, low, high)

    # 哨兵划分：以第 1 位元素 nums[low] 为基准数，然后将比基准数小的元素移动到基准数左侧，将比基准数大的元素移动到基准数右侧，最后将基准数放到正确位置上
    def partition(self, nums: [int], low: int, high: int) -> int:
        # 以第 1 位元素为基准数
        pivot = nums[low]

        i, j = low, high
        while i < j:
```

```
            # 从右向左找到第 1 个小于基准数的元素
            while i < j and nums[j] >= pivot:
                j -= 1
            # 从左向右找到第 1 个大于基准数的元素
            while i < j and nums[i] <= pivot:
                i += 1
            # 交换元素
            nums[i], nums[j] = nums[j], nums[i]

        # 将基准数放到正确位置上
        nums[j], nums[low] = nums[low], nums[j]
        return j

    def quickSort(self, nums: [int], low: int, high: int, k: int, size: int) ->
[int]:
        if low < high:
            # 按照基准数的位置，将数组划分为左右两个子数组
            pivot_i = self.randomPartition(nums, low, high)
            if pivot_i == size - k:
                return nums[size - k]
            if pivot_i > size - k:
                self.quickSort(nums, low, pivot_i - 1, k, size)
            if pivot_i < size - k:
                self.quickSort(n     pivot_i + 1, high, k, size)

        return nums[size - k]


    def findKthLargest(self, nums: List[int], k: int) -> int:
        size = len(nums)
        return self.quickSort(nums, 0, len(nums) - 1, k, size)
```

思路 2：复杂度分析

- **时间复杂度**：$O(n)$。证明过程可参考「算法导论 9.2：期望为线性的选择算法」。
- **空间复杂度**：$O(\log n)$。递归使用栈空间的空间代价期望为 $O(\log n)$。

思路 3：借用标准库（不建议）

提交代码中的最快代码是调用了 Python 的 `sort` 方法。这种做法适合在打算法竞赛的时候节省时间，日常练习可以尝试一下自己写。

思路 3：代码

```py
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        nums.sort()
        return nums[len(nums) - k]
```

思路 3：复杂度分析

- **时间复杂度**：$O(n \times \log n)$。
- **空间复杂度**：$O(1)$。

思路 4：优先队列

1. 遍历数组元素，对于挡圈元素 num：
    1. 如果优先队列中的元素个数小于 k 个，则将当前元素 num 放入优先队列中。
    2. 如果优先队列中的元素个数大于等于 k 个，并且当前元素 num 大于优先队列的队头元素，则弹出队头元素，并将当前元素 num 插入到优先队列中。
2. 遍历完，此时优先队列的队头元素就是第 k 个最大元素，将其弹出并返回即可。

这里我们借助了 Python 中的 `heapq` 模块实现优先队列算法，这一步也可以通过手写堆的方式实现优先队列。

思路 4：代码

```py
import heapq
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        res = []
        for num in nums:
            if len(res) < k:
```

```
                heapq.heappush(res, num)
            elif num > res[0]:
                heapq.heappop(res)
                heapq.heappush(res, num)
        return heapq.heappop(res)
```

思路 4：复杂度分析

- **时间复杂度**：$O(n \times \log k)$。
- **空间复杂度**：$O(k)$。