Top K 的两种经典解法（堆/快排变形）与优劣比较

https://leetcode-cn.com/problems/zui-xiao-de-kge-shu-lcof/solution/tu-jie-top-k-wen-ti-de-liang-chong-jie-fa-you-lie-/

Top K 问题有两种不同的解法，

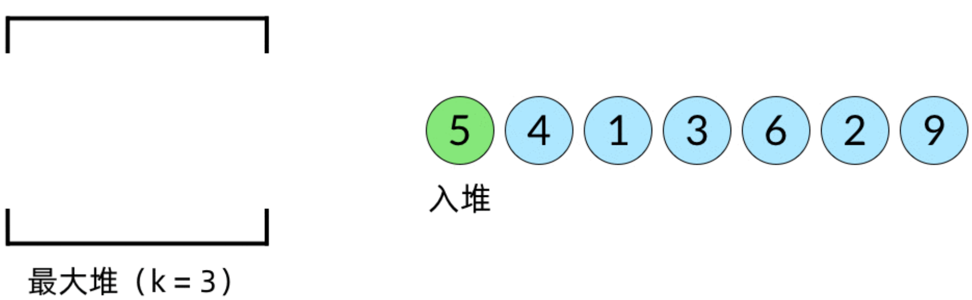
1. 一种解法使用堆（优先队列），
2. 另一种解法使用类似快速排序的分治法。

这两种方法各有优劣，最好都掌握。本文用图解的形式讲解这道问题的两种解法，包括三个部分：
方法一：堆，时间复杂度 O(nlogk)
方法二：快排变形，（平均）时间复杂度 O(n)

两种方法的优劣比较

方法一：堆

比较直观的想法是使用堆数据结构来辅助得到最小的 k 个数。堆的性质是每次可以找出最大或最小的元素。我们可以使用一个大小为 k 的最大堆（大顶堆），将数组中的元素依次入堆，当堆的大小超过 k 时，便将多出的元素从堆顶弹出。我们以数组
[5, 4, 1, 3, 6, 2, 9]
k=3 为例展示元素入堆的过程，如下面动图所示：



这样，由于每次从堆顶弹出的数都是堆中最大的，最小的 **k** 个元素一定会留在堆里。这样，把数组中的元素全部入堆之后，堆中剩下的 **k** 个元素就是最小的 **k** 个数了。
注意在动画中，我们并没有画出堆的内部结构，因为这部分内容并不重要。我们只需要知道堆每次会弹出最大的元素即可。在写代码的时候，我们使用的也是库函数中的优先队列数据结构，如

Java 中的 PriorityQueue。在面试中，我们不需要实现堆的内部结构，把数据结构使用好，会分析其复杂度即可。
以下是题解代码。感谢评论区提醒，这里的代码可以做一些优化，如果当前数字不小于堆顶元素，数字可以直接丢掉，不入堆。下方的代码已更新：

```
public int[] getLeastNumbers(int[] arr, int k) {
    if (k == 0) {
        return new int[0];
    }
    // 使用一个最大堆（大顶堆）
    // Java 的 PriorityQueue 默认是小顶堆，添加 comparator 参数使其变成最大堆
    Queue<Integer> heap = new PriorityQueue<>(k, (i1, i2) ->
Integer.compare(i2, i1));

    for (int e : arr) {
        // 当前数字小于堆顶元素才会入堆
        if (heap.isEmpty() || heap.size() < k || e < heap.peek()) {
            heap.offer(e);
        }
        if (heap.size() > k) {
            heap.poll(); // 删除堆顶最大元素
        }
    }

    // 将堆中的元素存入数组
    int[] res = new int[heap.size()];
    int j = 0;
    for (int e : heap) {
        res[j++] = e;
    }
    return res;
}
```

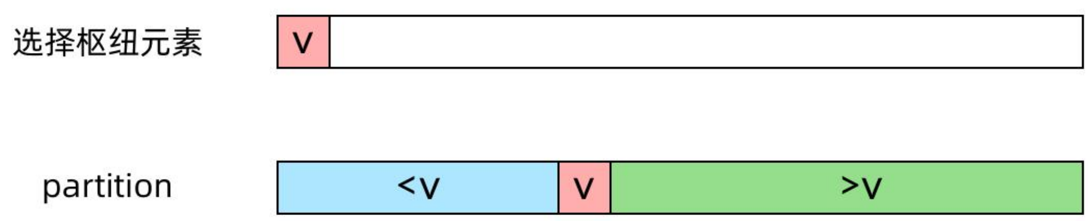
算法的复杂度分析：
由于使用了一个大小为 k 的堆，空间复杂度为 O(k)；
入堆和出堆操作的时间复杂度均为 O(logk)，每个元素都需要进行一次入堆操作，故算法的时间复杂度为 O(nlogk)。

方法二：快排变形

Top K 问题的另一个解法就比较难想到，需要在平时有算法的积累。实际上，“查找第 k 大的元素”是一类算法问题，称为选择问题。找第 k 大的数，或者找前 k 大的数，有一个经典的 quick select

（快速选择）算法。这个名字和 quick sort（快速排序）看起来很像算法的思想也和快速排序类似，都是分治法的思想。

让我们回顾快速排序的思路。快速排序中有一步很重要的操作是 partition（划分），从数组中随机选取一个枢纽元素 v，然后原地移动数组中的元素，使得比 v 小的元素在 v 的左边，比 v 大的元素在 v 的右边，如下图所示：



这个 partition 操作是原地进行的，需要 O(n) 的时间，接下来，快速排序会递归地排序左右两侧的数组。而快速选择（quick select）算法的不同之处在于，接下来只需要递归地选择一侧的数组。快速选择算法想当于一个“不完全”的快速排序，因为我们只需要知道最小的 k 个数是哪些，并不需要知道它们的顺序。

我们的目的是寻找最小的 k 个数。假设经过一次 partition 操作，枢纽元素位于下标 m，也就是说，左侧的数组有 m 个元素，是原数组中最小的 m 个数。那么：

- 若 k = m，我们就找到了最小的 k 个数，就是左侧的数组；
- 若 k<m，则最小的 k 个数一定都在左侧数组中，我们只需要对左侧数组递归地 parition 即可；
- 若 k>m，则左侧数组中的 m 个数都属于最小的 k 个数，我们还需要在右侧数组中寻找最小的 k-m 个数，对右侧数组递归地 partition 即可。

这种方法需要多加领会思想，如果你对快速排序掌握得很好，那么稍加推导应该不难掌握 **quick select** 的要领。

以下是题解代码：

```
public int[] getLeastNumbers(int[] arr, int k) {
    if (k == 0) {
        return new int[0];
    } else if (arr.length <= k) {
        return arr;
    }

    // 原地不断划分数组
    partitionArray(arr, 0, arr.length - 1, k);

    // 数组的前 k 个数此时就是最小的 k 个数，将其存入结果
    int[] res = new int[k];
    for (int i = 0; i < k; i++) {
        res[i] = arr[i];
    }
```

```java
    return res;
}

void partitionArray(int[] arr, int lo, int hi, int k) {
    // 做一次 partition 操作
    int m = partition(arr, lo, hi);
    // 此时数组前 m 个数，就是最小的 m 个数
    if (k == m) {
        // 正好找到最小的 k(m) 个数
        return;
    } else if (k < m) {
        // 最小的 k 个数一定在前 m 个数中，递归划分
        partitionArray(arr, lo, m-1, k);
    } else {
        // 在右侧数组中寻找最小的 k-m 个数
        partitionArray(arr, m+1, hi, k);
    }
}

// partition 函数和快速排序中相同，具体可参考快速排序相关的资料
// 代码参考 Sedgewick 的《算法4》
int partition(int[] a, int lo, int hi) {
    int i = lo;
    int j = hi + 1;
    int v = a[lo];
    while (true) {
        while (a[++i] < v) {
            if (i == hi) {
                break;
            }
        }
        while (a[--j] > v) {
            if (j == lo) {
                break;
            }
        }

        if (i >= j) {
            break;
        }
        swap(a, i, j);
    }
```

```
    swap(a, lo, j);
    // a[lo .. j-1] <= a[j] <= a[j+1 .. hi]
    return j;
}

void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

上述代码中需要注意一个细节（评论区有好几个小伙伴问到，这里补充说明一下）：partitionArray 函数中，两次递归调用传入的参数为什么都是 k？特别是第二个调用，我们在右侧数组中寻找最小的 k-m 个数，但是对于整个数组而言，这是最小的 k 个数。所以说，函数调用传入的参数应该为 k。

算法的复杂度分析：

- 空间复杂度 O(1)，不需要额外空间。
- 时间复杂度的分析方法和快速排序类似。由于快速选择只需要递归一边的数组，时间复杂度小于快速排序，期望时间复杂度为 O(n)，最坏情况下的时间复杂度为 O(n^2)

两种方法的优劣性比较

在面试中，另一个常常问的问题就是这两种方法有何优劣。看起来分治法的快速选择算法的时间、空间复杂度都优于使用堆的方法，但是要注意到快速选择算法的几点局限性：
第一，算法需要修改原数组，如果原数组不能修改的话，还需要拷贝一份数组，空间复杂度就上去了。
第二，算法需要保存所有的数据。如果把数据看成输入流的话，使用堆的方法是来一个处理一个，不需要保存数据，只需要保存 **k** 个元素的最大堆。而快速选择的方法需要先保存下来所有的数据，再运行算法。当数据量非常大的时候，甚至内存都放不下的时候，就麻烦了。所以当数据量大的时候还是用基于堆的方法比较好

Top K 问题的最优解 - 快速选择算法（Quickselect）

胖头鱼
在计算机科学中，快速选择算法主要是用于在未排序的数组中找到第 k 个最小/大数字的算法。它的方法和快速排序算法类似，快速排序算法和快速选择选择算法都是由 **Tony Hoare** 发明。

Top K 问题

```
在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
// 示例 1:
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
```

快速排序算法

对于 Top K 问题，最简单的方案是把数组通过快排排序之后直接取对应的 k 值。
JavaScript 示例代码

```
var findKthLargest = (nums, k) => {
  const newNums = quickSort(nums)

  return newNums[newNums.length - k]
}

var quickSort = function(arr) {
  if (arr.length <= 1) {
    return arr;
  }

  var pivotIndex = Math.floor(arr.length / 2);

  var pivot = arr.splice(pivotIndex, 1)[0];

  var left = [];

  var right = [];

  for (var i = 0; i < arr.length; i++) {
    if (arr[i] < pivot) {
      left.push(arr[i]);
    } else {
      right.push(arr[i]);
    }
  }

  return quickSort(left).concat([pivot], quickSort(right));
```

```
};
```

它的时间复杂度是 O(n * log(n))，空间复杂度是 O(log n)

Min-Heap 最小堆

维护一个大小为 k 的最小堆，从头开始遍历数组，扫描到值若大于堆顶则入队，然后删除堆顶。

```
import PriorityQueue from "common/priority-queue";

var findKthLargest = (nums, k) => {
  const pq = new PriorityQueue();

  for (let num of nums) {
    pq.offer(num);

    if (pq.size() > k) {
      pq.poll();
    }
  }

  return pq.peek();
};
```

它的时间复杂度是 O(Nlogk) ，空间复杂度是 O(k)

快速选择算法

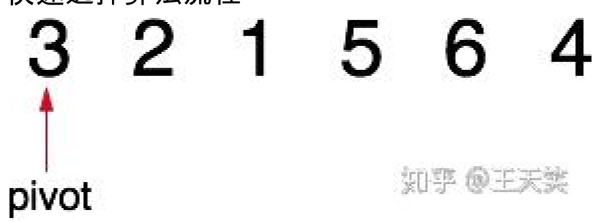
如果我们仔细审查一下我们的问题的话，会发现「Min-Heap 最小堆」和「快速排序算法」都做了一些我们不需要的工作。

- 快速排序算法中，我们排序了数组中的所有值，通过排序后的数组，我们可以得到 Top 1，..., Top K, Top K + 2 ... 的值，但实际上我们只需要 Top K 的值。
- Min-Heap 最小堆中，我们可以得到 Top 1，Top K - 1，Top K 的值，因为 k 一般都比数组长度小，所以我们能减少一些重复计算，但仍然重复计算了 Top 1，Top K - 1 等值。

重新观察一下快速排序后的数组。
k = 2
n = nums.length // 6
[1, 2, 3, 4, 5, 6]
↑
n - k

在一个排序后的数组中， n - k 位置的值就是我们要找的 Top k。
k = 2
n = nums.length // 6
[3, 1, 4, 2, 5, 6]
↑
n - k
我们甚至不需要一个排序的数组，只要 n - k 的左边是小于 nums[n-k] 的值，n - k 的右边是大于 nums[n-k] 的值，那么 n - k 就是我们要找的 Top k。
所以问题转化为如何切割左右数组，并找到 Top k 对应的 pivot。

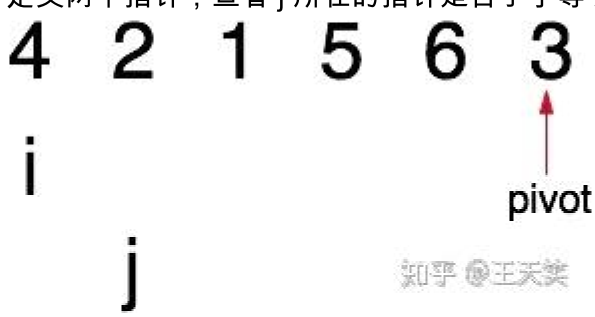
快速选择算法流程



随机选择一个 pivot，通过一系列计算，查看是否是我们需要找到的 Top k 对应的 pivot

把 pivot 移动到最右的位置，已最右为标杆，从头开始对剩余部分进行选择查找

定义两个指针，查看 j 所在的指针是否小于等于 pivot，4 大于 3，所以我们把 j 指针右移一位。

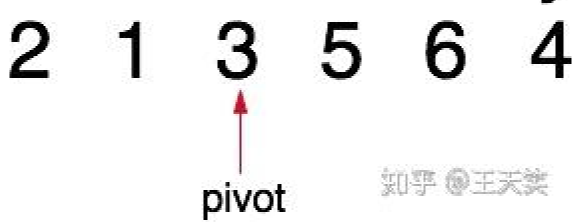


查看 j 所在的指针是否小于等于 pivot，2 小于等于 3，我们替换 i 指针和 j 指针所在的位置，同时把 i 和 j 指针都右移一位。

查看 j 所在的指针是否小于等于 pivot，1 小于等于 3, 我们替换 i 指针和 j 指针所在的位置，同时把 i 和 j 指针都右移一位。

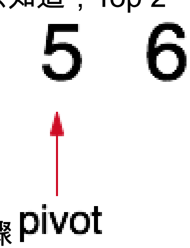
重复上述步骤，直到 j 指针移动到最右边

替换 i 和 j 指针的位置，把 pivot 复位。



此时 pivot 3 左边的值都小于 3，右边的值都大于 3，pivot 3 对应的是 Top 4 而不是 我们需要找的 Top 2，但我们可以知道，Top 2 一定在 pivot 3 右边的位置。

把右半部分按照重复执行上述步骤



最终找到 Top 2 的 pivot, 对应的值为 5

注意事项

pivot 的选择很重要，如果对于一个已排序的数组，我们每次都选择最大/最小的值为 pivot，那么时间复杂度为 O(N^2) 。
每次通过 random 选择 pivot 可以尽量避免最坏情况发生。

```
const findKthLargest = (nums, k) => {
  return quickSelect(nums, 0, nums.length - 1, k);
};

const quickSelect = (nums, lo, hi, k) => {
  // 避免最坏情况发生
  const p = Math.floor(Math.random() * (hi - lo + 1)) + lo;
  swap(nums, p, hi);

  let i = lo;
  let j = lo;

  while (j < hi) {
    if (nums[j] <= nums[hi]) {
      swap(nums, i++, j);
    }
    j++;
  }
  swap(nums, i, j);
  // pivot 是我们要找的 Top k
  if (hi === k + i - 1) return nums[i];
  // Top k 在右边
  if (hi > k + i - 1) return quickSelect(nums, i + 1, hi, k);
  // Top k 在左边
  return quickSelect(nums, lo, i - 1, k - (hi - i + 1));
};
```

const swap = (nums, i, j) => ([nums[i], nums[j]] = [nums[j], nums[i]]);

快速选择算法的平均时间复杂度是 O(N)，但最坏情况下的时间复杂度是 O(N^2) ，因为我们已经随机选择 pivot，所以能够最大程度上的减少最坏情况发生。

算法变体

最简单的快速排序变化是每次随机选择基准值，这样可以达到近乎线性的复杂度。更为确定的做法是采用“取三者中位数”[2]的基准值选择策略，这样对已部分排序的数据依然能够达到线性复杂度。但是，特定人为设置的数组在此方法下仍然会导致最差时间复杂度，如大卫·穆塞尔所描述的“取三者中位数杀手”数列，这成为他发表反省式选择算法的动机。
利用中位数的中位数算法，可以在最坏情形下依然保证线性时间复杂度。但是这一方法中的基准值计算十分复杂，实际应用中并不常见。改进方法是在快速选择算法的基础上，使用“中位数的中

位数”算法处理极端特例，这样可以保证平均状态与最差情形下的时间复杂度都是线性的，这也是反省式选择算法的做法。[1]