入选理由

- 经典第 K 个最大(小)元素问题
- 堆问题入门

标签

- 堆

难度

- 中等

题目地址(215. 数组中的第 K 个最大元素)

https://leetcode-cn.com/problems/kth-largest-element-in-an-array/

题目描述

```
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
示例 2:

输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
说明:

你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。
```

前置知识

- 堆
- 排序

解法一 （排序）

思路

很直观的解法就是给数组排序，这样求解第 `K` 大的数，就等于是从小到大排好序的数组的第 `(n-K)` 小的数 (n 是数组的长度）。

> 当然你也可以从大到小排序，然后直接取第 k 个。

例如：

```
[3,2,1,5,6,4], k = 2

第一步. 数组排序:
 [1,2,3,4,5,6],
第二步. 找第 (n-k) 小的数
 n-k=4, nums[4]=5 (第2大的数)
```

代码

代码支持：Java，Python3，CPP

Java Code：

```java
class KthLargestElementSort {

    public int findKthlargest2(int[] nums, int k) {

        Arrays.sort(nums);
        return nums[nums.length - k];
    }
}
```

Python3 Code：

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

CPP Code:

```cpp
#include <iostream>
#include <vector>

using namespace std;

class Solution {
public:
    int findKthLargest(vector<int> &nums, int k) {
        int size = nums.size();
        sort(begin(nums), end(nums));
        return nums[size - k];
    }
};
```

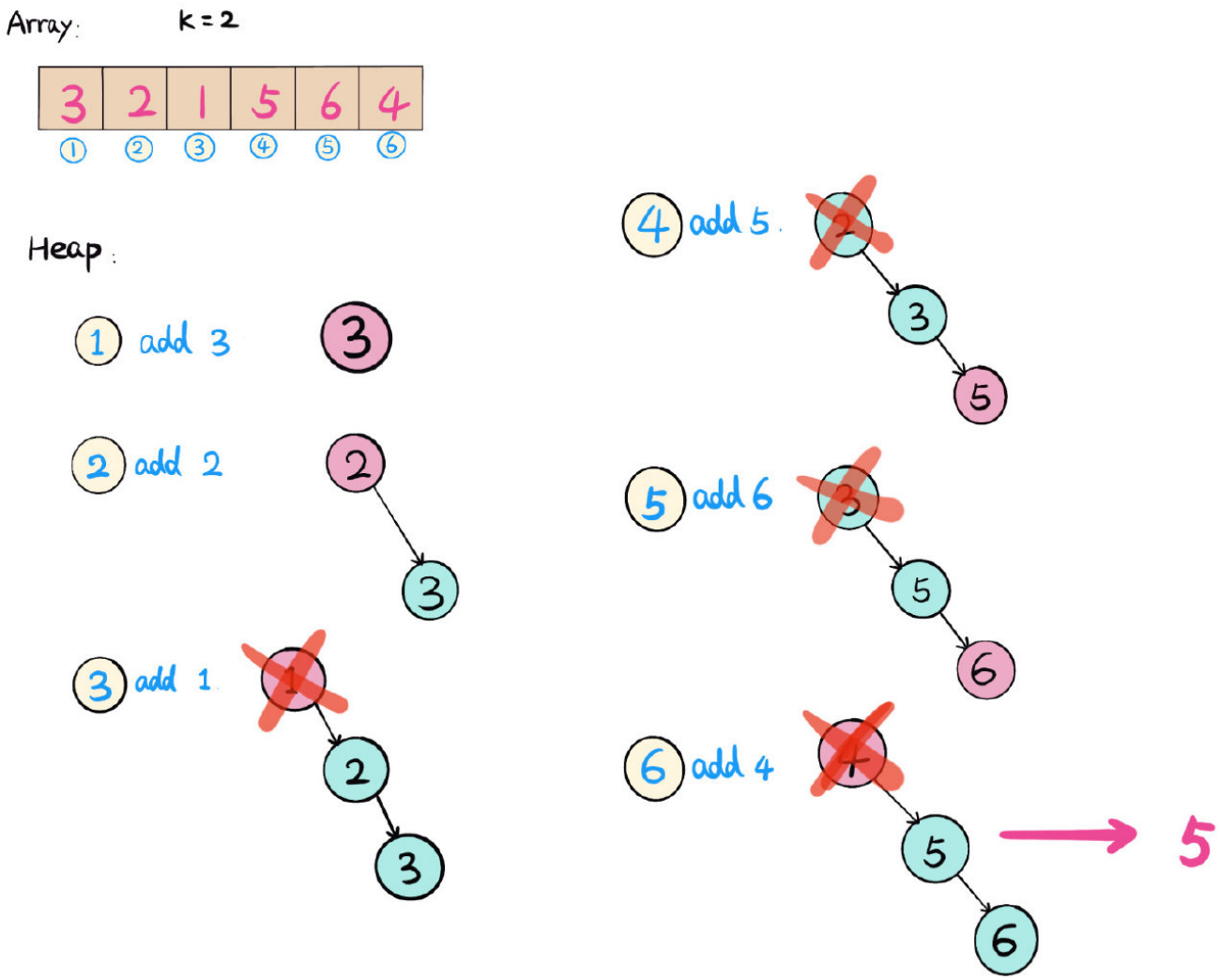
时间和空间复杂度取决于排序算法本身。

解法二 - 小顶堆（Heap）

思路

可以维护一个大小为 `K` 的小顶堆，堆顶是最小元素，当堆的 `size > K` 的时候，删除堆顶元素. 扫描一遍数组，最后堆顶就是第 `K` 大的元素。 直接返回。

例如：



这其实就是讲义中提到的**固定堆**技巧。

代码

使用自带的数据结构。

代码支持：Java，CPP, Python3,JS

Java Code:

```
class Solution {

    public int findKthLargest(int[] nums, int k) {

        PriorityQueue<Integer> pq = new PriorityQueue<>();

        for (int num : nums) {

            if (pq.size() < k)
                pq.offer(num);
```

```
            else if (pq.peek() < num) {

                pq.poll();
                pq.offer(num);
            }
        }

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

Cpp Code:

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {

        priority_queue<int, vector<int>, less<int>> big_heap;
        for(int i = 0; i < nums.size(); ++i)
            big_heap.push(nums[i]);

        for(int i = 0; i < k-1; ++i)
            big_heap.pop();
        return big_heap.top();
    }
};
```

Python3 Code:

```
import heapq
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        size = len(nums)

        h = []
        for index in range(k):
            # heapq 默认就是小顶堆
            heapq.heappush(h, nums[index])

        for index in range(k, size):
            if nums[index] > h[0]:
                heapq.heapreplace(h, nums[index])
        return h[0]
```

JS Code:

```
```

```
var findKthLargest = function (nums, k) {
  const pq = new MinPriorityQueue();
  i = 0;
  while (i < nums.length) {
    pq.enqueue("x", nums[i]);
    if (pq.size() > k) pq.dequeue();
    i++;
  }
  return pq.dequeue()["priority"];
};
```

手撕实现。

代码支持：Java，CPP

Java Code：

```
class Solution {

    int count = 0;

    public int findKthLargest(int[] nums, int k) {

        int[] heap = new int[k + 1];
        for (int num: nums)
            add(heap, num, k);
        return heap[1];
    }

    public void add(int[] heap, int elem, int k) {

        if (count == k && heap[1] >= elem)
            return;

        if (count == k && heap[1] < elem) {

            heap[1] = heap[count--];
            siftDown(heap, k, 1);
        }

        heap[++count] = elem;
        siftUp(heap, count);
    }

    public void siftUp(int[] heap, int index) {

        while (index > 1 && heap[index] <= heap[index / 2]) {

            exch(heap, index, index / 2);
            index /= 2;
        }
```

```
    }

    public void siftDown(int[] heap, int k, int index) {

        while (index * 2 <= k) {

            int j = index * 2;

            if (j < k && heap[j] > heap[j + 1])
                j++;

            if (heap[index] < heap[j])
                break;

            exch(heap, index, j);
            index = j;
        }
    }

    public void exch(int[] heap, int x, int y) {

        int temp = heap[x];
        heap[x] = heap[y];
        heap[y] = temp;
    }
}
```

CPP Code：

```
class Solution {
public:
    void maxHeapify(vector<int>& a, int i, int heapSize) {
        int l = i * 2 + 1, r = i * 2 + 2, largest = i;
        if (l < heapSize && a[l] > a[largest]) {
            largest = l;
        }
        if (r < heapSize && a[r] > a[largest]) {
            largest = r;
        }
        if (largest != i) {
            swap(a[i], a[largest]);
            maxHeapify(a, largest, heapSize);
        }
    }

    void buildMaxHeap(vector<int>& a, int heapSize) {
        for (int i = heapSize / 2; i >= 0; --i) {
            maxHeapify(a, i, heapSize);
        }
    }

    int findKthLargest(vector<int>& nums, int k) {
        int heapSize = nums.size();
```

```
        buildMaxHeap(nums, heapSize);
        for (int i = nums.size() - 1; i >= nums.size() - k + 1; --i) {
            swap(nums[0], nums[i]);
            --heapSize;
            maxHeapify(nums, 0, heapSize);
        }
        return nums[0];
    }
};
```

时间复杂度：`O(n * logk) , n is array length`

空间复杂度：`O(k)`

> 跟排序相比，以空间换时间。

总结

1. 直接排序很简单，但是时间复杂度过高。
2. 堆（Heap）主要是要维护一个 K 大小的小顶堆，扫描一遍数组，最后堆顶元素即是所求。本质上是空间换时间。

上一页 下一页