找到无序数组中最小的K个数

【题目】

给定一个无序的整型数组arr，找到其中最小的k个数。

【要求】

如果数组arr的长度为N，排序之后自然可以得到最小的k个数，此时时间复杂度为排序的时间复杂度即O(N*logN)。本题要求读者实现时间复杂度O(N*logK)和O(N)的方法。

【难度】

O(N*logK)的方法 尉 ★★☆☆

O(N)的方法 将 ★★★★

【解答】

依靠把arr排序的方法太过简单，时间复杂度也不好，所以本书不再详述。

O(N*logK)的方法。说起来也非常简单，就是一直维护一个k个数的大根堆，这个堆代表目前选出的k个最小的数，在堆里的k个元素中堆顶的元素是最小的k个数里最大的那个。

接下来遍历整个数组，遍历的过程中看看当前数是否比堆顶元素小。如果是，就把堆顶的元素替换成当前的数，然后从堆顶的位置调整整个堆，让替换操作后的堆的最大元素继续处在堆顶的位置；如果不是，不进行任何的操作，继续遍历下一个数；在遍历完成后，堆中的k个数就是所有数组中最小的k个数。

具体请参看如下代码中的getMinKNumsByHeap方法，代码中的heapInsert和heapify方法分别为堆排序中的建堆和调整堆的实现。

```java
public int[] getMinKNumsByHeap(int[] arr, int k) {
	if (k < 1 || k > arr.length) {
		return arr;
	}
	int[] kHeap = new int[k];
	for (int i = 0; i != k; i++) {
		heapInsert(kHeap, arr[i], i);
	}
	for (int i = k; i != arr.length; i++) {
		if (arr[i] < kHeap[0]) {
			kHeap[0] = arr[i];
			heapify(kHeap, 0, k);
		}
	}
	return kHeap;
}

public void heapInsert(int[] arr, int value, int index) {
	arr[index] = value;
	while (index != 0) {
		int parent = (index - 1) / 2;
		if (arr[parent] < arr[index]) {
			swap(arr, parent, index);
			index = parent;
		} else {
			break;
		}
	}
}

public void heapify(int[] arr, int index, int heapSize) {
	int left = index * 2 + 1;
	int right = index * 2 + 2;
	int largest = index;
```

```
		while (left < heapSize) {
			if (arr[left] > arr[index]) {
				largest = left;
			}
			if (right < heapSize && arr[right] > arr[largest]) {
				largest = right;
			}
			if (largest != index) {
				swap(arr, largest, index);
			} else {
				break;
			}
			index = largest;
			left = index * 2 + 1;
			right = index * 2 + 2;
		}
	}

	public void swap(int[] arr, int index1, int index2) {
		int tmp = arr[index1];
		arr[index1] = arr[index2];
		arr[index2] = tmp;
	}
```

O(N)的解法。需要用到一个经典的算法—BFPRT算法，该算法于1973年，由Blum、Floyd、Pratt、Rivest和Tarjan联合发明，其中蕴含的深刻思想改变了世界。BFPRT算法解决了这样一个问题，在时间复杂度O(N)内，从无序的数组中找到第k小的数。显而易见的是，如果我们找到了第k小的数，那么想求arr中最小的k个数，就是再遍历一次数组的工作量而已，所以关键问题就变成了如何理解并实现BFPRT算法。

BFPRT算法是如何找到第k小的数的呢？以下是BFPRT算法的过程，假设BFPRT算法的函数是int select(int[] arr, k)，该函数的功能为在arr中找到第k小的数，然后返回该数。

select(arr, k)的过程为：

1，将arr中的n个元素划分成n/5组，每组5个元素，如果最后的组不够5个元素，那么最后剩下的元素为一组(n%5个元素)；

2，对每个组进行插入排序，只是每个组最多5个元素之间的组内排序，组与组之间并不排序。排序后找到每个组的中位数，如果组的元素个数为偶数，这里规定找到下中位数；

3，步骤2中一共会找到n/5个中位数，让这些中位数组成一个新的数组，记为mArr。递归调用select(mArr,mArr.length/2)，意义是找到mArr这个数组中的中位数，即mArr中的第(mArr.length/2)小的数；

4，假设步骤3中递归调用select(mArr,mArr.length/2)后，返回的数为x。根据这个x划分整个arr数组(partition过程)，划分完成的功能为在arr中，比x小的数都在x的左边，大于x的数都在x的右边，x在中间。假设划分完成后，x在arr中的位置记为i；

5，如果i==k，说明x为整个数组中第k小的数，直接返回。

　　如果i<k，说明x的处在第k小的数的左边，应该在x的右边寻找第k小的数，所以递归调用select函数，在左半区寻找第k小的数。

　　如果i>k，说明x的处在第k小的数的右边，应该在x的左边寻找第k小的数，所以递归调用select函数，在右半区寻找第(i-k)小的数。

BFPRT算法为什么在时间复杂度上可以做到稳定的O(N)呢？以下是BFPRT的时间复杂度分析，我们假设BFPRT算法处理大小为N的数组时，时间复杂度函数为T(N)。

1，如上的过程中，除了步骤3和步骤5要递归调用select函数之外，其他所有处理过程都可以在O(N)的时间内完成；

2，步骤3中有递归调用select的过程，且递归处理的数组大小最大为n/5，即T(N/5)；

3，步骤5也递归调用了select，那么递归处理的数组大小最大为多少呢？具体来说，我们关心的是由x划分出的左半区最大有多大和由x划分出的左半区最大有多大。以下是右半区域的大小计算过程(左半区域的计算过程也类似)，这也是整个BFPRT算法的精髓。

　　因为x是5个数一组的中位数组成的数组(mArr)中的中位数，所以在mArr中(mArr大小为N/5)，有一半的数(N/10个)都比x要小。

所有在mArr中比x小的所有数，在各自的组中又肯定比2个数要大，因为在mArr中的每一个数都是各自组中的中位数。

所以至少有(N/10)*3的数比x要小，这里我们必须减去两个特殊的组，一个是x自己所在的组，一个是可能元素数量不足5个的组，所以至少有(N/10-2)*3的数比x要小。

既然至少有(N/10-2)*3的数比x要小，那么至多有N-(N/10-2)*3的数比x要大，也就是7N/10+6个数比x要大，也就是右半区最大的量。

左半区可以用类似的分析过程求出依然是至多有7N/10+6个数比x要小。

所以整个步骤5的复杂度为T(7N/10 + 6)。

综上所述，T(N) = O(N) + T(N/5) + T(7N/10+6)，可以在数学上证明T(N)的复杂度就是O(N)，详细证明过程请参看《算法导论》9.3章节，本书不再详述。

为什么要如此费力的这么处理arr数组呢？又要5个数分1组，又要求中位数的中位数，又要划分的，好麻烦啊。就是因为以中位数的中位数x划分的数组，可以在步骤5的递归时，确保肯定淘汰掉一定的数据量，起码淘汰掉3N/10-6的数据量！

不得不说的是，关于选择划分元素的问题，很多实现都是随便找一个数进行数组的划分，也就是类似随机快排的划分方式，这种划分方式无法达到时间复杂度O(N)的原因就是并不能确定淘汰的数据量，而BFPRT算法在划分时，使用的是中位数的中位数进行划分，从而确定了淘汰的数据量，最后成功的让时间复杂度收敛到O(N)的程度。

本书的实现对BFPRT算法做了更好的改进，主要改进的地方是当中位数的中位数x，在arr中大量出现的时候，那么在划分之后到底返回什么位置上的x呢？

在本书的实现中，返回了在通过x划分arr后，等于x的整个位置区间，比如pivotRange=[a,b]表示arr[a..b]上都是x，并以此区间去命中第k小的数如果k在[a,b]上就是命中，如果没在[a,b]上表示没命中。这样即可以尽量少的进行递归过程，又可以增加淘汰的数据量，使得步骤5递归过程变得数据量更少。

具体过程请参看如下代码中的getMinKNumsByBFPRT方法。

```
	public int[] getMinKNumsByBFPRT(int[] arr, int k) {
		if (k < 1 || k > arr.length) {
			return arr;
		}
		int minKth = getMinKthByBFPRT(arr, k);
		int[] res = new int[k];
		int index = 0;
		for (int i = 0; i != arr.length; i++) {
			if (arr[i] < minKth) {
				res[index++] = arr[i];
			}
		}
		for (; index != res.length; index++) {
			res[index] = minKth;
		}
		return res;
	}

	public int getMinKthByBFPRT(int[] arr, int K) {
		int[] copyArr = copyArray(arr);
		return select(copyArr, 0, copyArr.length - 1, K - 1);
	}

	public int[] copyArray(int[] arr) {
		int[] res = new int[arr.length];
		for (int i = 0; i != res.length; i++) {
			res[i] = arr[i];
		}
		return res;
	}

	public int select(int[] arr, int begin, int end, int i) {
		if (begin == end) {
			return arr[begin];
		}
```

```java
        int pivot = medianOfMedians(arr, begin, end);
        int[] pivotRange = partition(arr, begin, end, pivot);
        if (i >= pivotRange[0] && i <= pivotRange[1]) {
            return arr[i];
        } else if (i < pivotRange[0]) {
            return select(arr, begin, pivotRange[0] - 1, i);
        } else {
            return select(arr, pivotRange[1] + 1, end, i);
        }
    }

    public int medianOfMedians(int[] arr, int begin, int end) {
        int num = end - begin + 1;
        int offset = num % 5 == 0 ? 0 : 1;
        int[] mArr = new int[num / 5 + offset];
        for (int i = 0; i < mArr.length; i++) {
            int beginI = begin + i * 5;
            int endI = beginI + 4;
            mArr[i] = getMedian(arr, beginI, Math.min(end, endI));
        }
        return select(mArr, 0, mArr.length - 1, mArr.length / 2);
    }

    public int[] partition(int[] arr, int begin, int end, int pivotValue) {
        int small = begin - 1;
        int cur = begin;
        int big = end + 1;
        while (cur != big) {
            if (arr[cur] < pivotValue) {
                swap(arr, ++small, cur++);
            } else if (arr[cur] > pivotValue) {
                swap(arr, cur, --big);
            } else {
                cur++;
            }
        }
        int[] range = new int[2];
        range[0] = small + 1;
        range[1] = big - 1;
        return range;
    }

    public int getMedian(int[] arr, int begin, int end) {
        insertionSort(arr, begin, end);
        int sum = end + begin;
        int mid = (sum / 2) + (sum % 2);
        return arr[mid];
    }

    public void insertionSort(int[] arr, int begin, int end) {
        for (int i = begin + 1; i != end + 1; i++) {
            for (int j = i; j != begin; j--) {
                if (arr[j - 1] > arr[j]) {
                    swap(arr, j - 1, j);
                } else {
                    break;
                }
            }
        }
    }
```