题目：

给定一个无序的整型数组arr，找到其中最小的k个数。

方法一：

将数组排序，排序后的数组的前k个数就是最小的k个数。

时间复杂度：O(nlogn)

方法二：

时间复杂度：O(nlogk)

维护一个有k个数的大根堆，这个堆代表目前选出的k个最小的数。在堆的k个元素中堆顶元素是最小的k个数中最大的那个。

接下来要遍历整个数组，遍历的过程中看当前数是否比堆顶元素小。如果是，就把堆顶元素替换成当前数，然后调整堆。如果不是，则不做任何操作，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。

程序：

```
public static int[] getMinKNumsByHeap(int[] arr, int k) {
        if (k < 1 || k > arr.length) {
            return arr;
        }
        int[] heap = new int[k];
        for (int i = 0; i != k; i++) {
            heapInsert(heap, arr[i], i);
        }
        for (int i = k; i < arr.length; i++) {
            if (arr[i] < heap[0]) {
                heap[0] = arr[i];
                heapify(heap, 0, k);
            }
        }
        return heap;
    }
    private static void heapInsert(int[] heap, int value, int index) {
        heap[index] = value;
        while (index != 0) {
            int parent = (index - 1) / 2;
            if (heap[parent] < heap[index]) {
                swap(heap, parent, index);
                index = parent;
            } else {
                break;
            }
        }
    }
    private static void heapify(int[] heap, int index, int heapSize) {
        int left = index * 2 + 1;
        int right = index * 2 + 2;
```

```
        int largest = index;
        while (left < heapSize) {
            if (heap[left] > heap[index]) {
                largest = left;
            }
            if (right < heapSize && heap[right] > heap[largest]) {
                largest = right;
            }
            if (largest != index) {
                swap(heap, largest, index);
            } else {
                break;
            }
            index = largest;
            left = index * 2 + 1;
            right = index * 2 + 2;
        }
    }
    private static void swap(int[] heap, int parent, int index) {
        int tmp = heap[index];
        heap[index] = heap[parent];
        heap[parent] = tmp;
    }
```

方法三：

时间复杂度：O(n)

这里用到了一个经典算法----BFPRT算法。

1973 年， Blum 、 Floyd 、 Pratt 、 Rivest 、 Tarjan 集体出动，合写了一篇题为 “Time bounds for selection” 的论文，给出了一种在数组中选出第 k 大元素的算法，俗称"中位数之中位数算法"。依靠一种精心设计的 pivot 选取方法，该算法从理论上保证了最坏情形下的线性时间复杂度，打败了平均线性、最坏 O(n^2) 复杂度的传统算法。一群大牛把递归算法的复杂度分析玩弄于股掌之间，构造出了一个当之无愧的来自圣经的算法。

算法步骤：

step1：将n个元素每5个一组，分成n/5(上界)组，最后的一个组的元素个数为n%5，有效的组数为n/5。

step2：取出每一组的中位数，最后一个组的不用计算中位数，任意排序方法，这里的数据比较少只有5个，可以用简单的冒泡排序或是插入排序。

setp3 : 将各组的中位数与数组开头的数据在组的顺序依次交换，这样各个组的中位数都排在了数据的左边。递归的调用中位数选择算法查找上一步中所有组的中位数的中位数，设为x，偶数个中位数的情况下设定为选取中间小的一个。

setp4: 按照x划分,大于或者等于x的在右边,小于x的在左边，关于setp4数据的划分，中位数放在左边或是右边会有些影响。后面的代码调试将会看到。

step5：setp4中划分后数据后返回一个下表i，i左边的元素均是小于x，i右边的元素包括i都是大于或是等于x的。

- 若i==k，返回x；
- 若i<k，在小于x的元素中递归查找第i小的元素；
- 若i>k，在大于等于x的元素中递归查找第i-k小的元素。

```
public static 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 static int getMinKthByBFPRT(int[] arr, int K) {
        int[] copyArr = copyArray(arr);
        return select(copyArr, 0, copyArr.length - 1, K - 1);
    }
    public static 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 static int select(int[] arr, int begin, int end, int i) {
        if (begin == end) {
            return arr[begin];
        }
        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 static 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 static 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 static 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 static 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;
                }
            }
        }
    }
```