```java
public void printMatrixZigZag(int[][] matrix) {
	int tR = 0;
	int tC = 0;
	int dR = 0;
	int dC = 0;
	int endR = matrix.length - 1;
	int endC = matrix[0].length - 1;
	boolean fromUp = false;
	while (tR != endR + 1) {
		printLevel(matrix, tR, tC, dR, dC, fromUp);
		tR = tC == endC ? tR + 1 : tR;
		tC = tC == endC ? tC : tC + 1;
		dC = dR == endR ? dC + 1 : dC;
		dR = dR == endR ? dR : dR + 1;
		fromUp = !fromUp;
	}
	System.out.println();
}

public void printLevel(int[][] m, int tR, int tC, int dR, int dC, boolean f) {
	if (f) {
		while (tR != dR + 1) {
			System.out.print(m[tR++][tC--] + " ");
		}
	} else {
		while (dR != tR - 1) {
			System.out.print(m[dR--][dC++] + " ");
		}
	}
}
```

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，此时时间复杂度与排序的时间复杂度相同，均为 $O(N\log N)$。本题要求读者实现时间复杂度为 $O(N\log k)$和 $O(N)$ 的方法。

【难度】

$O(N\log k)$的方法 尉 ★★☆☆

$O(N)$的方法 将 ★★★★

【解答】

依靠把 arr 进行排序的方法太简单，时间复杂度也不好，所以本书不再详述。

$O(N\log k)$的方法。说起来也非常简单，就是一直维护一个有 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，让替换操作后堆的最大元素继续处在堆顶的位置；如果不是，则不进行任何操作，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。

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。

```
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？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)的过程如下：

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，只针对每个组最多 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），划分的过程为：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 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 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，其他所有的处理过程都可以在 $O(N)$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？具体地说，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$(N/10)\times3$ 的数比 x 要小，这里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$(N/10-2)\times3$ 的数比 x 要小。
- 既然至少有$(N/10-2)\times3$ 的数比 x 要小，那么至多有 $N-(N/10-2)\times3$ 的数比 x 要大，也就是 $7N/10+6$ 个数比 x 要大，即右半区最大的量。
- 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相关图书（例如，《算法导论》中 9.3 节的内容），本书不再详述。

为什么要如此费力地这么处理 arr 数组呢？要 5 个数分 1 组，又要求中位数的中位数，

还要划分，好麻烦。这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，也就是类似随机快速排序的划分方式，这种划分方式无法达到时间复杂度为 $O(N)$的原因是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，最后成功地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，如果在[a,b]上，就是命中，如果没在[a,b]上，表示没命中。这样既可以尽量少地进行递归过程，又可以增加淘汰的数据量，使得步骤 5 的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。

```java
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) {      复制数组以便进行操作
	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) { 选择，左或右区间
		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 int medianOfMedians(int[] arr, int begin, int end) { 中位数的中位数
		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;                以pivot值划分左右区间
		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) { 获取子数组的中位数
		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) {   插入排序
        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;
                }
            }
        }
    }
```

需要排序的最短子数组长度

【题目】

给定一个无序数组 arr，求出需要排序的最短子数组长度。

例如：arr = [1，5，3，4，2，6，7]返回 4，因为只有[5，3，4，2]需要排序。

【难度】

士 ★☆☆☆

【解答】

解决这个问题可以做到时间复杂度为 $O(N)$、额外空间复杂度为 $O(1)$。

初始化变量 noMinIndex=-1，从右向左遍历，遍历的过程中记录右侧出现过的数的最小值，记为 min。假设当前数为 arr[i]，如果 arr[i]>min，说明如果要整体有序，min 值必然会挪到 arr[i]的左边。用 noMinIndex 记录最左边出现这种情况的位置。如果遍历完成后，noMinIndex 依然等于-1，说明从右到左始终不升序，原数组本来就有序，直接返回 0，即完全不需要排序。

接下来从左向右遍历，遍历的过程中记录左侧出现过的数的最大值，记为 max。假设当前数为 arr[i]，如果 arr[i]<max，说明如果排序，max 值必然会挪到 arr[i]的右边。用变量 noMaxIndex 记录最右边出现这种情况的位置。

遍历完成后，arr[noMinIndex..noMaxIndex]是真正需要排序的部分，返回它的长度即可。

具体过程参看如下代码中的 getMinLength 方法。