第 8 章

数组和矩阵问题

转圈打印矩阵

【题目】

给定一个整型矩阵 matrix，请按照转圈的方式打印它。

例如：

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 16

打印结果为：1，2，3，4，8，12，16，15，14，13，9，5，6，7，11，10

【要求】

额外空间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

本题在算法上没有难度，关键在于设计一种逻辑容易理解、代码易于实现的转圈遍历

方式。这里介绍这样一种矩阵处理方式，该方式不仅可用于这道题，还适合很多其他的面试题，就是矩阵分圈处理。在矩阵中用左上角的坐标(tR,tC)和右下角的坐标(dR,dC)就可以表示一个子矩阵，比如，题目中的矩阵，当(tR,tC)=(0,0)、(dR,dC)=(3,3)时，表示的子矩阵就是整个矩阵，那么这个子矩阵最外层的部分如下：

```
1   2   3   4
5               8
9               12
13  14  15  16
```

如果能把这个子矩阵的外层转圈打印出来，那么在(tR,tC)=(0,0)、(dR,dC)=(3,3)时，打印的结果为：1，2，3，4，8，12，16，15，14，13，9，5。接下来令 tR 和 tC 加 1，即(tR,tC)=(1,1)，令 dR 和 dC 减 1，即(dR,dC)=(2,2)，此时表示的子矩阵如下：

```
6   7
10  11
```

再把这个子矩阵转圈打印出来，结果为：6，7，11，10。把 tR 和 tC 加 1，即(tR,tC)=(2,2)，令 dR 和 dC 减 1，即(dR,dC)=(1,1)。如果发现左上角坐标跑到了右下角坐标的右方或下方，整个过程就停止。已经打印的所有结果连起来就是我们要求的打印结果。具体请参看如下代码中的 spiralOrderPrint 方法，其中 printEdge 方法是转圈打印一个子矩阵的外层。

```
public void spiralOrderPrint(int[][] matrix) {
        int tR = 0;
        int tC = 0;
        int dR = matrix.length - 1;
        int dC = matrix[0].length - 1;
        while (tR <= dR && tC <= dC) {
                printEdge(matrix, tR++, tC++, dR--, dC--);
        }
}

public void printEdge(int[][] m, int tR, int tC, int dR, int dC) {
        if (tR == dR) { // 子矩阵只有一行时
                for (int i = tC; i <= dC; i++) {
                        System.out.print(m[tR][i] + " ");
                }
        } else if (tC == dC) { // 子矩阵只有一列时
                for (int i = tR; i <= dR; i++) {
                        System.out.print(m[i][tC] + " ");
                }
        } else { // 一般情况
                int curC = tC;
                int curR = tR;
```

```
                while (curC != dC) {
                        System.out.print(m[tR][curC] + " ");
                        curC++;
                }
                while (curR != dR) {
                        System.out.print(m[curR][dC] + " ");
                        curR++;
                }
                while (curC != tC) {
                        System.out.print(m[dR][curC] + " ");
                        curC--;
                }
                while (curR != tR) {
                        System.out.print(m[curR][tC] + " ");
                        curR--;
                }
        }
    }
```

将正方形矩阵顺时针转动 90°

【题目】

给定一个 $N \times N$ 的矩阵 matrix，把这个矩阵调整成顺时针转动 90° 后的形式。

例如：

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

顺时针转动 90° 后为：

13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

【要求】

额外空间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

这里仍使用分圈处理的方式，在矩阵中用左上角的坐标(tR,tC)和右下角的坐标(dR,dC)就可以表示一个子矩阵。比如，题目中的矩阵，当(tR,tC)=(0,0)、(dR,dC)=(3,3)时，表示的子矩阵就是整个矩阵，那么这个子矩阵最外层的部分如下。

1 2 3 4
5 8
9 12
13 14 15 16

在这个外圈中，1，4，16，13 为一组，然后让 1 占据 4 的位置，4 占据 16 的位置，16 占据 13 的位置，13 占据 1 的位置，一组就调整完了。然后 2，8，15，9 为一组，继续占据调整的过程，最后 3，12，14，5 为一组，继续占据调整的过程。然后(tR,tC)=(0,0)、(dR,dC)=(3,3)的子矩阵外层就调整完毕。接下来令 tR 和 tC 加 1，即(tR,tC)=(1,1)，令 dR 和 dC 减 1，即(dR,dC)=(2,2)，此时表示的子矩阵如下。

6 7
10 11

这个外层只有一组，就是 6，7，11，10，占据调整之后即可。所以，如果子矩阵的大小是 $M \times M$，一共就有 M-1 组，分别进行占据调整即可。

具体过程请参看如下代码中的 rotate 方法。

```
public void rotate(int[][] matrix) {
        int tR = 0;
        int tC = 0;
        int dR = matrix.length - 1;
        int dC = matrix[0].length - 1;
        while (tR < dR) {
                rotateEdge(matrix, tR++, tC++, dR--, dC--);
        }
}

public void rotateEdge(int[][] m, int tR, int tC, int dR, int dC) {
        int times = dC - tC; // times 就是总的组数
        int tmp = 0;
        for (int i = 0; i != times; i++) { // 一次循环就是一组占据调整
                tmp = m[tR][tC + i];
```

```
                m[tR][tC + i] = m[dR - i][tC];
                m[dR - i][tC] = m[dR][dC - i];
                m[dR][dC - i] = m[tR + i][dC];
                m[tR + i][dC] = tmp;
            }
        }
```

“之”字形打印矩阵

【题目】

给定一个矩阵 matrix，按照“之”字形的方式打印这个矩阵，例如：

1　　2　　3　　4

5　　6　　7　　8

9　　10　　11　　12

“之”字形打印的结果为：1，2，5，9，6，3，4，7，10，11，8，12

【要求】

额外空间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

本书提供的实现方法是这样处理的：

1．上坐标(tR,tC)初始为(0,0)，先沿着矩阵第一行移动(tC++)，当到达第一行最右边的元素后，再沿着矩阵最后一列移动(tR++)。

2．下坐标(dR,dC)初始为(0,0)，先沿着矩阵第一列移动(dR++)，当到达第一列最下边的元素时，再沿着矩阵最后一行移动(dC++)。

3．上坐标与下坐标同步移动，每次移动后的上坐标与下坐标的连线就是矩阵中的一条斜线，打印斜线上的元素即可。

4．如果上次斜线是从左下向右上打印的，这次一定是从右上向左下打印，反之亦然。总之，可以把打印的方向用 boolean 值表示，每次取反即可。

具体请参看如下代码中的 printMatrixZigZag 方法。

```
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)×3 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)×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，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 3*N*/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。

```
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];
        }
        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;
                }
            }
        }
    }
```

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。

```
public int getMinLength(int[] arr) {
        if (arr == null || arr.length < 2) {
                return 0;
        }
        int min = arr[arr.length - 1];
        int noMinIndex = -1;
        for (int i = arr.length - 2; i != -1; i--) {
                if (arr[i] > min) {
                        noMinIndex = i;
                } else {
                        min = Math.min(min, arr[i]);
                }
        }
        if (noMinIndex == -1) {
                return 0;
        }
        int max = arr[0];
        int noMaxIndex = -1;
        for (int i = 1; i != arr.length; i++) {
                if (arr[i] < max) {
                        noMaxIndex = i;
                } else {
                        max = Math.max(max, arr[i]);
                }
        }
        return noMaxIndex - noMinIndex + 1;
}
```

在数组中找到出现次数大于 *N*/*K* 的数

【题目】

给定一个整型数组 arr，打印其中出现次数大于一半的数，如果没有这样的数，打印提示信息。

【进阶】

给定一个整型数组 arr，再给定一个整数 *K*，打印所有出现次数大于 *N*/*K* 的数，如果没有这样的数，打印提示信息。

【要求】

原问题要求时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。进阶问题要求时间复杂度为 $O(N \times K)$，额外空间复杂度为 $O(K)$。

【难度】

校 ★★★☆

【解答】

无论是原问题还是进阶问题，都可以用哈希表记录每个数及其出现的次数，但是额外空间复杂度为 $O(N)$，不符合题目要求，所以本书不再详述这种简单的方法。本书提供方法的核心思路是，一次在数组中删掉 K 个不同的数，不停地删除，直到剩下数的种类不足 K 就停止删除，那么，如果一个数在数组中出现的次数大于 N/K，则这个数最后一定会被剩下来。

对于原问题，出现次数大于一半的数最多只会有一个，还可能不存在这样的数。具体的过程为，一次在数组中删掉两个不同的数，不停地删除，直到剩下的数只有一种，如果一个数出现次数大于一半，这个数最后一定会剩下来。如下代码中的 printHalfMajor 方法就是这种思路的具体实现，我们先列出代码，然后进行解释。

```
public void printHalfMajor(int[] arr) {
        int cand = 0;
        int times = 0;
        for (int i = 0; i != arr.length; i++) {
                if (times == 0) {
                        cand = arr[i];
                        times = 1;
                } else if (arr[i] == cand) {
                        times++;
                } else {
                        times--;
                }
        }
        times = 0;
        for (int i = 0; i != arr.length; i++) {
                if (arr[i] == cand) {
                        times++;
                }
        }
        if (times > arr.length / 2) {
                System.out.println(cand);
        } else {
                System.out.println("no such number.");
        }
}
```

printHalfMajor 方法中第一个 for 循环就是一次在数组中删掉两个不同的数的代码实现。

我们把变量 cand 叫作候选，times 叫作次数，读者先不用纠结这两个变量是什么意义，我们看在第一个 for 循环中发生了什么。

- times==0 时，表示当前没有候选，则把当前数 arr[i]设成候选，同时把 times 设置成 1。
- times!=0 时，表示当前有候选，如果当前的数 arr[i]与候选一样，就把 times 加 1；如果当前的数 arr[i]与候选不一样，就把 times 减 1，减到 0 则表示又没有候选了。

这具体是什么意思呢？当没有候选时，我们把当前的数作为候选，说明我们找到了两个不同的数中的第一个；当有候选且当前的数和候选一样时，说明目前没有找到两个不同的数中的另外一个，反而是同一种数反复出现了，那么就把 times++表示反复出现的数在累计自己的点数。当有候选且当前的数和候选不一样时，说明找全了两个不同的数，但是候选可能在之前多次出现，如果此时把候选完全换掉，候选的这个数相当于一下被删掉了多个，对吧？所以这时候选“付出”一个自己的点数，即 times 减 1，然后当前数也被删掉。这样还是相当于一次删掉了两个不同的数。当然，如果 times 被减到为 0，说明候选的点数完全被消耗完，那么又表示候选空缺，arr 中的下一个数(arr[i+1])就又被作为候选。

综上所述，第一个 for 循环的实质就是我们的核心解题思路，一次在数组中删掉两个不同的数，不停地删除，直到剩下的数只有一种，如果一个数出现次数大于一半，则这个数最后一定会被剩下来，也就是最后的 cand 值。

这里请注意一点，一个数出现次数虽然大于一半，它肯定会被剩下来，但那并不表示剩下来的数一定是符合条件的。例如，1，2，1。其中 1 符合出现次数超过了一半，所以 1 肯定会剩下来。再如 1，2，3，其中没有任何一个数出现的次数超过了一半，可 3 最后也剩下来了。所以 printHalfMajor 方法中第二个 for 循环的工作就是检验最后剩下来的那个数（即 cand）是否真的是出现次数大于一半的数。如果 cand 都不符合条件，那么其他的数也一定都不符合，说明 arr 中没有任何一个数出现了一半以上。

进阶问题解法核心也是类似的，一次在数组中删掉 K 个不同的数，不停地删除，直到剩下的数的种类不足 K，那么，如果某些数在数组中出现次数大于 N/K，则这些数最后一定会被剩下来。原问题中，我们解决了找到出现次数超过 $N/2$ 的数，解决的办法是立了 1 个候选 cand，以及这个候选的 times 统计。进阶问题具体的实现也类似，只要立 K-1 个候选，然后有 K-1 个 times 统计即可，具体过程如下。

遍历到 arr[i]时，看 arr[i]是否与已经被选出的某一个候选相同：

如果与某一个候选，就把属于那个候选的点数统计加 1。

如果与所有的候选都不相同，先看当前的候选是否选满了，K-1 就是满，否则就是不满：

- 如果不满，把 arr[i]作为一个新的候选，属于它的点数初始化为 1。
- 如果已满，说明此时发现了 *K* 个不同的数，arr[i]就是第 *K* 个。此时把每一个候选各自的点数全部减 1，表示每个候选“付出”一个自己的点数。如果某些候选的点数在减 1 之后等于 0，则还需要把这些候选都删除，候选又变成不满的状态。

在遍历过程结束后，再遍历一次 arr，验证被选出来的所有候选有哪些出现次数真的大于 *N*/*K*，符合条件的候选就打印。具体请参看如下代码中的 printKMajor 方法。

```
public void printKMajor(int[] arr, int K) {
        if (K < 2) {
                System.out.println("the value of K is invalid.");
                return;
        }
        HashMap<Integer, Integer> cands = new HashMap<Integer, Integer>();
        for (int i = 0; i != arr.length; i++) {
                if (cands.containsKey(arr[i])) {
                        cands.put(arr[i], cands.get(arr[i]) + 1);
                } else {
                        if (cands.size() == K - 1) {
                                allCandsMinusOne(cands);
                        } else {
                                cands.put(arr[i], 1);
                        }
                }
        }
        HashMap<Integer, Integer> reals = getReals(arr, cands);
        boolean hasPrint = false;
        for (Entry<Integer, Integer> set : cands.entrySet()) {
                Integer key = set.getKey();
                if (reals.get(key) > arr.length / K) {
                        hasPrint = true;
                        System.out.print(key + " ");
                }
        }
        System.out.println(hasPrint ? "" : "no such number.");
}

public void allCandsMinusOne(HashMap<Integer, Integer> map) {
        List<Integer> removeList = new LinkedList<Integer>();
        for (Entry<Integer, Integer> set : map.entrySet()) {
                Integer key = set.getKey();
                Integer value = set.getValue();
                if (value == 1) {
                        removeList.add(key);
                }
                map.put(key, value - 1);
        }
        for (Integer removeKey : removeList) {
                map.remove(removeKey);
```

```
        }
    }

    public HashMap<Integer, Integer> getReals(int[] arr,
                    HashMap<Integer, Integer> cands) {
            HashMap<Integer, Integer> reals = new HashMap<Integer, Integer>();
            for (int i = 0; i != arr.length; i++) {
                    int curNum = arr[i];
                    if (cands.containsKey(curNum)) {
                            if (reals.containsKey(curNum)) {
                                    reals.put(curNum, reals.get(curNum) + 1);
                            } else {
                                    reals.put(curNum, 1);
                            }
                    }
            }
            return reals;
    }
```

【扩展】

这种一次删掉 K 个不同的数的思想在面试中通常会变形之后反复出现。例如，下面这道面试真题：有一场投票，投票有效的条件是必须有一个候选人得票数超过半数，但是验票人员不能看到每张选票上选了谁，只能把任意两张选票放到一台机器上看这两张选票是否一样，若一样，则机器给出 true 的提醒，不一样则给出 false 的提醒。如果你作为验票的人员，怎么判断这场投票是有效的？

这道题目就是原问题的变形，但是“不能看到每张选票上选了谁”的这个限制实际上把用哈希表来解题的可能性完全堵死了。但本文的方法却可以满足题目的要求，因为我们实现的方法只需要当前数和候选数做比较，而不需要知道每个数的值。

在行列都排好序的矩阵中找数

【题目】

给定一个有 $N \times M$ 的整型矩阵 matrix 和一个整数 K，matrix 的每一行和每一列都是排好序的。实现一个函数，判断 K 是否在 matrix 中。

例如：

0　1　2　5

2　3　4　7

4　　4　　4　　8
5　　7　　7　　9
如果 *K* 为 7，返回 true；如果 *K* 为 6，返回 false。

【要求】

时间复杂度为 *O*(*N*+*M*)，额外空间复杂度为 *O*(1)。

【难度】

士 ★☆☆☆

【解答】

符合要求的解法比较巧妙且易于理解。
可以用以下步骤解决：
1．从矩阵最右上角的数开始寻找(row=0，col=M-1)。
2．比较当前数 matrix[row][col]与 *K* 的关系：
- 如果与 *K* 相等，说明已找到，直接返回 true；
- 如果比 *K* 大，因为矩阵每一列都已排好序，所以在当前数所在的列中，处于当前数下方的数都会比 *K* 大，则没有必要继续在第 col 列上寻找，令 col=col-1，重复步骤 2。
- 如果比 *K* 小，因为矩阵每一行都已排好序，所以在当前数所在的行中，处于当前数左方的数都会比 *K* 小，则没有必要继续在第 row 行上寻找，令 row=row+1，重复步骤 2。

3．如果找到越界都没有发现与 *K* 相等的数，则返回 false。
或者，也可以用以下步骤：
1．从矩阵最左下角的数开始寻找（row=N-1，col=0）。
2．比较当前数 matrix[row][col]与 *K* 的关系：
- 如果与 *K* 相等，说明已找到，直接返回 true。
- 如果比 *K* 大，因为矩阵每一行都已排好序，所以在当前数所在的行中，处于当前数右方的数都会比 *K* 大，则没有必要继续在第 row 行上寻找，令 row=row-1，重复步骤 2。
- 如果比 *K* 小，因为矩阵每一列都已排好序，所以在当前数所在的列中，处于当前

数上方的数都会比 *K* 小，则没有必要继续在第 col 列上寻找，令 col=col+1，重复步骤 2。

3．如果找到越界都没有发现与 *K* 相等的数，则返回 false。

具体请参看如下代码中的 isContains 方法：

```
public boolean isContains(int[][] matrix, int K) {
        int row = 0;
        int col = matrix[0].length - 1;
        while (row < matrix.length && col > -1) {
                if (matrix[row][col] == K) {
                        return true;
                } else if (matrix[row][col] > K) {
                        col--;
                } else {
                        row++;
                }
        }
        return false;
}
```

最长的可整合子数组的长度

【题目】

先给出可整合数组的定义。如果一个数组在排序之后，每相邻两个数差的绝对值都为 1，则该数组为可整合数组。例如，[5,3,4,6,2]排序之后为[2,3,4,5,6]，符合每相邻两个数差的绝对值都为 1，所以这个数组为可整合数组。

给定一个整型数组 arr，请返回其中最大可整合子数组的长度。例如，[5,5,3,2,6,4,3]的最大可整合子数组为[5,3,2,6,4]，所以返回 5。

【难度】

尉 ★★☆☆

【解答】

时间复杂度高但容易理解的做法。对 arr 中的每一个子数组 arr[i..j](0<=i<=j<=N-1)，都验证一下是否符合可整合数组的定义，也就是把 arr[i..j]排序一下，看是否依次递增且每次递增 1。然后在所有符合可整合数组定义的子数组中，记录最大的那个长度，返回即可。

需要注意的是，在考查每一个 arr[i..j]是否符合可整合数组定义的时候，都得把 arr[i..j]单独复制成一个新的数组，然后把这个新的数组排序、验证，而不能直接改变 arr 中元素的顺序。所以大体过程如下：

1．依次考查每一个子数组 arr[i..j](0<=i<=j<=N-1)，一共有 $O(N^2)$个。

2．对每一个子数组 arr[i..j]，复制成一个新的数组，记为 newArr，把 newArr 排序，然后验证是否符合可整合数组的定义，这一步代价为 $O(N\log N)$。

3．步骤 2 中符合条件的、最大的那个子数组的长度就是结果。

具体请参看如下代码中的 getLIL1 方法，时间复杂度为 $O(N^2)\times O(N\log N)->O(N^3 \log N)$。

```
public int getLIL1(int[] arr) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        int len = 0;
        for (int i = 0; i < arr.length; i++) {
                for (int j = i; j < arr.length; j++) {
                        if (isIntegrated(arr, i, j)) {
                                len = Math.max(len, j - i + 1);
                        }
                }
        }
        return len;
}

public boolean isIntegrated(int[] arr, int left, int right) {
        int[] newArr = Arrays.copyOfRange(arr, left, right + 1); // O(N)
        Arrays.sort(newArr); // O(N*logN)
        for (int i = 1; i < newArr.length; i++) {
                if (newArr[i - 1] != newArr[i] - 1) {
                        return false;
                }
        }
        return true;
}
```

第一种方法严格按照题目的意思来验证每一个子数组是否是可整合数组，但是验证可整合数组真的需要如此麻烦吗？有没有更好的方法来加速验证过程？这也是本书提供方法的核心。判断一个数组是否是可整合数组还可以用以下方法来判断，一个数组中如果没有重复元素，并且如果最大值减去最小值，再加 1 的结果等于元素个数（max-min+1==元素个数），那么这个数组就是可整合数组。比如[3,2,5,6,4]，max-min+1=6-2+1=5==元素个数，所以这个数组是可整合数组。

这样，验证每一个子数组是否是可整合数组的时间复杂度可以从第一种方法的

$O(N\log N)$加速至 $O(1)$，整个过程的时间复杂度就可加速到 $O(N^2)$。具体请参看如下代码中的 getLIL2 方法。

```
public int getLIL2(int[] arr) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        int len = 0;
        int max = 0;
        int min = 0;
        HashSet<Integer> set = new HashSet<Integer>(); // 判断重复
        for (int i = 0; i < arr.length; i++) {
                max = Integer.MIN_VALUE;
                min = Integer.MAX_VALUE;
                for (int j = i; j < arr.length; j++) {
                        if (set.contains(arr[j])) {
                                break;
                        }
                        set.add(arr[j]);
                        max = Math.max(max, arr[j]);
                        min = Math.min(min, arr[j]);
                        if (max - min == j - i) { // 新的检查方式
                                len = Math.max(len, j - i + 1);
                        }
                }
                set.clear();
        }
        return len;
}
```

不重复打印排序数组中相加和为给定值的所有二元组和三元组

【题目】

给定排序数组 arr 和整数 k，不重复打印 arr 中所有相加和为 k 的不降序二元组。

例如，arr=[-8,-4,-3,0,1,2,4,5,8,9]，k=10，打印结果为：

1,9

2,8

【补充题目】

给定排序数组 arr 和整数 k，不重复打印 arr 中所有相加和为 k 的不降序三元组。

例如，arr=[-8,-4,-3,0,1,2,4,5,8,9]，k=10，打印结果为：

-4,5,9

-3,4,9

-3,5,8

0,1,9

0,2,8

1,4,5

【难度】

尉 ★★☆☆

【解答】

利用排序后的数组的特点，打印二元组的过程可以用一个左指针和一个右指针不断向中间压缩的方式实现，具体过程为：

1．设置变量 left=0，right=arr.length-1。

2．比较 arr[left]+arr[right]的值(sum)与 *k* 的大小：

- 如果 sum 等于 *k*，打印“arr[left],arr[right]”，则 left++，right--。
- 如果 sum 大于 *k*，right--。
- 如果 sum 小于 *k*，left++。

3．如果 left<right，则一直重复步骤 2，否则过程结束。

那么如何保证不重复打印相同的二元组呢？只需在打印时增加一个检查即可，检查 arr[left]是否与它前一个值 arr[left-1]相等，如果相等就不打印。具体解释为：因为整体过程是从两头向中间压缩的过程，如果 arr[left]+arr[right]==k，又有 arr[left]==arr[left-1]，那么之前一定已经打印过这个二元组，此时无须重复打印。比如 arr=[1,1,1,9]，k=10。首先打印 arr[0]和 arr[3]的组合，接下来就不再重复打印 1 和 9 这个二元组。

具体过程请参看如下代码中的 printUniquePair 方法，时间复杂度 *O*(*N*)。

```
public void printUniquePair(int[] arr, int k) {
        if (arr == null || arr.length < 2) {
                return;
        }
        int left = 0;
        int right = arr.length - 1;
        while (left < right) {
```

```
                if (arr[left] + arr[right] < k) {
                        left++;
                } else if (arr[left] + arr[right] > k) {
                        right--;
                } else {
                        if (left == 0 || arr[left - 1] != arr[left]) {
                            System.out.println(arr[left] + "," + arr[right]);
                        }
                        left++;
                        right--;
                }
            }
    }
```

三元组的问题类似于二元组的求解过程。

例如：

arr=[-8,-4,-3,0,1,2,4,5,8,9]，k=10。

- 当三元组的第一个值为-8 时，寻找-8 后面的子数组中所有相加为 18 的不重复二元组。
- 当三元组的第一个值为-4 时，寻找-4 后面的子数组中所有相加为 14 的不重复二元组。
- 当三元组的第一个值为-3 时，寻找-3 后面的子数组中所有相加为 13 的不重复二元组。

依此类推。

如何不重复打印相同的三元组呢？首先要保证每次寻找过程开始前，选定的三元组中第一个值不重复，其次就是和原问题的打印检查一样，要保证不重复打印二元组。

具体请参看如下代码中的 printUniqueTriad 方法，时间复杂度为 $O(N^2)$。

```
public void printUniqueTriad(int[] arr, int k) {
        if (arr == null || arr.length < 3) {
            return;
        }
        for (int i = 0; i < arr.length - 2; i++) {
            if (i == 0 || arr[i] != arr[i - 1]) {
                    printRest(arr, i, i + 1, arr.length - 1, k - arr[i]);
            }
        }
}

public void printRest(int[] arr, int f, int l, int r, int k) {
        while (l < r) {
            if (arr[l] + arr[r] < k) {
                l++;
```

```
                } else if (arr[l] + arr[r] > k) {
                    r--;
                } else {
                    if (l == f + 1 || arr[l - 1] != arr[l]) {
                        System.out.println(arr[f] + "," + arr[l] + "," + arr[r]);
                    }
                    l++;
                    r--;
                }
            }
        }
```

未排序正数数组中累加和为给定值的最长子数组长度

【题目】

给定一个数组 arr，该数组无序，但每个值均为正数，再给定一个正数 k。求 arr 的所有子数组中所有元素相加和为 k 的最长子数组长度。

例如，arr=[1,2,1,1,1]，k=3。

累加和为 3 的最长子数组为[1,1,1]，所以结果返回 3。

【难度】

尉 ★★☆☆

【解答】

最优解可以做到时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。首先用两个位置来标记子数组的左右两头，记为 left 和 right，开始时都在数组的最左边(left=0,right=0)。整体过程如下：

1．开始时变量 left=0，right=0，代表子数组 arr[left..right]。

2．变量 sum 始终表示子数组 arr[left..right]的和。开始时 sum=arr[0]，即 arr[0..0]的和。

3．变量 len 一直记录累加和为 k 的所有子数组中最大子数组的长度。开始时，len=0。

4．根据 sum 与 k 的比较结果决定是 left 移动还是 right 移动，具体如下：

- 如果 sum==k，说明 arr[left..right]累加和为 k，如果 arr[left..right]长度大于 len，则更新 len，此时因为数组中所有的值都为正数，那么所有从 left 位置开始，在 right 之后的位置结束的子数组，即 arr[left..i(i>right)]，累加和一定大于 k。所以，令 left 加 1，这表示我们开始考查以 left 之后的位置开始的子数组，同时令 sum-=arr[left]，

sum 此时开始表示 arr[left+1..right]的累加和。

- 如果 sum 小于 k，说明 arr[left..right]还需要加上 right 后面的值，其和才可能达到 k，所以，令 right 加 1，sum+=arr[right]。需要注意的是，right 加 1 后是否越界。
- 如果 sum 大于 k，说明所有从 left 位置开始，在 right 之后的位置结束的子数组，即 arr[left..i(i>right)]，累加和一定大于 k。所以，令 left 加 1，这表示我们开始考查以 left 之后的位置开始的子数组，同时令 sum-=arr[left]，sum 此时表示 arr[left+1..right]的累加和。

5．如果 right<arr.length，重复步骤 4。否则直接返回 len，全部过程结束。

具体请参看如下代码中的 getMaxLength 方法。

```
public int getMaxLength(int[] arr, int k) {
        if (arr == null || arr.length == 0 || k <= 0) {
                return 0;
        }
        int left = 0;
        int right = 0;
        int sum = arr[0];
        int len = 0;
        while (right < arr.length) {
                if (sum == k) {
                        len = Math.max(len, right - left + 1);
                        sum -= arr[left++];
                } else if (sum < k) {
                        right++;
                        if (right == arr.length) {
                                break;
                        }
                        sum += arr[right];
                } else {
                        sum -= arr[left++];
                }
        }
        return len;
}
```

未排序数组中累加和为给定值的最长子数组系列问题

【题目】

给定一个无序数组 arr，其中元素可正、可负、可 0，给定一个整数 k。求 arr 所有的子数组中累加和为 k 的最长子数组长度。

【补充题目】

给定一个无序数组 arr，其中元素可正、可负、可 0。求 arr 所有的子数组中正数与负数个数相等的最长子数组长度。

【补充题目】

给定一个无序数组 arr，其中元素只是 1 或 0。求 arr 所有的子数组中 0 和 1 个数相等的最长子数组长度。

【难度】

尉★★☆☆

【解答】

本书提供的方法可以做到时间复杂度为 $O(N)$、额外空间复杂度为 $O(N)$，首先来看原问题。

为了说明解法，先定义 s 的概念，s(i)代表子数组 arr[0..i]所有元素的累加和。那么子数组 arr[j..i](0<=j<=i<arr.length)的累加和为 s(i)-s(j-1)，因为根据定义，s(i)=arr[0..i]的累加和=arr[0..j-1]的累加和+arr[j..i]的累加和，又有 arr[0..j-1]的累加和为 s(j-1)。所以，arr[j..i]的累加和为 s(i)-s(j-1)，这个结论是求解这道题的核心。

原问题解法只遍历一次 arr，具体过程为：

1. 设置变量 sum=0，表示从 0 位置开始一直加到 i 位置所有元素的和。设置变量 len=0，表示累加和为 k 的最长子数组长度。设置哈希表 map，其中，key 表示从 arr 最左边开始累加的过程中出现过的 sum 值，对应的 value 值则表示 sum 值最早出现的位置。

2．从左到右开始遍历，遍历的当前元素为 arr[i]。

1）令 sum=sum+arr[i]，即之前所有元素的累加和 s(i)，在 map 中查看是否存在 sum-k。

- 如果 sum-k 存在，从 map 中取出 sum-k 对应的 value 值，记为 j，j 代表从左到右不断累加的过程中第一次加出 sum-k 这个累加和的位置。根据之前得出的结论，arr[j+1..i]的累加和为 s(i)-s(j)，此时 s(i)=sum，又有 s(j)=sum-k，所以 arr[j+1..i]的累加和为 k。同时因为 map 中只记录每一个累加和最早出现的位置，所以此时的 arr[j+1..i]是在必须以 arr[i]结尾的所有子数组中，最长的累加和为 k 的子数组，如果该子数组的长度大于 len，就更新 len。

- 如果 sum-k 不存在，说明在必须以 arr[i]结尾的情况下没有累加和为 k 的子数组。

2）检查当前的 sum（即 s(i)）是否在 map 中。如果不存在，说明此时的 sum 值是第一次出现的，就把记录(sum,i)加入到 map 中。如果 sum 存在，说明之前已经出现过 sum，map 只记录一个累加和最早出现的位置，所以此时什么记录也不加。

3．继续遍历下一个元素，直到所有的元素遍历完。

大体过程如上，但还有一个很重要的问题需要处理。根据 arr[j+1..i]的累加和为 s(i)-(j)，所以，如果从 0 位置开始累加，会导致 j+1>=1。也就是说，所有从 0 位置开始的子数组都没有考虑过。所以，应该从-1 位置开始累加，也就是在遍历之前先把(0,-1)这个记录放进 map，这个记录的意义是如果任何一个数也不加时，累加和为 0。这样，从 0 位置开始的子数组就被我们考虑到了。

比如，数组[1,2,3,3]，k=6。如果从 0 位置开始累加，也就是遍历之前不加入(0,-1)记录，当遍历到第一个 3 时，sum=6，在 map 中的记录是：

key	value
1	0 -> 累加和 1 最早出现在 0 位置
3	1 -> 累加和 3 最早出现在 1 位置

此时 sum-k=6-6=0，所以在 map 中查询累加和 0 最早出现的位置，发现没有出现过。那么子数组[1,2,3]就被我们忽略。接下来遍历到第二个 3 时，sum=9，在 map 中的记录是：

key	value
1	0 -> 累加和 1 最早出现在 0 位置
3	1 -> 累加和 3 最早出现在 1 位置
6	2 -> 累加和 2 最早出现在 2 位置

此时 sum-k=9-6=3，所以在 map 中查询累加和 3 最早出现的位置，发现累加和 3 最早出现在 1 位置，所以 arr[j+1..i]即 arr[2..3]（也即[3,3]）被找到。但很明显，[1,2,3]这个子数组才是正确的，所以不加入(0,-1)会导致这样的问题。

如果遍历之前先加入(0,-1)这个记录，当遍历到第一个 3 时，sum=6，在 map 中的记录是：

key	value
0	-1 -> 累加和 0 最早出现在-1 位置，即一个元素也没有时，累加和为 0
10	-> 累加和 1 最早出现在 0 位置
3	1 -> 累加和 3 最早出现在 1 位置

此时 sum-k=6-6=0，所以，在 map 中查询累加和 0 最早出现的位置，发现累加和 0 最早出现在-1 位置，所以 arr[j+1..i]即 arr[0..2]（也即[1,2,3]）被找到。

具体过程请参看如下代码中的 maxLength 方法。

```
public int maxLength(int[] arr, int k) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        map.put(0, -1); // 重要
        int len = 0;
        int sum = 0;
        for (int i = 0; i < arr.length; i++) {
                sum += arr[i];
                if (map.containsKey(sum - k)) {
                        len = Math.max(i - map.get(sum - k), len);
                }
                if (!map.containsKey(sum)) {
                        map.put(sum, i);
                }
        }
        return len;
}
```

理解了原问题的解法后，补充问题是可以迅速解决的。第一个补充问题，先把数组 arr 中的正数全部变成 1，负数全部变成-1，0 不变，然后求累加和为 0 的最长子数组长度即可。第二个补充问题，先把数组 arr 中的 0 全部变成-1，1 不变，然后求累加和为 0 的最长子数组长度即可。两个补充问题的代码略。

未排序数组中累加和小于或等于给定值的最长子数组长度

【题目】

给定一个无序数组 arr，其中元素可正、可负、可 0，给定一个整数 *k*。求 arr 所有的子数组中累加和小于或等于 *k* 的最长子数组长度。

例如：arr=[3,-2,-4,0,6]，*k*=-2，相加和小于或等于-2 的最长子数组为{3,-2,-4,0}，所以结果返回 4。

【难度】

校 ★★★☆

【解答】

本书提供的方法可以做到时间复杂度为 $O(N\log N)$，额外空间复杂度为 $O(N)$。

依次求以数组的每个位置结尾的、累加和小于或等于 k 的最长子数组长度，其中最长的那个子数组的长度就是我们要的结果。为了便于读者理解，我们举一个比较具体的例子。

假设我们处理到位置 30，从位置 0 到位置 30 的累加和为 100（sum[0..30]=100），现在想求以位置 30 结尾的、累加和小于或等于 10 的最长子数组长度。再假设从位置 0 开始累加到位置 10 的时候，累加和第一次大于或等于 90（sum[0..10]>=90），那么可以知道以位置 30 结尾的相加和小于或等于 10 的最长子数组就是 arr[11..30]。也就是说，如果从 0 位置到 j 位置的累加和为 sum[0..j]，此时想求以 j 位置结尾的相加和小于或等于 k 的最长子数组长度。那么只要知道大于或等于 sum[0..j]-k 这个值的累加和最早出现在 j 之前的什么位置就可以，假设那个位置是 i 位置，那么 arr[i+1..j]就是在 j 位置结尾的相加和小于或等于 k 的最长子数组。

为了很方便地找到大于或等于某一个值的累加和最早出现的位置，可以按照如下方法生成辅助数组 helpArr。

1．首先生成 arr 每个位置从左到右的累加和数组 sumArr。以[1,2,-1,5,-2]为例，生成的 sumArr=[0,1,3,2,7,5]。注意，sumArr 中的第一个数为 0，表示当没有任何一个数时的累加和为 0。

2．生成 sumArr 的左侧最大值数组 helpArr，sumArr={0,1,3,2,7,5} -> helpArr={0,1,3,3,7,7}。为什么原来的 sumArr 数组中的 2 和 5 变为 3 和 7 呢？因为我们只关心大于或等于某一个值的累加和最早出现的位置，而累加和 3 出现在 2 之前，并且大于或等于 3 必然大于 2。所以，当然要保留一个更大的、出现更早的累加和。

3．helpArr 是 sumArr 每个位置上的左侧最大值数组，那么它当然是有序的。在这样一个有序的数组中，就可以二分查找大于或等于某一个值的累加和最早出现的位置。例如，在[0,1,3,3,7,7]中查找大于或等于 4 这个值的位置，就是第一个 7 的位置。

以原题中给的例子来说明整个计算过程。

arr = [3,-2,-4,0,6]，k = -2。

1．arr=[3,-2,-4,0,6]，求得 arr 的累加数组 sumArr=[0,3,1,-3,-3,3]，进一步求得 sumArr 的左侧最大值数组[0,3,3,3,3,3]。

2．j=0 时，sum[0..0]=3，所以在 helpArr 中二分查找大于或等于 3-k=3-(-2)=5 这个值第一次出现的位置，结果是没有。所以，可知以位置 0 结尾的所有子数组累加后没有小于或

等于 k（即-2）的。

3．*j*=1 时，sum[0..1]=1，所以在 helpArr 中二分查找大于或等于 1-*k*=1-(-2)=3 这个值第一次出现的位置，在 helpArr 中的位置是 1，对应的 arr 中的位置是 0，所以，arr[1..1]是满足条件的最长数组。

4．*j*=2 时，sum[0..2]=-3，所以在 helpArr 中二分查找大于或等于-3-*k*=-3-(-2)=-1 这个值第一次出现的位置，在 helpArr 中的位置是 0，对应的 arr 中的位置是-1，表示一个数都不累加的情况，所以 arr[0..2]是满足条件的最长数组。

5．*j*=3 时，sum[0..3]=-3，所以在 helpArr 中二分查找大于或等于-3-*k*=-3-(-2)=-1 这个值第一次出现的位置，在 helpArr 中的位置是 0，对应的 arr 中的位置是-1，表示一个数都不累加的情况，所以 arr[0..3]是满足条件的最长数组。

6．*j*=4 时，sum[0..4]=3，所以在 helpArr 中二分查找大于或等于 3-*k*=3-(-2)=5 这个值第一次出现的位置，结果是没有。所以，可知以位置 4 结尾的所有子数组累加后没有小于或等于 *k*（即-2）的。

全部过程请参看如下代码中的 maxLength 方法。

```
public int maxLength(int[] arr, int k) {
        int[] h = new int[arr.length + 1];
        int sum = 0;
        h[0] = sum;
        for (int i = 0; i != arr.length; i++) {
                sum += arr[i];
                h[i + 1] = Math.max(sum, h[i]);
        }
        sum = 0;
        int res = 0;
        int pre = 0;
        int len = 0;
        for (int i = 0; i != arr.length; i++) {
                sum += arr[i];
                pre = getLessIndex(h, sum - k);
                len = pre == -1 ? 0 : i - pre + 1;
                res = Math.max(res, len);
        }
        return res;
}

public int getLessIndex(int[] arr, int num) {
        int low = 0;
        int high = arr.length - 1;
        int mid = 0;
        int res = -1;
        while (low <= high) {
                mid = (low + high) / 2;
```

```
                if (arr[mid] >= num) {
                        res = mid;
                        high = mid - 1;
                } else {
                        low = mid + 1;
                }
        }
        return res;
}
```

计算数组的小和

【题目】

数组小和的定义如下：

例如，数组 s=[1,3,5,2,4,6]，在 s[0]的左边小于或等于 s[0]的数的和为 0，在 s[1]的左边小于或等于 s[1]的数的和为 1，在 s[2]的左边小于或等于 s[2]的数的和为 1+3=4，在 s[3]的左边小于或等于 s[3]的数的和为 1，在 s[4]的左边小于或等于 s[4]的数的和为 1+3+2=6，在 s[5]的左边小于或等于 s[5]的数的和为 1+3+5+2+4=15，所以 s 的小和为 0+1+4+1+6+15=27。

给定一个数组 s，实现函数返回 s 的小和。

【难度】

校 ★★★☆

【解答】

用时间复杂度为 $O(N^2)$的方法比较简单，按照题目例子描述的求小和的方法求解即可，本书不再详述。下面介绍一种时间复杂度为 $O(N\log N)$、额外空间复杂度为 $O(N)$的方法，这是一种在归并排序的过程中，利用组间在进行合并时产生小和的过程。

1．假设左组为 l[]，右组为 r[]，左右两个组的组内都已经有序，现在要利用外排序合并成一个大组，并假设当前外排序是 l[i]与 r[j]在进行比较。

2．如果 l[i]<=r[j]，那么产生小和。假设从 r[j]往右一直到 r[]结束，元素的个数为 m，那么产生的小和为 l[i]*m。

3．如果 l[i]>r[j]，不产生任何小和。

4．整个归并排序的过程该怎么进行就怎么进行，排序过程没有任何变化，只是利用步

骤 1~步骤 3，也就是在组间合并的过程中累加所有产生的小和，总共的累加和就是结果。

还是以题目的例子来说明计算过程。

1．归并排序的过程中会进行拆组再合并的过程。[1,3,5,2,4,6]拆分成左组[1,3,5]和右组[2,4,6]，[1,3,5]再拆分成[1,3]和[5]，[2,4,6]再拆分成[2,4]和[6]，[1,3]再拆分成[1]和[3]，[2,4]再拆分成[2]和[4]，如图 8-1 所示。

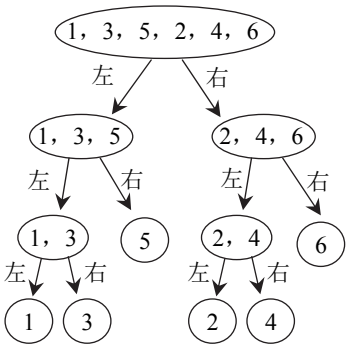

图 8-1

2．[1]与[3]合并。1 和 3 比较，左组的数小，右组从 3 开始到最后一共只有 1 个数，所以产生小和为 1×1=1，合并为[1,3]。

3．[1,3]与[5]合并。1 和 5 比较，左组的数小，右组从 5 开始到最后一共只有 1 个数，所以产生小和为 1×1=1。同理，3 和 5 比较，产生小和为 3×1=3，合并为[1,3,5]。

4．[2]与[4]合并。2 和 4 比较，左组的数小，右组从 4 开始到最后一共只有 1 个数，所以产生小和为 2×1=2，合并为[2,4]。

5．[2,4]与[6]合并。与步骤 3 同理，产生小和为 6，合并为[2,4,6]。

6．[1,3,5]与[2,4,6]合并。1 和 2 比较，左组的数小，右组从 2 开始到最后一共有 3 个数，所以产生小和为 1×3=3。3 和 2 比较，右组的数小，不产生小和。3 和 4 比较，左组的数小，右组从 4 开始到最后一共有 2 个数，所以产生小和为 3×2=6。5 和 4 比较，右组的数小，不产生小和。5 和 6 比较，左组的数小，右组从 6 开始到最后一共有 1 个数，所以产生小和为 5，合并为[1,2,3,4,5,6]。

7．归并过程结束，总的小和为 1+1+3+2+6+3+6+5=27。合并的全部过程如图 8-2 所示。

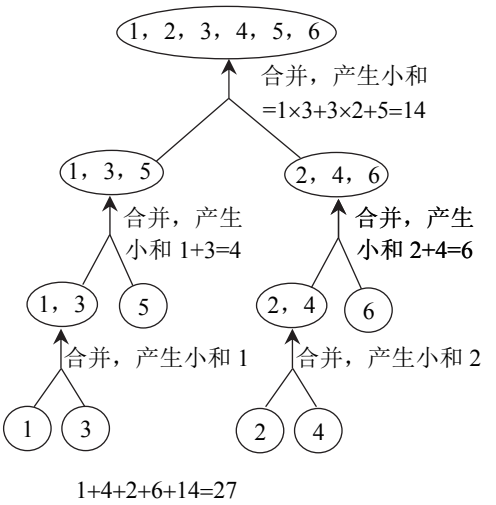

图 8-2

在归并排序中，尤其是在组与组之间进行外排序合并的过程中，按照如上方式把小和一点一点地“榨”出来，最后收集到所有的小和。具体过程请参看如下代码中的 getSmallSum 方法。

```
public int getSmallSum(int[] arr) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        return func(arr, 0, arr.length - 1);
}

public int func(int[] s, int l, int r) {
        if (l == r) {
                return 0;
        }
        int mid = (l + r) / 2;
        return func(s, l, mid) + func(s, mid + 1, r) + merge(s, l, mid, r);
}

public int merge(int[] s, int left, int mid, int right) {
        int[] h = new int[right - left + 1];
        int hi = 0;
        int i = left;
        int j = mid + 1;
        int smallSum = 0;
        while (i <= mid && j <= right) {
                if (s[i] <= s[j]) {
```

```
                        smallSum += s[i] * (right - j + 1);
                        h[hi++] = s[i++];
                } else {
                        h[hi++] = s[j++];
                }
        }
        for (; (j < right + 1) || (i < mid + 1); j++, i++) {
                h[hi++] = i > mid ? s[j] : s[i];
        }
        for (int k = 0; k != h.length; k++) {
                s[left++] = h[k];
        }
        return smallSum;
}
```

自然数数组的排序

【题目】

给定一个长度为 N 的整型数组 arr，其中有 N 个互不相等的自然数 1～N，请实现 arr 的排序，但是不要把下标 0～N-1 位置上的数通过直接赋值的方式替换成 1～N。

【要求】

时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

arr 在调整之后应该是下标从 0 到 N-1 的位置上依次放着 1~N，即 arr[index]=index+1。

本书提供两种实现方法，先介绍方法一：

1．从左到右遍历 arr，假设当前遍历到 i 位置。

2．如果 arr[i]==i+1，说明当前的位置不需要调整，继续遍历下一个位置。

3．如果 arr[i]!=i+1，说明此时 i 位置的数 arr[i]不应该放在 i 位置上，接下来将进行跳的过程。

举例来说明，比如[1,2,5,3,4]，假设遍历到位置 2，也就是 5 这个数。5 应该放在位置 4 上，所以把 5 放过去，数组变成[1,2,5,3,5]。同时，4 这个数是被 5 替下来的数，应该放在位置 3，所以把 4 放过去，数组变成[1,2,5,4,5]。同时 3 这个数是被 4 替下来的数，应该放

在位置 2，所以把 3 放过去，数组变成[1,2,3,4,5]。当跳了一圈回到原位置后，会发现此时 arr[i]==i+1，继续遍历下一个位置。

方法一的具体过程请参看如下代码中的 sort1 方法。

```
public void sort1(int[] arr) {
        int tmp = 0;
        int next = 0;
        for (int i = 0; i != arr.length; i++) {
                tmp = arr[i];
                while (arr[i] != i + 1) {
                        next = arr[tmp - 1];
                        arr[tmp - 1] = tmp;
                        tmp = next;
                }
        }
}
```

下面介绍方法二：

1．从左到右遍历 arr，假设当前遍历到 *i* 位置。

2．如果 arr[i]==i+1，说明当前的位置不需要调整，继续遍历下一个位置。

3．如果 arr[i]!=i+1，说明此时 *i* 位置的数 arr[i]不应该放在 *i* 位置上，接下来将在 *i* 位置进行交换过程。

比如[1,2,5,3,4]，假设遍历到位置 2，也就是 5 这个数。5 应该放在位置 4 上，所以位置 4 上的数 4 和 5 交换，数组变成[1,2,4,3,5]。但此时还是 arr[2]!=3，4 这个数应该放在位置 3 上，所以 3 和 4 交换，数组变成[1,2,3,4,5]。此时 arr[2]==3，遍历下一个位置。

方法二的具体过程请参看如下代码中的 sort2 方法。

```
public void sort2(int[] arr) {
        int tmp = 0;
        for (int i = 0; i != arr.length; i++) {
                while (arr[i] != i + 1) {
                        tmp = arr[arr[i] - 1];
                        arr[arr[i] - 1] = arr[i];
                        arr[i] = tmp;
                }
        }
}
```

奇数下标都是奇数或者偶数下标都是偶数

【题目】

给定一个长度不小于 2 的数组 arr，实现一个函数调整 arr，要么让所有的偶数下标都是偶数，要么让所有的奇数下标都是奇数。

【要求】

如果 arr 的长度为 N，函数要求时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

实现方法有很多，本书介绍一种易于实现的方法，步骤如下：

1．设置变量 even，表示目前 arr 最左边的偶数下标，初始时 even=0。

2．设置变量 odd，表示目前 arr 最左边的奇数下标，初始时 odd=1。

3．不断检查 arr 的最后一个数，即 arr[N-1]。如果 arr[N-1]是偶数，交换 arr[N-1]和 arr[even]，然后令 even=even+2。如果 arr[N-1]是奇数，交换 arr[N-1]和 arr[odd]，然后令 odd=odd+2。继续重复步骤 3。

4．如果 even 或者 odd 大于或等于 N，过程停止。

举例说明整个过程。比如[1,8,3,2,4,6]，当前最后一个数记为 end=6，even=0，odd=1。此时 end=6 为偶数，所以 6 和 arr[even=0]交换，数组变成[6,8,3,2,4,1]，even=even+2=2。此时 end=1 为奇数，所以 1 和 arr[odd=1]交换，数组变成[6,1,3,2,4,8]，odd=odd+2=3。此时 end=8 为偶数，所以 8 和 arr[even=2]交换，数组变成[6,1,8,2,4,3]，even=even+2=4。此时 end=3 为奇数，所以 3 和 arr[odd=3]交换，数组变成[6,1,8,3,4,2]，odd=odd+2=5。此时 end=2 为偶数，所以 2 和 arr[odd=4]交换，数组变成[6,1,8,3,2,4]，even=even+2=6。此时 even 大于或等于长度 6，说明偶数下标已经都是偶数，过程停止。

再解释得直白一点，最后位置的数是偶数，就向偶数下标发送，最后位置的数是奇数，就向奇数下标发送，如果偶数下标或者奇数下标已经无法再向右移动，说明调整结束。调

整的全部过程请参看如下代码中的 modify 方法。

```
public void modify(int[] arr) {
        if (arr == null || arr.length < 2) {
                return;
        }
        int even = 0;
        int odd = 1;
        int end = arr.length - 1;
        while (even <= end && odd <= end) {
                if ((arr[end] & 1) == 0) {
                        swap(arr, end, even);
                        even += 2;
                } else {
                        swap(arr, end, odd);
                        odd += 2;
                }
        }
}

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

子数组的最大累加和问题

【题目】

给定一个数组 arr，返回子数组的最大累加和。

例如，arr=[1,-2,3,5,-2,6,-1]，所有的子数组中，[3,5,-2,6]可以累加出最大的和 12，所以返回 12。

【要求】

如果 arr 长度为 N，要求时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

如果 arr 中没有正数，产生的最大累加和一定是数组中的最大值。

如果 arr 中有正数，从左到右遍历 arr，用变量 cur 记录每一步的累加和，遍历到正数 cur 增加，遍历到负数 cur 减少。当 cur<0 时，说明累加到当前数出现了小于 0 的结果，那么累加的这一部分肯定不能作为产生最大累加和的子数组的左边部分，此时令 cur=0，表示重新从下一个数开始累加。当 cur>=0 时，每一次累加都可能是最大的累加和，所以，用另外一个变量 max 全程跟踪记录 cur 出现的最大值即可。

举例来说明一下，arr=[1,-2,3,5,-2,6,-1]，开始时，max=极小值，cur=0。

遍历到 1，cur=cur+1=1，max 更新成 1。遍历到-2，cur=cur-2=-1，开始出现负的累加和，所以，说明[1,-2]这一部分肯定不会作为产生最大累加和的子数组的左边部分，于是令 cur=0，max 不变。遍历到 3，cur=cur+3=3，max 更新成 3。遍历到 5，cur=cur+5=8，max 更新成 8。遍历到-2，cur=cur-2=6，虽然累加了一个负数，但是 cur 依然大于 0，说明累加的这一部分（也就是[3,5,-2]）仍可能作为最大累加和的子数组的左边部分。max 不更新。遍历到 6，cur=cur+6=12，max 更新成 12。遍历到-1，cur=cur-1=11，max 不更新。最后返回 12。解释得再直白一点，cur 累加成为负数就清零重新累加，max 记录 cur 的最大值即可。

求解最大累加和具体过程请参看如下代码中的 maxSum 方法。

```
public int maxSum(int[] arr) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        int max = Integer.MIN_VALUE;
        int cur = 0;
        for (int i = 0; i != arr.length; i++) {
                cur += arr[i];
                max = Math.max(max, cur);
                cur = cur < 0 ? 0 : cur;
        }
        return max;
}
```

子矩阵的最大累加和问题

【题目】

给定一个矩阵 matrix，其中的值有正、有负、有 0，返回子矩阵的最大累加和。

例如，矩阵 matrix 为：

-90　48　78

64　-40　64

-81　-7　66

其中，最大累加和的子矩阵为：

48　78

-40　64

-7　66

所以返回累加和 209。

例如，matrix 为：

-1　-1　-1

-1　2　2

-1　-1　-1

其中，最大累加和的子矩阵为：

2　2

所以返回累加和 4。

【难度】

尉 ★★☆☆

【解答】

在阅读本题的解释之前，请先阅读上一道题“子数组的最大累加和问题”，因为本题的最优解深度利用了上一题的解法。首先来看这样一个例子，假设一个 2 行 4 列的矩阵如下：

-2　3　-5　7

1　4　-1　-3

如何求必须含有 2 行元素的子矩阵中的最大累加和？可以把两列的元素累加，然后得到累加数组[-1,7,-6,4]，接下来求这个累加数组的最大累加和，结果是 7。也就是说，必须含有 2 行元素的子矩阵中的最大和为 7，且这个子矩阵是：

3

4

也就是说，如果一个矩阵一共有 k 行且限定必须含有 k 行元素的情况下，我们只要把矩阵中每一列的 k 个元素累加生成一个累加数组，然后求出这个数组的最大累加和，这个

最大累加和就是必须含有 *k* 行元素的子矩阵中的最大累加和。

请读者务必理解以上解释，下面看原问题如何求解。为了方便讲述，我们用题目的第一个例子来展示求解过程，首先考虑只有一行的矩阵[-90,48,78]，因为只有一行，所以累加数组 arr 就是[-90,48,78]，这个数组的最大累加和为 126。

接下来考虑含有两行的矩阵：

-90　48　78

64　-40　64

这个矩阵的累加数组就是在上一步的累加数组[-90,48,78]的基础上，依次在每个位置上加上矩阵最新一行[64, -40, 64]的结果，即[-26,8,142]，这个数组的最大累加和为 150。

接下来考虑含有三行的矩阵：

-90　48　78

64　-40　64

-81　-7　66

这个矩阵的累加数组就是在上一步累加数组[-26,8,142]的基础上，依次在每个位置上加上矩阵最新一行[-81,-7,66]的结果，即[-107,1,208]，这个数组的最大累加和为 209。

此时，必须从矩阵的第一行元素开始，并往下的所有子矩阵已经查找完毕，接下来从矩阵的第二行开始，继续这样的过程，含有一行矩阵：

64 -40　64

因为只有一行，所以累加数组就是[64,-40,64]，这个数组的最大累加和为 88。

接下来考虑含有两行的矩阵：

64　-40　64

-81　-7　66

这个矩阵的累加数组就是在上一步累加数组[64,-40,64]的基础上，依次在每个位置上加上矩阵最新一行[-81,-7,66]的结果，即[-17,-47,130]，这个数组的最大累加和为 130。

此时，必须从矩阵的第二行元素开始，并往下的所有子矩阵已经查找完毕，接下来从矩阵的第三行开始，继续这样的过程，含有一行矩阵：

-81　-7　66

因为只有一行，所以累加数组就是[-81,-7,66]，这个数组的最大累加和为 66。

全部过程结束，所有的子矩阵都已经考虑到了，结果为以上所有最大累加和中最大的 209。

整个过程最关键的地方有两处：

- 用求累加数组的最大累加和的方式得到每一步的最大子矩阵的累加和。
- 每一步的累加数组可以利用前一步求出的累加数组很方便地更新得到。

如果矩阵大小为 $N \times N$ 的，以上全部过程的时间复杂度为 $O(N^3)$，具体请参看如下代码中的 maxSum 方法。

```
public int maxSum(int[][] m) {
        if (m == null || m.length == 0 || m[0].length == 0) {
                return 0;
        }
        int max = Integer.MIN_VALUE;
        int cur = 0;
        int[] s = null; // 累加数组
        for (int i = 0; i != m.length; i++) {
                s = new int[m[0].length];
                for (int j = i; j != m.length; j++) {
                        cur = 0;
                        for (int k = 0; k != s.length; k++) {
                                s[k] += m[j][k];
                                cur += s[k];
                                max = Math.max(max, cur);
                                cur = cur < 0 ? 0 : cur;
                        }
                }
        }
        return max;
}
```

在数组中找到一个局部最小的位置

【题目】

定义局部最小的概念。arr 长度为 1 时，arr[0]是局部最小。arr 的长度为 $N(N>1)$时，如果 arr[0]<arr[1]，那么 arr[0]是局部最小；如果 arr[N-1]<arr[N-2]，那么 arr[N-1]是局部最小；如果 $0<i<N-1$，既有 arr[i]<arr[i-1]，又有 arr[i]<arr[i+1]，那么 arr[i]是局部最小。

给定无序数组 arr，已知 arr 中任意两个相邻的数都不相等。写一个函数，只需返回 arr 中任意一个局部最小出现的位置即可。

【难度】

尉 ★★☆☆

【解答】

本题可以利用二分查找做到时间复杂度为 $O(\log N)$、额外空间复杂度为 $O(1)$，步骤如下：

1．如果 arr 为空或者长度为 0，返回-1 表示不存在局部最小。

2．如果 arr 长度为 1 或者 arr[0]<arr[1]，说明 arr[0]是局部最小，返回 0。

3．如果 arr[N-1]<arr[N-2]，说明 arr[N-1]是局部最小，返回 N-1。

4．如果 arr 长度大于 2 且 arr 的左右两头都不是局部最小，则令 left=1，right=N-2，然后进入步骤 5 做二分查找。

5．令 mid=(left+right)/2，然后进行如下判断：

1）如果 arr[mid]>arr[mid-1]，可知在 arr[left..mid-1]上肯定存在局部最小，令 right=mid-1，重复步骤 5。

2）如果不满足 1)，但 arr[mid]>arr[mid+1]，可知在 arr[mid+1..right]上肯定存在局部最小，令 left=mid+1，重复步骤 5。

3）如果既不满足 1)，也不满足 2)，那么 arr[mid]就是局部最小，直接返回 mid。

6．步骤 5 一直进行二分查找，直到 left==right 时停止，返回 left 即可。

如此可见，二分查找并不是数组有序时才能使用，只要你能确定二分两侧的某一侧肯定存在你要找的内容，就可以使用二分查找。具体过程请参看如下的 getLessIndex 方法。

```
public int getLessIndex(int[] arr) {
        if (arr == null || arr.length == 0) {
                return -1; // 不存在
        }
        if (arr.length == 1 || arr[0] < arr[1]) {
                return 0;
        }
        if (arr[arr.length - 1] < arr[arr.length - 2]) {
                return arr.length - 1;
        }
        int left = 1;
        int right = arr.length - 2;
        int mid = 0;
        while (left < right) {
                mid = (left + right) / 2;
                if (arr[mid] > arr[mid - 1]) {
                        right = mid - 1;
                } else if (arr[mid] > arr[mid + 1]) {
                        left = mid + 1;
                } else {
                        return mid;
                }
```

```
            }
            return left;
        }
```

数组中子数组的最大累乘积

【题目】

给定一个 double 类型的数组 arr，其中的元素可正、可负、可 0，返回子数组累乘的最大乘积。例如，arr=[-2.5，4，0，3，0.5，8，-1]，子数组[3，0.5，8]累乘可以获得最大的乘积 12，所以返回 12。

【难度】

尉 ★★☆☆

【解答】

本题可以做到时间复杂度为 $O(N)$、额外空间复杂度为 $O(1)$。所有的子数组都会以某一个位置结束，所以，如果求出以每一个位置结尾的子数组最大的累乘积，在这么多最大累乘积中最大的那个就是最终的结果。也就是说，结果=Max{以 arr[0]结尾的所有子数组的最大累乘积，以 arr[1]结尾的所有子数组的最大累乘积……以 arr[arr.length-1]结尾的所有子数组的最大累乘积}。

如何快速求出所有以 i 位置结尾(arr[i])的子数组的最大乘积呢？假设以 arr[i-1]结尾的最小累乘积为 min，以 arr[i-1]结尾的最大累乘积为 max。那么，以 arr[i]结尾的最大累乘积只有以下三种可能：

- 可能是 max*arr[i]。max 既然表示以 arr[i-1]结尾的最大累乘积，那么当然有可能以 arr[i]结尾的最大累乘积是 max*arr[i]。例如，[3,4,5]在算到 5 的时候。
- 可能是 min*arr[i]。min 既然表示以 arr[i-1]结尾的最小累乘积，当然有可能 min 是负数，而如果 arr[i]也是负数，两个负数相乘的结果也可能很大。例如，[-2,3,-4]在算到-4 的时候。
- 可能仅是 arr[i]的值。以 arr[i]结尾的最大累乘积并不一定非要包含 arr[i]之前的数。例如，[0.1,0.1,100]在算到 100 的时候。

这三种可能的值中最大的那个就作为以 i 位置结尾的最大累乘积，最小的作为最小累

乘积，然后继续计算以 *i*+1 位置结尾的时候，如此重复，直到计算结束。

具体过程请参看如下代码中的 maxProduct 方法。

```
public double maxProduct(double[] arr) {
        if (arr == null || arr.length == 0) {
                return 0;
        }
        double max = arr[0];
        double min = arr[0];
        double res = arr[0];
        double maxEnd = 0;
        double minEnd = 0;
        for (int i = 1; i < arr.length; ++i) {
                maxEnd = max * arr[i];
                minEnd = min * arr[i];
                max = Math.max(Math.max(maxEnd, minEnd), arr[i]);
                min = Math.min(Math.min(maxEnd, minEnd), arr[i]);
                res = Math.max(res, max);
        }
        return res;
}
```

打印 *N* 个数组整体最大的 Top *K*

【题目】

有 *N* 个长度不一的数组，所有的数组都是有序的，请从大到小打印这 *N* 个数组整体最大的前 *K* 个数。

例如，输入含有 *N* 行元素的二维数组可以代表 *N* 个一维数组。

219,405,538,845,971

148,558

52,99,348,691

再输入整数 *k*=5，则打印：

Top 5: 971,845,691,558,538

【要求】

1．如果所有数组的元素个数小于 *K*，则从大到小打印所有的数。

2．要求时间复杂度为 *O*(*K*log*N*)。

【难度】

尉 ★★☆☆

【解答】

本题的解法是利用堆结构和堆排序的过程完成的，具体过程如下：

1．构建一个大小为 *N* 的大根堆 heap，建堆的过程就是把每一个数组中的最后一个值，也就是该数组的最大值，依次加入到堆里，这个过程是建堆时的调整过程(heapInsert)。

2．建好堆之后，此时 heap 堆顶的元素是所有数组的最大值中最大的那个，打印堆顶元素。

3．假设堆顶元素来自 a 数组的 *i* 位置。那么接下来就把堆顶的前一个数（即 a[i-1]）放在 heap 的头部，也就是用 a[i-1]替换原本的堆顶，然后从堆的头部开始调整堆，使其重新变为大根堆（heapify 过程）。

4．这样每次都可以得到一个堆顶元素 max，在打印完成后都经历步骤 3 的调整过程。整体打印 *k* 次，就是从大到小全部的 Top *K*。

5．在重复步骤 3 的过程中，如果 max 来自的那个数组（仍假设是 a 数组）已经没有元素。也就是说，max 已经是 a[0]，再往左没有数了。那么就把 heap 中最后一个元素放在 heap 头部的位置，然后把 heap 的大小减 1(heapSize-1)，最后依然是从堆的头部开始调整堆，使其重新变为大根堆(堆大小减 1 之后的 heapify 过程)。

6，直到打印了 *k* 个数，过程结束。

为了知道每一次的 max 来自什么数组的什么位置，放在堆里的元素是如下的 HeapNode 类：

```
public class HeapNode {
        public int value; // 值是什么
        public int arrNum; // 来自哪个数组
        public int index; // 来自数组的哪个位置

        public HeapNode(int value, int arrNum, int index) {
                this.value = value;
                this.arrNum = arrNum;
                this.index = index;
        }
}
```

整个打印过程请参看如下代码中的 printTopK 方法。

```
public void printTopK(int[][] matrix, int topK) {
```

```
        int heapSize = matrix.length;
        HeapNode[] heap = new HeapNode[heapSize];
        for (int i = 0; i != heapSize; i++) {
                int index = matrix[i].length - 1;
                heap[i] = new HeapNode(matrix[i][index], i, index);
                heapInsert(heap, i);
        }
        System.out.println("TOP " + topK + " : ");
        for (int i = 0; i != topK; i++) {
                if (heapSize == 0) {
                        break;
                }
                System.out.print(heap[0].value + " ");
                if (heap[0].index != 0) {
                        heap[0].value = matrix[heap[0].arrNum][--heap[0].index];
                } else {
                        swap(heap, 0, --heapSize);
                }
                heapify(heap, 0, heapSize);
        }
}

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

public void heapify(HeapNode[] heap, int index, int heapSize) {
        int left = index * 2 + 1;
        int right = index * 2 + 2;
        int largest = index;
        while (left < heapSize) {
            if (heap[left].value > heap[index].value) {
                largest = left;
            }
            if (right < heapSize && heap[right].value > heap[largest].value) {
                largest = right;
            }
            if (largest != index) {
                swap(heap, largest, index);
            } else {
                break;
            }
            index = largest;
            left = index * 2 + 1;
            right = index * 2 + 2;
```

```
        }
    }

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

边界都是 1 的最大正方形大小

【题目】

给定一个 $N\times N$ 的矩阵 matrix，在这个矩阵中，只有 0 和 1 两种值，返回边框全是 1 的最大正方形的边长长度。

例如：

0　1　1　1　1

0　1　0　0　1

0　1　0　0　1

0　1　1　1　1

0　1　0　1　1

其中，边框全是 1 的最大正方形的大小为 4×4，所以返回 4。

【难度】

尉 ★★☆☆

【解答】

先介绍一个比较容易理解的解法：

1．矩阵中一共有 $N\times N$ 个位置。$O(N^2)$

2．对每一个位置都看是否可以成为边长为 N~1 的正方形左上角。比如，对于(0,0)位置，依次检查是否是边长为 5 的正方形左上角，然后检查边长为 4、3 等。$O(N)$

3．如何检查一个位置是否可以成为边长为 N 的正方形的左上角呢？遍历这个边长为 N 的正方形边界看是否只由 1 构成，也就是走过 4 个边的长度(4N)。$O(N)$

所以普通方法总的时间复杂度为 $O(N^2)\times O(N)\times O(N)=O(N^4)$。

本书提供的方法的时间复杂度为 $O(N^3)$，基本过程也是如上三个步骤。但是对于步骤 3，

可以把时间复杂度由 $O(N)$降为 $O(1)$。具体地说，就是能够在 $O(1)$的时间内检查一个位置假设为(i,j)，是否可以作为边长为 a(1<=a<=N)的边界全是 1 的正方形左上角。关键是使用预处理技巧，这也是面试经常使用的技巧之一，下面介绍得到预处理矩阵的过程。

1．预处理过程是根据矩阵 matrix 得到两个矩阵 right 和 down。right[i][j]的值表示从位置(i,j)出发向右，有多少个连续的 1。down[i][j]的值表示从位置(i,j)出发向下有多少个连续的 1。

2．right 和 down 矩阵如何计算？

1）从矩阵的右下角$(n-1,n-1)$位置开始计算，如果 matrix[n-1][n-1]==1，那么，right[n-1][n-1]=1 且 down[n-1][n-1]=1，否则都等于 0。

2）从右下角开始往上计算，即在 matrix 最后一列上计算，位置就表示为$(i,n-1)$。对 right 矩阵来说，最后一列的右边没有内容，所以，如果 matrix[i][n-1]==1，则令 right[i][n-1]=1，否则为 0。对 down 矩阵来说，如果 matrix[i][n-1]==1，因为 down[i+1][n-1]表示包括位置$(i+1,n-1)$在内并往下有多少个连续的 1，所以，如果位置$(i,n-1)$是 1，那么，令 down[i][n-1]=down[i+1][n-1]+1；如果 matrix[i][n-1]==0，则令 down[i][n-1]=0。

3）从右下角开始往左计算，即在 matrix 最后一行上计算，位置可以表示为$(n-1,j)$。对 right 矩阵来说，如果 matrix[n-1][j]==1，因为 right[n-1][j+1]表示包括位置$(n-1,j+1)$在内右边有多少个连续的 1。所以，如果位置$(n-1,j)$是 1，则令 right[n-1][j]==right[n-1][j+1]+1；如果 matrix[n-1][j]==0，则令 right[n-1][j]==0。对 down 矩阵来说，最后一列的下边没有内容，所以，如果 matrix[n-1][j]==1，令 down[n-1][j]=1，否则为 0。

4）计算完步骤 1）～步骤 3）之后，剩下的位置都是既有右，也有下，假设位置表示为(i,j)：

如果 matrix[i][j]==1，则令 right[i][j]=right[i][j+1]+1，down[i][j]=down[i+1][j]+1。

如果 matrix[i][j]==0，则令 right[i][j]==0，down[i][j]==0。

预处理的具体过程请参看如下代码中的 setBorderMap 方法。

得到 right 和 down 矩阵后，如何加速检查过程呢？比如现在想检查一个位置，假设为(i,j)。是否可以作为边长为 a(1<=a<=N)的边界全为 1 的正方形左上角。

1）位置(i,j)的右边和下边连续为 1 的数量必须都大于或等于 a(right[i][j]>=a&&down[i][j]>=a)，否则说明上边界和左边界的 1 不够。

2）位置(i,j)向右跳到位置$(i,j+a-1)$，这个位置是正方形的右上角，那么这个位置的下边连续为 1 的数量也必须大于或等于 a(down[i][j+a-1]>=a)，否则说明右边界的 1 不够。

3）位置(i,j)向下跳到位置$(i+a-1,j)$，这个位置是正方形的左下角，那么这个位置的右边

连续为 1 的数量也必须大于或等于 a(right[i+a-1][j]>=a)，否则说明下边界的 1 不够。

以上三个条件都满足时，就说明位置(*i*,*j*)符合要求，利用 right 和 down 矩阵之后，加速的过程很明显，不需要遍历边长上的所有值了，只看 4 个点即可。

全部过程请参看如下代码中的 getMaxSize 方法。

```
public void setBorderMap(int[][] m, int[][] right, int[][] down) {
        int r = m.length;
        int c = m[0].length;
        if (m[r - 1][c - 1] == 1) {
                right[r - 1][c - 1] = 1;
                down[r - 1][c - 1] = 1;
        }
        for (int i = r - 2; i != -1; i--) {
                if (m[i][c - 1] == 1) {
                        right[i][c - 1] = 1;
                        down[i][c - 1] = down[i + 1][c - 1] + 1;
                }
        }
        for (int i = c - 2; i != -1; i--) {
                if (m[r - 1][i] == 1) {
                        right[r - 1][i] = right[r - 1][i + 1] + 1;
                        down[r - 1][i] = 1;
                }
        }
        for (int i = r - 2; i != -1; i--) {
                for (int j = c - 2; j != -1; j--) {
                        if (m[i][j] == 1) {
                                right[i][j] = right[i][j + 1] + 1;
                                down[i][j] = down[i + 1][j] + 1;
                        }
                }
        }
}

public int getMaxSize(int[][] m) {
        int[][] right = new int[m.length][m[0].length];
        int[][] down = new int[m.length][m[0].length];
        setBorderMap(m, right, down);
        for (int size = Math.min(m.length, m[0].length); size != 0; size--) {
                if (hasSizeOfBorder(size, right, down)) {
                        return size;
                }
        }
        return 0;
}

public boolean hasSizeOfBorder(int size, int[][] right, int[][] down) {
        for (int i = 0; i != right.length - size + 1; i++) {
                for (int j = 0; j != right[0].length - size + 1; j++) {
                        if (right[i][j] >= size && down[i][j] >= size
```

```
                                        && right[i + size - 1][j] >= size
                                        && down[i][j + size - 1] >= size) {
                                return true;
                        }
                }
        }
        return false;
}
```

不包含本位置值的累乘数组

【题目】

给定一个整型数组 arr，返回不包含本位置值的累乘数组。

例如，arr=[2,3,1,4]，返回[12,8,24,6]，即除自己外，其他位置上的累乘。

【要求】

1．时间复杂度为 $O(N)$。

2．除需要返回的结果数组外，额外空间复杂度为 $O(1)$。

【进阶题目】

对时间和空间复杂度的要求不变，而且不可以使用除法。

【难度】

士 ★☆☆☆

【解答】

先介绍可以使用除法的实现，结果数组记为 res，所有数的乘积记为 all。如果数组中不含 0，则设置 res[i]=all/arr[i](0<=i<n)即可。如果数组中有 1 个 0，对唯一的 arr[i]==0 的位置令 res[i]=all，其他位置上的值都是 0 即可。如果数组中 0 的数量大于 1，那么 res 所有位置上的值都是 0。具体过程请参看如下代码中的 product1 方法。

```
public int[] product1(int[] arr) {
        if (arr == null || arr.length < 2) {
                return null;
        }
        int count = 0;
```

```
        int all = 1;
        for (int i = 0; i != arr.length; i++) {
                if (arr[i] != 0) {
                        all *= arr[i];
                } else {
                        count++;
                }
        }
        int[] res = new int[arr.length];
        if (count == 0) {
                for (int i = 0; i != arr.length; i++) {
                        res[i] = all / arr[i];
                }
        }
        if (count == 1) {
                for (int i = 0; i != arr.length; i++) {
                        if (arr[i] == 0) {
                                res[i] = all;
                        }
                }
        }
        return res;
}
```

不能使用除法的情况下，可以用以下方法实现进阶问题：

1．生成两个长度和 arr 一样的新数组 lr[]和 rl[]。lr[]表示从左到右的累乘（即 lr[i]=arr[0..i]）的累乘。rl 表示从右到左的累乘（即 rl[i]=arr[i..N-1]）的累乘。

2．一个位置上除去自己值的累乘，就是自己左边的累乘再乘以自己右边的累乘，即 res[i]=lr[i-1]*rl[i+1]。

3．最左的位置和最右位置的累乘比较特殊，即 res[0]=rl[1]，res[N-1]=lr[N-2]。

以上思路虽然可以得到结果 res，但是除 res 之外，又使用了两个额外数组，怎么省掉这两个额外数组呢？可以通过 res 数组复用的方式。也就是说，先把 res 数组作为辅助计算的数组，然后把 res 调整成结果数组返回。具体过程请参看如下代码中的 product2 方法。

```
public static int[] product2(int[] arr) {
        if (arr == null || arr.length < 2) {
                return null;
        }
        int[] res = new int[arr.length];
        res[0] = arr[0];
        for (int i = 1; i < arr.length; i++) {
                res[i] = res[i - 1] * arr[i];
        }
        int tmp = 1;
        for (int i = arr.length - 1; i > 0; i--) {
                res[i] = res[i - 1] * tmp;
```

```
                tmp *= arr[i];
            }
            res[0] = tmp;
            return res;
        }
```

数组的 partition 调整

【题目】

给定一个有序数组 arr，调整 arr 使得这个数组的左半部分没有重复元素且升序，而不用保证右部分是否有序。

例如，arr=[1,2,2,2,3,3,4,5,6,6,7,7,8,8,8,9]，调整之后 arr=[1,2,3,4,5,6,7,8,9,…]。

【补充题目】

给定一个数组 arr，其中只可能含有 0、1、2 三个值，请实现 arr 的排序。

另一种问法为：有一个数组，其中只有红球、蓝球和黄球，请实现红球全放在数组的左边，蓝球放在中间，黄球放在右边。

另一种问法为：有一个数组，再给定一个值 *k*，请实现比 *k* 小的数都放在数组的左边，等于 k 的数都放在数组的中间，比 *k* 大的数都放在数组的右边。

【要求】

1．所有题目实现的时间复杂度为 $O(N)$。

2．所有题目实现的额外空间复杂度为 $O(1)$。

【难度】

士 ★☆☆☆

【解答】

先来介绍原问题的解法：

1．生成变量 *u*，含义是在 arr[0..u]上都是无重复元素且升序的。也就是说，*u* 是这个区域最后的位置，初始时 *u*=0，这个区域记为 A。

2．生成变量 *i*，利用 *i* 做从左到右的遍历，在 arr[u+1..i]上是不保证没有重复元素且升

序的区域，*i* 是这个区域最后的位置，初始时 *i*=1，这个区域记为 B。

3．*i* 向右移动(*i*++)。因为数组整体有序，所以如果 arr[i]!=arr[u]，说明当前数 arr[i]应该加入到 A 区域里，所以交换 arr[u+1]和 arr[i]，此时 A 的区域增加一个数(*u*++)；如果 arr[i]==arr[u]，说明当前数 arr[i]的值之前已经加入到 A 区域，此时不用再加入。

4．重复步骤 3，直到所有的数遍历完。

具体请参看如下代码中的 leftUnique 方法。

```
public void leftUnique(int[] arr) {
        if (arr == null || arr.length < 2) {
                return;
        }
        int u = 0;
        int i = 1;
        while (i != arr.length) {
                if (arr[i++] != arr[u]) {
                        swap(arr, ++u, i - 1);
                }
        }
}
```

再来介绍补充问题的解法：

1．生成变量 left，含义是在 arr[0..left]（左区）上都是 0，left 是这个区域当前最右的位置，初始时 left 为-1。

2．生成变量 index，利用这个变量做从左到右的遍历，含义是在 arr[left+1..index]（中区）上都是 1，index 是这个区域的当前最右位置，初始时 index 为 0。

3．生成变量 right，含义是在 arr[right..N-1]（右区）上都是 2，right 是这个区域的当前最左位置，初始时 right 为 N。

4．index 表示遍历到 arr 的一个位置：

1）如果 arr[index]==1，这个值应该直接加入到中区，index++之后重复步骤 4。

2）如果 arr[index]==0，这个值应该加入到左区，arr[left+1]是中区最左的位置，所以把 arr[index]和 arr[left+1]交换之后，左区就扩大了，index++之后重复步骤 4。

3）如果 arr[index]==2，这个值应该加入到右区，arr[right-1]是右区最左边的数的左边，但也不属于中区，总之，在中区和右区的中间部分。把 arr[index]和 arr[right-1]交换之后，右区就向左扩大了(right--)，但是此时 arr[index]上的值未知，所以 index 不变，重复步骤 4。

5．当 index==right 时，说明中区和右区成功对接，三个区域都划分好后，过程停止。

遍历中的每一步，要么 index 增加，要么 right 减少，如果 index==right，过程就停止，

所以时间复杂度就是 $O(N)$，具体过程请参看如下代码中的 sort 方法。

```
public void sort(int[] arr) {
        if (arr == null || arr.length < 2) {
                return;
        }
        int left = -1;
        int index = 0;
        int right = arr.length;
        while (index < right) {
                if (arr[index] == 0) {
                        swap(arr, ++left, index++);
                } else if (arr[index] == 2) {
                        swap(arr, index, --right);
                } else {
                        index++;
                }
        }
}
```

求最短通路值

【题目】

用一个整型矩阵 matrix 表示一个网络，1 代表有路，0 代表无路，每一个位置只要不越界，都有上下左右 4 个方向，求从最左上角到最右下角的最短通路值。

例如，matrix 为：

1 0 1 1 1

1 0 1 0 1

1 1 1 0 1

0 0 0 0 1

通路只有一条，由 12 个 1 构成，所以返回 12。

【难度】

尉 ★★☆☆

【解答】

使用宽度优先遍历即可，如果矩阵大小为 $N \times M$，本文提供的方法的时间复杂度为 $O(N \times M)$，具体过程如下：

1．开始时生成 map 矩阵，map[i][j]的含义是从(0,0)位置走到(*i*,*j*)位置最短的路径值。然后将左上角位置(0,0)的行坐标与列坐标放入行队列 rQ，和列队列 cQ。

2．不断从队列弹出一个位置(*r*,*c*)，然后看这个位置的上下左右四个位置哪些在 matrix 上的值是 1，这些都是能走的位置。

3．将那些能走的位置设置好各自在 map 中的值，即 map[r][c]+1。同时将这些位置加入到 rQ 和 cQ 中，用队列完成宽度优先遍历。

4．在步骤 3 中，如果一个位置之前走过，就不要重复走，这个逻辑可以根据一个位置在 map 中的值来确定，比如 map[i][j]!=0，就可以知道这个位置之前已经走过。

5．一直重复步骤 2～步骤 4。直到遇到右下角位置，说明已经找到终点，返回终点在 map 中的值即可，如果 rQ 和 cQ 已经为空都没有遇到终点位置，说明不存在这样一条路径，返回 0。

每个位置最多走一遍，所以时间复杂度为 $O(N\times M)$、额外空间复杂度也是 $O(N\times M)$。具体过程请参看如下代码中的 minPathValue 方法。

```
public int minPathValue(int[][] m) {
        if (m == null || m.length == 0 || m[0].length == 0 || m[0][0] != 1
                        || m[m.length - 1][m[0].length - 1] != 1) {
                return 0;
        }
        int res = 0;
        int[][] map = new int[m.length][m[0].length];
        map[0][0] = 1;
        Queue<Integer> rQ = new LinkedList<Integer>();
        Queue<Integer> cQ = new LinkedList<Integer>();
        rQ.add(0);
        cQ.add(0);
        int r = 0;
        int c = 0;
        while (!rQ.isEmpty()) {
                r = rQ.poll();
                c = cQ.poll();
                if (r == m.length - 1 && c == m[0].length - 1) {
                        return map[r][c];
                }
                walkTo(map[r][c], r - 1, c, m, map, rQ, cQ); // up
                walkTo(map[r][c], r + 1, c, m, map, rQ, cQ); // down
                walkTo(map[r][c], r, c - 1, m, map, rQ, cQ); // left
                walkTo(map[r][c], r, c + 1, m, map, rQ, cQ); // right
        }
        return res;
}

public void walkTo(int pre, int toR, int toC, int[][] m,
```

```
                int[][] map, Queue<Integer> rQ, Queue<Integer> cQ) {
        if (toR < 0 || toR == m.length || toC < 0 || toC == m[0].length
                        || m[toR][toC] != 1 || map[toR][toC] != 0) {
                return;
        }
        map[toR][toC] = pre + 1;
        rQ.add(toR);
        cQ.add(toC);
    }
```

数组中未出现的最小正整数

【题目】

给定一个无序整型数组 arr，找到数组中未出现的最小正整数。

【举例】

arr=[-1,2,3,4]。返回 1。

arr=[1,2,3,4]。返回 5。

【难度】

尉 ★★☆☆

【解答】

原问题。如果 arr 长度为 N，本题的最优解可以做到时间复杂度为 $O(N)$，额外空间复杂度为 $O(1)$。具体过程如下：

1．在遍历 arr 之前先生成两个变量。变量 l 表示遍历到目前为止，数组 arr 已经包含的正整数范围是[1,l]，所以没有开始遍历之前令 l=0，表示 arr 目前没有包含任何正整数。变量 r 表示遍历到目前为止，在后续出现最优状况的情况下，arr 可能包含的正整数范围是[1,r]，所以没有开始遍历之前，令 r=N，因为还没有开始遍历，所以后续出现的最优状况是 arr 包含 1~N 所有的整数。r 同时表示 arr 当前的结束位置。

2．从左到右遍历 arr，遍历到位置 l，位置 l 的数为 arr[l]。

3．如果 arr[l]==l+1。没有遍历 arr[l]之前，arr 已经包含的正整数范围是[1,l]，此时出现了 arr[l]==l+1 的情况，所以 arr 包含的正整数范围可以扩到[1,l+1]，即令 l++。然后重复步骤 2。

4．如果 arr[l]<=l。没有遍历 arr[l]之前，arr 在后续最优的情况下可能包含的正整数范围是[1,*r*]，已经包含的正整数范围是[1,*l*]，所以需要[*l*+1,*r*]上的数。而此时出现了 arr[l]<=l，说明[*l*+1,*r*]范围上的数少了一个，所以 arr 在后续最优的情况下，可能包含的正整数范围缩小了，变为[1,*r*-1]，此时把 arr 最后位置的数(arr[r-1])放在位置 *l* 上，下一步检查这个数，然后令 r--。重复步骤 2。

5．如果 arr[l]>r，与步骤 4 同理，把 arr 最后位置的数(arr[r-1])放在位置 *l* 上，下一步检查这个数，然后令 r--。重复步骤 2。

6．如果 arr[arr[l]-1]==arr[l]。如果步骤 4 和步骤 5 没中，说明 arr[l]是在[*l*+1,*r*]范围上的数，而且这个数应该放在 arr[l]-1 位置上。可是此时发现 arr[l]-1 位置上的数已经是 arr[l]，说明出现了两个 arr[l]，既然在[*l*+1,*r*]上出现了重复值，那么[*l*+1,*r*]范围上的数又少了一个，所以与步骤 4 和步骤 5 一样，把 arr 最后位置的数(arr[r-1])放在位置 *l* 上，下一步检查这个数，然后令 r--。重复步骤 2。

7．如果步骤 4、步骤 5 和步骤 6 都没中，说明发现了[*l*+1,r]范围上的数，并且此时并未发现重复。那么 arr[l]应该放到 arr[l]-1 位置上，所以把 *l* 位置上的数和 arr[l]-1 位置上的数交换，下一步继续遍历 *l* 位置上的数。重复步骤 2。

8．最终 *l* 位置和 *r* 位置会碰在一起（l==r），arr 已经包含的正整数范围是[1,*l*]，返回 *l*+1 即可。

具体过程请参看如下代码中的 missNum 方法。

```
public int missNum(int[] arr) {
    int l = 0;
    int r = arr.length;
    while (l < r) {
        if (arr[l] == l + 1) {
                l++;
        } else if (arr[l] <= l || arr[l] > r || arr[arr[l] - 1] == arr[l]) {
                arr[l] = arr[--r];
        } else {
                swap(arr, l, arr[l] - 1);
        }
    }
    return l + 1;
}
```

数组排序之后相邻数的最大差值

【题目】

给定一个整型数组 arr，返回排序后的相邻两数的最大差值。

【举例】

arr=[9,3,1,10]。如果排序，结果为[1,3,9,10]，9 和 3 的差为最大差值，故返回 6。
arr=[5,5,5,5]。返回 0。

【要求】

如果 arr 的长度为 N，请做到时间复杂度为 $O(N)$。

【难度】

尉 ★★☆☆

【解答】

本题如果用排序法实现，其时间复杂度是 $O(N\log N)$，而如果利用桶排序的思想（不是直接进行桶排序），可以做到时间复杂度为 $O(N)$，额外空间复杂度为 $O(N)$。遍历 arr 找到最小值和最大值，分别记为 min 和 max。如果 arr 的长度为 N，那么我们准备 N+1 个桶，把 max 单独放在第 N+1 号桶里。arr 中在[min,max)范围上的数放在 1~N 号桶里，对于 1~N 号桶中的每一个桶来说，负责的区间大小为(max-min)/N。比如长度为 10 的数组 arr 中，最小值为 10，最大值为 110。那么就准备 11 个桶，arr 中等于 110 的数全部放在第 11 号桶里。区间[10,20)的数全部放在 1 号桶里，区间[20,30)的数全部放在 2 号桶里……，区间[100,110)的数全部放在 10 号桶里。那么如果一个数为 num，它应该分配进(num - min) × len / (max - min)号桶里。

arr 一共有 N 个数，min 一定会放进 1 号桶里，max 一定会放进最后的桶里，所以，如果把所有的数放入 N+1 个桶中，必然有桶是空的。如果 arr 经过排序，相邻的数有可能此时在同一个桶中，也可能在不同的桶中。在同一个桶中的任何两个数的差值都不会大于区间值，而在空桶左右两边不空的桶里，相邻数的差值肯定大于区间值。所以产生最大差值的两个相邻数肯定来自不同的桶。所以只要计算桶之间数的间距就可以，也就是只用记录

每个桶的最大值和最小值，最大差值只可能来自某个非空桶的最小值减去前一个非空桶的最大值。

具体过程请参看如下代码中的 maxGap 方法。

```
public int maxGap(int[] nums) {
        if (nums == null || nums.length < 2) {
                return 0;
        }
        int len = nums.length;
        int min = Integer.MAX_VALUE;
        int max = Integer.MIN_VALUE;
        for (int i = 0; i < len; i++) {
            min = Math.min(min, nums[i]);
            max = Math.max(max, nums[i]);
        }
        if (min == max) {
            return 0;
        }
        boolean[] hasNum = new boolean[len + 1];
        int[] maxs = new int[len + 1];
        int[] mins = new int[len + 1];
        int bid = 0;
        for (int i = 0; i < len; i++) {
            bid = bucket(nums[i], len, min, max); // 算出桶号
            mins[bid] = hasNum[bid] ? Math.min(mins[bid], nums[i]) : nums[i];
            maxs[bid] = hasNum[bid] ? Math.max(maxs[bid], nums[i]) : nums[i];
            hasNum[bid] = true;
        }
        int res = 0;
        int lastMax = 0;
        int i = 0;
        while (i <= len) {
            if (hasNum[i++]) { // 找到第一个不为空的桶
                lastMax = maxs[i - 1];
                break;
            }
        }
        for (; i <= len; i++) {
            if (hasNum[i]) {
                res = Math.max(res, mins[i] - lastMax);
                lastMax = maxs[i];
            }
        }
        return res;
}

// 使用 long 类型是为了防止相乘时溢出
public int bucket(long num, long len, long min, long max) {
        return (int) ((num - min) * len / (max - min));
}
```