题目一：找出数组中是否有重复的数字

在一个长度为 n 的数组里的所有数字都在 0 ~ n-1 的范围内，数组中某些数字是重复的，但不知道哪几个数组重复了，也不知道重复了几次。请找出数组中任意一个重复的数字。例如，如果输入长度为 7 的数组{2,3,1,0,2,5,3},那么对应的输出是重复的数字 2 或者 3

解法1：

先对这个数组进行排序，然后从头到尾扫描排序后的数组，排序一个长度为 n 的数组的时间复杂度是 O(nlogn)

解法2：

从头到尾按顺序扫描数组的每个数字，每扫描一个数字，都可以用 O(1) 的时间来判断哈希表里有没有这个数字，如果没有这个数字，则把它加入哈希表，如果已经存在该数字，就找到了一个重复数字，这个算法的时间复杂度是 O(n),但它提高时间效率的代价是一个大小为 O(n) 的哈希表。

解法3：

根据题目所述，我们注意到数组中的数字都在 0~n-1 的范围内，如果这个数组中没有重复的数字，那么当数组排序之后数字 i 将出现在下标为 i 的位置。由于数组中有重复的数字，有些位置可能存在多个数字，同时有些位置可能没有数字。

从头到尾依次扫描这个数组中的每个数字，当扫描到下标为 i 的数字时，首先比较这个数字（用m 表示）是不是等于 i。如果是，则接着扫描下一个数字，如果不是，则拿它和第 m 个数字进行比较，如果它和第 m 个数字相等，则该数字重复（该数字在下标为 i 和 m 的位置都出现了）。如果它和第 m 个数字不相等，就把 i 个数字和 第 m 个数字交换， 把 m 放到属于它的位置，接下来再重复这个比较过程，直到发现一个重复数字。

例：

{2,3,1,0,2,5,3}

- 下标 0 的数字 2，与下标不相等
- 把 2 和下标2处的数字 1 交换

{1,3,2,0,2,5,3}

- 下标 0 处数字是 1，仍然和下标不相等
- 继续把 1 和 3 交换

{3,1,2,0,2,5,3}

- 3 和下标 0 还是不相等
- 把 3 和 0 交换

{0,1,2,3,2,5,3}

- 接下来 0 等于下标0
- 下次 for循环 1 等于下标1
- 下次 for循环 2 等于下标2
- 再次 for 循环 3 等于下标 3
- 继续循环 2 不等于下标 4
- 比较 2 和下标为 2 处的数字，发现相等，也就是数字2 在下标为 2 和下标为 4 的位置都出现了。

```
public class DuplicationInArray_03_01 {

    public static void main(String[] args) {
        int[] a = {2,3,4,1,4};
        System.out.println("a has duplicate number: " + findDuplicateNumber(a, 5));
    }

    private static boolean findDuplicateNumber(int[] array, int length) {
        if (array == null || length <= 0) {
            // 处理无效输入
            return false;
        }

        for(int i=0; i < length; ++i) {
            // 输入不满足 0 ~ n-1 之间
            if (array[i] < 0 || array[i]> length -1) {
                return false;
            }
        }

        for(int i = 0; i < length; ++i) {
            while (array[i] != i) {
                // 当前数组在 i 和 位置 array[i] 都出现了，表示重复
                if (array[i] == array[array[i]]) {
                    return true;
                }

                // 把 array[i] 的值换到对应下标位置
                int temp = array[i];
                array[i] = array[array[i]];
                array[temp] = temp;
            }
        }
        return false;
    }
}
```

复杂度分析

这个算法尽管有双重循环，但每个数字只需要交换两次就能找到属于它自己的位置，因此总的时间复杂度是 O(n),因为所有的操作都在输入数组上进行，因此不需要额外分配内存，因此空间复杂度是

O(1)

题目二：不修改数组找出重复的数字

在一个长度为 n+1 的数组里的所有数字都在 1~n 的范围内，所以这个数组中至少有一个数字是重复的，请找出数组中任意一个重复的数字，但不能修改输入的数组。例如，如果输入的长度为8 的数组{2，3，5，4，3，2，6，7}，那么对应的输出是重复数字 2 或者 3

解法1：

由于不能修改输入的数组，我们创建一个长度为 n+1 的辅助数组，然后逐一把原数组的每个数字复制到辅助数组，如果原数组中数字是 m，就复制到下标为 m 的地方，这样很容易判断重复。

解法2：

有没有什么办法可以避免使用 O(n) 的辅助空间。为什么数组中有重复数字，假如没有重复的数字，那么从 1~n 的范围里只有 n 个数字，由于数组里超过了 n 个数字，则一定包含重复数字。

我们把 1 ~ n 的数字从中间的数字 m 分成两部分，前面一半为 1~m, 后面的一半为 m+1 ~ n。 如果 1 ~ m 的数字的数目超过 m，那么这一半的区间里一定包含重复的数字，否则，重复的数字在 m+1 ~ n 里，依次类推，这样这个方法就转换成了类似二分查找。

例：

{2,3,5,4,3,2,6,7}

- 数组里的数字在 1 ~ 7 的范围内，用 4 把数组分两半 1~4 和 5~7
- 1~4 这几个数字在数组中出现了 5 次，因此这 4 个数字一定有重复的。
- 把 1 ~ 4 一分为二， 1，2 出现了 2 次， 3，4 出现了 3次，则这两个数字一定有重复的
- 分别统计3,4 发现 3 出现了两次，是一个重复数字。

```
public class DuplicationInArrayNoEdit {
    public static void main(String[] args) {
        int[] array = {2,3,5,4,3,2,6,7};
        System.out.println("duplicate number is " + getDuplication(array, 8));
    }

    public static int getDuplication(int[] array, int length) {
        if(array == null || length <= 0) {
            return -1;
        }
        int start = 1;
        int end = length -1;    // 长度为 n + 1，  范围是 1~n，所以这里是 length -1
        while (end >= start) {
            int middle = ((end - start) >> 1) + start;
            int count = countRange(array, length, start, middle);
            // 开始等于结束，最后一个数
            if (end == start) {
                if (count > 1) {
```

```
                    return start;   // 找到重复
                } else {
                    break;  // 未找到，退出循环
                }
            }
            if (count > (middle - start + 1)) { // 出现次数大于数字个数，这部分有重复的
                end = middle;
            } else {
                start = middle + 1;
            }
        }
        return -1;
    }

    private static int countRange(int [] array, int length, int start, int end) {
        if (array == null)
            return 0;
        int count = 0;
        for (int i = 0; i < length; i++) {
            // start 到 end 出现的次数
            if (array[i] >= start && array[i] <= end) {
                ++count;
            }
        }
        return count;
    }
}
```

复杂度分析

上述代码按照二分查找的思路，对于长度为 n 的数组， countRange 将被调用 O{logn} 次，每次需要 O{n} 的时间，因此总的时间复杂度是 O{nlogn}, 空间复杂度为 O{1}，相比解法1，这种是以时间换空间。