题目：

Given an array *nums* containing *n* + 1 integers where each integer is between 1 and *n* (inclusive), prove that at least one duplicate number must exist. Assume that there is only one duplicate number, find the duplicate one.

Note:

1. You **must not** modify the array (assume the array is read only).
2. You must use only constant, *O*(1) extra space.
3. Your runtime complexity should be less than `O(n²)`.
4. There is only one duplicate number in the array, but it could be repeated more than once.

/* 参考博客 :http://bookshadow.com/weblog/2015/09/28/leetcode-find-duplicate-number/ */

1.时间：O（NLOGN）; 空间：O（1） ->鸽笼原理

```
class Solution {
    /* 根据鸽笼原理，给定 n+1 个范围[1,n]的整数，其中一定存在数字出现至少两次，
        假设枚举的数字为 n / 2
        遍历数组，若数组中不大于 n / 2 的数字个数超过 n / 2，则可以确定[1, n / 2]范围内一定有解
        否则，可以确定解在(n / 2, n]内*/
public:
    int findDuplicate(vector<int>& nums) {
```

```
        if (nums.empty()) return 0;

        /* nums.size() == n + 1, 中位数为：( 1+n ) / 2 */
        int lower = 1, upper = nums.size() - 1;
        while (lower < upper){
            int mid = lower + ((upper - lower) >> 1);
            int count = std::count_if(nums.begin(), nums.end(), [&](const int num){
                return num <= mid;
            });
            if (count <= mid) lower = mid + 1;
            else upper = mid;
        }

        return lower;
    }
};
```

2.时间：O (N)；空间：O (1)

> 这道题（据说）花费了计算机科学界的传奇人物 Don Knuth 24 小时才解出来。并且我只见过一个人（Keith Amling）用更短时间解出此题。

问题的第一部分 - 证明至少存在一个重复元素 - 是鸽笼原理的直接应用。如果元素的范围是[1, n]，那么只存在 n 种不同的值。如果有 n+1 个元素，其中一个必然重复。

问题的第二部分 - 在给定约束条件下寻找重复元素 - 可就难多了。 要解决这个问题，我们需要敏锐的洞察力，使问题通过一列的转化，变为一个完全不同的问题。

解决本题需要的主要技巧就是要注意到：由于数组的 n + 1 个元素范围从 1 到 n，我们可以将数组考虑成一个从集合{1, 2, ..., n}到其本身的函数 f。这个函数的定义为 f(i) = A[i]。基于这个设定，重复元素对应于一对下标 i != j 满足 f(i) = f(j)。我们的任务就变成了寻找一对(i, j)。一旦我们找到这个值对，只需通过 f(i) = A[i]即可获得重复元素。

但是我们怎样寻找这个重复值呢？这变成了计算机科学界一个广为人知的“环检测”问题。问题的一般形式如下：给定一个函数 f，序列 x_i 的定义为

```
x_0     = k       (for some k)

x_1     = f(x_0)

x_2     = f(f(x_0))

...

x_{n+1} = f(x_n)
```

假设函数 f 从定义域映射到它本身，此时会有 3 种情况。首先，如果定义域是无穷的，则序列是无限长并且没有循环的。例如，函数 f(n) = n + 1，在整数范围内满足这个性质 - 没有数字是重复的。 第二， 序列可能是一个闭合循环，这意味着存在一个 i 使得 x_0 = x_i。在这个例子中，序列在一组值内无限循环。第三，序列有可能是的“ρ 型的”，此时序列看起来像下面这样：

```
x_0 -> x_1 -> ... x_k -> x_{k+1} ... -> x_{k+j}

                    ^                        |

                    |                        |

                    +------------------------+
```

也就是说，序列从一列链条型的元素开始进入一个环，然后无限循环。我们将环的起点称为环的“入口”。

对于从数组中寻找重复元素这个问题，考虑序列从位置 n 开始重复调用函数 f。亦即从数组的最后一个元素开始，然后移动到其元素值对应的下标处，并且重复此过程。可以得到：此序列是 ρ 型的。要明白这一点，需要注意到其中一定有环，因为数组是有限的并且当访问 n 个元素时，一定会对某个元素访问两次。无论从数组的哪一个位置开始，这都是成立的。

另外，注意由于数组元素范围 1 到 n，因此不存在值为 0 的元素。进而，从数组的第一个元素开始调用一次函数 f 之后，再也不会回到这里。这意味着第一个元素不会是环的一部分，但如果我们继续重复调用函数 f，最终总会访问某个节点两次。从 0 节点开始的链条与环形相接，使得其形状一定是 ρ 型。

此外，考虑一下环的入口。由于节点位于环的入口，一定存在两个输入，其对应的函数 f 的输出值都等于入口元素下标。要使其成立，一定存在两个下标 i != j，满足 f(i) = f(j)，亦即 A[i] = A[j]。因而环的入口一定是重复值。

这是由 Robert Floyd 提出的一个著名算法，给定一个 ρ 型序列，在线性时间，只使用常数空间寻找环的起点。这个算法经常被称为“龟兔”算法，至于原因下面就明白了。

算法背后的思想是定义两个变量。首先，令 c 为进入环的链的长度，然后令 l 为环的长度。接下来，令 l'为大于 c 的 l 的倍数的最小值。可以得出结论：对于上文定义的任意 ρ 型序列的 l'，都有

```
x_{l'} = x_{2l'}
```

证明实际上非常直观并且具有自明性 - 这是计算机科学中我最喜欢的证明之一。思路就是由于 l'至少为 c，它一定包含在环内。同时，由于 l'是环长度的倍数，我们可以将其写作 ml，其中 m 为常数。如果我们从位置 x_{l'}开始（其在环内），然后再走 l'步到达 x_{2l'}，则我们恰好绕环 m 次，正好回到起点。

Floyd 算法的一个关键点就是即使我们不明确知道 c 的值，依然可以在 O(l')时间内找到值 l'。思路如下。我们追踪两个值"slow"和"fast"，均从 x_0 开始。然后迭代计算

```
slow = f(slow)

fast = f(f(fast))
```

我们重复此步骤直到 slow 与 fast 彼此相等。此时，我们可知存在 j 满足 slow = x_j，并且 fast = x_{2j}。 由于 x_j = x_{2j}，可知 j 一定至少为 c，因为此时已经在环中。另外，可知 j 一定是 l 的倍数，因为 x_j = x_{2j}意味着在环内再走 j 步会得到同样的结果。最后，j 一定是大于 c 的 l 的最小倍数，因为如果存在一个更小的大于 c 的 l 的倍数，我们一定会在到达 j 之前到达那里。所以，我们一定有 j = l'，意味着我们可以在不知道环的长度或者形状的情况下找到 l'。

要完成整个过程，我们需要明白如何使用 l'来找到环的入口（记为 x_c）。要做到这一步，我们再用最后一个变量，记为"finder"，从 x_0 出发。然后迭代重复执行过程：

```
finder = f(finder)
```

```
    slow   = f(slow)
```

直到 finder = slow 为止。我们可知：(1) 两者一定会相遇 (2) 它们会在环的入口相遇。要理解这两点，我们注意由于 slow 位于 x_{l'}，如果我们向前走 c 步，那么 slow 会到达位置 x_{l' + c}。由于 l'是环长度的倍数，相当于向前走了 c 步，然后绕环几圈回到原位。换言之，x_{l' + c} = x_c。另外，考虑 finder 变量在行进 c 步之后的位置。 它由 x_0 出发，因此 c 步之后会到达 x_c。这证明了(1)和(2)，由此我们已经证明两者最终会相遇，并且相遇点就是环的入口。

算法的美妙之处在于它只用 O(1)的额外存储空间来记录两个不同的指针 - slow 指针和 fast 指针（第一部分），以及 finder 指针（第二部分）。但是在此之上，运行时间是 O(n)的。要明白这一点，注意 slow 指针追上 fast 指针的时间是 O(l')。由于 l'是大于 c 的 l 的最小倍数，有两种情况需要考虑。首先，如果 l > c，那么就是 l。 否则，如果 l < c，那么我们可知一定存在 l 的倍数介于 c 与 2c 之间。要证明这一点，注意在范围 c 到 2c 内，有 c 个不同的值，由于 l < c，其中一定有值对 l 取模运算等于 0。最后，寻找环起点的时间为 O(c)。这给出了总的运行时间至多为 O(c + max{l, 2c})。所有这些值至多为 n，因此算法的运行时间复杂度为 O(n)。

```
/* 类似于环状链表求入口结点一样，"龟兔"算法 */

class Solution {

public:

    int findDuplicate(vector<int>& nums) {

        if (nums.empty()) return 0;



        /* 默认 nums 内没有>nums.size()的数 */

        int slower = 0, faster = 0;
```

```
        do{
            slower = nums[slower];
            faster = nums[nums[faster]];
        } while (slower != faster);

        int finder = 0;
        do{
            slower = nums[slower];
            finder = nums[finder];
        } while (slower != finder);

        return finder;
    }
};
```