Previous Next

287. Find the Duplicate Number

Dec. 11, 2017 | 233.4K views

Average Rating: 4.44 (298 votes)

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.

Example 1:

```
Input: [1,3,4,2,2]
Output: 2
```

Example 2:

```
Input: [3,1,3,4,2]
Output: 3
```

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 $O(n^2)$.
4. There is only one duplicate number in the array, but it could be repeated more than once.

Note

The first two approaches mentioned do not satisfy the constraints given in the prompt, but they are solutions that you might be likely to come up with during a technical interview. As an interviewer, I personally would *not* expect someone to come up with the cycle detection solution unless they have heard it before.

Proof

Proving that at least one duplicate must exist in `nums` is simple application of the pigeonhole principle. Here, each number in `nums` is a "pigeon" and each distinct number that can appear in `nums` is a "pigeonhole". Because there are $n+1$ numbers are n distinct possible numbers, the pigeonhole principle implies that at least one of the numbers is duplicated.

Approach 1: Sorting

Intuition

If the numbers are sorted, then any duplicate numbers will be adjacent in the sorted array.

Algorithm

Given the intuition, the algorithm follows fairly simply. First, we sort the array, and then we compare each element to the previous element. Because there is exactly one duplicated element in the array, we know that the array is of at least length 2, and we can return the duplicate element as soon as we find it.

Java | Python3 — Copy

```python
class Solution:
    def findDuplicate(self, nums):
        nums.sort()
        for i in range(1, len(nums)):
            if nums[i] == nums[i-1]:
                return nums[i]
```

Complexity Analysis

- Time complexity : $\mathcal{O}(nlgn)$

  The `sort` invocation costs $\mathcal{O}(nlgn)$ time in Python and Java, so it dominates the subsequent linear scan.

- Space complexity : $\mathcal{O}(1)$ (or $\mathcal{O}(n)$)

  Here, we sort `nums` in place, so the memory footprint is constant. If we cannot modify the input array, then we must allocate linear space for a copy of `nums` and sort that instead.

Approach 2: Set

Intuition

If we store each element as we iterate over the array, we can simply check each element as we iterate over the array.

Algorithm

In order to achieve linear time complexity, we need to be able to insert elements into a data structure (and look them up) in constant time. A `Set` satisfies these constraints nicely, so we iterate over the array and insert each element into `seen`. Before inserting it, we check whether it is already there. If it is, then we found our duplicate, so we return it.

Java | Python3 — Copy

```python
class Solution:
    def findDuplicate(self, nums):
        seen = set()
        for num in nums:
            if num in seen:
                return num
            seen.add(num)
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n)$

  `Set` in both Python and Java rely on underlying hash tables, so insertion and lookup have amortized constant time complexities. The algorithm is therefore linear, as it consists of a `for` loop that performs constant work n times.

- Space complexity : $\mathcal{O}(n)$

  In the worst case, the duplicate element appears twice, with one of its appearances at array index $n-1$. In this case, `seen` will contain $n-1$ distinct values, and will therefore occupy $\mathcal{O}(n)$ space.

Approach 3: Floyd's Tortoise and Hare (Cycle Detection)

Intuition

The idea is to reduce the problem to Linked List Cycle II:

> Given a linked list, return the node where the cycle begins.

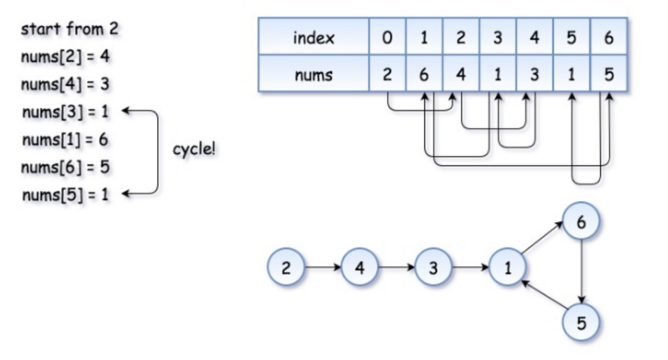
First of all, where does the cycle come from? Let's use the function `f(x) = nums[x]` to construct the sequence: `x, nums[x], nums[nums[x]], nums[nums[nums[x]]], ...`.

Each new element in the sequence is an element in nums at the index of the *previous* element.

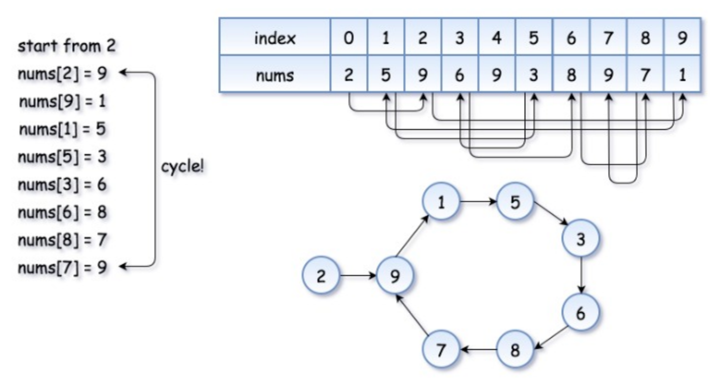
If one starts from `x = nums[0]`, such a sequence will produce a linked list with a cycle.

> The cycle appears because `nums` contains duplicates. The duplicate node is a cycle entrance.

Here is how it works:



The example above is simple because the loop is small. Here is a more interesting example (special thanks to @sushant_chaudhari)



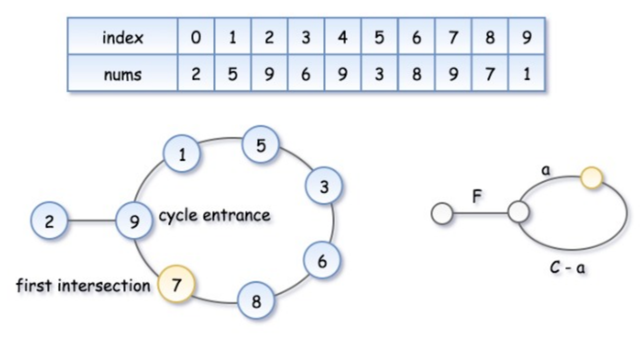
Now the problem is to find the entrance of the cycle.

Algorithm

Floyd's algorithm consists of two phases and uses two pointers, usually called `tortoise` and `hare`.

In phase 1, `hare = nums[nums[hare]]` is twice as fast as `tortoise = nums[tortoise]`. Since the hare goes fast, it would be the first one who enters the cycle and starts to run around the cycle. At some point, the tortoise enters the cycle as well, and since it's moving slower the hare catches the tortoise up at some *intersection* point. Now phase 1 is over, and the tortoise has lost.

> Note that the intersection point is not the cycle entrance in the general case.

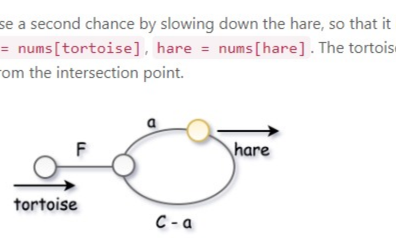


To compute the intersection point, let's note that the hare has traversed twice as many nodes as the tortoise, i.e. $2d(\text{tortoise}) = d(\text{hare})$, that means

$2(F+a) = F+nC+a$, where n is some integer.

> Hence the coordinate of the intersection point is $F+a=nC$.

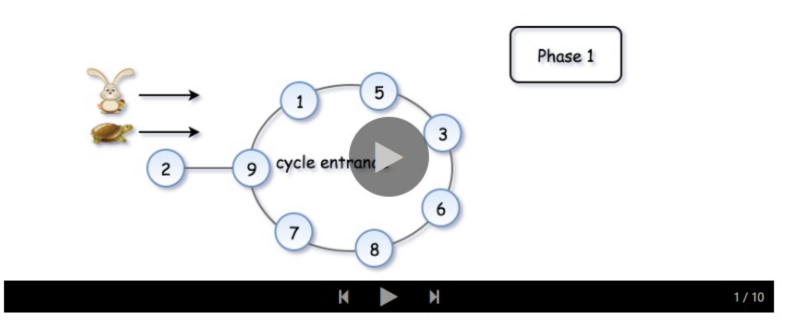
In phase 2, we give the tortoise a second chance by slowing down the hare, so that it now moves with the speed of tortoise: `tortoise = nums[tortoise]`, `hare = nums[hare]`. The tortoise is back at the starting position, and the hare starts from the intersection point.



Let's show that this time they meet at the cycle entrance after F steps.

- The tortoise started from zero, so its position after F steps is F.
- The hare started at the intersection point $F+a=nC$, so its position after F steps is $nC+F$, that is the same point as F.
- So the tortoise and the (slowed down) hare will meet at the entrance of the cycle.

Implementation



Java | Python3 — Copy

```python
class Solution:
    def findDuplicate(self, nums):
        # Find the intersection point of the two runners.
        tortoise = hare = nums[0]
        while True:
            tortoise = nums[tortoise]
            hare = nums[nums[hare]]
            if tortoise == hare:
                break

        # Find the "entrance" to the cycle.
        tortoise = nums[0]
        while tortoise != hare:
            tortoise = nums[tortoise]
            hare = nums[hare]

        return hare
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n)$

  For detailed analysis, refer to Linked List Cycle II.

- Space complexity : $\mathcal{O}(1)$

  For detailed analysis, refer to Linked List Cycle II.

Analysis and solutions written by: @emptyset

Rate this article:

Previous Next

Comments: 169 Sort By

Type comment here... (Markdown is supported)

Preview Post

nella ★310 April 25, 2019 9:50 AM

Using sort and set doesn't meet the requirement of non-modifiable and O(1) space

301 Share Reply

SHOW 10 REPLIES

tanja ★86 September 11, 2018 4:17 AM

When the tortoise and hare meet, both values are 6 - why can't we just return 6 at that point? Why do we need phase 2? Thanks!

82 Share Reply

SHOW 15 REPLIES

runningsnail ★112 December 2, 2018 4:41 AM

The intuition here is that:

Index range is `[0, n]` inclusive, value range is `[1, n]` inclusive. Value is non-zero and range is always inside of index's range (**note** that the detect cycling way wouldn't work if array values range is [0, n-1] because if value `0` appears at `index 0` then `0` forms a cycle with itself even though `0`

Read More

91 Share Reply

SHOW 7 REPLIES

tddaniel ★102 November 23, 2018 4:25 AM

I don't see any point of Approach 1 or 2. Both don't satisfy the constraint of the problem.

98 Share Reply

SHOW 4 REPLIES

myih ★117 December 28, 2018 12:11 AM

Only one solution actually satisfied the requirements, should be a hard question with all the requirements enforced. Binary search solution is somewhere medium to hard...

49 Share Reply

coder_xyzzy ★142 March 31, 2019 5:09 AM

There is an O(n*log(n)) solution which matches the stated constraints and which might be easier to come up with for people who haven't seen the tortoise and hare trick. Do a binary search on the set of numbers [1,...,n]. For each number, count how many elements of the array are <= or >= that number. Recurse as appropriate.

43 Share Reply

SHOW 11 REPLIES

PorcoRedwood ★14 January 8, 2018 6:25 AM

Here's my understanding of the Floyd's Tortoise and Hare solution, and the analysis of its time complexity.

First of all, when traversing the array described in the problem by always using the current value as the next index to go to, there must be a loop. Let's say C is the length of the loop, which is smaller than the

Read More

14 Share Reply

SHOW 1 REPLY

Razzy6 ★21 January 24, 2019 12:30 PM

does the author require array read only? how can solution1 sort it?

17 Share Reply

SHOW 1 REPLY

mengmengfan ★64 November 3, 2018 6:04 AM

My python 3-line method, O(n) time, O(1) space, beats 98%:

```
while nums[nums[0]] != nums[0]:
    nums[nums[0]], nums[0] = nums[0], nums[nums[0]]
return nums[0]
```

Read More

32 Share Reply

SHOW 2 REPLIES

LLLLLL ★124 October 10, 2018 12:26 AM

I think the binary search approach which is much more interesting should be include, though.

16 Share Reply

SHOW 6 REPLIES