<https://leetcode.com/problems/find-the-duplicate-number/solution>

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](https://en.wikipedia.org/wiki/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*n*+1 numbers are n*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.

class Solution {

public int findDuplicate(int[] nums) {

Arrays.sort(nums);

for (int i = 1; i < nums.length; i++) {

if (nums[i] == nums[i-1]) {

return nums[i];

}

}

return -1;

}

}

**Complexity Analysis**

* Time complexity :  O(*nlgn*)

The sort invocation costs O(*nlgn*) time in Python and Java, so it dominates the subsequent linear scan.

* Space complexity :  O(1) (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.

class Solution {

public int findDuplicate(int[] nums) {

Set<Integer> seen = new HashSet<Integer>();

for (int num : nums) {

if (seen.contains(num)) {

return num;

}

seen.add(num);

}

return -1;

}

}

**Complexity Analysis**

* Time complexity :  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 :  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  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](https://leetcode.com/problems/linked-list-cycle-ii/solution/):

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

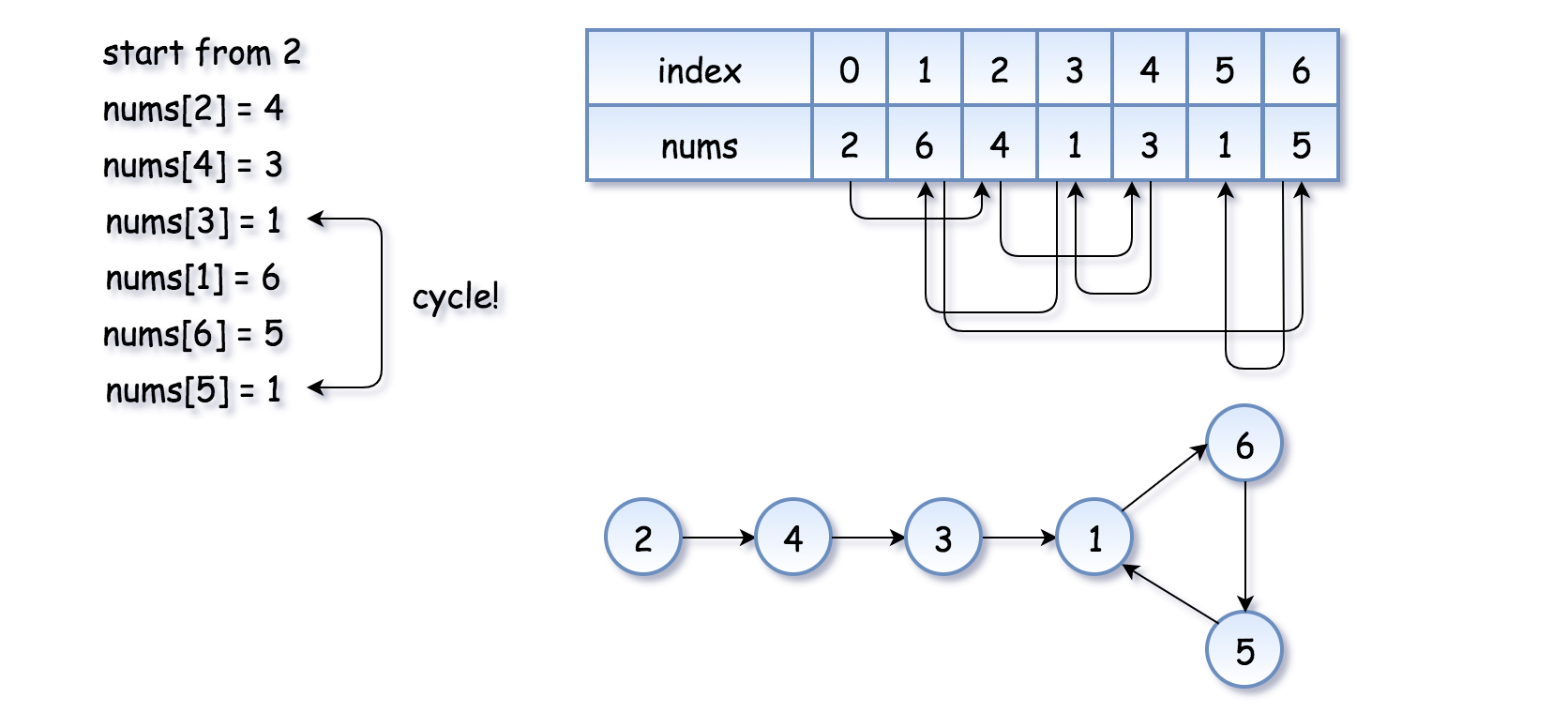
First of all, where does the cycle come from? Let's use the function f(x) = nums[x] to construct the sequence: x, nums[x], nums[nums[x]], nums[nums[nums[x]]], ....

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

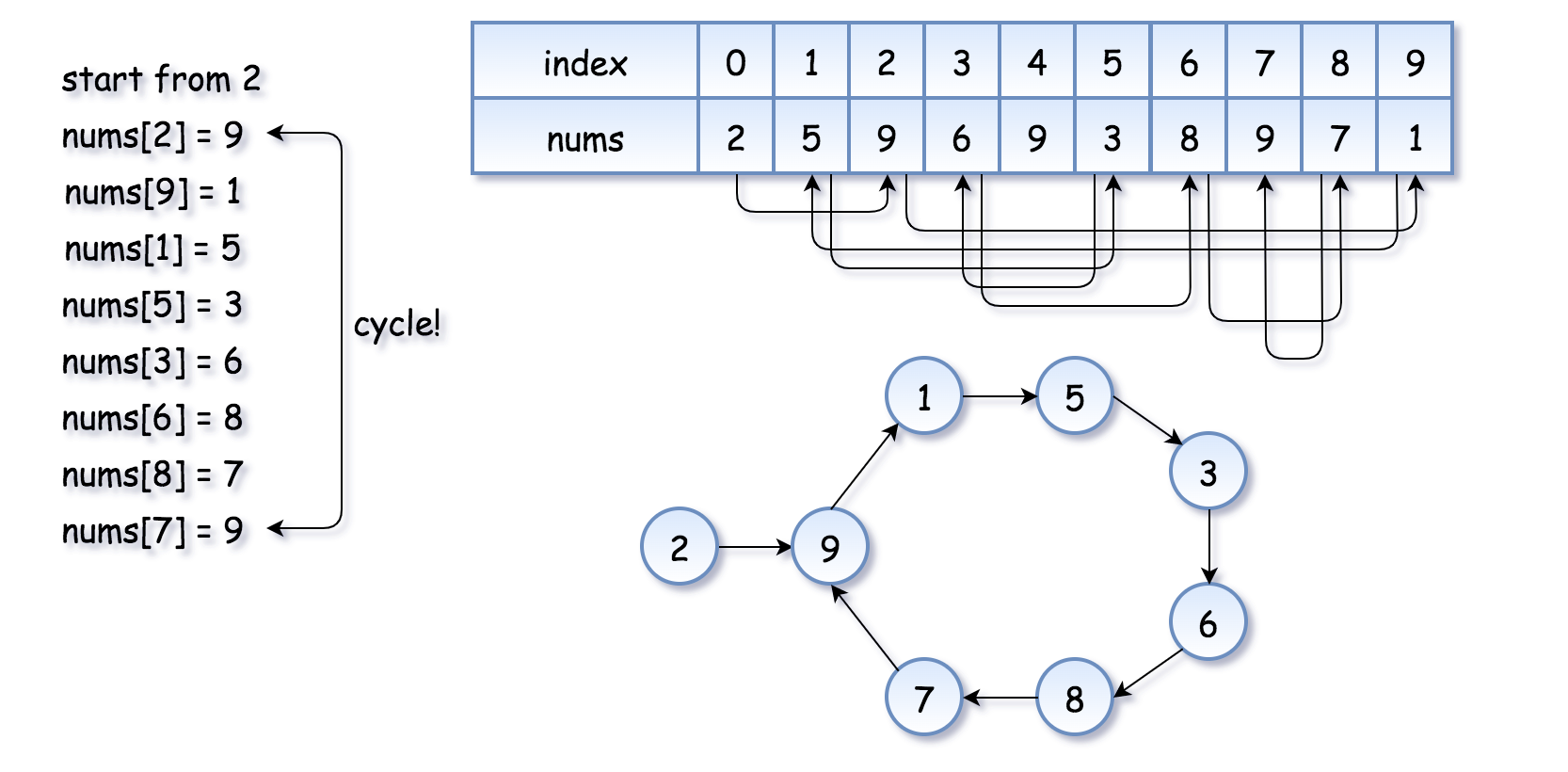
If one starts from x = nums[0], such a sequence will produce a linked list with a cycle.

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](https://leetcode.com/sushant_chaudhari))



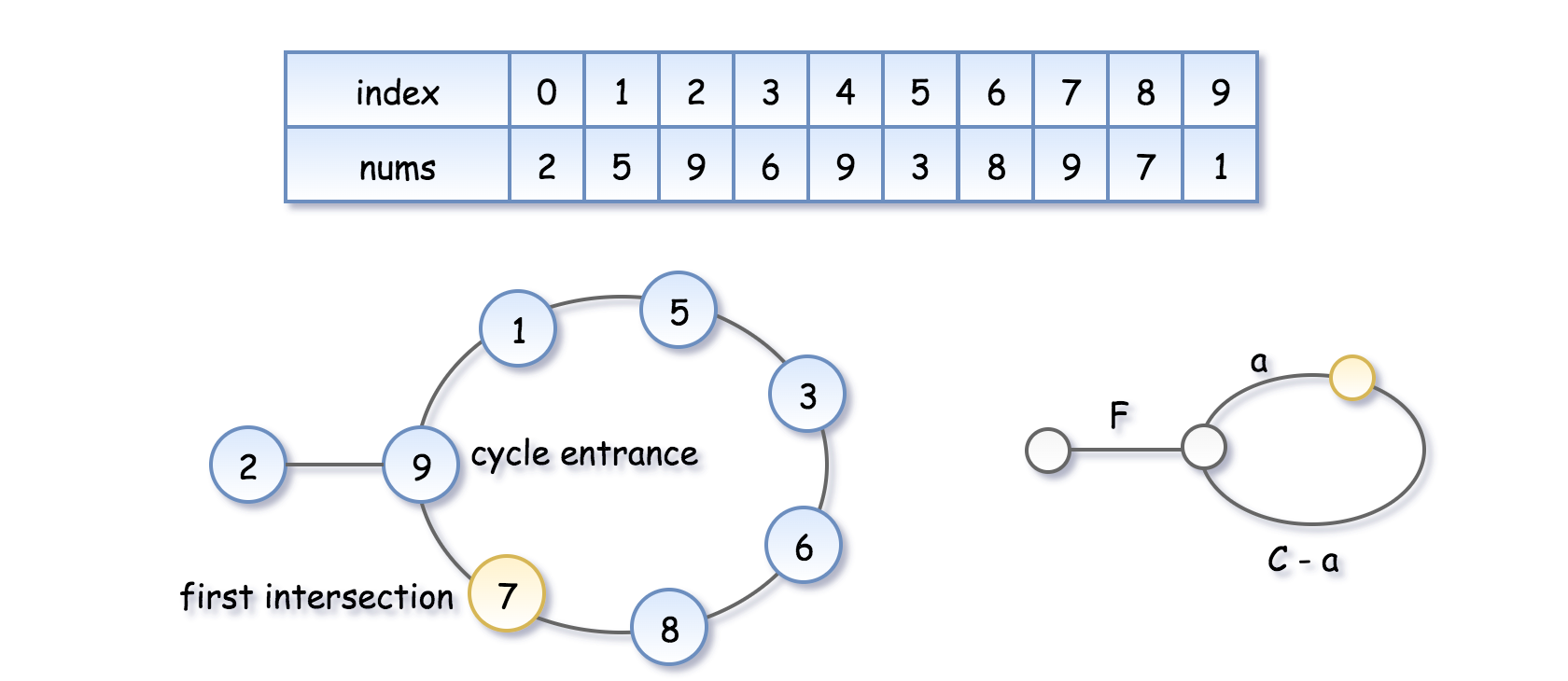
Now the problem is to find the entrance of the cycle.

**Algorithm**

[Floyd's algorithm](https://en.wikipedia.org/wiki/The_Tortoise_and_the_Hare) 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.

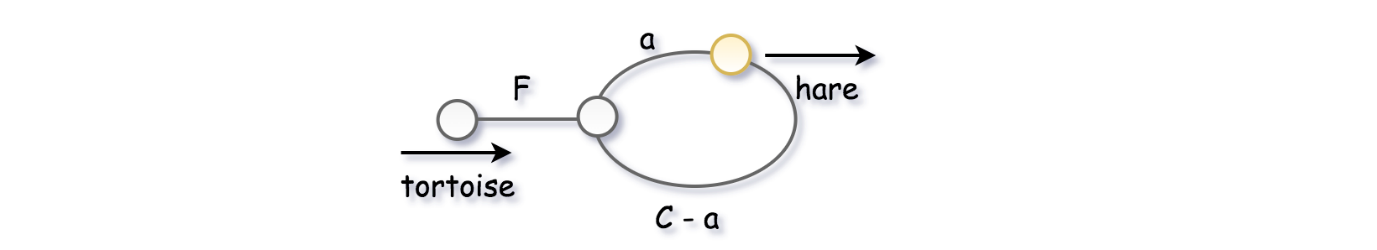


To compute the intersection point, let's note that the hare has traversed twice as many nodes as the tortoise, i.e. 2d(\text{tortoise}) = d(\text{hare})2*d*(tortoise)=*d*(hare), that means

2(F + a) = F + nC + a2(*F*+*a*)=*F*+*nC*+*a*, where n*n* is some integer.

Hence the coordinate of the intersection point is F + a = nC*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*F* steps.

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

**Implementation**

class Solution {

public int findDuplicate(int[] nums) {

// Find the intersection point of the two runners.

int tortoise = nums[0];

int hare = nums[0];

do {

tortoise = nums[tortoise];

hare = nums[nums[hare]];

} while (tortoise != hare);

// Find the "entrance" to the cycle.

tortoise = nums[0];

while (tortoise != hare) {

tortoise = nums[tortoise];

hare = nums[hare];

}

return hare;

}

}