Solution: Find The Duplicate Number

Let's solve the Find The Duplicate Number problem using the Fast and Slow Pointers pattern.

We'll cover the following

Statement

Given an unsorted array of positive numbers, `nums`, such that the values lie in the range $[1, n]$, inclusive, and that there are $n + 1$ numbers in the array, find and return the duplicate number present in `nums`. There is only one repeated number in `nums`.

> **Note:** You cannot modify the given array `nums`. You have to solve the problem using only constant extra space.

Constraints:

- $1 \leq n \leq 10^5$
- `nums.length` $= n + 1$
- $1 \leq$ `nums[i]` $\leq n$
- All the integers in `nums` are unique except for one integer that will appear more than once.

Solution

We solve this problem using fast and slow pointers technique, without modifying the `nums` array and using only constant extra space.

For this problem, the duplicate number will create a cycle in the `nums` array. The cycle in the `nums` array helps identify the duplicate number.

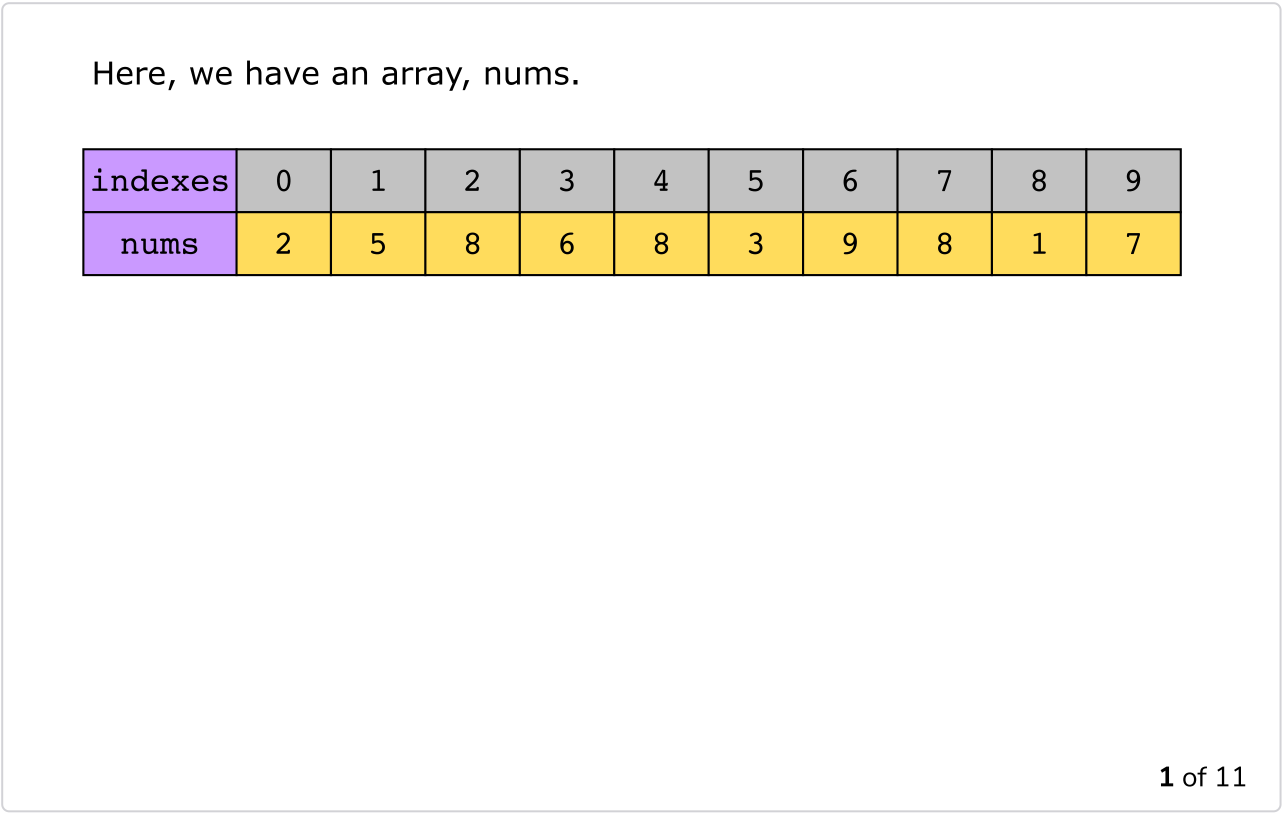
To find the cycle, we'll move in the `nums` array using the $f(x) = nums[x]$ function, where x is the index of the array. This function constructs the following sequence to move:

$$x,\ nums[x],\ nums[nums[x]],\ nums[nums[nums[x]]],\ \ldots$$

In the sequence above, every new element is an element in `nums` present at the index of the previous element.

Let's say we have an array, $[2,\ 3,\ 1,\ 3]$. We'll start with $x = nums[0]$, which is 2, present at the 0^{th} index of the array and then move to $nums[x]$, which is 1, present at the 2^{nd} index. Since we found 1 at the 2^{nd} index, we'll move to the 1^{st} index, and so on. This example shows that if we're given an array of length $n - 1$, with values in the range $[1,\ n]$, we can use this traversal technique to visit all the locations in the array.

The following example illustrates this traversal:


Here, we have an array, nums.

indexes	0	1	2	3	4	5	6	7	8	9
nums	2	5	8	6	8	3	9	8	1	7

1 of 11

As seen above, we've found that there is a cycle in an array with a duplicate number. Now, the problem is to find the entry point of the cycle, which will be our duplicate number.

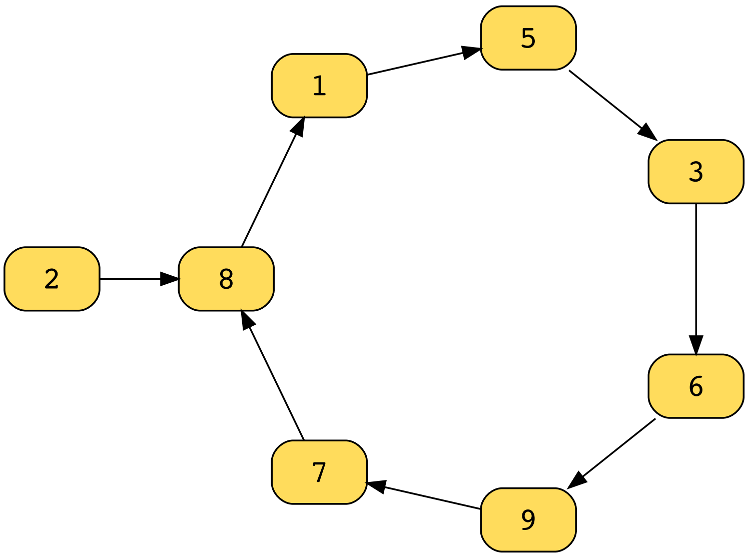
We solve this problem in two parts using the `slow` and `fast` pointers.

In the first part, the `slow` pointer moves once, while the `fast` pointer moves twice as fast as the `slow` pointer until both of the pointers meet each other. Since the `fast` pointer is moving twice as fast as the `slow` pointer, it will be the first one to enter and move around the cycle. At some point after the `slow` pointer also enters and moves in the cycle, the `fast` pointer will meet the `slow` pointer. This will be the intersection point.

> **Note:** The *intersection* point of the two pointers is, in the general case, not the *entry* point of the cycle.

Let’s look at a visual representation of the first part of our solution:

This graph shows the graphical presentation of array.



1 of 9

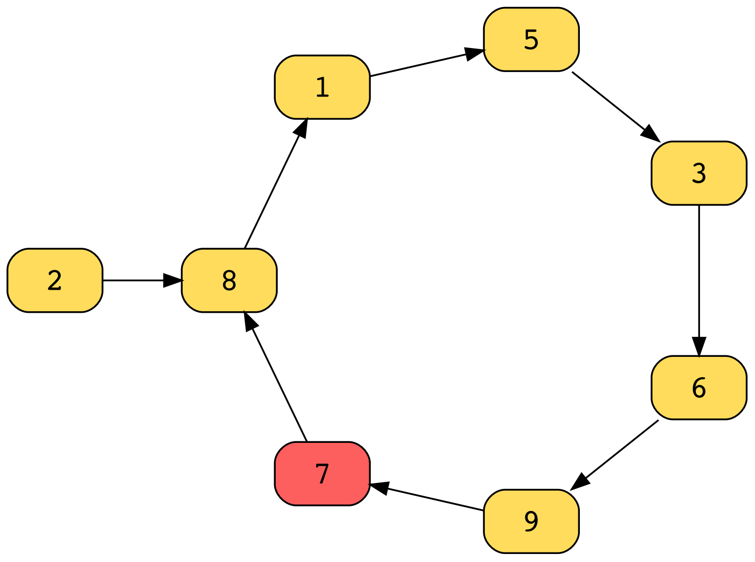
In **part two**, we’ll start moving again in the cycle, but this time, we’ll slow down the `fast` pointer so that it moves with the same speed as the `slow` pointer.

Let’s look at the journeys of the two pointers in part two:

- The `slow` pointer will start from the 0^{th} position.
- The `fast` pointer will start from the intersection point.
- After a certain number of steps, let’s call it F, the `slow` pointer meets the `fast` pointer. This is the ending point for both pointers.
- This common ending position will be the entry point of the cycle.

Let’s look at the visual presentation of the second part of our solution:

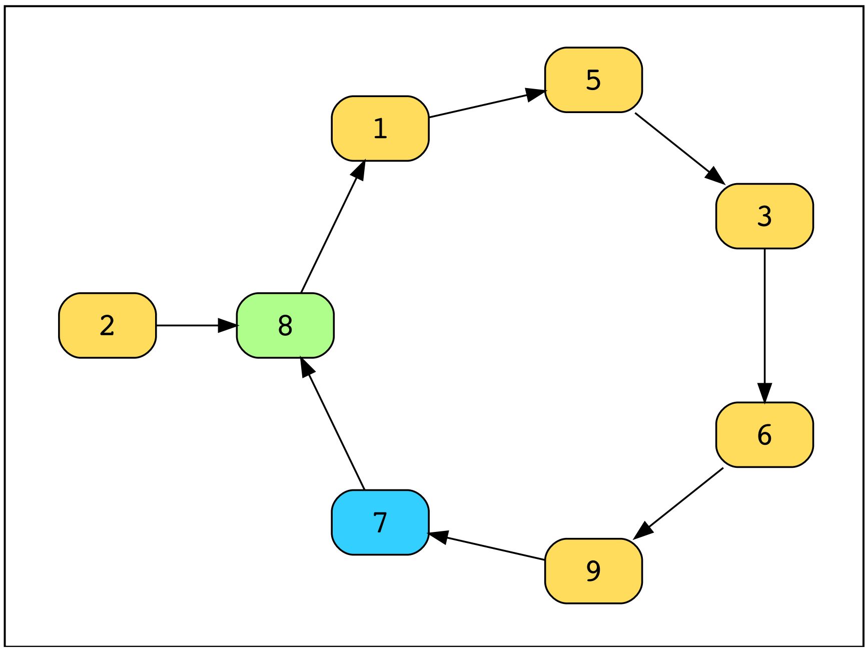
We have found the intersection point, which is 7. So, the first part of the solution ends here.
Now, we'll move to the second part of the solution.



1 of 3

Now, let’s try to understand how it is that our solution is able to *always* locate the entry point of the cycle.

Let’s return to the example we just discussed, using this graphical representation:



A graphical presentation of the array

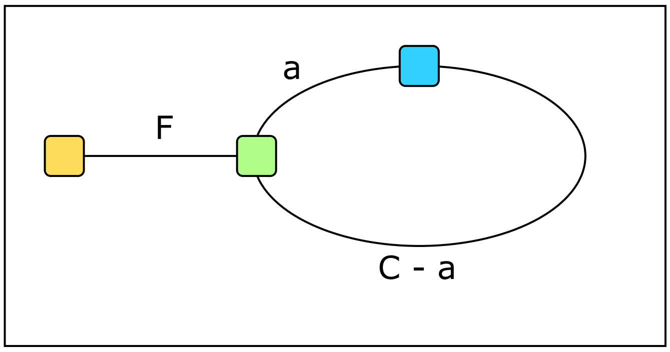
- 7 is the *intersection* point where the `slow` and `fast` pointers will meet.
- 8 is the *entry* point of the cycle, which is our duplicate number.

The `fast` pointer is traversing two times faster than the `slow` pointer. This can be represented by the following equation:

$$d_{fast} = 2d_{slow} \text{ ——— } (1)$$

Here, d represents the number of elements traversed.

Let's look at the following diagram to see the steps taken by the `slow` and `fast` pointers from the starting point to the intersection point:



A list with a cycle

In the diagram above:

- Green represents the entry point of the cycle.
- Blue represents the intersection point.
- Yellow represents the starting point.
- F represents the steps taken from the starting point to the entry point.
- a represents the steps taken to reach the intersection point from the entry point.
- C represents the cycle length, in terms of the number of steps taken to go once around the cycle.

With this setup in mind, let's see the distance traveled by the `slow` and `fast` pointers.

The `slow` pointer travels F steps from the starting point to the entry point of the cycle and then takes a steps from the entry point to the intersection point of the cycle, that is, the point where both pointers intersect. So, we can express the distance traveled by the `slow` pointer in the form of this equation:

$$d_{slow} = F + a \text{ ——— } (2)$$

In the time it takes the `slow` pointer to travel $F + a$ steps, the `fast` pointer, since it's traveling twice as fast as the `slow` pointer, will have also traveled around the cycle at least once. So, we can say the `fast` pointer, first, travels F steps from the starting point to the entry point of the cycle, then travels at least a cycle, and at the end travels a steps from the entry point to the intersection point of the cycle. Now, we can express the distance traveled by the `fast` pointer as the following equation:

$$d_{fast} = F + C + a \text{ ——— } (3)$$

Recall eq. (1) :

$d_{fast} = 2d_{slow}$ ——— (1)

If we substitute the equivalent expression of d_{slow} given in eq. (2) and the equivalent expression of d_{fast} given in eq. (3) into eq. (1), we get:

$F + C + a = 2(F + a)$

Let's simplify this equation:

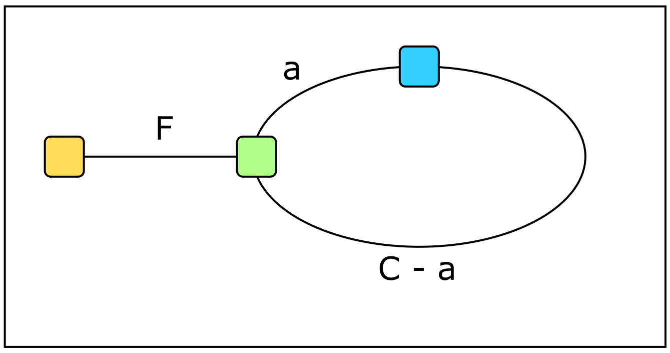
$C = F + a$

Therefore, the distance from the starting point to the intersection point, $F + a$, equals C.

We can also re-arrange this equality as follows:

$F = C - a$

Let's consult our diagram again:



A list with a cycle

As we can see, $C - a$ is, in fact, the distance from the intersection point back to the entry point. This illustrates why, when we move one pointer forward, starting at the intersection point, and another pointer from the starting point, the point where they meet is the entry point of the cycle.

> **Note:** The proof above does not consider the case where F is longer than the length of the cycle. In this situation, it's possible that the `fast` pointer will go around the cycle more than once. To express this more general case, we can say that the distance covered by the `fast` pointer from the entry point to the intersection point is: $F + nC + a$, where n is a positive integer. As a result, our substitution will take this form: $F + nC + a = 2(F + a)$, which simplifies to $nC = F + a$, that is: $F = nC - a$. This simply means that after going around the cycle n times, the `fast` pointer will still be a steps behind the entry point of the cycle.

Let's look at the implementation of the solution discussed above:

Java

```java
class FindDuplicate {
    public static int findDuplicate(int[] nums) {
        // Intialize the fast and slow pointers and make them point the fi
        // element of the array
        int fast = nums[0];
        int slow = nums[0];
        // PART #1
        // Traverse in array until the intersection point is found
        while (true) {
            // Move the slow pointer using the nums[slow] flow
            slow = nums[slow];
            // Move the fast pointer two times fast as the slow pointer us
            // nums[nums[fast]] flow
            fast = nums[nums[fast]];
            // Break the loop when slow pointer becomes equal to the fast
            // if the intersection is found
            if (slow == fast) {
                break;
            }

        }
        // PART #2
        // Make the slow pointer point the starting position of an array a
        // start the slow pointer from starting position
        slow = nums[0];
        // Traverse the array until the slow pointer becomes equal to the
        // fast pointer
```