Find the Missing Number (easy)

We'll cover the following

Problem Statement

We are given an array containing `n` distinct numbers taken from the range `0` to `n`. Since the array has only `n` numbers out of the total `n+1` numbers, find the missing number.

Example 1:

```
Input: [4, 0, 3, 1]
Output: 2
```

Example 2:

```
Input: [8, 3, 5, 2, 4, 6, 0, 1]
Output: 7
```

Try it yourself

Try solving this question here:

Java | Python3 | JS | C++

```
class MissingNumber {

  public static int findMissingNumber(int[] nums) {
    // TODO: Write your code here
    return -1;
  }
}
```

Test | Save | Reset

Solution

This problem follows the **Cyclic Sort** pattern. Since the input array contains unique numbers from the range `0` to `n`, we can use a similar strategy as discussed in Cyclic Sort to place the numbers on their correct index. Once we have every number in its correct place, we can iterate the array to find the index which does not have the correct number, and that index will be our missing number.

However, there are two differences with Cyclic Sort:

1. In this problem, the numbers are ranged from `0` to `n`, compared to `1` to `n` in the Cyclic Sort. This will result in two changes in our algorithm:
   - In this problem, each number should be equal to its `index`, compared to `index - 1` in the Cyclic Sort; this means => `nums[i] == nums[nums[i]]`
   - Since the array will have `n` numbers, which means array indices will range from `0` to `n-1`. Therefore, we will ignore the number `n` as we can't place it in the array, so => `nums[i] < nums.length`
2. Say we are at index `i`. If we swap the number at index `i` to place it at the correct index, we can still have the wrong number at index `i`. This was true in Cyclic Sort too. It didn't cause any problems in Cyclic Sort as over there, we made sure to place one number at its correct place in each step, but that wouldn't be enough in this problem as we have one extra number due to the larger range. Therefore, before swapping we will check if the number at index `i` is within the permissible range i.e., it is less than the length of the input array, if not, we will skip ahead.

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
class MissingNumber {

  public static int findMissingNumber(int[] nums) {
    int i = 0;
    while (i < nums.length) {
      if (nums[i] < nums.length && nums[i] != nums[nums[i]])
        swap(nums, i, nums[i]);
      else
        i++;
    }

    // find the first number missing from its index, that will be our required number
    for (i = 0; i < nums.length; i++)
      if (nums[i] != i)
        return i;

    return nums.length;
  }

  private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
  }

  public static void main(String[] args) {
    System.out.println(MissingNumber.findMissingNumber(new int[] { 4, 0, 3, 1 }));
    System.out.println(MissingNumber.findMissingNumber(new int[] { 8, 3, 5, 2, 4, 6, 0, 1 }));
  }
}
```

Run | Save | Reset

Time complexity

The time complexity of the above algorithm is $O(n)$. In the `while` loop, although we are not incrementing the index `i` when swapping the numbers, this will result in more than `n` iterations of the loop, but in the worst-case scenario, the `while` loop will swap a total of `n-1` numbers and once a number is at its correct index, we will move on to the next number by incrementing `i`. In the end, we iterate the input array again to find the first number missing from its index, so overall, our algorithm will take $O(n) + O(n-1) + O(n)$ which is asymptotically equivalent to $O(n)$.

Space complexity

The algorithm runs in constant space $O(1)$.