Previous Next

448. Find All Numbers Disappeared in an Array

Oct. 13, 2019 | 17K views

Average Rating: 4.83 (24 votes)

Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.

Find all the elements of [1, *n*] inclusive that do not appear in this array.

Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.

Example:

```
Input:
[4,3,2,7,8,2,3,1]

Output:
[5,6]
```

Solution

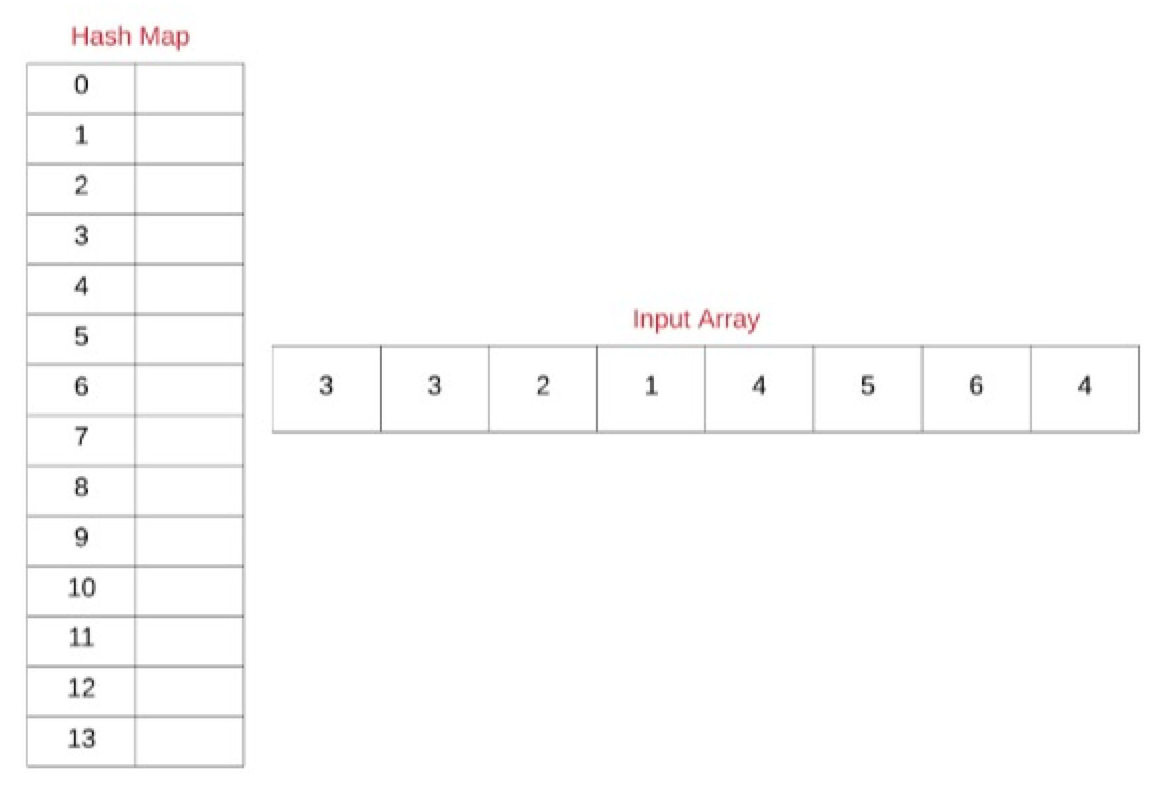
Approach 1: Using Hash Map

Intuition

The intuition behind using a hash map is pretty clear in this case. We are given that the array would be of size `N` and it should contain numbers from `1` to `N`. However, some of the numbers are missing. All we have to do is keep track of which numbers we encounter in the array and then iterate from $1 \cdots N$ and check which numbers did not appear in the hash table. Those will be our missing numbers. Let's look at a formal algorithm based on this idea and then an animation explaining the same with the help of a simple example.

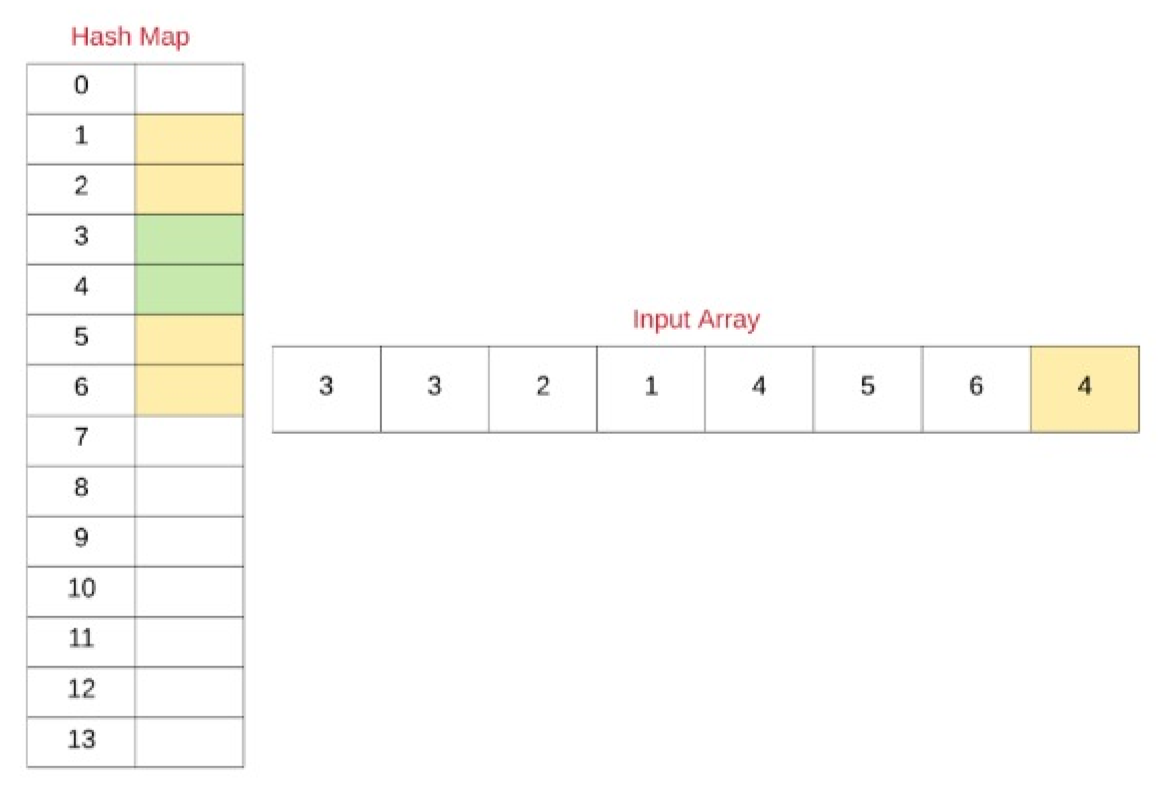
Algorithm

1. Initialize a hash map, `hash` to keep track of the numbers that we encounter in the array. Note that we can use a `set` data structure as well in this case since we are not concerned about the frequency counts of elements.



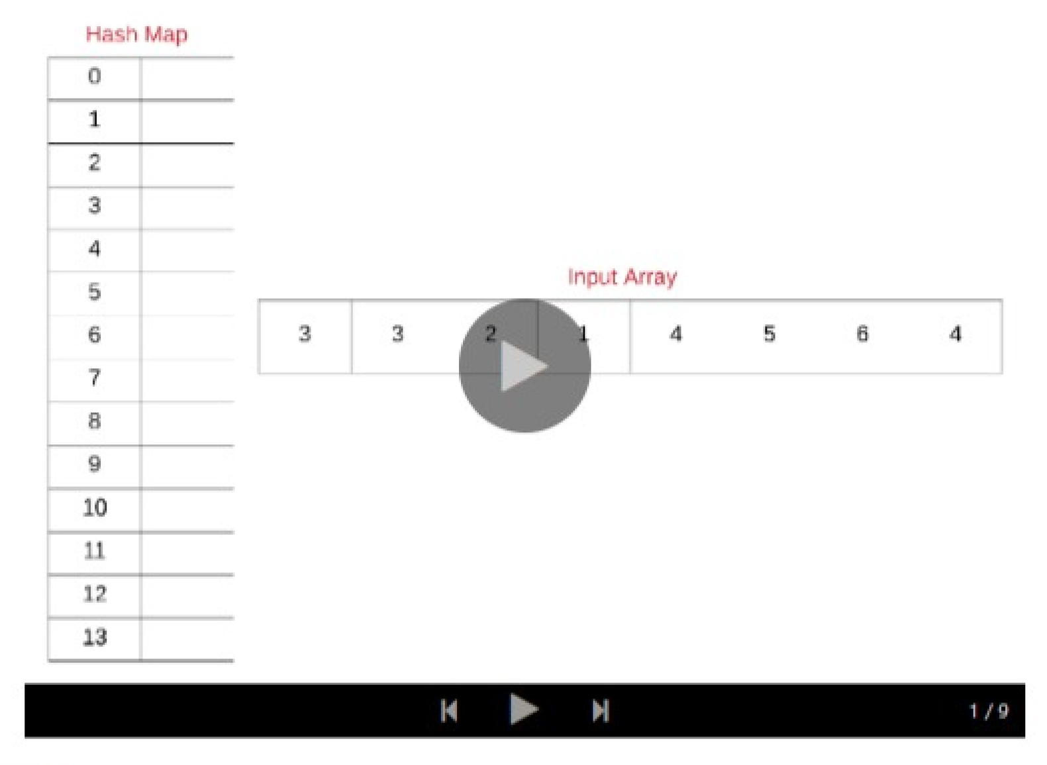
> Note that for the purposes of illustration, we have use a hash map of size 14 and have ordered the keys of the hash map from 0 to 14. Also, we will be using a simple hash function that directly maps the array entries to their corresponding keys in the hash map. Usually, the mapping is not this simple and is dependent upon the hash function being used in the implementation of the hash map.

2. Next, iterate over the given array one element at a time and for each element, insert an entry in the hash map. Even if an entry were to exist before in the hash map, it will simply be over-written. For the above example, let's look at the final state of the hash map once we process the last element of the array.



3. Now that we know the `unique` set of elements from the array, we can simply find out the missing elements from the range $1 \cdots N$.

4. Iterate over all the numbers from $1 \cdots N$ and for each number, check if there's an entry in the hash map. If there is no entry, add that missing number to a result array that we will return from the function eventually.



Java | Python | Copy

```python
class Solution(object):
    def findDisappearedNumbers(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """

        # Hash table for keeping track of the numbers in the array
        # Note that we can also use a set here since we are not
        # really concerned with the frequency of numbers.
        hash_table = {}

        # Add each of the numbers to the hash table
        for num in nums:
            hash_table[num] = 1

        # Response array that would contain the missing numbers
        result = []

        # Iterate over the numbers from 1 to N and add all those
        # that don't appear in the hash table.
        for num in range(1, len(nums) + 1):
            if num not in hash_table:
                result.append(num)

        return result
```

Complexity Analysis

- Time Complexity : $O(N)$
- Space Complexity : $O(N)$

Approach 2: O(1) Space InPlace Modification Solution

Intuition

We definitely need to keep track of all the `unique` numbers that appear in the array. However, we don't want to use any extra space for it. This solution that we will look at in just a moment springs from the fact that

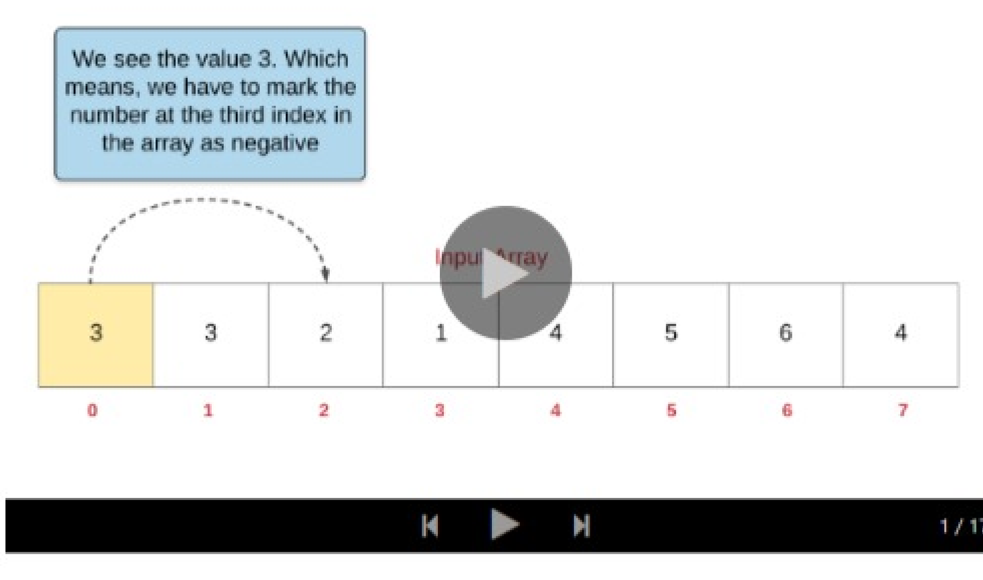
> All the elements are in the range [1, N]

Since we are given this information, we can make use of the input array itself to somehow `mark visited` numbers and then find our missing numbers. Now, we don't want to change the actual data in the array but who's stopping us from changing the `magnitude` of numbers in the array? That is the basic idea behind this algorithm.

> We will be negating the numbers seen in the array and use the sign of each of the numbers for finding our missing numbers. We will be treating numbers in the array as indices and mark corresponding locations in the array as negative.

Algorithm

1. Iterate over the input array one element at a time.
2. For each element `nums[i]`, mark the element at the corresponding location negative if it's not already marked so i.e. $nums[\ nums[i]\ - 1\ ] \times -1$.
3. Now, loop over numbers from $1 \cdots N$ and for each number check if `nums[j]` is negative. If it is negative, that means we've seen this number somewhere in the array.
4. Add all the numbers to the resultant array which don't have their corresponding locations marked as negative in the original array.



Java | Python | Copy

```python
class Solution(object):
    def findDisappearedNumbers(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """

        # Iterate over each of the elements in the original array
        for i in range(len(nums)):

            # Treat the value as the new index
            new_index = abs(nums[i]) - 1

            # Check the magnitude of value at this new index
            # If the magnitude is positive, make it negative
            # thus indicating that the number nums[i] has
            # appeared or has been visited.
            if nums[new_index] > 0:
                nums[new_index] *= -1

        # Response array that would contain the missing numbers
        result = []

        # Iterate over the numbers from 1 to N and add all those
        # that have positive magnitude in the array
        for i in range(1, len(nums) + 1):
            if nums[i - 1] > 0:
```

Complexity Analysis

- Time Complexity : $O(N)$
- Space Complexity : $O(1)$ since we are reusing the input array itself as a hash table and the space occupied by the output array doesn't count toward the space complexity of the algorithm.

Rate this article: ★★★★★

Previous Next

Comments: 11 Sort By

Type comment here... (Markdown is supported)

Preview Post

roireshef ★ 44 May 14, 2020 6:05 PM

Python 4-liner O(n) runtime with O(1) space using swapping (without negating, IMHO clearer solution)

```
def findDisappearedNumbers(self, nums: List[int]) -> List[int]:
    for idx in range(len(nums)):
        while nums[nums[idx]-1] != nums[idx]: # value at target index misplaced
```

Read More

11 Share Reply

lidaivet ★ 87 May 14, 2020 7:26 AM

I like this question.
But the optimal solution is like one of those either you get it right away, or you need some hint.

6 Share Reply

yh32 ★ 36 December 26, 2019 11:26 PM

Swapping Solution, O(n) runtime, O(1) space:

```
class Solution {
    public List<Integer> findDisappearedNumbers(int[] nums) {
        for(int i = 0; i < nums.length; i++){
```

Read More

4 Share Reply

SHOW 3 REPLIES

treemantan ★ 27 October 22, 2019 3:33 AM

quick and simple

```
return set(list(range(1,len(nums)+1)))-set(nums)
```

2 Share Reply

SHOW 4 REPLIES

DawsonDev ★ 3 October 28, 2019 12:57 AM

This is just a set difference `{1...n} - {nums} = {missing numbers}`. Here's an easy solution in JavaScript:

```
var findDisappearedNumbers = function(nums) {
    let range = [];
```

Read More

1 Share Reply

SHOW 2 REPLIES

KarimullaSuneel ★ 0 July 14, 2020 12:49 AM

```
  Simple solution beats 80%

   newlist = []
   setnums = set(nums)
```

Read More

0 Share Reply

matbot ★ 4 June 20, 2020 2:21 PM

The efficient solution stumped me for a minute. I came up with essentially the opposite method as negation, where N+1 is added to index values. Not as smooth of a solution due to the need to modulo indexes, but a neat example, I think.

```
vector<int> findDisappearedNumbers(vector<int>& nums) {
```

Read More

0 Share Reply

sriharik ★ 170 May 5, 2020 3:35 AM

Both Brute force and optimal solutions in Java:

```
class Solution {
    public List<Integer> findDisappearedNumbers(int[] nums) {
        //Brute Force
```

Read More

0 Share Reply

kumom ★ 7 May 4, 2020 2:29 PM

Very interesting...the original problem is an easy question while the follow-up suddenly becomes a hard one (it's the same trick used for the problem First Missing Positive)

0 Share Reply

knq512412 ★ 0 April 23, 2020 7:02 AM

wow

0 Share Reply