442. Find All Duplicates in an Array

May 25, 2020 | 3.7K views

Previous Next

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

Find all the elements that appear **twice** in this array.

Could you do it without extra space and in O(*n*) runtime?

Example:

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

Output:
[2,3]
```

Solution

Approach 1: Brute Force

Intuition

Check for a second occurrence of every element in the rest of the array.

Algorithm

When we iterate over the elements of the input array, we can simply look for any other occurrence of the current element in the rest of the array.

Since an element can only occur once or twice, we don't have to worry about getting duplicates of elements that appear twice: + **Case - A:** If an element occurs only once in the array, when you look for it in the rest of the array, you'll find nothing. + **Case - B:** If an element occurs twice, you'll find the second occurrence of the element in the rest of the array. When you chance upon the second occurrence in a later iteration, it'd be the same as **Case - I** (since there are no more occurrences of this element in the rest of the array).

C++ Java Copy

```
class Solution {
  public:
    vector<int> findDuplicates(vector<int>& nums) {
      vector<int> ans;

      for (int i = 0; i < nums.size(); i++)
        for (int j = i + 1; j < nums.size(); j++) {
          if (nums[i] == nums[j]) {
            ans.push_back(nums[i]);
            break;
          }
        }

      return ans;
    }
};
```

Complexity Analysis

- Time complexity : $O(n^2)$. \ \ For each element in the array, we search for another occurrence in the rest of the array. Hence, for the i^{th} element in the array, we might end up looking through all $n - i$ remaining elements in the worst case. So, we can end up going through about n^2 elements in the worst case. \ \ $n - 1 + n - 2 + n - 3 + ... + 1 + 0 = \sum_{i}^{n}(n - i) \simeq n^2$

- Space complexity : No extra space required, other than the space for the output list.

Approach 2: Sort and Compare Adjacent Elements

Intuition

After sorting a list of elements, all elements of equivalent value get placed together. Thus, when you sort an array, equivalent elements form contiguous blocks.

Algorithm

1. Sort the array.
2. Compare every element with it's neighbors. If an element occurs more than once, it'll be equal to at-least one of it's neighbors.

To *simplify*: 1. Compare every element with its predecessor. + Obviously the first element doesn't have a predecessor, so we can skip it. 2. Once we've found a match with a predecessor, we can skip the next element entirely! + **Why?** Well, if an element matches with its predecessor, it cannot possibly match with its successor *as well*. Thus, the next iteration (i.e. comparison between the next element and the current element) can be safely skipped.

C++ Java Copy

```
class Solution {
  public:
    vector<int> findDuplicates(vector<int>& nums) {
      vector<int> ans;

      sort(nums.begin(), nums.end());

      for (int i = 1; i < nums.size(); i++)
        if (nums[i] == nums[i - 1]) {
          ans.push_back(nums[i]);
          i++;              // skip over next element
        }

      return ans;
    }
};
```

Complexity Analysis

- Time complexity : $O(n \log n) + O(n) \simeq O(n \log n)$.

- A performant comparison-based sorting algorithm will run in $O(n \log n)$ time. Note that this can be reduced to $O(n)$ using a special sorting algorithm like Radix Sort.

- Traversing the array after sorting takes linear time i.e. $O(n)$.

- Space complexity : No extra space required, other than the space for the output list. Sorting can be done in-place.

Approach 3: Store Seen Elements in a Set / Map

Intuition

In Approach 1 we used two loops (one nested within the other) to look for two occurrences of an element. In almost all similar situations, you can usually substitute one of the loops with a set / map. Often, it's a worthy trade-off! **for a bit of extra memory, you can reduce the order of your runtime complexity.**

Algorithm

We store all elements that we've seen till now in a map / set. When we visit an element, we query the map / set to figure out if we've seen this element before.

C++ Java Copy

```
class Solution {
  public:
    vector<int> findDuplicates(vector<int>& nums) {
      vector<int> ans;
      unordered_set<int> seen;

      for (auto& num : nums) {
        if (seen.count(num) > 0)
          ans.push_back(num);
        else
          seen.insert(num);
      }

      return ans;
    }
};
```

Complexity Analysis

- Time complexity : $O(n)$ average case. $O(n^2)$ worst case.

- It takes a linear amount of time to iterate through the array.

- Lookups in a hashset are constant time on average, however those can degrade to linear time in the worst case. Note that an alternative is to use tree-based sets, which give logarithmic time lookups always.

- Space complexity : Upto $O(n)$ extra space required for the set.

- If you are tight on space, you can significantly reduce your physical space requirements by using bitsets [1] instead of sets. This data-structure requires just one bit per element, so you can be done in just *n* bits of data for elements that go up-to *n*. Of course, this doesn't reduce your space complexity; bitsets still grow linearly with the range of values that the elements can take.

Approach 4: Mark Visited Elements in the Input Array itself

Intuition

All the above approaches have ignored a key piece of information in the problem statement:

> The integers in the input array `arr` satisfy `1 ≤ arr[i] ≤ n`, where `n` is the size of array. [2]

This presents us with two key insights:

1. All the integers present in the array are positive, i.e. `arr[i] > 0` for any valid index `i`. [3]
2. The decrement of any integers present in the array must be an accessible index in the array, i.e. for any integer `x` in the array, `x-1` is a valid index, and thus, `arr[x-1]` is a valid reference to an element in the array. [4]

Algorithm

1. Iterate over the array and for every element `x` in the array, negate the value at index `abs(x)-1`. [5]
   - The negation operation effectively marks the value `abs(x)` as *seen / visited*.

> **Pop Quiz:** Why do we need to use `abs(x)`, instead of `x`?

1. Iterate over the array again, for every element `x` in the array:
   - If the value at index `abs(x)-1` is positive, it must have been negated twice. Thus `abs(x)` must have appeared twice in the array. We add `abs(x)` to the result.
   - In the above case, when we reach the second occurrence of `abs(x)`, we need to avoid fulfilling this condition again. So, we'll additionally negate the value at index `abs(x)-1`.

C++ Java Copy

```
class Solution {
  public:
    vector<int> findDuplicates(vector<int>& nums) {
      vector<int> ans;

      for (auto num : nums)
        nums[abs(num) - 1] *= -1;

      for (auto num : nums)
        if (nums[abs(num) - 1] > 0) {
          ans.push_back(abs(num));
          nums[abs(num) - 1] *= -1;
        }

      return ans;
    }
};
```

> **Pop Quiz:** Can you do this in a single loop?

Definitely! Notice that if an element `x` occurs just once in the array, the value at index `abs(x)-1` becomes negative and remains so for all of the iterations that follow.

1. Traverse through the array. When we see an element `x` for the first time, we'll negate the value at index `abs(x)-1`.
2. But, the next time we see an element `x`, we don't need to negate again! If the value at index `abs(x)-1` is already negative, we know that we've seen element `x` before.

So, now we are relying on a single negation to mark the visited status of an element. This is similar to what we did in Approach 3, except that we are re-using the array (with some smart negations) instead of a separate set.

C++ Java Copy

```
class Solution {
  public:
    vector<int> findDuplicates(vector<int>& nums) {
      vector<int> ans;

      for (auto num : nums) {
        if (nums[abs(num) - 1] > 0) { // seen before
          ans.push_back(abs(num));
          nums[abs(num) - 1] *= -1;
        }
      }

      return ans;
    }
};
```

Complexity Analysis

- Time complexity : $O(n)$. We iterate over the array twice. Each negation operation occurs in constant time.

- Space complexity : No extra space required, other than the space for the output list. We re-use the input array to store visited status.

---

1. C++ provides an excellent `std::bitset` in the standard library. ↩
2. Some readers will notice a similarity with the pigeonhole principle. While this doesn't really come into play in Approach 4, we utilized it indirectly in Approach 3: since some elements appear twice, the number of unique elements is less than the size of the array. If every unique element gets a bucket in our map / set, some buckets are bound to have more than one element in them! ↩
3. Because, `arr[i] >= 1` for any valid index `i` of array `arr`. ↩
4. Because, all elements in the array are integers that lie in the range $[1, n]$ (where *n* is length of the array). Thus, their decrements are integers that lie in the range $[0, n-1]$ (which is precisely the set of valid indices for an array of length *n*). ↩
5. The `abs()` function provides the absolute value. ↩

Rate this article:

Previous Next

Comments: 4 Sort By

Type comment here... (Markdown is supported)

Preview Post

prema-p ★ 11 June 22, 2020 1:38 AM

Pop Quiz: Why do we need to use abs() instead of x?

Here's a crazy idea, how about you explain this since this is a solution article lol?

5 Share Reply

SHOW 1 REPLY

rickfguy ★ 0 May 26, 2020 1:38 AM

If the elements of the array fall within some fixed range, eg: between 0 and 100, a fifth way to detect duplicates using O(1) space and O(N) time is to concatenate two 64 bit integers side by side, treat it like a single 128 bit number and turn on the k'th bit to indicate if the number k was seen already in the array. It only works for small ranges though.

1 Share Reply

SHOW 1 REPLY

vmnstance ★ 0 July 6, 2020 10:20 AM

The last approach is a clever one. Here is another idea - if we're OK with modifying the input array itself, we could leverage the fact that each element can be placed on the "right" spot only once. I.e. (assuming 1-based indexing) if there would be `nums[i] * 1` then any other `i` element in the array would be identified as a duplicate.

Read More

0 Share Reply

monstrosiang ★ 0 July 3, 2020 12:36 AM

sigh this one is simple but tricky.

0 Share Reply