268. Missing Number

Approach #1 Sorting [Accepted]

Intuition

If `nums` were in order, it would be easy to see which number is missing.

Algorithm

First, we sort `nums`. Then, we check the two special cases that can be handled in constant time - ensuring that 0 is at the beginning and that n is at the end. Given that those assumptions hold, the missing number must somewhere between (but not including) 0 and n. To find it, we ensure that the number we expect to be at each index is indeed there. Because we handled the edge cases, this is simply the previous number plus 1. Thus, as soon as we find an unexpected number, we can simply return the expected number.

Java Python3 Copy

```
class Solution {
    public int missingNumber(int[] nums) {
        Arrays.sort(nums);

        // Ensure that n is at the last index
        if (nums[nums.length-1] != nums.length) {
            return nums.length;
        }
        // Ensure that 0 is at the first index
        else if (nums[0] != 0) {
            return 0;
        }

        // If we get here, then the missing number is on the range (0, n)
        for (int i = 1; i < nums.length; i++) {
            int expectedNum = nums[i-1] + 1;
            if (nums[i] != expectedNum) {
                return expectedNum;
            }
        }

        // Array was not missing any numbers
        return -1;
    }
}
```

Complexity Analysis

- Time complexity : $\mathcal{O}(nlgn)$

  The only elements of the algorithm that have asymptotically nonconstant time complexity are the main `for` loop (which runs in $\mathcal{O}(n)$ time), and the `sort` invocation (which runs in $\mathcal{O}(nlgn)$ time for Python and Java). Therefore, the runtime is dominated by `sort`, and the entire runtime is $\mathcal{O}(nlgn)$.

- Space complexity : $\mathcal{O}(1)$ (or $\mathcal{O}(n)$)

  In the sample code, we sorted `nums` in place, allowing us to avoid allocating additional space. If modifying `nums` is forbidden, we can allocate an $\mathcal{O}(n)$ size copy and sort that instead.

Approach #2 HashSet [Accepted]

Intuition

A brute force method for solving this problem would be to simply check for the presence of each number that we expect to be present. The naive implementation might use a linear scan of the array to check for containment, but we can use a `HashSet` to get constant time containment queries and overall linear runtime.

Algorithm

This algorithm is almost identical to the brute force approach, except we first insert each element of `nums` into a set, allowing us to later query for containment in $\mathcal{O}(1)$ time.

Java Python3 Copy

```
class Solution {
    public int missingNumber(int[] nums) {
        Set<Integer> numSet = new HashSet<Integer>();
        for (int num : nums) numSet.add(num);

        int expectedNumCount = nums.length + 1;
        for (int number = 0; number < expectedNumCount; number++) {
            if (!numSet.contains(number)) {
                return number;
            }
        }
        return -1;
    }
}
```

Complexity Analysis

- Time complexity : $\mathcal{O}(n)$

  Because the set allows for $\mathcal{O}(1)$ containment queries, the main loop runs in $\mathcal{O}(n)$ time. Creating `num_set` costs $\mathcal{O}(n)$ time, as each set insertion runs in amortized $\mathcal{O}(1)$ time, so the overall runtime is $\mathcal{O}(n + n) = \mathcal{O}(n)$.

- Space complexity : $\mathcal{O}(n)$

  `nums` contains $n - 1$ distinct elements, so it costs $\mathcal{O}(n)$ space to store a set containing all of them.

Approach #3 Bit Manipulation [Accepted]

Intuition

We can harness the fact that XOR is its own inverse to find the missing element in linear time.

Algorithm

Because we know that `nums` contains n numbers and that it is missing exactly one number on the range $[0..n-1]$, we know that n definitely replaces the missing number in `nums`. Therefore, if we initialize an integer to n and XOR it with every index and value, we will be left with the missing number. Consider the following example (the values have been sorted for intuitive convenience, but need not be):

Index	0	1	2	3
Value	0	1	3	4

$$\begin{aligned} missing &= 4 \wedge (0 \wedge 0) \wedge (1 \wedge 1) \wedge (2 \wedge 3) \wedge (3 \wedge 4) \\ &= (4 \wedge 4) \wedge (0 \wedge 0) \wedge (1 \wedge 1) \wedge (3 \wedge 3) \wedge 2 \\ &= 0 \wedge 0 \wedge 0 \wedge 0 \wedge 2 \\ &= 2 \end{aligned}$$

Java Python3 Copy

Complexity Analysis

- Time complexity : $\mathcal{O}(n)$

  Assuming that XOR is a constant-time operation, this algorithm does constant work on n iterations, so the runtime is overall linear.

- Space complexity : $\mathcal{O}(1)$

  This algorithm allocates only constant additional space.

Approach #4 Gauss' Formula [Accepted]

Intuition

One of the most well-known stories in mathematics is of a young Gauss, forced to find the sum of the first 100 natural numbers by a lazy teacher. Rather than add the numbers by hand, he deduced a closed-form expression (https://brilliant.org/wiki/sum-of-n-n2-or-n3/) for the sum, or so the story goes. You can see the formula below:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

Algorithm

We can compute the sum of `nums` in linear time, and by Gauss' formula, we can compute the sum of the first n natural numbers in constant time. Therefore, the number that is missing is simply the result of Gauss' formula minus the sum of `nums`, as `nums` consists of the first n natural numbers minus some number.

Java Python3 Copy

Complexity Analysis

- Time complexity : $\mathcal{O}(n)$

  Although Gauss' formula can be computed in $\mathcal{O}(1)$ time, summing `nums` costs us $\mathcal{O}(n)$ time, so the algorithm is overall linear. Because we have no information about *which* number is missing, an adversary could always design an input for which any algorithm that examines fewer than n numbers fails. Therefore, this solution is asymptotically optimal.

- Space complexity : $\mathcal{O}(1)$

  This approach only pushes a few integers around, so it has constant memory usage.