Solution: First Missing Positive

Let's solve the First Missing Positive problem using the Cyclic sort pattern.

We'll cover the following

Statement

Given an unsorted integer array, `nums`, return the smallest missing positive integer. Create an algorithm that runs with an $O(n)$ time complexity and utilizes a constant amount of space.

> **Note:** The smallest missing positive isn't the first positive number that's missing in the range of elements in the input, but the first positive number that's missing if we start from 1.

Constraints:

- $1 \leq$ `nums.length` $\leq 10^5$
- $-2^{31} \leq$ `nums[i]` $\leq 2^{31} - 1$

Solution

So far, you have probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

- The brute force approach would be to search for every positive integer in the array, starting from 1 and incrementing the number to be searched until we find the first missing positive number. It will result in an algorithm with the time complexity of $O(n^2)$.
- A solution that everyone thinks about is sorting the array and then searching for the smallest positive number. However, this also gives us a time complexity of $O(n \log n)$where n is the number of elements present in our array.
- The final approach we would discuss is storing all the numbers in a hash table and returning the first positive value we don't encounter. This solution would fulfill the time complexity condition of $O(n)$ but still requires extra space. That's where cyclic sort comes in.

Optimized approach using cyclic sort

An array of size n containing numbers ranging from 1 to n can be sorted using the cyclic sort approach in $O(n)$. Here, we have two problems. First, the array may have negative numbers as well. Second, the array does not have all consecutive numbers. The first problem can be solved by simply ignoring the negative values. The second problem can be solved by linearly scanning the sorted array in $O(n)$. So, overall, our problem is solved in $O(n)$ time and $O(1)$ space, because cyclic sort sorts the elements in-place with $O(1)$ extra space. Now, we can proceed further and discuss our algorithm.

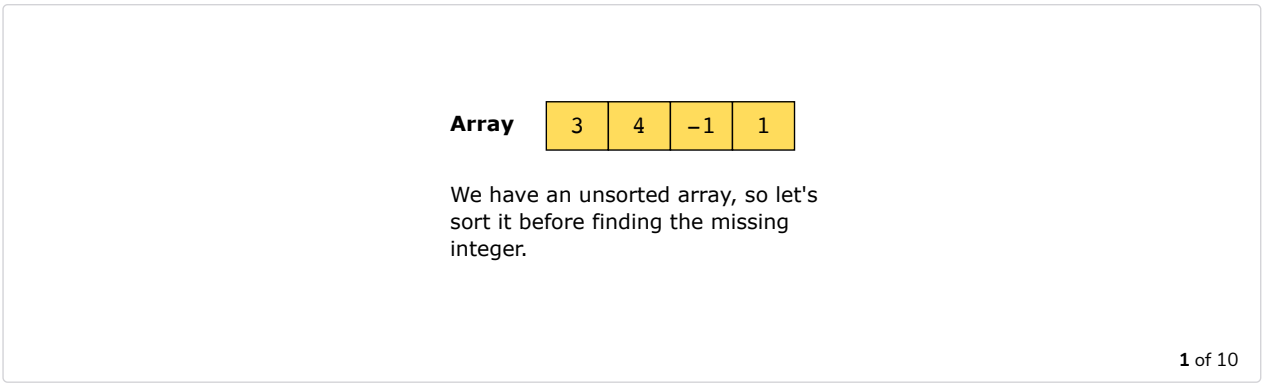
The algorithm iterates over each element of the array, and for each element, it determines the correct position where the element should be placed. It does this by subtracting 1 from the element value because of 0-based indexing. It checks the following conditions for each element after determining its correct position:

- If the current element is in the $[1 - n]$ range, and if it is not already at its correct position, swap the current element with the element at its correct position.
- Otherwise, move on to the next element.

After placing all the elements at the correct positions, traverse the array again and check if the value at each index is equal to its index plus 1. If it is not, return the index plus 1 as the smallest missing positive integer.

If all the values are in order, return the length of the array plus 1 as the smallest missing positive integer.

Let's look at the following example to understand this in a better way:



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

Java

```
15          }
16
17          // Iterate through the array again and check if each element is equal to its index plus 1
18          for (int j = 0; j < nums.length; j++) {
19              if (j + 1 != nums[j]) {
20                  return j + 1;  // Return the smallest missing positive integer
21              }
22          }
23
24          return nums.length + 1;  // If all the elements are in order, return the next positive integer
25      }
26      public static void main(String[] args) {
27          int[][] A = {
28              {1, 2, 3, 4},
29              {-1, 3, 5, 7, 1},
30              {1, 5, 4, 3, 2},
31              {-1, 0, 2, 1, 4},
32              {1, 4, 3}
33          };
34          for (int i = 0; i < A.length; i++) {
35              System.out.print(i + 1);
```

```
36              System.out.println(".\tThe first missing positive integar in the list " + Arrays.toString(A[i
37              System.out.println("\t" + firstMissingPositiveInteger(A[i]));
38              System.out.println(PrintHyphens.repeat("-", 100));
39          }
40      }
41  }
```


Solution summary

1. Iterate over all the elements in `nums` and swap them with their correct position until all elements are in their correct positions.
2. Iterate over `nums` again and check if each element is equal to its index plus 1.
3. If an element is not in the correct position, return the index plus 1, of the element that is out of order, as the smallest missing positive number.
4. If all elements are in order, return the length of the array plus 1 as the smallest missing positive integer.

Time complexity

The algorithm traverses over the array two times, and in both traversals, it iterates over each element exactly once. Therefore, the time complexity of this algorithm is $O(n)$, where n is the length of the `nums`.

Space complexity

The space complexity of the algorithm is $O(1)$ as it uses only a constant amount of space.

Back
First Missing Positive

Next
Find the Corrupt Pair

Mark as Completed