41. First Missing Positive

Hard · Array · Hash Table

Leetcode Link

Problem Description

The LeetCode problem asks for the smallest missing positive integer from an unsorted integer array `nums`. The challenge is to write an algorithm that can find this number efficiently, with the constraint that the algorithm should have a time complexity of `O(n)` and a space complexity of `O(1)`, which means the solution must run in linear time and use a constant amount of extra space.

Intuition

The intuition behind the solution comes from the realization that the smallest missing positive integer must be within the range [1, n+1], where `n` is the length of the array. This is because, in the worst case, the array contains all consecutive positive integers from 1 to n. Hence, the smallest missing positive integer would be just outside this range, which is n + 1.

To find this number within the given constraints, we apply the concept of placing each number in its 'correct position'. If `nums` contains a number `x` where `1 <= x <= n`, `x` should be placed at index `x-1`. By swapping elements to their correct positions (when they are not already there), we aim to have an array where the element at each index `i` is equal to `i + 1`.

Once the placement process is complete, we perform a final linear scan. If we find an index `i` where `nums[i]` is not equal to `i + 1`, then `i + 1` is our missing positive, as the number `i + 1` is not in its correct position (which would be the index `i`). If all elements are in their correct positions, it means our array contains all numbers from 1 to n, so the smallest missing positive integer is `n + 1`.

Solution Approach

The implementation of the solution leverages the fact that we only care about positive integers up to `n`, the length of the array. The key operations are swaps and a final scan to identify which positive integer is missing.

Here's a step-by-step breakdown:

1. Iterate through the array (`nums`), and for each element, perform the following actions:
   - Check if the current element is a positive integer and it's within the range `[1, n]`.
   - Ensure that it is not already in the correct position (meaning the element at the index `nums[i] - 1` should be `nums[i]` itself).
2. If an element meets the above criteria, swap it with the element at its "correct" position (the position it would have if all the elements `[1, n]` were sorted), which is index `nums[i] - 1`. This is done using the helper function `swap(i, j)`.
3. Repeat the process until the current element is out of range or already in the correct position.
4. After the swapping loop, the array is scanned again from the beginning to find the first index `i` where `nums[i]` is not equal to `i + 1`. This index `i` indicates that `i + 1` is the smallest missing positive integer because all the integers before `i + 1` are already in their correct positions, and `i + 1` is the first one that is missing from its correct position.
5. If no such index `i` is found, it means all integers from `1` to `n` are present and in their correct positions, so the smallest missing positive integer is `n + 1`.

The algorithm employs the input array itself as the data structure to store information, which is why it's able to achieve `O(1)` auxiliary space complexity, while still keeping the time complexity at `O(n)` due to the linear number of operations performed.

Example Walkthrough

Let's take an example array to illustrate the solution approach: `nums = [3, 4, -1, 1]`.

1. We start by iterating through the array. The length of the array `n = 4`.
2. The first element `nums[0]` is `3`, which is a positive integer and within the range `[1, n]`. The correct position for `3` is at index `2` (`3 - 1 = 2`). Since `nums[2]` is `-1`, we swap `nums[0]` with `nums[2]`. Now the array looks like this: `nums = [-1, 4, 3, 1]`.
3. Next, we look at the element in the current index `0` which is now `-1`. Since `-1` is negative and not in the range `[1, n]`, we move to the next index.
4. At index `1`, the element is `4`, which is greater than `n` and does not need to be placed within the range `[1, n]`. So we move on.
5. At index `2`, the element is `3` and is already in its correct position (`index 2 + 1 = 3`), so we move forward.
6. At index `3`, we have the number `1`, which should be at index `0` (`1 - 1 = 0`). We swap `nums[3]` with `nums[0]`. The array now becomes: `nums = [1, 4, 3, -1]`.
7. At this point, we have iterated through the entire array, placing all positive integers within the range `[1, n]` in their correct positions.
8. We now perform a final scan through the array. At index `0`, we have `nums[0] = 1`, which is the correct placement.
9. At index `1`, we should have `2`, but instead, we have `nums[1] = 4`. This tells us that `2` is the smallest missing positive integer because it is not at its correct position—which would be index `1`.
10. We conclude that `2` is the smallest missing positive integer in the array `nums`.

The array after processing is `nums = [1, 4, 3, -1]`, and the smallest missing positive integer is `2`.

Python Solution

```python
class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        # Function to swap elements at indices i and j
        def swap_elements(index1, index2):
            nums[index1], nums[index2] = nums[index2], nums[index1]

        # Get the length of the list
        list_length = len(nums)

        # Iterating through the list to place numbers on their correct positions
        for i in range(list_length):
            # Continuously swap the current element until it's in its correct position
            # or it's out of range [1, n]
            while 1 <= nums[i] <= list_length and nums[i] != nums[nums[i] - 1]:
                swap_elements(i, nums[i] - 1)

        # After placing each element in its correct position, or as correct as possible,
        # traverse the list to find the first missing positive integer
        for i in range(list_length):
            # If the current number isn't the right number at index i, return i + 1,
            # because it is the first missing positive integer
            if i + 1 != nums[i]:
                return i + 1

        # If all previous positions contain the correct integers,
        # then the first missing positive integer is n + 1
        return list_length + 1

```

Java Solution

```java
class Solution {
    public int firstMissingPositive(int[] nums) {
        int size = nums.length;

        // Iterate over the array elements.
        for (int i = 0; i < size; ++i) {
            // While the current number is in the range [1, size] and it is not in the correct position
            // (Which means nums[i] does not equal to nums[nums[i] - 1])
            while (nums[i] > 0 && nums[i] <= size && nums[i] != nums[nums[i] - 1]) {
                // Swap nums[i] with nums[nums[i] - 1]
                // The goal is to place each number in its corresponding index based on its value.
                swap(nums, i, nums[i] - 1);
            }
        }

        // Now that nums is reorganized, loop through the array
        // to find the first missing positive number.
        for (int i = 0; i < size; ++i) {
            // If the number doesn't match its index (+1 because we are looking for positive numbers),
            // that index (+1) is the first missing positive number.
            if (nums[i] != i + 1) {
                return i + 1;
            }
        }

        // If no missing number found within [1, size], return size + 1 as the first missing positive number
        return size + 1;
    }

    // Helper method to swap two elements in an array
    private void swap(int[] nums, int firstIndex, int secondIndex) {
        int temp = nums[firstIndex];
        nums[firstIndex] = nums[secondIndex];
        nums[secondIndex] = temp;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to find the first missing positive integer in an array.
    int firstMissingPositive(vector<int>& nums) {
        int size = nums.size(); // Get the size of the array

        // Process each element in the array
        for (int i = 0; i < size; ++i) {
            // Continue swapping elements until the current element is out of bounds
            // or it is in the correct position (value at index i should be i + 1)
            while (nums[i] >= 1 && nums[i] <= size && nums[nums[i] - 1] != nums[i]) {
                // Swap the current element to its correct position
                swap(nums[i], nums[nums[i] - 1]);
            }
        }

        // After rearranging, find the first position where the index does not match the value
        for (int i = 0; i < size; ++i) {
            if (nums[i] != i + 1) {
                // If such a position is found, the missing integer is i + 1
                return i + 1;
            }
        }

        // If all positions match, the missing integer is size + 1
        return size + 1;
    }

private:
    // Helper function to swap two elements in the array
    void swap(int& a, int& b) {
        int temp = a;
        a = b;
        b = temp;
    }
};

```

Typescript Solution

```typescript
function firstMissingPositive(nums: number[]): number {
    const size = nums.length; // Store the length of the array

    // Iterate over the array to place each number in its correct position if possible
    for (let currentIndex = 0; currentIndex < size; currentIndex++) {
        // Calculate the target position for the current number
        const targetIndex = nums[currentIndex] - 1;

        // While the current number is in the range of the array indices,
        // is not in its target position, and is not a duplicate,
        // swap it with the number at its target position
        while (
            nums[currentIndex] > 0 &&
            nums[currentIndex] <= size &&
            nums[currentIndex] !== nums[targetIndex] &&
            currentIndex !== targetIndex
        ) {
            // Swap the current number with the number at its target index
            [nums[currentIndex], nums[targetIndex]] = [nums[targetIndex], nums[currentIndex]];
            // Update the target index based on the new number at the current index
            targetIndex = nums[currentIndex] - 1;
        }
    }

    // After reordering, find the first number that is not at its correct index
    // The index + 1 gives the missing positive integer
    for (let index = 0; index < size; index++) {
        if (nums[index] !== index + 1) {
            return index + 1; // +1 to convert index to positive integer
        }
    }

    // If all numbers are at their correct indices, then return the size + 1
    return size + 1;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, which is linear with respect to the number of elements in the array `nums`. Within the first for loop, each element in `nums` that is within the range `[1, n]` is potentially swapped to its corresponding position (where the value `v` should be at index `v - 1`). Even though there is a while loop inside the for loop, each element can be swapped at most once because once an element is in its correct position, it no longer satisfies the while loop condition. Since each element is swapped at most once, the series of swaps can be considered to have a complexity of `O(n)`.

After that, a second for loop runs which again is of complexity `O(n)`. This loop does not contain any other loops or operations that would increase the time complexity. Therefore, considering both loops, the time complexity remains `O(n)`.

Space Complexity

The space complexity of the code is `O(1)`, which is constant space because the code only uses a fixed number of extra variables (for the swap function and for iterating over the elements of the array). The swaps are done in place and do not require any additional space that scales with the input size.