2295. Replace Elements in an Array

Medium Array Hash Table Simulation

Problem Description

In this problem, we are given an array `nums` which contains `n` distinct positive integers, meaning all the elements in the array are unique and start from index 0. Additionally, we are provided with a set of `m` operations, each representing a change we need to apply to the array. Each operation is a pair of integers where the first integer is a number that currently exists in the array and the second integer is the number we want to replace it with. The conditions are such that the first integer definitely exists in the array while the second is guaranteed not to be in the array before the operation is carried out. Our task is to perform all these operations on the array and return the final state of the array after all the replacements have been made.

Key Points:

- Each element in the array is unique.
- Each operation consists of replacing an existing element with a new one that is not already in the array.

The challenge is to apply these operations efficiently, keeping in mind that the naive approach of searching for each element to replace would lead to a less optimal solution.

Intuition

The intuition behind the solution is to optimize the search and replace process. If we try to look for the element to replace linearly every time, it would be inefficient, especially for a large array. Therefore, we use a hash map, in Python it's a dictionary, to keep track of the indices of the current elements in the array.

Here's how we arrive at the solution step by step:

1. Create a hash map (dictionary) to map each value in the array to its index for constant time access. This allows us to quickly find where the replacement should occur without searching the entire array.
2. Iterate over the list of operations. For each operation:
   - Find the index of the current value we need to replace (since it exists in the nums array as per the problem's guarantee).
   - Replace the value with the new one in the array.
   - Update the hash map by changing the mapping of the old value to the new value, essentially changing the key while keeping the value (which is the index) the same.

Solution Approach

The implementation of the solution follows a straightforward approach leveraging a dictionary data structure for fast lookups and updates to the `nums` array based on the operations provided. Here's how the code brings the intuition to life:

1. Initialize a dictionary (hash map) named `d`. Use a dictionary comprehension to map each value `v` of the array `nums` to its corresponding index `i`. This is done in the form of `{v: i for i, v in enumerate(nums)}`, which will allow constant-time access to the indices of elements we need to update.
2. Iterate over all the given operations using a for loop. In each iteration, you are given a pair (a tuple in Python) containing two integers: `[a, b]` where `a` is the value you need to replace in `nums`, and `b` is the value you're going to replace it with.
3. Inside the loop, access the index of value `a` directly from the dictionary `d` – this gives us the exact position in the `nums` array where replacement is to take place, thus avoiding a full array scan. The operation `nums[d[a]]` accesses the element we want to replace and sets it to `b`.
4. Now, update the dictionary for the new value `b` to have the same index as `a` had before. The operation `d[b] = d[a]` ensures that going forward, you can now find the position of `b` just as easily as you could find `a`.
5. After the loop has processed all operations, we return the modified `nums` array, which now reflects all the replacements that have been made.

By using a hash map, the solution ensures that lookup and update operations are done in `O(1)` time, making the overall time complexity of the solution `O(m)`, where `m` is the number of operations, as each operation involves a constant amount of work.

This approach is efficient and avoids the need for repeated array traversal, which would be necessary if we tried to find elements by value rather than by keeping track of their indices.

Example Walkthrough

Let's consider an example to illustrate the solution approach described above. Suppose we have the following input:

The `nums` array is `[5, 3, 9, 7]`, and the list of operations is `[[5, 10], [3, 15], [9, 20]]`. Let's walk through each step of the approach.

1. We initialize a dictionary named `d` that will map each number in `nums` to its index: `{5: 0, 3: 1, 9: 2, 7: 3}`.
2. We begin iterating over the list of operations:
   - The first operation is `[5, 10]`. We look up the index of `5` in `d`, which is `0`. We replace the element at index `0` in the `nums` array with `10`. The `nums` array now looks like `[10, 3, 9, 7]`. We then update `d` to reflect the change: `{10: 0, 3: 1, 9: 2, 7: 3}`.
   - The second operation is `[3, 15]`. We look up the index of `3` in `d`, which is `1`. We replace the element at index `1` in the `nums` array with `15`. The `nums` array now looks like `[10, 15, 9, 7]`. We then update `d` to reflect the change: `{10: 0, 15: 1, 9: 2, 7: 3}`.
   - The third operation is `[9, 20]`. We look up the index of `9` in `d`, which is `2`. We replace the element at index `2` in the `nums` array with `20`. The `nums` array now looks like `[10, 15, 20, 7]`. We then update `d` to reflect the change: `{10: 0, 15: 1, 20: 2, 7: 3}`.
3. After completing all the operations, we end with the final state of the `nums` array, which is `[10, 15, 20, 7]`.

Following this approach, we have efficiently performed all replacements without having to search through the `nums` array multiple times, showcasing the benefits of maintaining a hash map to track indices of array elements.

Solution Implementation

Python

```python
class Solution:
    def arrayChange(self, nums: List[int], operations: List[List[int]]) -> List[int]:
        # Create a dictionary "value_to_index" to track the indices
        # of the elements in "nums" array
        value_to_index = {value: index for index, value in enumerate(nums)}

        # Iterate through each operation in "operations" list
        for old_value, new_value in operations:
            # Retrieve the index of the element we're going to change
            idx = value_to_index[old_value]
            # Update the element in "nums" at the index to the new_value
            nums[idx] = new_value
            # Update the "value_to_index" dictionary to reflect the change
            # Now "new_value" points to the updated index
            value_to_index[new_value] = idx

        # After processing all operations, return the updated "nums" array
        return nums
```

Java

```java
class Solution {

    // This method takes an array of integers and an array of operations
    // and applies the operations to the array.
    public int[] arrayChange(int[] nums, int[][] operations) {
        // Create a HashMap to quickly find the index of each number in 'nums'.
        Map<Integer, Integer> indexMap = new HashMap<>();

        // Fill the map with the numbers' values as keys and their indices as values.
        for (int i = 0; i < nums.length; ++i) {
            indexMap.put(nums[i], i);
        }

        // Iterate through each operation in the operations array.
        for (int[] operation : operations) {
            // Extract 'from' and 'to' values from the current operation.
            int fromValue = operation[0], toValue = operation[1];

            // Get the index of the 'fromValue' number in the 'nums' array.
            int indexToUpdate = indexMap.get(fromValue);

            // Update the number at that index to the 'toValue'.
            nums[indexToUpdate] = toValue;

            // Update the map with the new value ('toValue') pointing to the same index.
            indexMap.put(toValue, indexToUpdate);
        }

        // Return the modified 'nums' array with all operations applied.
        return nums;
    }
}
```

C++

```cpp
class Solution {
public:
    vector<int> arrayChange(vector<int>& nums, vector<vector<int>>& operations) {
        // Create an unordered_map to keep track of each number's index in the array
        unordered_map<int, int> indexMap;
        for (int i = 0; i < nums.size(); ++i) {
            indexMap[nums[i]] = i;
        }

        // Loop over operations
        for (auto& operation : operations) {
            // Extract the original and new values from the operation
            int originalValue = operation[0];
            int newValue = operation[1];

            // Update the nums array at the index where the original value was found
            nums[indexMap[originalValue]] = newValue;

            // Update the indexMap to reflect the index of the new value
            indexMap[newValue] = indexMap[originalValue];
        }

        // Return the modified nums array after all operations have been performed
        return nums;
    }
};
```

TypeScript

```typescript
function arrayChange(inputArray: number[], operations: number[][]): number[] {
    // Create a map to store the value and its corresponding index in the input array
    const indexMap = new Map<number, number>();

    // Populate the map with each number and its index
    inputArray.forEach((value, index) => {
        indexMap.set(value, index);
    });

    // Iterate through the operations
    for (const [oldValue, newValue] of operations) {
        // Get the index of the element to be changed (oldValue)
        const indexToChange = indexMap.get(oldValue);

        // If the index exists, proceed with the update
        if (indexToChange !== undefined) {
            // Update the inputArray at the specific index to the newValue
            inputArray[indexToChange] = newValue;

            // Update the indexMap with the newValue pointing to the same index
            indexMap.set(newValue, indexToChange);
        }
    }

    // Return the modified input array
    return inputArray;
}
```

```python
class Solution:
    def arrayChange(self, nums: List[int], operations: List[List[int]]) -> List[int]:
        # Create a dictionary "value_to_index" to track the indices
        # of the elements in "nums" array
        value_to_index = {value: index for index, value in enumerate(nums)}

        # Iterate through each operation in "operations" list
        for old_value, new_value in operations:
            # Retrieve the index of the element we're going to change
            idx = value_to_index[old_value]
            # Update the element in "nums" at the index to the new_value
            nums[idx] = new_value
            # Update the "value_to_index" dictionary to reflect the change
            # Now "new_value" points to the updated index
            value_to_index[new_value] = idx

        # After processing all operations, return the updated "nums" array
        return nums
```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed by looking at the two major steps in the function: the creation of the dictionary `d` that maps current values to their indices, and the iteration over the `operations` list to apply the value changes.

1. Constructing the dictionary `d` takes `O(n)` time, where `n` is the length of the `nums` array. This is because it involves iterating over each element once.
2. Iterating over the `operations` list takes `O(m)` time, where `m` is the number of operations because each operation involves a constant amount of work: updating a value in the list and updating a single entry in the dictionary `d`.

The total time complexity is the sum of these two parts: `O(n + m)`.

Space Complexity

For space complexity, we consider the extra space used by the algorithm, not including the input and output.

1. The extra space comes from the dictionary `d` that maps values to indices, which contains at most `n` key-value pairs, where `n` is the length of `nums`. Thus, the space complexity for the dictionary is `O(n)`.
2. No additional space that grows with the input size is used during the iteration over `operations`.

Hence, the total space complexity is `O(n)`.