1920. Build Array from Permutation

Easy Array Simulation

Problem Description

The problem provides an array called `nums`, which is a zero-based permutation. This means that `nums` is an array where each number is unique and ranges from `0` to `nums.length - 1`, with no duplicates and no numbers out of this range. The task is to create a new array `ans` with the same length as `nums`. The value at each index `i` in the new array, `ans[i]`, should be determined by finding the value at index `nums[i]` within `nums`. More clearly, for every index `i` from `0` to `nums.length - 1`, `ans[i]` is equal to `nums[nums[i]]`. Overall, the objective is to re-map the array based on the current values pointing to other indices within the same array.

Intuition

To arrive at the solution, we do not need to modify the original array or use additional data structures. The problem can be solved with a straightfoward approach:

- Iterate through each element of the array `nums`.
- For each index `i`, find the element at the index specified by the value of `nums[i]`, which is `nums[nums[i]]`.
- Append this element to our answer array `ans`.

In the language of the given solution in Python, this translates into a simple list comprehension, which is a concise and efficient way of creating a list in Python from another iterable while applying some condition or operation to each element.

The line `return [nums[num] for num in nums]` is Python's way of saying, "for each element `num` in the list `nums`, take `nums[num]`, and put it in a new list," which is exactly the new array `ans` we want to return.

Solution Approach

Since the Reference Solution Approach section provided is empty, I'll elaborate based on the solution code given. The implementation of the problem's solution is straightforward:

- **Algorithm**: The algorithm is a simple iteration over the original array. The permutation property of the array assures us that we will not have out-of-bounds errors or duplicates.
- **Data Structures**: The only data structure used is the list itself in Python (`List[int]`). There is no need for additional data structures like stacks, queues, or hash maps because we're not dealing with operations that require such complexity.
- **Pattern Used**: The pattern used here is direct addressing where indices of the array directly correspond to the values because of the permutation's nature.

Here's a walk-through of the implementation in the given solution:

- We start by using the list comprehension feature in Python, which allows us to construct a new list in one line of code.
- The expression within the list comprehension `[nums[num] for num in nums]` is the core of our solution. `for num in nums` is a loop that goes through each number (as index) in the `nums` array.
- For each iteration, the value `nums[num]` is calculated. Due to the nature of the permutation array (every value in the array is a valid index into the array), `num` is always a valid index of `nums`, and thus `nums[num]` is always a valid operation.
- Each result of `nums[num]` is collected into a new list. This constructed list is immediately returned as the final answer without the necessity of a temporary holder.

The absence of nested loops or complicated branching indicates that this operation is linear in time complexity, essentially `O(n)`, where `n` is the number of elements in the `nums` array. Space complexity is also `O(n)` due to the creation of the `ans` array which holds the same number of elements as `nums`.

Nothing beyond the list itself and basic iteration is used to arrive at the solution, making this a cleanly implemented answer for the given problem.

Example Walkthrough

Let's walkthrough the solution approach with a given example. Suppose we have an input array `nums` defined as follows:

```
nums = [2, 0, 1]
```

Following the problem description, the result array `ans` will be constructed where each of its elements `ans[i]` must be equal to `nums[nums[i]]`. Here's the step-by-step logic to get `ans`:

1. Start with an empty array `ans`.
2. For the first element `nums[0] = 2`, find `nums[nums[0]]`, which is `nums[2]`. Since `nums[2] = 1`, we add `1` to the `ans` array, making it now `ans = [1]`.
3. Move to the second element `nums[1] = 0`, find `nums[nums[1]]`, which is `nums[0]`. Since `nums[0] = 2`, we add `2` to the `ans` array, now `ans = [1, 2]`.
4. Lastly, for the third element `nums[2] = 1`, find `nums[nums[2]]`, which is `nums[1]`. Since `nums[1] = 0`, we add `0` to the `ans` array, resulting in `ans = [1, 2, 0]`.

The final `ans` array is `[1, 2, 0]`, which directly corresponds to the permutation of indices defined by the original `nums` array. Thus, by iterating over the indices of `nums` and accessing the values in the order given by the elements of `nums`, we applied the permutation to create a new array that matches the problem's requirements.

We can convert this step-by-step process into a one-liner in Python using list comprehension:

```
ans = [nums[num] for num in nums]
```

This single line of code loops through `nums`, takes each element as an index (`num`), and fetches the value of `nums` at that index, appending it to the new list. It carries out the steps we manually processed earlier but does so in a concise and efficient way, which is why it is the optimal solution for this problem.

Solution Implementation

Python

```python
from typing import List  # Import the List type from the typing module for type hinting

class Solution:
    def build_array(self, nums: List[int]) -> List[int]:
        # Creating a new list where each element is obtained by accessing the
        # index of the current element in the original list 'nums'
        return [nums[num] for num in nums]  # List comprehension to build the new list
```

Java

```java
class Solution {
    // This method is responsible for constructing a new array based on specific rules.
    public int[] buildArray(int[] nums) {
        // Create an array of the same size as the input array to store the new sequence.
        int[] resultArray = new int[nums.length];

        // Iterate over the input array.
        for (int index = 0; index < nums.length; ++index) {
            // For each position in the result array, find the value at the index
            // specified by the value in the input array at the current position.
            // Assign this value to the current position in the result array.
            resultArray[index] = nums[nums[index]];
        }

        // Return the constructed result array.
        return resultArray;
    }
}
```

C++

```cpp
#include <vector> // Include the vector header for using the vector container.

// Solution class as provided in the original code snippet.
class Solution {
public:
    // Function 'buildArray' takes a vector of integers as input and returns a vector of integers.
    vector<int> buildArray(vector<int>& nums) {
        // Create an empty vector 'result' to store the final output.
        vector<int> result;

        // Use a range-based for loop to iterate over the elements in the 'nums' input vector.
        for (int num : nums) {
            // Access the element of 'nums' indexed by the value of 'num', and append it to 'result'.
            result.push_back(nums[num]);
        }

        // Return the 'result' vector.
        return result;
    }
};
```

TypeScript

```typescript
// Function to build a new array based on the rules given by an input array.
// Each element at index 'i' in the output array is the element at index nums[i]
// in the input array nums.
function buildArray(nums: number[]): number[] {
    // Use the 'map' function to transform each value of the original array.
    // The 'v' represents the value at the current index in the nums array.
    return nums.map((v) => nums[v]); // For each index 'i', place nums[nums[i]] in the new array.
}
```

```python
from typing import List  # Import the List type from the typing module for type hinting

class Solution:
    def build_array(self, nums: List[int]) -> List[int]:
        # Creating a new list where each element is obtained by accessing the
        # index of the current element in the original list 'nums'
        return [nums[num] for num in nums]  # List comprehension to build the new list
```

Time and Space Complexity

The given Python code takes an input list `nums` and returns a new list based on the values from `nums` where the `i-th` element in the new list is `nums[nums[i]]`.

Time Complexity:

The time complexity of the code is determined by the single loop which iterates through the list `nums`. Since we are going through each element exactly once to construct the new list, the time complexity is `O(n)`, where `n` is the number of elements in `nums`.

Space Complexity:

The space complexity includes the space needed for the input and the additional space required by the program. Since the input `nums` is given, we typically don't consider this in the calculation.

The code creates a new list as a result. No additional data structures were created that would depend on the size of the input. Therefore, the space complexity is `O(n)`, where `n` is the length of the resultant list, which is the same as the input `nums`.