1389. Create Target Array in the Given Order

Easy Array Simulation

Problem Description

The problem gives us two arrays: `nums` and `index`. The goal is to construct a new array called `target`. To create this target array, we need to follow specific rules:

- Start with an empty `target` array.
- Read elements from `nums` and `index` arrays from left to right. For each pair `(nums[i], index[i])`, insert the value `nums[i]` into the `target` array at the position specified by `index[i]`.
- Keep inserting elements into the `target` array until there are no more elements to read from `nums` and `index`.

The challenge requires us to return this `target` array after all insertions are complete. It is guaranteed in the problem statement that the insertion operations, as described by the `index` array, will be valid—which means they won't lead to any out-of-bounds errors.

Intuition

The intuition behind the solution is straightforward, as the problem specifies the exact steps needed to arrive at the `target` array. With each pair of elements from `nums` and `index`, we directly follow the rule and use the insert operation.

Here's the intuitive approach step-by-step:

- Start with an empty list for the `target`.
- Loop over `nums` and `index` using the `zip()` function in Python, which conveniently gives us pairs of `(nums[i], index[i])`.
- For each pair, use the `insert()` method on the `target` list, which allows us to place elements not just at the end of the list (like `append`) but at any position we specify.
- The iteration continues until every element from `nums` has been placed into the `target` at the correct positions.
- Finally, return the `target` array, which now contains all the elements from `nums` sorted by the rules of the `index` array.

Solution Approach

The implementation of the solution is straightforward and follows the problem description closely. It utilizes the built-in list data structure in Python and the `insert()` method it provides. The solution does not rely on any sophisticated algorithms or complex patterns; rather, it uses a basic iterative approach that corresponds with the rules defined in the problem.

Here's the detailed implementation description:

- Start by initializing an empty list `target` that will eventually hold the final array.
- Use the built-in Python function `zip()` to iterate over both the `nums` and `index` arrays simultaneously. `zip(nums, index)` creates an iterator of tuples where the first item in each passed iterator is paired together, then the second item, and so on. In this case, it pairs each element in `nums` with its corresponding element in `index`.
- For each pair `(x, i)` obtained from zipping `nums` and `index`, perform an insert operation: `target.insert(i, x)`. This line is the core of the solution, where `i` is the position in the target array where we want to insert the element, and `x` is the actual element from `nums` we want to insert.
- The `insert()` method takes two arguments: the first argument is the index at which to insert the item, and the second argument is the item to insert. It modifies the list in place, which means no new list is created, the existing `target` list is updated.
- This process is repeated until there are no more elements to read, meaning every element from `nums` has been placed into the `target` list in the order specified by `index`.
- Return the `target` list.

By utilizing the `insert()` method, we can insert elements at specific positions, which provides a seamless way to construct the `target` array. The solution follows the insertion rules exactly as specified and the simplicity of the approach reflects the clarity of the problem's instructions. It is worth noting that while `insert()` operation is efficient for small to medium-sized lists, inserting elements into a list has a time complexity of O(n) per operation because it may require shifting over other elements. However, for the constraints of this problem, the approach is suitable and efficient.

Example Walkthrough

Let's consider the following small example to illustrate the solution approach:

Suppose `nums = [0, 1, 2, 3, 4]` and `index = [0, 1, 2, 2, 1]`.

Following the algorithm:

1. Initialize the `target` list as empty: `target = []`.
2. Now begin iterating over the `nums` and `index` arrays simultaneously.
   - First pair: `(0, 0)` - insert `0` at index `0` of `target` → `target = [0]`.
   - Second pair: `(1, 1)` - insert `1` at index `1` of `target` → `target = [0, 1]`.
   - Third pair: `(2, 2)` - insert `2` at index `2` of `target` → `target = [0, 1, 2]`.
   - Fourth pair: `(3, 2)` - insert `3` at index `2` of `target`. This will push the current element at index `2` (which is `2`) to the right → `target = [0, 1, 3, 2]`.
   - Fifth pair: `(4, 1)` - insert `4` at index `1` of `target`. This will push elements starting from index `1` to the right → `target = [0, 4, 1, 3, 2]`.
3. At this point, we have read all elements from `nums` and `index`, and the `target` list is fully constructed.
4. The final step is to return the `target` list which is `[0, 4, 1, 3, 2]`.

This example clearly demonstrates how elements from the `nums` array are inserted into the `target` array at the positions dictated by the corresponding `index` values. Each insert operation respects the current state of the `target` array, potentially shifting elements to make room for the new ones. The final `target` array reflects the ordered insertions as per the given `nums` and `index` arrays.

Solution Implementation

Python

```python
from typing import List  # Importing List from typing module for type hints

class Solution:
    def createTargetArray(self, nums: List[int], indices: List[int]) -> List[int]:
        # Initialize an empty target array
        target = []

        # Looo over the pairs of elements from nums and their corresponding indices
        for num, idx in zip(nums, indices):
            # Insert the element 'num' at the index 'idx' of the target array
            target.insert(idx, num)

        # Return the final target array
        return target
```

Java

```java
class Solution {
    public int[] createTargetArray(int[] nums, int[] index) {
        // Get the length of the input array.
        int n = nums.length;
        // Initialize an ArrayList to hold the target elements.
        List<Integer> targetList = new ArrayList<>();
        // Iterate through each element of nums and index arrays.
        for (int i = 0; i < n; ++i) {
            // Add the current element from nums into the targetList at the position given by index[i].
            targetList.add(index[i], nums[i]);
        }

        // Initialize the target array.
        int[] targetArray = new int[n];
        // Convert the ArrayList back into an array.
        for (int i = 0; i < n; ++i) {
            targetArray[i] = targetList.get(i);
        }
        // Return the resultant target array.
        return targetArray;
    }
}
```

C++

```cpp
#include <vector>   // Include vector header for std::vector
using namespace std;

class Solution {
public:
    /**
     * Create a target array by insertina elements from the 'nums' array into the
     * 'target' array at positions specified by the 'index' array.
     *
     * @param nums A vector of integers to insert into the target array.
     * @param index A vector of integers indicating the indices at which
     *              to insert the elements from the nums vector.
     * @return The target vector after all insertions.
     */
    vector<int> createTargetArray(vector<int>& nums, vector<int>& index) {
        vector<int> target; // The target vector that we'll return

        // Iterate over all elements in 'nums' and 'index'
        for (int i = 0; i < nums.size(); ++i) {
            // At the position index[i], insert the value nums[i] into the 'target' vector
            target.insert(target.begin() + index[i], nums[i]);
        }

        return target; // Return the completed target vector
    }
};
```

TypeScript

```typescript
// Function to create a target array according to the specified order
function createTargetArray(nums: number[], indices: number[]): number[] {
    // Initialize the target array to hold the result
    const targetArray: number[] = [];

    // Iterate over the nums array to populate the target array
    for (let i = 0; i < nums.length; i++) {
        // Insert the current number at the specified index in the target array
        // The splice method modifies the array in place and can insert elements
        targetArray.splice(indices[i], 0, nums[i]);
    }

    // Return the resulting target array after the loop completes
    return targetArray;
}
```

```python
from typing import List  # Importing List from typing module for type hints

class Solution:
    def createTargetArray(self, nums: List[int], indices: List[int]) -> List[int]:
        # Initialize an empty target array
        target = []

        # Looo over the pairs of elements from nums and their corresponding indices
        for num, idx in zip(nums, indices):
            # Insert the element 'num' at the index 'idx' of the target array
            target.insert(idx, num)

        # Return the final target array
        return target
```

Time and Space Complexity

Time Complexity

The given code has a time complexity of `O(n^2)`. This is because for each element `x` in `nums`, the method `insert()` is called which can take `O(n)` time in the worst case, as it requires shifting all the elements after the inserted element by one position to make space for the new element. Since there are `n` insert operations, and each might take up to `O(n)` time, the overall time complexity is `O(n * n)`, which simplifies to `O(n^2)`.

Space Complexity

The space complexity of the code is `O(n)`. The `target` list that is created, in the worst case, will contain all the elements from the `nums` list, thus the amount of space used grows linearly with the input size `n`. No additional space other than the `target` list (which is the desired output) is used, hence the space complexity is `O(n)`.