760. Find Anagram Mappings

Easy Array Hash Table [Leetcode Link]

Problem Description

In this problem, you have two integer arrays, `nums1` and `nums2`. The array `nums2` is an anagram of `nums1`, meaning `nums2` contains all the elements of `nums1`, possibly in a different order, and both arrays may have repeated numbers. The goal is to create a mapping from `nums1` to `nums2` in such a way that for each index `i` in `nums1`, `mapping[i]` gives you the index `j` in `nums2` where the `i`-th element of `nums1` is located. Since `nums2` is an anagram of `nums1`, there is always at least one index `j` in `nums2` for every element in `nums1`.

Your task is to return any one of these index mapping arrays, as there may be multiple valid mappings due to duplicates.

Intuition

The intuition behind the solution involves two key observations:

1. Since `nums1` and `nums2` are anagrams with possible duplicates, the elements in `nums1` will surely be found in `nums2`.
2. There can be multiple positions for a single element of `nums1` in `nums2` due to duplicates.

The approach uses a dictionary data structure to keep track of the indices of elements in `nums2`. A Python `defaultdict` is used here with a set as the default data type to handle multiple indices for duplicates in an efficient way. We iterate over `nums2` and fill the dictionary such that each unique number in `nums2` is a key, and the corresponding value is a set of indices where this number appears in `nums2`.

After we have this dictionary ready, we iterate over `nums1` and for each element, we `pop` an index from the set in the dictionary. This works well because we can remove any index from the set (since any mapping is valid), and we are guaranteed to have at least one index in the set for each element due to the anagram property. The result is an index mapping list that satisfies the problem requirement.

Solution Approach

Let's dive into the implementation details of the given solution:

1. **Data Structures Used**:
   - `defaultdict` from Collections module: A dictionary subclass that calls a factory function to supply missing values. In this case, the factory function is `set` which holds indices.
   - `set`: A built-in Python data structure used here to store indices of the elements in `nums2`. Sets are chosen because they allow efficient addition and removal of elements, and we don't care about the order of indices.
2. **Algorithm Steps**:
   - Initialize a `defaultdict` with `set` as the default factory function: `mapper = defaultdict(set)`.
   - Enumerate over `nums2` and fill the `mapper`: For each `num` encountered at index `i` in `nums2`, add index `i` to the set corresponding to `num` in the `mapper`. This essentially records the indices where each number from `nums1` can map to in `nums2`.
   - Generate the Index Mapping Array: Iterate over `nums1`, and for each `num`, `pop` a value from the set corresponding to `num` in the `mapper`. This value is an index in `nums2` where `num` occurs. `pop` is used because it removes an element from the set, ensuring that an index is not used twice. The index is appended to the resulting list.
3. **Complexity Analysis**:
   - Time Complexity: `O(N)` where `N` is the number of elements in `nums1` (or `nums2`, since they have the same length). It's `O(N)` because we do a single pass over both `nums1` and `nums2`.
   - Space Complexity: `O(N)` for the `mapper`, as it stores indices for elements in `nums2`. In the worst case, where all elements in `nums2` are distinct, the `mapper` will store one index for each element.

The pattern used in this solution can be identified as **Hash Mapping**. It utilizes the constant time access feature of the dictionary (`hash map`) for storing and retrieving element indices.

By following these steps, the function `anagramMappings(nums1, nums2)` will return the index mapping list that will form a correct anagram mapping from `nums1` to `nums2`.

Example Walkthrough

Consider the following small example to illustrate the solution approach:

Let `nums1` be `[12, 28, 46, 32, 50]` and `nums2` be `[50, 12, 32, 46, 28]`.

Firstly, we'll initialize our `mapper` as a `defaultdict` of sets. It will look like this after we process `nums2`:

- `50` appears at index `0` in `nums2`
- `12` appears at index `1` in `nums2`
- `32` appears at index `2` in `nums2`
- `46` appears at index `3` in `nums2`
- `28` appears at index `4` in `nums2`

So, our `mapper` would be filled as follows:

```
mapper = {
    50: {0},
    12: {1},
    32: {2},
    46: {3},
    28: {4}
}
```

Next, we iterate over `nums1`:

- For `12`, we `pop` an index from `mapper[12]`, which gives us `1`. Now `mapper[12]` is empty.
- For `28`, we `pop` an index from `mapper[28]`, which gives us `4` and `mapper[28]` becomes empty.
- For `46`, we `pop` an index from `mapper[46]`, which is `3` and `mapper[46]` is now empty.
- For `32`, we `pop` from `mapper[32]` giving us `2` and `mapper[32]` is emptied.
- Finally for `50`, `pop` from `mapper[50]` gives us `0` and empties `mapper[50]`.

The resulting index mapping array would be `[1, 4, 3, 2, 0]` which is a valid anagram mapping from `nums1` to `nums2` since if we position the elements of `nums1` using these indices in `nums2`, we get the same list as `nums1`.

To summarize, the resulting mapping array is built by popping an index from the set of indices associated with each corresponding element in `nums1` from our `mapper` dictionary. This process ensures that each element from `nums1` is uniquely matched to an element in `nums2`, following the stipulation of the anagram relationship between the two arrays.

Python Solution

```python
from collections import defaultdict
from typing import List

class Solution:
    def anagramMappings(self, nums1: List[int], nums2: List[int]) -> List[int]:
        # Dictionary to store the mapping of each number in nums2 to its indices
        index_mapper = defaultdict(set)

        # Populate the index_mapper with elements of nums2 and their respective indices
        for index, number in enumerate(nums2):
            index_mapper[number].add(index)

        # Create the result list by mapping each number in nums1 to an index in nums2
        # The pop method is used to ensure the same index is not reused
        result = [index_mapper[number].pop() for number in nums1]

        return result

```

Java Solution

```java
class Solution {
    // Function to find an anagram mapping from nums1 to nums2.
    public int[] anagramMappings(int[] nums1, int[] nums2) {
        // Create a map to hold the indices of each number in nums2.
        Map<Integer, Set<Integer>> numIndicesMap = new HashMap<>();
        // Fill the map: number -> its indices in nums2.
        for (int i = 0; i < nums2.length; ++i) {
            // If the key is not already in the map, put it with a new empty set.
            // Then add the current index to the set of indices.
            numIndicesMap.computeIfAbsent(nums2[i], k -> new HashSet<>()).add(i);
        }

        // Initialize the result array that will hold the mappings.
        int[] result = new int[nums1.length];
        // Find the anagram mappings from nums1 to nums2.
        for (int i = 0; i < nums1.length; ++i) {
            // Get the set of indices for the current number from nums1 in nums2.
            Set<Integer> indicesSet = numIndicesMap.get(nums1[i]);
            // Get and use the next available index (iterator's next).
            int index = indicesSet.iterator().next();
            // Save the index in the result array.
            result[i] = index;
            // Remove the used index to ensure one-to-one mapping.
            indicesSet.remove(index);
        }
        // Return the completed result array.
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <unordered_set>

class Solution {
public:
    // Function to find an anagram mapping from nums1 to nums2.
    std::vector<int> anagramMappings(std::vector<int>& nums1, std::vector<int>& nums2) {
        // Create a map to hold the indices of each number in nums2.
        std::unordered_map<int, std::unordered_set<int>> numIndicesMap;
        // Fill the map: number -> its indices in nums2.
        for (int i = 0; i < nums2.size(); ++i) {
            // Add the current index to the set of indices for the number in nums2.
            numIndicesMap[nums2[i]].insert(i);
        }

        // Initialize the result vector that will hold the mappings.
        std::vector<int> result(nums1.size());
        // Find the anagram mappings from nums1 to nums2.
        for (int i = 0; i < nums1.size(); ++i) {
            // Get the set of indices for the current number from nums1 in nums2.
            std::unordered_set<int>& indicesSet = numIndicesMap[nums1[i]];
            // Get and use the next available index from the set.
            auto it = indicesSet.begin();
            int index = *it;
            // Save the index in the result vector.
            result[i] = index;
            // Remove the used index to ensure one-to-one mapping.
            indicesSet.erase(it);
        }
        // Return the completed result vector.
        return result;
    }
};

```

Typescript Solution

```typescript
// Define a function to find an anagram mapping from nums1 to nums2.
function anagramMappings(nums1: number[], nums2: number[]): number[] {
    // Create a map to hold indices for each number in nums2.
    const numIndicesMap: Map<number, Set<number>> = new Map();

    // Fill the map with number -> its indices in nums2.
    nums2.forEach((num, index) => {
        const indicesSet = numIndicesMap.get(num) || new Set<number>();
        indicesSet.add(index);
        numIndicesMap.set(num, indicesSet);
    });

    // Initialize the result array to hold the mappings.
    const result: number[] = new Array(nums1.length);

    // Find the anagram mappings from nums1 to nums2.
    nums1.forEach((num, i) => {
        // Get the set of indices for the current number from nums1 in nums2.
        const indicesSet = numIndicesMap.get(num);
        if (!indicesSet) {
            throw new Error('No index found for number ' + num);
        }

        // Get and use the next available index.
        const index = indicesSet.values().next().value;
        // Save the index in the result array.
        result[i] = index;
        // Remove the used index to ensure one-to-one mapping.
        indicesSet.delete(index);
    });

    // Return the completed result array.
    return result;
}

// Example usage of the function
const nums1 = [12, 28, 46, 32, 50];
const nums2 = [50, 12, 32, 46, 28];
const mapping = anagramMappings(nums1, nums2); // Should produce the anagram mapping of nums1 to nums2.

```

Time and Space Complexity

The given Python function `anagramMappings` finds an index mapping from `nums1` to `nums2`, indicating where each number in `nums1` appears in `nums2`. Here is the complexity analysis:

Time Complexity:

The function consists of two main parts: building the `mapper` dictionary and constructing the result list.

- Constructing `mapper`: We iterate through `nums2` once to build the `mapper` dictionary. For each element in `nums2` of size `n`, inserting into a set in the dictionary takes amortized O(1) time. Thus, this part has a time complexity of O(n).
- Constructing the result list: For each element in `nums1`, which also can be size `n` in the worst case, we pop an element from the corresponding set in `mapper`. Each pop operation takes O(1) time because it removes an arbitrary element from the set. Thus, this part also has a time complexity of O(n).

The total time complexity of the function is O(n) + O(n) = O(n), where `n` is the length of `nums2`.

Space Complexity:

- The space used by `mapper`: In the worst case, if all elements in `nums2` are unique, the dictionary will contain `n` sets, each with a single index. So, the space complexity for `mapper` will be O(n).
- The space for the result list: A new list of the same size as `nums1` is created for the output. Since `nums1` and `nums2` can be of size `n`, this list will also have a space complexity of O(n).

The total space complexity of the function is O(n) [for `mapper`] + O(n) [for the result list] = O(2n), which simplifies to O(n), where `n` is the length of `nums2`.

Note: The space complexity only considers additional space used by the program, not the input and output space. If we consider the inputs and outputs, the space complexity would technically be O(n + n + n), which still simplifies to O(n).