1865. Finding Pairs With a Certain Sum

Medium Design Array Hash Table

Leetcode Link

Problem Description

You are provided with two integer arrays, `nums1` and `nums2`, and the goal is to create a data structure that supports two types of queries. Firstly, you should be able to add a positive integer to an element at a given index in `nums2`. Secondly, you need to count the number of pairs `(i, j)` where the sum of `nums1[i]` and `nums2[j]` is equal to a specified value (`tot`). The pairs should only be considered if `i` and `j` are valid indexes within `nums1` and `nums2`, respectively.

To accomplish the task, you should implement a class `FindSumPairs` with the following methods:

- `FindSumPairs(int[] nums1, int[] nums2)`: a constructor that initializes the object with the two integer arrays, `nums1` and `nums2`.
- `add(int index, int val)`: a method that adds the integer `val` to the element at `index` in `nums2`.
- `count(int tot)`: a method that returns the number of pairs `(i, j)` where `nums1[i] + nums2[j]` equals the integer `tot`.

Intuition

The intuitive approach to solving this problem involves efficient data retrieval and update methods. If we were to perform a brute force approach for the count operation, we would iterate through all possible `(i, j)` pairs to find their sum and compare with `tot`, which would result in a time-consuming process, especially for large arrays.

A better way is to use a hash table to keep track of the frequency of each number in `nums2`. This allows us to quickly calculate how many times a certain number that can be added to elements of `nums1` appears in `nums2` to reach the required sum (`tot`).

Here's the intuition for the methods:

- The constructor initializes the object and also creates a counter (`Counter` from `collections` in Python) that maps each number in `nums2` to its frequency.
- The `add` method updates the specific element in `nums2` and adjusts the counter correspondingly. If a number's frequency changes (because it's being increased by `val`), we decrease the count of the old number and increase the count for the new, updated number.
- The `count` method calculates the required pairs by traversing through `nums1` and checking if the complement to reach `tot` (calculated as `tot - nums1[i]`) exists in the hash table (counter for `nums2`). The sum of all the frequencies of these complements gives us the total number of valid pairs that meet the condition.

Solution Approach

The implementation of the `FindSumPairs` class makes use of hash tables to store elements and their frequencies from `nums2`. This is essentially a mapping from each unique integer in `nums2` to the number of times it appears. The Python `Counter` class from the `collections` module is used here for this purpose as it automatically counts the frequency of items in a list.

Here's the breakdown of the implementation:

The `__init__` function of the `FindSumPairs` class initializes two properties, `nums1` and `nums2`, with the corresponding input arrays. Additionally, a `Counter` object named `cnt` is created to store the frequency count of elements in `nums2`.

```
1 def __init__(self, nums1: List[int], nums2: List[int]):
2     self.nums1 = nums1
3     self.nums2 = nums2
4     self.cnt = Counter(nums2)
```

The `add` function takes in an `index` and a value `val` to add to `nums2` at the specified index. Before updating `nums2[index]` with the new value, it decreases the count of the old value in `cnt`. Then, it increases the count of `nums2[index]` plus `val`. After updating the `cnt`, `nums2` is updated with the new value.

```
5 def add(self, index: int, val: int) -> None:
6     self.cnt[self.nums2[index]] -= 1
7     self.cnt[val] += 1
8     if self.cnt[old] == 0:
9         del self.cnt[old]  # Remove the entry from the counter if the frequency is zero
10    self.nums2[index] = val + 1
11    self.nums2[index] = val
```

The `count` function takes in a value `tot` and returns the number of pairs `(i, j)` where `nums1[i] + nums2[j] == tot`. To do this, it sums up the counts of `tot - nums1[i]` in `cnt` for each element `i` in `nums1`. In other words, for every number in `nums1`, it calculates the complement (the number that needs to be added to it in order to reach `tot`) and uses this to get the frequency of such complements from the `Counter`.

```
12 def count(self, tot: int) -> int:
13     return sum(self.cnt[tot - v] for v in self.nums1)
```

By utilizing the `Counter`, the class is able to execute the `count` function in O(n) time relative to the size of `nums1`, as it only has to iterate over `nums1` and can retrieve the complement counts in constant time from the hash table. This is significantly more efficient than attempting to calculate the pairs through nested loops, which would be O(n * m) where n and m are the sizes of `nums1` and `nums2`.

Example Walkthrough

Let's step through the problem using a small example to illustrate the solution approach.

Suppose `nums1` is `[1, 2, 3]` and `nums2` is `[4, 4, 5, 2]`, and we are interested in finding pairs whose sum equals `tot = 5`.

Upon class initialization:

1. `nums1` remains `[1, 2, 3]`.
2. `nums2` remains `[1, 4, 5, 2]`.
3. A `Counter` is created from `nums2`, resulting in `{1: 1, 4: 1, 5: 1, 2: 1}`.

Now, let's say we perform an `add` operation – `add(2, 2)` – which adds `2` to the element at index `2` in `nums2`.

1. The old value at `nums2[2]` is `5`, so the counter for `5` is decreased from `1` to `0` and consequently removed since its count is now zero.
2. The value at `nums2[2]` is updated to `7` (`5 + 2`), so the counter for `7` is increased from `0` to `1`.
3. Now `nums2` becomes `[1, 4, 7, 2]`, and the `Counter` reflects `{1: 1, 4: 1, 7: 1, 2: 1}`.

Next, we perform a `count` operation – `count(5)` to find pairs summing up to `5`.

1. We iterate through `nums1`, looking for values that, when added to values from `nums2`, yield `5`.
2. For `nums1[0] = 1`, we find `tot - nums1[0] = 5 - 1 = 4`; `cnt[4]` equals `1` indicating a pair `(1, 4)`.
3. For `nums1[1] = 2`, `tot - nums1[1] = 5 - 2 = 3`; `cnt[3]` equals `0` indicating no valid pair exists with the second element `2`.
4. For `nums1[2] = 3`, `tot - nums1[2] = 5 - 3 = 2`; `cnt[2]` equals `1` indicating a pair `(3, 2)`.
5. Sum up the counts of valid pairs for all elements in `nums1`, giving us `cnt[4] + cnt[3] + cnt[2] = 1 + 0 + 1 = 2`.

Therefore, the `count(5)` operation tells us there are `2` valid pairs `(1, 4)` and `(3, 2)` whose sum equals `5`.

This example demonstrates the efficiency of the approach using a `Counter` to avoid iterating over `nums2` each time we call `count`. Instead, we make use of the precomputed frequencies to quickly tally the number of pairs that sum up to `tot`.

Python Solution

```
1  from collections import Counter
2  from typing import List
3
4  class FindSumPairs:
5      def __init__(self, nums1: List[int], nums2: List[int]):
6          # Initialize lists and create a counter for nums2 to keep track of occurrences
7          self.nums1 = nums1
8          self.nums2 = nums2
9          self.count = Counter(nums2)  # using 'counts' in place of 'cnt' for readability
10
11     def add(self, index: int, val: int) -> None:
12         # Function to update the value at a given index in nums2 and adjust the counter
13         old_val = self.nums2[index]
14         # Decrease the count for the old value
15         self.count[old_val] -= 1
16         # If the count reaches zero, remove it from the counter to keep it clean
17         if self.count[old_val] == 0:
18             del self.count[old_val]
19         # Update the value in nums2
20         self.nums2[index] += val
21         # Increase the count for the new value
22         self.count[self.nums2[index]] += 1
23
24     def count(self, total: int) -> int:
25         # Function to find the number of pairs from nums1 and nums2 that sum up to 'total'
26         ans = 0
27         # Iterate over values in nums1 and check if (total - value) exists in nums2's counter
28         for value in self.nums1:
29             ans += self.count[total - value]
30         return ans
31
32 # The class FindSumPairs can be used as follows:
33 # obj = FindSumPairs(nums1, nums2)
34 # obj.add(index, val)
35 # param_count = obj.count(total)
36
```

Java Solution

```
1  import java.util.HashMap;
2  import java.util.Map;
3
4  // Class to find count of pairs from two arrays that sum up to a given value
5  class FindSumPairs {
6      // Arrays to store the two input integer arrays
7      private int[] nums1;
8      private int[] nums2;
9      // Map to keep the frequency count of elements in the second array
10     private Map<Integer, Integer> frequencyMap = new HashMap<>();
11
12     // Constructor initializes the class with two integer arrays
13     public FindSumPairs(int[] nums1, int[] nums2) {
14         this.nums1 = nums1;
15         this.nums2 = nums2;
16
17         // Populating the frequency map with the count of each number in nums2
18         for (int value : nums2) {
19             frequencyMap.put(value, frequencyMap.getOrDefault(value, 0) + 1);
20         }
21     }
22
23     // Method that increments an element of nums2 at a given index by a given value
24     public void add(int index, int value) {
25         // Get the original value at the given index in nums2
26         int originalValue = nums2[index];
27         // Decrement the frequency of this original value in the frequency map
28         frequencyMap.put(originalValue, frequencyMap.get(originalValue) - 1);
29         // Update the value at the index in nums2 by adding the given value
30         nums2[index] += value;
31         // Increment the frequency of the new value in the frequency map
32         frequencyMap.put(nums2[index], frequencyMap.getOrDefault(nums2[index], 0) + 1);
33     }
34
35     // Method that counts the pairs across nums1 and nums2 that sum up to a given total
36     public int count(int total) {
37         int count = 0;
38         // Iterate through each value in nums1
39         for (int value : nums1) {
40             // For the current value in nums1, check if there's a complement in nums2 that sums up to total
41             count += frequencyMap.getOrDefault(total - value, 0);
42         }
43         // Return the count of such pairs
44         return count;
45     }
46 }
47
48 /*
49  * The usage of the FindSumPairs class:
50  *
51  * FindSumPairs obj = new FindSumPairs(nums1, nums2);
52  * obj.add(index, value);
53  * int result = obj.count(total);
54  */
55
```

C++ Solution

```
1  #include <vector>
2  #include <unordered_map>
3  using namespace std;
4
5  class FindSumPairs {
6  public:
7      // Constructor initializes the object with two integer vectors
8      FindSumPairs(vector<int>& nums1, vector<int>& nums2) {
9          this->nums1 = nums1; // Assign the first vector to the class member
10         this->nums2 = nums2; // Assign the second vector to the class member
11
12         // Populate the count map with the frequency of each number in nums2
13         for (int value : nums2) {
14             ++countMap[value];
15         }
16     }
17
18     // Function to add a value to an element in nums2 and update the count map
19     void add(int index, int value) {
20         int oldValue = nums2[index]; // Retrieve the old value from nums2 at the given index
21         --countMap[oldValue]; // Decrease the count of the old value in the map
22         ++countMap[oldValue + value]; // Increase the count of the new value in the map
23         nums2[index] += value; // Update the value in nums2 by adding the given value
24     }
25
26     // Function to count the pairs with the given sum 'total'
27     int count(int total) {
28         int pairCount = 0; // Initialize the count of valid pairs
29
30         // Iterate through elements in nums1 and calculate the complement
31         for (int value : nums1) {
32             pairCount += countMap[total - value]; // Add the count of the complement from the map to the pairs count
33         }
34
35         return pairCount; // Return the total count of valid pairs
36     }
37
38 private:
39     vector<int> nums1; // First input vector
40     vector<int> nums2; // Second input vector
41     unordered_map<int, int> countMap; // Map to store the frequency of each number in nums2
42 };
43
44 /**
45  * The FindSumPairs class is instantiated and used as follows:
46  * FindSumPairs* obj = new FindSumPairs(nums1, nums2); // Create a new FindSumPairs object
47  * obj->add(index, value); // Add a value to the element at the given index in nums2
48  * int param_count = obj->count(total); // Count the pairs that sum to the total variable
49  * delete obj; // Clean up the object when done (important for memory management)
50  */
51
```

Typescript Solution

```
1  // The original C++ includes and namespace declaration are not needed in TypeScript.
2
3  // Global variables for the two number arrays and the count map
4  let nums1: number[];
5  let nums2: number[];
6  let countMap: { [key: number]: number } = {};
7
8  // Function to initialize the object with two integer arrays
9  function initializeFindSumPairs(nums1Input: number[], nums2Input: number[]): void {
10     // Assign the first input array to the global variable
11     nums1 = nums1Input; // Assign the second array to the global variable
12     nums2 = nums2Input; // Assign the second array to the global variable
13
14     // Populate the count map with the frequency of each number in nums2
15     countMap = {}; // Reset count map
16     nums2.forEach((value) => {
17         if (countMap[value] === undefined) {
18             countMap[value] = 0;
19         }
20         countMap[value]++;
21     });
22 }
23
24 // Function to add a value to an element in nums2 and update the count map
25 function add(index: number, value: number): void {
26     const oldValue = nums2[index]; // Retrieve the old value from nums2 at the given index
27     countMap[oldValue]--; // Decrease the count of the old value in the map
28     const newValue = oldValue + value;
29     if (countMap[newValue] === undefined) {
30         countMap[newValue] = 0;
31     }
32     countMap[newValue]++; // Increase the count of the new value in the map
33     nums2[index] = newValue; // Update the value in nums2 by adding the given value
34 }
35
36 // Function to count the pairs with the given sum 'total'
37 function count(total: number): number {
38     let pairCount = 0; // Initialize the count of valid pairs
39
40     // Iterate through elements in nums1 and calculate the complement
41     nums1.forEach((value) => {
42         const complement = total - value;
43         pairCount += countMap[complement] ?? 0; // Add the count of the complement from the map to the pairs count
44     });
45
46     return pairCount; // Return the total count of valid pairs
47 }
48
49 // Example usage:
50 /*
51 initializeFindSumPairs([1, 2, 3], [2, 3, 4, 5]);
52 add(1, 2); // Now nums2 is [2, 5, 4, 5]
53 const result = count(8); // There are 2 pairs that add up to 8: (1,7) and (4,4)
54 console.log(result); // Outputs: 2
55 */
```

Time and Space Complexity

Time Complexity

- `__init__(self, nums1: List[int], nums2: List[int])`: The constructor initializes two lists `nums1` and `nums2`. It also counts the elements of `nums2` using `Counter` which takes `O(n)` time where `n` is the length of `nums2`.
- `add(self, index: int, val: int) -> None`: This method updates an element in `nums2` and modifies the count of the old and the new value in `cnt`. The update operation and the changes in the counter have a constant time complexity `O(1)` since dictionary (counter) operations in Python have an average-case complexity of `O(1)`.
- `count(self, tot: int) -> int`: The count method iterates over `nums1` and for each element `v` in `nums1`, it accesses the counter `cnt` to find the count of `(tot - v)`. If `nums1` has `n` elements, the time complexity is `O(n)` since each lookup in the counter is `O(1)` on average.

Space Complexity

- `__init__(self, nums1: List[int], nums2: List[int])`: Apart from the input lists `nums1` and `nums2`, a counter `cnt` is created to store the frequency of each element in `nums2`. The space complexity is `O(n)` where `n` is the number of unique elements in `nums2`.
- `add(self, index: int, val: int) -> None`: This method uses no extra space and thus has a space complexity of `O(1)`.
- `count(self, tot: int) -> int`: This method does not use extra space apart from a few variables for its operation, hence the space complexity is `O(1)`.

Overall, the space complexity of the `FindSumPairs` class is determined by the space required for storing the input lists and the counter, which is `O(n)` where `n` is the size of `nums2` and the number of unique elements it contains.