2395. Find Subarrays With Equal Sum

Easy Array Hash Table

Leetcode Link

Problem Description

The problem is asking us to check if an integer array named `nums`, which is 0-indexed, contains at least two subarrays of length 2 with the same sum. Importantly, these two subarrays must start at different indices within the `nums` array. We define a subarray as a continuous part of the original array that contains one or more consecutive elements.

For example, given `nums = [4,2,4,5,6]`, we have subarrays with length 2 such as `[4,2]`, `[2,4]`, `[4,5]`, and `[5,6]`. The question is whether any two of these pairs have an equal sum.

We are to return `true` if such subarrays exist or `false` otherwise.

Intuition

The intuition behind the solution is to use a set to keep track of sums of all possible subarrays of length 2. Since we're looking for two different subarrays that have the same sum, we can iterate through the `nums` array constructing subarrays of length 2 on the fly, sum them up, and then check if their sum has already been encountered before (i.e., it's present in the set).

The steps to arrive at the solution are straightforward:

1. Initialize an empty set to store the unique sums of subarrays of length 2.
2. Iterate through the `nums` array while considering every adjacent pair of elements as a potential subarray of length 2.
3. For each pair, calculate the sum and check if this sum has been seen before (i.e., it is in the set).
    - If the sum is already in the set, this means we have found another subarray with the same sum, hence we can return `true`.
    - If it's not, we add the sum to the set and continue with the next pair.
4. If we finish iterating over the array without finding such a pair, return `false`.

By utilizing the set for the lookup operation, which is O(1) on average, we can ensure that our solution is efficient, with an overall time complexity of O(n) where n is the number of elements in `nums`.

Solution Approach

The implementation of the solution approach involves a loop and a `set`. The `set` data structure is chosen because it provides O(1) lookup time for checking if an element exists, making the overall solution more efficient. This efficiency is central to the algorithm we use to solve the problem.

Here's how the algorithm works step by step:

1. Initialize an empty `set` named `vis` which will store the sums of subarrays of length 2.
2. Iterate over the array `nums` using a for loop. Here, we are using Python's `pairwise` utility (which, assuming it's similar to or a placeholder for `itertools.pairwise`, generates consecutive pairs of elements). For each pair `(a, b)` in `nums`, we perform the following operations inside the loop:
    - Calculate the sum of the current pair with `x := a + b`. This is an example of Python's walrus operator (`:=`), which assigns the value of `a + b` to `x` and then allows `x` to be used immediately within the condition.
    - Check if `x` is already in `vis`. If it is, that means we have found a previous pair with the same sum, and we can immediately return `true`.
    - If `x` is not found in `vis`, add `x` to `vis` to keep track of this sum and then continue to the next pair.
3. If the loop completes without returning `true`, no two subarrays with the same sum have been found. Therefore, we return `false`.

The pattern used here is known as the 'Sliding Window' pattern, which is commonly used for problems dealing with contiguous subarrays or subsequences of a given size. The sliding window in this solution is of fixed size 2, which slides over the array to consider all subarrays of length 2.

Code

Here is the provided code for the solution:

```python
class Solution:
    def findSubarrays(self, nums: List[int]) -> bool:
        vis = set()
        for a, b in pairwise(nums):
            if (x := a + b) in vis:
                return True
            vis.add(x)
        return False
```

The code succinctly executes the algorithm described and solves the problem effectively with an O(n) time complexity.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using the array `nums = [1, 2, 3, 4, 5]`.

1. Initialize an empty set `vis` to store the sums of subarrays of length 2.
2. Start iterating over the array to consider all pairs of consecutive numbers:
    - First pair: `[1, 2]`
        - Calculate the sum: `x = 1 + 2` which is `3`.
        - Check if `3` is in `vis`. It is not, so add `3` to `vis`.
        - Now, `vis = {3}`.
    - Second pair: `[2, 3]`
        - Calculate the sum: `x = 2 + 3` which is `5`.
        - Check if `5` is in `vis`. It is not, so add `5` to `vis`.
        - Now, `vis = {3, 5}`.
    - Third pair: `[3, 4]`
        - Calculate the sum: `x = 3 + 4` which is `7`.
        - Check if `7` is in `vis`. It is not, so add `7` to `vis`.
        - Now, `vis = {3, 5, 7}`.
    - Fourth and last pair: `[4, 5]`
        - Calculate the sum: `x = 4 + 5` which is `9`.
        - Check if `9` is in `vis`. It is not, so add `9` to `vis`.
        - Now, `vis = {3, 5, 7, 9}`.
3. We have finished iterating over the array and have not found any sums that occur more than once in `vis`. Therefore, we return `false`, meaning no two subarrays of length 2 with the same sum exist in the array `nums = [1, 2, 3, 4, 5]`.

This small example illustrates how the algorithm effectively checks each subarray of length 2 for a sum that might have appeared before, and efficiently keeps track of the sums using a set.

Python Solution

```python
from itertools import pairwise  # Python 3 does not have 'pairwise' in itertools by default. Make sure it's available, or define an e

class Solution:
    def findSubarrays(self, nums):
        # Initialize a set to store the sum of pairs
        seen_sums = set()

        # Iterate over pairwise combinations using zip to get adjacent elements.
        for a, b in zip(nums, nums[1:]):
            current_sum = a + b
            # Check if the sum of the current pair has been seen before
            if current_sum in seen_sums:
                # If we have seen this sum, a subarray with equal sum exists
                return True
            # Add the sum to the set of seen sums
            seen_sums.add(current_sum)

        # If no two subarrays have the same sum, return False
        return False
```

Java Solution

```java
class Solution {
    public boolean findSubarrays(int[] nums) {
        // Create a HashSet to store the sums of consecutive elements
        Set<Integer> seenSums = new HashSet<>();

        // Iterate through the array, starting from the second element
        for (int i = 1; i < nums.length; ++i) {
            // Calculate the sum of the current and previous elements
            int sum = nums[i - 1] + nums[i];

            // Attempt to add the sum to the set
            boolean isAdded = seenSums.add(sum);

            // If the sum could not be added because it's already present
            // in the set, then we found a repeated sum and return true
            if (!isAdded) {
                return true;
            }
        }

        // If the loop completes without returning, no repeating sum was found
        // so we return false
        return false;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
using namespace std; // Directives for convenience, commonly placed at the top

class Solution {
public:
    // The function check if there are any subarrays with length 2 that sum to the same value.
    bool findSubarrays(vector<int>& nums) {
        unordered_set<int> seenSums; // This set will store the sums of adjacent elements

        // Iterate through the array, stopping one element before the last to avoid out-of-bounds access.
        for (int i = 1; i < nums.size(); ++i) {
            int currentSum = nums[i - 1] + nums[i]; // Calculate sum of current and previous element

            // Check if the current sum has already been seen.
            if (seenSums.count(currentSum)) {
                return true; // If so, we found two subarrays with the same sum.
            }

            // If it hasn't been seen, add the current sum to the set of seen sums.
            seenSums.insert(currentSum);
        }

        // If we've iterate through all elements without finding duplicate sums, return false.
        return false;
    }
};
```

Typescript Solution

```typescript
function findSubarrays(nums: number[]): boolean {
    // Initialize a set to keep track of the sums of adjacent pairs previously seen
    const seenSums: Set<number> = new Set<number>();

    // Loop through each number in the array except the last one
    for (let i = 1; i < nums.length; ++i) {
        // Calculate the sum of the current number and the one before it
        const currentSum = nums[i - 1] + nums[i];

        // Check if the sum has been seen before in the set
        if (seenSums.has(currentSum)) {
            // If yes, we have found a duplicate sum of subarrays
            return true;
        }

        // Add the current sum to the set
        seenSums.add(currentSum);
    }

    // If no duplicate sums of subarrays have been found, return false
    return false;
}
```

Time and Space Complexity

The provided Python function `findSubarrays` iterates through the `nums` list to find if any pair of consecutive elements has a sum that has been seen before. The function utilizes the `pairwise` utility to iterate over the list.

Time Complexity:

- The `pairwise` utility returns an iterator that generates pairs of consecutive elements. The loop thus runs exactly $n - 1$ times, where n is the length of `nums`.
- Checking if a sum is in `vis` set has an average time complexity of O(1) due to the nature of set operations in Python.
- Adding an element to a set also has an average time complexity of O(1).

Hence, the time complexity of the function can be considered as O(n) because it involves just one pass through the `nums` list, with constant-time operations within the loop.

Space Complexity:

- The `vis` set stores sums of consecutive elements. In the worst case, all pairs will have different sums, resulting in $n - 1$ entries in the set.
- This worst-case space complexity of storing the sums is O(n) where n is the number of elements in the list.

In conclusion, the function `findSubarrays` has a time complexity of $O(n)$ and a space complexity of $O(n)$.