548. Split Array with Equal Sum

Hard Array Prefix Sum

Leetcode Link

Problem Description

In this problem, we are given an array of integers, `nums`, and we need to determine if there exists a triplet of indices `(i, j, k)` in the array that meet certain conditions. Specifically, the conditions for the indices are:

- Indices `i`, `j`, and `k` must maintain the sequence `0 < i < j < k < n-1`, where `n` is the length of the array.
- The sum of elements in four subarrays is equal. The subarrays are:
  - `Subarray 1`: Elements from index `0` to `i-1` (both inclusive).
  - `Subarray 2`: Elements from index `i+1` to `j-1` (both inclusive).
  - `Subarray 3`: Elements from index `j+1` to `k-1` (both inclusive).
  - `Subarray 4`: Elements from index `k+1` to the end of the array (index `n-1`).

A subarray `(l, r)` represents the sequence of elements in the array `nums` from the `l`th element to the `r`th element, inclusive.

The aim is to return `true` if at least one such triplet `(i, j, k)` exists, and `false` otherwise.

Intuition

The intuition behind the solution is to find a way to efficiently determine if the sum of elements in the given subarrays are equal without recalculating them each time for different values of `i`, `j`, and `k`. To do this, we can make use of prefix sums and a set.

Here is how we could approach the problem:

1. **Prefix Sum Computation:**
   - First, we can calculate a prefix sum array `s` for the given `nums` array. The prefix sum array contains the sum of all numbers up to and including the `i`th index of `nums`. This helps us calculate the sum of any subarray in constant time.
2. **Determining Equal Subarray Sums:**
   - To find the correct index `j`, we need to ensure that we have enough space on either side for indices `i` and `k` to exist. Therefore, we scan for `j` starting from index 3 to `n - 3`. If `j` is too close to the start or end, there can't be four subarrays with equal sums.
   - For each `j`, we establish two loops:
     - An inner loop over `i`, which ranges from 1 to `j - 1`, looking for subarrays that could be equal in sum to the other subarrays.
     - We check if the sum of elements from `0` to `i-1` is equal to the sum from `i+1` to `j-1`.
     - When we find such an instance, we add the sum to a set `seen` as a potential candidate for the subarray sum.
     - An outer loop over `k`, which ranges from `j + 2` to `n - 1`, looking for subarrays that could match the ones identified by `i`.
     - We check if the sum of elements from `j+1` to `k-1` is equal to the sum from `k+1` to `n-1`.
     - If it is and the sum is already in `seen`, there exists at least one combination of i, j, k that satisfies all conditions, thus, we return true.
3. **Checking for Match:**
   - If we exit both loops without finding such a triplet, then we conclude that no such triplet exists, and we return false.

By implementing this approach, we avoid recalculating the sum for each subarray from scratch, which would otherwise result in a much less efficient solution.

Solution Approach

To implement the solution described in the intuition, we make use of a couple of important programming concepts: prefix sums and hash sets. This enables us to have an efficient algorithm that can solve the problem without repeatedly computing the sums of the subarrays for each possible combination of `(i, j, k)` triplet.

Here's a step-by-step walk-through of the implementation based on the provided solution code:

1. **Initialization and Prefix Sum Computation:**
   - Initialize an array `s` with length `n + 1` where `n` is the length of the input array `nums`. The `s` array is going to store the prefix sums of `nums`.
   - Fill the `s` array with prefix sums, where `s[i + 1] = s[i] + nums[i]`. This means `s[i]` holds the total sum from the start of `nums` up to (and including) index `i-1`.
2. **Finding the Index `j`:**
   - Iterate over possible values of `j` starting from index 3 to `n - 3`. The choice of starting from 3 ensures that there is space for at least two elements before `j` and ending at `n - 3` ensures there's space for at least two elements after `j`.
3. **Exploring Potential Values for `i`:**
   - For each `j`, initialize an empty set `seen`. This set is used to store sums of subarrays that could potentially match with other subarrays.
   - Loop through `i` which starts from 1 and goes up to `j - 1`, perform the following:
     - Check if the sum of the subarray `(0, i - 1)` which is `s[i]`, is equal to the sum of the subarray `(i + 1, j - 1)` which is `s[j] - s[i + 1]`.
     - If they are equal, add the sum to the set `seen`.
4. **Exploring Potential Values for `k`:**
   - Next, loop through `k` which starts from `j + 2` and ends at `n - 1`, perform the following:
     - Check if the sum of the subarray `(j + 1, k - 1)` which is `s[k] - s[j + 1]`, is equal to the sum of the subarray `(k + 1, n - 1)` which is `s[n] - s[k + 1]`.
     - If they are equal and the sum (the sum of the subarray `(j + 1, k - 1)`) exists in the `seen` set, we have found a triplet `(i, j, k)` that satisfies all the conditions.
     - At this point, return `true` since the required triplet exists.
5. **Returning the Result:**
   - If we finish both loops and haven't returned `true`, we determine that no such triplet exists, and therefore, we return `false`.

This approach is efficient because we utilized the pre-computed prefix sum array to quickly access the sum of any subarray in constant time. Additionally, by using the set to keep track of seen sums, we avoid redundant comparisons for each potential `k`, making it a much faster algorithm.

Example Walkthrough

Consider the array `nums` with the following elements:

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

Let's walk through the solution approach with this example:

1. **Initialization and Prefix Sum Computation:**
   - First, we initialize an array `s` for storing prefix sums with `n + 1` elements, where `n` is the length of `nums`. In this case, `n` is 7, so `s` will have 8 elements.
   - Then we compute the prefix sums. So the `s` array after prefix sum computation will be `[0, 1, 3, 4, 6, 7, 9, 10]`. For instance, `s[4] = 6` representing the sum of `nums` from index `0` to `2`.
2. **Finding the Index `j`:**
   - We iterate over possible values of `j` from 3 to 4 (as `n - 3` is 4 for this example).
3. **Exploring Potential Values for `i` for `j = 3`:**
   - We initialize an empty set `seen` for `j = 3`.
   - Loop through `i` which starts from 1 and goes up to `2` (`j - 1`):
     - For `i = 1`, we check the sums: `s[1] = 1` and `s[3] - s[2] = 4 - 3 = 1`. They are equal, so we add `1` to `seen`.
4. **Exploring Potential Values for `k` for `j = 3`:**
   - We loop through `k` which can only be `5` since it starts from `j + 2` and ends at `n - 1` for this `j`.
     - For `k = 5`, we check the sums: `s[5] - s[4] = 7 - 6 = 1` and `s[7] - s[6] = 10 - 9 = 1`. They are equal, and `1` is in the `seen` set.
     - Having found a `k` such that the sum from `j+1` to `k-1` matches the sum from `k+1` to `n-1` and is also in `seen`, we have found a valid triplet `(i, j, k)` which is `(1, 3, 5)`.
     - We return `true` since we have found the required triplet.
5. **Returning the Result:**
   - In this case, we found at least one valid triplet that satisfies all the conditions. Hence, if we were to implement this example in code, the result would be `true`.

This example illustrates how the intuition and approach to the problem can be used to efficiently find a triplet in the array that meets the requirements.

Python Solution

```python
from typing import List

class Solution:
    def splitArray(self, nums: List[int]) -> bool:
        # Get the length of the input array
        n = len(nums)

        # Initialize a prefix sum array with an extra position for simplicity
        prefix_sum = [0] * (n + 1)

        # Compute the prefix sum array where prefix_sum[i] represents
        # the sum of elements from nums[0] to nums[i-1]
        for i, num in enumerate(nums):
            prefix_sum[i + 1] = prefix_sum[i] + num

        # The main loop to check for split positions starting at index 3
        # and ending at n-4 to ensure there are enough elements on both sides
        for mid in range(3, n - 3):
            # Store sums that can be created from the first half of the array
            seen_sums = set()

            # Check for all possible splits in the first half
            for left in range(1, mid - 1):
                # If a valid split is found, add the sum to the set
                if prefix_sum[left] == prefix_sum[mid] - prefix_sum[left + 1]:
                    seen_sums.add(prefix_sum[left])

            # Check for all possible splits in the second half of the array
            for right in range(mid + 2, n - 1):
                # Check if there is a valid split that matches any sum in 'seen_sums'
                if (prefix_sum[n] - prefix_sum[right + 1] == prefix_sum[right] - prefix_sum[mid + 1]
                        and prefix_sum[n] - prefix_sum[right + 1] in seen_sums):
                    return True

        # If we reach this point, no valid split was found
        return False
```

Java Solution

```java
class Solution {
    public boolean splitArray(int[] nums) {
        int n = nums.length;
        // Array to store the prefix sums, one extra element for ease of calculations
        int[] prefixSums = new int[n + 1];

        // Calculate the prefix sums
        for (int i = 0; i < n; ++i) {
            prefixSums[i + 1] = prefixSums[i] + nums[i];
        }

        // Traverse through the array, starting from index 3 to n - 4
        // 'j' is the potential middle split point
        for (int j = 3; j < n - 3; ++j) {
            Set<Integer> seenSums = new HashSet<>();
            // First pass to check possible sums from the left subarray
            for (int i = 1; i < j - 1; ++i) {
                if (prefixSums[i] == prefixSums[j] - prefixSums[i + 1]) {
                    seenSums.add(prefixSums[i]);
                }
            }

            // Second pass to check matching sums from the right subarray
            for (int k = j + 2; k < n - 1; ++k) {
                if (prefixSums[n] - prefixSums[k + 1] == prefixSums[k] - prefixSums[j + 1] && seenSums.contains(prefixSums[n] - prefi
                    return true; // Found a valid split
                }
            }
        }
        // If no valid split is found
        return false;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function that determines if the array can be split into four parts
    // with the same sum, with one element between these parts.
    bool splitArray(vector<int>& nums) {
        int n = nums.size();
        vector<int> prefixSum(n + 1, 0); // Initialize prefix sums array with an additional 0 at the start.

        // Calculate prefix sums for all elements.
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }

        // Use a three-pointer approach to find the split points.
        for (int middle = 3; middle < n - 3; ++middle) { // middle is the middle cut, avoiding the first 2 and last 2 elements.
            unordered_set<int> seenSums; // Store sums that we've seen which are candidates for the first section.

            // Find all possible sums for the left section.
            for (int left = 1; left < middle - 1; ++left) {
                if (prefixSum[left] == prefixSum[middle] - prefixSum[left + 1]) {
                    // If a sum that can be the left section is found, add it to 'seenSums'.
                    seenSums.insert(prefixSum[left]);
                }
            }

            // Find if there's a corresponding sum for the right section.
            for (int right = middle + 2; right < n - 1; ++right) {
                if (prefixSum[n] - prefixSum[right + 1] == prefixSum[right] - prefixSum[middle + 1]
                    && seenSums.count(prefixSum[n] - prefixSum[right + 1])) {
                    // If the sum for the right section equals one of the left section sums, return true for a successful split.
                    return true;
                }
            }
        }
        return false; // Return false if no such split is found.
    }
};
```

Typescript Solution

```typescript
// Function that determines if the array can be splitted into four parts with the same sum,
// with one element between these parts.
function splitArray(nums: number[]): boolean {
    let n: number = nums.length;
    let prefixSum: number[] = new Array(n + 1).fill(0); // Initialize prefix sums array with an additional 0 at the start.

    // Calculate prefix sums for all elements.
    for (let i = 0; i < n; ++i) {
        prefixSum[i + 1] = prefixSum[i] + nums[i];
    }

    // Use a three-pointer approach to find the split points.
    for (let middle = 3; middle < n - 3; ++middle) { // 'middle' is the middle cut, avoiding the first 2 and last 2 elements.
        let seenSums: Set<number> = new Set<number>(); // Store sums that we've seen which are candidates for the first section.

        // Find all possible sums for the left section
        for (let left = 1; left < middle - 1; ++left) {
            if (prefixSum[left] === prefixSum[middle] - prefixSum[left + 1]) {
                // If a sum that can be the left section is found, add it to 'seenSums'.
                seenSums.add(prefixSum[left]);
            }
        }

        // Find if there's a corresponding sum for the right section.
        for (let right = middle + 2; right < n - 1; ++right) {
            if (prefixSum[n] - prefixSum[right + 1] === prefixSum[right] - prefixSum[middle + 1]
                && seenSums.has(prefixSum[n] - prefixSum[right + 1])) {
                // If the sum for the right section equals one of the left section sums, return true for a successful split.
                return true;
            }
        }
    }
    return false; // Return false if no such split is found.
}

// Example usage:
let nums: number[] = [1, 2, 1, 2, 1, 2, 1];
console.log(splitArray(nums)); // Output should be true or false depending on the array content.
```

Time and Space Complexity

Time Complexity

The given Python function `splitArray` is designed to determine if an array can be split into four parts with equal sums. The function uses a prefix sum array `s` to efficiently calculate the sums of subarrays.

Looking at the nested loops, the first loop runs `(n-6)` times, where `j` ranges from 3 to `(n-4)`. The second nested loop runs at most `(j-2)` times for each `j`. The worst-case scenario for the second loop would be when `j` is around `n/2`, which would yield roughly `(n/2)` iterations. Thus, the innermost condition is checked `O(n^2)` times.

The third loop can be considered separately and again runs at most `(n-j-2)` iterations for each `j`. But this time, the check involves a hash set lookup, which is an `O(1)` operation on average. In the worst case, this loop will also contribute to `O(n^2)` iterations.

Overall, the time complexity of the code is `O(n^2)` due to the nested loops.

Space Complexity

The space complexity for this function is determined by the space required for:

1. The prefix sum array `s`, which contains `(n+1)` integers. This contributes `O(n)` space complexity.
2. The hash set `seen`, which in the worst-case scenario could store up to `((n/2)-2)` sums (since only sums before `j` are considered and `j` starts at 3). This also contributes `O(n)` space complexity.

Therefore, the overall space complexity of the function is `O(n)`.