1983. Widest Pair of Indices With Equal Range Sum

Medium | Arrays | Hash Table | Prefix Sum

Problem Description

In this problem, we have two binary arrays `nums1` and `nums2`. Both arrays are of the same length and contain only 0s and 1s. Our goal is to find the widest pair of indices `(i, j)` that satisfy two conditions:

1. `i` must be less than or equal to `j`, and
2. The sum of elements from `i` to `j` in `nums1` must be equal to the sum of elements from `i` to `j` in `nums2`.

The concept of the "widest" pair refers to the pair `(i, j)` with the biggest difference between `i` and `j`. This means that we are looking for the longest subarray where the sums of `nums1` and `nums2` are equal. The "distance" indicates the number of elements in this subarray, which is calculated as `j - i + 1`.

We need to return the "distance" of the widest pair of indices. If no pairs meet the condition, we should return 0.

Intuition

To find the solution, we use the cumulative sums of both arrays and keep track of their differences at each index. If the difference between the sums of subarrays from `nums1` and `nums2` up to the current index `j` has been seen before at some index `j`, then the subarrays from `j+1` to `j` in both `nums1` and `nums2` must be equal.

Here are the steps to solve this problem:

1. Maintain a dictionary `d` to store the first occurrence of each cumulative sum difference with its index. Initialize this dictionary with `{0: -1}`, meaning that if the difference is 0 at index 0, it can be considered as a subarray starting from index 0.
2. Iterate through the two arrays `nums1` and `nums2` simultaneously using `enumerate(zip(nums1, nums2))` to get both the index and the elements.
3. Calculate the cumulative difference `s` by subtracting the element of `nums2` from the element of `nums1` and adding it to the previous cumulative difference.
4. Check if this difference `s` has occurred before by looking it up in the dictionary `d`. If it exists, it means that there is a subarray that starts from the stored index plus 1 and ends at the current index `i` which has equal sums in `nums1` and `nums2`. Update `ans` with the maximum distance found so far, which is `i - d[s]`.
5. If the difference `s` does not exist in the dictionary, it means this is the first time we've seen this cumulative sum difference, so we add the current index `i` to the dictionary with the key being the sum difference `s`.
6. After traversing through the whole arrays, the variable `ans` will contain the distance of the widest pair of indices. Return `ans`.

Solution Approach

The solution to this problem utilizes a dictionary data structure for efficient lookups and an iterative approach to find the widest pair where the subarrays have equal sums. Here's a deep dive into how the `Solution` class implements this:

1. **Dictionary for Cumulative Difference Tracking**:
   - The dictionary `d` is used to store cumulative sum differences as keys and their first occurring indices as values. It's initialized with `{0: -1}` to handle cases where the subarray starts from the first element.
2. **Running Sum Difference Calculation**:
   As we loop through each element of both arrays, we calculate a running sum difference `s` by adding `nums1[i] - nums2[i]` to the current sum difference. This represents the difference between the cumulative sums up to the current index `i`.
3. **Lookup and Widest Pair Tracking**:
   - During each iteration, we check if the current sum difference `s` is found in dictionary `d`.
     - If it exists, it indicates we have previously seen this sum difference at an earlier index, meaning the sums of subarrays from the previous index to the current index in both `nums1` and `nums2` are equal.
     - The distance of the current widest pair is calculated by subtracting the earlier index where the sum difference was the same from the current index, i.e., `i - d[s]`.
     - We update `ans` to the maximum of the current distance and the existing widest distance found so far.
   - If the sum difference `s` is not found in `d`, it implies this difference is encountered the first time at index `i`, so we add `s` as a key to `d` with the current index `i` as its value. This will potentially serve as the starting index for a future widest pair.
4. **Returning the Result**:
   - After the loop is completed, `ans` holds the distance of the widest pair found. This value is returned as the output of the function.

The algorithm's complexity is primarily dictated by the single traversal of the input arrays, leading to an overall time complexity of O(n) where n is the number of elements in the input arrays. The space complexity is O(k), where k is the number of unique sum differences, which can be at most n in the worst case.

The solution is efficient because it avoids nested loops by using the cumulative sum difference, which allows us to identify equal sum subarrays in linear time.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach using two binary arrays `nums1` and `nums2`.

Initial State:

- Let `nums1 = [1, 0, 0, 1]` and `nums2 = [0, 1, 1, 0]`.
- Initialize the dictionary `d` with `{0: -1}` to handle the case where the sum difference starts from index 0.

Step-by-Step Solution:

1. Start iterating over both arrays. Initialize a variable `s` to keep track of the cumulative sum difference, and `ans` to store the maximum width found so far.
2. **Iteration 0:** The elements at index 0 are `nums1[0] = 1` and `nums2[0] = 0`.
   - The sum difference `s` becomes `1 - 0 = 1`.
   - This difference is not in `d`, so we add it: `d[1] = 0`.
3. **Iteration 1:** The elements at index 1 are `nums1[1] = 0` and `nums2[1] = 1`.
   - The sum difference `s` becomes `1 + (0 - 1) = 0`, which is already in `d`.
   - The width between `d[0] + 1 = 0` and index 1 is `1 - 0 + 1 = 2`.
   - Update `ans` to 2.
4. **Iteration 2:** The elements at index 2 are `nums1[2] = 0` and `nums2[2] = 1`.
   - The sum difference `s` remains `0` (since `0 - 0 + 0 - 1 = -1` and we add this to the previous `s` which was `1`, making the new `s` back to `0`).
   - We have already seen `0` in `d`.
   - The width is `2 - (-1) + 1 = 4`.
   - Update `ans` to 4, since this is larger than the previous value of `ans`.
5. **Iteration 3:** The elements at index 3 are `nums1[3] = 1` and `nums2[3] = 0`.
   - The sum difference `s` becomes `0 + (1 - 0) = 1`, which is already in `d`.
   - We do not update `ans` because the current distance `3 - d[1] = 3` is not greater than the current `ans` (which is 4).
6. After traversing all items, `ans` is 4, which represents the distance of the widest pair of indices where the sum of elements in `nums1` is equal to the sum of elements in `nums2`.

The solution finds the widest subarray efficiently, utilizing a dictionary to track the first occurrence of each cumulative sum difference and iterating over the arrays only once.

Solution Implementation

Python

```python
class Solution:
    def widestPairOfIndices(self, nums1: List[int], nums2: List[int]) -> int:
        # Initialize a dictionary to store the first occurrence of a
        # particular prefix sum difference. The sum difference is initialized
        # with a base case: 0 sum difference occurs at index -1.
        index_map = {0: -1}

        # Initialize variable to track the maximum width
        max_width = 0

        # Initialize a prefix sum difference variable
        sum_diff = 0

        # Loop through each pair of values from the two lists
        for index, (value1, value2) in enumerate(zip(nums1, nums2)):
            # Update the running sum difference with the difference of the current pair
            sum_diff += value1 - value2

            # If the sum difference has been seen before, calculate the width
            if sum_diff in index_map:
                # The width is the current index minus the index where the same
                # sum difference was first recorded
                width = index - index_map[sum_diff]
                # Update the maximum width if this width is greater
                max_width = max(max_width, width)
            else:
                # Otherwise, record the first occurrence of this sum difference
                index_map[sum_diff] = index

        # Return the maximum width found
        return max_width
```

Java

```java
class Solution {
    public int widestPairOfIndices(int[] nums1, int[] nums2) {
        // HashMap to store the first occurrence of the sum difference 'sumDiff' as a key and its index as the value.
        Map<Integer, Integer> firstOccurrenceMap = new HashMap<>();
        firstOccurrenceMap.put(0, -1); // Initialize with sumDiff 0 occurring before the array starts.

        int arrayLength = nums1.length; // Length of the input arrays.
        int sumDiff = 0; // Variable to keep track of the cumulative sum difference between nums1 and nums2.
        int maxDistance = 0; // Variable to store the maximum distance (widest pair) found.

        // Iterate over the elements of the input arrays.
        for (int i = 0; i < arrayLength; ++i) {
            // Calculate the cumulative sum difference at current index.
            sumDiff += nums1[i] - nums2[i];

            // Check if the cumulative sum difference has been seen before.
            if (firstOccurrenceMap.containsKey(sumDiff)) {
                // If seen before, calculate the distance and update the maximum distance if current is greater.
                maxDistance = Math.max(maxDistance, i - firstOccurrenceMap.get(sumDiff));
            } else {
                // If not seen before, record the first occurrence of this sum difference with its index.
                firstOccurrenceMap.put(sumDiff, i);
            }
        }

        // Return the maximum distance found (the widest pair of indices).
        return maxDistance;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to calculate the widest pair of indices where the sum of subarrays are equal
    int widestPairOfIndices(vector<int>& nums1, vector<int>& nums2) {
        unordered_map<int, int> prefixSum; // Map: prefix sum difference to the earliest index
        prefixSum[0] = -1; // Initialize with 0 difference at index -1
        int maxWidth = 0; // To store the maximum width
        int prefixSumDifference = 0; // To store the running difference of prefix sums
        int length = nums1.size(); // Length of the input arrays

        // Iterate over the array elements
        for (int i = 0; i < length; ++i) {
            // Update the running difference between nums1 and nums2 at index i
            prefixSumDifference += nums1[i] - nums2[i];

            // If this difference has been seen before, calculate width
            if (prefixSum.count(prefixSumDifference)) {
                // Update maxWidth if we found a wider pair
                maxWidth = max(maxWidth, i - prefixSum[prefixSumDifference]);
            } else {
                // Else, store the current index for this difference
                prefixSum[prefixSumDifference] = i;
            }
        }

        // Return the maximum width found
        return maxWidth;
    }
};
```

TypeScript

```typescript
// Importing necessary utility
import { max } from "lodash";

// Function to calculate the widest pair of indices where the sum of subarrays are equal
function widestPairOfIndices(nums1: number[], nums2: number[]): number {
    const prefixSum: { [key: number]: number } = {}; // Map: prefix sum difference to the earliest index
    prefixSum[0] = -1; // Initialize with 0 difference at index -1
    let maxWidth = 0; // To store the maximum width
    let prefixSumDifference = 0; // To store the running difference of prefix sums
    let length = nums1.length; // Length of the input arrays

    // Iterate over the array elements
    for (let i = 0; i < length; ++i) {
        // Update the running difference between nums1 and nums2 at index i
        prefixSumDifference += nums1[i] - nums2[i];

        // If this difference has been seen before, calculate width
        if (prefixSum[prefixSumDifference] !== undefined) {
            // Update maxWidth if we found a wider pair
            maxWidth = max([maxWidth, i - prefixSum[prefixSumDifference]]);
        } else {
            // Else, store the current index for this difference
            prefixSum[prefixSumDifference] = i;
        }
    }
    // Return the maximum width found
    return maxWidth;
}
```

```python
class Solution:
    def widestPairOfIndices(self, nums1: List[int], nums2: List[int]) -> int:
        # Initialize a dictionary to store the first occurrence of a
        # particular prefix sum difference. The sum difference is initialized
        # with a base case: 0 sum difference occurs at index -1.
        index_map = {0: -1}

        # Initialize variable to track the maximum width
        max_width = 0

        # Initialize a prefix sum difference variable
        sum_diff = 0

        # Loop through each pair of values from the two lists
        for index, (value1, value2) in enumerate(zip(nums1, nums2)):
            # Update the running sum difference with the difference of the current pair
            sum_diff += value1 - value2

            # If the sum difference has been seen before, calculate the width
            if sum_diff in index_map:
                # The width is the current index minus the index where the same
                # sum difference was first recorded
                width = index - index_map[sum_diff]
                # Update the maximum width if this width is greater
                max_width = max(max_width, width)
            else:
                # Otherwise, record the first occurrence of this sum difference
                index_map[sum_diff] = index

        # Return the maximum width found
        return max_width
```

Time and Space Complexity

The given Python function `widestPairOfIndices` seeks to find the widest pair of indices `(i, j)` such that `sum(nums1[k]) for k in range(i, j + 1)` is equal to `sum(nums2[k]) for k in range(i, j + 1)`. It accomplishes this by using a dictionary `d` to keep track of the differences between the prefixes of the two lists and when each difference was first seen.

Time Complexity:

The time complexity of the function can be analyzed as follows:

- There is a single loop that iterates over the elements of `nums1` and `nums2`, which both have the same length, say `n`.
- Within each iteration, the function performs constant-time operations: a calculation involving `a` and `b`, an `if` condition check, a dictionary lookup, and an update to the dictionary (which on average is constant time for a hash table) and variable for the answer `ans`.
- These operations within the loop are conducted in constant time, regardless of the size of the input.

As the loop runs `n` times where `n` is the length of the input lists, and each iteration is done in constant time, the time complexity of the function is $O(n)$.

Space Complexity:

The space complexity of the function depends on the auxiliary space used, which is mainly for storing the dictionary `d`:

- In the worst-case scenario, the dictionary could store a distinct entry for each element in the input lists if all prefix sums were unique. Thus, in the worst case, it could have up to `n` entries.
- Aside from this, the space used by variables `s` and `ans` is constant, as well as a few other intermediate variables.

Therefore, the worst-case space complexity is $O(n)$ where `n` is the length of the input lists.