2321. Maximum Score Of Spliced Array

Hard · Array · Dynamic Programming · Leetcode Link

Problem Description

In this problem, we are given two integer arrays, `nums1` and `nums2`, both with the same length `n`. We have the option to perform a single operation: choose two indices `left` and `right` such that `0 <= left <= right < n` and swap the subarray `nums1[left...right]` with the subarray `nums2[left...right]`. This operation can be done only once or we may choose not to perform any swap at all.

The goal is to maximize the score of the arrays after the swap operation. The score is defined as the maximum sum of either `nums1` or `nums2`. The objective is to determine the maximum possible score. A subarray is a contiguous sequence of elements within an array, and `arr[left...right]` represents the subarray that contains all elements in `nums` from index `left` to index `right`, inclusive.

Intuition

To arrive at the solution, we need to determine if performing the swap operation would be beneficial and if so, which indices to choose for the left and right boundaries of the subarrays to swap. Since we are trying to maximize the sum of the higher-scoring array, we should look for the subarray within `nums1` and `nums2` that, when swapped, will lead to the greatest increase in sum for the resulting array.

The intuition here is to calculate the difference between corresponding elements of `nums1` and `nums2`. By finding the contiguous subsequence with the maximum sum (max subarray sum) which can be positive or negative, we can identify the impact of swapping this particular subarray on the overall score.

We implement a helper function `f(nums1, nums2)` that calculates this maximum sum for the differences between `nums1` and `nums2`. This is similar to the classic maximum subarray problem, which can be solved using Kadane's algorithm. For the chosen subarray, we add this difference to the sum of `nums2`, and similarly, for the opposite operation, we add the difference to the sum of `nums1` when considering the swap from `nums2` to `nums1`.

In the provided solution, the `maximumSplicedArray` function performs this comparison and returns the higher value achieved by either adding the maximum contiguous sum difference of `nums1` to the sum of `nums2`, or vice versa. The maximum of these two gives us the maximum possible score after performing the optimal swap operation, and that's what we return as the solution.

Solution Approach

The implementation of the solution involves a function `f(nums1, nums2)` that employs the idea of Kadane's algorithm to find the maximum subarray sum difference between the two input arrays `nums1` and `nums2`. Kadane's algorithm is a classic dynamic programming approach to solve the maximum subarray problem.

Here's a step-by-step breakdown of the algorithm and patterns used in the implementation:

1. **Difference Calculation**: We start by calculating the difference array `d` where each element `d[i]` is the result of `nums1[i] - nums2[i]`. This array is critical because it represents the change in the sum when we swap corresponding elements from `nums1` and `nums2`.
2. **Kadane's Algorithm**: The helper function `f(nums1, nums2)` runs a modified version of Kadane's algorithm on the difference array `d`. The algorithm iterates through the array and looks for the subarray with the maximum sum, which indicates the best subarray to swap (if beneficial):
   - Initialize two variables `t` and `mx` with the value of the first element in the array `d`.
   - Iterate through the difference array starting from the second element, updating `t` with the sum of `t` and the current element if `t` is positive; otherwise, reset `t` to the current element's value.
   - At each step, update `mx` to be the maximum of `mx` and `t`.
   - After iterating through the entire array, `mx` holds the maximum subarray sum difference.
3. **Sum Calculation and Comparison**: We calculate `s1` and `s2` as the sums of `nums1` and `nums2` respectively. The final score is computed by adding the maximum subarray sum difference to the sum of the opposite array.
   - To consider the swap from `nums1` to `nums2`, compute `s2 + f(nums1, nums2)` which represents the score if we swap the most beneficial subarray from `nums1` to `nums2`.
   - To consider the swap from `nums2` to `nums1`, compute `s1 + f(nums2, nums1)` which represents the score if we swap the most beneficial subarray from `nums2` to `nums1`.
4. **Maximization**: Since we want the maximum score, we take the maximum value from the two calculated scores mentioned above. This gives us the maximum possible score after considering whether a swap would be advantageous or not.

The final call `max(s2 + f(nums1, nums2), s1 + f(nums2, nums1))` evaluates both scenarios and returns the higher score as the solution to the problem.

This implementation is efficient because Kadane's algorithm runs in linear time `O(n)`, and the rest of the operations are simple sums and comparisons, also done in linear time. Thus, the overall time complexity of the solution is `O(n)`.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose `nums1 = [1, 3, 5, 7, 9]` and `nums2 = [2, 4, 6, 8, 10]` with `n = 5`.

Step 1: Difference Calculation

First, we compute the difference array `d` by subtracting each element of `nums1` from `nums2`:

`d = [2 - 1, 4 - 3, 6 - 5, 8 - 7, 10 - 9] d = [1, 1, 1, 1, 1]`

Step 2: Kadane's Algorithm

Now, let's apply Kadane's algorithm to find the maximum subarray sum in `d`.

- Initialize `t = d[0] = 1` and `mx = d[0] = 1`.
- Move through the difference array starting from index 1:
  - At each element `d[i]`, if `t > 0`, then update `t = t + d[i]`.
  - Otherwise, set `t = d[i]`.
  - Update `mx` to the larger of `mx` or `t` at each step.

After iterating through the array `d`, the maximum subarray sum `mx` turns out to be `5`.

Step 3: Sum Calculation and Comparison

Calculate the sum of the original arrays:

- `s1 = sum(nums1) = 1 + 3 + 5 + 7 + 9 = 25`
- `s2 = sum(nums2) = 2 + 4 + 6 + 8 + 10 = 30`

We consider the scores after the swap operation:

- Swapping a beneficial subarray from `nums1` to `nums2` gives us `s2 + mx = 30 + 5 = 35`.
- Swapping a beneficial subarray from `nums2` to `nums1` gives us `s1 + mx = 25 + 5 = 30`.

Step 4: Maximization

Finally, we choose the higher of the two calculated scores:

- `max(35, 30) = 35`

Hence, the operation that gives us the maximum score is swapping the beneficial subarray from `nums1` to `nums2`, and the maximum score achievable is `35`. This is our final solution.

Python Solution

```python
from typing import List

class Solution:
    def maximumSplicedArray(self, nums1: List[int], nums2: List[int]) -> int:
        # Helper function to calculate the maximum difference between subarrays
        def max_subarray_difference(nums1, nums2):
            # Calculate the differences between elements of the two arrays
            differences = [a - b for a, b in zip(nums1, nums2)]

            # Initialize the current and maximum subarray sum with the first difference element
            current_sum = max_sum = differences[0]

            # Loop through the rest of the differences
            for value in differences[1:]:
                # If current_sum is positive, add the value to it; else, start a new subarray sum with the value
                if current_sum > 0:
                    current_sum += value
                else:
                    current_sum = value
                # Update max_sum with the maximum sum found so far
                max_sum = max(max_sum, current_sum)

            # Return the maximum subarray sum found
            return max_sum

        # Calculate the total sums of both input arrays
        total_sum1, total_sum2 = sum(nums1), sum(nums2)

        # Calculate the maximum possible sum by splicing subarrays from one array to another
        # Two cases: splicing from nums1 to nums2, and from nums2 to nums1
        # We add the subarray_sum to the total sum of the opposite array
        return max(
            total_sum2 + max_subarray_difference(nums1, nums2),
            total_sum1 + max_subarray_difference(nums2, nums1)
        )

# The Solution class can now be used to find the maximum spliced array sum

```

Java Solution

```java
class Solution {
    public int maximumsSplicedArray(int[] nums1, int[] nums2) {
        int sum1 = 0, sum2 = 0; // Initialize sums of both arrays to 0
        int length = nums1.length; // Length of the arrays
        // Calculate the sum of each array
        for (int i = 0; i < length; ++i) {
            sum1 += nums1[i];
            sum2 += nums2[i];
        }
        // Return the maximum sum possible by splicing. Calculate twice, swapping the arrays, to ensure all possibilities are examine
        return Math.max(sum2 + delta(nums1, nums2), sum1 + delta(nums2, nums1));
    }

    // Helper function to calculate the maximum difference subarray when splicing nums2 into nums1, represented as 'delta'.
    private int delta(int[] nums1, int[] nums2) {
        int temporarySum = nums1[0] - nums2[0];
        int maxDiff = temporarySum; // Maximum difference found so far

        // Iterate over the arrays starting from the second element
        for (int i = 1; i < nums1.length; ++i) {
            int valueDifference = nums1[i] - nums2[i];

            // If the current temporary sum is positive, continue the subarray
            if (temporarySum > 0) {
                temporarySum += valueDifference;
            } else {
                // Else start a new subarray
                temporarySum = valueDifference;
            }
            // Update the maximum difference if the new temporary sum is greater
            maxDiff = Math.max(maxDiff, temporarySum);
        }

        return maxDiff; // Return the maximum difference found
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the maximum sum we can obtain by splicing two arrays
    int maximumsSplicedArray(vector<int>& nums1, vector<int>& nums2) {
        // Initialize the sum for each array
        int sumNums1 = 0, sumNums2 = 0;
        int n = nums1.size(); // Get the size of the arrays

        // Calculate the sum of elements for each array
        for (int i = 0; i < n; ++i) {
            sumNums1 += nums1[i];
            sumNums2 += nums2[i];
        }

        // The maximum sum is the maximum of splicing in both directions
        return max(sumNums2 + getMaxDiff(nums1, nums2), sumNums1 + getMaxDiff(nums2, nums1));
    }

    // Helper function to calculate the maximum difference splicing sequence
    int getMaxDiff(vector<int>& nums1, vector<int>& nums2) {
        // Starting difference between the first elements of both arrays
        int currentDiff = nums1[0] - nums2[0];
        int maxDiff = currentDiff; // Initialize maxDiff with the first difference

        // Iterate through the arrays starting from the second element
        for (int i = 1; i < nums1.size(); ++i) {
            int diff = nums1[i] - nums2[i]; // Calculate the difference for the current index

            // If the accumulated difference is positive, continue the sequence
            // Otherwise, start a new sequence with the current difference
            currentDiff = (currentDiff > 0) ? currentDiff + diff : diff;

            // Update maxDiff if the current accumulated difference is larger
            maxDiff = max(maxDiff, currentDiff);
        }

        // Return the maximum difference found
        return maxDiff;
    }
};

```

Typescript Solution

```typescript
// Function to calculate the sum of the elements of an array
function sumArray(array: number[]): number {
  return array.reduce((acc, value) => acc + value, 0);
}

// Function to find the maximum sum we can obtain by splicing two arrays
function maximumSplicedArray(nums1: number[], nums2: number[]): number {
  // Calculate the sum of elements for each array using the sumArray function
  const sumNums1: number = sumArray(nums1);
  const sumNums2: number = sumArray(nums2);

  // The maximum sum is the maximum of splicing in both directions
  return Math.max(
    sumNums1 + getMaxDiff(nums2, nums1),
    sumNums2 + getMaxDiff(nums1, nums2)
  );
}

// Helper function to calculate the maximum difference splicing sequence
function getMaxDiff(nums1: number[], nums2: number[]): number {
  // Starting difference between the first elements of both arrays
  let currentDiff: number = nums1[0] - nums2[0];
  let maxDiff: number = currentDiff; // Initialize maxDiff with the first difference

  // Iterate through the arrays starting from the second element
  for (let i: number = 1; i < nums1.length; i++) {
    const diff: number = nums1[i] - nums2[i]; // Calculate the difference for the current index

    // If the accumulated difference is positive, continue the sequence
    // Otherwise, start a new sequence with the current difference
    currentDiff = currentDiff > 0 ? currentDiff + diff : diff;

    // Update maxDiff if the current accumulated difference is larger
    maxDiff = Math.max(maxDiff, currentDiff);
  }

  // Return the maximum difference found
  return maxDiff;
}

```

Time and Space Complexity

The given Python code defines a method `maximumsSplicedArray` which compares the sums of two arrays after possibly splicing in continuous subarray sections from one to the other to maximize the sum. The code makes use of a helper function `f` which computes the maximum subarray sum difference.

Time Complexity

The time complexity of the function `maximumsSplicedArray` is determined by the helper function `f`, which is called twice, and the sum computation for each array.

Here is the breakdown of the time complexity:

- Computing the sum for `nums1` and `nums2` each takes `O(n)` time where `n` is the length of each array.
- The function `f` is called twice, each call involves:
  - Creating a difference array `d` which takes `O(n)` time.
  - Iterating over the difference array excluding the first element to find the maximum subarray sum difference, which also takes `O(n)` time due to the use of the Kadane's algorithm approach.

So the overall time complexity of the function is `O(n)` for the sum computations plus `2 * O(n)` for the two calls to function `f`, resulting in a total time complexity of `O(n)`.

Space Complexity

As for space complexity:

- The difference array `d` requires `O(n)` space.
- The variables `t`, `mx`, and the sums `s1` and `s2` use constant space `O(1)`.

Hence, the space complexity of the function `maximumsSplicedArray` is `O(n)`.