918. Maximum Sum Circular Subarray

Medium Queue Array Divide and Conquer Dynamic Programming Monotonic Queue Leetcode Link

Problem Description

The task is to find the maximum sum of a non-empty subarray within a given circular integer array, `nums`. A circular array means that the array is conceptually connected end-to-end, so the elements wrap around. For example, in such an array, the next element of the last element is the first element of the array, and the previous element of the first element is the last element of the array.

The critical challenge here is accounting for the circular nature of the array, which allows for subarrays that cross the end of the array and start again at the beginning. The maximum sum could come from a subarray in the middle of the array, a subarray that includes elements from both ends of the array, or even the entire array if all elements are positive.

The subarray we are trying to find should not include the same element more than once, meaning we can't wrap around the circular array more than once when selecting our subarray elements.

Intuition

To solve the maximum subarray problem for a circular array, we must consider two cases:

1. The maximum sum subarray is similar to the one found in a non-circular array (Kadane's algorithm is useful here).
2. The maximum sum subarray is the result of wrapping around the circular array.

For the first case, we use Kadane's algorithm to find the maximum sum subarray that does not wrap around. This is done by iterating through the array while maintaining the maximum sum ending at each index (denoted as `f1` in the code), and the maximum sum so far (denoted as `s1`).

For the second case, to handle subarrays that wrap, consider that the answer could be the total sum of the array minus the minimum sum subarray. This is akin to "selecting" the rest of the array that doesn't include the minimum sum subarray. To find the minimum sum subarray, we modify Kadane's algorithm to keep track of the minimum sum ending at each index (denoted as `f2`), and the minimum sum so far (denoted as `s2`).

At the end, the maximum possible sum for the case when the subarray wraps around is computed as the total sum of the array minus the minimum sum subarray (`sum(nums) - s2`).

The edge case to consider is when all numbers in the array are negative. In this situation, Kadane's algorithm yields a subarray sum which is the maximum negative number (i.e., the least negative), and thus is the maximum sum we can achieve without wrapping around. Subtracting any subarray would result in a smaller sum, so we return the maximum subarray sum found without wrapping in this case.

Putting it all together, the final answer is the larger of the two possibilities: the maximum subarray sum found without wrapping (using Kadane's algorithm) or the total array sum minus the minimum subarray sum, unless all numbers are negative, in which case we return the maximum subarray sum without wrapping.

Solution Approach

The solution approach involves implementing two variations of Kadane's algorithm once for finding the maximum subarray sum, and another time for finding the minimum subarray sum. We then combine the results of these variations to account for the circular nature of the array.

Here's how we approach it step by step:

1. **Initialization**: First, we initialize four variables `s1`, `s2`, `f1`, and `f2` with the value of the first element in the array. `s1` will hold the maximum subarray sum found so far, `s2` the minimum subarray sum, `f1` the current maximum subarray sum ending at the current index, and `f2` the current minimum subarray sum ending at the current index.

2. **Iterate Over Array**: Start iterating the array from the second element because the first element is used for initialization.

3. **Kadane's Algorithm for Maximum Sum**: For each element `num`, we calculate the current maximum ending here (`f1`) as the maximum of `num` and `num + f1` (i.e., either start a new subarray from current element or add the current element to the existing subarray). We then update the overall maximum (`s1`) with the maximum of itself and `f1`.

4. **Kadane's Algorithm for Minimum Sum**: Likewise, we calculate the current minimum ending here (`f2`) as the minimum of `num` and `num + f2` (i.e., either start a new subarray from current element or add the current element to the existing negative subarray). We update the overall minimum (`s2`) with the minimum of itself and `f2`.

5. **Check for All Negative Elements**: After iterating through the array, we check if the maximum subarray sum found (`s1`) is less than or equal to 0. If true, then all the elements are negative, and `s1` is our answer as no subarray wrapping can result in a higher sum.

6. **Find Maximum Sum Considering Circular Wrap**: Otherwise, we find the maximum of `s1` and `sum(nums) - s2`. This accounts for the possibility that the maximum sum subarray might be wrapping over the circular array. We subtract `s2` (the minimum sum subarray) from the total sum of the array, which effectively gives us the sum of the subarray that wraps around.

7. **Return the Result**: The maximum of these two values gives us our answer, which is the maximum sum of a non-empty subarray for the circular array.

The final Python function looks like:

```python
class Solution:
    def maxSubarraySumCircular(self, nums: List[int]) -> int:
        s1 = s2 = f1 = f2 = nums[0]
        for num in nums[1:]:
            f1 = num + max(f1, 0)
            f2 = num + min(f2, 0)
            s1 = max(s1, f1)
            s2 = min(s2, f2)
        return s1 if s1 <= 0 else max(s1, sum(nums) - s2)
```

In this code snippet:

- `max(f1, 0)` and `min(f2, 0)` are used to decide whether to start a new subarray or to continue with the current subarray.
- `sum(nums)` is called only once to improve efficiency, just before the comparison of sums, as it could be computationally expensive for large arrays to call it repeatedly.

Example Walkthrough

Let's illustrate the solution approach using a small example with the circular array `nums = [5, -3, 5]`.

1. **Initialization**: We initialize `s1`, `s2`, `f1`, and `f2` with the value of the first element in the array. So, `s1 = s2 = f1 = f2 = 5`.

2. **Iterate Over Array**: Start iterating from the second element `-3`.

3. **Kadane's Algorithm for Maximum Sum**:

   - For `-3`: `f1 = max(-3, 5 - 3) = max(-3, 2) = 2`, so `s1 = max(5, 2) = 5`.
   - For `5` (last element): `f1 = max(5, 5 + 2) = max(5, 7) = 7`, which is a wrap-around as we count `5` from both ends of the array. `s1` is updated to `max(5, 7) = 7`.

4. **Kadane's Algorithm for Minimum Sum**:

   - For `-3`: `f2 = min(-3, 5 - 3) = min(-3, 2) = -3`, so `s2` becomes `min(5, -3) = -3`.
   - For `5`: `f2 = min(5, 5 - 3) = min(5, 2) = 2`, and `s2` remains at `-3`.

5. **Check for All Negative Elements**: Since the maximum subarray sum `s1` is greater than 0, not all elements are negative.

6. **Find Maximum Sum Considering Circular Wrap**: We find `max(7, sum([5, -3, 5]) - -3)`. The sum of `nums` is `7`, and subtracting `s2` (which is `-3`), we get `7 - (-3) = 10`.

7. **Return the Result**: The answer is the maximum of `7` and `10`, which is `10`. So the maximum sum of a non-empty subarray for the given circular array `nums` is `10`.

This walkthrough clearly demonstrates how the algorithm incorporates circularity by considering the inverse of the minimum subarray sum to find the potential maximum in a wrapping scenario. It also shows how to handle non-wrap-around scenarios using Kadane's algorithm directly for the maximum subarray sum.

Python Solution

```python
from typing import List

class Solution:
    def max_subarray_sum_circular(self, nums: List[int]) -> int:
        max_sum_end_here = min_sum_end_here = max_subarray_sum = min_subarray_sum = nums[0]

        # Iterate through the given nums list starting from the second element
        for num in nums[1:]:
            # Update max_sum_end_here to be the maximum of the current number or the current number plus max_sum_end_here
            max_sum_end_here = num + max(max_sum_end_here, 0)
            # Update min_sum_end_here to be the minimum of the current number or the current number plus min_sum_end_here
            min_sum_end_here = num + min(min_sum_end_here, 0)

            # Update the max_subarray_sum if the newly computed max_sum_end_here is larger
            max_subarray_sum = max(max_subarray_sum, max_sum_end_here)
            # Update the min_subarray_sum if the newly computed min_sum_end_here is smaller
            min_subarray_sum = min(min_subarray_sum, min_sum_end_here)

        # If the max_subarray_sum is non-positive, the whole array could be non-positive
        # Thus, the max subarray sum is the max_subarray_sum itself
        if max_subarray_sum <= 0:
            return max_subarray_sum
        else:
            # Otherwise, we compare the max_subarray_sum vs. total_sum minus min_subarray_sum
            # The latter represents the maximum sum obtained by considering the circular nature of the array
            # We subtract min_subarray_sum from the total sum to get the maximum sum subarray which wraps around the array
            total_sum = sum(nums)
            return max(max_subarray_sum, total_sum - min_subarray_sum)

# Usage example:
# solution = Solution()
# print(solution.max_subarray_sum_circular([5, -3, 5])) # Output: 10

```

Java Solution

```java
class Solution {
    public int maxSubarraySumCircular(int[] nums) {
        // Initialize variables to hold
        // s1 and s2 as the max and min subarray sums respectively
        // f1 and f2 as the local max and min subarray sums at the current position
        // total as the sum of all numbers in the array
        int maxSubarraySum = nums[0], minSubarraySum = nums[0];
        int currentMaxSum = nums[0], currentMinSum = nums[0], totalSum = nums[0];

        // Iterate through the array starting from the second element
        for (int i = 1; i < nums.length; ++i) {
            // Update the total sum
            totalSum += nums[i];

            // Calculate the local max subarray sum (Kadane's algorithm)
            currentMaxSum = nums[i] + Math.max(currentMaxSum, 0);
            // Determine the global max subarray sum so far
            maxSubarraySum = Math.max(maxSubarraySum, currentMaxSum);

            // Calculate the local min subarray sum (Inverse Kadane's algorithm)
            currentMinSum = nums[i] + Math.min(currentMinSum, 0);
            // Determine the global min subarray sum so far
            minSubarraySum = Math.min(minSubarraySum, currentMinSum);
        }

        // If maxSubarraySum is non-positive, all numbers are non-positive.
        // Hence, return maxSubarraySum because wrapping doesn't make sense.
        // Otherwise, return the maximum between maxSubarraySum and
        // totalSum - minSubarraySum which represents the maximum circular subarray sum.
        return maxSubarraySum > 0 ? Math.max(maxSubarraySum, totalSum - minSubarraySum) : maxSubarraySum;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    int maxSubarraySumCircular(vector<int>& nums) {
        int maxEndingHere = nums[0]; // current max subarray ending at this position
        int minEndingHere = nums[0]; // current min subarray ending at this position
        int maxSoFar = nums[0];      // max subarray found so far
        int minSoFar = nums[0];      // min subarray found so far
        int totalSum = nums[0];      // sum of all elements

        // Loop starting from the second element in nums
        for (int i = 1; i < nums.size(); ++i) {
            totalSum += nums[i]; // accumulate the total sum of the array

            // Update maxEndingHere by including the current number or restarting at current number if it is bigger
            maxEndingHere = nums[i] + std::max(maxEndingHere, 0);
            // Update maxSoFar to the maximum of itself and maxEndingHere
            maxSoFar = std::max(maxSoFar, maxEndingHere);

            // Update minEndingHere by including the current number or restarting at current number if it is smaller
            minEndingHere = nums[i] + std::min(minEndingHere, 0);
            // Update minSoFar to the minimum of itself and minEndingHere
            minSoFar = std::min(minSoFar, minEndingHere);
        }

        // If maxSoFar is less than 0, all numbers are negative, return maxSoFar directly
        // Otherwise, return the maximum of maxSoFar and totalSum - minSoFar
        // (since the maximum sum circular subarray may wrap around the end of the array)
        return maxSoFar > 0 ? std::max(maxSoFar, totalSum - minSoFar) : maxSoFar;
    }
};

```

Typescript Solution

```typescript
/**
 * Finds the maximum sum of a non-empty subarray for a circular array.
 *
 * @param {number[]} nums - The input array of numbers.
 * @returns {number} - The maximum sum of the subarray.
 */
function maxSubarraySumCircular(nums: number[]): number {
    let currentMax = nums[0]; // Initialize current max subarray sum ending at the current position
    let globalMax = nums[0];  // Initialize the global max subarray sum found so far
    let currentMin = nums[0]; // Initialize current min subarray sum ending at the current position
    let globalMin = nums[0];  // Initialize the global min subarray sum found so far
    let totalSum  = nums[0];  // Initialize the total sum of the array

    for (let i = 1; i < nums.length; ++i) {
        const currentValue = nums[i]; // Current element being processed
        totalSum += currentValue; // Add the current value to the total sum

        // Calculate max subarray sum for a normal array (not accounting for circularity)
        currentMax = Math.max(currentMax + currentValue, currentValue);
        globalMax = Math.max(currentMax, globalMax);

        // Calculate min subarray sum for a normal array (to help with circular cases)
        currentMin = Math.min(currentMin + currentValue, currentValue);
        globalMin = Math.min(currentMin, globalMin);
    }

    // If all numbers are negative, max subarray sum is the maximum element (as taking an empty subarray is not allowed)
    // Otherwise, return the maximum between the globalMax and totalSum - globalMin (which accounts for wrap-around subarrays)
    return globalMax > 0 ? Math.max(globalMax, totalSum - globalMin) : globalMax;
}

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input list `nums`. This is because the code iterates through all the elements of the array exactly once to calculate the maximum subarray sum for both the non-circular and circular cases.

The space complexity of the code is `O(1)` since it only uses a constant amount of extra space for variables `s1`, `s2`, `f1`, `f2`, and some temporary variables to perform the calculations, regardless of the size of the input list.