1191. K-Concatenation Maximum Sum

Medium Array Dynamic Programming Leetcode Link

Problem Description

The problem deals with finding the maximum sum of a sub-array within a modified array, which is the result of repeating the original integer array `arr` a total of `k` times. The challenge involves not just the repetition of the array but computing the maximum possible sum of a contiguous sub-array. This includes the possibility of a sub-array having a length of `0`, which would correspond to a sum of `0`.

Intuition

To tackle this problem, we need to understand a few important concepts:

1. Kadane's Algorithm: This algorithm is used for finding the maximum sum of a contiguous sub-array within a single unmodified array. It does this by looking for all positive contiguous segments of the array (max_ending_here) and keeping track of the maximum sum contiguous segment among all positive segments (max_so_far). The algorithm increments the sum when the running sum is positive, and resets it to zero when the running sum becomes negative.

2. Prefix Sum and Suffix Sum: The prefix sum is the sum of elements from the start of the array up to a certain index, and the suffix sum is the sum of elements from a certain index to the end of the array. These can be useful when handling repeated arrays, as the maximum sum can span across the boundary between two repeated segments.

Considering these concepts, the intuition for the solution approach can be broken down into steps:

- The `Kadane's Algorithm` is used to find the maximum sum of a contiguous sub-array from the original `arr`.
- The sum of the entire array (`s`) is calculated to be used in checking if we can increase the maximum sum by including sums from multiple repetitions of `arr`.
- We find the maximum prefix sum (`mx_pre`), i.e., the largest sum obtained from the start of an array up to any index.
- We find the maximum suffix sum (`mx_suf`), i.e., the largest sum that starts at any index and goes to the end of an array.
- Based on the value of `k`, we decide how to combine these sums to compute the final answer:
  - If `k` is equal to 1, the maximum sub-array sum is only within the single original array and it's the result obtained from `Kadane's Algorithm`.
  - If `k` is greater than 1 and the total sum of the array (`s`) is positive, the maximum sum could potentially span across the middle arrays completely, so we consider the array sum multiplied by (k-2) and add both the maximum prefix and suffix sums.
  - If `k` is greater than 1 and the total sum of the array is non-positive, we just need to consider the maximum prefix and suffix sums, as spanning across copies would not increase the overall sub-array sum.

Finally, since the answer can be very large, we return the result modulo `10^9 + 7` to ensure it stays within the integer limits for the problem.

Solution Approach

In the provided solution, the implementation walks through the array and applies the concepts mentioned in the Intuition section.

Here's a breakdown of the logic used:

1. **Initialization**: We start by initializing variables to store the sum of the array (`s`), the maximum prefix sum (`mx_pre`), the minimum prefix sum (`mi_pre`), and the maximum sub-array sum (`mx_sub`).

2. **Single Pass for Kadane's Algorithm and Prefix Sums**:

   - We loop through each element in `arr`:
     - Update the sum of the array (`s`) by adding the current element `x`.
     - Update the maximum prefix sum (`mx_pre`) to the maximum of `mx_pre` and the current sum `s`.
     - Update the minimum prefix sum (`mi_pre`) to the minimum of `mi_pre` and the current sum `s`.
     - Update the maximum sub-array sum (`mx_sub`) to the maximum of the current `mx_sub` and the difference between the current sum `s` and the minimum prefix sum (`mi_pre`), which represents Kadane's algorithm execution for the sub-array ending at the current element.

3. **Handling Multiple Concatenations**:

   - After the loop, we calculate the suffix maximum sum (`mx_suf`) by subtracting the minimum prefix sum (`mi_pre`) from the total sum of the array (`s`).
   - The base answer variable `ans` is initialized with the value obtained from Kadane's Algorithm (`mx_sub`).
   - If `k` equals 1, which means the array is not concatenated, we use the answer gotten from the Kadane's pass earlier and return it modulo `10^9 + 7`.
   - For the case where `k` is greater than 1, we explore the options by combining different parts of the array:
     - First, we use the maximum prefix sum plus the maximum suffix sum (`mx_pre + mx_suf`) and update `ans` if this is greater than the current `ans`.
     - Next, if the sum of the array (`s`) is positive, we explore the possibility that the maximum sum spans across the entire middle part of the concatenated array. This leads us to consider `(k - 2) * s + mx_pre + mx_suf`. We then again update `ans` if this is greater than the current `ans`.

4. **Returning the Result**:

   - In the final step, we return the answer `ans` modulo `10^9 + 7`.

This approach effectively handles the possibility of maximizing the sub-array sum by including the sum of the entire array when it is beneficial (i.e., when the total sum is positive) due to the concatenation specified by `k`. It ensures that the maximum sub-array sum is found even if it spans across multiple copies of the array.

Example Walkthrough

Let's walk through an example to illustrate the solution approach:

Suppose our given array is `arr = [3, -1, 2]` and `k = 2`. That means we need to find the maximum sub-array sum in an array that would look like `[3, -1, 2, 3, -1, 2]` after concatenating `arr` to itself once (as `k = 2`).

1. **Initialization**:

   - `s` = 0 (sum of the array)
   - `mx_pre` = 0 (maximum prefix sum)
   - `mi_pre` = 0 (minimum prefix sum)
   - `mx_sub` = 0 (maximum sub-array sum)

2. **Single Pass for Kadane's Algorithm and Prefix Sums**:

   - For the first element `3`:
     - `s` = `3`
     - `mx_pre` = `3`
     - `mi_pre` = `0`
     - `mx_sub` = `3`
   - For the second element `-1`:
     - `s` = `3 - 1` = `2`
     - `mx_pre` remains `3`
     - `mi_pre` remains `0`
     - `mx_sub` remains `3`
   - For the third element `2`:
     - `s` = `2 + 2` = `4`
     - `mx_pre` = `4`
     - `mi_pre` remains `0`
     - `mx_sub` = `4` (since `4` - `0` > `3`)

3. **Handling Multiple Concatenations**:

   - After the loop, we calculate `mx_suf` which is `s - mi_pre` = `4 - 0` = `4`.
   - The base answer `ans` is `mx_sub` which is currently `4`.
   - Since `k > 1`, we examine the maximum sum using concatenation:
     - We calculate `mx_pre + mx_suf` = `4 + 4` = `8` and compare it with `ans`, thus `ans` becomes `8`.
     - Since the sum of the array `s` is positive, we explore the maximum sum crossing the entire middle part which would be `(k - 2) * s + mx_pre + mx_suf`. Since `k = 2`, multiplying by `k - 2` equals `0`, so this step does not change `ans`.

4. **Returning the Result**:

   - The function would return `ans` modulo `10^9 + 7`, which is `8` in this case, as the maximum sum sub-array is `[3, -1, 2, 3, -1, 2]` itself when `k = 2`, with the sum being `8`.

This example demonstrates that even when our given array has negative numbers, by strategically utilizing the concatenation of the array `k` times, we can maximize the sub-array sum without including the negative sub-arrays. The technique of combining prefix and suffix sums with Kadane's algorithm and handling different cases based on the total sum `s` and the count `k` leads to a comprehensive solution.

Python Solution

```python
from typing import List

class Solution:
    def kConcatenationMaxSum(self, arr: List[int], k: int) -> int:
        # Initialize variables
        total_sum = max_prefix = min_prefix = max_subarray_sum = 0

        # Calculate max subarray sum for a single array iteration
        for num in arr:
            total_sum += num
            max_prefix = max(max_prefix, total_sum)
            min_prefix = min(min_prefix, total_sum)
            max_subarray_sum = max(max_subarray_sum, total_sum - min_prefix)

        # The result after a single iteration
        result = max_subarray_sum
        mod = 10**9 + 7

        # If k is 1, return the result of a single array's max subarray sum
        if k == 1:
            return result % mod

        # Calculate the maximum suffix sum for potential use in concatenated arrays
        max_suffix = total_sum - min_prefix

        # Update result for potential double array combination
        result = max(result, max_prefix + max_suffix)

        # If the array sum is positive, calculate the max sum when array is concatenated k times
        if total_sum > 0:
            result = max(result, ((k - 2) * total_sum) + max_prefix + max_suffix)

        return result % mod  # Return the result modulo the provided modulus

```

Java Solution

```java
class Solution {

    // Computes the maximum sum of a subsequence in an array that can be achieved by
    // concatenating the array k times.
    public int kConcatenationMaxSum(int[] arr, int k) {
        long sum = 0; // Total sum of the array elements.
        long maxPrefixSum = 0; // Maximum prefix sum found so far.
        long minPrefixSum = 0; // Minimum prefix sum found so far.
        long maxSubarraySum = 0; // Maximum subarray sum found so far.

        // Iterate over the array to find the maximum subarray sum, maximum prefix,
        // and minimum prefix sums.
        for (int value : arr) {
            sum += value;
            maxPrefixSum = Math.max(maxPrefixSum, sum);
            minPrefixSum = Math.min(minPrefixSum, sum);
            maxSubarraySum = Math.max(maxSubarraySum, sum - minPrefixSum);
        }

        long answer = maxSubarraySum; // This holds the result, which is initialized to maxSubarraySum.
        final int mod = (int) 1e9 + 7; // Module to perform the answer under modulo operation.

        // If there's only one concatenation, simply return the max subarray sum modulo mod.
        if (k == 1) {
            return (int) (answer % mod);
        }

        long maxSuffixSum = sum - minPrefixSum; // Maximum suffix sum after one traversal.

        // Check if adding the entire array sum (suffix and prefix) is better.
        answer = Math.max(answer, maxPrefixSum + maxSuffixSum);

        // If the sum of the array is positive, the best option might be to take the sum k-2 times,
        // then add the maxPrefix and maxSuffix sums.
        if (sum > 0) {
            answer = Math.max(answer, (k - 2) * sum + maxPrefixSum + maxSuffixSum);
        }

        // Return the maximal sum found under modulo mod.
        return (int) (answer % mod);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int kConcatenationMaxSum(vector<int>& arr, int k) {
        long sumOfArray = 0, maxPrefixSum = 0, minPrefixSum = 0, maxSubarraySum = 0;
        const int MOD = 1e9 + 7; // Define the modulus for the answer

        // Calculate the maximum subarray sum for one array
        for (int num : arr) {
            sumOfArray += num; // Sum of elements so far
            maxPrefixSum = max(maxPrefixSum, sumOfArray); // Max sum from the start to current
            minPrefixSum = min(minPrefixSum, sumOfArray); // Min sum from the start to current
            maxSubarraySum = max(maxSubarraySum, sumOfArray - minPrefixSum); // Kadane's algorithm update
        }

        long result = maxSubarraySum; // Initialize the result with max subarray sum

        // Handle the case when the array is concatenated only once
        if (k == 1) {
            return result % MOD;
        }

        long maxSuffixSum = sumOfArray - minPrefixSum; // Sum of max suffix
        result = max(result, maxPrefixSum + maxSuffixSum); // Max of result and sum of max prefix and suffix

        // If the sum of the array is positive, we can take the whole arrays k-2 times along with maxPrefix and maxSuffix
        if (sumOfArray > 0) {
            result = max(result, maxPrefixSum + (k - 2) * sumOfArray + maxSuffixSum);
        }

        // Return the result modulo 10^9 + 7
        return result % MOD;
    }
};

```

Typescript Solution

```typescript
const MOD: number = 1e9 + 7; // Define the modulus for the answer

function kConcatenationMaxSum(arr: number[], k: number): number {
    let sumOfArray: number = 0, maxPrefixSum: number = 0, minPrefixSum: number = 0, maxSubarraySum: number = 0;

    // Calculate the maximum subarray sum for one array
    for (const num of arr) {
        sumOfArray = (sumOfArray + num) % MOD; // Keep updating the sum of elements
        maxPrefixSum = Math.max(maxPrefixSum, sumOfArray); // Max sum from start to current position
        minPrefixSum = Math.min(minPrefixSum, sumOfArray); // Min sum from start to current position
        maxSubarraySum = Math.max(maxSubarraySum, (sumOfArray - minPrefixSum + MOD) % MOD); // Kadane's algorithm update
    }

    let result: number = maxSubarraySum; // Initialize the result with max subarray sum

    // Handle the case when the array is concatenated only once
    if (k === 1) {
        return result;
    }

    let maxSuffixSum: number = (sumOfArray - minPrefixSum + MOD) % MOD; // Sum of the max suffix
    result = Math.max(result, (maxPrefixSum + maxSuffixSum) % MOD); // Max of result and the sum of max prefix and max suffix

    // If the sum of the array is positive, we include the whole arrays (k - 2) times along with maxPrefix and maxSuffix
    if (sumOfArray > 0) {
        result = Math.max(result, (maxPrefixSum + ((k - 2) * sumOfArray + maxSuffixSum) % MOD) % MOD);
    }

    // Return the result modulo 10^9 + 7
    return result;
}

```

Time and Space Complexity

The given Python code defines a method `kConcatenationMaxSum` which finds the maximum sum of a subarray in the K-concatenated array formed by repeating the given array `k` times.

Time Complexity

The function iterates once through the array `arr`, performing a constant amount of work in each iteration, including finding maximum and minimum prefixes, and computing the maximum subarray sum ending at any element. Therefore, the time complexity of iterating the array is `O(n)` where `n` is the length of `arr`.

After this iteration, there is a constant amount of work done to compute `mx_suf` and the maximum of `ans`, `mx_pre + mx_suf`, and `(k - 2) * s + mx_pre + mx_suf` if `s > 0`. These operations take `O(1)` time.

Hence, the overall time complexity of the function is `O(n + 1)`, which simplifies to `O(n)`.

Space Complexity

In terms of space, the function allocates a few variables (`s`, `mx_pre`, `mi_pre`, `mx_sub`, and `mod`), which use `O(1)` space as their number does not scale with the input size.

The inputs `arr` and `k` are used without any additional space being allocated that depends on their size (no extra arrays or data structures are created). Thus, the space used is constant.

As such, the space complexity is `O(1)`.

Combining the analysis above, the code has a linear time complexity and constant space complexity.