1918. Kth Smallest Subarray Sum

Medium Array Binary Search Sliding Window Leetcode Link

Problem Description

The problem presents us with an array of integers, `nums`, and asks us to find the `kth` smallest sum of any subarray within this array. A subarray is defined as a contiguous part of the original array—which means any sequence of elements that are next to each other without gaps—and the sum of a subarray is the total sum of its elements. So, if `nums` is `[1, 2, 3]`, then `[1, 2]`, `[2, 3]`, and `[1, 2, 3]` are all subarrays, with sums `3`, `5`, and `6`, respectively.

In simple terms, we want to look through all possible continuous sequences of numbers in the list, calculate the sum for each, and find out which is the `kth` smallest among these sums.

Intuition

The challenge lies in efficiently finding the `kth` smallest sum without having to compute and sort all possible subarray sums, which would be a very time-consuming process, especially for large arrays.

The intuition behind this kind of problem is to use Binary Search in an innovative way. Instead of searching for an explicit value in an array, we use Binary Search to find the smallest sum `s` such that there are at least `k` subarrays in `nums` whose sums are less than or equal to `s`. This is not a straightforward application of Binary Search because there is no sorted list of subarray sums to begin with.

Here's how the solution works:

First, we define a function `f(s)` which counts how many subarrays have a sum less than or equal to `s`. We use a two-pointer technique to maintain a sliding window of elements whose sum is less than or equal to `s`. As we iterate through `nums`, we keep expanding our window by adding new elements to the sum `t`, and we shrink the window from the left by removing elements whenever the total sum exceeds `s`. This way, `f(s)` returns true if there are at least `k` such subarrays.

Next, we set up our binary search range between `l` and `r` where `l` is the minimum element in `nums` (since no subarray sum can be smaller than the smallest element) and `r` is the sum of all elements in `nums` (since no subarray sum can be larger than this).

Using the `bisect_left` method from the `bisect` module, we perform our Binary Search within the range `[l, r]`. The `key` argument in `bisect_left` allows us to use the function `f` to check if a mid-value is feasible (i.e., the count of subarrays with sum less than or equal to this value is at least `k`). The method zeroes in on the smallest `s` that satisfies `f(s)`, which will be the kth smallest subarray sum in `nums`.

This approach is efficient because each iteration of Binary Search narrows down the range of possible sums by half, and for each such guess, counting the subarrays takes linear time, resulting in a total time complexity of O(n log x), where `x` is the maximum subarray sum.

Solution Approach

The `kthSmallestSubarraySum` function in the provided Python code leverages binary search, which is a classic optimization for problems involving sorted arrays or monotonic functions. In this case, the monotonic function is the number of subarrays with sums less than or equal to a given value.

The code defines a helper function `f(s)` to use within the binary search. We can describe what each part of this function does and how it integrates with the binary search mechanism:

1. **Sliding Window via Two Pointers**
   - `f(s)` uses a sliding window with two pointers, `i` and `j`, which represent the start and end of the current subarray being considered.
   - As we iterate over the array with `i`, we add each `nums[i]` to the temporary sum `t`.
   - If the sum `t` exceeds the value `s`, we subtract elements starting from `nums[j]` and increment `j` to reduce `t`. This maintains the window where the subarray sum is less than or equal to `s`.
2. **Counting Subarrays**
   - The line `cnt += i - j + 1` is crucial. It adds the number of subarrays ending at `i` for which the sum is less than or equal to `s`.
   - This works because for every new element added to the window, there are `i - j + 1` subarrays. For instance, if our window is `[nums[j], ..., nums[i]]`, then `[nums[i]]`, `[nums[i-1], nums[i]]`, up to `[nums[j], ..., nums[i]]` are all valid subarrays.
3. **Binary Search**
   - The actual search takes place over a range of potential subarray sums, from `min(nums)` to `sum(nums)`. We use the binary search algorithm to efficiently find the smallest sum that has at least `k` subarrays less than or equal to it.
   - The `bisect_left` function from the `bisect` module is used with a custom `key` function, which is the `f(s)` we defined earlier. The `key` function transforms the value we're comparing against in the binary search.
   - `bisect_left` will find the smallest value within the range `[l, r]` for which `f(s)` returns `True`, indicating it is the kth smallest sum.

The combination of these techniques results in an efficient solution where, rather than having to explicitly sort all the subarray sums, we can deduce the kth smallest sum by narrowing down our search space logarithmically with binary search and counting subarrays linearly with the two-pointer technique.

Example Walkthrough

Let's apply the solution approach to a small example. Suppose `nums = [1, 2, 3]` and we are looking for the `2nd` smallest subarray sum.

First, we initialize our helper function `f(s)` that counts subarrays with sums less than or equal to `s`. Now let's illustrate this function:

1. **Initializing Variables**:
   - We would start with `i = 0`, meaning our subarray only includes `nums[0]` for now which is `[1]`. The temporary sum `t` starts at `0`.
2. **Sliding Window via Two Pointers**:
   - Let's say we're checking for `s = 3`, we want to count how many subarrays have sum `≤ 3`.
   - At `i = 0`, we add `nums[0]` to `t` so, `t = 1`. Since `t ≤ 3`, we continue to `i = 1`.
   - At `i = 1`, `t` becomes `t + nums[1] = 3`. This sum is still `≤ 3` and we can include `[1, 2]`.
   - If we move to `i = 2`, `t` would become `6` which exceeds `3`. Therefore, we adjust `j` to remove elements from the start until `t ≤ 3` again. However, with numbers `[1, 2, 3]`, no subarray ending at index `2` has a sum `≤ 3` other than `[3]` itself.
3. **Counting Subarrays**:
   - For each `i`, the number of subarrays ending at `i` with a sum `≤ 3` is added up. We use `cnt += i - j + 1` to do this count.
   - Subarrays are `[1]`, `[2]`, `[3]`, `[1, 2]`. Thus, the count here is `4`.
4. **Binary Search**:
   - We set `l = 1` (smallest element) and `r = 6` (sum of all elements). We then search for the smallest sum `s` where `f(s)` gives us at least `k` subarrays. In this case, `k = 2`.
   - Using binary search, we check `s = (l + r) / 2` which is `3.5` initially, then we apply the helper function to count how many subarrays with sum `≤ 3.5`, which is `4`. Since we are looking for the `2nd` smallest sum, `4` subarrays mean we can try a smaller `s`.
   - We adjust our binary search boundaries to `l = 1` and `r = 3`. We check for `s = 2`. The subarrays within this constraint are `[1]` and `[2]`, which gives us count `2`. Here, `f(2)` would return `True` since it matches our `k` value.
   - The binary search will now try to see if there is a sum smaller than `2` that also satisfies the condition, but since `2` is the sum of our smallest individual element and there's no smaller sum that could give us `2` subarrays, `2` is indeed our `2nd` smallest sum.

By using this approach of binary search combined with a two-pointer technique to count subarrays, we efficiently find the `2nd` smallest subarray sum without having to explicitly calculate every possible subarray sum.

Python Solution

```python
from bisect import bisect_left
from typing import List

class Solution:
    def kthSmallestSubarraySum(self, nums: List[int], k: int) -> int:

        # Helper function to check if the count of subarrays with sum less than or equal to 'limit'
        # is at least k
        def has_k_or_more_subarrays_with_sum_at_most(limit):
            total_sum = 0  # Sum of the current subarray
            start = 0  # Start index for the current subarray
            count = 0  # Count of subarrays with sum less than or equal to 'limit'
            # Iterate over the numbers in the array
            for end, num in enumerate(nums):
                total_sum += num
                # Shrink the window from the left if the total sum exceeds the limit
                while total_sum > limit:
                    total_sum -= nums[start]
                    start += 1
                # The count is increased by the number of subarrays ending with nums[end]
                count += end - start + 1
            # Check if we have at least k subarrays
            return count >= k

        # Binary search to find the kth smallest subarray sum
        left, right = min(nums), sum(nums)
        # Perform binary search with a custom key function by using the bisect_left function
        kth_smallest_sum = left + bisect_left(range(left, right + 1), True,
                                              key=has_k_or_more_subarrays_with_sum_at_most)
        return kth_smallest_sum

# Explanation:
# 1. A binary search is applied to find the kth smallest sum within the range of the minimum
#    element (left) and the sum of all elements (right), inclusively.
# 2. The `has_k_or_more_subarrays_with_sum_at_most` function checks if the number of all
#    possible subarrays with a sum less than or equal to the passed limit is at least k.
# 3. The bisect_left function is used to find the insertion point (the kth smallest sum) for
#    which the condition in `has_k_or_more_subarrays_with_sum_at_most` returns True.
```

Java Solution

```java
class Solution {
    public int kthSmallestSubarraySum(int[] nums, int k) {
        // Initialize the left and right boundaries for the binary search.
        // Assume the smallest subarray sum is large, and find the smallest element of the array.
        int left = Integer.MAX_VALUE, right = 0;
        for (int num : nums) {
            left = Math.min(left, num);
            right += num;
        }
        // Perform binary search to find the kth smallest subarray sum.
        while (left < right) {
            int mid = left + (right - left) / 2;
            // If there are more than k subarrays with a sum <= mid, move the right pointer.
            if (countSubarraysWithSumAtMost(nums, mid) >= k) {
                right = mid;
            } else {
                // Otherwise, move the left pointer.
                left = mid + 1;
            }
        }
        // The left pointer points to the kth smallest subarray sum.
        return left;
    }

    // Helper method to count the number of subarrays with a sum at most 's'.
    private int countSubarraysWithSumAtMost(int[] nums, int s) {
        int currentSum = 0, start = 0;
        int count = 0;
        for (int end = 0; end < nums.length; ++end) {
            currentSum += nums[end];
            // If the current sum exceeds 's', shrink the window from the left.
            while (currentSum > s) {
                currentSum -= nums[start++];
            }
            // Add the number of subarrays ending at index 'end' with a sum at most 's'.
            count += end - start + 1;
        }
        return count;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the kth smallest subarray sum
    int kthSmallestSubarraySum(vector<int>& nums, int k) {
        // Initialize the left and right boundaries for binary search
        int leftBound = INT_MAX, rightBound = 0;
        // Find the smallest number in the nums array and the total sum
        for (int x : nums) {
            leftBound = min(leftBound, x);
            rightBound += x;
        }

        // Lambda function to count the subarrays with sum less than or equal to 'sum'
        auto countSubarraysLEQ = [&](int sum) {
            int totalCount = 0;
            int subarraySum = 0;
            for (int i = 0, j = 0; i < nums.size(); ++i) {
                subarraySum += nums[i];
                // Increment j to maintain the condition that subarraySum <= sum
                while (subarraySum > sum) {
                    subarraySum -= nums[j++];
                }
                // Update the total count of subarrays ending at index i
                totalCount += i - j + 1;
            }
            return totalCount;
        };

        // Binary search to find the kth smallest subarray sum
        while (leftBound < rightBound) {
            int mid = leftBound + (rightBound - leftBound) / 2;
            if (countSubarraysLEQ(mid) >= k) {
                rightBound = mid;
            } else {
                leftBound = mid + 1;
            }
        }

        // The left bound will be the kth smallest subarray sum
        return leftBound;
    }
};
```

Typescript Solution

```typescript
// Function to find the kth smallest subarray sum
function kthSmallestSubarraySum(nums: number[], k: number): number {
    // Initialize the left and right boundaries for binary search
    let leftBound = Number.MAX_SAFE_INTEGER;
    let rightBound = 0;

    // Find the smallest number in the nums array and the total sum
    nums.forEach((x) => {
        leftBound = Math.min(leftBound, x);
        rightBound += x;
    });

    // Function to count the subarrays with sum less than or equal to 'sum'
    const countSubarraysLEQ = (sum: number): number => {
        let totalCount = 0;
        let subarraySum = 0;
        for (let i = 0, j = 0; i < nums.length; ++i) {
            subarraySum += nums[i];
            // Increment j to maintain the condition that subarraySum <= sum
            while (subarraySum > sum) {
                subarraySum -= nums[j++];
            }
            // Update the total count of subarrays ending at index i
            totalCount += i - j + 1;
        }
        return totalCount;
    };

    // Binary search to find the kth smallest subarray sum
    while (leftBound < rightBound) {
        let mid = leftBound + Math.floor((rightBound - leftBound) / 2);
        if (countSubarraysLEQ(mid) >= k) {
            rightBound = mid;
        } else {
            leftBound = mid + 1;
        }
    }

    // The left bound will be the kth smallest subarray sum
    return leftBound;
}
```

Time and Space Complexity

Time Complexity

The given Python code implements a binary search combined with a two-pointer technique to find the k-th smallest subarray sum in an integer list `nums`.

1. The binary search is performed on a range from the minimum element in `nums` to the sum of all elements in `nums`. This range is `r - l + 1`, where `l` is the minimum value (i.e., `min(nums)`) and `r` is the sum of the numbers in `nums`. Since we are effectively halving the search space with each iteration, the time complexity for the binary search alone is `O(log(r - l))`.

2. Inside the binary search, we use function `f()` to count the number of subarray sums that are less than or equal to a given sum `s` by using a sliding window technique with the two pointers `i` and `j`. This is done by iterating through the array once for each value of `s`. The inner loop moves `j` appropriately but does not iterate more than `n` times across all iterations of `i` since it only subtracts the values that were previously added. Therefore, for each `s`, the function `f()` has a time complexity of `O(n)` where `n` is the length of `nums`.

Combining the binary search and the sliding window, the total time complexity is `O(n * log(r - l))`.

Space Complexity

The space complexity of the code is `O(1)`, not counting the input `nums`. This is because the code only uses a fixed number of integer variables (`l`, `r`, `s`, `t`, `j`, and `cnt`) and does not create additional data structures that grow with the size of the input.