3026. Maximum Good Subarray Sum

Medium Array Hash Table Prefix Sum

Problem Description

Given an array named `nums` with length `n` and a positive integer `k`, we define a *subarray* of `nums` as being *good* if the absolute difference between the first and last elements of the subarray is exactly `k`. In more formal terms, the subarray `nums[i...j]` is considered good if `|nums[i] - nums[j]| == k`. Our task is to identify the good subarray that has the maximum sum among all such good subarrays and return this maximum sum. If no such good subarray exists, we return `0`.

To visualize this, imagine sliding a window along the array and at each step, checking if the window results in a good subarray by comparing the difference of the first and last elements with `k`. Simultaneously we'd need to calculate the sum of elements within this window. If the window qualifies as a good subarray, we compare its sum with the best sum available until that point (the 'maximum sum' we've found). Ultimately, we aim to find the subarray that both meets the good subarray condition and has the largest sum in comparison to all other good subarrays.

Intuition

The key to solving this problem lies in two concepts: prefix sum and efficient lookups using a hash table. The insight is that in order to quickly assess if a subarray ending at position `i` is good, we can consider the current element `nums[i]` and look for previously encountered elements that would satisfy the criteria for a good subarray when paired with `nums[i]`. This means looking for values `nums[i] - k` and `nums[i] + k` among the values we've seen.

Firstly, prefix sum comes into play because it helps us calculate the sum of any subarray efficiently. By keeping a running total as we work through the array, we can find the sum of any subarray in `O(1)` time using subtraction.

Secondly, a hash table is used to achieve `O(1)` lookup time for previously encountered values. This is critical because it helps us avoid checking every possible subarray one by one, which would result in `O(n^2)` complexity. Instead, we maintain a hash table where each entry is a pair of the form `value, prefix_sum`. The `value` is an element from the array, and `prefix_sum` is the sum of all elements up to but not including `value`. As we iterate through the array, the hash table is updated with the smallest prefix sum encountered for each value.

In practice, if `nums[i] - k` is in the hash table, this means there exists a subarray ending at `i` that could potentially be a good subarray. By referencing the stored prefix sum for `nums[i] - k`, we can quickly calculate the sum of this subarray. The same applies if `nums[i] + k` is found in the hash table. We then update the answer with the larger of the current answer and this new sum, ensuring we always hold the maximum sum encountered so far.

At the end of the iteration, if the answer is still the placeholder for the smallest number (`-infinity` in programming contexts), we know no good subarray was found, and we return `0`. Otherwise, we return the answer which represents the maximum sum among all good subarrays identified.

Solution Approach

The implementation takes advantage of data structures and algorithms such as prefix sums, hash tables, and the iteration of array elements.

Starting off, a hash table `p` is used to store the smallest prefix sum encountered so far for each value in the array `nums`. The prefix sum `s` represents the sum of numbers from the start of the array up to the current index. This allows for efficient computation of subarray sums. A variable `ans` is initialized to negative infinity to keep track of the maximum subarray sum.

As the array is iterated over with the variable `i` tracking the index and `x` being the value of `nums[i]`, the current prefix sum `s` is updated by adding `x`. The search for a good subarray involves checking if `x - k` or `x + k` exists in the hash table. If it does, the current subarray is potentially a good subarray, and the sum of this subarray can be calculated using the current prefix sum `s` minus the prefix sum stored for `x - k` or `x + k` in the hash table. If the calculated sum is greater than the current answer `ans`, it is updated with this new sum.

Moreover, for each element, if the next element `nums[i + 1]` is not present in the table, or if it exists but the associated prefix sum is greater than `s`, the table is updated with `nums[i + 1]: s`. This ensures that the hash table always contains the smallest prefix sum for each value, which is essential for calculating the maximum sum of a good subarray accurately.

Finally, if after iteration the answer `ans` remains negative infinity, this indicates that no good subarray was found; in this case `0` is returned. Otherwise, the maximum subarray sum that has been calculated by checking all the good subarrays is returned.

This solution effectively reduces the problem to a runtime of `O(n)` by avoiding iterative comparison of all possible subarrays, taking advantage of the properties of prefix sums for rapid calculation, and hash table lookups for constant-time access to the prefix sums needed to check for good subarrays and calculate their sums.

Example Walkthrough

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

Imagine we have an array `nums = [1, 5, 3, 2, 5]` and `k = 2`. We need to find the maximum sum among all the good subarrays where a good subarray has an absolute difference of `k` between its first and last elements.

We begin by initializing an empty hash table `p`, a variable `s` to keep track of the current prefix sum (starting with `0`), and a variable `ans` with the value negative infinity to store the maximum sum found.

1. **First iteration (i=0):**
   `x = nums[0] = 1`, `s = 0 + x = 1`.
   `p` doesn't contain `x - k = -1` or `x + k = 3`, so we do not update `ans`.
   Update `p` with `nums[i + 1] = 5` and `s = 1`.
2. **Second iteration (i=1):**
   `x = nums[1] = 5`, `s = 1 + x = 6`.
   `p` contains `x - k = 3` from the future iteration but we don't use it yet.
   Update `p` with `nums[i + 1] = 3` (next element) and `s = 6`.
3. **Third iteration (i=2):**
   `x = nums[2] = 3`, `s = 6 + x = 9`.
   `p` contains `x - k = 1`, the prefix sum for `1` is `1` (from iteration i=0).
   So, we find a good subarray `(1, 3)` with a sum of `9 - 1 = 8`. Update `ans` to `8`.
   We also check if `x + k = 5`, though `p` contains it, it refers to a future value we'll get to.
4. **Fourth iteration (i=3):**
   `x = nums[3] = 2`, `s = 9 + x = 11`.
   `p` contains `x + k = 4`, but no such value has been encountered yet, so we don't update `ans`.
   Then `p` is not updated this time, because we already stored `5` with a smaller prefix sum `6`.
5. **Fifth iteration (i=4):**
   `x = nums[4] = 5`, `s = 11 + x = 16`.
   Again check if `p` contains `x - k = 3` (from i=2) and it does.
   We find another good subarray `(3, 5)` with a sum of `16 - 9 = 7`, but `7 < 8`, so `ans` remains `8`.

After finishing the iteration, our answer `ans = 8` is the maximum sum of all good subarrays. Thus, we return `8` as the result since `ans` did not remain negative infinity.

This walk-through exemplifies how we compute the maximum sum of a good subarray in linear time by utilizing prefix sums and hash table lookups.

Solution Implementation

Python

```python
from typing import List

# Class to define the solution
class Solution:
    def maximum_subarray_sum(self, nums: List[int], k: int) -> int:
        # Initialize the maximum answer to negative infinity
        max_sum = float('-inf')

        # A dictionary to keep track of the prefix sums encountered
        prefix_sums = {nums[0]: 0}

        # Initialize the sum variable to 0
        current_sum = 0

        # Loop through the list of numbers
        for index, num in enumerate(nums):
            current_sum += num  # Update the running sum with the current number

            # Check if the difference (current_sum - k) exists as a prefix sum
            if current_sum - k in prefix_sums:
                # Update the maximum subarray sum if a larger sum is found
                max_sum = max(max_sum, current_sum - prefix_sums[current_sum - k])

            # Check if the difference (current_sum + k) exists as a prefix sum
            if current_sum + k in prefix_sums:
                # Update the maximum subarray sum if a larger sum is found
                max_sum = max(max_sum, current_sum - prefix_sums[current_sum + k])

            # Prepare for the next iteration; check the next element if it's not the last
            if index + 1 < len(nums):
                next_value = nums[index + 1]
                # Save the current sum in the prefix sums dictionary if the
                # next number has not been seen or a smaller sum has been seen before it
                if next_value not in prefix_sums or prefix_sums[next_value] > current_sum:
                    prefix_sums[next_value] = current_sum

        # Return the maximum sum if found; otherwise, return 0
        return max_sum if max_sum != float('-inf') else 0
```

Java

```java
class Solution {
    public long maximumSubarraySum(int[] nums, int k) {
        // A map to store the prefix sum along with the corresponding number just after it.
        Map<Integer, Long> prefixSumMap = new HashMap<>();
        // Initialize the first value of the map.
        prefixSumMap.put(nums[0], 0L);

        // Initialize a variable to keep track of the sum.
        long currentSum = 0;
        // Get the length of the input array.
        int arrayLength = nums.length;
        // Initialize the answer with the smallest possible value.
        long maxSubarraySum = Long.MIN_VALUE;

        // Iterate through the elements of the array.
        for (int i = 0; i < arrayLength; ++i) {
            // Add the current value to the sum.
            currentSum += nums[i];
            // Check if there's a prefix sum that is current sum minus k
            if (prefixSums.containsKey(nums[i] - k)) {
                maxSubarraySum = Math.max(maxSubarraySum, currentSum - prefixSums.get(nums[i] - k));
            }
            // Check if there's a prefix sum that is the current sum plus k
            if (prefixSums.containsKey(nums[i] + k)) {
                maxSubarraySum = Math.max(maxSubarraySum, currentSum - prefixSums.get(nums[i] + k));
            }
            // If we're not on the last element and the upcoming element is either
            // not in the map or has a larger sum than the current, update the map.
            if (i + 1 < arrayLength && (!prefixSums.containsKey(nums[i + 1]) || prefixSums.get(nums[i + 1]) > currentSum)) {
                prefixSums.put(nums[i + 1], currentSum);
            }
        }
        // If the answer hasn't been updated, return 0; otherwise, return the calculated max sum.
        return maxSubarraySum == Long.MIN_VALUE ? 0 : maxSubarraySum;
    }
}
```

C++

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

class Solution {
public:
    long long maximumSubarraySum(std::vector<int>& nums, int k) {
        // A map to store the prefix sum along with its corresponding index.
        std::unordered_map<int, long long> prefixSumMap;
        // Initialize with the first element
        long long currentSum = 0; // Current running sum
        const int n = nums.size(); // Size of the input array
        long long maxSum = LLONG_MIN; // Initialize max sum as minimum possible long long value

        for (int i = 0; i < n; ++i) {
            currentSum += nums[i]; // Update the running sum

            // Check if there is a prefix sum (currentSum - k), and update maxSum if needed
            auto it = prefixSumMap.find(currentSum - k);
            if (it != prefixSumMap.end()) {
                maxSum = std::max(maxSum, currentSum - it->second);
            }

            // Similarly, check for (currentSum + k)
            it = prefixSumMap.find(currentSum + k);
            if (it != prefixSumMap.end()) {
                maxSum = std::max(maxSum, currentSum - it->second);
            }

            // Break the loop if we've reached the end of the array
            if (i + 1 >= n) {
                break;
            }

            // Update the prefix sum map for the next element if needed
            // If a prefix sum for nums[i+1] does not exist or there's a greater sum found, update it
            it = prefixSumMap.find(nums[i + 1]);
            if (it == prefixSumMap.end() || it->second > currentSum) {
                prefixSumMap[nums[i + 1]] = currentSum;
            }
        }

        // If maxSum was not updated, return 0; otherwise, return the updated maxSum
        return maxSum == LLONG_MIN ? 0 : maxSum;
    }
};
```

TypeScript

```typescript
// Function to calculate the maximum subarray sum with sum at most `k`
function maximumSubarraySum(nums: number[], k: number): number {
    // Initialize a map to keep track of prefix sums and their indices
    const prefixSumIndices: Map<number, number> = new Map();

    // Set the prefix sum for the first element
    prefixSumIndices.set(0, -1);

    // Initialize the answer with the smallest possible number
    let maxSum: number = -Infinity;

    // Initialize summation variable to keep track of the running sum
    let sum: number = 0;

    // Loop through the array
    for (let i = 0; i < nums.length; ++i) {
        // Add current number to the running sum
        sum += nums[i];

        // Check if there's a prefix sum such that current sum - previous sum equals to k
        if (prefixSumIndices.has(sum - k)) {
            // Update maxSum with the largest sum found so far
            maxSum = Math.max(maxSum, sum - prefixSumIndices.get(sum - k));
        }

        // If this sum has not been seen before or the previous index is greater,
        // update the map with the current index for this sum.
        // This helps in finding the smallest index (leftmost) for this prefixSum
        if (!prefixSumIndices.has(sum) || prefixSumIndices.get(sum) > i) {
            prefixSumIndices.set(sum, i);
        }
    }

    // If maxSum was not updated, there's no valid subarray sum; return 0. Otherwise, return maxSum.
    return maxSum === -Infinity ? 0 : maxSum;
}
```

```python
from typing import List

# Class to define the solution
class Solution:
    def maximum_subarray_sum(self, nums: List[int], k: int) -> int:
        # Initialize the maximum answer to negative infinity
        max_sum = float('-inf')

        # A dictionary to keep track of the prefix sums encountered
        prefix_sums = {nums[0]: 0}

        # Initialize the sum variable to 0
        current_sum = 0

        # Loop through the list of numbers
        for index, num in enumerate(nums):
            current_sum += num  # Update the running sum with the current number

            # Check if the difference (current_sum - k) exists as a prefix sum
            if current_sum - k in prefix_sums:
                # Update the maximum subarray sum if a larger sum is found
                max_sum = max(max_sum, current_sum - prefix_sums[current_sum - k])

            # Check if the difference (current_sum + k) exists as a prefix sum
            if current_sum + k in prefix_sums:
                # Update the maximum subarray sum if a larger sum is found
                max_sum = max(max_sum, current_sum - prefix_sums[current_sum + k])

            # Prepare for the next iteration; check the next element if it's not the last
            if index + 1 < len(nums):
                next_value = nums[index + 1]
                # Save the current sum in the prefix sums dictionary if the
                # next number has not been seen or a smaller sum has been seen before it
                if next_value not in prefix_sums or prefix_sums[next_value] > current_sum:
                    prefix_sums[next_value] = current_sum

        # Return the maximum sum if found; otherwise, return 0
        return max_sum if max_sum != float('-inf') else 0
```

Time and Space Complexity

Time Complexity

The function iterates over all the elements in the given `nums` list with a single loop, which results in a time complexity of `O(n)` where `n` is the length of the array `nums`. Despite the updates and lookups in the dictionary `p`, these operations are done in constant time on average, so they do not change the overall linear time complexity.

Space Complexity

The space complexity of the code is determined by the space needed to store the dictionary `p` which can potentially contain an entry for each unique number that appears as a running sum at each index of `nums`. Therefore, the space complexity is `O(n)` where `n` is the length of the array `nums`.