644. Maximum Average Subarray II

Hard   Array   Binary Search   Prefix Sum   Leetcode Link

Problem Description

You're given an integer array `nums` which contains `n` elements, and an integer `k`. The task is to find a contiguous subarray whose length is at least `k` which has the maximum possible average value. It's important to note that the answer needs to be accurate but is allowed a small margin for error. Specifically, any answer within `10^-5` (0.00001) of the actual answer will be considered correct.

A subarray, in this context, is a sequence of elements from the array `nums` that lies next to each other, without any gaps. For example, if `nums` is `[1, 3, 5, 7, 9]` and `k` is `2`, we have to find a contiguous sequence of length `2` or more that gives the highest average. The returned value should be the average of that subarray.

The essence of the problem is to find the optimal section of the list - it can be from `k` to `n` elements long - and calculate the average of the numbers within this range. The difficulty lies in doing this efficiently, as brute force methods that check all possible subarrays will be too slow when the array `nums` is large.

Intuition

The intuition behind the solution involves binary search and prefix sums. Instead of directly searching for the maximum average, we flip the problem on its head: for a given average `v`, determine if there's a subarray with an average greater than or equal to `v`. Then we use binary search on `v`, narrowing down the range until we find the maximum average to satisfy the condition.

Here's how we arrive at the solution approach:

- Initially, we know that the maximum average lies between the minimum and maximum values in our array `nums`. These are our initial lower (`l`) and upper (`r`) bounds for binary search.
- We define a check function to verify if a subarray exists that has an average at least as large as `v`. This involves some smart tricks:
  - Calculate the prefix sum of the array elements minus `v` times `k`. This transforms the problem into finding a subarray sum that is non-negative.
  - Use a running sum (`s`) to keep track of the current sum of the last `k` elements, and a minimum sum (`mi`) to keep the smallest sum of any `k` elements we've seen. We update these as we move the running sum forward through `nums`.
  - If, at any point, our running sum minus the minimum sum we've seen so far (`mi`) is non-negative, it means there is a subarray with an average of at least `v`.
- The binary search operates by repeatedly checking the midpoint of our `l` and `r` range against the `check` function. If the function returns `true`, we know an average of at least `mid` is possible, so we move the lower bound `l` to `mid`. Otherwise, we move the upper bound `r` to `mid`.
- We continue the binary search until our range is less than `10^-5`, at which point we can return either `l` or `r` as our result, since they will be sufficiently close to the maximum average.

This solution approach is much more efficient than checking every possible subarray, as it takes advantage of the properties of averages and the structure of contiguous subarrays to reduce the problem to a binary search over possible average values.

Solution Approach

The solution uses a combination of binary search and prefix sums to efficiently calculate the maximum average subarray. Here is a step-by-step explanation of how the implementation works:

1. **Binary Search Initialization**: We start by establishing our search range for the possible maximum average, with `l` being the minimum value in `nums` and `r` being the maximum value. These represent the absolute possible bounds for the average.

2. **Helper Function - `check()`**: This function is crucial; it checks if there exists a contiguous subarray with an average value of at least `v`. Here's how it operates:

   - Sum up the first `k` elements of `nums`, then subtract `k*v` from this sum to get `s`. This operation shifts the problem from finding a maximum average subarray to finding a non-negative sum subarray.
   - Initialize two variables, `t` and `mi`, that represent the total sum and minimum sum encountered so far, respectively. Initially, `t` is `0` and `mi` is the smallest possible value.
   - Iterate over the array starting from the `k-th` element. For each element at index `i`, add `nums[i] - v` to `s` and `nums[i - k] - v` to `t`, effectively moving the `k-length` window one step forward.
   - Then, update `mi` to be the minimum of itself and `t`, which is the sum of the first `i-k` elements of the array minus `k*v`. This tracks the smallest sum we have seen up to that point, before considering the current `k-length` window.
   - If `s - mi >= 0`, the function returns `True`, indicating that it's possible to have an average of at least `v`. This is because we found a contiguous subarray where the sum of the numbers minus `k*v` is at least zero, which means the average of that subarray is at least `v`.
3. **Binary Search Loop**: The binary search loop keeps iterating while `r - l` is greater than a very small epsilon value (`1e-5`). Within this loop:

   - Calculate the midpoint `mid` as `(l + r) / 2`.
   - Use the `check()` function with this midpoint value. If `check(mid)` returns `True`, we know that it's possible to have a subarray with an average greater than or equal to `mid`, so we move the lower bound up to `mid`. If the result is `False`, we make the upper bound `r` equal to `mid`. This continuously narrows the range of possible averages.
4. Getting the Result: Once the binary search loop exits, the value of `l` (or `r`, as they are very close to each other) is a sufficiently accurate representation of the maximum average, within an error margin of `10^-5`.

The solution elegantly combines binary search, which is a classic divide and conquer algorithm, with a manipulation of sums to reframe the problem into one that is far more computationally efficient. The running time complexity is `O(n * log(max(nums) - min(nums)))`, as we have to run our check once for each iteration of binary search, and the range of the binary search is dictated by the values within `nums`. The linear pass in the check function, where we iterate over the array updates running sums, takes `O(n)` time. The constant factor has been reduced significantly as compared to a brute force method that might consider every possible subarray individually, moving the solution from possibly quadratic time complexity to linearithmic.

Example Walkthrough

Let's illustrate the solution approach with a small example. Assume we have an array `nums = [1, 12, 2, 6, 7]` and `k = 2`.

Step 1: Binary Search Initialization

- The minimum value in `nums` is `1`, and the maximum value is `12`. So we initialize our binary search range with `l = 1` and `r = 12`.

Step 2: Helper Function - `check()`

- For illustration purposes, let's assume the binary search is at a point where it's testing `v = 6.5`. We would initialize `s` as the sum of the first `k` elements minus `k*v`, which is `1 + 12 - 2*6.5 = 0`. Initialize `t` and `mi` as well. Here, `t` will start at `0` and `mi` will be `0`, as we have not seen any sum yet.

Step 3: Iterate and Check

- Start iterating from the k-th element of `nums` (index `1`): For each element `nums[i]`, where `i >= k`, do the following (example for `i = 2`):
  - Add `nums[i] - v` to `s` (running sum), so `s += 2 - 6.5`, and we get `s = -4.5`.
  - Add `nums[i - k] - v` to `t`. For `i = 2`, it's `1 - 6.5` (since `k=2`), so `t = -5.5`.
  - Update `mi` to be the minimum of `mi` and `t`. As this is the first update and `t` is less than our initial `mi`, we now set `mi = t = -5.5`.
- Continuing this process, when `i = 4` (`nums[i] = 7`):
  - `s += 7 - 6.5`, now `s = 0.5 - 4.5 = -4`.
  - `t` has been updated as we moved along, let's say it is now `-1.5`.
  - `mi` was updated along the way and it's `min(mi, t)`; we'll consider it `-5.5`.
- When we check `s - mi` for each `i`, we are looking for it to be `>= 0`. In this case, at `i = 4`, `s - mi` is `-4 - (-5.5) = 1.5`, which is `>= 0`. So, a subarray with an average at least as high as `6.5` exists.

Step 4: Binary Search Loop

- We'd repeat the above process for different values of `v`, adjusting `l` and `r` with each iteration. If `check(6.5)` returned `True`, we'd set `l = 6.5`. If it returned `False`, we'd set `r = 6.5`.
- Continue until `r - l < 10^-5`.

Step 5: Result

- Once `l` and `r` are within `10^-5` of each other, the binary search terminates, and either value gives us the maximum possible average subarray to within the accepted error margin.

In this example, let's say the binary search concluded with `l = 6.49995` and `r = 6.50000`, with an error margin below `10^-5`, either `l` or `r` would serve as the result for the maximum average, which is approximately `6.5`.

Python Solution

```python
from typing import List

class Solution:
    def findMaxAverage(self, nums: List[int], k: int) -> float:
        # Helper function to check if average value v can be the result
        def can_be_average(v: float) -> bool:
            # Initialize the sum of the first k elements adjusted by subtracting v
            current_sum = sum(nums[:k]) - k * v
            # If the current average is already >= 0, return True
            if current_sum >= 0:
                return True
            prev_sum = min_sum = 0
            # Iterate over the rest of the elements
            for i in range(k, len(nums)):
                # Update the sum for the new window by including the new element and excluding the old
                current_sum += nums[i] - v
                # Update the sum for the previous window
                prev_sum += nums[i - k] - v
                # Keep track of the minimum sum encountered so far
                min_sum = min(min_sum, prev_sum)
                # If the current window sum is greater than any seen before, return True
                if current_sum >= min_sum:
                    return True
            return False

        # Defining precision for the binary search result
        precision = 1e-5
        # Setting the lower and upper bounds of binary search to the min and max of nums
        left, right = min(nums), max(nums)

        # Binary search routine to find maximum average
        while right - left >= precision:
            mid = (left + right) / 2
            # If the current mid value can be an average, update the lower bound
            if can_be_average(mid):
                left = mid
            # Otherwise, update the upper bound
            else:
                right = mid
        # The result is the left bound after the binary search loop ends
        return left

```

Java Solution

```java
class Solution {
    // Calculates the maximum average of any subarray of length k
    public double findMaxAverage(int[] nums, int k) {
        double precision = 1e-5;
        double lowerBound = 1e10;
        double upperBound = -1e10;

        // Finding the lowest and highest numbers in the input array nums
        for (int num : nums) {
            lowerBound = Math.min(lowerBound, num);
            upperBound = Math.max(upperBound, num);
        }

        // Binary search to find the maximum average subarray within the precision range
        while (upperBound - lowerBound >= precision) {
            double mid = (lowerBound + upperBound) / 2;
            if (canFindLargerAverage(nums, k, mid)) {
                lowerBound = mid;
            } else {
                upperBound = mid;
            }
        }
        // After the loop, lowerBound is the maximum average within specified precision
        return lowerBound;
    }

    // Helper method to check if we can find an average larger than 'averageValue'
    private boolean canFindLargerAverage(int[] nums, int k, double averageValue) {
        double sum = 0;
        // Calculate the initial sum of the first k elements reduced by the averageValue
        for (int i = 0; i < k; ++i) {
            sum += nums[i] - averageValue;
        }

        if (sum >= 0) {
            return true;
        }

        double prevSum = 0; // Sum of the values from the start to i - k
        double minPrevSum = 0; // Minimum sum encountered so far for prevSum

        // Iterate through the rest of the elements
        for (int i = k; i < nums.length; ++i) {
            sum += nums[i] - averageValue; // Increment current sum
            prevSum += nums[i - k] - averageValue; // Increment previous sum
            minPrevSum = Math.min(minPrevSum, prevSum); // Update the minimum of previous sums

            // If the current sum - the smallest sum of prevSum is non-negative,
            // then there exists a subarray with an average ≥ averageValue
            if (sum >= minPrevSum) {
                return true;
            }
        }
        return false;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to find the maximum average subarray of size 'k' in the given vector 'nums'
    double findMaxAverage(vector<int>& nums, int k) {
        // Epsilon value to control the precision of our answer
        double epsilon = 1e-5;

        // Initialize left and right boundaries for binary search
        double left = *min_element(nums.begin(), nums.end());
        double right = *max_element(nums.begin(), nums.end());

        // Lambda function to check if there's a subarray with average greater than 'value'
        auto canFindLargerAverage = [&](double value) {
            // Initial sum of the first 'k' elements with 'value' subtracted from each one
            double sum = 0;
            for (int i = 0; i < k; ++i) {
                sum += nums[i] - value;
            }
            // If the sum is already non-negative, we can return true
            if (sum >= 0) {
                return true;
            }
            double sum_excluding_first_i_elements = 0, min_sum_excluding_first_i_elements = 0;
            // Check the rest of the array to find if any subarray can have an average larger than 'value'
            for (int i = k; i < nums.size(); ++i) {
                sum += nums[i] - value; // Add the next element to 'sum'
                sum_excluding_first_i_elements += nums[i - k] - value; // Update the sum excluding the first i elements
                min_sum_excluding_first_i_elements = min(min_sum_excluding_first_i_elements, sum_excluding_first_i_elements); // Up

                // If the sum is greater than or equal to the minimum sum, we can return true
                if (sum >= min_sum_excluding_first_i_elements) {
                    return true;
                }
            }
            // If no subarray has an average greater than 'value', return false
            return false;
        };

        // Perform a binary search to find the maximum average within an epsilon range
        while (right - left >= epsilon) {
            // Calculate mid-point of the range
            double mid = (left + right) / 2;
            // Use the lambda function to check if we can find a larger average
            if (canFindLargerAverage(mid)) {
                left = mid;
            } else {
                right = mid;
            }
        }
        // Return the maximum average which is on the 'left' due to the way we updated it in binary search
        return left;
    }
};

```

Typescript Solution

```typescript
function findMaxAverage(nums: number[], k: number): number {
    const epsilon = 1e-5; // This is the precision for the floating-point comparison.
    let left = Math.min(...nums); // Initialize 'left' to the smallest element in the array.
    let right = Math.max(...nums); // Initialize 'right' to the largest element in the array.

    // 'check' function checks if there exists a subarray of length 'k' with average value greater than or equal to 'targetAverage'
    const check = (targetAverage: number): boolean => {
        // Initial sum of the first 'k' elements adjusted by 'targetAverage'.
        let sum = nums.slice(0, k).reduce((a, b) => a + b) - targetAverage * k;
        if (sum >= 0) {
            return true; // The average of the first 'k' elements is already greater than 'targetAverage'.
        }

        // 'totalSum' stores the sum of previous elements adjusted by 'targetAverage' in the sliding window.
        // 'minPrevSum' stores the minimum sum encountered in the sliding window.
        let totalSum = 0;
        let minPrevSum = 0;

        for (let i = k; i < nums.length; ++i) {
            sum += nums[i] - targetAverage; // Include the next element into 'sum' while shifting the window.
            totalSum += nums[i - k] - targetAverage; // Include the element that is exiting the window into 'totalSum'.

            // Update 'minPrevSum' to the minimum sum we have seen so far.
            minPrevSum = Math.min(minPrevSum, totalSum);

            // If the current sum adjusted by the minimum previous sum we've seen is non-negative,
            // it means there exists a subarray with an average at least 'targetAverage'.
            if (sum >= minPrevSum) {
                return true;
            }
        }
        return false;
    };

    // Use binary search to find the maximum average to a precision of 'epsilon'.
    while (right - left >= epsilon) {
        const mid = (left + right) / 2; // Calculate the mid point between left and right.
        if (check(mid)) {
            left = mid; // If there exists a subarray with average greater than 'mid', move 'left' to 'mid'.
        } else {
            right = mid; // Otherwise, move 'right' to 'mid'.
        }
    }
    // 'left' will contain the maximum average subarray value within the desired precision.
    return left;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the `findMaxAverage` function consists of two main parts: the binary search over the range of values and the sliding window check within the `check` function.

- The binary search runs in `O(log((max(nums) - min(nums)) / eps))` time because it narrows down the range `[min(nums), max(nums)]` by half in each iteration until the range is smaller than `eps (= 1e-5)`.

- Within each iteration of the binary search, the `check` function is called once. The `check` function uses a sliding window approach that takes `O(n)` time, where `n` is the length of the `nums` list.

Combining these two parts, the overall time complexity of the function is `O(n * log((max(nums) - min(nums)) / eps))`.

Space Complexity

The space complexity of the function is `O(1)`.

- There are only a constant number of extra variables used (`s`, `t`, `mi`, `eps`, `l`, `r`, `mid`), and no additional space that scales with the input size is required.

Together, the function achieves linearithmic time complexity and constant space complexity.