2233. Maximum Product After K Increments

Medium Greedy Array Heap (Priority Queue) Leetcode Link

Problem Description

The problem provides us with a list of non-negative integers named `nums` and an integer `k`. The task is to increase any element of `nums` by `1` during each operation, and you can perform this operation at most `k` times. The goal is to maximize the product of all the elements in the array `nums`. After finding the maximum product, we need to return the result modulo `10^9 + 7` because the maximum product could be a very large number. The core of the problem lies in strategically choosing which numbers to increment to maximize the final product.

Intuition

In order to maximize the product of the numbers, we want to even out the values as much as possible. This is because the product of numbers that are closer together is generally higher than the product of the same numbers that are not evenly spread. For example, the product of [4, 4, 4] is 64, while the product of [2, 4, 6] is only 48, even though both sets have the same sum of 12.

With this in mind, we arrive at the solution approach:

1. Use a min-heap (a data structure that allows us to always extract the smallest number efficiently) to manage the numbers. This ensures that we can always increment the smallest number available, which helps in balancing them as evenly as possible.
2. Perform `k` operations of incrementing the smallest value in the min-heap. After each increment, the updated number is pushed back into the heap, maintaining the heap order.
3. Once all the operations are done, calculate the product of all the elements in the heap. We have to keep in mind the modulo `10^9 + 7` during this step by taking the modulo after each multiplication to prevent integer overflow.

By following this approach, we effectively distribute the increments in a way that pushes the product to its maximum possible value before taking the modulo.

Solution Approach

The solution to this problem is implemented in Python and makes use of the heap data structure, which allows us to easily and efficiently access and increment the smallest element in the `nums` list. Here is the step-by-step explanation of the implementation:

1. **Heapify the List**: The first step is to convert the list `nums` into a min-heap using the `heapify` function from Python's `heapq` module. In a min-heap, the smallest element is always at the root, which allows us to apply our increments as effectively as possible.

```
heapify(nums)
```

2. **Perform Increment Operations**: Next, we perform `k` increments. For each operation, we extract the smallest element from the heap using `heappop(nums)`, increment it by `1`, and then push it back into the heap using `heappush(nums, heappop(nums) + 1)`. This ensures that our heap remains in a consistent state, with the smallest element always on top, ready to be incremented in the next operation.

```
for _ in range(k):
    heappush(nums, heappop(nums) + 1)
```

3. **Calculate the Product**: Once all increments are done, we iterate through all elements in the heap to calculate the product. Because we're looking for the product modulo `10^9 + 7`, we take the modulo after each multiplication to prevent integer overflow. We initialize the `ans` variable to `1` and iterate through each value `v` in `nums`, applying the modulo operation as we multiply:

```
ans = 1
mod = 10**9 + 7
for v in nums:
    ans = (ans * v) % mod
```

4. **Return the Result**: Finally, we return the calculated `ans` as the result.

The key algorithm used here is the heap (priority queue), which allows us to prioritize incrementing the smallest numbers. The pattern is simple yet effective: by giving priority to the smallest elements for incrementation, we use the operations to balance the numbers, aiming for a more uniform distribution which leads to a maximized product. The implementation is careful to take the product modulo `10^9 + 7` at each step, ensuring that the final result is within the correct range and doesn't overflow.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Imagine we have `nums = [1, 2, 3]` and `k = 3`.

1. **Heapify the List**: We convert `nums` into a min-heap.

   Starting array: [1, 2, 3]

   After heapify: [1, 2, 3] (Note: Since the array is already a valid min-heap, there's no visible change)

2. **Perform Increment Operations**: We are allowed 3 increments.

   1st increment: Smallest is 1. We pop 1 out, increment to 2, and push back to heap.

   Heap after 1st increment: [2, 2, 3]

   2nd increment: Now we have two 2s at the top. We pop one out, increment to 3, and push back.

   Heap after 2nd increment: [2, 3, 3]

   3rd increment: One more increment to the remaining 2.

   Heap after 3rd increment: [3, 3, 3]

   We've used our 3 increments to even out the numbers, as predicted by our intuition.

3. **Calculate the Product**: Now we calculate the product of the heap's elements modulo `10^9 + 7`.

   Starting with `ans = 1`,

   For 3: `ans = (1 * 3) % 1000000007 = 3`

   For next 3: `ans = (3 * 3) % 1000000007 = 9`

   For last 3: `ans = (9 * 3) % 1000000007 = 27`

   So the final product modulo `10^9 + 7` is 27.

4. **Return the Result**: The result, 27, is returned as the final output.

Python Solution

```python
from heapq import heapify, heappop, heappush

class Solution:
    def maximumProduct(self, nums, k):
        # Convert the list nums into a min-heap in-place
        heapify(nums)

        # Increment the smallest element in the heap k times
        for _ in range(k):
            smallest = heappop(nums)  # Pop the smallest element
            heappush(nums, smallest + 1)  # Increment the smallest element and push back onto heap

        # Calculate the product of all elements mod 10^9 + 7
        product = 1
        modulo = 10**9 + 7
        for num in nums:
            product = (product * num) % modulo

        return product

```

Java Solution

```java
class Solution {

    // Define the MOD constant to use for avoiding integer overflow issues.
    private static final int MOD = (int) 1e9 + 7;

    // Function to calculate the maximum product of array elements after
    // incrementing any element 'k' times.
    public int maximumProduct(int[] nums, int k) {
        // Initialize a min-heap (PriorityQueue) to store the elements.
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        // Add all the elements to the min-heap.
        for (int num : nums) {
            minHeap.offer(num);
        }

        // Increment the smallest element in the heap 'k' times.
        while (k-- > 0) {
            // Retrieve and remove the smallest element from the heap,
            // increment it and add it back to the heap.
            int incrementedValue = minHeap.poll() + 1;
            minHeap.offer(incrementedValue);
        }

        // Initialize the answer as a long to prevent overflow during the computation.
        long answer = 1;

        // Calculate the product of all elements now in the heap.
        while (!minHeap.isEmpty()) {
            // Take each element from the heap, multiply it with the current answer
            // and compute the modulus.
            answer = (answer * minHeap.poll()) % MOD;
        }

        // Return the final product modulo MOD as an integer.
        return (int) answer;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm>
#include <functional> // for std::greater

class Solution {
public:
    int maximumProduct(std::vector<int>& nums, int k) {
        const int mod = 1e9 + 7; // Modulo value for the final result

        // Create a min-heap from the given vector nums
        std::make_heap(nums.begin(), nums.end(), std::greater<int>());

        // Iteratively increment the smallest element and then reheapify
        while (k-- > 0) {
            std::pop_heap(nums.begin(), nums.end(), std::greater<int>());
            ++nums.back(); // Increment the smallest element
            std::push_heap(nums.begin(), nums.end(), std::greater<int>()); // Reheapify
        }

        // Compute the product of all elements modulo mod
        long long product = 1; // Use long long to avoid integer overflow
        for (int v : nums) {
            product = (product * v) % mod; // Update the product with each element
        }

        return static_cast<int>(product); // Cast to int to match return type
    }
};

```

Typescript Solution

```typescript
import { MinPriorityQueue } from '@datastructures-js/priority-queue';

/**
 * Calculates the maximum product of an array after incrementing any element "k" times.
 * @param {number[]} nums An array of numbers.
 * @param {number} k The number of increments to perform.
 * @returns {number} The maximum product modulo 10^9 + 7.
 */
function maximumProduct(nums: number[], k: number): number {
    const n: number = nums.length;
    let priorityQueue: MinPriorityQueue<number> = new MinPriorityQueue<number>();

    // Initialize the priority queue with all elements from the nums array
    for (let i = 0; i < n; i++) {
        priorityQueue.enqueue(nums[i]);
    }

    // Increment the smallest element in the queue k times
    for (let i = 0; i < k; i++) {
        let minElement: number = priorityQueue.dequeue().element;
        priorityQueue.enqueue(minElement + 1);
    }

    let product: number = 1;
    const MODULO: number = 10 ** 9 + 7;

    // Calculate the product of all elements in the queue
    for (let i = 0; i < n; i++) {
        product = (product * priorityQueue.dequeue().element) % MODULO;
    }

    return product;
}

// Note: This code assumes that MinPriorityQueue<number> is imported correctly and available.

```

Time and Space Complexity

Time Complexity

The time complexity of this code is determined by the following parts:

1. **Heapify Operation**: Converting the `nums` array into a min-heap has a time complexity of `O(n)` where `n` is the number of elements in `nums`.
2. **K Pop and Push Operations**: We pop the smallest element and then push an incremented value back onto the heap, `k` times. Each such operation might take `O(log n)` time since in the worst case, the element might need to sift down/up the heap which is a tree of height `log n`. Therefore, the time complexity for this part is `O(k log n)`.
3. **Final Iteration to Calculate Product**: We iterate over the heap of size `n` once and do a multiplication each time. Since the heap is already a valid heap structure, and we are simply iterating over it, the iteration takes `O(n)` time.

Thus, the overall time complexity is `O(n + k log n + n)` = `O(n + k log n)`, assuming `k` pop and push operations dominate for larger values of `k`.

Space Complexity

The space complexity is determined by:

1. **Heap In-Place**: Since the heap is constructed in-place, it does not require additional space proportional to the input size beyond the initial list. Therefore, we consider this `O(1)` additional space.
2. **Intermediate Variables**: Only a constant amount of extra space is used for variables like `ans` and `mod`, which is also `O(1)`.

Therefore, the space complexity of the algorithm is `O(1)`.