3066. Minimum Operations to Exceed Threshold Value II

Medium Array Simulation Heap (Priority Queue)

Problem Description

In this problem, we are given an array of integers `nums` and a target number `k`. Our goal is to perform a series of operations on the array until every number in the array is greater than or equal to `k`. An operation consists of the following steps:

1. Select the two smallest integers in the array, let's call them `x` and `y`.
2. Remove both `x` and `y` from the array.
3. Calculate a new number by using the formula `min(x, y) * 2 + max(x, y)` and insert this new number back into the array.

We repeat these operations until it's no longer possible to perform them (when the array has fewer than two elements) or when all elements of the array are at least `k`. Our task is to determine the minimum number of such operations required to achieve the goal.

Intuition

To solve this problem, we follow a greedy strategy. We always select the two smallest numbers in the array because this has the potential to increase the smallest number as much as possible in a single operation, thereby getting closer to our target `k` faster.

We can efficiently find the two smallest numbers using a min heap, which is a tree-like data structure that allows us to always access the smallest number in constant time, and to add a new number or remove the smallest number in logarithmic time relative to the number of elements in the heap.

The steps are as follows:

1. Put all the numbers from the given array into a min heap.
2. Check the smallest number in the min heap. If it's already greater than or equal to `k`, or if there is only one number in the min heap, the process ends.
3. If the smallest number is less than `k`, pop the two smallest numbers from the min heap, calculate the new number using the formula, and push the new number back into the min heap.
4. Increment the count of operations.
5. Repeat these steps until you can no longer perform an operation or you've reached the goal.

Using this approach, the solution provided above implements these steps and returns the count of operations needed to meet the condition that all array elements are greater than or equal to `k`.

Solution Approach

The implementation of the solution for the given problem heavily utilizes a data structure known as a "priority queue," more specifically, a min heap. A min heap allows us to efficiently perform operations like insertion and extraction of the smallest element. This aligns with our requirement to always choose the two smallest elements for each operation.

Here's the step-by-step breakdown of the implementation based on the algorithms and patterns we are using:

1. **Heapify the Input Array:** We start by converting the input array `nums` into a min heap using the `heapify` function. This ensures that we can access the smallest elements quickly, which is crucial for our operations.
2. **Initialize Operation Counter:** We set `ans` to 0 to keep track of the number of operations performed.
3. **Loop Until Conditions Are Met:** We keep performing operations if the array has more than one element and the smallest element is less than `k`.
4. **Pop Two Smallest Elements:** In each iteration, we use `heappop` twice to remove and get the two smallest elements `x` and `y` from the min heap.
5. **Combine and Add New Element:** We calculate `min(x, y) * 2 + max(x, y)` to get the new element and then use `heappush` to add this new element back into the min heap.
6. **Increment the Operation Counter:** Increase the count of operations `ans` by 1 after each iteration.
7. **Check for Completion:** The loop continues until the smallest element in the heap is at least `k` or until the heap contains fewer than 2 elements.
8. **Return the Result:** The variable `ans` now holds the minimum number of operations needed, and we return it as the result of the function.

By following this approach, we can ensure that at every step, we are increasing the smallest number in the most efficient way possible to reach or exceed our target `k`. Since extracting from a priority queue is an $O(\log n)$ operation, and heap insertion (push) is also $O(\log n)$, the total time complexity of this approach becomes $O(n \log n)$ where `n` is the size of the input array.

Example Walkthrough

Let's take a small example to illustrate the solution approach:

Suppose `nums = [1, 2, 9, 3]` and the target number `k = 10`.

Building the Min Heap

Using the `heapify` function on `nums`, we get the heap:

```
[1, 2, 9, 3]  // Min heap representation (smallest number is at the root)
```

Operations

Now we perform the following steps until every number in the array is greater than or equal to `k`:

1. **First Operation:**
   - Pop the two smallest elements (`1` and `3`).
   - Calculate the new element: `min(1, 2) * 2 + max(1, 2) = 1 * 2 + 2 = 4`.
   - Push the new element (`4`) back into the min heap.
   - The updated min heap: `[3, 4, 9]`
2. **Second Operation:**
   - Pop the two smallest elements (`3` and `4`).
   - Calculate the new element: `min(3, 4) * 2 + max(3, 4) = 3 * 2 + 4 = 10`.
   - Push the new element (`10`) back into the min heap.
   - The updated min heap: `[9, 10]`

Check for Completion

Now, the smallest number in the min heap is `9`, and we are left with two elements. However, we need every element to be at least `10`, so we perform one more operation:

3. **Third Operation:**
   - Pop the two smallest elements (`9` and `10`).
   - Calculate the new element: `min(9, 10) * 2 + max(9, 10) = 9 * 2 + 10 = 28`.
   - Push the new element (`28`) back into the min heap.
   - The updated min heap: `[28]`

Finished

The min heap now contains only one element, `28`, which is greater than `k`. We cannot perform any more operations.

We performed a total of `3` operations to ensure every number in the array became greater than or equal to `k`. Therefore, our function would return `3` as the minimum number of operations needed for this example.

Solution Implementation

Python

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

class Solution:
    def min_operations(self, nums: List[int], k: int) -> int:
        # Turn nums into a min-heap in-place
        heapify(nums)

        # Initialize the count of operations to 0
        operations_count = 0

        # Continue operations until heap has at least two elements
        # and the smallest element is less than k
        while len(nums) > 1 and nums[0] < k:
            # Pop the two smallest elements from the heap
            first_smallest = heappop(nums)
            second_smallest = heappop(nums)

            # Combine the two elements as per the given operation
            # The new element is the sum of double the smaller element and the larger element
            new_element = first_smallest * 2 + second_smallest

            # Add the new element to the heap
            heappush(nums, new_element)

            # Increment the count of operations
            operations_count += 1

        # Return the count of operations performed
        return operations_count
```

Java

```java
class Solution {
    // Method to calculate the minimum number of operations to reach numbers >= k.
    public int minOperations(int[] nums, int k) {
        // Create a priority queue to store the elements in non-decreasing order.
        PriorityQueue<Long> priorityQueue = new PriorityQueue<>();
        // Add all numbers from the given array to the priority queue.
        for (int num : nums) {
            priorityQueue.offer((long) num);
        }
        // Initialize a counter to keep track of the number of operations performed.
        int operationsCount = 0;
        // Process the elements in the priority queue while there are more than
        // one element and the smallest element is less than k.
        while (priorityQueue.size() > 1 && priorityQueue.peek() < k) {
            // Increment the operations counter.
            operationsCount++;
            // Pop the two smallest elements from the priority queue.
            long first = priorityQueue.poll();
            long second = priorityQueue.poll();
            // Combine the elements as per given in the problem requirement:
            // Replace them with min * 2 + max and add back to priority queue.
            priorityQueue.offer(Math.min(first, second) * 2 + Math.max(first, second));
        }
        // Return the final count of operations or -1 if the requirement is not met.
        return priorityQueue.peek() >= k ? operationsCount : -1;
    }
}
```

C++

```cpp
#include <vector>
#include <queue>

class Solution {
public:
    int minOperations(std::vector<int>& nums, int target) {
        using ll = long long; // Define a shorthand type name for 'long long'

        // Create a min-heap to store numbers in increasing order
        std::priority_queue<ll, std::vector<ll>, std::greater<ll>> minHeap;

        // Add all numbers from the input vector to the min-heap
        for (int num : nums) {
            minHeap.push(num);
        }

        int operations = 0;  // Initialize the number of operations to 0

        // Process the heap until only one element is left or the smallest element
        // in the heap is not less than the target value 'k'
        while (minHeap.size() > 1 && minHeap.top() < target) {
            ll first = minHeap.top(); // Take out the smallest number
            minHeap.pop();

            ll second = minHeap.top(); // Take out the next smallest number
            minHeap.pop();

            // Combine the two numbers by the given operation and add back to heap
            // Only consider combinations lesser than k to avoid unnecessary operations
            minHeap.push(std::min(first, second) * 2 + std::max(first, second));

            operations++;  // Increment the number of operations performed
        }

        // Return the number of operations needed
        // If the remaining number is smaller than k, then it is impossible
        // to reach or exceed k using the operations, return -1 in this case
        return (minHeap.top() >= target) ? operations : -1;
    }
};
```

TypeScript

```typescript
// Importing MinPriorityQueue from required source/library
import { MinPriorityQueue } from 'some-priority-queue-library'; // Replace with actual path

/**
 * Computes the minimum number of operations required to increase each element
 * in the given array to at least 'k' by using a special operation.
 * The operation consists of removing the two smallest elements 'x' and 'y',
 * and then adding an element '2 * x + min(x, y) + max(x, y)' back to the array.
 *
 * @param {number[]} nums - The input array of numbers.
 * @param {number} k - The target minimum value for every element.
 * @returns {number} The minimum number of operations required,
 *                   or -1 if the operation can't be completed.
 */
function minOperations(nums: number[], k: number): number {
    // Initialize a new priority queue to store the numbers.
    const priorityQueue = new MinPriorityQueue<number>();

    // Enqueue all the numbers from the input array into the priority queue.
    nums.forEach(num => {
        priorityQueue.enqueue(num);
    });

    // Initialize a counter for the number of operations performed.
    let operationsCount = 0;

    // Perform operations until the smallest element is at least 'k'
    // or until there is only one element left in the priority queue.
    while (priorityQueue.size() > 1 && priorityQueue.front().element < k) {
        // Increment the operation count.
        ++operationsCount;

        // Dequeue the two smallest elements.
        const smaller = priorityQueue.dequeue().element;
        const larger = priorityQueue.dequeue().element;

        // Perform the operation and enqueue the result back into the priority queue.
        const newElement = 2 * smaller + larger;
        priorityQueue.enqueue(newElement);
    }

    // Check if the operation was successful, i.e., all elements are >= k.
    if (priorityQueue.size() === 1 && priorityQueue.front().element < k) {
        // If it's not possible to reach at least 'k' for the remaining element,
        // return -1 indicating the operation cannot be completed.
        return -1;
    }

    // Return the total number of operations performed.
    return operationsCount;
}

// Usage example:
// const result = minOperations([1, 2, 3, 4], 10);
// console.log(result); // Output will depend on the result of the operations.
```

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

class Solution:
    def min_operations(self, nums: List[int], k: int) -> int:
        # Turn nums into a min-heap in-place
        heapify(nums)

        # Initialize the count of operations to 0
        operations_count = 0

        # Continue operations until heap has at least two elements
        # and the smallest element is less than k
        while len(nums) > 1 and nums[0] < k:
            # Pop the two smallest elements from the heap
            first_smallest = heappop(nums)
            second_smallest = heappop(nums)

            # Combine the two elements as per the given operation
            # The new element is the sum of double the smaller element and the larger element
            new_element = first_smallest * 2 + second_smallest

            # Add the new element to the heap
            heappush(nums, new_element)

            # Increment the count of operations
            operations_count += 1

        # Return the count of operations performed
        return operations_count
```

Time and Space Complexity

The time complexity of the given code is $O(n \log n)$. This is derived from the operations on the heap. Each `heappop` operation takes $O(\log n)$ time since it needs to maintain the heap invariant after removing the smallest element. In the worst case, every element in the heap might need to be combined to reach a value at least `k`, leading to $O(n)$ such operations. Each combination requires two `heappop` operations and one `heappush` operation, each of which takes $O(\log n)$ time. Therefore, the combined complexity is $O(n \log n)$.

The space complexity of the code is $O(n)$. The main extra space is used for the min-heap, which stores all the elements of the array. Since the size of the heap is directly proportional to the number of elements `n`, the space complexity is linear relative to the input size, hence $O(n)$.