2530. Maximal Score After Applying K Operations

Medium Greedy Array Heap (Priority Queue) Leetcode Link

Problem Description

You are presented with an integer array `nums` and an integer `k`, and you start with a score of `0`. In a single operation:

1. Choose an index `i` such that `0 <= i < nums.length`.
2. Increase your score by `nums[i]`.
3. Replace `nums[i]` with `ceil(nums[i] / 3)`.

The goal is to find the maximum score you can achieve after exactly `k` operations. Note that `ceil(val)` is the smallest integer greater than or equal to `val`.

Here's what you need to remember:

- You can apply operations only `k` times.
- You want to maximize your score with these `k` operations.

Intuition

The intuition behind solving this problem comes from the need to maximize the score gained from each operation. To do this, you should always choose the maximum number currently in the array because that will give you the highest possible addition to your score before it gets reduced.

To efficiently track and extract the maximum number at every step, a max heap is ideal. A max heap is a data structure that always allows you to access and remove the largest element in constant time. In Python, you use a min heap by negating the values, as the standard library `heapq` module supports only min heap.

Here's the step-by-step reasoning:

1. Negate all the values in the array and build a min heap, which simulates a max heap.
2. For `k` iterations, do the following:
   - Pop the heap's top element (the smallest negative number, which is the max in the original array).
   - Add the negation of the popped value to your total score (since the heap contains negatives, you negate again to get the original positive value).
   - Calculate `ceil(v / 3)` for the popped value `v`, negate it and then push it back to the heap.

By repeating this process `k` times, you are each time taking the current maximum value, getting the points from it, and then reducing its value before putting it back into the heap. This allows you to maximize the score at each of the `k` steps.

Solution Approach

The solution leverages a priority queue, particularly a max heap, to efficiently manage the elements while performing the operations. Since Python's built-in `heapq` module provides a min heap implementation, the elements are negated before being added to the heap so that retrieving the element with the maximum value (min heap for the negated values) can be done in constant time.

Here's a breakdown of the solution approach:

1. **Initialization:** Transform `nums` into a heap in-place by negating its values, effectively creating a max heap. The negative sign is necessary because Python doesn't have a built-in max heap, so this workaround is used (`h = [-v for v in nums]`). Then, the `heapify` function converts the array into a heap (`heapify(h)`).

2. **Iteration:** Repeat the following steps `k` times, corresponding to the number of operations allowed:

   - Extract (`heappop`) the root of the heap, which is the smallest negative number or the largest number before negation. Then negate it (`v = -heappop(h)`) to get back to the original value.
   - Add this value to `ans`, which keeps track of the score after each operation.
   - Compute the new value to be inserted back into the heap. The new value is the ceiling of `v / 3`, negated to maintain the correct heap order (`heappush(h, -(ceil(v / 3)))`).
3. **Scoring & Updating:** Each iteration increases the score by the max element's value, and then the heap is updated to reflect the change in the selected element's value post-operation.

By following these steps, the solution ensures that we are always picking the maximum available element, thereby maximizing the score with each operation. After `k` iterations, the score `ans` is returned, which is the maximum possible score after performing exactly `k` operations.

Example Walkthrough

Let's go through the process with a small example:

Suppose `nums = [4, 5, 6]` and `k = 2`.

Before we start any operations, we will first negate all the values in `nums` to simulate a max heap: `[-4, -5, -6]`. Then we convert this array into a heap.

Step 1: Build a Heap

We transform the array into a heap in-place: `heapify(h)` makes `h = [-6, -5, -4]`.

Step 2: Perform the First Operation

a. We pop the largest element (which is the smallest when negated): `v = -heappop(h) => v = 6` (since `heappop(h)` gives `-6`).

b. We add this value to our score `ans`: `ans = 0 + 6 = 6`.

c. We calculate the new value for the next heap insert, which is the ceiling of `v / 3` and then negate it to maintain the max heap property: `heappush(h, -(ceil(6/3))) => heappush(h, -2)`, so `h` becomes `[-5, -2, -4]`.

Step 3: Perform the Second Operation

a. Again, we pop the largest element from `h`: `v = -heappop(h) => v = 5`.

b. We add the value to our score: `ans = 6 + 5 = 11`.

c. We calculate the new value for the next heap insert: `heappush(h, -(ceil(5/3))) => heappush(h, -2)`, and `h` now becomes `[-4, -2, -2]`.

At this point, we have performed `2` operations, which is our limit `k`, so we are done.

Result:

The maximum score that can be achieved after exactly `k = 2` operations is `11`. This result comes from choosing the operations that maximize the score at each step, using the heap to efficiently select these operations.

Python Solution

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

class Solution:
    def maxKElements(self, nums: List[int], k: int) -> int:
        # Invert the values of the nums array to use the heapq as a min-heap,
        # since Python's heapq module implements a min-heap instead of a max-heap.
        min_heap = [-value for value in nums]
        heapify(min_heap)  # Convert the list into a heap

        total_sum = 0  # Initialize the sum of the max 'k' elements

        # Extract 'k' elements from the heap
        for _ in range(k):
            # Pop the smallest (inverted largest) element from the heap
            # and revert it to its original value
            value = -heappop(min_heap)
            total_sum += value  # Add the element to the sum

            # Calculate the new value by taking the ceiling division by 3
            # and push its inverted value onto the heap
            new_value = -(ceil(value / 3))
            heappush(min_heap, new_value)

        return total_sum  # Return the sum of the extracted elements

# Example usage:
# solution = Solution()
# print(solution.maxKElements([4, 5, 6], 3))

```

Java Solution

```java
class Solution {
    // Method to find the maximum sum of k elements with each element replaced by a third of its value.
    public long maxKelements(int[] nums, int k) {
        // Create a max heap using PriorityQueue to store the elements in descending order.
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

        // Add all elements from the array to the max heap.
        for (int value : nums) {
            maxHeap.offer(value);
        }

        // Initialize a variable to store the sum of the maximum k elements.
        long sum = 0;
        // Iterate k times to get the sum of the maximum k elements.
        while (k-- > 0) {
            // Poll the top element from the max heap (the maximum element).
            int maxElement = maxHeap.poll();
            // Add the maximum element to the sum.
            sum += maxElement;
            // Replace the maximum element with a third of its value and add it back to the max heap.
            maxHeap.offer((maxElement + 2) / 3);
        }

        // Return the sum of the maximum k elements.
        return sum;
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to calculate the sum of the max 'k' elements following a specific rule
    long long maxKelements(vector<int>& nums, int k) {
        // Creating a max priority queue using the elements of 'nums'
        priority_queue<int> maxHeap(nums.begin(), nums.end());

        // This variable will store the sum of the max 'k' elements
        long long sum = 0;

        // Iterate 'k' times
        while (k > 0) {
            // Retrieve the top element from the max heap (which is the current maximum)
            int currentValue = maxHeap.top();
            maxHeap.pop(); // Remove the top element

            sum += currentValue;  // Add the current maximum to the sum

            // Apply the rule: replace the extracted value with (value + 2) / 3
            int newValue = (currentValue + 2) / 3;
            maxHeap.push(newValue);  // Push the new value back into the max heap

            k--; // Decrement the counter
        }

        // Return the calculated sum
        return sum;
    }
};

```

Typescript Solution

```typescript
import { MaxPriorityQueue } from '@datastructures-js/priority-queue'; // assuming required package is imported

/**
 * Function to compute the sum of the 'k' maximum elements after each is replaced
 * with the flooring result of that element plus two, divided by three.
 *
 * @param nums Array of numbers from which we are finding the max 'k' elements.
 * @param k The number of maximum elements we want to sum.
 * @return The sum of the 'k' maximum elements.
 */
function maxKElements(nums: number[], k: number): number {
    // Initialize a priority queue to store the numbers.
    const priorityQueue = new MaxPriorityQueue<number>();

    // Iterate over the array of numbers and enqueue each element into the priority queue.
    nums.forEach(num => priorityQueue.enqueue(num));

    // Initialize the answer variable to accumulate the sum of max 'k' elements.
    let sum = 0;

    // Loop to process 'k' elements.
    while (k > 0) {
        // Dequeue the largest element from the priority queue.
        const currentValue = priorityQueue.dequeue().element;
        // Add the value to the sum.
        sum += currentValue;
        // Compute the new value using the specified formula and enqueue it back to the priority queue.
        priorityQueue.enqueue(Math.floor((currentValue + 2) / 3));
        // Decrement 'k' to move to the next element.
        k--;
    }

    // Return the accumulated sum.
    return sum;
}

```

Time and Space Complexity

The time complexity of this function primarily comes from two operations: the heapification of the `nums` list and the operations within the loop that runs `k` times.

- The `heapify(h)` function takes `O(n)` time to create a heap from an array of `n` elements.
- Inside the loop, each `heappop(h)` and `heappush(h, -(ceil(v / 3)))` operation has a time complexity of `O(log n)` because both operations involve maintaining the heap invariant, which requires a restructuring of the heap. Since these operations are called `k` times, the complexity within the loop is `O(k * log n)`.

So, the total time complexity of the function is `O(n + k * log n)`, where `n` is the length of the `nums` list.

Now, let's consider the space complexity. The function uses additional memory for the heap, which is `h` in the code. This copy of the list adds a space complexity of `O(n)`. If we consider auxiliary space complexity (i.e., not counting space taken up by the input itself), it could be argued that the space complexity is `O(1)` because we are reusing the space of the given input list `nums` to create the heap `h`.

Thus, the space complexity can be `O(n)` considering the total space used, including input, or `O(1)` if considering only auxiliary space, without the input.