2558. Take Gifts From the Richest Pile

Easy Array Simulation Heap (Priority Queue) Leetcode Link

Problem Description

In this problem, you are given an array called `gifts`, which contains integers representing the number of gifts in different piles. You have a task to perform every second for `k` seconds, and the task goes as follows:

1. Identify the pile with the most gifts. If there are multiple piles with the same maximum number of gifts, you may choose any one of them.
2. Leave behind the largest integer less than or equal to the square root of the number of gifts in that pile, also known as the floor of the square root. Take the rest of the gifts from the pile.

After performing this task every second for `k` seconds, you need to determine the total number of gifts left across all piles.

Intuition

To efficiently manage the process of finding and updating the largest pile every second, a max heap data structure is employed. A max heap is a binary tree where the parent node is always greater than its children nodes, making it easy to retrieve and remove the maximum element in the heap.

Here is the intuition behind the solution approach:

1. Convert the list of gifts into a max heap. In Python, a max heap is not provided by default, but a min heap is provided by the `heapq` library. Therefore, we can simulate a max heap by negating all values in the `gifts` array when adding them to the heap. This way, the smallest number (after negation, which was actually the largest) comes at the top of the heap.
2. Iterate `k` times to simulate the seconds passing. During each iteration, pop the maximum element from the heap (which will be returned as a negative number due to our earlier negation), calculate the floor of the square root of that number (negate it again to maintain the heap property), and put the result back into the heap.
3. After doing this for `k` seconds, the heap contains negative numbers representing the gifts left in each pile. Sum up the negative numbers and negate the result to get the total number of gifts remaining.

This approach lets us update the piles and find out the total number of remaining gifts efficiently.

Solution Approach

The key data structure used in the solution approach is a heap, which is a specialized tree-based data structure that satisfies the heap property. In a max heap, for any given node `i`, the value of `i` is greater than or equal to the values of its children, and the maximum element is always at the root node. The algorithm utilizes a min heap with negated values to mimic the behavior of a max heap, as Python's `heapq` module only provides a min heap implementation.

The process of the algorithm is as follows:

1. We first negate all the values in the `gifts` array and convert this negated list into a heap using `heapify`. The `heapify` function transforms the list into a heap in O(n) time.

```
h = [-v for v in gifts]
heapify(h)
```

2. We then repeatedly perform `k` iterations to simulate each second. In each iteration, we use the `heapreplace` function, which first pops the root element of the heap (the smallest element, or, in our negated heap, the pile with the maximum gifts), then calculates the floor of the square root of the negated number (effectively the square root of the original maximum), negates it (to keep it consistent with the negated heap), and finally places this value back into the heap.

```
for _ in range(k):
    heapreplace(h, -int(sqrt(-h[0])))
```

The `heapreplace` function consolidates the operations of popping and pushing an element while ensuring the heap property is maintained during the entire iteration. It operates in O(log n) time because it needs to maintain the heap structure after each replacement.

3. Lastly, after all `k` iterations are completed, we have the negated values of gifts remaining in each pile. The sum of all remaining gifts is the sum of the negated values negative to turn it back into a positive number.

```
return -sum(h)
```

This solution method is designed to optimize the process of choosing and updating the pile with the maximum gifts efficiently while also ensuring that the overall time complexity is kept to O(k log n), where `n` is the number of piles, and `k` is the number of seconds the process is run.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Suppose we have the following array of `gifts` and we want to perform the task for `k = 3` seconds:

```
gifts = [9, 7, 4, 1]
```

Initially, we have four piles with 9, 7, 4, and 1 gifts, respectively.

Step 1: Convert to a Max Heap

Firstly, we negate all the values and convert the `gifts` array to a heap to effectively create a max heap:

```
h = [-9, -7, -4, -1]
heapify(h)  # h becomes [-9, -7, -4, -1]
```

Now, our heap (max heap with negated values) represents the piles as [-9, -7, -4, -1].

Step 2: Iterations for `k` Seconds

Next, we simulate the task for `k` seconds:

For the first second (`k=1`):

- The maximum (negated minimum) value is -9.
- We pop this value and calculate the floor of the square root: `int(sqrt(9)) = 3`.
- We leave behind 3 gifts in the pile and take the rest, which is 6.
- We negate the leftover (-3) and push it back into the heap.

```
heapreplace(h, -3)  # h becomes [-7, -3, -4, -1]
```

For the second second (`k=2`):

- Now the maximum (negated minimum) value is -7.
- We pop this value and calculate the floor of the square root: `int(sqrt(7)) = 2`.
- We leave behind 2 gifts and take 5 from this pile.
- We negate the leftover (-2) and push it back into the heap.

```
heapreplace(h, -2)  # h becomes [-4, -3, -2, -1]
```

For the third second (`k=3`):

- The maximum (negated minimum) value is now -4.
- We pop this value and calculate the floor of the square root: `int(sqrt(4)) = 2`.
- We leave behind 2 gifts and take 2 from this pile.
- After negating the leftover (-2) we push it back into the heap.

```
heapreplace(h, -2)  # h becomes [-3, -2, -2, -1]
```

Step 3: Calculate Total Gifts Remaining

After `k` seconds, our heap represents the remaining gifts in each pile as [-3, -2, -2, -1]. To get the total number of gifts remaining, we sum these values and negate the result:

```
total_gifts_remaining = -sum(h)  # -(-3 -2 -2 -1) = 8
```

So, after performing the task for `k = 3` seconds, we have a total of 8 gifts remaining across all piles.

Python Solution

```
from heapq import heapify, heapreplace
from math import sqrt

class Solution:
    def pickGifts(self, gifts: List[int], k: int) -> int:
        # Negate the values of gifts for min-heap behavior with max-heap semantics
        min_heap = [-gift for gift in gifts]
        # Transform the list into a heap (in-place)
        heapify(min_heap)

        # Perform the operation 'k' times
        for _ in range(k):
            # Replace the root of the heap with the negated square root of the -ve root value.
            # since the root is the largest number due to min-heap representation
            heapreplace(min_heap, -int(sqrt(-min_heap[0])))

        # Return the negated sum of values in the heap, which restores their original sign
        return -sum(min_heap)
```

Java Solution

```
import java.util.PriorityQueue;

public class Solution {

    /**
     * Picks gifts by processing the top k elements with the highest values, replacing each with their square root,
     * then sums up the values left in the gifts array.
     *
     * @param gifts An array of integers representing gift values.
     * @param k The number of times to pick the gift with the highest value and replace it with its square root.
     * @return The sum of the final values of the gifts.
     */
    public long pickGifts(int[] gifts, int k) {
        // Create a max-heap priority queue to store the gifts, such that the largest value is always at the top.
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

        // Add all the gifts into the max-heap.
        for (int value : gifts) {
            maxHeap.offer(value);
        }

        // Process the top k elements by replacing each with the integer part of its square root.
        while (k-- > 0 && !maxHeap.isEmpty()) {
            int highestValue = maxHeap.poll(); // Retrieve and remove the gift with the highest value.
            maxHeap.offer((int) Math.sqrt(highestValue)); // Replace it with its square root and reinsert into the queue.
        }

        // Calculate the sum of the final values of the gifts.
        long totalValue = 0;
        for (int value : maxHeap) {
            totalValue += value;
        }

        // Return the total sum of values.
        return totalValue;
    }
}
```

C++ Solution

```
#include <vector>
#include <algorithm>
#include <numeric>
#include <cmath>

class Solution {
public:
    // Function to calculate the sum of the largest `k` gifts after taking the square root once for each gift.
    long long pickGifts(vector<int>& gifts, int k) {
        // Convert the array into a max-heap to facilitate easy retrieval of the largest element.
        make_heap(gifts.begin(), gifts.end());

        // Apply the operation `k` times.
        while (k--) {
            // Move the largest element to the end of the vector.
            pop_heap(gifts.begin(), gifts.end());

            // Replace the last element (previously the largest) with its square root.
            // The static_cast<int> is used to convert the result of sqrt to an integer,
            // since the gifts array contains integers.
            gifts.back() = static_cast<int>(sqrt(gifts.back()));

            // Restore the heap property after modifying the value.
            push_heap(gifts.begin(), gifts.end());
        }

        // Sum all the elements in the heap and return the result.
        // The third argument (0LL) is the initial sum value, specified as a long long to prevent overflow.
        return accumulate(gifts.begin(), gifts.end(), 0LL);
    }
};
```

Typescript Solution

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

/**
 * Adjust the value of gifts by replacing the largest value with its square root k times
 * and then calculate the sum of the adjusted values.
 *
 * @param {number[]} gifts - The array of initial gift values.
 * @param {number} k - The number of replacements to make.
 * @returns {number} - The sum of the gift values after k replacements.
 */
function pickGifts(gifts: number[], k: number): number {
    // Initialize a max priority queue to manage gift values by priority.
    const maxQueue = new MaxPriorityQueue<number>();

    // Enqueue all gifts into the priority queue.
    gifts.forEach(value => maxQueue.enqueue(value));

    // Perform k replacements of the max gift value with its square root.
    while (k > 0) {
        const maxGiftValue = maxQueue.dequeue().element; // Take out the max gift value.
        const adjustedValue = Math.floor(Math.sqrt(maxGiftValue)); // Calculate its square root.
        maxQueue.enqueue(adjustedValue); // Put the adjusted value back into the queue.
        k -= 1; // Decrement the number of replacements left.
    }

    // Sum up all the values that are left in the priority queue.
    let totalValue = 0;
    while (!maxQueue.isEmpty()) {
        totalValue += maxQueue.dequeue().element;
    }

    // Return the total sum of the gift values after k adjustments.
    return totalValue;
}
```

Time and Space Complexity

The time complexity of the provided code is `O(n + k * log n)`. This is because initializing a heap using `heapify` from a list of `n` elements has a time complexity of `O(n)`. After heapification, the code performs `k` operations where each operation involves popping the smallest element from the heap and replacing it with the square root of that element negated, which takes `O(log n)` time due to the need to maintain the heap structure after each replacement. Therefore, the loop will contribute `O(k * log n)` to the total time complexity. Combined, we have an overall time complexity of `O(n)` from the `heapify` plus `O(k * log n)` from the loop, yielding `O(n + k * log n)`.

The space complexity of the code is `O(n)`, which accounts for the storage of the heap. No additional data structures that are dependent on the size of the input are used beyond the initial heap `h`. As such, the space consumed is directly proportional to the size of the input list `gifts`.