1354. Construct Target Array With Multiple Sums

Hard Array Heap (Priority Queue) Leetcode Link

Problem Description

The LeetCode problem provides an array called `target` which contains `n` integers. The objective is to determine if this `target` array can be formed from an initial array called `arr`, which consists of `n` elements each with a value of 1. To transform the `arr` into the `target`, a specific procedure can be followed:

- Calculate the sum (`s`) of all elements in your current array.
- Choose an index `i`, where `0 <= i < n`, and update the value at index `i` in `arr` to `x`.
- Repeat this process as necessary.

The goal is to find out if the `target` array can be created from the initial `arr` by following the aforementioned procedure. The function should return `true` if it is possible, or `false` if it is not.

Intuition

To tackle this problem, we need to think about it in reverse. Instead of starting from the initial array and trying to build up to the `target`, we'll start from the `target` and attempt to reduce it to the initial array. Why? The operations described in the problem tend to exponentially increase the values in the array, making it challenging to directly reach the `target` array values.

Since we're attempting to reverse the operation, here are the steps that lead us to the intuitive solution:

1. We use a max heap to keep track of the largest element in the `target` array because our operation always works on the current largest element.
2. At each step, we replace the largest element with the difference between it and the sum of the rest of the elements. This effectively simulates the reverse operation.
3. If the largest element happens to be 1 or if the sum of the other elements is 1, we've effectively reached our starting array of all 1's, so we can return `true` immediately.
4. We must also consider a couple of special cases where our problem can have no solution:
   - If the largest element remains the same after the modulo operation or becomes 0, we can't reach our goal.
   - Since the target array was built up from an array of ones, it's impossible to make the largest number smaller than the sum of the other numbers - if that happens, we cannot build the `target` from the initial array.

While iterating through the max heap process, if all elements in the `target` can be reduced to 1, it's possible to construct the `target` from the `arr`. If we hit a point where elements can't be reduced further following the rules, it's impossible, and we return `false`.

Solution Approach

The implementation of the solution uses a max heap data structure to manage the `target` array during the conversion process. This is because we need to efficiently access and update the largest element, and a max heap enables us to do this with efficiency. The Python `heapq` module, which only supports min heaps by default, is used to create a max heap by inserting the negation of each number in the `target` array.

Here's a detailed walk-through of the implementation steps:

1. Special Case Check: If the length of the `target` array is `1`, the only way to have a valid target is if the lone element is also `1`. This returns `True` immediately in such a scenario.
2. Initialize a max heap using the negative values of the `target` array elements and get the sum of all elements in `target`.
3. While the largest element (the root of the max heap) is greater than `1`, we perform the following actions:
   - Retrieve and negate the largest element from the max heap (to convert it back to a positive number).
   - Calculate the sum of the remaining elements by subtracting the largest element from the total sum.
   - Check if the sum of the remaining elements is `1`. If so, we return `True` immediately because this implies the array can be reduced to all `1`'s (as per the problem description).
   - Calculate the updated value for the largest element by taking the modulo of it with the sum of the remaining elements.
   - Check if the updated value is `0` or didn't change. If either is true, we cannot form the target array and return `False`.
   - Push the negated updated value back onto the max heap.
   - Update the total sum with the new value of the updated element.
4. If the loop concludes without returning `False`, this means all elements have been reduced to `1` (or were already `1`), so we return `True`.

The algorithm utilized here involves both elements of heap manipulation and a clever mathematical trick of working in reverse. The modulo operation serves as a substitute for multiple iterations of the described procedure in the problem, thus greatly optimizing the number of steps required to reach the solution.

Although the problem appears to be a forward problem, the implemented solution uses backward reasoning. By considering how we could've arrived at the `target` array, we can more easily determine if it's possible or not. This form of reasoning is often applied in problems that involve building or transforming collections of elements, especially when the transformations are complex or expansive.

Example Walkthrough

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

Suppose we have a `target` array of `[1, 3, 5]`.

1. Since there is more than one element in the `target`, we don't return `True` from the special case check.
2. Initialize a max heap with the negative values of the `target` array to help us easily find and extract the maximum element. After inserting the elements, our heap will look like `[-5, -3, -1]`. The total sum at this point is `1+3+5=9`.
3. Enter the while loop and start processing the heap:
   - Extract the largest element, `-5` (which is `5` when converted back to positive).
   - Calculate the sum of the remaining elements, `9 - 5 = 4`.
   - The sum of the remaining elements is not `1`, so we do not return `True` at this point.
   - Update the largest element's value by taking `5 % 4 = 1`.
   - Check if the updated value is `0` or did not change (Neither is true in this case).
   - Push the negated updated value back onto the heap, which becomes `[-3, -1, -1]`.
   - Update the total sum to `4 + 1 = 5`.
4. Repeat the process until the largest element is no longer greater than `1`:
   - Extract `-3`, the largest element now, giving us `3`.
   - The sum of the rest is `5 - 3 = 2`.
   - Update the largest element's value with `3 % 2 = 1`.
   - Check for the unchanged or zero value (None are true).
   - The heap is now updated to `[-1, -1, -1]` and the total sum to `2 + 1 = 3`.

This process continues, and each time we are either left with the heap elements being all `1`s or finding a situation where the largest number is modulo'd to 1. When all elements in the heap are `1`, or the sum of the rest of elements becomes `1`, we exit the while loop and return `True`.

For our example, after inserting all the negated updated values (-1 for each), the heap is `[-1, -1, -1]`, and the sum is `3` (which equals the number of elements in the heap). Since the loop has concluded and we didn't encounter a condition that would return `False`, we return `True`; it is indeed possible to construct the `target` from an array of all `1`'s using the described procedure.

Python Solution

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

class Solution:
    def isPossible(self, target: List[int]) -> bool:
        # If the target array has only one element, it should be 1 to form a valid array.
        if len(target) == 1:
            return target[0] == 1

        # Calculate the sum of elements in the target array.
        total_sum = sum(target)
        # Convert target elements to negative numbers because Python has a min heap by default.
        max_heap = [-num for num in target]
        # Transform the array into a heap in-place.
        heapify(max_heap)

        while -max_heap[0] > 1:
            # Pop the largest element from the heap.
            max_element = -heappop(max_heap)
            # Calculate the sum of the rest of the elements.
            remaining_sum = total_sum - max_element

            # If the remaining sum is 1, it is always possible to build the array.
            if remaining_sum == 1:
                return True

            # Find the updated value of the target's largest element.
            updated_element = max_element % remaining_sum

            # If updated_element is 0 or the same as the original max_element, there won't be any progress.
            if updated_element == 0 or updated_element == max_element:
                return False

            # Push the updated element back into the heap and update the total sum.
            heappush(max_heap, -updated_element)
            total_sum = total_sum - max_element + updated_element

        # All elements have been reduced to 1, hence the original array can be constructed.
        return True
```

Java Solution

```java
import java.util.PriorityQueue;

public class Solution {
    public boolean isPossible(int[] target) {
        // If the target array has only one element, it should be 1 to form a valid array.
        if (target.length == 1) {
            return target[0] == 1;
        }

        // Calculate the sum of elements in the target array.
        long totalSum = 0;
        // Create a max heap in Java by providing a custom comparator.
        PriorityQueue<Long> maxHeap = new PriorityQueue<>((a, b) -> Long.compare(b, a));
        for (int num : target) {
            totalSum += num;
            maxHeap.add((long)num);
        }

        while (maxHeap.peek() > 1) {
            // Pop the largest element from the heap.
            long maxElement = maxHeap.poll();
            // Calculate the sum of the rest of the elements.
            long remainingSum = totalSum - maxElement;

            // If the remaining sum is 1, it is always possible to build the array.
            if (remainingSum == 1) {
                return true;
            }

            // Find the updated value of the target's largest element.
            long updatedElement = maxElement % remainingSum;

            // If updatedElement is 0 or does not change the maxElement, there won't be any progress.
            if (updatedElement == 0 || updatedElement == maxElement) {
                return false;
            }

            // Push the updated element back into the heap and update the total sum.
            maxHeap.add(updatedElement);
            totalSum = totalSum - maxElement + updatedElement;
        }

        // All elements have been reduced to 1, hence the original array can be constructed.
        return true;
    }
}
```

C++ Solution

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

class Solution {
public:
    bool isPossible(std::vector<int>& target) {
        // If the target array has only one element, it should be 1 to form a valid array.
        if (target.size() == 1) {
            return target[0] == 1;
        }

        // Calculate the sum of elements in the target array.
        long total_sum = 0;
        for (int num : target) {
            total_sum += num;
        }

        // Create a max heap from the target vector.
        std::priority_queue<int> max_heap(target.begin(), target.end());

        while (max_heap.top() > 1) {
            // Pop the largest element from the heap.
            int max_element = max_heap.top();
            max_heap.pop();

            // Calculate the sum of the rest of the elements.
            long remaining_sum = total_sum - max_element;

            // If the remaining sum is 1, it is always possible to build the array.
            if (remaining_sum == 1) {
                return true;
            }

            // Find the updated value of the target's largest element.
            int updated_element = max_element % remaining_sum;

            // If updated_element is 0 or the same as the original max_element, there won't be any progress.
            if (updated_element == 0 || updated_element == max_element) {
                return false;
            }

            // Push the updated element back into the heap and update the total sum.
            max_heap.push(updated_element);
            total_sum = total_sum - max_element + updated_element;
        }

        // All elements have been reduced to 1, hence the original array can be constructed.
        return true;
    }
};
```

Typescript Solution

```typescript
// Importing the priority queue module for maintaining a max-heap
import { PriorityQueue } from 'typescript-collections';

// Function to check if it is possible to form a target array starting with an array full of 1's
function isPossible(target: number[]): boolean {
    // If the target array has only one element, it should be 1 to be formed by the described process.
    if (target.length === 1) {
        return target[0] === 1;
    }

    // Calculate the total sum of elements in the target array.
    let totalSum: number = target.reduce((a, b) => a + b, 0);

    // Create a max heap from the target array elements.
    const maxHeap: PriorityQueue<number> = new PriorityQueue<number>((a, b) => b - a);
    target.forEach(element => {
        maxHeap.enqueue(element);
    });

    while (maxHeap.peek() > 1) {
        // Extract the largest element from the heap.
        const maxElement: number = maxHeap.dequeue();
        // Calculate the sum of the remaining elements.
        const remainingSum: number = totalSum - maxElement;

        // If the remaining sum is 1, then it's always possible to reconstruct the array.
        if (remainingSum === 1) {
            return true;
        }

        // Find the updated value for the largest element in the target array.
        const updatedElement: number = maxElement % remainingSum;

        // If the updated element is 0 or doesn't make progress, return false.
        if (updatedElement === 0 || updatedElement === maxElement) {
            return false;
        }

        // Push the updated element back into the heap and update the total sum.
        maxHeap.enqueue(updatedElement);
        totalSum = totalSum - maxElement + updatedElement;
    }

    // The process has reduced all elements to 1, thus the target array can be constructed.
    return true;
}
```

Time and Space Complexity

Time Complexity

The primary operations here involve a while loop that runs as long as the maximum value in the heap is greater than `1`. In each iteration of the while loop, the operation complexity can be broken down as follows:

- Extracting the maximum element from the heap: $O(\log n)$, where n is the number of elements in the heap since a heap operation typically takes logarithmic time.
- Calculating `restSum`: $O(1)$ as it is just a subtraction operation.
- Calculating `updated`: $O(1)$ since it is a modulo operation.
- Insertion into the heap: $O(\log n)$ for the same reason as the extraction.

Considering the loop can run up to n times, where n is the value of the largest element in the initial array, in the average or worst case, the time complexity could be approximated to $O(n \log n)$. However, this is not strict because n can decrease drastically in each iteration after the modulo operation is applied.

Space Complexity

The extra space used in the algorithm is for the heap. Since all elements of the input array must be stored, the space complexity is $O(n)$, where n is the size of the input array. There is no additional space usage that grows with the size of the input; hence the space complexity remains linear with respect to the input array size.