1354. Construct Target Array With Multiple Sums Hard

### Heap (Priority Queue) Array

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:

Leetcode Link

- Calculate the sum (x) 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.</li> 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.

**Problem Description** 

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.

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.

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

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

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

- 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

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.

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

is because we need to frequently access and update the largest element, and a max heap enables us to do this efficiently. The

# 2. Initialize a max heap using the negative values of the target array elements and get the sum of all elements in target.

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

reduced to all 1's (as per the problem description).

Update the total sum with the new value of the updated element.

returns True immediately in such a scenario.

 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. o Check if the sum of the remaining elements is 1. If so, we return True immediately because this implies the array can be

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.

3. While the largest element (the root of the max heap) is greater than 1, we perform the following actions:

- 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.
- 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.

The algorithm utilized here involves both elements of heap manipulation and a clever mathematical trick of working in reverse. The

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

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:

### ◦ Push the negated updated value back onto the heap, which becomes [-3, -1, -1]. Update the total sum to 4 + 1 = 5.

Python Solution

class Solution:

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

36

37

38

39

40

13

14

15

15

16

17

19

20

21

22

23

24

25

26

27

28

29

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

48

49

51

50 };

from typing import List

expansive.

4. Repeat the process until the largest element is no longer greater than 1:

Extract -3, the largest element now, giving us 3.

 $\circ$  The sum of the rest is 5 - 3 = 2.

from heapq import heapify, heappop, heappush

def isPossible(self, target: List[int]) -> bool:

max\_heap = [-num for num in target]

heapify(max\_heap)

return True

1 import java.util.PriorityQueue;

public boolean isPossible(int[] target) {

return target[0] == 1;

if (target.length == 1) {

for (int num : target) {

totalSum += num;

total\_sum += num;

while (max\_heap.top() > 1) {

if (remaining\_sum == 1) {

return true;

return false;

return true;

max\_heap.push(updated\_element);

max\_heap.pop();

// Create a max heap from the target vector.

int max\_element = max\_heap.top();

// Pop the largest element from the heap.

// Calculate the sum of the rest of the elements.

long remaining\_sum = total\_sum - max\_element;

std::priority\_queue<int> max\_heap(target.begin(), target.end());

// Find the updated value of the target's largest element.

if (updated\_element == 0 || updated\_element == max\_element) {

int updated\_element = max\_element % remaining\_sum;

total\_sum = total\_sum - max\_element + updated\_element;

// If the remaining sum is 1, it is always possible to build the array.

// Push the updated element back into the heap and update the total sum.

// All elements have been reduced to 1, hence the original array can be constructed.

// If updated\_element is 0 or the same as the original max\_element, there won't be any progress.

public class Solution {

while -max\_heap[0] > 1:

if remaining\_sum == 1:

return True

# Transform the array into a heap in-place.

max\_element = -heappop(max\_heap)

# Pop the largest element from the heap.

remaining\_sum = total\_sum - max\_element

# Calculate the sum of the rest of the elements.

updated\_element = max\_element % remaining\_sum

# Find the updated value of the target's largest element.

total\_sum = total\_sum - max\_element + updated\_element

 $\circ$  Calculate the sum of the remaining elements, 9 - 5 = 4.

Update the largest element's value by taking 5 % 4 = 1.

- Update the largest element's value with 3 % 2 = 1. Check for the unchanged or zero value (None are true).
- 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.

• The heap is now updated to [-1, -1, -1] and the total sum to 2 + 1 = 3.

Extract the largest element, -5 (which is 5 when converted back to positive).

• The sum of the remaining elements is not 1, so we do not return True at this point.

Check if the updated value is 0 or did not change (Neither is true in this case).

# 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 9 # Calculate the sum of elements in the target array. 10 total\_sum = sum(target) 11

# Convert target elements to negative numbers because Python has a min heap by default.

# If the remaining sum is 1, it is always possible to build the array.

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

// If the target array has only one element, it should be 1 to form a valid array.

PriorityQueue<Long> maxHeap = new PriorityQueue<>((a, b) -> Long.compare(b, a));

This process continues, and each time we are either left with the heap elements being all 1 or finding a situation where the largest

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

False, we return True: it is indeed possible to construct the target from an array of all 1's using the described procedure.

equals the number of elements in the heap). Since the loop has concluded and we didn't encounter a condition that would return

29 30 # 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: 31 32 return False 33 34 # Push the updated element back into the heap and update the total sum. 35 heappush(max\_heap, -updated\_element)

```
9
           // Calculate the sum of elements in the target array.
10
11
            long totalSum = 0;
           // Use a max heap in Java by providing a custom comparator.
```

Java Solution

```
maxHeap.add((long)num);
16
17
18
           while (maxHeap.peek() > 1) {
19
20
               // Pop the largest element from the heap.
21
                long maxElement = maxHeap.poll();
22
               // Calculate the sum of the rest of the elements.
23
                long remainingSum = totalSum - maxElement;
24
25
               // If the remaining sum is 1, it is always possible to build the array.
26
               if (remainingSum == 1) {
27
                    return true;
28
29
               // Find the updated value of the target's largest element.
30
                long updatedElement = maxElement % remainingSum;
31
32
33
               // If updatedElement is 0 or does not change the maxElement, there won't be any progress.
               if (updatedElement == 0 || updatedElement == maxElement) {
34
35
                    return false;
36
37
38
               // Push the updated element back into the heap and update the total sum.
39
               maxHeap.add(updatedElement);
                totalSum = totalSum - maxElement + updatedElement;
40
41
           // All elements have been reduced to 1, hence the original array can be constructed.
44
           return true;
45
46 }
47
C++ Solution
 1 #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;
10
11
12
           // Calculate the sum of elements in the target array.
           long total_sum = 0;
13
            for (int num : target) {
14
```

```
Typescript Solution
   // 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;
10
       // Calculate the total sum of elements in the target array.
       let totalSum: number = target.reduce((a, b) => a + b, 0);
13
14
       // Create a max heap from the target array elements.
       const maxHeap: PriorityQueue<number> = new PriorityQueue<number>((a, b) => b - a);
15
       target.forEach(element => {
16
17
           maxHeap.enqueue(element);
       });
18
19
       while (maxHeap.peek() > 1) {
20
           // Extract the largest element from the heap.
21
22
           const maxElement: number = maxHeap.dequeue();
23
24
           // Calculate the sum of the remaining elements.
25
           const remainingSum: number = totalSum - maxElement;
26
27
           // If the remaining sum is 1, then it's always possible to reconstruct the array.
           if (remainingSum === 1) {
28
29
               return true;
30
31
32
           // Find the updated value for the largest element in the target array.
33
           const updatedElement: number = maxElement % remainingSum;
34
35
           // If the updated element is 0 or doesn't make progress, return false.
           if (updatedElement === 0 || updatedElement === maxElement) {
36
37
               return false;
38
39
           // Push the updated element back into the heap and update the total sum.
40
           maxHeap.enqueue(updatedElement);
41
42
           totalSum = totalSum - maxElement + updatedElement;
43
44
       // The process has reduced all elements to 1, thus the target array can be constructed.
45
       return true;
46
47 }
48
```

## 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:

**Time and Space Complexity** 

# typically takes logarithmic time. Calculating restSum: 0(1) as it is just a subtraction operation.

Space Complexity

**Time Complexity** 

Calculating updated: 0(1) since it is a modulo operation.

• Extracting the maximum element from the heap: 0(log n), where n is the number of elements in the heap since a heap operation

- Insertion into the heap: O(log n) for the same reason as the extraction. Considering the loop can run up to m times, where m is the value of the largest element in the initial array, in the average or worst
- case, the time complexity could be approximated to 0(m log n). However, this is not strict because m can decrease drastically in each iteration after the modulo operation is applied.

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.