2208. Minimum Operations to Halve Array Sum Greedy Array Heap (Priority Queue) Medium

Problem Description

are allowed to select reduced numbers in subsequent operations. The objective is to determine the minimum number of such operations needed to achieve the goal of halving the array's sum.

The goal of this problem is to reduce the sum of a given array nums of positive integers to at least half of its original sum through a

series of operations. In each operation, you can select any number from the array and reduce it exactly to half of its value. You

Intuition

To solve this problem, a greedy approach is best, as taking the largest number and halving it will most significantly impact the

sum in the shortest number of steps. Using a max heap data structure is suitable for efficiently retrieving and halving the largest

1. Calculating the target sum, which is half the sum of the array nums.

number at each step.

The solution involves:

2. Creating a max heap to access the largest element quickly in each operation. 3. Continuously extracting the largest element from the heap, halving it, and adding the halved value back to the heap. 4. Accumulating the count of operations until the target sum is reached or passed.

operation.

This approach guarantees that each step is optimal in terms of reducing the total sum and reaching the goal using the fewest operations possible.

Solution Approach

The solution uses the following steps, algorithms, and data structures:

Max Heap (Priority Queue): The Python code uses a max heap, which is implemented using a min heap with negated values (the heapq library in Python only provides a min heap). A max heap allows us to easily and efficiently access and remove the largest element in the array.

factor comes from the push and pop operations of the heap, which occur for each of the n elements.

heappush(h, -t) # Push the negated halved value back onto the heap

heappush(h, -v) # Negate and push all values to the heap

Sum Reduction Tracking: We keep track of the current sum of nums we need to halve, starting with half of the original sum of nums. After halving and pushing the largest element back onto the heap, we decrement this sum by the halved value.

s = sum(nums) / 2 # Target sum to achieve

ans += 1 # Increment operation count

Let's walk through an example to illustrate the solution approach.

Suppose we have the following array of numbers: nums = [10, 20, 30].

The sum of nums is 60, so halving the sum gives us a target of 30.

h = [] # Initial empty heap

for v in nums:

Calculating the Target Sum:

Creating a Max Heap:

negated again when it's popped off to maintain the original sign.

Condition Check: The loop continues until the sum we are tracking is less than or equal to zero, meaning the total sum of the original array has been reduced by at least half. In terms of algorithm complexity, this solution operates in O(n log n) time where n is the number of elements in nums. The log n

Halving the Largest Element: At each step, the code pops the largest value in the heap, halves it, and pushes the negated

Counting Operations: A variable ans is used to count the number of operations. It is incremented by one with each halving

half value back on to the heap. Since we're using a min heap, every value is negated when it's pushed onto the heap and

The code implementation based on these steps is shown in the reference solution with proper comments to highlight each step: class Solution: def halveArray(self, nums: List[int]) -> int:

ans = 0 # Operation counter while s > 0: # Continue until we've halved the array sum t = -heappop(h) / 2 # Extract and halve the largest value s -= t # Reduce the target sum by the halved value

return ans # Return the total number of operations needed **Example Walkthrough**

Reducing Sum with Operations: We now perform operations which will involve halving the largest element and keeping a tally of operations. **First Operation:** The current sum we need to track is 30.

Extract the largest element (-30) and halve its value (15).

Push the negated halved value back onto the heap (-15).

Extract the largest element (-15) and halve its value (7.5).

Push the negated halved value back onto the heap (-7.5).

Since the sum we are tracking is now less than 0, the loop ends.

Calculate the sum of half the elements to determine the target

Add negative values of nums to the heap to simulate max heap

Retrieve and negate the largest element, then halve it

Push the halved negative value back to maintain the max heap

Return the total number of operations needed to reach the target sum

// Priority queue to store the array elements in descending order.

// Add all elements to the priority queue and calculate the total sum.

// The target is to reduce the sum to less than half of its original value.

// Divide it by 2 (halving the element) and subtract from halfSum.

// Function to find the minimum number of operations to reduce array sum to less than or equal to half of the initial sum.

double targetHalf = total / 2.0; // Our target is to reduce the total to this value or less

operations to reduce the sum of the array to less than or equal to half its original sum by performing

let targetSum: number = nums.reduce((accumulator, currentValue) => accumulator + currentValue) / 2;

// Enqueue all numbers in the array into the max priority queue with their values as priorities

// Re-enqueue the halved element to ensure correct ordering in the priority queue

Initialize a max heap (using negative values because Python has a min heap by default)

// Initialize a max priority queue to facilitate the retrieval of the largest element

double topValue = maxHeap.top() / 2.0; // Halve the largest number in max heap

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

total -= topValue; // Subtract the halved value from the total sum

maxHeap.push(topValue); // Push the halved value back into max heap

maxHeap.pop(); // Remove this largest number from max heap

return operations; // Return the total number of operations performed

operations++; // Increment the number of operations

// This function takes an array of numbers and returns the minimum number of

// Calculate the target sum which is half the sum of the input array

// Continue reducing the total sum until it's less than or equal to targetHalf

PriorityQueue<Double> maxHeap = new PriorityQueue<>(Collections.reverseOrder());

Subtract the halved value from the target sum

Subtract 7.5 from the target sum, which would make it negative (-2.5).

Subtract 15 from the target sum, leaving us with 15.

The heap is now h = [-20, -15, -10].

Increment the operation count (ans = 1).

The heap is now h = [-15, -10, -10].

The heap is now h = [-10, -10, -7.5].

Increment the operation count (ans = 3).

We create a max heap with the negated values of nums: h = [-30, -20, -10].

Extract the largest element (-20) and halve its value (10). Push the negated halved value back onto the heap (-10).

Third Operation:

0

Result:

Solution Implementation

 $max_heap = []$

for value in nums:

while target_sum > 0:

operations += 1

for (int value : nums) {

double halfSum = sum / 2.0;

sum += value;

operations++;

return operations;

while (total > targetHalf) {

// Importing PriorityQueue to use in the implementation

function halveArray(nums: number[]): number {

for (const value of nums) {

// Initialize the operation counter

let operationCount: number = 0;

dequeuedItem /= 2;

operationCount += 1;

return operationCount;

from heapq import heappush, heappop

max heap = []

for value in nums:

while target_sum > 0:

operations += 1

Time and Space Complexity

halving operations needed.

The space complexity of the algorithm is determined by:

Space Complexity

return operations

target_sum -= largest

operations = 0

target_sum = sum(nums) / 2

// Example usage:

class Solution:

targetSum -= dequeuedItem;

// const result = halveArray([10, 20, 7]);

def halveArray(self, nums: List[int]) -> int:

heappush(max_heap, -value)

Counter for the number of operations performed

largest = -heappop(max_heap) / 2

heappush(max_heap, -largest)

Increment the operation count

// Increment the operation counter

// Return the total number of operations performed

import { MaxPriorityQueue } from 'typescript-collections';

// operations that halve the value of any element in the array.

const maxPriorityQueue = new MaxPriorityQueue<number>();

// Subtract the halved value from the remaining sum

maxPriorityQueue.engueue(degueuedItem, degueuedItem);

// console.log(result); // Outputs the number of operations needed

Calculate the sum of half the elements to determine the target

Add negative values of nums to the heap to simulate max heap

Keep reducing the target sum until it reaches 0 or below

Subtract the halved value from the target sum

Retrieve and negate the largest element, then halve it

Push the halved negative value back to maintain the max heap

Return the total number of operations needed to reach the target sum

into the heap is $O(\log(n))$, and it is performed n times for n elements.

maxPriorityQueue.enqueue(value, value);

maxHeap.offer((double) value);

double largest = maxHeap.poll();

halfSum -= largest / 2.0;

maxHeap.offer(largest / 2.0);

// Increment the operation counter.

return operations

target_sum -= largest

target_sum = sum(nums) / 2

heappush(max_heap, -value)

largest = -heappop(max_heap) / 2

heappush(max_heap, -largest)

Increment the operation count

Second Operation:

Subtract 10 from the target sum, leaving us with 5. Increment the operation count (ans = 2).

The minimum number of operations required to reduce the sum of nums to at least half its original sum is 3.

from heapq import heappush, heappop class Solution: def halveArray(self, nums: List[int]) -> int:

Python

Counter for the number of operations performed operations = 0 # Keep reducing the target sum until it reaches 0 or below

Initialize a max heap (using negative values because Python has a min heap by default)

```
Java
class Solution {
   public int halveArray(int[] nums) {
       // Initial sum of the array elements.
       double sum = 0;
```

// Counter for the number of operations performed. int operations = 0; // Continue until we reduce the sum to less than half. while (halfSum > 0) {

// Add the halved element back to the priority queue.

// Retrieve and remove the largest element from the queue.

// Return the number of operations required to achieve the target.

```
int halveArray(vector<int>& nums) {
   // Use a max heap to keep track of the largest numbers in the array
   priority_queue<double> maxHeap;
   double total = 0; // Original total sum of the array elements
    for (int value : nums) {
        total += value; // Accumulate total sum
        maxHeap.push(value); // Add current value to the max heap
```

#include <vector>

using namespace std;

#include <queue>

class Solution {

C++

public:

}; **TypeScript**

```
// Continue until the remaining sum is reduced to targetSum or less
while (targetSum > 0) {
   // Dequeue the largest element
    let dequeuedItem = maxPriorityQueue.dequeue().value;
   // Halve the dequeued element
```

```
Time Complexity
  The given algorithm consists of multiple steps that contribute to the overall time complexity:
```

the sum by half. In the worst-case scenario, every element is divided multiple times. For each halving operation, we remove the maximum element (0(log(n)) complexity for removal) and insert it back into the heap (0(log(n)) complexity for insertion). The number of such operations could vary, but it could potentially be 0(m * log(n)), where m is the number of

Sum calculation: The sum of the array is calculated with a complexity of O(n), where n is the number of elements in nums.

Heap construction: Inserting all elements into a heap has an overall complexity of 0(n * log(n)) as each insertion operation

Halving elements until sum is reduced: The complexity of this part depends on the number of operations we need to reduce

• Sum computation: O(n) Heapification: O(n * log(n)) Halving operations: 0(m * log(n))

Therefore, the time complexity of the algorithm is determined by summing these complexities:

Since the number of halving operations m is not necessarily linear and depends on the values in the array, we cannot directly relate it to n. As a result, the overall worst-case time complexity of the algorithm is 0(n * log(n) + m * log(n)).

Heap storage: We store all n elements in the heap, which requires O(n) space.

Auxiliary space: Aside from the heap, the algorithm uses a constant amount of extra space for variables like s, t, and ans. Hence, the space complexity is O(n).