2279. Maximum Bags With Full Capacity of Rocks

because adding rocks to them will quickly increase the count of fully filled bags.

```
Problem Description
```

Medium

Greedy Array Sorting

In this problem, we're dealing with a set of n bags, each with a certain capacity. The capacity array represents the maximum number of rocks each bag can hold, while the rocks array shows the current number of rocks in each bag. We also have a certain number of additionalRocks that we can distribute across these bags.

The objective is to maximize the number of bags that are filled to their full capacity using the additionalRocks available. A bag is considered to be at full capacity if the number of rocks it contains equals its capacity.

Intuition

The intuition behind the solution is based on the idea that to maximize the number of bags at full capacity, we should fill the bags that need the fewest additional rocks first. This is a <u>greedy</u> approach where we prioritize the bags that are closest to being full

We calculate the difference between the capacity and rocks for each bag, which represents the number of additional rocks needed to fill each bag to capacity. We then sort this list to get the bags that need the least additional rocks at the beginning.

Stepping through this sorted list, we distribute the additionalRocks as long as we have enough to fill a bag. Each time we fill a bag, we decrement the number of additionalRocks by the number used and increment the count of fully-filled bags by one. This process continues until we run out of additionalRocks or fill all the bags. The final count of fully-filled bags is the maximum

number we can achieve with the given additionalRocks.

Solution Approach

The solution implements a simple greedy strategy using Python lists and sorting algorithm. Here's a step-by-step walkthrough of

reach full capacity. This is done using a list comprehension:

ans = 0

the implementation:

d = [a - b for a, b in zip(capacity, rocks)]
Here, a represents an element from the capacity array and b represents the corresponding element from the rocks array. The

Calculate the difference between the capacity and the rocks for each bag to find out how many more rocks are needed to

zip function pairs each element from capacity with the corresponding element from rocks.

2. The problem is now reduced to filling the bags with the least difference first. To do this efficiently, sort the list d in non-decreasing order:

Initialize a variable ans to count the number of bags that can be filled to full capacity:

d.sort()

ans by 1 and reduce additionalRocks by the amount used:

for v in d:
 if v <= additionalRocks:
 ans += 1
 additionalRocks -= v</pre>

Iterate through the sorted list d and try to fill each bag. If the additionalRocks is enough to fill the current bag, increase the

If additionalRocks is at least v, it means we can fill this bag. Then we update additionalRocks to reflect the rocks used.
The loop continues either until there are no more rocks left (additionalRocks is less than the next v in the list) or all bags are checked.
Once the loop is complete, ans is the maximum number of bags that can be filled to full capacity, and the function returns this value:

return ans

First, we calculate the difference between capacity and rocks for each bag.

d = [a - b for a, b in zip(capacity, rocks)] # d will become [1, 1, 2]

We initialize ans to keep track of the bags that can be filled to full capacity:

Now, we iterate through the sorted list d and distribute additionalRocks:

ans += 1 # Increment the count of full bags

additionalRocks -= v # Decrease the additionalRocks by v

v represents the number of additional rocks needed for the current bag.

Lette illustrate the colution engrees with a small example. Suppose we have the following inpute:

time complexity of this approach is $0(n \log n)$. The space complexity is 0(n) due to the creation of the list d.

Let's illustrate the solution approach with a small example. Suppose we have the following inputs:

• capacity array: [5, 3, 7]

This approach uses the built-in sorting function which typically has a time complexity of 0(n log n) where n is the number of

elements in the list. The subsequent iteration through the sorted list has a linear time complexity of O(n). As a result, the overall

We want to find out the maximum number of bags that can be filled to their full capacity using these 3 additional rocks.

rocks array: [4, 2, 5]

additionalRocks: 3

2. We then sort this list to consider the bags that need the fewest additional rocks first:

The first bag needs 1 rock to be full, which we have, so ans becomes 1 and additionalRocks becomes 2.

The second bag also needs 1 rock, so ans is incremented to 2 and additionalRocks is reduced to 1.

Loop through [1, 1, 2] with additionalRocks starting at 3
for v in d:
 if v <= additionalRocks:</pre>

d.sort() # The sorted list d will remain [1, 1, 2]

Following the steps outlined in the solution approach:

Using the list comprehension:

ans = 0 # Starts at 0

of ans:

Python

from typing import List

remaining_capacity.sort()

class Solution:

The third bag needs 2 rocks to be full. However, we only have 1 additionalRock left, so we cannot fill this bag to capacity.

The process stops here as we have distributed all additional rocks that we can. The final count of fully-filled bags is the value

return ans # Returns 2 as the maximum number of full bags

currently in each bag from the bag's total capacity.

use those rocks to fill the bag

filled_bags += 1

break

} else {

break;

return maxFilledBags;

// with a given number of additional rocks.

// Get the number of bags

let filledBags = 0;

const numBags = capacity.length;

requiredRocks.sort((a, b) => a - b);

// 'capacity' array represents the capacity of each bag.

// Iterate over the sorted bags and try to fill them

filledBags++; // Increment filled bags count

// 'rocks' array represents the current number of rocks in each bag.

// 'additionalRocks' represents the total number of additional rocks available.

const requiredRocks = capacity.map((cap, index) => cap - rocks[index]);

// Calculate the difference between bag capacity and current number of rocks

// Initialize a counter to keep track of the number of bags that can be filled

function maximumBags(capacity: number[], rocks: number[], additionalRocks: number): number {

// Sort the required rocks in ascending order — to prioritize bags that need fewer rocks to reach capacity

for (let i = 0; i < numBags && (requiredRocks[i] === 0 || requiredRocks[i] <= additionalRocks); i++) {</pre>

additionalRocks -= requiredRocks[i]; // Subtract the used rocks from the additional rocks

// Return the maximum number of bags that can be filled.

if required_rocks <= additionalRocks:</pre>

Increment the filled bags counter

as no further bags can be completely filled

- Therefore, we have maximized the number of full bags (2 out of 3) using the 3 additionalRocks provided.

 Solution Implementation
 - # Initialize the counter for the number of bags that can be completely filled
 filled_bags = 0

 # Iterate through the sorted remaining capacities
 for required_rocks in remaining_capacity:

If the current bag requires fewer or equal rocks than we have available,

Sort the remaining capacities to prioritize bags that need fewer rocks to reach capacity

remaining_capacity = [total_cap - current_rocks for total_cap, current_rocks in zip(capacity, rocks)]

def maximumBags(self, capacity: List[int], rocks: List[int], additionalRocks: int) -> int:

Calculate the remaining capacity of each bag by subtracting the number of rocks

```
# Decrement the available rocks by the number of rocks used for the current bag
additionalRocks -= required_rocks
else:
# If the current bag requires more rocks than available, break the loop,
```

```
# Return the total number of completely filled bags
        return filled_bags
Java
class Solution {
    // Function to determine the maximum number of bags that can be filled given capacities, current rocks, and additional rocks.
    public int maximumBags(int[] capacity, int[] rocks, int additionalRocks) {
       // Get the number of bags by checking the length of the capacity array.
        int numBags = capacity.length;
       // Create an array to store the difference between capacity and current rocks in each bag.
        int[] remainingCapacity = new int[numBags];
       // Calculate the remaining capacity for each bag.
        for (int i = 0; i < numBags; ++i) {</pre>
            remainingCapacity[i] = capacity[i] - rocks[i];
       // Sort the remaining capacities in ascending order; to fill as many bags as possible starting with the ones requiring th
       Arrays.sort(remainingCapacity);
        // Initialize a counter for the maximum number of bags that can be filled.
        int maxFilledBags = 0;
        // Iterate over the sorted remaining capacities.
        for (int requiredRocks : remainingCapacity) {
           // If the required rocks to fill a bag is less than or equal to the available additional rocks...
            if (requiredRocks <= additionalRocks) {</pre>
                // Increment the count of filled bags.
                maxFilledBags++;
                // Subtract the used rocks from the available additional rocks.
                additionalRocks -= requiredRocks;
```

// If the remaining rocks are not sufficient to fill the next bag, break out of the loop.

```
C++
class Solution {
public:
    // Function to find the maximum number of bags that can be filled given the remaining capacity.
    int maximumBags(vector<int>& capacity, vector<int>& rocks, int additionalRocks) {
        int numBags = capacity.size(); // Get the number of bags.
        vector<int> remainingCapacity(numBags); // Vector to hold remaining capacities of the bags.
        // Calculate the remaining capacity for each bag.
        for (int i = 0; i < numBags; ++i) {</pre>
            remainingCapacity[i] = capacity[i] - rocks[i];
        // Sort the remaining capacities in ascending order.
        sort(remainingCapacity.begin(), remainingCapacity.end());
        int maxFilledBags = 0; // Counter for maximum number of bags that can be completely filled.
        // Iterate over each bag's remaining capacity.
        for (int& remaining : remainingCapacity) {
            // If there are not enough rocks to fill the next bag, break the loop.
            if (remaining > additionalRocks) break;
            // If we have enough rocks to fill the current bag:
            maxFilledBags++;
additionalRocks -= remaining; // Increment the count of filled bags.

// Use the rocks to fill the bag.
            maxFilledBags++;
        return maxFilledBags; // Return the maximum number of bags that can be filled.
};
TypeScript
// Function to determine the maximum number of bags that can be filled to capacity
```

```
// Return the number of bags that have been filled to capacity
return filledBags;
}
```

```
from typing import List
class Solution:
   def maximumBags(self, capacity: List[int], rocks: List[int], additionalRocks: int) -> int:
       # Calculate the remaining capacity of each bag by subtracting the number of rocks
       # currently in each bag from the bag's total capacity.
        remaining_capacity = [total_cap - current_rocks for total_cap, current_rocks in zip(capacity, rocks)]
       # Sort the remaining capacities to prioritize bags that need fewer rocks to reach capacity
        remaining_capacity.sort()
       # Initialize the counter for the number of bags that can be completely filled
        filled_bags = 0
       # Iterate through the sorted remaining capacities
        for required_rocks in remaining_capacity:
           # If the current bag requires fewer or equal rocks than we have available,
           # use those rocks to fill the bag
           if required_rocks <= additionalRocks:</pre>
               # Increment the filled bags counter
               filled_bags += 1
               # Decrement the available rocks by the number of rocks used for the current bag
               additionalRocks -= required rocks
           else:
               # If the current bag requires more rocks than available, break the loop,
               # as no further bags can be completely filled
               break
       # Return the total number of completely filled bags
        return filled_bags
Time and Space Complexity
```

Time Complexity The time complexity of the provided code consists of several parts:

Space Complexity

The list comprehension d = [a - b for a, b in zip(capacity, rocks)] takes O(n) time, where n is the number of elements in capacity and rocks.
 Sorting the list d.sort() has a time complexity of O(n log n) because it uses the Timsort algorithm which is Python's standard sorting

3. The variables ans and additionalRocks use constant space, 0(1).

- algorithm.

 3. The loop for v in d: iterates through each element of the list d once, giving a time complexity of O(n).

 Since sorting the list is the most expensive operation, the overall time complexity of the code is O(n log n).
- The space complexity of the code also involves a few components:

 1. The list comprehension generates a new list d of size n, resulting in O(n) space complexity.

2. Sorting the list is done in-place in Python, so it doesn't require additional space other than some constant workspace, hence 0(1).

Combining these, the total space complexity of the code is O(n) because the new list d is the dominant factor.