<https://leetcode.com/problems/maximum-bags-with-full-capacity-of-rocks/description/>

You have n bags numbered from 0 to n - 1. You are given two **0-indexed** integer arrays capacity and rocks. The ith bag can hold a maximum of capacity[i] rocks and currently contains rocks[i] rocks. You are also given an integer additionalRocks, the number of additional rocks you can place in **any** of the bags.

Return*the****maximum****number of bags that could have full capacity after placing the additional rocks in some bags.*

**Example 1:**

**Input:** capacity = [2,3,4,5], rocks = [1,2,4,4], additionalRocks = 2

**Output:** 3

**Explanation:**

Place 1 rock in bag 0 and 1 rock in bag 1.

The number of rocks in each bag are now [2,3,4,4].

Bags 0, 1, and 2 have full capacity.

There are 3 bags at full capacity, so we return 3.

It can be shown that it is not possible to have more than 3 bags at full capacity.

Note that there may be other ways of placing the rocks that result in an answer of 3.

**Example 2:**

**Input:** capacity = [10,2,2], rocks = [2,2,0], additionalRocks = 100

**Output:** 3

**Explanation:**

Place 8 rocks in bag 0 and 2 rocks in bag 2.

The number of rocks in each bag are now [10,2,2].

Bags 0, 1, and 2 have full capacity.

There are 3 bags at full capacity, so we return 3.

It can be shown that it is not possible to have more than 3 bags at full capacity.

Note that we did not use all of the additional rocks.

**Constraints:**

n == capacity.length == rocks.length

1 <= n <= 5 \* 10^4

1 <= capacity[i] <= 10^9

0 <= rocks[i] <= capacity[i]

1 <= additionalRocks <= 10^9

**Attempt 1: 2025-05-04**

**Solution 1: Sorting + Greedy (10min)**

**Style 1: Separately handle already full capacity ones**

class Solution {

    public int maximumBags(int[] capacity, int[] rocks, int additionalRocks) {

        int n = capacity.length;

        List<Integer> remainCapacity = new ArrayList<>();

        int count = 0;

        for(int i = 0; i < n; i++) {

            int diff = capacity[i] - rocks[i];

            if(diff == 0) {

                count++;

            } else {

                remainCapacity.add(diff);

            }

        }

        Collections.sort(remainCapacity);

        for(int a : remainCapacity) {

            additionalRocks -= a;

            if(additionalRocks >= 0) {

                count++;

            } else {

// Can't fill any more bags

                break;

            }

        }

        return count;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Style 2: No need separately handle already full capacity ones**

class Solution {

    public int maximumBags(int[] capacity, int[] rocks, int additionalRocks) {

        int n = capacity.length;

        int[] remainCapacity = new int[n];

        int count = 0;

        for(int i = 0; i < n; i++) {

            remainCapacity[i] = capacity[i] - rocks[i];

        }

        Arrays.sort(remainCapacity);

        for(int a : remainCapacity) {

            additionalRocks -= a;

            if(additionalRocks >= 0) {

                count++;

            } else {

// Can't fill any more bags

                break;

            }

        }

        return count;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Refer to Deepseek**

This problem requires us to maximize the number of bags that can be filled to their full capacity using additional rocks. The optimal approach involves a greedy algorithm that prioritizes filling bags requiring the fewest additional rocks first.

**Approach**

Calculate the remaining capacity for each bag (capacity[i] - rocks[i])

Sort these remaining capacities in ascending order

Iterate through the sorted capacities, using the additional rocks to fill as many bags as possible

Count how many bags can be completely filled

**Solution Code**

class Solution {

public int maximumBags(int[] capacity, int[] rocks, int additionalRocks) {

int n = capacity.length;

int[] remainingCapacity = new int[n];

// Calculate remaining capacity for each bag

for (int i = 0; i < n; i++) {

remainingCapacity[i] = capacity[i] - rocks[i];

}

// Sort remaining capacities in ascending order

Arrays.sort(remainingCapacity);

int filledBags = 0;

// Fill bags starting with those needing fewest additional rocks

for (int i = 0; i < n && additionalRocks > 0; i++) {

if (remainingCapacity[i] <= additionalRocks) {

filledBags++;

additionalRocks -= remainingCapacity[i];

} else {

break; // Can't fill any more bags

}

}

return filledBags;

}

}

**Explanation**

Calculate Remaining Capacity: For each bag, we determine how many more rocks are needed to fill it to capacity by subtracting the current rocks from the total capacity .

Sort Capacities: By sorting the remaining capacities in ascending order, we ensure we always try to fill the bags that are closest to being full first, which is the optimal greedy strategy .

Fill Bags: We iterate through the sorted capacities, using our additional rocks to fill each bag if possible. Each time we fill a bag, we decrement our available additional rocks and increment our count of filled bags .

Early Termination: If at any point we don't have enough rocks to fill the next bag, we break out of the loop early since all remaining bags will require even more rocks .

This solution has a time complexity of O(n log n) due to the sorting step, and a space complexity of O(n) for storing the remaining capacities, which is optimal for this problem .

Alternative implementations using priority queues are possible but would have the same time complexity while potentially using more space . The sorting-based approach shown here is the most straightforward and efficient solution.

**Refer to**

[L1011.Capacity To Ship Packages Within D Days (Ref.L410,L1482)](note://WEB87ca8f9dd9a14fb4535856a4e0f8f4b1)

[L1710.Maximum Units on a Truck (Ref.L2279)](note://WEBd76efa6c4eb53430e820c0a54488436e)