# 2790. Maximum Number of Groups With Increasing Length

Sorting

## **Problem Description**

**Greedy Array Math** 

represents the maximum times each number, from 0 to n-1, can be used to create groups. The goal is to form the maximum number of groups under certain conditions: • Each group is made up of distinct numbers, meaning a particular number can't be included more than once within the same group.

In this problem, you are provided with an array called usageLimits which is 0-indexed and has a length of n. This array

- You are required to determine the maximum number of groups that can be created while adhering to the above constraints.

Each group must have more members than the previous one, except for the very first group.

**Binary Search** 

Intuition

#### To tackle this problem, we need a strategy that systematically utilizes the availability constraints specified by usageLimits. Since we need each subsequent group to have more elements than the previous, we could benefit from starting with the least usage

limits.

Hard

Hence, the first step in our approach is to sort the usageLimits array. This way, we start forming groups with numbers which have the lowest usage limits, thereby preserving the more usable numbers for later, potentially larger, groups. Now, we have to keep track of how many groups we can form. To do this, we iterate through the sorted usageLimits array and

aggregate a sum (s) of the elements, which helps us to understand the total available "slots" for forming groups at each step. The number of groups (k) is initially set to 0.

As we iterate, if the total available slots (sum s) are greater than the number of groups formed so far (k), this indicates that there is enough capacity to form a new group with one more member than the last. So we increment our group count (k) and then adjust our available slots by subtracting the size of the new group (k) from our sum (s).

The iteration continues until we've processed all elements in usageLimits. The final value of k will be the maximum number of groups we can form. This approach leverages greedy and sorting techniques to efficiently satisfy the conditions of distinct numbers and strictly

The implementation of the solution uses a <u>sorting</u> technique, iteration, and simple arithmetic calculations:

**Sorting:** We start by sorting the usageLimits array. Sorting is crucial because it ensures that we use the numbers with

Iteration: After sorting, we iterate over each element in the usageLimits array. This step helps us to accumulate the total

form a new group and we increment k. This check is essentially asking, "Do we have enough available numbers to form a

### Group Formation and Incrementation: We use two variables, k for the number of groups formed, and s for the aggregated

increasing group sizes.

**Solution Approach** 

sum of usageLimits. With each iteration, we check if the current sum s is greater than the number of groups k. If so, we can

group). The final k value when we exit the loop is the maximum number of groups we can form.

smaller limits first, allowing us to potentially create more groups.

number of times we can use the numbers to form groups.

the number of formed groups k. Initially, s = 0 and k = 0.

**Group Formation and Incrementation:** 

group larger than the current largest group?"

- Sum Adjustment: Once we increment k to add a new group, we must also adjust the sum s to reflect the creation of this group. This is achieved by subtracting k from s. The subtraction of k from s signifies that we've allocated k distinct
- numbers for the new largest group, and these numbers cannot be re-used for another group. The strategy employs a greedy approach, where we "greedily" form new groups as soon as we have enough capacity to do so,

and always ensure that the new group complies with the conditions (distinct numbers and strictly greater size than the previous

Here, we don't need any complex data structures. Array manipulation, following the sorting, and the use of a couple of integer variables to keep track of the sum and group count suffice in implementing the solution. In conclusion, the combination of sorting, greedy algorithms with a simple condition check and variable adjustment within a single

scan of the array yield the solution for the maximum number of groups that can be formed under the given constraints.

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

Here's how we might walk through the solution:

**Sorting:** First, we sort the usageLimits array in non-decreasing order: usageLimits = [0, 1, 2, 3]

Iteration: Next, we begin to iterate over the sorted usageLimits array while keeping track of the sum s of the elements, and

 $\circ$  At the third iteration, s = 1 + 2 = 3 and k = 1. There are enough numbers to form another group with 2 distinct numbers (k + 1), so k is

Sum Adjustment: After each group formation, we need to adjust s for the new group size. After the third iteration, we

#### $\circ$ At the first iteration, s = 0 + 0 = 0 and k = 0. We cannot form a group yet because there are not enough distinct numbers. $\circ$ At the second iteration, s = 0 + 1 = 1 and k = 0. We now have enough numbers to form a group with 1 distinct number, so k is

s = 3 - 2 = 1

s = 4 - 3 = 1

from typing import List

usage\_limits.sort()

for limit in usage limits:

current\_sum += limit

aroups count = 0

current\_sum = 0

class Solution:

Java

incremented to 1.

incremented to 2.

**Continuing Iteration:** 

Example Walkthrough

usageLimits = [3, 1, 2, 0]

subtract k from s:

we maximize the number of distinct groups that can be created following the rules specified.

# Initialize the number of groups and the current sum of the group limits.

# If the current sum is greater than the number of groups,

def maxIncreasingGroups(self, usage limits: List[int]) -> int:

# Iterate through the sorted usage limits.

# it means we can form a new group.

# Return the total number of groups formed.

public int maxIncreasingGroups(List<Integer> usageLimits) {

// Sort the usage limits list in ascending order

if current sum > groups count:

# Add the current limit to the current sum.

# First, sort the usage limits in non-decreasing order.

 $\circ$  At the fourth and final iteration, s = 1 + 3 = 4 and k = 2. Here, we can form one last group with 3 distinct numbers since s (4) is greater than k (2), so k is incremented to 3. We adjust s one last time:

**Result**: Since there are no more elements in usageLimits, our iteration ends. The final value of k, which is 3, is the maximum

number of groups that can be formed given the constraints. Each group having 1, 2, and 3 distinct members respectively.

The example demonstrates how by sorting the array and using a greedy approach, systematically checking and updating counts,

Solution Implementation **Python** 

# Increment the number of groups. groups count += 1 # Adiust the current sum by subtracting the new group's count. current\_sum -= groups\_count

```
// 'k' represents the number of increasing groups formed
int k = 0;
// 'sum' represents the cumulative sum of the usage limits
long sum = 0;
```

Collections.sort(usageLimits);

// Iterate over the sorted usage limits

// Add the current usage limit to 'sum'

function maxIncreasingGroups(usageLimits: number[]): number {

// Iterate through each usage limit in the sorted array

// Add the current limit to the sum of group limits

// to prepare for the next potential group

// Return the total number of increasing groups formed

// this indicates the formation of a new increasing group

// If the sum of group limits exceeds the current number of groups,

// Subtract the number of groups from the sum of group limits

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

for (const limit of usageLimits) {

sumOfGroupLimits += limit;

if (sumOfGroupLimits > groups) {

sumOfGroupLimits -= groups;

// Increment the number of groups

// Sort the input array of usage limits in non-decreasing order

// Initialize the variable 'groups' to count the maximum number of increasing groups

// Initialize the variable 'sumOfGroupLimits' to keep the sum of limits in the current group

// we can form a new group by increasing 'k'

// and then adjust the sum by subtracting 'k'

// Return the maximum number of increasing groups formed

// If the sum is greater than the number of groups formed,

// Increment the number of groups

sum -= k; // Decrement the sum by the new number of groups

for (int usageLimit : usageLimits) {

sum += usageLimit;

 $if (sum > k) {$ 

k++;

return k;

import iava.util.Collections;

import java.util.List;

class Solution {

return groups\_count

```
C++
class Solution {
public:
    // Method to calculate the maximum number of increasing groups
    int maxIncreasingGroups(vector<int>& usageLimits) {
        // Sort the usage limits in non-decreasing order
        sort(usageLimits.begin(), usageLimits.end());
        // Initialize the number of groups that can be formed
        int numberOfGroups = 0;
        // Initialize a sum variable to calculate the cumulative sum of elements
        long long sum = 0;
        // Iterate over each usage limit
        for (int limit : usageLimits) {
            // Add the current limit to the cumulative sum
            sum += limit;
            // Check if after adding the current element, the sum becomes
            // greater than the number of groups we have so far.
            // If ves, it means we can form a new group with higher limit.
            if (sum > numberOfGroups) {
                // Increment the number of groups that can be formed
                numberOfGroups++;
                // Decrease the sum by the number of groups since
                // we allocate each element of a group with a distinct size
                sum -= numberOfGroups;
        // Return the maximum number of increasing groups that can be formed
        return numberOfGroups;
```

```
class Solution:
   def maxIncreasingGroups(self. usage limits: List[int]) -> int:
       # First, sort the usage limits in non-decreasing order.
        usage_limits.sort()
```

return groups;

from typing import List

**Space Complexity** 

**TypeScript** 

let groups = 0;

let sumOfGroupLimits = 0;

groups++;

```
# Initialize the number of groups and the current sum of the group limits.
        groups count = 0
        current sum = 0
       # Iterate through the sorted usage limits.
        for limit in usage limits:
           # Add the current limit to the current sum.
           current_sum += limit
           # If the current sum is greater than the number of groups,
           # it means we can form a new group.
            if current sum > groups count:
               # Increment the number of groups.
               groups count += 1
               # Adjust the current sum by subtracting the new group's count.
               current_sum -= groups_count
       # Return the total number of groups formed.
        return groups_count
Time and Space Complexity
  The given Python code is a function that aims to determine the maximum number of increasing groups from a list of usage limits.
  Here's the analysis of its time and space complexity:
Time Complexity
  The most expensive operation in the algorithm is the sort operation, which has a time complexity of O(n log n), where n is the
  number of elements in usageLimits. After sorting, the function then iterates over the sorted list once, producing an additional
  O(n) complexity. Therefore, the total time complexity of the algorithm is dominated by the sorting step, resulting in O(n log n)
  overall.
```

additional variables used, but due to the sort's potential additional space, it could be 0(n) or 0(log n). Since Python's sort is

typically implemented in a way that aims to minimize space overhead, the expected space complexity in practical use cases would likely be closer to  $O(\log n)$ . Therefore, the final space complexity of the algorithm is  $O(\log n)$  under typical conditions. Note: The space complexity stated above assumes typical conditions based on Python's sorting implementation. The actual space complexity could vary depending on the specific details of the sort implementation in the environment where the code is

As for space complexity, the sorting operation can be O(n) in the worst case or  $O(\log n)$  if an in-place sort like Timsort (used by

Python's sort() method) is applied efficiently. The rest of the algorithm uses a fixed amount of additional variables (k and s)

and does not rely on any additional data structures that depend on the input size. Thus, the space complexity is 0(1) for the

executed.