2171. Removing Minimum Number of Magic Beans

```
Medium
         <u>Array</u>
                 Prefix Sum
                              Sorting
```

**Problem Description** 

The LeetCode problem provides us with an array beans, in which each element represents the number of magic beans in a respective bag. Our task is to make the number of beans in each non-empty bag equal by removing beans from the bags. There

are two conditions we need to obey: 1. We cannot add beans back to any bag once removed.

The goal is to determine the minimum number of beans that we need to eliminate to achieve equal amounts in all non-empty

2. After the removal, no bag should be empty; each should contain at least one bean.

bags. Intuition

To solve the problem, one might initially consider trying to make all bags have the same number of beans as the bag with the

while taking into account the total bean count. Given that we want all the non-empty bags to have an equal number of beans, a sensible approach is to make them all equal to a certain number and we never want that number to be more than the smallest non-zero number of beans in any bag. Therefore,

least amount, but that is not necessarily optimal. To minimize bean removal, we must intelligently decide which beans to remove

we can sort beans array first, which helps us consider potential equal bean counts in order. While iterating through the sorted array, we will simulate making all non-empty bags have the same number of beans as the

current bag. The number of beans to remove will then be the total sum of beans minus the beans in the current bag times the

number of remaining bags. This is because each remaining bag will have the same number of beans as the current bag, so we

multiply that by the remaining bags count (including the current bag). We repeat this process for each bag and keep track of the smallest number of beans we need to remove. This will give us the answer.

The implementation uses a simple algorithm and takes advantage of the sorted properties of the array to reduce the number of beans removed to a minimum.

## Sort the beans array. beans.sort()

n = len(beans)

**Solution Approach** 

Sorting the array allows us to approach the problem in ascending order of bean counts, ensuring we always consider

Initialize the variable ans to the sum of beans; this represents the total count that will be considered for potential removal. ans = s = sum(beans)

Calculate the length of beans array and store it in variable n.

equalizing to the smallest possible non-zero count.

Here's the step-by-step approach, paired with the code provided:

Iterate through each bag via its index i and the value v representing the number of beans in the current bag.

v is the targeted number of beans for each bag.

∘ n - i count of bags including and after the current one to be equalized.

First, we sort the beans array to have beans = [1, 4, 5, 6].

give us the count of beans left if we were to equalize all bags to the current count v.

Return the smallest value of ans found through the iteration, which represents the minimum removal.

Calculate the number of beans to be removed if we make all non-empty bags equal to the current bag's bean count. ans = min(ans, s - v \* (n - i))

for i, v in enumerate(beans):

that each remaining bag (including the current one) should contain v beans. s is the total initial number of beans.

The mathematical formula here is crucial. s - v \* (n - i) calculates the total number of beans to be removed if we decide

We subtract from the total (s) the product of the targeted number of beans (v) and the remaining bags (n - i), which would

with a single pass through a sorted array.

return ans This algorithm is efficient because it leverages the sorting of the array, which provides a natural order to consider potential equal

bean counts. The only significant data structure used is the list itself for storing and iterating over the bean counts. The simplicity

and efficiency of this solution come from not needing a more complex data structure or pattern, as the problem can be solved

Let's illustrate the solution approach using a small example. Given the array beans = [4, 1, 6, 5], we want to make the number of magic beans in each non-empty bag equal by removing the least number of beans possible, with the rule that no bags can be

We calculate the length of beans array, n = 4.

each bag.

class Solution:

beans.sort()

total\_sum = sum(beans)

# Number of bean piles

num\_piles = len(beans)

return min\_removals

long totalBeans = 0;

for (int beanCount : beans) {

long minRemovals = totalBeans;

// Iterate through the sorted beans vector

return minimumBeansToRemove;

**}**;

for (int index = 0; index < numberOfBags; ++index) {</pre>

// Calculate the number of beans to remove if all bags have beans[i] beans

// This is done by subtracting the total number of beans that would remain if

// all remaining bags had beans[index] beans from the initial sum 'totalBeans'

// Update the result with the minimum number of beans to remove found so far

minimumBeansToRemove = min(minimumBeansToRemove, beansToRemove);

// such that all the remaining non-empty bags have the same number of beans

// Return the result which is the minimum number of beans to remove

long long beansToRemove = totalBeans - 1ll \* beans[index] \* (numberOfBags - index);

totalBeans += beanCount;

min\_removals = total\_sum

def minimumRemoval(self, beans: List[int]) -> int:

# the initial answer before any removals

# Iterate through the sorted bean piles

# Sort the list of beans to enable uniform removal

# Calculate the sum of all the beans, this also represents

# Initialize the minimum removals with the total sum as the upper bound

current\_total = total\_sum - num\_beans \* (num\_piles - i)

# Return the minimum number of beans that needs to be removed

min\_removals = min(min\_removals, current\_total)

# Update the minimum removals if the current total is lower.

// Calculate the total sum of beans to prepare for further calculations.

// Initialize the answer with the total sum, assuming no removals are made yet.

this isn't optimal.

variable s.

**Example Walkthrough** 

empty after the removal.

As we iterate, we calculate the beans to be removed if we make all non-empty bags have v beans (the current bean count in our iteration): • With i=0, v=1: We could remove no beans, as every bag including and after can be 1. But since there is the maximum number of bags here,

Now, we start iterating through the sorted beans array. On each iteration, we have the index i and the beans number v for

We initialize ans to the sum of beans, which is in this case 1 + 4 + 5 + 6 = 16. At the same time, we store this sum in the

After iterating through each bag, the smallest number of beans to remove is 4. Hence, the answer to the problem is 4, which is achieved by making all non-empty bags have 4 magic beans each (equalizing to the bag with 4 beans in it).

 $\circ$  With i=1, v=4: ans = min(16, 16 - 4 \* (4-1)) which simplifies to ans = min(16, 16 - 12) resulting in ans = 4.

 $\circ$  With i=2, v=5: ans = min(4, 16 - 5 \* (4-2)) which simplifies to ans = min(4, 16 - 10) resulting in ans = 4.

 $\circ$  With i=3, v=6: ans = min(4, 16 - 6 \* (4-3)) which is ans = min(4, 16 - 6) resulting in ans = 4.

Solution Implementation **Python** 

By following this algorithm, we ensure we are checking every possible scenario to find the minimum beans we need to remove in

order to make the bags equal without leaving any empty, doing so efficiently by piggybacking off of the sorted nature of beans.

for i, num\_beans in enumerate(beans): # Calculate the current total after removing beans from the current pile onwards. # This means all piles to the right side of the current one (inclusive) # will have 'num beans' beans. # The total beans removed will be the original sum minus the beans left.

```
class Solution {
    public long minimumRemoval(int[] beans) {
       // Sort the array to allow the calculation of removals based on ascending bean quantities.
       Arrays.sort(beans);
```

Java

```
// Calculate the minimum removals needed by trying to make all piles equal to each pile's number of beans.
       int numOfPiles = beans.length;
        for (int i = 0; i < numOfPiles; ++i) {
           // Calculate the beans to remove if all piles were to be made equal to the current pile's beans.
            long beansToRemove = totalBeans - (long) beans[i] * (numOfPiles - i);
           // Update minRemovals if the current calculation yields a smaller number of beans to remove.
           minRemovals = Math.min(minRemovals, beansToRemove);
       // Return the minimum removals necessary to make all piles have an equal number of beans.
       return minRemovals;
C++
#include <vector>
#include <algorithm>
#include <numeric>
class Solution {
public:
   long long minimumRemoval(vector<int>& beans) {
       // Sort the beans vector to ensure increasing order
       sort(beans.begin(), beans.end());
       // Calculate the sum of all elements in the beans vector
        long long totalBeans = accumulate(beans.begin(), beans.end(), 0ll);
       // Initialize the answer with the total beans as the worst case
        long long minimumBeansToRemove = totalBeans;
        int numberOfBags = beans.size(); // Store the number of bags
```

```
TypeScript
  function minimumRemoval(beans: number[]): number {
      // Calculate the total number of beans.
      let totalBeans = beans.reduce((accumulator, current) => accumulator + current, 0);
      // Sort the array of beans in ascending order for easier processing.
      beans.sort((a, b) \Rightarrow a - b);
      // Initialize the answer with the total number, as a worst-case scenario.
      let minBeansToRemove = totalBeans;
      const numOfBeans = beans.length;
      // Iterate through the beans array to find the minimum beans to remove.
      for (let index = 0; index < num0fBeans; index++) {</pre>
          // Calculate the total beans to remove if we make all the bags equal to beans[index].
          let currentRemoval = totalBeans - beans[index] * (numOfBeans - index);
          // Update minBeansToRemove to the minimum of the current removal and previous minimum.
          minBeansToRemove = Math.min(currentRemoval, minBeansToRemove);
      // Return the minimum number of beans to remove.
      return minBeansToRemove;
class Solution:
   def minimumRemoval(self, beans: List[int]) -> int:
       # Sort the list of beans to enable uniform removal
        beans.sort()
       # Calculate the sum of all the beans, this also represents
       # the initial answer before any removals
        total_sum = sum(beans)
       # Number of bean piles
        num_piles = len(beans)
        # Initialize the minimum removals with the total sum as the upper bound
        min_removals = total_sum
       # Iterate through the sorted bean piles
        for i, num_beans in enumerate(beans):
            # Calculate the current total after removing beans from the current pile onwards.
            # This means all piles to the right side of the current one (inclusive)
            # will have 'num_beans' beans.
            # The total beans removed will be the original sum minus the beans left.
            current_total = total_sum - num_beans * (num_piles - i)
            # Update the minimum removals if the current total is lower.
            min_removals = min(min_removals, current_total)
        # Return the minimum number of beans that needs to be removed
        return min_removals
```

## through the sorted list to find the minimum number of beans to remove.

Time and Space Complexity

The time complexity of the code is determined by the sorting step and the iteration step. Sorting the beans list takes O(nlogn) time, where n is the number of elements in the beans list. This is because the sort()

The given Python code snippet is designed to find the minimum number of beans that need to be removed so that all the

remaining piles have the same number of beans. It first sorts the beans list, calculates the sum of all beans, and then iterates

## The for loop iterates over the sorted list exactly once, which takes O(n) time.

**Space Complexity** 

**Time Complexity** 

Combining these two steps, the overall time complexity is O(nlogn) as this is the dominating term.

method in Python uses TimSort, which has this time complexity.

- As for space complexity: The sorting operation is done in-place, and does not require additional space proportional to the input size, so its space
- complexity is 0(1). The variables ans, s, and n, and the iteration using i and v require constant space, which is also 0(1).

Therefore, the overall space complexity of the function is 0(1), indicating that it uses a constant amount of extra space.