1283. Find the Smallest Divisor Given a Threshold

Binary Search

Medium Array

Problem Description

integer divisor such that when each element of the array nums is divided by this divisor, the results of the division are rounded up to the nearest integer, and these integers are summed up, the sum is less than or equal to the given threshold. The division rounding should round up to the nearest integer greater than or equal to the result of the division. For instance, 7 divided by 3 should be rounded up to 3 (since 7/3 = 2.33, and rounding up gives us 3), and similarly, 10 divided by 2 should be rounded to 5. The problem ensures that there will always be an answer.

In this problem, you are given an array of integers nums and another integer threshold. Your task is to find the smallest positive

The key observation here is that as the divisor increases, the resultant sum after the division decreases. This relationship between the divisor and the sum is monotonous; hence we can employ a binary search strategy to efficiently find the smallest

Intuition

divisor which results in a sum that is less than or equal to the threshold. We can start our binary search with the left boundary (1) set to 1, since we are looking for positive divisors only, and the right boundary (r) set to the maximum value in nums, because dividing by a number larger than the largest number in nums would result in every division result being 1 (or close to 1), which would be the smallest possible sum. We then repeatedly divide the

range by finding the midpoint (mid) and testing if the sum of the rounded division results is less than or equal to the threshold. If it is, we know that we can possibly reduce our divisor even further; thus we adjust our right boundary to mid. If the sum exceeds threshold, we need a larger divisor, so we adjust our left boundary to mid + 1. By iteratively applying this binary search, we converge upon the smallest possible divisor that satisfies the condition. When our search interval is reduced to a single point (1 == r), we can return 1 as our answer, as it represents the smallest divisor for which the sum of the division results does not exceed the threshold.

Solution Approach The solution uses a binary search algorithm, which is a classic method to find a target value within a sorted array by repeatedly dividing the search interval in half. This pattern relies on the property that the array is monotonous, meaning the function we are

trying to optimize (in this case, the sum of the division results) moves in a single direction as we traverse the array.

Binary Search Implementation

in nums, as the sum will certainly be less than or equal to the threshold when divided by the maximum value or any larger number since the division results will be close to 1. The <u>binary search</u> proceeds by computing the midpoint mid = (l + r) >> 1 (this is equivalent to (l + r) / 2, but uses bit shifting for potentially faster computation in some languages). For each element x in nums, we divide x by the current mid and

The search starts with setting up two pointers, 1 and r, which represent the boundaries of the potential divisor values. The left

boundary 1 is initialized to 1 (since we are looking for a positive divisor), while the right boundary r is set to the maximum value

then round up the result to the nearest integer by performing (x + mid - 1) // mid. We take the sum of all these numbers and

compare them to threshold.

can be at most log(max(nums)) such iterations.

smaller one. Hence, we move the right boundary r to mid to narrow down the search towards smaller divisors. On the contrary, if the sum is greater than threshold, mid is too small of a divisor, resulting in a larger sum. Thus, we move the left boundary 1 to mid + 1 to search among larger divisors.

This search process loops until 1 becomes equal to r, meaning that the space for search has been narrowed down to a single

element which is the smallest possible divisor to meet the condition. As soon as this condition is met, the implementation returns

If the sum is less than or equal to threshold, then mid can be a valid candidate for the smallest divisor, but there might be a

1 as the solution. The patterns and data structures used in this solution are fairly simple—a single array to iterate through the integers of nums, and integer variables to manage the binary search boundaries and the running sum. This results in an efficient solution that requires

only O(log(max(nums)) * len(nums)) operations, as each binary search iteration involves summing len(nums) elements, and there

Suppose we have nums = [4, 9, 8] and threshold = 6. **Initial Setup** We initialize our binary search bounds: 1 starts at 1 because we want a positive divisor, and r starts at 9, which is the maximum

• l = 1, r = 9• mid = (l + r) // 2. Here, (1 + 9) // 2 = 5.

∘ For 9: (9 + 5 − 1) // 5 yields 3.

∘ For 8: (8 + 5 − 1) // 5 yields 2.

• mid = (l + r) // 2. Here, (1 + 5) // 2 = 3.

 \circ For 4: (4 + 3 - 1) // 3 yields 2.

∘ For 9: (9 + 3 - 1) // 3 yields 4.

∘ For 8: (8 + 4 − 1) // 4 yields 3.

Therefore, l = mid + 1, making l = 5.

• The sum is 1 + 3 + 3 = 7, which is greater than threshold.

First iteration:

Example Walkthrough

 We now divide each element of nums by mid and round up: \circ For 4: (4 + 5 - 1) // 5 yields 1.

• The sum of rounded divisions is 1 + 3 + 2 = 6 which is equal to threshold.

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

```
• Hence, we can still try to find a smaller divisor. So we move the right boundary r to mid, so r = 5.
```

value in <a hreshold.

```
∘ For 8: (8 + 3 − 1) // 3 yields 3.

 The sum is 2 + 4 + 3 = 9, which is greater than threshold.
```

Second iteration:

• l = 1, r = 5

• Divide and round up:

Third iteration: • l = 4, r = 5• mid = (l + r) // 2. Here, (4 + 5) // 2 = 4. Divide and round up: ○ For 4: (4 + 4 - 1) // 4 yields 1. ∘ For 9: (9 + 4 − 1) // 4 yields 3.

• At this point, l = 5 and r = 5, so we have reached our single point where l and r are equal.

• Because the sum exceeded the threshold, we adjust our left boundary to mid + 1, which makes l = 4.

• The loop terminates as the space for search has been narrowed down to a single element. We conclude that the smallest divisor given nums = [4, 9, 8] and threshold = 6 is 5 since any smaller divisor would result in a

Python

Java

class Solution {

threshold.

class Solution:

Solution Implementation

left, right = 1, max(nums)

right = mid

int left = 1, right = 1000000;

for (int num : nums) {

for (int num : nums) {

if (sum <= threshold) {</pre>

// when left and right converge

} else {

return left;

sum += (num + mid - 1) / mid;

// This function finds the smallest divisor such that the sum of each number

function smallestDivisor(nums: number[], threshold: number): number {

// Calculate the middle value of the current search space.

// Initialize the left bound of the search space to 1.

// can't be larger than the largest number in the array.

const mid = Math.floor((left + right) / 2);

// in the array divided by this divisor is less than or equal to the threshold.

// Initialize the right bound to the maximum number in the array, as the divisor

while (left < right) {</pre>

int sum = 0;

while left < right:</pre>

else:

from typing import List

Fourth iteration:

mid = (left + right) // 2# Calculate the sum using the current divisor (mid) # The sum is of ceil(x / mid) for each x in nums, implemented as (x + mid - 1) // mid to use integer division if sum((num + mid - 1) // mid for num in nums) <= threshold:</pre>

// Main method to find the smallest divisor given an integer array and a threshold

// Initialize the search range: lower bound 'left' and upper bound 'right'

int mid = (left + right) >> 1; // Equivalent to (left + right) / 2

sum += (num + mid - 1) / mid; // `(num + mid - 1) / mid` rounds up

// Calculate the sum of the divided numbers which are rounded up

public int smallestDivisor(int[] nums, int threshold) -

// Perform binary search for the optimal divisor

Use binary search to find the smallest divisor within the search space

Calculate the middle point of the current search space

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

```
# If the sum is greater than the threshold, we need to increase the divisor
        left = mid + 1
# The left variable will be the smallest divisor at the end of the loop, return it
return left
```

If the sum is less than or equal to the threshold, we can try to find a smaller divisor

Define the search space for the smallest divisor, which starts from 1 to the max of the input list

sum greater than the threshold, and it is the smallest divisor for which the division results, when summed up, are equal to

```
// Compare the sum with the threshold to adjust the search range
            if (sum <= threshold) {</pre>
                // If the sum is less than or equal to threshold, narrow the upper bound
                right = mid;
            } else {
                // If the sum exceeds threshold, narrow the lower bound
                left = mid + 1;
        // The left boundary of the search range is the smallest divisor
        return left;
C++
#include <vector>
#include <algorithm> // Needed for std::max_element
class Solution {
public:
    int smallestDivisor(std::vector<int>& nums. int threshold) {
        // Initialize the lower bound of divisor search range to 1
        int left = 1;
        // Initialize the upper bound of divisor search range to the maximum value in the input nums array
        int right = *std::max_element(nums.begin(), nums.end());
        // Conduct a binary search within the range to find the smallest divisor
        while (left < right) {</pre>
```

int mid = (left + right) >> 1; // This finds the middle point by averaging left and right

// The division operation here takes the ceiling to ensure we don't underestimate

left = mid + 1; // Sum too high, increase the divisor by adjusting the lower bound

right = mid; // Possible divisor found, we can discard the upper half of the search space

// Loop through all numbers in the array and divide them by the current divisor 'mid'

// If the sum is less than or equal to the threshold, we adjust the right bound

// The left bound is now the smallest divisor that meets the condition as while loop exists

int sum = 0; // Initialize the sum to store cumulative division results

// Perform a binary search to find the smallest divisor. while (left < right) {</pre>

};

TypeScript

let left = 1;

let right = Math.max(...nums);

```
// Initialize the sum of divided elements.
        let sum = 0;
        // Calculate the division sum of the array by the current mid value.
        for (const num of nums) {
            sum += Math.ceil(num / mid); // Ceil to ensure the division result is an integer.
       // If the current sum is within the threshold, try to find a smaller divisor,
        // so narrow the right bound of the search space.
        if (sum <= threshold) {</pre>
            right = mid;
        } else {
           // If the current sum exceeds the threshold, a larger divisor is needed,
            // so narrow the left bound of the search space.
            left = mid + 1;
   // The left bound becomes the smallest divisor that satisfies the condition at
   // the end of the loop, as the loop keeps narrowing the search space.
   return left;
from typing import List
class Solution:
   def smallestDivisor(self, nums: List[int], threshold: int) -> int:
       # Define the search space for the smallest divisor, which starts from 1 to the max of the input list
        left, right = 1, max(nums)
       # Use binary search to find the smallest divisor within the search space
       while left < right:</pre>
           # Calculate the middle point of the current search space
           mid = (left + right) // 2
           # Calculate the sum using the current divisor (mid)
           # The sum is of ceil(x / mid) for each x in nums, implemented as (x + mid - 1) // mid to use integer division
            if sum((num + mid - 1) // mid for num in nums) <= threshold:</pre>
                # If the sum is less than or equal to the threshold, we can try to find a smaller divisor
                right = mid
           else:
                # If the sum is greater than the threshold, we need to increase the divisor
```

Time Complexity

return left

left = mid + 1

Time and Space Complexity

The given algorithm utilizes a binary search, and within each iteration of this search, it computes the sum of the quotients obtained by dividing each number in the nums list by the current divisor (mid). The binary search varies the divisor mid from 1 to max(nums), shrinking this range by half with each iteration until it converges to the smallest possible divisor that satisfies the condition. Since we're halving the range in which we're searching, the binary search has a logarithmic complexity relative to the range of possible divisors. The complexity of the binary search part can be expressed as O(log M), where M is the maximum value in nums. For each iteration of binary search, we perform a sum operation which involves iterating over all n elements in nums, calculating

The left variable will be the smallest divisor at the end of the loop, return it

number of elements, which is O(n). Combining the two parts, the total computational complexity is the product of the complexity of the binary search and the complexity of the sum operation performed at each step. Therefore, the time complexity of the entire algorithm is 0(n log M).

the quotient for each element, and summing those quotients up. This process has a linear time complexity concerning the

Space Complexity

The space complexity of the algorithm refers to the amount of additional memory space that the algorithm requires aside from the input itself. This algorithm uses a constant amount of extra space for variables such as 1, r, mid, and the space used for the running total in the sum operation. Since this does not grow with the size of the input list nums, the space complexity is 0(1).