1283. Find the Smallest Divisor Given a Threshold

Medium Array Binary Search

Leetcode Link

Problem Description

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

Intuition

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 `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 (`l`) 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 (`l == r`), we can return `l` 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

The search starts with setting up two pointers, `l` and `r`, which represent the boundaries of the potential divisor values. The left boundary `l` is initialized to 1 (since we are looking for a positive divisor), while the right boundary `r` is set to the maximum value 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 binary search 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 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`.

- 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 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 `l` to `mid + 1` to search among larger divisors.

This search process loops until `l` 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 `l` 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 can be at most `log(max(nums))` such iterations.

Example Walkthrough

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

Suppose we have `nums = [4, 9, 8]` and `threshold = 6`.

Initial Setup

We initialize our binary search bounds: `l` starts at 1 because we want a positive divisor, and `r` starts at 9, which is the maximum value in `nums`, since it is the largest the divisor can be to still produce a sum `<= threshold`.

First iteration:

- `l = 1`, `r = 9`
- `mid = (l + r) // 2`. Here, `(1 + 9) // 2 = 5`.
- We now divide each element of `nums` by `mid` and round up:
  - For `4`: `(4 + 5 - 1) // 5` yields `1`.
  - For `9`: `(9 + 5 - 1) // 5` yields `3`.
  - For `8`: `(8 + 5 - 1) // 5` yields `2`.
- The sum of rounded divisions is `1 + 3 + 2 = 6` which is equal to `threshold`.
- Hence, we can still try to find a smaller divisor. So we move the right boundary `r` to `mid`, so `r = 5`.

Second iteration:

- `l = 1`, `r = 5`
- `mid = (l + r) // 2`. Here, `(1 + 5) // 2 = 3`.
- Divide and round up:
  - For `4`: `(4 + 3 - 1) // 3` yields `2`.
  - For `9`: `(9 + 3 - 1) // 3` yields `4`.
  - For `8`: `(8 + 3 - 1) // 3` yields `3`.
- The sum is `2 + 4 + 3 = 9`, which is greater than `threshold`.
- Because the sum exceeded the `threshold`, we adjust our left boundary to `mid + 1`, which makes `l = 4`.

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`.
  - For `8`: `(8 + 4 - 1) // 4` yields `3`.
- The sum is `1 + 3 + 3 = 7`, which is greater than `threshold`.
- Therefore, `l = mid + 1`, making `l = 5`.

Fourth iteration:

- At this point, `l = 5` and `r = 5`, so we have reached our single point where `l` and `r` are equal.
- 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 sum greater than the threshold, and it is the smallest divisor for which the division results, when summed up, are equal to `threshold`.

Python Solution

```python
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:
            # 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:
                # 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
                left = mid + 1
        # The left variable will be the smallest divisor at the end of the loop, return it
        return left
```

Java Solution

```java
class Solution {

    // Main method to find the smallest divisor given an integer array and a threshold
    public int smallestDivisor(int[] nums, int threshold) {
        // Initialize the search range: lower bound 'left' and upper bound 'right'
        int left = 1, right = 1000000;

        // Perform binary search for the optimal divisor
        while (left < right) {
            int mid = (left + right) >> 1; // Equivalent to (left + right) / 2

            // Calculate the sum of the divided numbers which are rounded up
            int sum = 0;
            for (int num : nums) {
                sum += (num + mid - 1) / mid; // `(num + mid - 1) / mid` rounds up
            }

            // Compare the sum with the threshold to adjust the search range
            if (sum <= threshold) {
                // 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++ Solution

```cpp
#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) {
            int mid = (left + right) >> 1; // This finds the middle point by averaging left and right
            int sum = 0; // Initialize the sum to store cumulative division results

            // Loop through all numbers in the array and divide them by the current divisor 'mid'
            for (int num : nums) {
                // The division operation here takes the ceiling to ensure we don't underestimate
                sum += (num + mid - 1) / mid;
            }

            // If the sum is less than or equal to the threshold, we adjust the right bound
            if (sum <= threshold) {
                right = mid; // Possible divisor found, we can discard the upper half of the search space
            } else {
                left = mid + 1; // Sum too high, increase the divisor by adjusting the lower bound
            }
        }
        // The left bound is now the smallest divisor that meets the condition as while loop exists
        // when left and right converge
        return left;
    }
};
```

Typescript Solution

```typescript
// This function finds the smallest divisor such that the sum of each number
// in the array divided by this divisor is less than or equal to the threshold.
function smallestDivisor(nums: number[], threshold: number): number {
    // Initialize the left bound of the search space to 1.
    let left = 1;
    // Initialize the right bound to the maximum number in the array, as the divisor
    // can't be larger than the largest number in the array.
    let right = Math.max(...nums);

    // Perform a binary search to find the smallest divisor.
    while (left < right) {
        // Calculate the middle value of the current search space.
        const mid = Math.floor((left + right) / 2);

        // 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) {
            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;
}
```

Time and Space Complexity

Time 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 quotient for each element, and summing those quotients up. This process has a linear time complexity concerning the 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 $O(n \log M)$.

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 `l`, `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 $O(1)$.