2064. Minimized Maximum of Products Distributed to Any Store

Medium Array Binary Search Leetcode Link

Problem Description

The problem presents a scenario where there is a given number of specialty retail stores, denoted by an integer `n`, and a certain number of product types, denoted by an array `quantities` where each entry `quantities[i]` represents the amount available for the `i-th` product type. The objective is to distribute these products to the stores following certain rules:

- Each store can receive at most one product type.
- A store can receive any amount of the one product type it is given.
- The goal is to minimize the maximum number of products given to any single store.

This situation is akin to finding the most balanced way of distributing products such that the store with the largest stock has as little product as possible, in order to minimize potential waste or overstock.

In simple terms, you are asked to find the smallest number `x` which represents the maximum number of products that any store will have after the distribution is completed.

Intuition

The solution involves using a binary search algorithm. The key insight here is that if we can determine a minimum possible `x` such that all products can be distributed without exceeding `x` products in any store, we have our answer.

Here's the reasoning process:

- We know that the value of `x` must be between 1 (if there's at least one store for each product) and the maximum quantity in `quantities` (if there's only one store).
- Using binary search, we can quickly narrow down the range to find the smallest possible `x`.
- To test if a given `x` is valid, we check if all products can be distributed to stores without any store getting more than `x` products. This is done by calculating the number of stores needed for each product type (rounded up) and ensuring the sum does not exceed `n`.

The `check` function in the provided code snippet helps with this validation by returning `True` if a given `x` allows for all the products to be distributed within the `n` stores.

The Python `bisect_left` function is used to perform the binary search efficiently. It returns the index of the first element in the given range that matches `True` when passed to the `check` function. This means it effectively finds the lowest number `x` such that `check(x)` is true, which is our desired minimum possible maximum number of products per store (`x`).

By starting the binary search at `1` and going up to `10**6` (an arbitrary high number to ensure the maximum quantity is covered), we guarantee finding the minimum `x` within the valid range.

Solution Approach

The solution adopts a binary search algorithm to efficiently find the minimum value of `x` that satisfies the distribution conditions. Binary search is an optimal choice here because it allows for a significant reduction in the number of guesses needed compared to linear search, especially when there's a large range of possible values for `x`.

Here is the step-by-step implementation of the solution:

1. **Binary Search Algorithm**:
   - The binary search is used to find the minimum `x` such that all products can be distributed according to the rules. It is efficient because it repeatedly divides the search interval in half.
   - In this case, the lower and upper bounds for the search are `1` and `10**6` respectively. The upper bound is a sufficiently large number to cover the maximum possible quantity.
2. **Helper Function (`check`)**:
   - A helper function, `check`, is used during the binary search to validate whether a given `x` can be used to distribute all the products within the available stores.
   - For each product type in `quantities`, the function calculates the number of stores required by dividing the quantity of that product type by `x` and rounding up (since we can't have a fraction of a store). This is done using `(v + x - 1) // x` for each `v` in `quantities`. The rounding up is achieved by adding `x - 1` before performing integer division.
3. **Using `bisect_left`**:
   - The Python function `bisect_left` from the `bisect` module is employed to perform the binary search. The function takes in a range, a target value (`True` in this case, as `check` returns a boolean), and a key function (`check`). The key function is called with each mid-value during the search to determine if `x` is too high or too low.
   - `bisect_left` will find the position `i` to insert the target value (`True`) in the range in order to maintain the sorted order. Since the key function effectively turns the range into a sorted boolean array (`False` for values below our target and `True` for values at or above it), the position `i` represents the smallest `x` for which `check(x)` is `True`.
4. **Outcome**:
   - The value of `x` found by the `bisect_left` search is incremented by 1 because the range is 0-indexed, whereas the quantities need to be 1-indexed (since `x` can't be zero).

This approach of binary search combined with a check for the validity of the distribution ensures that the solution is both efficient and correct, taking `O(log M * N)` time complexity, where `M` is the range of possible values for `x` and `N` is the length of `quantities`.

Example Walkthrough

Let's consider an example where we have `n = 2` specialty retail stores and `quantities = [3, 6, 14]` for the product types.

1. **Binary Search Initialization**:
   - The possible values for `x` start with a lower bound of `1`, and an upper bound of `10**6`.
2. **First Mid-point Check**:
   - Let's pick a mid-point for `x` during the binary search. Since our bounds are `1` and `10**6`, a computationally reasonable mid-point to start could be `500,000`. However, for the sake of example, we'll use `5` as it's a more illustrative number.
   - We run the `check` function with `x = 5`. We need one store for the first product type (3/5 rounded up is still 1 store), two stores for the second product type (6/5 rounded up is 2 stores), and three stores for the third product type (14/5 rounded up is 3 stores).
   - In total, we would need `1 + 2 + 3 = 6` stores to distribute the products with each store receiving no more than 5 products. However, we only have `2` stores.
   - Since we can't distribute the products without exceeding `5` products in any store with only `2` stores, `check(5)` returns `False`.
3. **Adjusting the Search Range**:
   - Since `5` is too small, we adjust our search range. The next mid-point we check is halfway between `5` and `10**6`, but again as this is an example, we choose `7`.
   - We run the `check` function with `x = 7`. This time we would need `1` store for the first product type (3/7 rounded up is still 1 store), `1` store for the second product type (6/7 rounded up is still 1 store), and `2` stores for the third product type (14/7 is exactly 2 stores), totaling `1 + 1 + 2 = 4` stores. This is still more than `2` stores, so `check(7)` returns `False`.
4. **Finding the Valid `x`**:
   - Continuing the binary search, let's try `x = 10`.
   - Running `check(10)`, the first product type requires `1` store (3/10 rounded up is still 1 store), the second product type also requires `1` store (6/10 rounded up is still 1 store), and the third product type requires `2` stores (14/10 rounded up is 2 stores). The total is `1 + 1 + 2 = 4` stores, which is still too many.
   - However, if we check `x = 15`, we see that each product type would only require `1` store: (3/15, 6/15, 14/15 all round up to 1 since we can't have a fraction of a store). The total is `3` stores, which is still more than `2`, so `check(15)` returns `False`.
5. **Binary Search Conclusion**:
   - Finally, we try `x = 20`.
   - Now running `check(20)`, each product type will require only `1` store: (3/20, 6/20, 14/20 all round up to 1). The sum is exactly `2` stores, which matches `n = 2`.
   - Since `check(20)` returns `True` and we cannot go lower than `20` without needing more stores than we have, `20` is our smallest possible `x`.

In practice, the binary search would proceed by narrowing down the range between the values where `check` returns `False` and where `check` returns `True`. In our manually stepped-through example, `x = 20` is the solution. This means that we can distribute the products to the 2 stores in such a way that no store has more than 20 products, thereby minimizing the maximum number of products per store.

Python Solution

```python
from bisect import bisect_left
from typing import List

class Solution:
    def minimizedMaximum(self, n: int, quantities: List[int]) -> int:
        # Define a check function that will be used to determine if a specific value of 'x'
        # allows distributing all the quantities within 'n' stores such that the maximum
        # quantity in any store does not exceed 'x'.
        def is_distribution_possible(x):
            # Calculate the total number of stores required if each store can hold up to 'x'
            # items. The expression (quantity + x - 1) // x is used to ceiling divide the quantity
            # by x to find out how many stores are required for each quantity.
            total_stores_needed = sum((quantity + x - 1) // x for quantity in quantities)
            # Check if the total number of stores needed does not exceed the available 'n' stores.
            return total_stores_needed <= n

        # Use binary search (bisect_left) to find the smallest 'x' such that distributing
        # the items does not exceed the number of stores. We search in the range from 1 to 10**6
        # as an upper limit, assuming that the maximum quantity per store will not exceed 10**6.
        # The second argument to bisect_left is 'True' because we are interested in finding the point
        # where the 'is_distribution_possible' function returns 'True'.
        min_max_quantity = 1 + bisect_left(range(1, 10**6), True, key=is_distribution_possible)

        # Return the smallest possible maximum quantity that can be put in a store.
        return min_max_quantity

```

Java Solution

```java
class Solution {
    public int minimizedMaximum(int stores, int[] products) {
        // Initial search space: the lowest possible maximum is 1 (each store can have at least one of any product),
        // and the highest possible maximum is assumed to be 100000
        // (based on the problem constraints if given in the problem description).
        int left = 1, right = 100000;

        // Using binary search to find the minimized maximum value
        while (left < right) {
            // Midpoint of the current search space
            int mid = (left + right) / 2;

            // Counter for the number of stores needed
            int count = 0;

            // Distribute products among stores
            for (int quantity : products) {
                // Each store can take 'mid' amount, calculate how many stores are required
                // for this particular product, rounding up
                count += (quantity + mid - 1) / mid;
            }

            // If we can distribute all products to 'stores' or less with 'mid' maximum product per store,
            // we are possibly too high in the product capacity (or just right) so we try a lower capacity
            if (count <= stores) {
                right = mid;
            } else {
                // If we are too low and need more than 'stores' to distribute all products,
                // we need to increase the product capacity per store
                left = mid + 1;
            }
        }

        // 'left' will be our minimized maximum product per store that fits all products into 'stores' stores.
        return left;
    }
}

```

C++ Solution

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Method to find the minimized maximum number of products per shelf
    int minimizedMaximum(int n, vector<int>& quantities) {
        int left = 1; // Start with the minimum possible value per shelf
        int right = 1e5; // Assume an upper bound for the maximum value per shelf

        // Use binary search
        while (left < right) {
            int mid = (left + right) >> 1; // Calculate mid value
            int count = 0; // Initialize count of shelves needed

            // Iterate through the quantities array
            for (int& quantity : quantities) {
                // Calculate and add number of shelves needed for each quantity
                count += (quantity + mid - 1) / mid;
            }

            // If the count of shelves needed is less than or equal to the available shelves
            // there might be a solution with a smaller max quantity, search left side
            if (count <= n) {
                right = mid;
            }
            // Otherwise, search the right side with larger quantities
            else {
                left = mid + 1;
            }
        }
        // When left meets right, it's the minimized max quantity per shelf
        return left;
    }
};

```

Typescript Solution

```typescript
function minimizedMaximum(stores: number, products: number[]): number {
    // Define the search interval with a sensible start and end.
    let searchStart = 1;
    let searchEnd = 1e5; // Assuming 1e5 is the maximum possible quantity.

    // Binary search to find the minimized maximum number of products per store.
    while (searchStart < searchEnd) {
        // Calculate the middle value of the current search interval.
        const middle = Math.floor((searchStart + searchEnd) / 2);

        // Counter to keep track of the total number of stores needed.
        let storeCount = 0;

        // Iterating over each product quantity to distribute among stores.
        for (const quantity of products) {
            // Increment the store count by the number of stores needed for this product.
            // We take the ceiling to account for incomplete partitions.
            storeCount += Math.ceil(quantity / middle);
        }

        // If the store count fits within the number of available stores,
        // we proceed to check if there's an even smaller maximum.
        if (storeCount <= stores) {
            searchEnd = middle;
        } else {
            // Else we need to increase the number of products per store and keep looking.
            searchStart = middle + 1;
        }
    }

    // The minimized maximum number of products per store that will fit.
    return searchStart;
}

```

Time and Space Complexity

The given Python code aims to find a value of `x` that, when used to distribute the quantities of items amongst `n` stores, results in a minimized maximum number within a store while ensuring all quantities are distributed. This is achieved by using a binary search via `bisect_left` on a range of possible values for `x`.

Time Complexity:

The binary search is performed on a range from `1` to a constant value (`10**6`), resulting in `O(log(C))` complexity, where `C` is the upper limit of the search range. Inside the `check` function, there is a loop which computes the sum with complexity `O(Q)` for each check, where `Q` is the length of the `quantities` list. Therefore, the time complexity of the entire algorithm is `O(Q * log(C))`.

Space Complexity:

The code uses a constant amount of additional memory outside of the `quantities` list input. The `check` function computes the sum using the values in `quantities` without additional data storage that depends on the size of `quantities` or the range of values. Hence, the space complexity is `O(1)`, as no significant additional space is consumed in relation to the input size.