1760. Minimum Limit of Balls in a Bag

Medium · Array · Binary Search

Problem Description

Imagine you have a collection of bags, and each bag is filled with a certain number of balls. These bags are represented by an array, where each element of the array corresponds to the number of balls in a bag, for example, `nums[i]` is the number of balls in the `i`-th bag.

You're given a specific number of operations that you can perform, denoted by `maxOperations`. In a single operation, you can choose any one of your bags and split its contents into two new bags. The new bags must contain a positive number of balls, meaning each bag must have at least one ball.

The "penalty" is defined as the largest number of balls in any single bag. The goal is to minimize this penalty at the end of all your operations.

For example, if you start with a bag of 5 balls, you could split it into bags containing 3 and 2 balls, respectively. If this is your only bag, your initial penalty is 5 (since it's the only bag), but after the operation, it's reduced to 3.

You need to determine the minimum possible penalty you can achieve after performing at most `maxOperations` operations.

Intuition

To find the minimum possible penalty, we can utilize a binary search approach. The binary search targets the potential penalties rather than directly searching through the elements of the `nums` array.

Firstly, we need to establish the search range for the possible penalties. The lower bound is 1, since we cannot have bags with zero balls, and the maximum possible penalty is the largest number of balls in a bag from the input array, `max(nums)`.

During each step of the binary search, we check if a proposed penalty (midpoint of the range) can be achieved with at most `maxOperations` operations. To check this, we compute the number of operations required to ensure that no bag has more balls than the proposed penalty. For each bag, the number of necessary operations is the number of times we have to split the bag so that each resulting bag has a number of balls less than or equal to the proposed penalty.

If we can achieve the proposed penalty with `maxOperations` or fewer operations, it means we could possibly do better and thus should search the lower half (reduce the penalty). If not, we need to look for a solution in the upper half (increasing the penalty).

The solution provided uses the `bisect_left` function to perform the binary search and the `check` function as a custom condition to decide the direction of the search. The search ends when the `bisect_left` finds the minimum penalty that satisfies the condition specified by the `check` function.

Solution Approach

The solution to this problem makes use of a classic algorithmic pattern known as binary search. Binary search is a divide-and-conquer algorithm that quickly locates an item in a sorted list by repeatedly dividing the search interval in half.

Here's the step-by-step solution approach:

1. **Define a Check Function**: We need a function, `check(mx)`, that returns `True` if we can make sure that all bags have at most `mx` balls using no more than `maxOperations` operations. This function calculates the number of operations needed to reduce the number of balls in each bag to `mx` or less. It does this by taking each count of balls `x` in `nums` and dividing it by `mx` (after subtracting 1 to avoid an off-by-one error), summing these values up for all bags and comparing the result to `maxOperations`.

2. **Binary Search**: We then search for the smallest integer within the range of `1` to `max(nums)` that can serve as our potential minimum penalty. Within `bisect_left`, which is the binary search function provided by Python, we use the `check` function as a key for the binary search, thus guiding the direction of our search:

   - If `check` returns `True` for a proposed penalty `mx`, it means that it is possible to achieve this penalty with `maxOperations` operations or fewer, and we should continue searching towards a smaller penalty.
   - If `check` returns `False`, we have to move towards a larger penalty.
3. **Execute the Binary Search with `bisect_left`**: The `bisect_left` function is called with three arguments:

   - The range `range(1, max(nums))`, which is our search space for the penalty.
   - The value `True` which we're trying to find, meaning where our check function results in `True`.
   - The `check` function itself, which takes the place of the `key` argument, allowing `bisect_left` to use the `check` function's return value to decide on the search direction.
4. **Return the Result**: Finally, after the binary search is conducted, we add 1 to the result because `bisect_left` returns the position where `True` can be inserted to maintain sorted order, but since our range starts at 1 (not 0), we need to adjust for the zero-based index returned by `bisect_left`.

The actual code implementation is as follows:

```python
def minimumSize(self, nums: List[int], maxOperations: int) -> int:
    def check(mx: int) -> bool:
        return sum((x - 1) // mx for x in nums) <= maxOperations

    return bisect_left(range(1, max(nums)), True, key=check) + 1
```

In this code:

- `minimumSize` is the function that takes `nums` and `maxOperations` as input and returns the minimum penalty.
- The `check` function is nested inside the `minimumSize` function and is used to determine whether a given maximum number of balls per bag (`mx`) is achievable under the operation constraints.
- `bisect_left` performs the binary search and finds the optimal penalty while minimizing the number of operations used.

This code effectively uses binary search to navigate the potential solution space and efficiently arrives at the minimum possible penalty.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach for the problem described. Suppose we have the array of bags `nums = [9, 7, 8]`, and we're allowed `maxOperations = 2`.

The problem is asking for the minimal penalty after performing at most 2 operations, with the penalty being defined as the largest number of balls in any single bag.

Step-by-Step Solution Approach:

1. **Define a Check Function**: Our `check` function receives an integer `mx` and calculates whether we can ensure all bags have `mx` or fewer balls using up to `maxOperations`.

2. **Binary Search**: We use binary search to find the smallest penalty. We start with a lower bound of `1` (bag can't have less than 1 ball) and upper bound `max(nums)` which is `9` in this case.

3. **Execute the Binary Search with `bisect_left`**: We search for the boundary where our `check` function starts to return `True`.

4. **Return the Result**: Once we find the boundary, it represents the minimum possible penalty we can achieve which is compliant with our operation constraints.

Walkthrough on the Example:

- Lower bound: `1`
- Upper bound: `9`
- Search space: `[1, 2, 3, ... 8, 9]`

Now we check with binary search:

- Let's check with `mx = 5`. This would require dividing the bag with 9 balls into two bags `[5, 4]` requiring 1 operation, the bag with 7 balls into two bags `[5, 2]`, requiring 1 operation, and no operations for the bag with 8 balls as it already satisfies the condition. In total, we use `2` operations which are equal to `maxOperations`.

  The result of the `check` function is `True` because we can achieve this with `2` operations. But since we might minimize the penalty further, we continue.

- Narrow the search and check `mx = 4`. With this attempted penalty, we would need to split the bag with `9` balls into `[5,4]` and `[1,4]`, which uses `2` operations and thus exceeds `maxOperations`.

  The result of `check` is `False` because we've exceeded the allowed number of operations. So, we've gone too far, and the penalty must be higher than `4`.

Between `4` and `5`, our binary search would choose `5` as the lower `True` value in the search space, which can now be considered our best possible penalty.

Final Output:

With a `maxOperations` of `2`, the minimum possible penalty we can achieve is `5` as it's the lowest penalty value that returns a `True` value in our `check` function without exceeding the allowed number of operations. Thus, the `minimumSize` function would return `5 + 1` to adjust for the range starting at `1`, for a result of `6`.

However, in this example, as the penalty of `5` hasn't used all the available operations and is actually possible with the given operations, we do not need to add `1`, and the final output is indeed `5`.

Python Solution

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


class Solution:
    def minimumSize(self, nums: List[int], max_operations: int) -> int:
        # Helper function to check if a given maximum size 'max_size'
        # is feasible within the allowed number of operations
        def is_feasible(max_size: int) -> bool:
            # Calculate the total number of operations required to make all
            # balls in bags less than or equal to 'max_size'
            total_operations = sum((num - 1) // max_size for num in nums)
            # Check if the total number of operations needed is within
            # the maximum allowed operations
            return total_operations <= max_operations

        # Find the smallest maximum size of the bags (leftmost position) that
        # requires an equal or lower number of operations than max_operations.
        # The search range is between 1 and the maximum number in 'nums'
        smallest_max_size = bisect_left(range(1, max(nums) + 1), True, key=is_feasible)

        return smallest_max_size

# Example usage:
# solution = Solution()
# result = solution.minimumSize([9, 77, 63, 22, 92], 6)
# print(result) # Output will be the minimum possible max size of the bags
```

Java Solution

```java
class Solution {
    public int minimumSize(int[] nums, int maxOperations) {
        // Initialize the search boundaries
        int left = 1;
        int right = 0;

        // Find the maximum bag size from the input nums
        for (int num : nums) {
            right = Math.max(right, num);
        }

        // Perform the binary search
        while (left < right) {
            // Find the middle value to test
            int mid = (left + right) >>> 1;

            // Calculate the total number of operations required using 'mid' as a boundary
            long count = 0;
            for (int num : nums) {
                // For each bag, calculate the number of operations needed
                // to ensure the bag size is less than or equal to 'mid'
                count += (num - 1) / mid;
            }

            // If the number of operations is within the allowed maxOperations,
            // we should try a smaller max bag size hence update the right boundary
            if (count <= maxOperations) {
                right = mid;
            } else {
                // Otherwise, we need a larger bag size to reduce the operation count
                // so update the left boundary
                left = mid + 1;
            }
        }

        // When the loop exits, 'left' is the minimum possible largest bag size
        return left;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // Include algorithm header for max_element

class Solution {
public:
    int minimumSize(vector<int>& nums, int maxOperations) {
        // Initialize binary search bounds
        int left = 1;
        int right = *max_element(nums.begin(), nums.end()); // Find maximum value in nums

        // Perform binary search to find the minimum possible size of the largest bag after operations
        while (left < right) {
            int mid = left + (right - left) / 2; // Prevent potential overflow
            long long operationCount = 0; // Store number of operations needed for current bag size

            // Calculate the number of operations needed to reduce bags to size at most 'mid'
            for (int num : nums) {
                operationCount += (num - 1) / mid;
            }

            // If the number of operations is less than or equal to maxOperations, try smaller bag size
            if (operationCount <= maxOperations) {
                right = mid;
            } else {
                // If more operations are needed, increase the bag size
                left = mid + 1;
            }
        }

        // Once left == right, we've found the minimum size of the largest bag
        return left;
    }
};
```

Typescript Solution

```typescript
function minimumSize(nums: number[], maxOperations: number): number {
    let left = 1;
    let right = Math.max(...nums); // Find the maximum number in the array.

    // Use binary search to find the minimum possible largest ball size.
    while (left < right) {
        // Calculate the middle point to test.
        const mid = Math.floor((left + right) / 2);

        // Initialize count of operations needed to reduce all balls to `mid` size or smaller.
        let operationsCount = 0;

        // Iterate over all the ball sizes.
        for (const ballSize of nums) {
            // Calculate the number of operations to reduce current ball size to `mid` or smaller.
            // The operation consists of dividing the ball size by `mid` and rounding down.
            operationsCount += Math.floor((ballSize - 1) / mid);
        }

        // Check if the current `mid` satisfies the maximum operations constraint.
        if (operationsCount <= maxOperations) {
            // If yes, we might have a valid solution; we try smaller `mid` to minimize the largest ball size.
            right = mid;
        } else {
            // Otherwise, `mid` is too small, we increase `mid` to reduce the number of needed operations.
            left = mid + 1;
        }
    }

    // Once the loop ends, the smallest largest ball size is found, which is stored in `left`.
    return left;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by two factors: the computation within the `check` function and the binary search using `bisect_left`. The `check` function is called for each step in the binary search.

Binary Search: The use of `bisect_left` implies a binary search over a range determined by the values in `nums`. Since the range is from `1` to the maximum value in `nums`, this range can be represented as `N`, where `N = max(nums)`. The binary search will therefore take `O(log N)` steps to complete as it narrows down the search range by half with each iteration.

Check Function: The `check` function is called for each step of the binary search and iterates over all elements in the list `nums`. If we have `M` elements in `nums`, each call to `check` is `O(M)` since it potentially goes through the entire list once.

Combining these two factors, the overall time complexity is `O(M * log N)` where `M` is the length of the list `nums` and `N` is the value of the maximum element in `nums`.

Space Complexity

The space complexity of the given code is mainly influenced by the space used to store the input `nums`.

Input Storage: The list `nums` itself takes up `O(M)` space, where `M` is the number of elements in `nums`. **Additional Storage**: The space used for the binary search itself is constant, as it operates on a range and does not allocate additional memory proportional to the length of `nums` or the size of the maximum element in `nums`.

Thus, the overall space complexity is `O(M)`, as the check function and binary search operations do not use additional space that scales with the size of the input.