<https://leetcode.com/problems/minimum-limit-of-balls-in-a-bag/description/>

You are given an integer array nums where the ith bag contains nums[i] balls. You are also given an integer maxOperations.

You can perform the following operation at most maxOperations times:

Take any bag of balls and divide it into two new bags with a positive number of balls.

For example, a bag of 5 balls can become two new bags of 1 and 4 balls, or two new bags of 2 and 3 balls.

Your penalty is the maximum number of balls in a bag. You want to minimize your penalty after the operations.

Return the minimum possible penalty after performing the operations.

**Example 1:**

**Input:** nums = [9], maxOperations = 2

**Output:** 3

**Explanation:**

- Divide the bag with 9 balls into two bags of sizes 6 and 3. [**9**] -> [6,3].

- Divide the bag with 6 balls into two bags of sizes 3 and 3. [**6**,3] -> [3,3,3].

The bag with the most number of balls has 3 balls, so your penalty is 3 and you should return 3.

**Example 2:**

**Input:** nums = [2,4,8,2], maxOperations = 4

**Output:** 2

**Explanation:**

- Divide the bag with 8 balls into two bags of sizes 4 and 4. [2,4,**8**,2] -> [2,4,4,4,2].

- Divide the bag with 4 balls into two bags of sizes 2 and 2. [2,**4**,4,4,2] -> [2,2,2,4,4,2].

- Divide the bag with 4 balls into two bags of sizes 2 and 2. [2,2,2,**4**,4,2] -> [2,2,2,2,2,4,2].

- Divide the bag with 4 balls into two bags of sizes 2 and 2. [2,2,2,2,2,**4**,2] -> [2,2,2,2,2,2,2,2].

The bag with the most number of balls has 2 balls, so your penalty is 2, and you should return 2.

**Constraints:**

1 <= nums.length <= 10^5

1 <= maxOperations, nums[i] <= 10^9

**Attempt 1: 2024-12-09**

**Solution 1: Binary Search + Greedy (10 min)**

**Style 1: canDistribute**

class Solution {

    public int minimumSize(int[] nums, int maxOperations) {

        int lo = 1;

        // The largest possible maximum is max(nums),

        // as no bag can exceed its initial size.

        int hi = 1;

        for(int num : nums) {

            hi = Math.max(hi, num);

        }

        // Find lower boundary (Since requires return the minimum

        // possible penalty after performing the operations)

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            if(canDistribute(nums, maxOperations, mid)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

    // 'penalty' is the maximum number of balls in a bag

    private boolean canDistribute(int[] nums, int maxOperations, int minPenalty) {

        int count = 0;

        for(int balls : nums) {

            // Equivalent to ceil(balls / minPenalty) - 1

            // and ceil(balls / minPenalty) = (balls - 1) / minPenalty + 1

// '- 1' for the divide operation count always one less than

// divided

            count += (balls - 1) / minPenalty;

        }

        return count <= maxOperations;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Style 2: cannotDistribute**

class Solution {

    public int minimumSize(int[] nums, int maxOperations) {

        int lo = 1;

        // The largest possible maximum is max(nums),

        // as no bag can exceed its initial size.

        int hi = 1;

        for(int num : nums) {

            hi = Math.max(hi, num);

        }

        // Find lower boundary (Since requires return the minimum

        // possible penalty after performing the operations)

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            // Since want to find minimum possible penalty after

            // performing the operations, if 'mid' cannot satisfy,

            // we can move 'lo' forward to 'mid + 1' for larger

            // minimum penalty attempt, otherwise, we can move

            // 'hi' backward to 'mid - 1' for smaller minimum

            // penalty attempt

            if(cannotDistribute(nums, maxOperations, mid)) {

                lo = mid + 1;

            } else {

                hi = mid - 1;

            }

        }

        return lo;

    }

    // 'penalty' is the maximum number of balls in a bag

    private boolean cannotDistribute(int[] nums, int maxOperations, int minPenalty) {

        int count = 0;

        for(int balls : nums) {

            // Equivalent to ceil(balls / minPenalty) - 1

            // and ceil(balls / minPenalty) = (balls - 1) / minPenalty + 1

            count += (balls - 1) / minPenalty;

            // If over limited 'maxOperations' not able to distribute,

            // then early terminate

            if(count > maxOperations) {

                return true;

            }

        }

        return false;

    }

}

Time Complexity: O(nlogn)

Space Complexity: O(1)

**Refer to chatGPT**

This is a **binary search on the answer** problem. The task is to minimize the maximum number of balls in a bag after splitting bags.

**Key Idea:**

Binary Search:

The smallest possible maximum is 1.

The largest possible maximum is max⁡(nums), as no bag can exceed its initial size.

Feasibility Check:

For a given maximum size x, calculate how many operations are needed to ensure no bag contains more than x balls.

If the total operations required ≤max, then x is feasible.

class Solution {

public int minimumSize(int[] nums, int maxOperations) {

// Binary search bounds

int left = 1;

int right = Arrays.stream(nums).max().getAsInt();

while (left < right) {

int mid = left + (right - left) / 2;

if (canDistribute(nums, maxOperations, mid)) {

right = mid; // Try for a smaller maximum size

} else {

left = mid + 1; // Increase the maximum size

}

}

return left;

}

private boolean canDistribute(int[] nums, int maxOperations, int maxSize) {

int operations = 0;

for (int balls : nums) {

if (balls > maxSize) {

// Calculate operations needed to reduce this bag to <= maxSize

operations += (balls - 1) / maxSize; // Equivalent to ceil(balls / maxSize) - 1

}

}

return operations <= maxOperations;

}

}

**Explanation:**

Binary Search:

Start with left = 1 and right = max(nums).

Use the mid-point of the current range to test if it’s feasible to reduce all bag sizes to ≤ mid.

canDistribute **Function**:

For each bag, calculate how many splits are required to reduce the number of balls to ≤ maxSize.

Accumulate these operations and return true if the total ≤ maxOperations, otherwise false.

Splitting Logic:

For a bag containing balls, splitting it into sub-bags with ≤ maxSize balls requires:
splits = ⌈ballsmaxSize / maxSize⌉−1

This is implemented as (balls−1) / maxSize to avoid using floating-point arithmetic.

**Refer to**

[L410.Split Array Largest Sum (Ref.L1011,L704,L1482)](note://WEB4d9d6f26a965381b137615128e2648be)

[L774.Minimize Max Distance to Gas Station (Ref.L410,L1011,L1283,L1482)](note://WEB28e37ffbbab2b651254c65af5334111d)

[L1011.Capacity To Ship Packages Within D Days (Ref.L410,L1482)](note://WEB87ca8f9dd9a14fb4535856a4e0f8f4b1)

[L1130.Minimum Cost Tree From Leaf Values (Ref.L739,L503)](note://WEBeaa892931b126d0eee98ffb618834ced)

[L1482.Minimum Number of Days to Make m Bouquets (Ref.L410,L1011)](note://WEBd5ce394160da45a79e92dbbc806111ec)

[L1552.Magnetic Force Between Two Balls (Ref.L1802,L2226)](note://WEB4d72055c8eee671bf55dca2e0a3f951d)

[L1802.Maximum Value at a Given Index in a Bounded Array (Ref.L410)](note://WEB7d060d486a37a8c92696be41f18fb27c)

[L2064.Minimized Maximum of Products Distributed to Any Store (Ref.L410)](note://WEB2ca95ff77a9fee4bcced70f3e62a41df)

[L2226.Maximum Candies Allocated to K Children (Ref.L1802,L1552)](note://WEBbc250dc9fe15c478547298491ba2386f)