<https://leetcode.com/problems/koko-eating-bananas/>

Koko loves to eat bananas. There are n piles of bananas, the ith pile has piles[i] bananas. The guards have gone and will come back in h hours.

Koko can decide her bananas-per-hour eating speed of k. Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return the minimum integer k such that she can eat all the bananas within h hours.

**Example 1:**

Input: piles = [3,6,7,11], h = 8

Output: 4

**Example 2:**

Input: piles = [30,11,23,4,20], h = 5

Output: 30

**Example 3:**

Input: piles = [30,11,23,4,20], h = 6

Output: 23

**Constraints:**

1 <= piles.length <= 10^4

piles.length <= h <= 10^9

1 <= piles[i] <= 10^9

**Attempt 1: 2022-09-14 (30min, difficult on how to integrate into Find Lower Boundary template and handle overflow of integer max value limitation)**

class Solution {

    public int minEatingSpeed(int[] piles, int h) {

        int lo = 1;

        int hi = 0;

        for(int pile : piles) {

            hi = Math.max(hi, pile);

        }

        while(lo <= hi) {

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

            // Tricky point to integrate the condition judgement into a single

            // boolean function which following the nums[mid] >= target style

            // e.g 'h' as the target nested into the function is a breakthrough

            if(countHours(mid, piles, h)) {

                hi = mid - 1;

            } else {

                lo = mid + 1;

            }

        }

        return lo;

    }

    private boolean countHours(int k, int[] piles, int h) {

        int count = 0;

        for(int pile : piles) {

            //count += (pile - 1) / k + 1;

            // Test out by new added testing case:

            // piles=[805306368,805306368,805306368], h=1000000000

            // Since 805306368 \* 3 > Integer.MAX\_VALUE, we cannot

            // simply sum it up with plain adding, needs Math.ceil

            // to work with cased double value (double type will

            // by pass the Integer.MAX\_VALUE limitation)

            count += Math.ceil(pile \* 1.0 / k);

        }

        // Set '<=' exactly for if KoKo's eating hours 'count' less

        // than threshold 'h', KoKo still safe, we can continue relax

        // 'count', which means we can pick a smaller 'k', and 'k' is

        // value assigned by 'mid', 'mid' is derive by 'hi', eventually

        // means we can shrink 'hi' to 'mid - 1', and perfectly match

        // "Find Lower Boundary" template

        // if(nums[mid] >= target) {

        //      hi = mid - 1;

        //  } else {

        //      lo = mid + 1;

        //  }

        return count <= h;

    }

}

Space Complexity: O(1)

Time Complexity: O(nlogn), where n is no of piles & n is range of K (left to right)

**Refer to**

<https://leetcode.com/problems/koko-eating-bananas/discuss/152506/Binary-Search-Java-Python-with-Explanations>

Each hour, Koko chooses some pile of bananas, and eats K bananas from that pile.

There is a limited value range of K: [lo, hi].

There is a K' value, such that K(for any K >= K') can enable her to eat all the bananas within H hours: [K', hi].

We are asked to find K'.

Given a linear searching space [lo, hi], [mi, hi] (lo <= mi) satisfy a property, we can use Binary Searc to get mi.

Initially, we know that K belongs to [1, the largest element in piles[]]. And we follow the pattern of lower-bound Binary Search except that if (K == target) is replaced with if (canEatAll(piles, K, H)).

public int minEatingSpeed(int[] piles, int H) {

int lo = 1, hi = getMaxPile(piles);

// Binary search to find the smallest valid K.

while (lo <= hi) {

int K = lo + ((hi - lo) >> 1);

if (canEatAll(piles, K, H)) {

hi = K - 1;

} else {

lo = K + 1;

}

}

return lo;

}

private boolean canEatAll(int[] piles, int K, int H) {

int countHour = 0; // Hours take to eat all bananas at speed K.

for (int pile : piles) {

countHour += pile / K;

if (pile % K != 0)

countHour++;

}

return countHour <= H;

}

private int getMaxPile(int[] piles) {

int maxPile = Integer.MIN\_VALUE;

for (int pile : piles) {

maxPile = Math.max(pile, maxPile);

}

return maxPile;

}

**Refer to**

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

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

[L1283.Find the Smallest Divisor Given a Threshold (Ref.L410,L1011,L1482)](note://76952362C4844CB3B68E28EA09F2EE58)

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

[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)