875. Koko Eating Bananas

Binary Search

Problem Description

<u>Array</u>

Medium

Koko has h hours to eat all the bananas before the guards return, and she can set a constant eating speed of k bananas per hour. At each hour, she eats k bananas from a pile; if the pile has fewer than k bananas, she'll finish that pile and then stop eating for that hour. The goal is to determine the minimum eating speed k that allows Koko to consume all the bananas within the h hours available. The problem is essentially about finding the lowest possible rate of eating bananas per hour, which allows Koko to eat all bananas

In the given LeetCode problem, Koko loves bananas and has n piles of them, with the i-th pile containing piles[i] bananas.

in the given time without running out of time. The speed k must be an integer, and the challenge is to find the smallest such k that still meets the time constraint.

Intuition

finish all bananas within h hours, and k can range from 1 (eating very slowly) to the maximum number of bananas in any pile in the

worst-case scenario (eating very fast), we can use binary search to narrow down the possible values of k. We start with the lower bound left of 1 (the slowest possible eating speed) and an upper bound right which is set to a high enough value (like 10^9) that it's guaranteed to be higher than the maximum necessary speed.

The solution is based on a binary search approach. Since we're looking for the minimum integer value of k that enables Koko to

At each step of the binary search, we calculate the midpoint mid between left and right. We then calculate the total hours s it

would take for Koko to eat all the bananas at this speed. If s is less than or equal to h, this means mid is a viable eating speed and could possibly be the answer, so we move the right bound to mid, potentially reducing the range. If s is greater than h, then eating at speed mid is too slow and we need to increase the eating speed, so we move the left bound up to mid + 1.

The binary search continues until left and right converge to the minimum possible value of k that allows Koko to eat all bananas in h hours. **Solution Approach**

The reference solution approach suggests employing a binary search algorithm to efficiently find the minimum eating speed k. This approach is chosen due to the nature of the problem, which quite clearly forms a sorted space where the speed can be

increased or decreased based on whether Koko can finish the bananas within the given time h or not.

Here is a detailed walk-through of the algorithm: Initialize two pointers: one for the lower bound left set to 1 (since Koko has to eat at least one banana per hour), and another

require. While left is less than right, we continue the binary search: We find the midpoint mid by averaging the left and right pointers ((left + right) >> 1). Here, the >> 1 operation is a

for the upper bound right set to a large number, such as 10^9, to ensure that it is larger than any real-world scenario would

We then compute the total number of hours s it would take to eat all the piles at the eating speed mid. This is done by iterating over each pile in piles and calculating the hours needed for each pile using (x + mid - 1) // mid, which is a

reduces the number of calculations and, thus, the running time of the algorithm.

We set left = 1 since Koko needs to eat at least one banana per hour.

Since s <= h, we can try a smaller k. So we adjust right = mid, now right = 6.

• The new left and right bounds are left = 1 and right = 6.

Each time, if s <= h, decrease right. If s > h, increase left.

Eventually, left will equal right, giving the minimum speed k.

speed k is 4 bananas per hour. At this rate, Koko will finish exactly on time:

checking each speed sequentially, thus finding the correct k more efficiently.

def minEatingSpeed(self, piles: List[int], hours: int) -> int:

left represents the minimum possible eating speed

Use binary search to find the minimum eating speed

update the upper bound of the search space

Initialize the search space for Koko's possible eating speed

Calculate the middle point of the current search space

total_hours = sum((pile + mid - 1) // mid for pile in piles)

bitwise shift that effectively divides the sum by two.

way to perform ceiling division without using floats to ensure we get an integer result. This formula accounts for the fact that if there are less than mid bananas in a pile, Koko will spend an entire hour eating it (represented by the x + mid - 1 part).

If s is less than or equal to h, we know that mid is at least as fast as the eating speed needs to be, so we adjust the upper

If s is greater than h, the proposed eating speed mid is too slow, and we need to look for a faster eating speed; thus, we

After calculating s, we make a decision based on whether this proposed eating speed mid is fast enough:

bound right to mid. This narrows the search and potentially lowers the eating speed.

move the lower bound left up to mid + 1. The binary search ends when left equals right, at which point left represents the minimum integer eating speed k that allows Koko to eat all bananas within h hours.

The binary search efficiently zeros in on the optimal value without having to try every possible eating speed, which significantly

Let's consider a small example to illustrate the solution approach described above. Assume Koko has 4 piles of bananas where piles = [3, 6, 7, 11], and she has h = 8 hours to eat all of them. We need to determine the minimum integer value of k such

that Koko can finish the bananas within 8 hours. **Initialize Lower and Upper Bounds:**

• We set right = max(piles) = 11 since the minimum speed k cannot be more than the largest pile (Koko could eat any pile in an hour at this

Initially, left = 1, right = 11. ■ Midpoint mid = (left + right) >> 1 = (1 + 11) / 2 = 6.

Adjust Pointers:

• Pile 1 in 1 hour,

Solution Implementation

left, right = 1, int(1e9)

mid = (left + right) >> 1

if total_hours <= hours:</pre>

right = mid

while left < right:</pre>

Binary Search:

speed).

Example Walkthrough

 For mid = 6, we calculate the total hours to eat each pile: ■ Pile 1: (3 + 6 - 1) // 6 = 1 hour. ■ Pile 2: (6 + 6 - 1) // 6 = 1 hour. ■ Pile 3: (7 + 6 - 1) // 6 = 2 hours.

In this example, when mid = 4, the total hours s will be equal to 8 hours, which is exactly the time Koko has. Thus, the minimum

For simplicity, not all steps are shown here, but following this approach, the algorithm investigates fewer possibilities than

```
Repeat Binary Search:
\circ With left = 1 and right = 6, mid = (left + right) >> 1 = (1 + 6) / 2 = 3.5, which rounds down to 3.
```

Continue Until Left equals right:

Repeat the total hours s calculation for mid = 3.

• While left < right, we continue to search:</p>

Calculate Total Hours s at Speed mid:

■ Pile 4: (11 + 6 - 1) // 6 = 2 hours.

 \circ Total hours s = 1 + 1 + 2 + 2 = 6 hours.

• Pile 2 in 2 hours, Pile 3 in 2 hours, and Pile 4 in 3 hours.

• The binary search is repeated, updating left and right based on the calculated hours s until they equal.

Python from typing import List class Solution:

right represents the maximum possible eating speed which we initialize to 1e9

Calculate total hours needed to eat all piles at this eating speed

If Koko can eat all bananas within the given hours at this speed,

Equivalent to (left + right) // 2 but avoids possible overflow in other languages

Otherwise, this speed is too slow, update the lower bound left = mid + 1# The left pointer will be at the minimum eating speed at which Koko can # eat all the bananas within the given hours

return left

return minSpeed;

#include <vector> // Include the vector library

C++

public:

/**

*/

class Solution {

// Parameters:

// Returns:

Java

class Solution {

else:

```
/**
* Finds the minimum eating speed to eat all bananas in 'h' hours.
 * @param piles Array of banana piles.
* @param h Total hours within which all bananas must be eaten.
* @return The minimum integer eating speed.
public int minEatingSpeed(int[] piles, int h) {
   // Initialize the lower bound of the eating speed, cannot be less than 1.
   int minSpeed = 1;
   // Initialize the upper bound of the eating speed to a high number, (int) le9 is used as an approximation.
    int maxSpeed = (int) 1e9;
   // Binary search to find the minimum eating speed.
   while (minSpeed < maxSpeed) {</pre>
        // Find the mid point which is the candidate for our potential eating speed.
        int midSpeed = minSpeed + (maxSpeed - minSpeed) / 2;
        // Initialize the total hours needed with the chosen speed.
        int totalHours = 0;
        // Calculate the total hours needed to eat the piles at midSpeed.
        for (int bananas : piles) {
            // The time to eat a pile is the pile size divided by eating speed, rounded up.
            totalHours += (bananas + midSpeed - 1) / midSpeed;
       // If the total hours with midSpeed is less or equal to h, we might be able to do better,
       // so we bring down the maximum speed to midSpeed.
        if (totalHours <= h) {</pre>
            maxSpeed = midSpeed;
        } else {
            // Otherwise, if we need more than 'h' hours, the speed is too slow.
            // We need to increase our eating speed, so we update minSpeed to midSpeed + 1.
            minSpeed = midSpeed + 1;
   // Loop finishes when minSpeed == maxSpeed, which is the minimum speed to eat all bananas in 'h' hours.
```

```
left = mid + 1;
       // 'left' is now the minimum speed at which Koko can eat all the bananas within 'h' hours.
        return left;
};
TypeScript
```

* @param h Number of hours available to eat all the piles.

function minEatingSpeed(piles: number[], h: number): number {

// We use binary search to find the minimum speed.

while (minSpeed < maxSpeed) {</pre>

left, right = 1, int(1e9)

mid = (left + right) >> 1

if total_hours <= hours:</pre>

left = mid + 1

right = mid

else:

return left

while left < right:</pre>

let minSpeed = 1; // The minimum possible eating speed.

// The function minEatingSpeed calculates the minimum speed at which

// h: The total number of hours within which all bananas must be eaten.

// An integer representing the minimum eating speed (bananas per hour).

// Perform a binary search between the range [left, right]

hoursSpent += (pile + mid - 1) / mid;

// piles: A vector of integers representing the number of bananas in each pile.

int left = 1; // Minimum possible eating speed (cannot be less than 1)

// Calculate the total number of hours it would take at the current speed 'mid'

// (pile + mid - 1) / mid is an efficient way to calculate ceil(pile/mid)

// The number of hours spent on each pile is the ceil of pile/mid

// If the current speed 'mid' requires less than or equal to 'h' hours,

// it is a potential solution and we try to find a smaller one.

// If the current speed 'mid' requires more than 'h' hours,

* Determines the minimum eating speed required to eat all banana piles within `h` hours.

* @return The minimum integer speed at which all bananas can be eaten within `h` hours.

let hoursSpent = 0; // Initialize hours spent to 0 for each iteration.

Initialize the search space for Koko's possible eating speed

Calculate the middle point of the current search space

total_hours = sum((pile + mid - 1) // mid for pile in piles)

right represents the maximum possible eating speed which we initialize to 1e9

Calculate total hours needed to eat all piles at this eating speed

If Koko can eat all bananas within the given hours at this speed,

Otherwise, this speed is too slow, update the lower bound

The left pointer will be at the minimum eating speed at which Koko can

Equivalent to (left + right) // 2 but avoids possible overflow in other languages

left represents the minimum possible eating speed

Use binary search to find the minimum eating speed

update the upper bound of the search space

eat all the bananas within the given hours

* @param piles Array of integers representing the number of bananas in each pile.

// it is not a valid solution and we must try a larger speed.

int right = 1e9; // A large upper bound for the maximum eating speed.

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

int hoursSpent = 0; // Initialize the hours spent to 0

// all bananas in the piles can be eaten within 'h' hours.

int minEatingSpeed(vector<int>& piles, int h) {

while (left < right) {</pre>

for (int pile : piles) {

if (hoursSpent <= h) {</pre>

right = mid;

} else {

```
// Calculate total hours spent eating with the current speed.
          for (const pile of piles) {
              hoursSpent += Math.ceil(pile / midSpeed);
          // If the hours spent is within the allowed hours `h`, we can try to see if there is a smaller speed.
          if (hoursSpent <= h) {</pre>
              maxSpeed = midSpeed;
          } else {
              // If hours spent is more than `h`, we need to increase the speed.
              minSpeed = midSpeed + 1;
      // Return the minimum speed found that enables finishing within `h` hours.
      return minSpeed;
from typing import List
class Solution:
   def minEatingSpeed(self, piles: List[int], hours: int) -> int:
```

let maxSpeed = Math.max(...piles); // The maximum possible eating speed, which cannot exceed the largest pile.

const midSpeed = Math.floor((minSpeed + maxSpeed) / 2); // Calculate the middle speed in the current range.

Time and Space Complexity

The time complexity of the code is determined by the binary search and the computation required to sum up the hours needed to

Time Complexity

eat all the piles at a particular speed. The binary search runs in O(log(max(piles))) because it searches between 1 and max(piles) upper-bounded by 1e9. During each step of the search, we calculate the sum of hours which takes 0(n) where n is the number of piles. Therefore, the overall time complexity of the algorithm is O(n*log(max(piles))). **Space Complexity**

The space complexity of the code is 0(1) as only a constant amount of extra space is used besides the input piles. Variables like left, right, mid, and s occupy constant space and no additional data structures dependent on input size are introduced.