Koko Eating Bananas

15 June 2025 09:28 PM

You are given an array **arr[]** of n integers, where each element represents a pile of bananas. Koko has k hours to finish all the piles, and it's guaranteed that n ≤ k. Your task is to find the **minimum number of bananas per hour**, x, that Koko must eat to finish all the bananas within k hours.

Each hour, Koko can choose **any pile** and eat up to x bananas from it. If a pile has x or fewer bananas, she eats the whole pile in that hour.

Examples:

Input: *arr[] = [5, 10, 3], k = 4*

Output: *5*

Explanation: *If Koko eats at the rate of 5 bananas per hour:*

- *First pile of 5 bananas will be finished in **1** hour.*
- *Second pile of 10 bananas will be finished in **2** hours.*
- *Third pile of 3 bananas will be finished in **1** hours.*
  Therefore, Koko can finish all piles of bananas in 1 + 2 + 1 = 4 hours.

Input: *arr[] = [5, 10, 15, 20], k = 7*

Output: *10*

Explanation: *If Koko eats at the rate of 10 bananas per hour, it will take 6 hours to finish all the piles.*

Try it on GfG Practice

Table of Content

[Naive Approach] Using Linear Search - O(n * m) time and O(1) space

*The idea is to evaluate each possible speed, starting from 1, and calculate the total time needed for Koko to eat all the bananas at that speed. The minimum speed at which the total eating time is less than or equal to **k** hours is our answer. In this way, Koko eats all the bananas within the allowed time at the slowest possible pace.*

```python
def kokoEat(arr, k):
    mx = max(arr)

    for speed in range(1, mx + 1):

        time = 0
        for i in range(len(arr)):

            # Time required to eat this pile
            # of bananas at current speed
            time += arr[i] // speed

            # 1 extra hour to eat the remainder
            # number of bananas in this pile
            if arr[i] % speed != 0:
                time += 1

        # If total eating time at current speed
        # is smaller than given time
        if time <= k:
            return speed

    return mx

if __name__ == "__main__":
    arr = [5, 10, 3]
    k = 4
    print(kokoEat(arr, k))
```

Output

```
5
```

Time Complexity: O(n * m), where **m** is maximum bananas among all piles**.**

Auxiliary Space: O(1)

[Expected Approach] Using Binary Search - O(n * log m) time and O(1) space

The idea is to solve the problem by applying binary search on answer.

- *Lower limit of speed is 1 banana/hr as Koko must eat at least one banana per hour, and Upper limit is the maximum bananas among all piles.*
- *Apply binary search on the possible answer range to get minimum speed to eat all bananas within **k** hours.*
  - *If the current speed (mid) is enough to eat all bananas within given **k** hours then update the minimum eating time and continue the search in lower half of the range to check for slower eating speeds.*
  - *Else search in upper half of the range as we need to increase the eating speed.*

C++JavaPythonC#JavaScript

```
# Function to check whether mid speed is enough
# to eat all piles of bananas under k hours
def check(arr, mid, k):
    hours = 0
    for i in range(len(arr)):
        hours += arr[i] // mid

        # 1 extra hour to eat the remainder
        # number of bananas in this pile
        if arr[i] % mid != 0:
            hours += 1

    # return true if required time is less than
    # or equals to given hour, otherwise return false
    return hours <= k

def kokoEat(arr, k):

    # Minimum speed of eating is 1 banana/hours
    lo = 1

    # Maximum speed of eating is
    # the maximum bananas in given piles
    hi = max(arr)
    res = hi

    while lo <= hi:
        mid = lo + (hi - lo) // 2

        # Check if the mid(hours) is valid
        if check(arr, mid, k) == True:

            # If valid continue to search at
            # lower speed
            hi = mid - 1
            res = mid
        else:

            # If cant finish bananas in given
            # hours, then increase the speed
            lo = mid + 1

    return res

if __name__ == "__main__":
    arr = [5, 10, 3]
    k = 4
    print(kokoEat(arr, k))
```

Output

```
5
```

Time Complexity: O(n log m), where **m** is the max bananas from all piles.

Auxiliary Space: O(1)

Comment
More info
Campus Training Program

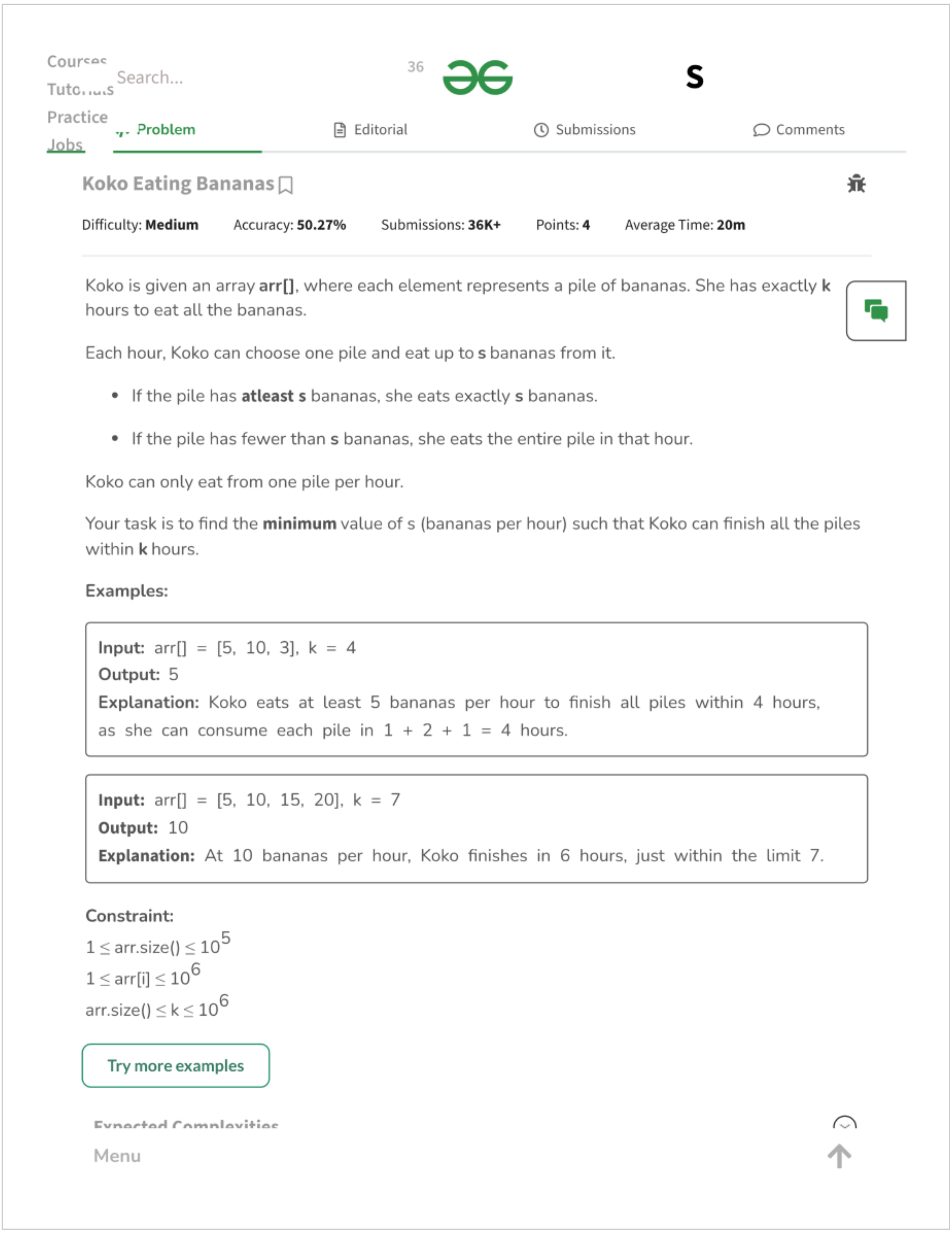
[Next Article
Koko Eating Bananas](Next Article)

Similar Reads

1. Maximum candies two friends can eat
2. Maximizing cakes on a journey
3. Sorting | Natural Language Programming
4. Replace '?' in string with zero or one
5. Distribute N candies among K people
6. Rotten Oranges - Minimum Time to Rot All

7. Number of chocolates left after k iterations
8. BankBazaar Interview Experience | Set 2
9. Length of longest substring having all characters as K
10. Counting k-mers via Suffix Array

From <https://www.geeksforgeeks.org/dsa/koko-eating-bananas/>



Koko Eating Bananas

Difficulty: **Medium** Accuracy: **50.27%** Submissions: **36K+** Points: **4** Average Time: **20m**

Koko is given an array **arr[]**, where each element represents a pile of bananas. She has exactly **k** hours to eat all the bananas.

Each hour, Koko can choose one pile and eat up to **s** bananas from it.

- If the pile has **atleast s** bananas, she eats exactly **s** bananas.
- If the pile has fewer than **s** bananas, she eats the entire pile in that hour.

Koko can only eat from one pile per hour.

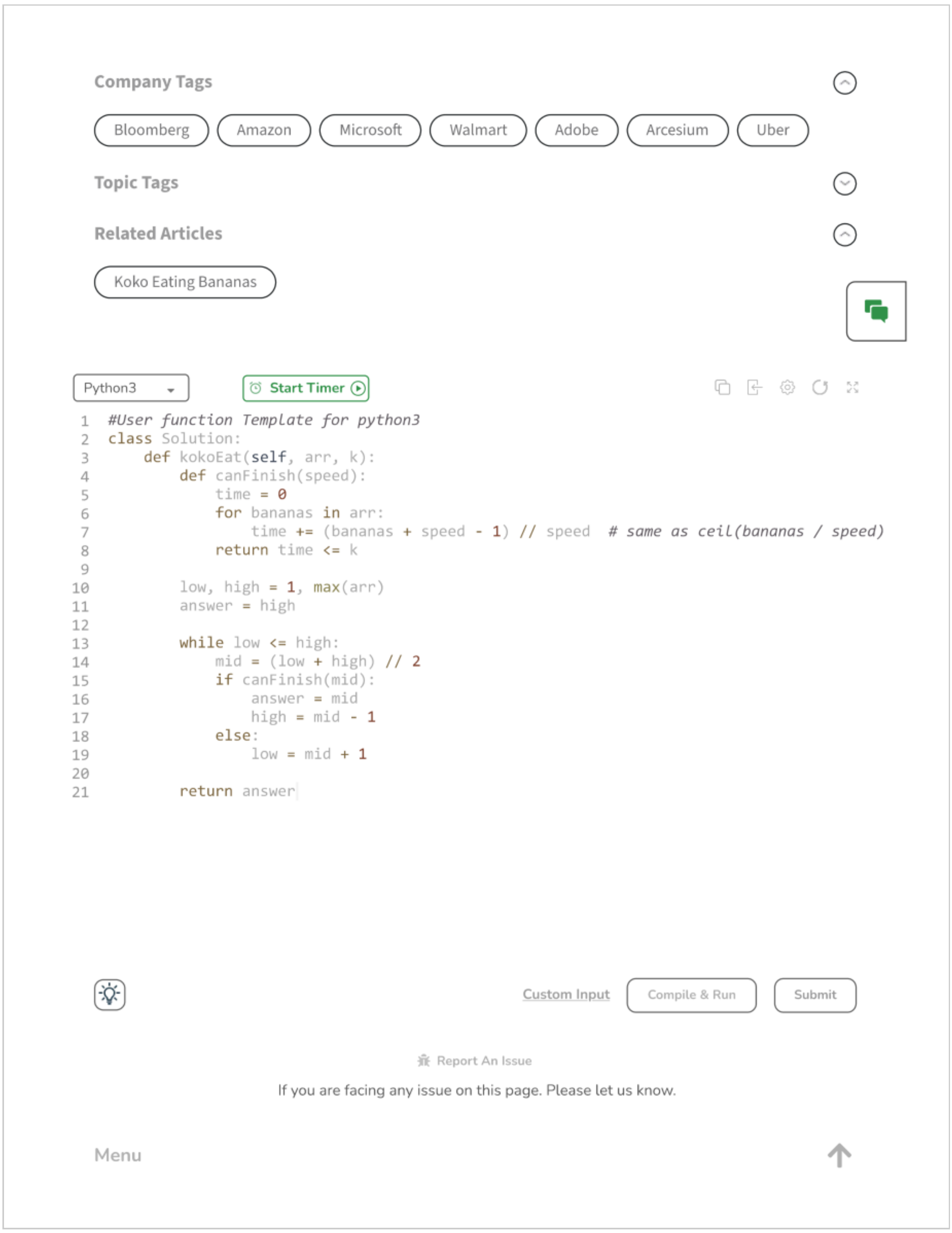
Your task is to find the **minimum** value of s (bananas per hour) such that Koko can finish all the piles within **k** hours.

Examples:

```
Input: arr[] = [5, 10, 3], k = 4
Output: 5
Explanation: Koko eats at least 5 bananas per hour to finish all piles within 4 hours,
as she can consume each pile in 1 + 2 + 1 = 4 hours.
```

```
Input: arr[] = [5, 10, 15, 20], k = 7
Output: 10
Explanation: At 10 bananas per hour, Koko finishes in 6 hours, just within the limit 7.
```

Constraint:
$1 \le arr.size() \le 10^5$
$1 \le arr[i] \le 10^6$
$arr.size() \le k \le 10^6$



```python
#User function Template for python3
class Solution:
    def kokoEat(self, arr, k):
        def canFinish(speed):
            time = 0
            for bananas in arr:
                time += (bananas + speed - 1) // speed  # same as ceil(bananas / speed)
            return time <= k

        low, high = 1, max(arr)
        answer = high

        while low <= high:
            mid = (low + high) // 2
            if canFinish(mid):
                answer = mid
                high = mid - 1
            else:
                low = mid + 1

        return answer
```

If you are facing any issue on this page. Please let us know.