[Click on the logo to find **Problem Statement**]

Intuition

The given code aims to find the minimum eating speed required for a person to consume bananas from a set of piles within a given time limit. It employs binary search to efficiently search for the optimal eating speed. The search space is initialized with the minimum possible speed, low, and the maximum possible speed, high, set to the maximum number of bananas in any pile. The code iteratively narrows down the search space by calculating the total hours needed to consume all bananas at the current mid speed. If the calculated hours are less than or equal to the given time limit, it updates the result with the current mid speed and further narrows the search space by adjusting the high bound. Conversely, if the calculated hours exceed the given time limit, the search space is adjusted by updating the low bound. The process continues until the low and high bounds converge. The final result represents the minimum eating speed satisfying the time constraint.

Approach

1. **Initialization:**
   - Initialize the search space with `low` set to 1 (minimum possible speed) and `high` set to the maximum number of bananas in any pile.
   - Initialize the `result` variable to the maximum speed initially.
2. **Binary Search:**
   - Use a binary search to efficiently explore the possible eating speeds within the initialized search space.
   - In each iteration, calculate the mid point of the current search space.
   - Use a loop to calculate the total hours needed to consume all bananas at the current mid speed.
   - Adjust the search space based on the comparison of calculated hours with the given time limit:
     - If hours are less than or equal to the given time (`hours <= h`), update the result with the current mid speed and adjust the high bound to explore lower speeds.
     - If hours exceed the given time (`hours > h`), adjust the low bound to explore higher speeds.

3. **Convergence and Result:**
   - Continue the binary search until the low and high bounds converge.
   - The final result represents the minimum eating speed satisfying the time constraint.

Complexity

- Time complexity: O(N * log(max(piles)))
- Space complexity: O(1)

Code

```
class Solution:
    def minEatingSpeed(self, piles: List[int], h: int) -> int:
        low = 1
        high = max(piles)
        result = high

        while low <= high:
            mid =  ((high - low) // 2) + low
            hours = 0
            for p in piles:
                hours += math.ceil(p/mid)

            if hours <= h:
                result = min(result, mid)
                high = mid - 1
            else:
                low = mid + 1
        return result
```

If you want to see more solutions to coding problems, you can visit:

GitHub