Problem 875: Koko Eating Bananas

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Koko loves to eat bananas. There are

n

piles of bananas, the

i

th

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

Code Snippets

C++:

```
class Solution {
public:
int minEatingSpeed(vector<int>& piles, int h) {

}
};
```

Java:

```
class Solution {
public int minEatingSpeed(int[] piles, int h) {

}
}
```

Python3:

```
class Solution:
def minEatingSpeed(self, piles: List[int], h: int) -> int:
```

Python:

```
class Solution(object):
def minEatingSpeed(self, piles, h):
"""
:type piles: List[int]
:type h: int
:rtype: int
"""
```

JavaScript:

```
/**
* @param {number[]} piles
* @param {number} h
* @return {number}
*/
var minEatingSpeed = function(piles, h) {

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public int MinEatingSpeed(int[] piles, int h) {

}
}
```

C:

```
int minEatingSpeed(int* piles, int pilesSize, int h) {

}
```

Go:

```
func minEatingSpeed(piles []int, h int) int {

}
```

Kotlin:

```
class Solution {
fun minEatingSpeed(piles: IntArray, h: Int): Int {

}
}
```

Swift:

```
class Solution {
func minEatingSpeed(_ piles: [Int], _ h: Int) -> Int {

}
}
```

Rust:

```
impl Solution {
pub fn min_eating_speed(piles: Vec<i32>, h: i32) -> i32 {

}
}
```

Ruby:

```
# @param {Integer[]} piles
# @param {Integer} h
# @return {Integer}
def min_eating_speed(piles, h)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $piles
* @param Integer $h
* @return Integer
*/
function minEatingSpeed($piles, $h) {

}
}
```

Dart:

```
class Solution {
int minEatingSpeed(List<int> piles, int h) {

}
}
```

Scala:

```
object Solution {
def minEatingSpeed(piles: Array[Int], h: Int): Int = {

}
}
```

Elixir:

```
defmodule Solution do
@spec min_eating_speed(piles :: [integer], h :: integer) :: integer
def min_eating_speed(piles, h) do
```

```
end
end
```

Erlang:

```
-spec min_eating_speed(Piles :: [integer()], H :: integer()) -> integer().
min_eating_speed(Piles, H) ->
.
```

Racket:

```
(define/contract (min-eating-speed piles h)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```

Solutions

C++ Solution:

```
/*
* Problem: Koko Eating Bananas
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

class Solution {
public:
int minEatingSpeed(vector<int>& piles, int h) {

}
};
```

Java Solution:

```
/**
* Problem: Koko Eating Bananas
```

```
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

}
}
```

Python3 Solution:

```
"""
Problem: Koko Eating Bananas
Difficulty: Medium
Tags: array, search

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(1) to O(n) depending on approach
"""

class Solution:
def minEatingSpeed(self, piles: List[int], h: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def minEatingSpeed(self, piles, h):
"""
:type piles: List[int]
:type h: int
:rtype: int
"""
```

JavaScript Solution:

```
/**
* Problem: Koko Eating Bananas
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

/**
* @param {number[]} piles
* @param {number} h
* @return {number}
*/
var minEatingSpeed = function(piles, h) {

};
```

TypeScript Solution:

```
/**
* Problem: Koko Eating Bananas
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

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

};
```

C# Solution:

```
/*
* Problem: Koko Eating Bananas
* Difficulty: Medium
* Tags: array, search
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

public class Solution {
public int MinEatingSpeed(int[] piles, int h) {

}
}
```

C Solution:

```
/*
* Problem: Koko Eating Bananas
* Difficulty: Medium
* Tags: array, search
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(1) to O(n) depending on approach
*/

int minEatingSpeed(int* piles, int pilesSize, int h) {

}
```

Go Solution:

```
// Problem: Koko Eating Bananas
// Difficulty: Medium
// Tags: array, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

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

}
```

Kotlin Solution:

```
class Solution {
fun minEatingSpeed(piles: IntArray, h: Int): Int {

}
}
```

Swift Solution:

```
class Solution {
func minEatingSpeed(_ piles: [Int], _ h: Int) -> Int {

}
}
```

Rust Solution:

```
// Problem: Koko Eating Bananas
// Difficulty: Medium
// Tags: array, search
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(1) to O(n) depending on approach

impl Solution {
pub fn min_eating_speed(piles: Vec<i32>, h: i32) -> i32 {

}
}
```

Ruby Solution:

```
# @param {Integer[]} piles
# @param {Integer} h
# @return {Integer}
def min_eating_speed(piles, h)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $piles
* @param Integer $h
* @return Integer
*/
function minEatingSpeed($piles, $h) {

}
}
```

Dart Solution:

```
class Solution {
int minEatingSpeed(List<int> piles, int h) {

}
}
```

Scala Solution:

```
object Solution {
def minEatingSpeed(piles: Array[Int], h: Int): Int = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec min_eating_speed(piles :: [integer], h :: integer) :: integer
def min_eating_speed(piles, h) do

end
end
```

Erlang Solution:

```
-spec min_eating_speed(Piles :: [integer()], H :: integer()) -> integer().
min_eating_speed(Piles, H) ->
.
```

Racket Solution:

```
(define/contract (min-eating-speed piles h)
(-> (listof exact-integer?) exact-integer? exact-integer?)
)
```