2604. Minimum Time to Eat All Grains Binary Search Sorting Array Hard

### Leetcode Link

# Problem Description

goal is to calculate the minimum time required for all hens to eat all the grains, assuming each hen can move one unit left or right in one second. Hens can move simultaneously and independently, and a hen can eat any grain if they share the same position on the line. Importantly, there's no limit to how many grains a single hen can eat, they just need to be at the same position. The question asks for the optimal strategy that minimizes the time for all the grains to be eaten.

In this problem, we are given two arrays representing the positions of n hens (hens) and m grains (grains) on a line respectively. The

Intuition The intuition behind the solution involves recognizing that since hens can move independently and multiple hens can eat from the same pile of grains, we want to minimize the distance each hen needs to move. To find the minimum time, we can use a binary search strategy, where the search space is the time it takes for the hens to eat all the grains. We are essentially asking "Can all grains be eaten within t seconds?" for various values of t.

The solution approach uses binary search to minimize the time by trying to find the smallest t such that all conditions are satisfied which involves checking whether the hens can eat all grains within that time frame. To facilitate the binary search check, we sort both the hens and grains because only the relative positions matter, not the specific values. After sorting, we iterate through each hen and determine if it can eat the available grains within t seconds. If a hen at

seconds (x - y <= t). If the grain is to the right of the hen (y > x), then we check if the hen can reach it within t seconds (y - x <= t). If during this check for all hens, we find that all grains have been accounted for within t seconds, we consider it successful.

position x encounters a grain at position y, and y is to the left of x (y <= x), then the hen can eat the grain if it can reach it within t

It's critical that this checking function is efficient because it is called many times during binary search. By moving our "grain pointer" only to the right, we ensure we don't do redundant checks, thus optimizing the process.

grains. To start, we need a check(t) function that takes t as an input and returns True if all grains can be eaten by hens within t seconds, else it returns False.

In the implementation, our main goal is to determine the minimum time t, using binary search, during which all hens can eat all

Solution Approach

3. Loop through each hen positioned at x, and for each, attempt to eat the grain located at grains[j], which is y. 4. For each hen, check the position of the grain relative to the hen's position. There are two cases: If the grain is to the left of or at the hen's position (y <= x):</p>

Calculate the distance d = x − y. If d > t, False is returned, because the hen can't reach the grain within t seconds.

Iterate over grains starting from j to the right, moving j forward as long as the distance for the hen to return from the

Here's a step-by-step breakdown of the check(t) function:

2. Use a variable j to keep track of the next grain to be checked.

grain to its original position and then to the next grain is less than or equal to t. If the grain is to the right of the hen (y > x):

1. Sort both hens and grains in ascending order to efficiently pair hens with the nearest grains.

 Increment j to the right as long as the distance from the hen to the next grain is less than or equal to t. 5. If by the end of this process j == m, which means all grains have been visited, return True. Otherwise, return False.

problem down to a manageable and computationally viable process.

Firstly, we sort both arrays for efficient processing:

positioned at hens = [1, 4] and the grains at grains = [2, 3, 5] on the line.

check function within a binary search over the range being considered for t, which is [0, r], where r is an initially large range to cover all possible time scenarios, we can efficiently home in on the smallest t that returns True.

Now, this check over all possible hens and grains determines if it's feasible for all hens to eat all grains in t seconds. By running this

In Python's bisect module, bisect\_left is used, which returns the index of the first item in the sorted array that is greater than or

equal to the specified value. Here, that specified value is True, meaning we're searching for the smallest t where check returns True.

This solution is efficient and leverages the capabilities of sorting and binary search to reduce an otherwise complicated simulation

Let's walk through a small example to illustrate the solution approach. Suppose we have n = 2 hens and m = 3 grains, with the hens

The binary search continues to adjust this range until the lower bound meets the criteria, which is the minimum time t that guarantees all grains are eaten.

 Hens: hens = [1, 4] (already sorted) Grains: grains = [2, 3, 5] (already sorted)

Now we will define a range for our binary search. The maximum possible time t would be if the furthest grain is directly opposite the

furthest hen, which in this case would be the hen at position 4 and the grain at position 5, thus maximum time t\_max = 1. We begin

1. Middle of our range (t = 0) is checked first to see if it is possible for hens to eat all grains in 0 seconds, which is clearly not

The second hen at position 4 can reach and eat the grain at position 5 within 1 second. It can also backtrack to eat the grain

possible because the hens and grains are not at the same positions. Thus, check(0) returns False.

The first hen at position 1 can reach and eat the grain at position 2 within 1 second.

2. Since t = 0 is not enough time, we need to increase t. The next value we check is t = 1, which is the midpoint of the updated

**Python Solution** 

class Solution:

12

13

14

15

16

17

18

19

20

21

22

23

24

33

34

35

36

37

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

71

5

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

61

60 }

};

**Time Complexity** 

Space Complexity

70 };

};

Typescript Solution

from bisect import bisect\_left

from typing import List

with t ranging from 0 to t\_max.

We then start our binary search:

range [1, 1]. We run check(1):

at position 3 after eating the grain at position 5.

def minimumTime(self, hens: List[int], grains: List[int]) -> int:

return True # All grains have been picked

 $range_max = abs(hens[0] - grains[0]) + grains[-1] - grains[0] + 1$ 

minimum\_time = bisect\_left(range(range\_max), True, key=can\_pick\_all)

// Set the initial range of time to search for the minimum time needed.

// 'left' will hold the minimum time needed when the while-loop finishes.

# Find the minimum time within which all grains can be picked using binary search

while grain\_index < m and grains[grain\_index] <= hen:</pre>

if grain\_index == m:

if next\_grain <= hen:

if diff > t:

else:

next\_grain = grains[grain\_index]

diff = hen - next\_grain

grain\_index += 1

grain\_index += 1

Example Walkthrough

To summarize with our example, we used binary search to determine that 1 second is the minimum time required for the hens at positions 1 and 4 to eat all grains at positions 2, 3, and 5. This demonstrates the effectiveness of sorting and binary search in finding the optimal solution for the problem.

Since check(1) returns True, we have found our solution: all hens can eat all grains in a minimum time of 1 second. There's no need

After simulating the process, we find that all the grains can be eaten within 1 second. Thus, check(1) returns True.

to continue the binary search because we can't find a smaller non-negative value of t that satisfies the conditions.

# Helper function to check if all grains can be picked within time 't' 6 # It simulates the picking process for a given time 't' and returns True if possible, False otherwise def can\_pick\_all(t: int) -> bool: 8 grain\_index = 0 # Start with the first grain 9 m = len(grains) 10 11 for hen in hens:

return False # hen is too far from the grain to pick it within time 't'

# The hen will pick the grains until it is closer to them than time 't'

25 while grain\_index < m and grains[grain\_index] - hen <= t:</pre> 26 grain\_index += 1 return grain\_index == m # Check if all grains are picked 27 28 29 # Sort both hens and grains lists to make it easier to process them in order 30 hens.sort() 31 grains.sort() 32

# Compute the initial range for 't', which can be up to the furthest distance a hen might need to travel

// It's between 0 and the maximum possible distance a hen needs to travel minus the first gain position.

int right = Math.abs(hensPositions[0] - grainPositions[0]) + grainPositions[totalGrains - 1] - grainPositions[0];

while grain\_index < m and min(diff, grains[grain\_index] - hen) + grains[grain\_index] - next\_grain <= t:

```
38
39
            return minimum_time
40
```

Java Solution

class Solution {

import java.util.Arrays;

private int[] hensPositions;

private int totalGrains;

int left = 0;

private int[] grainPositions;

hensPositions = hens;

while (left < right) {

} else {

return left;

grainPositions = grains;

totalGrains = grains.length;

Arrays.sort(hensPositions);

Arrays.sort(grainPositions);

right = mid;

left = mid + 1;

int mid = (left + right) / 2;

if (isTimeSufficient(mid)) {

public int minimumTime(int[] hens, int[] grains) {

// Use binary search to find the minimum time needed.

```
33
 34
         private boolean isTimeSufficient(int allowedTime) {
 35
             int grainIndex = 0;
 36
 37
             // Check if the time allowed is sufficient for each hen.
 38
             for (int henPosition : hensPositions) {
                 // If all grains have been checked, return true.
 39
                 if (grainIndex == totalGrains) {
 40
                     return true;
 41
 42
 43
 44
                 int grainPosition = grainPositions[grainIndex];
                 if (grainPosition <= henPosition) {</pre>
 45
                     int distance = henPosition - grainPosition;
 46
 47
                     if (distance > allowedTime) {
 48
                         // If the current grain is too far for the time allowed, return false.
 49
                         return false;
 50
 51
 52
                     // Find the next grain that is farther than the hen but within the allowed time.
 53
                     while (grainIndex < totalGrains && grainPositions[grainIndex] <= henPosition) {</pre>
 54
                         grainIndex++;
 55
 56
 57
                     // Attempt to get as close as possible to the current hen position without exceeding the allowed time.
 58
                     while (grainIndex < totalGrains && Math.min(distance, grainPositions[grainIndex] - henPosition) + grainPositions[gr
 59
                         grainIndex++;
 60
                 } else {
 61
 62
                     // Find the grain that is within the allowed time for the current hen.
 63
                     while (grainIndex < totalGrains && grainPositions[grainIndex] - henPosition <= allowedTime) {</pre>
                         grainIndex++;
 64
 65
 66
 67
 68
 69
             // Return true if all grains have been assigned, false otherwise.
 70
             return grainIndex == totalGrains;
 71
 72 }
 73
C++ Solution
    #include <vector>
     #include <algorithm>
     class Solution {
     public:
         // This method computes the minimum time needed for all hens to eat grains
         // given their positions and the positions of the grains.
         int minimumTime(vector<int>& hens, vector<int>& grains) {
  8
             // Sort the hens and grains in non-decreasing order
  9
 10
             sort(hens.begin(), hens.end());
             sort(grains.begin(), grains.end());
 11
 12
 13
             int numberOfGrains = grains.size();
 14
 15
             // Setup the binary search boundaries
 16
             // The starting left boundary 'l' is set to 0 indicating the lowest possible time
             // The starting right boundary 'r' is based on the furthest possible distance a hen
 17
             // might need to travel, which is from the first hen to the first grain location plus
 18
 19
             // from the first grain location to the last grain location
             int leftBoundary = 0;
 20
             int rightBoundary = abs(hens[0] - grains[0]) + grains[numberOfGrains - 1] - grains[0];
```

// Define the check loop as a lambda function to determine if a given time 't' is sufficient

// Move the grain index to the next grain past the current hen position

// Keep moving the grain index as long as the hen can reach the grain within the 'time'

while (grainIndex < numberOfGrains && min(distance, grains[grainIndex] - henPos) +</pre>

// Move the grain index as long as the hen can reach the grain within the 'time'

while (grainIndex < numberOfGrains && grains[grainIndex] - henPos <= time) {</pre>

while (grainIndex < numberOfGrains && grains[grainIndex] <= henPos) {</pre>

grains[grainIndex] - grainPos <= time) {</pre>

// If we've assigned a grain to each hen within the time, return true

// If midTime is enough for all hens to eat, try to find a smaller time

// The range of possible minimum times starts at 0 and extends up to a logical upper bound

let upperBound = Math.abs(hens[0] - grains[0]) + grains[grainCount - 1] - grains[0] + 1;

// A helper function that checks if all hens can eat grains within the time 'timeLimit'

auto check = [&](int time) -> bool {

return true;

if (grainPos <= henPos) {</pre>

if (grainIndex == numberOfGrains) {

int grainPos = grains[grainIndex];

if (distance > time) {

return false;

++grainIndex;

++grainIndex;

++grainIndex;

return grainIndex == numberOfGrains;

while (leftBoundary < rightBoundary) {</pre>

rightBoundary = midTime;

leftBoundary = midTime + 1;

if (check(midTime)) {

} else {

return leftBoundary;

hens.sort( $(a, b) \Rightarrow a - b);$ 

let grainIndex = 0;

let lowerBound = 0;

 $grains.sort((a, b) \Rightarrow a - b);$ 

const grainCount = grains.length;

for (const henPosition of hens) {

return true;

if (grainIndex === grainCount) {

const grainPosition = grains[grainIndex];

// Compute distance from hen to grain

if (grainPosition <= henPosition) {</pre>

if (distance > timeLimit) {

return false;

grainIndex++;

grainIndex++;

// Check if all grains are distributed

// Perform a binary search to find the minimum time needed

const mid = (lowerBound + upperBound) >> 1;

// Otherwise, search the upper half

// Return the smallest time within which all hens can eat

return grainIndex === grainCount;

if (canFeedWithinTimeLimit(mid)) {

while (lowerBound < upperBound) {</pre>

upperBound = mid;

lowerBound = mid + 1;

} else {

return lowerBound;

Time and Space Complexity

// Perform a binary search to find the minimum time required

// If midTime is not enough, ignore the left half

int midTime = (leftBoundary + rightBoundary) / 2;

// Return the smallest time in which all hens can eat

function minimumTime(hens: number[], grains: number[]): number {

const canFeedWithinTimeLimit = (timeLimit: number): boolean => {

// If all grains have been used, return true

const distance = henPosition - grainPosition;

// Hen can't reach grain within the time limit

// Move up to the grain that the hen can eat within the time limit

// If it's possible to feed within this time frame, search the lower half

while (grainIndex < grainCount && grains[grainIndex] - henPosition <= timeLimit) {</pre>

while (grainIndex < grainCount && Math.min(distance, grains[grainIndex] - henPosition) + grains[grainIndex] - grain

// Sort both hens and grains arrays in ascending order

int distance = henPos - grainPos;

for (int henPos : hens) {

int grainIndex = 0;

} else {

### 26 // Move up to the closest grain that the hen can reach within the time limit 27 28 while (grainIndex < grainCount && grains[grainIndex] <= henPosition) {</pre> 29 grainIndex++; 30 31 // Move up to the grain that can be eaten in the optimal time

} else {

## The time complexity of the given code can be analyzed as follows: 1. Sorting both the hens and grains lists: hens.sort() and grains.sort() both have a time complexity of O(n \log n) and O(m \log

step of the binary search, the time it takes is O(\log U) for the binary search times the complexity of the check function itself. 3. check function: Inside the binary search, the check function is called, which runs in 0(m + n) because it goes through all grains and potentially all hens in the worst case.

The space complexity of the given code can be analyzed as follows:

m) respectively, where n is the number of hens and m is the number of grains.

Combining these factors, we get a total time complexity of  $0(n \log n + m \log m + (m + n) \log U)$ .

2. Binary search: The bisect\_left function performs binary search on a range of size r. The size of this range can be considered

as U because it is determined by the maximum gap between positions of grains. Since it performs the check function at each

2. The binary search uses 0(1) space. 3. The check function uses 0(1) space since it only uses a few variables and all operations are done in place.

1. Sorting requires 0(1) additional space if the sort is in-place (which Python's sort method is, for example).

Adding these up, we get a space complexity of  $0(\log m + \log n)$  for the recursion stack of the sorting algorithms if the sort implementation used is not in-place.