In a universe known as Earth C-137, Rick has invented a new type of basket which exerts a magnetic force between balls placed

Problem Description

has m balls that he needs to place into these baskets. The goal is to distribute the balls in such a way that the minimum magnetic force between any two balls is maximized. Magnetic force here is simply the absolute difference in the positions (|x - y|) of two different balls. Given an integer array position

inside it. He has n such baskets, each located at a specific position delineated by the array position[i]. Morty, on the other hand,

representing the basket positions and an integer m representing the number of balls Morty has, the task is to return the maximum minimum magnetic force that can be achieved between any two balls placed in the baskets.

Intuition

The intuition behind the solution involves utilizing binary search to efficiently find the maximum minimum magnetic force. Since we

want to maximize the minimum distance between any two balls, we can search for the best distance by examining the feasible range

of distances. By sorting the positions of the baskets initially, we structure our problem space in a way that enables binary search.

• We know that the minimum possible force is 1 (when two balls are next to each other) and the maximum possible force is

• We use binary search to probe the middle value (mid) of our current range as a potential candidate for our maximum minimum

are less than mid distance apart. Starting from the first basket, we place a ball and then find the next basket that is at least mid

position[-1] minus position[0] (assuming we place the balls in the first and last basket).

to be distributed in the baskets without violating the minimum distance restriction.

force. • For each mid value tried, we use the check function to see whether we can place all m balls into baskets such that no two balls

The process works as follows:

- distance away to place the next ball. We continue this process until we either place all m balls or run out of baskets. • If we can place all m balls with at least mid distance between them, it means that a larger distance could also be feasible, and we
- shift our search range upwards. • If we cannot place all m balls with at least mid distance, it means we need to look for a smaller minimum force, and we shift our search range downwards.
- This process continues until we pinpoint the largest minimum distance that can accommodate all m balls, which is our desired maximum minimum force. The idea is akin to finding the "sweet spot" in terms of distance, which is the largest minimum distance that still allows for all m balls
- **Solution Approach**

The solution employs a binary search algorithm, a commonly used pattern for problems where one must find an element in a sorted

array or, like in our case, when trying to decide on the most suitable value within a certain range. Here's how it works step by step:

• Sorting: First, we sort the position array. This allows us to apply binary search since we can now reason about the baskets'

positions relative to each other. • Initializing Binary Search: We set variables left to 1 (the minimum possible force if balls are next to each other) and right to position[-1] - position[0] (the maximum possible force with all balls spanning the complete range of baskets). These will be

Termination:

constraints.

Example Walkthrough

baskets, and Morty has m = 3 balls to place.

1. Sort the position array: The array is already sorted: [1, 2, 4, 7, 8, 12].

 \circ Because check(mid) was False, we set right = mid - 1 = 6 - 1 = 5.

7. New Check Function: With mid = 3, we try to place the balls again:

Place the first ball at position 1 (since cnt starts at 1).

our search boundaries.

meaning we still have a range to explore.

up, preventing an infinite loop in certain conditions.

• Binary Search Decision: After each call to the check function:

• Mid Calculation: Inside the loop, mid is calculated with the expression (left + right + 1) >> 1, which is effectively (left + right + 1) / 2, but using bitwise right shift to do integer division by two. Adding 1 before dividing ensures that mid is rounded

• Check Function: The check function is the heart of our binary search. It tries to place balls in the baskets starting with the first

• Binary Search Loop: We enter a loop to perform the binary search. The condition for continuing the loop is left < right,

basket and moving to the right, ensuring that each new ball is at least mid distance apart from the previous. Note that cnt starts at 1 because we place the first ball in the first basket without checking the distance. If it finds a position that is mid or more away from the last filled basket, it places a ball there (cnt is incremented).

• The function returns True if all m balls were placed (cnt >= m), meaning the mid value is a feasible minimum force.

• The loop continues until we either run out of baskets or we have placed all m balls.

mid. If check(mid) returns False, it means mid is too large to fit all balls, so we decrease our search range by setting right = mid - 1.

• The loop terminates when left equals right, meaning we found the maximum left for which check(left) is True.

• Finally, we return left, which now represents the maximum minimum force between any two balls according to the

Using binary search, we've effectively reduced what could be a very time-consuming search across all possible distances to a much

Let's illustrate the solution approach using a small example. Suppose position = [1, 2, 4, 7, 8, 12], representing the positions of

If check(mid) returns True, it implies that mid is a valid minimum force and could possibly be increased, so we set left =

- faster logarithmic time complexity process. By iteratively halving our search space, we home in on the optimal solution without having to explicitly evaluate every possible distribution of balls among baskets.
- 3. Binary Search Loop: We now enter a loop where we will try to find the maximum minimum distance (mid) between balls that still allows us to place all m balls. 4. Mid Calculation: We calculate mid as (left + right + 1) >> 1. Initially, mid = (1 + 11 + 1) >> 1 = 6.

• The third ball can be placed no closer than position 12, which is 5 units away from position 7. However, this distance is less

5. Check Function: We try to place 3 balls in the baskets ensuring that each new ball is at least mid (6) distance apart.

• The next ball can be placed no closer than position 7, which is 6 units away from position 1.

We repeat steps 4 to 6 with the new value of right. The new mid would then be (1 + 5 + 1) >> 1 = 3.

The third ball can be placed at position 7, which is 3 units away from position 4.

All m balls are placed with at least mid distance between them, so check(mid) returns True.

2. Initialize Binary Search: We set left = 1 and right = 12 - 1 = 11 as the furthest possible distance any two balls could have.

Since we cannot place the third ball at the required distance of mid, check(mid) returns False. 6. Binary Search Decision:

than mid (6).

8. Binary Search Decision:

update left = mid = 4.

the optimal solution.

9. Termination:

positions.

Python Solution

class Solution:

10

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

9

11

12

13

14

15

16

17

18

19

20

21

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

50

9

10

11

12

13

14

15

16

17

19

20

21

22

49 }

from typing import List

count = 1

positions.sort()

while left < right:</pre>

else:

class Solution {

 Place the first ball at position 1. The next ball can be placed at position 4, which is 3 units away from position 1.

 As check(mid) was True, we update left = mid = 3. Next, we search between our updated left and right. The new mid would be (3 + 5 + 1) >> 1 = 4.

Repeating the check with mid = 4, we would find that we can place balls at positions 1, 4, 8, which satisfy the condition, so we

If we continue this process, we'd eventually narrow down the value of left and right until left equals right, at which point we find

Therefore, we'd return 4 as our answer. This is the maximum minimum magnetic force between any two balls in the given

• Eventually left equals right, meaning we found the maximum left for which check(left) is True. For this example, we would find that the maximum minimum distance that can be maintained while placing all m balls is 4.

def maxDistance(self, positions: List[int], m: int) -> int:

Iterate through the sorted positions to check

Sort the positions to simplify the distance checks

Binary search to find the largest minimum distance

Calculate middle value to test the placement

Adjust bounds based on the ability to place m balls

// Sort the positions array to establish ordered distances

// Set initial search boundary for the maximum distance

'left' now holds the largest minimum distance we can place 'm' balls

def can_place_balls(min_distance: int) -> bool:

for current_position in positions[1:]:

previous_position = positions[0]

Initialize the binary search bounds

if can_place_balls(mid):

right = mid - 1

public int maxDistance(int[] positions, int m) {

int mid = (left + right + 1) >>> 1;

if (isFeasible(positions, mid, m)) {

// Return the maximum minimum distance found

for (int i = 1; i < positions.length; ++i) {</pre>

prevPosition = currentPosition;

int currentPosition = positions[i];

sort(positions.begin(), positions.end());

int mid = (left + right + 1) / 2;

if (canPlaceBalls(positions, mid, m))

const canPlaceBalls = (distance: number): boolean => {

for (let i = 1; i < positions.length; ++i) {</pre>

const currentPosition = positions[i];

prevPosition = currentPosition;

// If we have placed at least m balls, return true.

++count;

if (canPlaceBalls(mid)) {

right = mid - 1;

return count >= m;

left = mid;

Time and Space Complexity

search that is performed to find the maximum distance.

while (left < right) {</pre>

} else {

let count = 1; // Start by placing the first ball.

// Loop through each position starting from the second one.

if (currentPosition - prevPosition >= distance) {

// If we cannot place all balls, reduce the distance.

check is performed every time the binary search narrows the search space.

Thus, the total time complexity of the binary search is O(n log(max(position) - min(position))).

let prevPosition = positions[0]; // The position of the last placed ball.

// Perform a binary search to find the largest minimum distance that we can achieve.

const mid = Math.floor((left + right + 1) / 2); // Calculate the mid-point distance.

// If we can place all balls with at least 'mid' distance between them, try a larger distance.

int right = positions.back();

while (left < right) {</pre>

// Start from the first position

int prevPosition = positions[0];

private boolean isFeasible(int[] positions, int distance, int m) {

// Iterate through the positions to place the rest of the balls

++count; // Increment the count of placed balls

// If the count of placed balls is at least m, it's feasible

// Initialize the left and right pointers for binary search.

// Use binary search to find the maximum minimum distance.

// Calculate the middle value of the current search range.

right = mid - 1; // If false, decrease the search range.

// left is the smallest possible distance, right is the largest possible distance.

// Check if it's possible to place m balls with at least 'mid' distance apart.

left = mid; // If true, we know that we can try a larger minimum distance.

if (currentPosition - prevPosition >= distance) {

// Place the ball and move to the next

// One ball is already placed at the first position

left = mid

Arrays.sort(positions);

while (left < right) {</pre>

} else {

return left;

int count = 1;

return count >= m;

int left = 1;

else

int left = 1;

left, right = 1, positions[-1] - positions[0]

if current_position - previous_position >= min_distance: 11 12 previous_position = current_position 13 count += 114 # Return True if we can place at least m balls 15 return count >= m

Helper function to check if we can place m balls with at least 'min_distance' distance apart.

mid = (left + right + 1) // 2 # (left + right + 1) to handle the middle for even length

// Minimum possible distance

// Check if it's possible to place m balls with at least 'mid' distance apart

right = mid - 1; // If not possible, continue the search on the left half

// If the current position is at least 'distance' away from the previously placed ball

left = mid; // If possible, continue the search on the right half

// Helper method to check if m balls can be placed with at least 'distance' units apart

int right = positions[positions.length - 1]; // Maximum possible distance

// Use binary search to find the largest minimum distance between m balls

// Calculate the middle value of the current search boundary

34 return left 35 Java Solution

1 class Solution { public: int maxDistance(vector<int>& positions, int m) { // Sort the positions vector to ensure that the elements are in increasing order.

C++ Solution

```
23
           // The final value of left is the maximum minimum distance we can achieve.
24
25
           return left;
26
27
       bool canPlaceBalls(vector<int>& positions, int minDistance, int m) {
28
29
           // Place the first ball at the first position.
           int prevPosition = positions[0];
30
31
           // Initialized at one since we already placed one ball.
32
           int count = 1;
33
34
           // Iterate through the positions starting from the second ball.
35
           for (int i = 1; i < positions.size(); ++i) {</pre>
               int currentPosition = positions[i];
36
37
               // Check if placing a ball here would maintain the minimum distance 'minDistance'.
               if (currentPosition - prevPosition >= minDistance) {
38
                   prevPosition = currentPosition; // Place the ball.
39
                   ++count; // Increment the ball count.
40
41
42
43
           // If we can place at least 'm' balls, return true.
44
45
           return count >= m;
46
47 };
48
Typescript Solution
  1 /**
     * Calculate the maximum distance between m balls placed in sorted positions.
      * @param {number[]} positions - Array of initial positions of balls (not necessarily sorted).
      * @param {number} m - Number of balls to be placed.
      * @returns {number} - The largest minimum distance between any two balls.
     const maxDistance = (positions: number[], m: number): number => {
         // First, sort the positions array in ascending order.
         positions.sort((a, b) => a - b);
  9
 10
         // Define the binary search boundaries.
 11
 12
         let left: number = 1; // The minimum possible distance.
 13
         let right: number = positions[positions.length - 1]; // The maximum possible distance, which is between the first and the last
 14
 15
         /**
          * Helper function to check if we can place m balls with at least 'distance' between them.
 16
          * @param {number} distance - The minimum distance to maintain between any two balls.
 17
 18
          * @returns {boolean} - True if it's possible to place all m balls with the minimum distance, false otherwise.
 19
```

// If the current position is at least 'distance' away from the last placed ball, place a new ball here.

46 47 48 49 // After binary search, 'left' will represent the largest minimum distance we can achieve. 50 return left; 51 };

Time Complexity

Space Complexity

20

21

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

52

};

2. The binary search runs in O(log(max(position) - min(position))) time since it operates on the range of possible answers, which is determined by the difference between the maximum and minimum elements in the sorted position array. Within each iteration of the binary search, there is a loop that operates in O(n) time to check if a given force f allows placing m magnets. This

The time complexity of the maxDistance function is determined by two main operations: sorting the position list and the binary

1. Sorting the position array has a time complexity of $O(n \log n)$, where n is the number of elements in the position list.

Combining these two operations, the overall time complexity of the maxDistance function is 0(n log n) + 0(n log(max(position) min(position))). Since the binary search is dependent on the range of position values which might be larger than n itself, in practice, the dominating term depends on the specific values of position. If max(position) - min(position) is significantly larger than n, the binary search could be the more significant term. Hence, the final time complexity is $O(n \log n + n \log(\max(position) - \min(position)))$.

The space complexity of the maxDistance function arises from the sorting operation and the additional space used by the check function.

Sorting the array in-place has a space complexity of 0(1).

 No additional data structures that grow with the size of the input are used. Therefore, the overall space complexity of the function is 0(1), which denotes constant space complexity.

• The check function uses constant space, only storing a few variables like prev, cnt, and curr.