Sorting

Problem Description

Geometry

Hard

Array

Math

presence of points does not block our view of other points.

given signifies the total span of our field of view in degrees, meaning we can see points that lie within an angular range of angle degrees from our current direction of sight. Every point is defined by coordinates $[x_i, y_i]$, and our location is $[pos_x, pos_y]$. The key task is to find out the maximum number of points we can see from our location when choosing the optimal rotation. Points located at our position are always visible,

In this problem, we are given a list of points on a 2-D plane, an angle, and a location which represents our position. Our initial view

direction faces due east, and we cannot physically move from our position, but we can rotate to change the field of view. The angle

Sliding Window

and the visibility of other points is determined by whether the angle formed by a point, our position, and the immediate east direction from our position is within our field of view. Moreover, multiple points may share the same coordinates, and there can also be points at our current location. Importantly, the

Intuition To solve the problem, we use the concept of angular sweep. We calculate the angle between the east direction and the line

connecting our location to each point. If there's a point at our location, it's always visible, so we count these points directly by

For the remaining points, we convert their coordinates into angles using the atan2 function, which handles the correct quadrant of

incrementing a counter each time we encounter such a point.

the angle for us. Then we sort these angles in ascending order for a sweeping process. Afterward, we replicate the sorted angles while adding 2 * pi to each, effectively simulating a circular sweep beyond the initial 0 to 2*pi range to handle the wrap-around at the 360-degree mark. This way, we can perform a linear sweep using a two-pointer technique or binary search.

We iterate through each angle, considering it as the left boundary of our field of view, and then find using binary search (via

bisect_right) the farthest angle that can fit within the field of view spanned by angle degrees. By doing this for all starting angles, and finding the maximum range of points that can fit within our field of view for each start angle, we can establish the maximum number of points visible after the optimal rotation.

is because points at our location are not subject to angular constraints and are always counted. Solution Approach The solution follows a specific pattern of dealing with geometric problems involving an angular field of view:

We then add the count of points at our location to this maximum number to get the total maximum number of points we can see. This

function atan2(yi - y, xi - x) to calculate the angle of each point relative to our location, location = [x, y]. The atan2 function helps in getting the angle from the x-axis to the point (xi, yi) which takes into account the correct quadrant of the point.

2. Count Overlapping Points: We maintain a counter, same, to count the number of points that exactly overlap with our current

1. Calculate Angles: We start by calculating the angles for all the points relative to our position and the east direction. We use the

sweeps we performed.

Example Walkthrough

Consider the following list of points:

location as these points are always visible.

3. Sort Angles: We sort the list of angles to prepare for the sweeping process. Since we'll be doing an angular sweep, a sorted array of angles is essential.

starting index i, gives us the number of points within this particular field of view window.

interval of the angle list to simulate a circular region. 5. Sliding Window / Binary Search: We apply the concept of a sliding window or a binary search to find the maximal number of points that fit within any angle span:

4. Extend Angles: In order to smoothly handle the wrap-around at 360 degrees (or 2 * pi radians), we extend our list of angles v

by appending each angle incremented by 2 * pi. This is done by v += [deg + 2 * pi for deg in v], effectively doubling the

Using a for loop, iterate over each angle in the original list v (not the extended list), treating it as the start of the field of view.

• For each starting angle, use bisect_right from the bisect module to conduct a binary search to find how far (to the right in

the sorted list) you can go before exceeding the boundary of your field of view. The endpoint is v[i] + t where t is the angle converted to radians (angle * pi / 180). The bisect_right function will return the index of the farthest angle that can be included which, when subtracted by the

6. Find Maximum: The maximum number of points within any such window across all possible starting points is found by mx =

max((bisect_right(v, v[i] + t) - i for i in range(n)), default=0). This takes the highest number from all the window

yielding the final answer: return mx + same. By wrapping up all of these steps inside the visiblePoints function, we get a robust solution that is capable of returning the maximum number of points that can be seen by rotating to the optimal angle.

Let's go through a small example to illustrate the solution approach. Imagine you are located at [0, 0] (the origin), and you have an

angle of 90 degrees for your field of view. This means you can see all points that are within 90 degrees to your east.

7. Add Overlapping Points: Finally, we add the same counter value to our mx to count those points that overlap with our position,

points = [[-1,0], [1,0], [0,0], [0,1], [1,1]]We will use the steps from the solution approach to find the maximum number of points visible:

2. Count Overlapping Points: • The same counter is 1 because one point overlaps with our position [0,0].

3. Sort Angles:

1. Calculate Angles:

4. Extend Angles: \circ The angle list is extended to handle the 360-degree wrap-around. It becomes [0, $\pi/4$, $\pi/2$, π , 2π , $9\pi/4$, $5\pi/2$, 3π].

 \circ For angle 0, we can see up to $\pi/4$ before the field of view ends (covering points [1,0] and [1,1]).

○ We calculate the angles for [-1,0], [1,0], [0,1], and [1,1] because [0,0] is the location itself.

 \circ For point [-1,0], atan2(0, -1) gives us an angle of π (180 degrees).

 \circ For point [0,1], atan2(1, 0) gives us an angle of $\pi/2$ (90 degrees).

 \circ For point [1,1], atan2(1, 1) gives us an angle of $\pi/4$ (45 degrees).

For point [1,0], atan2(0, 1) gives us an angle of 0 degrees (due east).

 \circ We sort the angles, not including the point at the origin: [0, $\pi/4$, $\pi/2$, π].

 \circ We use our field of view angle (90 degrees = $\pi/2$ radians) as the window size.

 \circ The maximum number of points visible in any window is 2 (from angles 0 and $\pi/4$).

 \circ For angle $\pi/4$, we can see up to $9\pi/4$ (also covering [1,0] and [1,1]).

Coordinates of the location from which we are looking

Counter for points at the same location as the observer

if point_x == location_x and point_y == location_y:

Append mirrored polar angles to simulate a circular sweep

polar_angles += [angle + 2 * pi for angle in polar_angles]

Convert the viewing angle to radians for comparison

Compute the polar angle and add it to the list

Calculate polar angles of points relative to the observer's location

angle = atan2(point_y - location_y, point_x - location_x)

Return the sum of the maximum visible points and any overlapping points

If the point is at the observer's location, increment overlap count

location_x, location_y = location

for point_x, point_y in points:

num_points = len(polar_angles)

max_radians = angle * pi / 180

return max_visible + overlap_count

overlap_count += 1

polar_angles.append(angle)

Sort the angles to prepare for angle-range search

Number of unique points (excluding overlaps)

 \circ For angle $\pi/2$, we can still see up to $9\pi/4$ (covering only [0,1]). For angle π, the visible range is beyond our extended list, so it doesn't cover any extra point in the original list.

7. Add Overlapping Points:

6. Find Maximum:

Python Solution

10

11

12

13

14

15

16

17

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

43

44

45

46

47

39

40

41

42

43

44

45

46

47

48

49

50

52

51 }

1 from math import atan2, pi

from typing import List

from bisect import bisect_right

overlap_count = 0

else:

polar_angles.sort()

5. Sliding Window / Binary Search:

the one at our position and the points at [1,0] and [1,1].

 \circ We add the same counter which is 1, so the final maximum number of points visible is 2 + 1 = 3.

class Solution: def visiblePoints(self, points: List[List[int]], angle: int, location: List[int]) -> int: # Stores the polar angles of points with respect to the location polar_angles = []

Hence, for this example, rotating to face any angle between due east (0 degrees) and 45 degrees would allow us to see three points:

38 # Use a sliding window approach to find the maximum number of points in any angle-range 39 # We use bisect_right to find the rightmost position to which max_radians can be added max_visible = max(40 (bisect_right(polar_angles, polar_angles[i] + max_radians) - i for i in range(num_points)), 41 42 default=0,

```
Java Solution
   class Solution {
       public int visiblePoints(List<List<Integer>> points, int angle, List<Integer> location) {
           // List to store the angles from the observer's location to each point
           List<Double> angles = new ArrayList<>();
           int observerX = location.get(0), observerY = location.get(1);
           int overlapCount = 0; // Count of points overlapping the observer's location
           // Iterate over all points to calculate angles or count overlaps
8
9
           for (List<Integer> point : points) {
               int pointX = point.get(0), pointY = point.get(1);
10
11
12
               // Check if the current point overlaps with the observer's location
               if (pointX == observerX && pointY == observerY) {
13
                   overlapCount++; // Increment overlap count
14
15
                    continue; // Skip the rest of the loop
16
17
18
               // Calculate the angle and add it to the list
               angles.add(Math.atan2(pointY - observerY, pointX - observerX));
19
20
21
22
           // Sort the angles in ascending order
23
           Collections.sort(angles);
24
25
           // Duplicate the angles list to handle the circular nature of angles
26
            int anglesCount = angles.size();
27
            for (int i = 0; i < anglesCount; ++i) {</pre>
28
                angles.add(angles.get(i) + 2 * Math.PI);
29
30
           // Convert the viewing angle to radians
31
32
           double threshold = angle * Math.PI / 180;
33
34
           // Initialize the maximum number of visible points
           int maxVisible = 0;
35
36
           // Two-pointer technique to find the maximum number of points visible within the angle range
37
           for (int left = 0, right = 0; right < 2 * anglesCount; ++right) {</pre>
38
```

// Shift the left pointer to the right until the points are outside the viewing angle

int visiblePoints(std::vector<std::vector<int>>& points, int angle, std::vector<int>& location) {

int overlappingPoints = 0; // Count of points that overlap with the observer's location

// Increment count if the point's location is the same as the observer's location

static_cast<double>(point_x - observer_x)));

std::vector<double> angles; // Vector to store the angles of the visible points

int observer_x = location[0], observer_y = location[1]; // Observer's location

// Calculate and store the angle from the observer to the point

angles.push_back(atan2(static_cast<double>(point_y - observer_y),

while (left < right && angles.get(right) - angles.get(left) > threshold) {

left++; // Narrow the range by moving the left pointer to the right

// Update maxVisible with the maximum number of points in the current range

// Return the maximum number of visible points plus any overlapping points

maxVisible = Math.max(maxVisible, right - left + 1);

return maxVisible + overlapCount;

for (const auto& point : points) {

++overlappingPoints;

// Sort the angles in ascending order

std::sort(angles.begin(), angles.end());

int point_x = point[0], point_y = point[1];

if (point_x == observer_x && point_y == observer_y) {

14 15 16 } else { 17 18

C++ Solution

1 #include <vector>

2 #include <algorithm>

#include <cmath>

class Solution {

public:

8

9

10

11

12

13

19

20

21

22

23

24

```
25
 26
             int totalAngles = angles.size(); // Total number of unique angles
 27
             // Duplicate the angles by appending 2*PI to each to handle the circular range
 28
             for (int i = 0; i < totalAngles; ++i) {</pre>
 29
                 angles.push_back(angles[i] + 2 * M_PI);
 30
 31
 32
             int maxVisible = 0; // Maximum number of points visible within the angle
 33
             double radianAngle = angle * M_PI / 180; // Convert angle to radians
 34
 35
             // Two-pointer approach to find the max number of points that fit within the angle
 36
             for (int left = 0, right = 0; right < 2 * totalAngles; ++right) {</pre>
 37
                 // Move the left pointer until the points no longer fit in the viewing angle
 38
                 while (left < right && angles[right] - angles[left] > radianAngle) ++left;
                 maxVisible = std::max(maxVisible, right - left + 1);
 39
 40
 41
 42
             // Return the maximum number of points visible plus any overlapping points
             return maxVisible + overlappingPoints;
 43
 44
 45 };
 46
Typescript Solution
    function visiblePoints(points: number[][], angle: number, location: number[]): number {
         const angles: number[] = []; // Array to store the angles of the visible points
         const observerX = location[0], observerY = location[1]; // Observer's location
         let overlappingPoints = 0; // Count of points that overlap with the observer's location
  5
         for (const point of points) {
  6
             const pointX = point[0], pointY = point[1];
             if (pointX === observerX && pointY === observerY) {
  8
                 // Increment count if the point's location overlaps with the observer's location
  9
 10
                 overlappingPoints++;
             } else {
 11
 12
                 // Calculate the angle from the observer to the point and store it in the array
                 const angleRadians = Math.atan2(pointY - observerY, pointX - observerX);
 13
 14
                 angles.push(angleRadians);
 15
 16
 17
 18
         // Sort the angles in ascending order
 19
         angles.sort((a, b) => a - b);
 20
 21
         const totalAngles = angles.length; // Total number of unique angles
 22
 23
         // Duplicate the angles by adding 2*PI to each to handle the wrap—around effect
 24
         for (let i = 0; i < totalAngles; i++) {</pre>
             angles.push(angles[i] + 2 * Math.PI);
 25
 26
 27
 28
         let maxVisible = 0; // Maximum number of points visible within the angle
 29
         const radianAngle = angle * Math.PI / 180; // Convert angle to radians for comparison
 30
         // Two-pointer technique to find the max number of points that fit within the angle
 31
 32
         for (let left = 0, right = 0; right < 2 * totalAngles; right++) </pre>
 33
             // Move the left pointer to ensure all points are within the viewing angle
```

40 // Return the sum of the maximum number of visible points and any overlapping points 41 return maxVisible + overlappingPoints; 42 } 43

Time Complexity

left++;

Time and Space Complexity

34

35

36

37

38

39

respect to location. The complexity for this loop is O(n) where n is the number of points since each operation inside the loop takes 0(1) time.

2. The sort operation on the angles list is $O(n \log n)$ since all non-same points are sorted.

The given code calculates how many points are visible within a given angle from a particular location.

while (angles[right] - angles[left] > radianAngle) {

maxVisible = Math.max(maxVisible, right - left + 1);

added, so it's still O(n).

notation.

4. The max function involves a generator expression with bisect_right calls for each of the n elements in the list. bisect_right has a complexity of $O(\log n)$, so this step has a complexity of $O(n \log n)$.

1. The first loop iterates over all points counting points that are the same as the location and for others, calculates the angle with

3. The list is doubled to handle the circular nature of the problem, which means you're doing another loop of O(n) for that addition.

However, this does not change the order of complexity given that list concatenation is O(k) where k is the size of the list being

5. Adding these together, the overall time complexity is dominated by the sorting and binary search steps, both of which are 0(n log n). Hence, the final time complexity is 0(n log n).

2. After this list v is doubled, making the space complexity 0(2n) or simply 0(n) since constant factors are dropped in Big O

- **Space Complexity** 1. v is a list that stores up to n angles when none of the points coincide with the location. This makes the space complexity O(n).
 - 3. There are no other data structures that grow with input size.

Hence, the final space complexity is O(n).