335. Self Crossing

Hard Geometry Array Math

Problem Description

This problem presents a scenario where you are simulating movement on an X-Y plane starting at the origin (0, 0). You're given an array of integers called `distance` that dictates how far you move in each direction. The movement is in a cycle: north, west, south, east, and then repeats - north, west, and so on. The question asks whether, at any point during this movement, you cross your own path.

Think of it as drawing a zigzag line on a piece of paper without lifting your pen. The task is to figure out if your line touches or crosses itself at any point. A self-crossing path would imply that you end up at a point on the plane that you have previously visited.

The sequence of the movement is important here – after each move, you change your direction in a counter-clockwise manner. This means if you start by going north, the next move will be west, followed by south, then east, and this pattern will continue in this order for each subsequent move.

Intuition

To solve the problem, we must understand the conditions that can cause the path to cross itself. There are generally three cases where crossing can happen:

1. **Case 1: Fourth Line Crosses the First** - This case happens when the current step overshoots the first step. More specifically, the path crosses if the current distance is greater than or equal to the distance two steps back, and the distance a step before is less than or equal to the distance three steps back.
2. **Case 2: Fifth Line Meets the First** - Here, the path's fifth segment overlaps with the first segment if the current step is equal to the step three steps back, and the sum of the current step and the step four steps back is greater than or equal to the step two steps back.
3. **Case 3: Sixth Line Crosses the First** - This is a more complex scenario where the sixth line crosses over the first. For this to happen, several conditions need to match: the fourth step is greater than or equal to the second step, the fifth step is less than or equal to the third step, the sum of the third and the sixth steps is greater than or equal to the first step, and the sixth step is greater than or equal to the difference between the second and fourth steps.

The provided solution checks for these three cases while iterating through the distance array starting from the third index, as cases for self-crossing only become possible from the fourth step onwards. If any of these conditions are satisfied at any point during the iteration, it confirms that the path has crossed itself and the function immediately returns `True`. If none of these cases are satisfied by the end of the iteration, we can confidently say the path does not cross itself, and the function returns `False`.

Overall, the code solution relies on understanding the geometry of movement and identifying cases where a line segment could intersect with non-adjacent segments of the path based on the distances traveled.

Solution Approach

To implement the solution, we follow a straightforward iteration through the `distance` array and apply condition checks based on the three cases that could result in crossing. The code does not use any complex data structures or algorithms, but instead relies on logical conditions and comparisons to determine if a crossing has occurred.

The solution iterates through the `distance` array starting from index 3, because the first three steps (to the north, west, and south) cannot result in a crossing. It is only from the fourth step onwards that we might have a crossing situation.

The implementation is based on the following checks corresponding to the cases illustrated in the Reference Solution Approach:

- **Case 1 Check:** This is when the current step (`d[i]`) crosses the path of the first step (`d[i - 2]`). In code terms, we are checking if `d[i] >= d[i - 2]` and simultaneously, if `d[i - 1] <= d[i - 3]`, if both conditions are true, the path crosses itself.
- **Case 2 Check:** Occurs when the fifth step exactly meets with the first step. The condition includes a check for equality, `d[i - 1] == d[i - 3]`, and also checks if the sum of the current step (`d[i]`) and four steps back (`d[i - 4]`) is greater than or equal to the distance two steps back (`d[i - 2]`).
- **Case 3 Check:** This deals with the potential crossing caused by the sixth line, which is the most complex case and has the most conditions. Here we must ensure the following:
  - The fourth step is greater than or equal to the second, `d[i - 2] >= d[i - 4]`.
  - The fifth step is less than or equal to the third, `d[i - 1] <= d[i - 3]`.
  - The sixth step (`d[i]`) is greater than or equal to the second step (`d[i - 2]`) minus the fourth (`d[i - 4]`), `d[i] >= d[i - 2] - d[i - 4]`.
  - Plus, the sum of the fifth (`d[i - 1]`) and first steps (`d[i - 5]`) is greater than or equal to the third step (`d[i - 3]`).

These checks are evaluated using a for-loop that proceeds to the end of the array, or until a crossing is detected. If none of the conditions is met, the loop finishes and the function returns `False`, indicating that *no* crossing occurred.

The algorithm's complexity is O(n), where n is the length of the `distance` array. This is because it involves iterating through the array once and performing constant-time checks in each iteration.

In summary, the solution applies a systematic check for each case of self-crossing at every step after the third step, immediately ceases the iteration, and returns the result once a self-crossing is detected. If the iteration completes without finding a crossing, the path does not cross itself, and `False` is returned as the final result.

Example Walkthrough

Let's consider a small example with the `distance` array: `[2, 1, 4, 3, 2]`. Following the solution approach, we will walk through the iteration process and apply our conditions to see if the path crosses itself.

1. We start with our initial position at the origin (0, 0) and move as follows:
   - Move 2 units to the north.
   - Turn left and move 1 unit to the west.
   - Turn left and move 4 units to the south.
   - At this point, the path looks like a "7" and has not crossed yet.
2. Now we must check for self-crossing from the fourth step onward:
   - Next, we move 3 units to the east.
3. Now we check our conditions:
   - **Case 1 Check**: We compare the most recent move (3 units east) with two steps back (1 unit west). Since 3 is not greater than or equal to 1, Case 1 does not indicate a crossing.
   - **Case 2 Check**: This is not applicable as we need a minimum of five moves to check this condition.
   - **Case 3 Check**: Also not applicable as we need a minimum of six moves for this case.
4. We continue with the next move:
   - Move 2 units to the north.
5. Now, we repeat the checks for our new step:
   - **Case 1 Check**: The current distance (2 units north) is compared to two steps back (4 units south). We find that `2 < 4`, so there is no crossing here.
   - **Case 2 Check**: Now, we have enough moves to check this condition. We find that the distance one step back (3 units east) is equal to the distance three steps back (1 unit west) + distance five steps back (2 units north). This is a match of Case 2, as `1 + 2 = 3`, and we also have the distance two steps back (4 units south) being greater than or equal to the sum of the distance four steps back (2 units north) and the current step (2 units north). So this case confirms that we have a crossing.

At this point, we detect a path crossing and stop our iteration. We can conclude that the path crosses itself based on the distance array `[2, 1, 4, 3, 2]` as we hit the true condition for Case 2. Thus, our functional implementation of the solution would return `True`.

In this example, the self-crossing case was detected after the fifth move, matching the conditions set out in the second case described in the solution approach.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def isSelfCrossing(self, distances: List[int]) -> bool:
        """
        Check if the given path crosses itself.

        Args:
        distances: A list of integers representing the lengths of consecutive moves.

        Returns:
        True if the path crosses itself, otherwise False.
        """

        # Iterate over the distances, starting from the fourth move
        for i in range(3, len(distances)):
            # Case 1: Current line crosses the line 3 steps behind
            if distances[i] >= distances[i - 2] and distances[i - 1] <= distances[i - 3]:
                return True

            # Case 2: Current line overlaps with the line 4 steps behind
            if i >= 4 and distances[i - 1] == distances[i - 3] and distances[i] + distances[i - 4] >= distances[i - 2]:
                return True

            # Case 3: Current line crosses the line 5 steps behind
            if i >= 5
                and distances[i - 2] >= distances[i - 4]
                and distances[i - 1] <= distances[i - 3]
                and distances[i] >= distances[i - 2] - distances[i - 4]
                and distances[i - 1] + distances[i - 5] >= distances[i - 3]):
                return True

        # If none of the cases cause a cross, the path does not cross itself
        return False
```

Java

```java
class Solution {
    public boolean isSelfCrossing(int[] distance) {
        // Enhanced for loop is unnecessary as we are directly accessing elements.
        // Naming of the variable 'd' is changed to 'distances' for better readability.

        // Start from the fourth element (index starts from 0) and check for self-crossing
        for (int i = 3; i < distance.length; ++i) {
            // Scenario 1: Current line crosses the line 3 steps behind it
            if (distance[i] >= distance[i - 2] && distance[i - 1] <= distance[i - 3]) {
                return true;
            }

            // Scenario 2: Current line touches or crosses the line 4 steps behind it
            if (i >= 4 && distance[i - 1] == distance[i - 3] && distance[i] + distance[i - 4] >= distance[i - 2]) {
                return true;
            }

            // Scenario 3: Current line crosses the line 5 steps behind it
            if (i >= 5 && distance[i - 2] >= distance[i - 4] &&
                distance[i - 1] <= distance[i - 3] &&
                distance[i] >= distance[i - 2] - distance[i - 4] &&
                distance[i - 1] + distance[i - 5] >= distance[i - 3]) {
                return true;
            }
        }

        // If none of the above scenarios occur, there is no self crossing
        return false;
    }
}
```

C++

```cpp
#include <vector>
using namespace std;

class Solution {
public:
    // Function to determine whether the path crosses itself
    bool isSelfCrossing(vector<int>& distance) {
        // Iterate through the distance vector starting from the fourth element
        for (int i = 3; i < distance.size(); ++i) {
            // Case 1: Current distance is greater than or equal to the distance two steps back
            // and the previous distance is less than or equal to the distance three steps back
            if (distance[i] >= distance[i - 2] && distance[i - 1] <= distance[i - 3]) return true;

            // Case 2: Crossover case for a square-like pattern
            // When the current index is at least 4, the distance four steps behind the current
            // distance is greater than or equal to two steps behind, and the previous step
            // distance is equal to the distance three steps back.
            if (i >= 4 && distance[i - 1] == distance[i - 3] && distance[i] + distance[i - 4] >= distance[i - 2]) return true;

            // Case 3: Complex crossover case
            // Occurs when there are at least 6 distances, the current distance is greater than or equal to
            // the difference of the distance two steps and four steps back, the distance four steps
            // back is greater than or equal to the distance three steps back minus the distance one step back
            // plus the distance five steps back is greater than or equal to the distance three steps back.
            if (i >= 5 && distance[i - 2] >= distance[i - 4] && distance[i - 1] <= distance[i - 3] &&
                distance[i] >= distance[i - 2] - distance[i - 4] && distance[i - 1] + distance[i - 5] >= distance[i - 3]) return true;
        }

        // If none of the conditions are met, then the path does not cross itself
        return false;
    }
};
```

TypeScript

```typescript
function isSelfCrossing(distance: number[]): boolean {
    // Iterate through the distance array starting from the fourth element
    for (let i = 3; i < distance.length; ++i) {
        // Case 1: Current distance is greater than or equal to the distance two steps back
        // and the previous distance is less than or equal to the distance three steps back
        if (distance[i] >= distance[i - 2] && distance[i - 1] <= distance[i - 3]) return true;

        // Case 2: Crossover case for a square-like pattern
        // When the current index is at least 4, the distance four steps before the current
        // distance is greater than or equal to two steps before, and the previous step
        // distance is equal to the distance three steps back.
        if (i >= 4 && distance[i - 1] === distance[i - 3] && distance[i] + distance[i - 4] >= distance[i - 2]) return true;

        // Case 3: Complex crossover case
        // This case occurs when there are at least 6 distances, the current distance is greater than or equal to
        // the difference between the distance two and four steps back, the distance four steps
        // back is greater than or equal to the distance six steps back, and the previous distance
        // plus the distance five steps back is greater than or equal to the distance three steps back.
        if (
            i >= 5 &&
            distance[i - 2] >= distance[i - 4] &&
            distance[i - 1] <= distance[i - 3] &&
            distance[i] >= distance[i - 2] - distance[i - 4] &&
            distance[i - 1] + distance[i - 5] >= distance[i - 3]
        ) {
            return true;
        }
    }

    // If none of the conditions are met, then the path does not cross itself
    return false;
}
```

```python
from typing import List

class Solution:
    def isSelfCrossing(self, distances: List[int]) -> bool:
        """
        Check if the given path crosses itself.

        Args:
        distances: A list of integers representing the lengths of consecutive moves.

        Returns:
        True if the path crosses itself, otherwise False.
        """

        # Iterate over the distances, starting from the fourth move
        for i in range(3, len(distances)):
            # Case 1: Current line crosses the line 3 steps behind
            if distances[i] >= distances[i - 2] and distances[i - 1] <= distances[i - 3]:
                return True

            # Case 2: Current line overlaps with the line 4 steps behind
            if i >= 4 and distances[i - 1] == distances[i - 3] and distances[i] + distances[i - 4] >= distances[i - 2]:
                return True

            # Case 3: Current line crosses the line 5 steps behind
            if (i >= 5
                and distances[i - 2] >= distances[i - 4]
                and distances[i - 1] <= distances[i - 3]
                and distances[i] >= distances[i - 2] - distances[i - 4]
                and distances[i - 1] + distances[i - 5] >= distances[i - 3]):
                return True

        # If none of the cases cause a cross, the path does not cross itself
        return False
```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the `distance` array. This is because the code iterates through the array once with a single loop that starts from index 3 and checks up to the end of the array. Within each iteration, the checks performed are constant time operations, so the total time complexity is linear with respect to the size of the input.

The space complexity of the given code is `O(1)`. The solution uses a fixed amount of additional space outside of the input array to store variables for the iteration and conditional checks. The space used does not scale with the input size, thereby making the space complexity constant.