1274. Number of Ships in a Rectangle

Hard Array Divide and Conquer Interactive Leetcode Link

Problem Description

In this interactive problem, we are tasked with finding the number of ships present within a certain rectangle in a Cartesian plane. Each ship occupies an integer point on the plane, and each point can have at most one ship.

We have access to a function `Sea.hasShips(topRight, bottomLeft)` which can inform us whether there's at least one ship in the rectangle formed by these two points, including the boundary points.

Our goal is to determine the number of ships in the rectangle defined by the coordinates of its top-right and bottom-left points. The number of ships in the rectangle is limited to at most 10, and we must make sure that our solution does not call the `hasShips` function more than 400 times, as that would lead to a wrong answer.

Moreover, we have to solve the problem without any prior knowledge of where the ships are located; we can only rely on the information returned by the `hasShips` function.

Intuition

The intuition behind the given solution is to apply a divide-and-conquer strategy, which in this case resembles the 'Divide and Conquer' algorithm design paradigm. This resembles a type of binary search in two dimensions, where we split the area where ships could be into smaller rectangles until we're able to conclude whether a ship is in a particular section or not.

Here is how we might arrive at this solution step-by-step:

1. If the coordinates of the bottom-left corner are greater than those of the top-right corner, the area is invalid, and we return 0 since there can't be any ships in an invalid area.

2. Next, we check if there are any ships in the current rectangle area by calling the `hasShips` function. If it returns false, there are no ships in that rectangle, and we return 0.

3. If the rectangle has been narrowed down to a single point (the coordinates of the bottom-left and top-right are the same), then we have found a ship, and we can return 1.

4. If the area still contains multiple points, we divide the rectangle into four smaller rectangles. This is done by finding the midpoint of the top-right and bottom-left coordinates, effectively splitting the rectangle both horizontally and vertically.

5. We then recursively call the search function on each of the four sub-rectangles. The sum of ships found in all four sub-rectangles gives us the total number of ships in the current rectangle.

6. This process continues, with each recursive call further splitting rectangles and counting ships, until no more subdivisions can be made or the hasShips function determines there are no ships in a particular sub-rectangle.

The divide-and-conquer approach is highly effective because it systematically reduces the search space while ensuring that no call to `hasShips` is wasted.

Solution Approach

The provided solution utilizes depth-first search (DFS) as part of a recursive divide-and-conquer strategy. The divide-and-conquer approach is a significant portion of the solution, as it systematically reduces the overall search space for locating the ships. Let's break down the implementation details:

1. The solution defines a nested function, `dfs(topRight, bottomLeft)`, which is the recursive function responsible for implementing the depth-first search within the given coordinates `topRight` and `bottomLeft`.

2. Initially, `dfs` checks if the current search area defined by the coordinates is valid. The coordinates are invalid if any of the x-coordinates of `bottomLeft` are greater than those of `topRight`, or any of the y-coordinates of `bottomLeft` are above those of `topRight`.

3. If the area is valid, the `dfs` function uses the given `sea.hasShips(topRight, bottomLeft)` function to check whether there are ships in the current rectangle. If there are no ships (`hasShips` returns false), the function returns 0, indicating this area does not need to be explored further.

4. If the rectangle area is down to a single point, meaning the coordinates for `topRight` and `bottomLeft` are equal, then there must be a ship at this point, and the function returns 1.

5. When the rectangle still consists of a range of points, the area is divided into four quarters. The division is done by calculating the midpoint for both x and y coordinates, using bitwise right shift `>>` to find the average of the coordinates, allowing efficient computation.

6. After finding the midpoints `midx` and `midy`, the rectangle is split into four smaller rectangles, and the `dfs` function is called on each. These sub-rectangles are determined by the new combinations of the midpoints and the original coordinates:

   - The first quadrant (`a`) is the upper right and is recursively searched from the original top right corner to the midpoint plus one.
   - The second quadrant (`b`) is the upper left, searched from the midpoint in the x-direction and the original top right in the y-direction to the bottom left x-coordinate and midpoint plus one in the y-coordinate.
   - The third quadrant (`c`) is the lower left, which is the original bottom left to the midpoint.
   - The fourth quadrant (`d`) is the lower right, from the midpoint plus one in the x-direction to the original top right x-coordinate and from the midpoint y-coordinates to the original bottom left y-coordinate.
7. Each recursive call returns the count of ships found within its respective quadrant. These counts are summed to get the total number of ships within the current rectangle.

8. Finally, the `countShips` function calls `dfs` with the original rectangle's coordinates, starting the recursive search and returning the total number of ships found within the entire rectangle.

This solution effectively balances the need to minimize the number of API calls to `hasShips` while ensuring that all potential ship locations are examined.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose we are given a grid where the top-right coordinate is (4,4) and the bottom-left is (0,0), and we want to know how many ships are in this rectangle.

We start by calling `dfs(4,4,0,0)`.

1. We check if the area is valid. In this case, it is valid since (0,0) is not greater than (4,4).

2. We call `Sea.hasShips((4,4),(0,0))`. If this call returns false, we return 0; there are no ships in this area. Let's assume it returns true, meaning there is at least one ship in the rectangle.

3. We check if the area is reduced to a single point, but since (4,4) is not equal to (0,0), it's not.

4. This is not a single-point rectangle, so we find the midpoint for x and y. `midx` is (4+0)/2 = 2 and `midy` is (4+0)/2 = 2, which splits the rectangle into four smaller rectangles.

5. We make recursive calls to `dfs` for the four quadrants:

   - `dfs((4,4),(3,3))` for the top-right quadrant
   - `dfs((2,4),(0,3))` for the top-left quadrant
   - `dfs((2,2),(0,0))` for the bottom-left quadrant
   - `dfs((4,2),(3,0))` for the bottom-right quadrant

Let's assume `Sea.hasShips` returned true for the top-right and bottom-right quadrants and false for the others. This means there are no ships to be found in the top-left and bottom-left quadrants, and we don't need to divide these further.

6. Since top-right returned true, we repeat the process and divide it into four quadrants again. Suppose it gets to the point where `dfs((3,4),(3,4))` is called. Since the area is reduced to a single point and `Sea.hasShips` returns true, we found a ship and return 1.

7. We continue the divide-and-conquer process for the bottom-right quadrant and suppose that through subsequent divisions down to single-point areas, we discover one more ship.

8. We add up the ships found in all quadrants: 1 (top-right) + 0 (top-left) + 0 (bottom-left) + 1 (bottom-right) for a total of 2 ships found.

The recursive process of dividing and conquering helps minimize the number of `Sea.hasShips` calls while making sure that all ships within the specified rectangle are found, adhering to the constraints of calling `hasShips` no more than 400 times.

Python Solution

```python
class Solution:
    def countShips(self, sea: "Sea", top_right: "Point", bottom_left: "Point") -> int:
        # Helper function to perform depth-first search
        def dfs(top_right, bottom_left):
            # Extract coordinates of the two points
            x1, y1 = bottom_left.x, bottom_left.y
            x2, y2 = top_right.x, top_right.y

            # Check for invalid rectangle or if there are no ships in this area
            if x1 > x2 or y1 > y2 or not sea.hasShips(top_right, bottom_left):
                return 0

            # If the rectangle has been reduced to a single point, return 1 (indicating a ship)
            if x1 == x2 and y1 == y2:
                return 1

            # Calculate middle points for dividing the search area
            mid_x = (x1 + x2) // 2
            mid_y = (y1 + y2) // 2

            # Perform the search on the four subdivided rectangles
            # a: top right sub-rectangle
            a = dfs(top_right, Point(mid_x + 1, mid_y + 1))
            # b: top left sub-rectangle
            b = dfs(Point(mid_x, y2), Point(x1, mid_y + 1))
            # c: bottom left sub-rectangle
            c = dfs(Point(mid_x, mid_y), bottom_left)
            # d: bottom right sub-rectangle
            d = dfs(Point(x2, mid_y), Point(mid_x + 1, y1))

            # Combine counts from all four rectangles
            return a + b + c + d

        # Start recursive depth-first search from the given points
        return dfs(top_right, bottom_left)

```

Java Solution

```java
class Solution {
    /**
     * Counts the number of ships present within a rectangular area.
     * This is achieved by recursively dividing the sea area into quadrants and counting the ships.
     *
     * @param sea The Sea interface which has a method to check if there are ships in a given rectangle.
     * @param topRight An array representing the top right coordinates of the rectangle.
     * @param bottomLeft An array representing the bottom left coordinates of the rectangle.
     * @return The number of ships within the provided rectangle.
     */
    public int countShips(Sea sea, int[] topRight, int[] bottomLeft) {
        // Coordinates for bottom left and top right points of the rectangle
        int bottomLeftX = bottomLeft[0], bottomLeftY = bottomLeft[1];
        int topRightX = topRight[0], topRightY = topRight[1];

        // If coordinates are out of bounds, return 0 as there are no ships
        if (bottomLeftX > topRightX || bottomLeftY > topRightY) {
            return 0;
        }

        // If there are no ships in the current rectangle, return 0
        if (!sea.hasShips(topRight, bottomLeft)) {
            return 0;
        }

        // If it is a single point, it must be a ship as per the previous check
        if (bottomLeftX == topRightX && bottomLeftY == topRightY) {
            return 1;
        }

        // Calculate midpoints for the x and y coordinates
        int midX = (bottomLeftX + topRightX) >> 1;
        int midY = (bottomLeftY + topRightY) >> 1;

        // Recursive calls to count ships in each of the four quadrants
        // Top-right quadrant
        int countTopRight = countShips(sea, topRight, new int[]{midX + 1, midY + 1});
        // Top-left quadrant
        int countTopLeft = countShips(sea, new int[]{midX, topRightY}, new int[]{bottomLeftX, midY + 1});
        // Bottom-left quadrant
        int countBottomLeft = countShips(sea, new int[]{midX, midY}, bottomLeft);
        // Bottom-right quadrant
        int countBottomRight = countShips(sea, new int[]{topRightX, midY}, new int[]{midX + 1, bottomLeftY});

        // Sum the counts from all 4 quadrants and return the result
        return countTopRight + countTopLeft + countBottomLeft + countBottomRight;
    }
}

```

C++ Solution

```cpp
// This code solves the problem of counting ships in a sea using a divide and conquer strategy
// The Sea API has a method hasShips that returns true if there are any ships in the rectangular area defined by its arguments

class Solution {
public:
    // countShips recursively counts the number of ships in the given rectangle area of the sea
    int countShips(Sea sea, vector<int> topRight, vector<int> bottomLeft) {
        // Decompose the problem by dividing the search area into smaller rectangles
        // and count the number of ships in each smaller rectangle

        // Bottom-left and top-right coordinates of the rectangle
        int bottomLeftX = bottomLeft[0], bottomLeftY = bottomLeft[1];
        int topRightX = topRight[0], topRightY = topRight[1];

        // Base case 1: If the rectangle is not valid (inversions in coordinates), return 0
        if (bottomLeftX > topRightX || bottomLeftY > topRightY) {
            return 0;
        }

        // Base case 2: If no ships are detected in the current rectangle, return 0
        if (!sea.hasShips(topRight, bottomLeft)) {
            return 0;
        }

        // Base case 3: If the rectangle is a single point and there is a ship,
        // there is exactly one ship in the current rectangle
        if (bottomLeftX == topRightX && bottomLeftY == topRightY) {
            return 1;
        }

        // Calculate midpoints for the x and y coordinates
        int midX = (bottomLeftX + topRightX) / 2;
        int midY = (bottomLeftY + topRightY) / 2;

        // Recursively count ships in the four subdivided rectangles:

        // Top-right rectangle
        int countTopRight = countShips(sea, topRight, {midX + 1, midY + 1});

        // Top-left rectangle
        int countTopLeft = countShips(sea, {midX, topRightY}, {bottomLeftX, midY + 1});

        // Bottom-left rectangle
        int countBottomLeft = countShips(sea, {midX, midY}, bottomLeft);

        // Bottom-right rectangle
        int countBottomRight = countShips(sea, {topRightX, midY}, {midX + 1, bottomLeftY});

        // The total count is the sum of ships in all four rectangles
        return countTopRight + countTopLeft + countBottomLeft + countBottomRight;
    }
};

```

Typescript Solution

```typescript
/**
 * This method counts the number of ships present in a rectangular area of the sea.
 * The Sea class has a method `hasShips` which returns a boolean indicating whether
 * there are any ships in a given rectangle defined by its top right and bottom left coordinates.
 * The method recursively divides the search area into quadrants to find ships.
 *
 * @param {Sea} sea - The sea instance on which we invoke the hasShips API.
 * @param {number[]} topRight - The top right coordinates [x, y] of the search area.
 * @param {number[]} bottomLeft - The bottom left coordinates [x, y] of the search area.
 * @returns {number} The total number of ships found in the given area.
 */
function countShips(sea: Sea, topRight: number[], bottomLeft: number[]): number {
    const [bottomLeftX, bottomLeftY] = bottomLeft;
    const [topRightX, topRightY] = topRight;

    // If the coordinates are out of order or if there are no ships in this area, return 0
    if (bottomLeftX > topRightX || bottomLeftY > topRightY || !sea.hasShips(topRight, bottomLeft)) {
        return 0;
    }

    // If the search area is a single point, return 1, as there is a ship
    if (bottomLeftX === topRightX && bottomLeftY === topRightY) {
        return 1;
    }

    // Calculate midpoints for the search area
    const midX = (bottomLeftX + topRightX) >> 1;
    const midY = (bottomLeftY + topRightY) >> 1;

    // Recursively search the four subdivisions of the current area
    const topLeftCount = countShips(sea, [midX, topRightY], [bottomLeftX, midY + 1]);
    const topRightCount = countShips(sea, topRight, [midX + 1, midY + 1]);
    const bottomLeftCount = countShips(sea, [midX, midY], bottomLeft);
    const bottomRightCount = countShips(sea, [topRightX, midY], [midX + 1, bottomLeftY]);

    // Sum the counts of ships in the four subdivisions
    return topLeftCount + topRightCount + bottomLeftCount + bottomRightCount;
}

```

Time and Space Complexity

The given Python code is implementing a divide-and-conquer algorithm to count the number of ships present in a rectangular section of the sea. It divides the search space into four smaller rectangles at every step until it reaches an individual point or the `hasShips` method returns false.

Time Complexity

The time complexity of this algorithm heavily depends on the implementation of the `hasShips` function, which is a black box to us. However, assuming `hasShips` has O(1) time complexity, we can analyze the recursion.

Every time the `dfs` function is called, it potentially makes up to four further recursive calls until it narrows down to a single point where the top-right and bottom-left corners coincide. This recursive division is similar to a quad-tree structure.

Let's consider `N` as the area of the sea (i.e., `N = (topRight.x - bottomLeft.x + 1) * (topRight.y - bottomLeft.y + 1)`). In the worst case, the algorithm will have to check each point in the input space, resulting in O(N) complexity.

However, due to the nature of the problem where it terminates early if no ships are present in the current quadrant, the average-case time complexity can be better than O(N). If ships are evenly distributed, the division will lead to `T(N) = 4 * T(N/4) + O(1)`, which simplifies to O(N) due to the Master Theorem. But again, in practice, early termination can make the time complexity significantly lower than O(N). A balanced case often cited is O(k * log N), where k is the number of ships.

Space Complexity

The space complexity of the algorithm consists of the recursive call stack depth, which, in the worst case, could be as deep as the number of points in the sea. This implies the worst-case space complexity is O(N).

However, considering the divide-and-conquer nature, a better approximation of space complexity would depend on the depth of the recursion tree. In a balanced scenario where the problem is divided into four equal parts at each level, the maximum depth would be log_4(N), leading to a space complexity of O(log N).

Assuming a more realistic scenario where there are fewer ships, and not every recursive call will result in four further calls due to early termination when no ships are within a quadrant, the average space complexity is less than O(log N), because not all quadrants will be recursed into. The space complexity in this case would be O(k * log N) where k is the number of ships.