874. Walking Robot Simulation

Medium Array Simulation

Leetcode Link

Problem Description

The problem describes a simulation where a robot navigates on an infinite two-dimensional grid starting from the origin point (0, 0) and facing north. The robot can process a series of commands:

- `-2`: this command instructs the robot to turn 90 degrees to its left.
- `-1`: this command instructs the robot to turn 90 degrees to its right.
- `1 <= k <= 9`: move the robot `k` units forward in the direction it's currently facing.

There may also be obstacles on the grid at various points that prevent the robot's movement. An obstacle is defined by its coordinates on the grid. When given a command to move, if the robot encounters an obstacle, it will not move and proceed to the next command.

The goal is to calculate the maximum squared Euclidean distance from the origin that the robot ever reaches during the simulation. The Euclidean distance measures the straight-line distance between two points, and in this problem, it's squared to avoid dealing with square roots.

Intuition

The intuition behind the solution is to simulate the movement of the robot on the grid step by step, while keeping track of the maximum squared distance from the origin it has achieved. To represent the four cardinal directions (north, east, south, and west), we use a `dirs` array. This array contains deltas, or changes in the `x` and `y` coordinates, corresponding to the four possible directions. The robot's direction can then be effectively accessed by keeping an index `k` into this array. The index is updated whenever the robot receives a turn command.

Along with simulating the robot turn directions, the solution must also take into account the obstacles. This is done using a set `s` to store the grid coordinates of all obstacles, allowing the robot to check whether its next move would result in a collision—in such a case, the robot's movement is halted for the current command.

Simulating each command involves checking the type of command and updating the robot's state accordingly — turning or moving forward. The key is to perform each forward movement step-by-step to check for potential collisions with obstacles at every single step instead of leaping the full commanded distance at once.

By keeping track of the current coordinates of the robot and updating the maximum distance (squared) when the robot moves to a new position, the algorithm ensures that the maximum value is always available. That way, when all commands have been processed, the maximum squared distance (which is our main concern) can be returned as the solution.

Solution Approach

The solution employs a hash table and simple simulation to track the robot's movements and handle obstacles efficiently. Here's a breakdown of how it is implemented:

- **Direction Array**: The `dirs` array `[(0, 1), (1, 0), (0, -1), (-1, 0)]` is cleverly designed so that pairs of adjacent elements correspond to directional vectors. Index `k` in this array corresponds to the current facing direction: `(dirs[k], dirs[k + 1])`. For instance, when `k = 0`, the direction is north `(0, 1)`, and when `k = 3`, the direction is east `(1, 0)`.
- **Obstacle Set**: A set `s` stores the coordinates of all obstacles. Rather than a list, a set is used for its O(1) average lookup time, which is critical for performance as the robot needs to quickly check for obstacles at each step.
- **Variables for State**: The `x` and `y` variables denote the current coordinates of the robot, whereas `k` represents the current facing direction. The `ans` variable keeps track of the maximum distance squared that the robot has been from the origin.
- **Command Processing**:
  - **Turning**: When a turn command `-2` (turn left) or `-1` (turn right) is encountered, the direction index `k` is updated modulo `4` to ensure it wraps around the `dirs` array correctly. Turning left (`-2`) results in `k` being incremented by 3 (equivalent to a 90-degree turn left), and modulo 4 ensures `k` stays within bounds. Turning right (`-1`) increments `k` by 1.
  - **Moving Forward**: On a forward move command denoted by a positive integer `c`, the robot moves one step at a time for `c` steps. For each step, temporary new coordinates `nx` and `ny` are calculated using the current direction `k`. The robot checks whether these new coordinates collide with an obstacle by checking the set `s`. If there is a collision, the move is aborted, and the next command is processed. Otherwise, the robot's position is updated, and the `ans` variable is updated with the squared distance from the origin if it's larger than the previous value.

The solution simulates the robot's movements exactly as specified, carefully considering obstacles, and calculates the maximum squared distance from the origin efficiently.

Finally, the solution returns the `ans` variable, which holds the maximum squared distance from the origin the robot has achieved during its journey.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. Assume the following inputs:

Commands: `[4, -1, 4, -2, 4]` Obstacles: `[[2, 4]]`

Now let's simulate the robot's behavior step by step:

1. The robot starts at the origin `(0, 0)` facing north. The maximum distance squared (`ans`) is currently `0`.
2. The first command is `4`. The robot moves forward 4 units to `(0, 4)`. No obstacles are encountered. The distance squared is now `0^2 + 4^2 = 16`, so `ans` is updated to `16`.
3. The next command is `-1`, a turn to the right. The robot now faces east but hasn't moved yet, so `ans` remains `16`.
4. The robot receives another `4` command to move forward. It will attempt to move from `(0, 4)` to `(4, 4)`, but there is an obstacle at `(2, 4)`. The robot stops before hitting the obstacle and remains at `(2, 4)`. The distance squared is `2^2 + 4^2 = 20`, so `ans` is updated to `20`.
5. The robot is given a `-2` command, which means turn left. Now it faces north again.
6. The last command is `4`, so the robot attempts to move four units north. However, since it is already at `(2, 4)`, it hits an obstacle immediately and does not move.

After processing all commands, the maximum distance squared the robot was from the origin is `20`. This is the final `ans` returned by our simulation.

Throughout the process, the robot kept checking for obstacles and updating its position and maximum distance accordingly, while the turns were managed efficiently by using the `dirs` array and updating the index `k`. This example successfully illustrates how the provided solution approach rightly calculates the maximum squared distance using the commands and obstacles given.

Python Solution

```python
class Solution:
    def robotSim(self, commands: List[int], obstacles: List[List[int]]) -> int:
        # Directions represent the movement of the robot: North, East, South, West
        directions = [0, 1, 0, -1, 0]
        # Set of obstacle positions for quick look-up
        obstacle_set = {(x, y) for x, y in obstacles}
        max_distance_squared = direction_index = 0
        position_x = position_y = 0

        # Loop through each command and execute
        for command in commands:
            if command == -2:  # Turn left
                direction_index = (direction_index + 3) % 4
            elif command == -1:  # Turn right
                direction_index = (direction_index + 1) % 4
            else:
                # Move the robot forward for the number of steps specified in the command
                for _ in range(command):
                    next_x = position_x + directions[direction_index], position_y + directions[direction_index + 1]
                    # Check if the new position is an obstacle
                    if next_x, next_y in obstacle_set:
                        break  # Stop if there's an obstacle ahead
                    # Update the robot's position
                    position_x, position_y = next_x, next_y
                    # Update the maximum Euclidean distance squared from the origin
                    max_distance_squared = max(max_distance_squared, position_x * position_x + position_y * position_y)

        # Return the maximum Euclidean distance squared the robot has been from the origin
        return max_distance_squared
```

Java Solution

```java
class Solution {
    public int robotSim(int[] commands, int[][] obstacles) {
        // Directions represent the movement of the robot: North, East, South, West
        int[] directionOffsets = {0, 1, 0, -1, 0};
        // Set to store obstacle coordinates in encoded form
        Set<Integer> obstacleSet = new HashSet<>(obstacles.length);
        // Encode and add obstacles to the set
        for (int[] obstacle : obstacles) {
            obstacleSet.add(encodeObstacle(obstacle[0], obstacle[1]));
        }
        int maxDistanceSquared = 0; // Will hold the maximum squared distance from the origin
        int directionIndex = 0; // Direction index points to the current direction
        int x = 0, y = 0; // Robot's starting position at the origin (0, 0)
        // Process each command to move the robot
        for (int command : commands) {
            if (command == -2) { // Turn left
                directionIndex = (directionIndex + 3) % 4;
            } else if (command == -1) { // Turn right
                directionIndex = (directionIndex + 1) % 4;
            } else {
                // Move the robot forward by the number of steps specified by command
                while (command-- > 0) {
                    int nextX = x + directionOffsets[directionIndex];
                    int nextY = y + directionOffsets[directionIndex + 1];
                    // Check if the next position is an obstacle
                    if (obstacleSet.contains(encodeObstacle(nextX, nextY))) {
                        break; // Stop moving if an obstacle is encountered
                    }
                    // Update the robot's position
                    x = nextX;
                    y = nextY;
                    // Calculate the squared distance from the origin and update maximum if necessary
                    maxDistanceSquared = Math.max(maxDistanceSquared, x * x + y * y);
                }
            }
        }
        return maxDistanceSquared; // Return the maximum squared distance from the origin
    }

    // Helper method to encode a 2D grid position into a single integer.
    // This is a common trick to store pairs as keys in a HashSet or HashMap.
    // Here, we make the assumption that -30000 <= x,y <= 30000.
    private int encodeObstacle(int x, int y) {
        return x + 60018 * y;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
#include <algorithm>

class Solution {
public:
    // This method simulates the movement of a robot according to given commands and obstacles.
    // Returns the maximum squared Euclidean distance from the origin a robot reaches.
    int robotSim(std::vector<int>& commands, std::vector<std::vector<int>>& obstacles) {
        // Direction vectors for north, east, south, west
        // 'directions[d]' and 'directions[d + 1]' is to allow easy access to 'y' direction with directions[k+1]
        int directions[5] = {0, 1, 0, -1, 0};

        // Function to encode a pair of coordinates into a single integer.
        auto hashFunction = [](int x, int y) {
            return x + 60018 * y;
        };

        // Set to hold the hashed locations of the obstacles.
        std::unordered_set<int> obstacleSet;
        for (const auto& obstacle : obstacles) {
            obstacleSet.insert(hashFunction(obstacle[0], obstacle[1]));
        }

        // Initialize variables to track the maximum Euclidean distance squared
        // and the current direction index and coordinates of the robot.
        int maxDistanceSquared = 0, directionIndex = 0;
        int x = 0, y = 0; // Robot starts at the origin (0,0)

        // Iterate through each command to move the robot.
        for (int command : commands) {
            if (command == -2) { // -2 means turn left 90 degrees
                directionIndex = (directionIndex + 3) % 4; // rotate left in the directions array
            } else if (command == -1) { // -1 means turn right 90 degrees
                directionIndex = (directionIndex + 1) % 4; // rotate right in the directions array
            } else {
                // Move forward by 'command' steps.
                while (command-- > 0) { // Loop over each step.
                    // Calculate the next x and y after moving a step in the current direction.
                    int nextX = x + directions[directionIndex];
                    int nextY = y + directions[directionIndex + 1];

                    // Check if next position is an obstacle.
                    if (obstacleSet.count(hashFunction(nextX, nextY)) {
                        break; // An obstacle blocks further movement in this direction.
                    }

                    // Update the robot's current position.
                    x = nextX;
                    y = nextY;

                    // Update the max distance squared if necessary.
                    maxDistanceSquared = std::max(maxDistanceSquared, x * x + y * y);
                }
            }
        }

        // Return the maximum Euclidean distance squared that the robot has been from the origin.
        return maxDistanceSquared;
    }
};
```

Typescript Solution

```typescript
function robotSim(commands: number[], obstacles: number[][]): number {
    // Directions array defines the delta changes for north, east, south, west.
    const directions = [0, 1, 0, -1, 0];

    // A Set to store the obstacles in a stringified form for quick look-up.
    const obstacleSet: Set<number> = new Set();

    // Helper function to encode an obstacle's x and y coordinate into a single number.
    const encodePosition = (x: number, y: number) => x + 60018 * y;

    // Add all obstacles to the Set using the encoding function.
    for (const [obstacleX, obstacleY] of obstacles) {
        obstacleSet.add(encodePosition(obstacleX, obstacleY));
    }

    // Initialize variables.
    // 'maxDistanceSquared' holds the maximum squared distance from the origin.
    // 'x' and 'y' represent the robot's current position.
    // 'directionIndex' represents the current direction index.
    let maxDistanceSquared = 0, x = 0, y = 0, directionIndex = 0;

    // Iterate over the commands.
    for (let command of commands) {
        if (command === -2) {
            // Turn left.
            directionIndex = (directionIndex + 3) % 4;
        } else if (command === -1) {
            // Turn right.
            directionIndex = (directionIndex + 1) % 4;
        } else {
            // Move forward 'command' steps.
            while (command-- > 0) {
                // Calculate the next position.
                const nextX = x + directions[directionIndex];
                const nextY = y + directions[directionIndex + 1];

                // Check if the next position is an obstacle.
                if (obstacleSet.has(encodePosition(nextX, nextY))) {
                    // Break if stay moving if next position is an obstacle.
                    break;
                }

                // Update the current position.
                x = nextX;
                y = nextY;

                // Calculate and update the maximum squared distance from the origin.
                maxDistanceSquared = Math.max(maxDistanceSquared, x * x + y * y);
            }
        }
    }

    // Return the maximum distance squared from the origin the robot can be.
    return maxDistanceSquared;
}
```

Time and Space Complexity

Time Complexity

The time complexity of the given code consists of iterating through each command and potentially iterating up to `c` times for each move command, where `c` is the max steps taken in any one move command.

Each turn command (-2 or -1) can be executed in constant time, `O(1)`. For each move command, we iterate up to `c` times, and since we have `n` commands, the max number of steps (`c`) could be performed potentially `n` times, leading to a `O(c * n)`.

Additionally, we check if the new position after each move is an obstacle; since we are using a set to contain obstacles and sets provide `O(1)` average time complexity for lookups, this does not significantly affect the overall time complexity per step.

Thus, the overall time complexity is `O(c * n + m)` where `n` is the total number of commands, and `m` is the total number of obstacles.

Space Complexity

The space complexity is determined by the storage used for the obstacle set and the few variables allocated for direction and position keeping, which are constant.

The set `s` constructed to contain the obstacles directly relates to the number of obstacles `m`. This takes up `O(m)` space as each obstacle needs one space in the set.

Other variables (`dirs`, `ans`, `k`, `x`, `y`, `nx`, `ny`) use a constant amount of space, so they do not scale with the input.

Therefore, the space complexity of the code is `O(m)`.