573. Squirrel Simulation

Medium Array Math

Problem Description

In this problem, you are given the dimensions of a garden that is `height` units tall and `width` units wide. There is a tree, a squirrel, and multiple nuts located at different positions within this garden. Your task is to calculate the minimal distance the squirrel must travel to collect all the nuts and bring them to the tree, one at a time. The squirrel can move in four directions - up, down, left, and right, moving to the adjacent cell with each move. The distance is measured in the number of moves the squirrel makes.

You are provided with:

- an array `tree` with two integers indicating the position (`tree_r`, `tree_c`), where `tree_r` is the row and `tree_c` is the column of the tree's location.
- an array `squirrel` with two integers indicating the position (`squirrel_r`, `squirrel_c`), where `squirrel_r` is the row and `squirrel_c` is the column of the squirrel's initial location.
- an array of `nuts`, where each element is an array `[nut_r, nut_c]` that provides the position of each nut in the garden.

The goal is to return a single integer that is the minimal total distance for all of this back-and-forth movement.

Intuition

The intuition behind the solution is to recognize that the problem is about optimizing the squirrel's path to minimize the distance traveled. One key observation is that, except for the first nut, the squirrel will be traveling from the tree to a nut and then back to the tree for every other nut. This means that for every nut except the first, it will cost double the distance from the tree to the nut since the squirrel must go to the nut and then return to the tree.

However, for the first nut, the squirrel doesn't start at the tree; it starts at its initial position. If the first nut that the squirrel collects is further from the tree than the squirrel's initial location, the squirrel could potentially reduce the total distance traveled by picking a nut that's closer to its starting location, even if that nut is slightly further from the tree.

To arrive at the solution approach, the solution calculates initially the sum of the distances for the squirrel to go from the tree to all the nuts and back to the tree, which would double the sum of the distances from the tree to each nut. Then, for each nut, it calculates the actual distance the squirrel would travel if it picked up that particular nut first, which involves going from its initial position to the nut, then to the tree, and subsequently to all other nuts and back to the tree (as already calculated).

The additional distance for the first nut is, therefore, the distance from the squirrel to the nut plus the distance from that nut to the tree. The optimization comes from finding the minimum of these additional distances by checking each nut to determine which one should be the first nut that reduces the total travel distance the most.

The solution iterates over all the nuts and calculates the difference between the optimized path (going first to each nut) and the non-optimized path (doubling the distance from the tree to each nut). The minimum of these differences, when added to the initially calculated sum, gives the result of the minimal total distance.

Solution Approach

The solution's main strategy is a greedy algorithm, which involves selecting the first nut to pick up based on which choice will minimize the overall distance traveled. To understand this strategy, let's break down the implementation step-by-step:

1. **Initialize variables**: We store the tree's coordinates in `x` and `y`, and the squirrel's starting coordinates in `a` and `b`.

```
x, y, a, b = *tree, *squirrel
```

2. **Precompute and Double the Constant Distances**: We compute the sum of all the distances from each nut to the tree. Since the squirrel must travel to each nut and then back to the tree, this distance is multiplied by 2.

```
s = sum(abs(i - x) + abs(j - y) for i, j in nuts) * 2
```

3. **Calculate the Minimum Additional Distance**: For each nut, we calculate two distances:
   - `c` is the distance from the current nut to the tree.
   - `d` is the distance from the squirrel's initial position to the current nut plus the distance from the current nut to the tree (which represents the path the squirrel takes if it chooses this nut first).

   We then compute the total distance if the squirrel chooses that nut first by adding `d` to the precomputed sum `s` and subtracting `c * 2` (since we had previously doubled the distance from this nut to the tree but now we need to replace one of those trips with the trip from the squirrel's starting position).

```
for i, j in nuts:
    c = abs(i - x) + abs(j - y)
    d = abs(i - a) + abs(j - b) + c
    ans = min(ans, s + d - c * 2)
```

4. **Find the Optimal First Nut**: The loop iterates over all nuts, finding the minimum possible total distance (`ans`) for the squirrel to collect all nuts and bring them to the tree by possibly choosing each nut as the first pickup. The first nut to be picked is implicitly chosen by finding the minimum value of `ans` over all possibilities.

5. **Return the Result**: Finally, the algorithm returns the minimum additional distance found, which, when added to the double sum of the constant distances, gives the minimum total distance needed to solve the problem.

By the end of the loop, `ans` holds the minimum total distance that factors in the optimal starting nut to pick up. This optimal path ensures that the squirrel does not waste additional distance on its first trip, thereby reducing the total distance over the entire nut collection process.

Example Walkthrough

Consider a scenario where our garden has dimensions `height` = 5 and `width` = 5, the tree is located at position (2, 2), the squirrel is at position (4, 4), and there are three nuts located at positions (0, 1), (1, 3), and (3, 2).

1. **Initialize variables**:

```
x, y = 2, 2 # Tree's position
a, b = 4, 4 # Squirrel's position
```

2. **Precompute and Double the Constant Distances**:
   - The distance for each nut to the tree would be:
     - For nut at (0, 1): `abs(0 - 2) + abs(1 - 2) = 3`
     - For nut at (1, 3): `abs(1 - 2) + abs(3 - 2) = 2`
     - For nut at (3, 2): `abs(3 - 2) + abs(2 - 2) = 1`
   - Sum of the distances = 3 + 2 + 1 = 6
   - Double the sum as the squirrel goes back and forth = `6 * 2 = 12`

```
s = 12 # Precomputed sum
```

3. **Calculate the Minimum Additional Distance**:
   - For each nut, calculate `c` (tree to nut) and `d` (squirrel to nut or nut to tree):
     - For nut at (0, 1):
       - `c = 3`
       - `d = abs(4 - 4) + abs(4 - 4) + c = 7` (distance from squirrel to nut) + 3 (nut to tree) = 10
       - `s + d - c * 2 = 12 + 10 - 3 * 2 = 16`
     - For nut at (1, 3):
       - `c = 2`
       - `d = abs(1 - 4) + abs(3 - 4) + c = 4` (distance from squirrel to nut) + 2 (nut to tree) = 6
       - `s + d - c * 2 = 12 + 6 - 2 * 2 = 14`
     - For nut at (3, 2):
       - `c = 1`
       - `d = abs(3 - 4) + abs(2 - 4) + c = 3` (distance from squirrel to nut) + 1 (nut to tree) = 4
       - `s + d - c * 2 = 12 + 4 - 1 * 2 = 14`
4. **Find the Optimal First Nut**:
   - The minimum distances calculated from picking each nut first are 16, 14, and 14, respectively.
   - Thus, the optimal first nut to pick up is either the one at (1, 3) or at (3, 2) since both result in a minimum additional distance of 14 (which is less than picking the nut at (0, 1) first).
5. **Return the Result**:
   - Since both of these nuts yield a minimum additional distance of 14, the squirrel can choose either of these as the first nut. Therefore, the result, which is the minimum total distance for the squirrel to collect all the nuts and bring them to the tree, is 14.

Solution Implementation

Python

```
from typing import List
from math import inf

class Solution:
    def minDistance(self, height: int, width: int, tree: List[int], squirrel: List[int], nuts: List[List[int]]) -> int:
        # Deconstruct the tree and squirrel coordinates for ease of use
        tree_x, tree_y = tree
        squirrel_x, squirrel_y = squirrel

        # Calculate the sum of distances from the tree to all nuts and back (doubled because of the round trip)
        total_distance = sum(abs(nut_x - tree_x) + abs(nut_y - tree_y) for nut_x, nut_y in nuts) * 2

        # Initialize the minimum distance as infinity
        min_distance = inf

        # Iterate through all the nuts to find the one with the minimum extra distance for the squirrel
        for nut_x, nut_y in nuts:
            # Distance from the tree to the current nut
            tree_to_nut = abs(nut_x - tree_x) + abs(nut_y - tree_y)

            # Distance from the squirrel to the current nut plus the distance from the current nut to the tree
            squirrel_to_nut_plus_nut_to_tree = abs(nut_x - squirrel_x) + abs(nut_y - squirrel_y) + tree_to_nut

            # Calculate the difference when the squirrel goes to this nut first instead of going to the tree
            # We replace one of the tree-to-nut round trips with squirrel-to-nut then nut-to-tree
            distance_diff = squirrel_to_nut_plus_nut_to_tree - tree_to_nut * 2

            # Update the minimum distance if we found a nut that results in a smaller extra distance for the squirrel
            min_distance = min(min_distance, total_distance + distance_diff)

        # The result is the smallest distance the squirrel needs to collect all nuts and put them in the tree
        return min_distance
```

Java

```
class Solution {
    public int minDistance(int height, int width, int[] tree, int[] squirrel, int[][] nuts) {
        int minimumDistance = Integer.MAX_VALUE;
        int totalDistanceToAllNuts = 0;

        // Calculate the total distance to collect all nuts and returning them to the tree, done twice (back and forth)
        for (int[] nut : nuts) {
            totalDistanceToAllNuts += calculateDistance(nut, tree);
        }
        totalDistanceToAllNuts *= 2;

        // Try each nut as the first nut to calculate the minimum distance needed
        for (int[] nut : nuts) {
            int distanceToTree = calculateDistance(nut, tree); // Distance from the current nut to the tree
            int distanceToSquirrel = calculateDistance(nut, squirrel) + distanceToTree; // Full trip from squirrel to nut and then to tree

            // Subtract the distance saved by the squirrel going directly to the nut, and then to the tree,
            // instead of going to the tree first.
            int currentDistance = totalDistanceToAllNuts + distanceToSquirrel - distanceToTree * 2;

            // Update the minimum distance if the current distance is less than what we have seen so far
            minimumDistance = Math.min(minimumDistance, currentDistance);
        }

        return minimumDistance;
    }

    // Helper method to calculate the Manhattan distance between two points a and b
    private int calculateDistance(int[] a, int[] b) {
        return Math.abs(a[0] - b[0]) + Math.abs(a[1] - b[1]);
    }
}
```

C++

```
class Solution {
public:
    // Calculate the minimum total distance the squirrel must travel to collect all nuts
    // and put them in the tree, starting from the squirrel's initial position.
    int minDistance(int height, int width, vector<int>& tree, vector<int>& squirrel, vector<vector<int>>& nuts) {
        int minimumDistance = INT_MAX;
        int totalDistanceToTree = 0;

        // Calculate the total distance for all nuts to the tree (each trip accounts for going to and returning from the tree).
        for (auto& nut : nuts) {
            totalDistanceToTree += distance(nut, tree);
        }
        totalDistanceToTree *= 2;

        // Evaluate each nut, calculating the distance savings if the squirrel starts from that nut.
        for (auto& nut : nuts) {
            int distanceToTree = distance(nut, tree);
            int distanceToSquirrel = distance(nut, squirrel);
            int currentDistance = distanceToSquirrel + distanceToTree; // Squirrel's first trip distance for this nut
            int distanceSaved = distanceToTree * 2; // The saved distance for fetching this nut last
            int totalCurrentDistance = totalDistanceToTree + currentDistance - distanceSaved;
            minimumDistance = min(minimumDistance, totalCurrentDistance);
        }

        return minimumDistance;
    }

    // Calculate the Manhattan distance between two points represented as vectors.
    int distance(vector<int>& pointA, vector<int>& pointB) {
        return abs(pointA[0] - pointB[0]) + abs(pointA[1] - pointB[1]);
    }
};
```

TypeScript

```
// Calculate the Manhattan distance between two points represented as arrays.
function distance(pointA: number[], pointB: number[]): number {
    return Math.abs(pointA[0] - pointB[0]) + Math.abs(pointA[1] - pointB[1]);
}

// Calculate the minimum total distance the squirrel must travel to collect all nuts
// and put them in the tree, starting from the squirrel's initial position.
function minDistance(height: number, width: number, tree: number[], squirrel: number[], nuts: number[][]): number {
    let minimumDistance = Number.MAX_SAFE_INTEGER;
    let totalDistanceToTree = 0;

    // Calculate the total distance for all nuts to the tree
    // (each trip accounts for going to and returning from the tree).
    for (const nut of nuts) {
        totalDistanceToTree += distance(nut, tree);
    }
    totalDistanceToTree *= 2;

    // Evaluate each nut, calculating the difference in distance if the squirrel goes to this nut first.
    nuts.forEach(nut => {
        const distanceToTree = distance(nut, tree);
        const distanceToSquirrel = distance(nut, squirrel);
        // Squirrel's first trip distance for this nut
        const currentDistance = distanceToSquirrel + distanceToTree;
        // The distance saved for not having to return from the tree
        const distanceSaved = distanceToTree * 2;
        // Total distance if starting from this nut
        const totalCurrentDistance = totalDistanceToTree - distanceSaved + currentDistance;
        minimumDistance = Math.min(minimumDistance, totalCurrentDistance);
    });

    return minimumDistance;
}

// Example usage:
// let minNuts = 5;
// let treePos = [2, 3];
// let squirrelStart = [4, 4];
// let nutPositions = [[3, 0], [2, 5]];
// let minDist = minDistance(minHeight, minWidth, treePos, squirrelStart, nutsPositions);
// console.log(minDist);
```

```
from typing import List
from math import inf

class Solution:
    def minDistance(self, height: int, width: int, tree: List[int], squirrel: List[int], nuts: List[List[int]]) -> int:
        # Deconstruct the tree and squirrel coordinates for ease of use
        tree_x, tree_y = tree
        squirrel_x, squirrel_y = squirrel

        # Calculate the sum of distances from the tree to all nuts and back (doubled because of the round trip)
        total_distance = sum(abs(nut_x - tree_x) + abs(nut_y - tree_y) for nut_x, nut_y in nuts) * 2

        # Initialize the minimum distance as infinity
        min_distance = inf

        # Iterate through all the nuts to find the one with the minimum extra distance for the squirrel
        for nut_x, nut_y in nuts:
            # Distance from the tree to the current nut
            tree_to_nut = abs(nut_x - tree_x) + abs(nut_y - tree_y)

            # Distance from the squirrel to the current nut plus the distance from the current nut to the tree
            squirrel_to_nut_plus_nut_to_tree = abs(nut_x - squirrel_x) + abs(nut_y - squirrel_y) + tree_to_nut

            # Calculate the difference when the squirrel goes to this nut first instead of going to the tree
            # We replace one of the tree-to-nut round trips with squirrel-to-nut then nut-to-tree
            distance_diff = squirrel_to_nut_plus_nut_to_tree - tree_to_nut * 2

            # Update the minimum distance if we found a nut that results in a smaller extra distance for the squirrel
            min_distance = min(min_distance, total_distance + distance_diff)

        # The result is the smallest distance the squirrel needs to collect all nuts and put them in the tree
        return min_distance
```

Time and Space Complexity

Time Complexity

The code performs a few distinct operations which each contribute to the overall time complexity:

1. **Initial Summation of Distances (`s` calculation)**: The code first calculates the sum of the double distances from the tree to all nuts. Since this operation is performed for each nut only once, it runs in $O(n)$, where n is the number of nuts.

2. **Main Loop Over Nuts**: The code then iterates over all the nuts to determine the minimum extra distance the squirrel has to travel for going to one of the nuts first before going to the tree. Each iteration includes constant time computations (addition, subtraction, and comparison). This loop runs in $O(n)$.

Therefore, the total time complexity of the given code is $O(n)$, dominated by the iterative operations done for each nut.

Space Complexity

The space complexity of the code can be analyzed based on the space used by variables that are not input-dependent:

- **Constant space for variables**: The variables `x`, `y`, `a`, `b`, `s`, and `ans` use constant space, $O(1)$.
- **No additional data structures**: There are no extra data structures that grow with the input size.

Hence, the space complexity of the code is $O(1)$.