1266. Minimum Time Visiting All Points

Easy Geometry Array Math Leetcode Link

Problem Description

The problem presents us with a collection of points on a 2D plane, with each point having integer coordinates given as a list, such as `points[i] = [xi, yi]`. The task involves finding the quickest way to visit all of these points sequentially.

When moving from one point to another, you are allowed to take one of three possible one-second movements:

1. Move one unit vertically.
2. Move one unit horizontally.
3. Move diagonally, which effectively means moving one unit vertically and one unit horizontally at the same time.

Considering these movement rules, the goal is to calculate the minimum total time in seconds required to visit all points in the order they are provided in the list.

Intuition

To solve this problem, we must figure out the fastest way to get from one point to the next. Because diagonal movements cover the maximum amount of distance in one second (both vertical and horizontal), they should be used whenever possible.

If we look carefully, we'll notice that when moving diagonally from one point to another, there will be a moment when we're aligned with the target point either horizontally or vertically. The number of diagonal moves we can make is determined by the maximum of the horizontal or vertical distance between the two points. Once aligned, if any distance remains, it will be strictly horizontal or vertical, which means we'll need additional moves to cover this remaining distance.

Taking all of this into account, the algorithm calculates the maximum of the horizontal and vertical differences for each consecutive pair of points in the provided list. Since the diagonal movement is equivalent to one second and it always reduces the maximum distance by 1, we can simply sum up these maximum differences to find the total time required.

Hence, the key to solving this problem is to iterate over the list of points, calculate the maximum difference in the x-axis and y-axis coordinates for each pair of points, and sum those differences to get the total minimum time.

Solution Approach

The solution uses a simple approach to iterate through the list of points and compute the time taken to travel from one point to the next.

The Python code provided employs a function `minTimeToVisitAllPoints` under a `Solution` class. This function makes use of a list comprehension along with the built-in Python function `sum` to accumulate a total.

Here's a step-by-step walkthrough of what the given solution does:

1. The function `minTimeToVisitAllPoints` accepts a list of points, where each point is itself a list containing the x and y coordinates.

2. The `max(abs(p1[0] - p2[0]), abs(p1[1] - p2[1]))` expression calculates the time required to move from point `p1` to point `p2`. This uses the concept that moving diagonally is equivalent to moving one unit horizontally and one unit vertically, hence the time to move diagonally is equal to the maximum of the horizontal and vertical distances.

   - `abs(p1[0] - p2[0])` computes the absolute difference between the x coordinates of `p1` and `p2`.
   - `abs(p1[1] - p2[1])` does the same for the y coordinates.
   - `max(...)` then takes the larger of the two distances to find out how many seconds it would take to travel from `p1` to `p2`.
3. The `pairwise(points)` function (which is likely assumed to be a custom or externally defined function that generates consecutive pairs of elements from the `points` list) is used to iterate through each pair of points in the order they appear.

4. For each tuple `(p1, p2)` where `p1` and `p2` are consecutive points, the aforementioned max expression calculates the required time to move from `p1` to `p2`.

5. The `sum` function adds together all the individual times calculated for moving between pairs of points, and this total represents the minimum time to visit all points in order.

The use of list comprehension along with `sum` makes for concise and efficient code. The underlying algorithm relies on the observation that, thanks to the rules of movement on the 2D plane, the time to move from one point to another is determined by the greater of the horizontal and vertical distances. By calculating and summing these times for each consecutive pair of points, the total travel time is found.

Note: The `pairwise` function is not a standard Python built-in function up to Python 3.9. It's available in the `itertools` module in Python 3.10. For earlier Python versions, a similar function can be implemented or imported from libraries such as `more_itertools`. If `pairwise` isn't available or defined, that line would raise an error.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Consider the following list of points on the 2D plane: `points = [[1, 1], [3, 4], [6, 6]]`.

- The first point is `(1, 1)`, and the second point is `(3, 4)`.

  - The horizontal distance between these two points is `abs(1 - 3) = 2`.
  - The vertical distance is `abs(1 - 4) = 3`.
  - The number of diagonal moves is the maximum of these two distances, which is `3`. We can move diagonally from `(1, 1)` to `(3, 3)` in 3 seconds and then one unit up to reach `(3, 4)`. Therefore, it takes a total of `3 + 1 = 4 seconds` to move from the first point to the second point.
- Now, let's move from the second point `(3, 4)` to the third point `(6, 6)`.

  - The horizontal distance is `abs(3 - 6) = 3`.
  - The vertical distance is `abs(4 - 6) = 2`.
  - Again, we will use the maximum of these two distances which is `3`. We can move diagonally from `(3, 4)` to `(6, 6)` in 3 seconds since both the horizontal and vertical distances decrease by 1 with each diagonal move.

Now we sum up the times it took to move between these pairs of points: `4 + 3 = 7 seconds`. According to the solution approach, it will take a minimum total of 7 seconds to visit all points in the order provided.

Breaking this down step-by-step based on the solution approach:

1. Start by looking at `points[0]` which is `[1, 1]`.

2. Moving from `points[0]` to `points[1]` (`[1, 1]` to `[3, 4]`):

   - Calculate the horizontal distance: `abs(1 - 3) = 2`.
   - Calculate the vertical distance: `abs(1 - 4) = 3`.
   - Use the maximum distance (vertical in this case) for diagonal movement: `max(2, 3) = 3`.
   - We can move diagonally for `3` seconds, and then we have `1` vertical unit to move up, so it takes `4` seconds total to reach the second point.
3. Next, moving from `points[1]` to `points[2]` (`[3, 4]` to `[6, 6]`):

   - Calculate the horizontal distance: `abs(3 - 6) = 3`.
   - Calculate the vertical distance: `abs(4 - 6) = 2`.
   - The maximum distance (horizontal in this case) is `3`, which is the time spent moving diagonally to reach the third point.
4. Add up the times for each move to get the total minimum time: `4 (first move) + 3 (second move) = 7 seconds`.

Using the given algorithm, the Python code to find this total minimum time would look like this:

```python
class Solution:
    def minTimeToVisitAllPoints(self, points):
        return sum(max(abs(p1[0] - p2[0]), abs(p1[1] - p2[1])) for p1, p2 in pairwise(points))
```

In this particular example, assuming `pairwise` is implemented correctly and works as intended, the function would return `7`.

Python Solution

```python
from typing import List

class Solution:
    def minTimeToVisitAllPoints(self, points: List[List[int]]) -> int:
        # Initialize the total time to 0.
        total_time = 0

        # A helper function to calculate the time to move from one point to another.
        # The time is determined by the maximum of the absolute horizontal (x-axis)
        # or vertical (y-axis) distances between two points, since one can move diagonally.
        def time_to_move(point1, point2):
            return max(abs(point1[0] - point2[0]), abs(point1[1] - point2[1]))

        # Iterate over each point and the next point in the list,
        # calculate the time to move between them, and add it to the total time.
        for i in range(len(points) - 1):
            total_time += time_to_move(points[i], points[i + 1])

        # Return the total time calculated.
        return total_time

# Example usage:
# solution = Solution()
# time_needed = solution.minTimeToVisitAllPoints([[1,1],[3,4],[-1,0]])
# print(time_needed)

```

Java Solution

```java
class Solution {
    public int minTimeToVisitAllPoints(int[][] points) {
        int totalTime = 0; // Initialize a variable to keep track of the total time

        // Iterate through all points starting from the second point
        for (int i = 1; i < points.length; ++i) {
            // Calculate the absolute difference in x-coordinates between the current point and the previous point
            int deltaX = Math.abs(points[i][0] - points[i - 1][0]);
            // Calculate the absolute difference in y-coordinates between the current point and the previous point
            int deltaY = Math.abs(points[i][1] - points[i - 1][1]);

            // The minimum time to move from one point to the next is the maximum of deltaX and deltaY
            // because we can move diagonally, which covers both x and y movement simultaneously.
            totalTime += Math.max(deltaX, deltaY);
        }

        // Return the total time calculated
        return totalTime;
    }
}

```

C++ Solution

```cpp
#include <vector> // Include the vector header for using the std::vector type
#include <cmath> // Include cmath for the abs() function

class Solution {
public:
    int minTimeToVisitAllPoints(std::vector<std::vector<int>>& points) {
        int totalTime = 0; // Initialize total time to 0
        // Iterate over each pair of consecutive points
        for (int i = 1; i < points.size(); ++i) {
            // Calculate the horizontal distance between the current point and the previous point
            int deltaX = std::abs(points[i][0] - points[i - 1][0]);
            // Calculate the vertical distance between the current point and the previous point
            int deltaY = std::abs(points[i][1] - points[i - 1][1]);

            // The time to move from one point to another is the maximum of deltaX and deltaY
            // because we can move diagonally, which covers one unit of both X and Y simultaneously
            totalTime += std::max(deltaX, deltaY);
        }
        // Return the total time calculated
        return totalTime;
    }
};

// Remember to also include the main function if you intend to execute this code.

```

Typescript Solution

```typescript
// Function to calculate the minimum time needed to visit all points
// in a 2D grid, moving in unit steps that can be diagonal, horizontal, or vertical.
function minTimeToVisitAllPoints(points: number[][]): number {
    let totalTime = 0; // Initialize total time to 0

    // Iterate over all the points except the first one
    for (let i = 1; i < points.length; i++) {
        // Calculate the horizontal distance between current point and the previous point
        let horizontalDistance = Math.abs(points[i][0] - points[i - 1][0]);
        // Calculate the vertical distance between current point and the previous point
        let verticalDistance = Math.abs(points[i][1] - points[i - 1][1]);

        // The time to move from one point to the next is determined by the maximum
        // of the horizontal and vertical distances because the movement can be diagonal
        totalTime += Math.max(horizontalDistance, verticalDistance);
    }
    // Return the total time
    return totalTime;
}

```

Time and Space Complexity

The time complexity of the code is `O(n)`, where `n` is the number of points. This is because the function iterates over `n-1` pairs of points (since each point is visited after the previous one, except for the first point which does not follow another point), and for each pair, it calculates the maximum of the absolute differences in x and y coordinates, which is done in constant time.

The space complexity of the code is `O(1)`. This is due to the fact that the sum is computed as the numbers are generated, and the extra space used does not grow with the input size. The variables `p1` and `p2` do not use additional space relative to the number of points because they are just references to the existing points in the input list.