1779. Find Nearest Point That Has the Same X or Y Coordinate

Easy Array

Problem Description

You are provided with your current coordinates `(x, y)` on a Cartesian grid. Additionally, you are given a list of points, each represented by a pair of coordinates `[a_i, b_i]`. A point is considered **valid** if has either the same x-coordinate as yours (shares the same vertical line) or the same y-coordinate (shares the same horizontal line). The goal is to find the closest valid point to you based on the **Manhattan distance**. The Manhattan distance between two points `(x_1, y_1)` and `(x_2, y_2)` is given by `abs(x_1 - x_2) + abs(y_1 - y_2)`, where `abs` stands for the absolute value function.

You need to return the smallest 0-indexed position of a point from the list that is valid and has the smallest Manhattan distance from your current location. If there is more than one valid point at the same minimum distance, return the one with the smallest index. If no valid points exist, return `-1`.

Intuition

To solve this problem, the straightforward approach is to iterate through the list of points, check for the conditions of validity (same x-coordinate or y-coordinate), and calculate the Manhattan distance between each valid point and your current coordinates.

While iterating, two things must be done:

1. Check if the current point is valid by comparing its coordinates with your coordinates `(x, y)`.
2. If valid, compute the Manhattan distance to the current point.

We maintain two variables, `mi` to track the minimum distance found so far, and `ans` to store the index of the point with that minimum distance. By default, `mi` is set to infinity (`inf`) as we are looking for the minimum distance, and `ans` is set to `-1` in case there are no valid points.

Each time we find a valid point with a Manhattan distance less than `mi`, we update `mi` with that distance and `ans` with the current index.

Solution Approach

The implementation of the solution uses a simple iterative approach with none of the more advanced algorithms or patterns. The main data structures used are the Python list to hold the points and a tuple to work with individual point coordinates.

Here's the breakdown of the implementation:

1. Initialize `ans` variable to `-1`. This will hold the index of the closest valid point, and if it stays `-1`, it means no valid point was found.
2. Initialize `mi` variable to `inf` (stands for infinity in Python), which will keep track of the smallest Manhattan distance found during iteration.
3. Iterate over each point using `enumerate` to get both the index `i` and the coordinates `(a, b)`.
4. Check if the current point is valid by comparing either the x-coordinate or the y-coordinate with the current location. This is done with the condition `if a == x or b == y:`.
5. If the point is valid, compute its Manhattan distance from the current location with `d = abs(a - x) + abs(b - y)`.
6. Compare the calculated distance `d` with the current minimum distance `mi`. If `d` is smaller, update `mi` to the new minimum distance and `ans` to the current index.
7. After exiting the loop, return `ans`, which represents the index of the point with the smallest Manhattan distance, or `-1` if no valid point was found.

This approach ensures that we are only considering valid points and always choose the valid point that is closest to the current location. Furthermore, since we update `ans` only when we find a closer valid point, if there are multiple points with the same distance, the first one found (and thus with the smallest index) will be chosen, consistent with the problem's requirement.

Example Walkthrough

Let's consider an example to illustrate the solution approach:

- Your current coordinates: `(1, 2)`
- List of points: `[[0, 3], [1, 9], [2, 3], [1, 4], [0, 2]]`

Following the steps outlined in the solution approach:

1. Initialize `ans` to `-1`.
2. Initialize `mi` to `inf`.
3. Start iterating over the list of points:
   - Index 0, Point `[0, 3]`: It's not valid since neither `0 == 1` nor `3 == 2`. Move to the next point.
   - Index 1, Point `[1, 9]`: This point is valid because it shares the same x-coordinate as yours (`1 == 1`). Calculate the Manhattan distance: `d = abs(1 - 1) + abs(9 - 2) = 7`. As `d` is smaller than `mi`, update `mi` to `7` and `ans` to `1`.
   - Index 2, Point `[2, 3]`: It's not valid since neither `2 == 1` nor `3 == 2`. Move to the next point.
   - Index 3, Point `[1, 4]`: This point is valid (`1 == 1`). Calculate the Manhattan distance: `d = abs(1 - 1) + abs(4 - 2) = 2`. Update `mi` to `2` and `ans` to `3` since `2` is smaller than the current `mi` of `7`.
   - Index 4, Point `[0, 2]`: This point is valid (`2 == 2`). Calculate the Manhattan distance: `d = abs(0 - 1) + abs(2 - 2) = 1`. Update `mi` to `1` and `ans` to `4` since `1` is smaller than the current `mi` of `2`.
4. Finish the iteration.
5. Return `ans` which is `4` as this is the point `[0, 2]` with the smallest Manhattan distance from your current location `(1, 2)`.

Using this example, we can see that the solution methodology effectively finds the closest valid point from the given list. It also displays that if multiple valid points have the same minimum distance, it shall return the point with the smallest index.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def nearestValidPoint(self, x: int, y: int, points: List[List[int]]) -> int:
        # Initialize the variable for the closest point index to -1 (no valid points by default)
        closest_index = -1
        # Initialize the variable for the minimum distance to infinity
        min_distance = float('inf')

        # Loop through each point and its index in the given list of points
        for index, (point_x, point_y) in enumerate(points):
            # Check if the current point is on the same x or y coordinate as the reference point (x, y)
            if point_x == x or point_y == y:
                # Calculate the Manhattan distance between the current point and the reference point
                distance = abs(point_x - x) + abs(point_y - y)
                # Check if the calculated distance is less than the current minimum distance
                if distance < min_distance:
                    # Update the closest index and minimum distance to the current index and distance
                    closest_index, min_distance = index, distance

        # Return the index of the nearest valid point
        return closest_index
```

Java

```java
class Solution {

    public int nearestValidPoint(int x, int y, int[][] points) {
        int nearestIndex = -1; // Initialize the nearest valid point index with -1
        int minimumDistance = Integer.MAX_VALUE; // Initialize minimum distance with maximum possible integer value

        for (int i = 0; i < points.length; ++i) {
            int currentX = points[i][0]; // Get the x-coordinate of the current point
            int currentY = points[i][1]; // Get the y-coordinate of the current point

            // Check if the current point is on the same x-axis or y-axis as the reference point (x,y)
            if (currentX == x || currentY == y) {
                int currentDistance = Math.abs(currentX - x) + Math.abs(currentY - y); // Calculate Manhattan distance from (x,y)

                // If the current distance is less than the previously recorded minimum distance
                if (currentDistance < minimumDistance) {
                    minimumDistance = currentDistance; // Update minimum distance
                    nearestIndex = i; // Update the index of the nearest valid point
                }
            }
        }

        return nearestIndex; // Return the index of the nearest valid point, or -1 if no valid point exists
    }
}
```

C++

```cpp
#include <vector>       // Include necessary header for vector
#include <cmath>        // Include cmath for the abs function
using namespace std;    // Use the standard namespace

class Solution {
public:
    // Function to find the nearest valid point to the point (x, y)
    // points: a 2D vector containing the other points' coordinates
    int nearestValidPoint(int x, int y, vector<vector<int>>& points) {
        int closestIndex = -1;              // Initialize closest point index as -1 (not found)
        int minDistance = INT_MAX;          // Initialize the minimum distance with a high value
        for (int i = 0; i < points.size(); ++i) {
            int currentX = points[i][0];    // X-coordinate of the current point
            int currentY = points[i][1];    // Y-coordinate of the current point
            // Check if the point is on the same axis as (x, y)
            if (currentX == x || currentY == y) {
                // Calculate the Manhattan distance between (x, y) and the current point
                int dist = abs(currentX - x) + abs(currentY - y);
                // If the distance is less than the minimum distance found so far
                if (dist < minDistance) {
                    minDistance = dist;      // Update the minimum distance
                    closestIndex = i;        // Update the index of the closest point
                }
            }
        }
        return closestIndex; // Return the index of the closest point, or -1 if no point is valid
    }
};
```

TypeScript

```typescript
function nearestValidPoint(x: number, y: number, points: number[][]): number {
    let nearestPointIndex = -1; // Initialized with -1 to indicate no valid point has been found
    let minimumDistance = Infinity; // Initialized to the largest possible value

    // Iterate over each point to find the valid nearest point
    points.forEach((point, index) => {
        const [pointX, pointY] = point; // Extract the x and y coordinates of the current point

        // Skip points that are not on the same x or y coordinate as the given point (x, y)
        if (pointX !== x && pointY !== y) {
            return; // Continue to the next iteration
        }

        // Calculate the Manhattan distance between the given point (x, y) and the current point
        const distance = Math.abs(pointX - x) + Math.abs(pointY - y);

        // If the current distance is less than the minimum found so far, update the result
        if (distance < minimumDistance) {
            minimumDistance = distance; // Update the minimum distance
            nearestPointIndex = index; // Update the index of the nearest valid point
        }
    });

    return nearestPointIndex; // Return the index of the nearest valid point or -1 if none found
}
```


```python
from typing import List

class Solution:
    def nearestValidPoint(self, x: int, y: int, points: List[List[int]]) -> int:
        # Initialize the variable for the closest point index to -1 (no valid points by default)
        closest_index = -1
        # Initialize the variable for the minimum distance to infinity
        min_distance = float('inf')

        # Loop through each point and its index in the given list of points
        for index, (point_x, point_y) in enumerate(points):
            # Check if the current point is on the same x or y coordinate as the reference point (x, y)
            if point_x == x or point_y == y:
                # Calculate the Manhattan distance between the current point and the reference point
                distance = abs(point_x - x) + abs(point_y - y)
                # Check if the calculated distance is less than the current minimum distance
                if distance < min_distance:
                    # Update the closest index and minimum distance to the current index and distance
                    closest_index, min_distance = index, distance

        # Return the index of the nearest valid point
        return closest_index
```


Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`, where `n` is the number of points in the input list `points`. This is because the function iterates through the list of points exactly once, performing a constant amount of work for each point by checking whether the point is valid and calculating the Manhattan distance if necessary.

Space Complexity

The space complexity of the code is `O(1)`. The extra space used by the algorithm includes a fixed number of integer variables (`ans`, `mi`, `a`, `b`, `d`), which do not depend on the size of the input. Since the amount of extra space required does not scale with the input size, the space complexity is constant.