String

Problem Description

Hash Table

In this LeetCode problem, we are given a string path that represents a sequence of moves. Each character in path stands for a directional move: 'N' for north, 'S' for south, 'E' for east, and 'W' for west. Each move is one unit long. We start at the origin point (0, 0) on a two-dimensional plane and follow the moves indicated in the path string. The task is to determine whether or not our path ever crosses itself. In other words, if we ever visit the same point more than once during our walk, we return true. If our path does not cross and we never visit the same point more than once, we return false.

ntuition

Easy

can use a set to store our visited positions because sets allow fast lookup times to check whether we have been to a position or not, as duplicates are not allowed in a set. We start by initializing our position to the origin (0, 0) and create an empty set called vis (short for "visited") which will hold

To solve this problem, the intuitive approach is to track every position we visit as we traverse the path defined by the string. We

tuples of our coordinates on the 2D plane. As we iterate over each move in the path string, we update our current position by incrementing or decrementing our i (for the north-south axis) and j (for the east-west axis) accordingly. After each move, we check whether the new coordinate (represented as a tuple (i, j)) is already present in our vis set. If it is, it

means we've just moved to a spot we've previously visited, which means our path has crossed, and we return true. If the coordinate is not in the set, we add it to the set and continue onto the next move in the path.

The solution to the problem implemented in Python uses a set data structure and simple coordinate manipulation to track the

Solution Approach

movement on the path. Below is an overview of the approach, breaking down how the algorithm works. 1. Initialize the current position to the origin, (i, j) = (0, 0). 2. Create a set named vis (short for visited) and add the initial position to it. Sets are chosen because they store unique elements, allowing us to

- quickly check if a position has been visited before.
- 3. Loop through each character in the path string:
- The for c in path: loop iterates over each character in the path string. • The match statement (a feature available in Python 3.10 and above) works like a switch-case statement found in other languages. It matches
 - the character c with one of the cases: 'N', 'S', 'E', or 'W'.
 - Based on the direction, we update our (i, j) coordinates:
 - For 'N', we decrement i to move north (i -= 1). For 'S', we increment i to move south (i += 1).
 - For 'E', we increment j to move east (j += 1).
 - For 'W', we decrement j to move west (j -= 1). 4. After updating the coordinates, we check if the new position (i, j) is already present in the vis set:
 - o If the condition (i, j) in vis: is True, we return True since the path has crossed a previously visited position.
 - 5. If the loop completes without finding any crossing, the return False statement at the end of the function ensures we return False, as no path has been crossed.
- due to the storage required for the set that holds the visited positions. **Example Walkthrough**

This approach uses straightforward coordinate tracking and set membership checks to efficiently solve the problem. The time

complexity is O(N), where N is the length of the path, since we visit each character once, and the space complexity is also O(N),

Following the solution approach, here's a step-by-step illustration of how the algorithm will execute: 1. We initialize our current position at the origin (0, 0), so (i, j) = (0, 0).

2. We create an empty set vis and add the initial position to it, so vis = $\{(0, 0)\}$.

Let's assume our given path string is "NESWW".

3. We start looping through each character in the path: \circ The first character is 'N'. We decrement i because we're moving north, so i = 0 - 1 = -1 and j remains 0. The new position is (-1, 0),

o If the position is not found in the set, we add the new position to the set with vis.add((i, j)).

- which is not in vis, so we add it: vis = $\{(0, 0), (-1, 0)\}$. ∘ The second character is 'E'. We increment j to move east, so i remains -1, and j = 0 + 1 = 1. The new position is (-1, 1), which is also not
- in vis, so we add it: vis = $\{(0, 0), (-1, 0), (-1, 1)\}$.
 - The third character is 'S'. We increment i to move south, so i = -1 + 1 = 0 and j remains 1. The new position is (0, 1), not in vis, so we add it: vis = $\{(0, 0), (-1, 0), (-1, 1), (0, 1)\}.$ The fourth character is 'W'. We decrement j to move west, so i remains 0, and j = 1 − 1 = 0. The position (0, 0) is already in vis,
 - indicating we've returned to the origin. Since this position is revisited, we would return True as the path crosses itself. Therefore, the function would return True based on the input path "NESWW", because we revisited the starting point, indicating a
- Solution Implementation

Python

x, y = 0, 0# set to keep track of visited coordinates

def isPathCrossing(self, path: str) -> bool:

iterate over each character in the path string

move in the corresponding direction

initialize starting point

if direction == 'N':

elif direction == 'S':

for direction in path:

x = 1

x += 1

```
visited = \{(0, 0)\}
```

crossing path.

class Solution:

```
elif direction == 'E':
                y += 1
           elif direction == 'W':
               y -= 1
           # check if the new position has already been visited
            if (x, y) in visited:
                # if we've been here before, path crosses. Return True
                return True
           # add the new position to the set of visited coordinates
            visited.add((x, y))
       # if visited all positions without crossing, return False
        return False
Java
class Solution {
   public boolean isPathCrossing(String path) {
       // Two variables to keep track of current position
       int x = 0, y = 0;
       // Use a HashSet to store visited coordinates.
       Set<Integer> visited = new HashSet<>();
       // Hash for the origin, adding it as the first visited coordinate
       visited.add(0);
       // Iterate over the path characters
        for (int index = 0; index < path.length(); ++index) {</pre>
            char direction = path.charAt(index);
```

switch (direction) {

break;

break;

y++;

int x = 0, y = 0;

} else {

```
X++;
                    break;
                case 'W': // Moving west decreases the x-coordinate
                    X--;
                    break;
            // Calculate a unique hash for the current position.
           // Multiplying by a large enough number to not mix coordinates
            int hash = y * 20000 + x;
           // Check if this position has been visited before, if so, path crosses
            if (!visited.add(hash)) {
                return true; // early return if the path crosses itself
       // If no crossing points were found, return false
       return false;
C++
#include <unordered_set>
#include <string>
class Solution {
public:
    // Determines if a path crosses itself based on commands in a string
    bool isPathCrossing(const std::string& path) {
       // Initialize (i, j) as the starting position (0, 0)
```

// Create a hash set to track visited positions with a unique key

std::unordered_set<int> visitedPositions{{0}};

// Update position based on direction

for (const char &direction : path) {

if (direction == 'N') {

--x; // Move north

++x; // Move south

++y; // Move east

position[1]++;

} else if (direction == 'S') {

} else if (direction == 'E') {

// Iterate through each character in the path string

// Move in the grid according to the current direction

case 'N': // Moving north decreases the y-coordinate

case 'S': // Moving south increases the y-coordinate

case 'E': // Moving east increases the x-coordinate

```
--y; // Move west
           // Calculate a unique key for the position
            int key = x * 20001 + y; // Use prime number to reduce collisions
           // Check if the position has been visited before
            if (visitedPositions.count(key)) {
               // If visited before, path crosses itself
               return true;
            // Add the new position to the set of visited positions
            visitedPositions.insert(key);
       // If no crossing occurred, return false
       return false;
TypeScript
function isPathCrossing(path: string): boolean {
   // Initialize current position at the origin (0,0)
    let position: [number, number] = [0, 0];
   // Create a set to store visited coordinates as a unique identifier
   const visited: Set<string> = new Set();
   // Add the starting position (origin) to the visited set
   visited.add(position.toString());
   // Iterate through each character in the path string
   for (const direction of path) {
       // Update the position according to the direction
       switch (direction) {
            case 'N': // North decreases the x coordinate
                position[0]--;
               break;
            case 'S': // South increases the x coordinate
                position[0]++;
               break;
            case 'E': // East increases the y coordinate
```

```
break;
              case 'W': // West decreases the y coordinate
                  position[1]--;
                  break;
          // Convert the tuple to a string to create a unique identifier for the position
          const positionKey = position.toString();
          // If the position has been visited, return true and exit
          if (visited.has(positionKey)) {
              return true;
          // Add the new position to the visited set
          visited.add(positionKey);
      // If no crossing paths are detected, return false
      return false;
class Solution:
   def isPathCrossing(self, path: str) -> bool:
       # initialize starting point
       x, y = 0, 0
       # set to keep track of visited coordinates
        visited = \{(0, 0)\}
       # iterate over each character in the path string
        for direction in path:
           # move in the corresponding direction
           if direction == 'N':
               x -= 1
            elif direction == 'S':
               x += 1
            elif direction == 'E':
               y += 1
           elif direction == 'W':
               y -= 1
           # check if the new position has already been visited
           if (x, y) in visited:
               # if we've been here before, path crosses. Return True
                return True
           # add the new position to the set of visited coordinates
            visited.add((x, y))
       # if visited all positions without crossing, return False
        return False
```

Time and Space Complexity The given Python code checks if a path crosses itself based on a string of movement commands ('N', 'S', 'E', 'W' corresponding to

Time Complexity: The time complexity of the code is O(n), where n is the length of the input string path. This is because the code iterates through each character of the path string exactly once.

For each character, the operations performed (updating coordinates and checking the set for the existence of the coordinates) are constant time operations, thus each character in the path requires a constant amount of time processing.

Space Complexity:

The space complexity of the code is O(n), where n is the length of the input string path. In the worst case, none of the positions will be revisited, so the set vis will contain a unique pair of coordinates for each move in the path. Thus, the maximum size of the set is proportional to the number of movements, which corresponds to the length of the path.

In summary, the code has a linear time and space complexity with respect to the length of the input path.

North, South, East, and West movements). The code uses a set vis to track all the visited coordinates.