469. Convex Polygon

Problem Description

Geometry

Medium

represents a point in the X-Y plane as $[x_i, y_i]$. The points are given in an order that represents the vertices of a polygon. We are asked to determine whether the polygon formed by these given points is a convex polygon.

A convex polygon is one in which all interior angles are less than 180 degrees and every line segment between two vertices of the

In this problem, we are given a set of points in the form of a two-dimensional array points, where each element points [i]

polygon stays inside or on the boundary of the polygon. This means, while traversing the polygon edges in a consistent direction (clockwise or counterclockwise), the direction in which we turn at each vertex to proceed to the next vertex should consistently be either all left (counter-clockwise turn) or all right (clockwise turn) for a convex polygon.

two edges meeting at each vertex.

The problem guarantees that the polygon is a simple polygon, meaning that the polygon does not intersect itself and has exactly

The intuition behind the solution is to check the sign of the cross product of consecutive edges of the polygon for the entire

direction (either all clockwise or all counterclockwise), then we have a convex polygon; otherwise, it is non-convex.

In mathematical terms, the cross product of two vectors can give us a sense of the rotational direction between them: if the cross product is positive, the second vector is rotated counterclockwise with respect to the first, and if negative - clockwise. When the

sequence of points. By doing this, we check the direction of the turn at each vertex. If at all vertices the turns are in the same

cross product is zero, the vectors are collinear.

To carry out this solution, the procedure is as follows:

1. Iterate through all vertices of the polygon.

4. If the cross product's sign changes at any point while we iterate through the vertices, this means that we've detected a change in the rotational direction, indicating that the polygon is not convey.

3. Compute the cross product of these two vectors.

direction, indicating that the polygon is not convex.

5. If we complete the iteration without detecting a sign change, the polygon is convex.

2. At each vertex, calculate the vectors (edges) that connect this vertex to the next one (i+1)%n and to the one after (i+2)%n.

Solution Approach

on is the number of points in the polygon.

The implementation follows directly from the established intuition. Since we are checking for convexity by looking at the rotational direction at each vertex, we are utilizing a very basic geometric algorithm that involves cross product calculation.

Here is a step-by-step breakdown of the code implementation:

Initialize variables pre and cur to 0. These will track the cross product of the current pair of consecutive edges and the

previous one, respectively.

2. Loop through each vertex in the polygon using for i in range(n):

Inside the loop, calculate two vectors (x1, y1) and (x2, y2):
The vector (x1, y1) extends from the current vertex points[i] to the next vertex points[(i + 1) % n].
The vector (x2, y2) extends from the current vertex points[i] to the vertex after the next points[(i + 2) % n].

4. Compute the cross product cur of these vectors using the determinant:
 ○ cur = x1 * y2 - x2 * y1

- 5. The very first non-zero cross product becomes the reference (pre) for the expected sign of all future cross products. This is
- because a non-zero cross product indicates the presence of an angle (not a straight line).
- convex, and the function returns False.

 If the signs are the same, update pre with the value of cur to compare with the cross product at the next vertex.

 After checking all vertices, if no sign change is detected, the function returns True, confirming that the polygon is convex.

This algorithm effectively uses a linear scan over the set of vertices and constant space for the variables. It reflects an efficient

On finding a non-zero cross product, check if the current cross product cur has a different sign from the previous, pre:

∘ If the signs differ (cur * pre < 0), then a change in the rotational direction has been detected, which implies that the polygon is not

Additionally, the code uses the modulo operator % which ensures that the indexing wraps around the list of points. This is especially useful for calculating vectors that involve the last point connecting to the first ((n-1) to 0). No additional data

structures or complicated patterns are used, making this solution straightforward and elegant in terms of space and time

approach to determining polygon convexity, with a time complexity of O(n) where n is the number of vertices.

Let's consider a given set of points for a quadrilateral: points = [(0, 0), (1, 1), (1, 0), (0, -1)].

Following the solution approach, we want to determine whether this quadrilateral is a convex polygon by examining the cross products of consecutive edges.

1. We initialize pre and cur to 0.

For i = 0: • The vector (x1, y1) is from points[0] to points[1], which is (1 - 0, 1 - 0) = (1, 1).

For i = 1:

For i = 2:

The number of points n is 4.

We begin looping through each vertex:

 \circ Since pre is zero, we set pre = cur, so now pre = -1.

prev_cross_product = current_cross_product = 0

Vector from current point to the next point

 $x1 = points[(i + 1) % num_points][0] - points[i][0]$

 $y1 = points[(i + 1) % num_points][1] - points[i][1]$

 $x2 = points[(i + 2) % num_points][0] - points[i][0]$

 $y2 = points[(i + 2) % num_points][1] - points[i][1]$

prev_cross_product = current_cross_product

Vector from current point to the point after the next

if current_cross_product * prev_cross_product < 0:</pre>

If all cross products have the same sign, the polygon is convex

Update previous cross product for the next iteration

// Function to determine if a given set of points constitutes a convex polygon

long long currentProduct = 0; // To store the current cross product

long long previousProduct = 0; // To store the last non-zero cross product

// Calculate vector from point[i] to point[i+1] (mod n for wrapping)

// Calculate vector from point[i] to point[i+2] (mod n for wrapping)

// Update previousProduct with the last non-zero cross product

// The polygon is convex if no sign change in cross products was detected

let prevCrossProduct: number = 0; // To store the last non-zero cross product

let currentCrossProduct: number = 0; // To store the current cross product

// Calculate vector from points[i] to points[(i+1) % n]

// Calculate vector from points[i] to points[(i+2) % n]

// Compute the cross product of the vectors

let deltaX1: number = points[(i + 1) % n][0] - points[i][0];

let deltaY1: number = points[(i + 1) % n][1] - points[i][1];

let deltaX2: number = points[(i + 2) % n][0] - points[i][0];

let deltaY2: number = points[(i + 2) % n][1] - points[i][1];

currentCrossProduct = deltaX1 * deltaY2 - deltaX2 * deltaY1;

int n = points.size(); // Number of points in the polygon

int deltaX1 = points[(i + 1) % n][0] - points[i][0];

int deltaY1 = points[(i + 1) % n][1] - points[i][1];

int deltaX2 = points[(i + 2) % n][0] - points[i][0];

int deltaY2 = points[(i + 2) % n][1] - points[i][1];

if (currentProduct * previousProduct < 0) {</pre>

// Compute the cross product of the vectors

// If the current cross product is non-zero

previousProduct = currentProduct;

// Typescript type to represent points as a 2D array of numbers

bool isConvex(vector<vector<int>>& points) {

for (int i = 0; i < n; ++i) {

if (currentProduct != 0) {

return false;

Compute cross product of the two vectors to find the orientation

Loop through each point

for i in range(num_points):

return False

efficiency.

Example Walkthrough

∘ The vector (x2, y2) is from points[0] to points[2], which is (1 - 0, 0 - 0) = (1, 0). ∘ The cross product cur is 1 * 0 - 1 * 1 = -1.

• The vector (x1, y1) is from points[2] to points[3], which is (0 - 1, -1 - 0) = (-1, -1).

• The vector (x2, y2) is from points[2] to points[0], which is (0 - 1, 0 - 0) = (-1, 0).

The vector (x1, y1) is from points[1] to points[2], which is (1 − 1, 0 − 1) = (0, −1).
The vector (x2, y2) is from points[1] to points[3], which is (0 − 1, −1 − 1) = (−1, −2).
The cross product cur is 0 * (−2) − (−1) * (−1) = −1.
cur has the same sign as pre, so we continue with pre still −1.

current_cross_product = x1 * y2 - x2 * y1 # If the cross product is not 0, check for convexity if current_cross_product != 0: # If cross products of adjacent edges have opposite signs, it's not convex

return True

Java

C++

public:

class Solution {

```
class Solution {
   public boolean isConvex(List<List<Integer>> points) {
        int numPoints = points.size(); // The number of points in the list.
        long previousProduct = 0; // To store the previous cross product.
        long currentProduct; // To store the current cross product.
       // Iterate through each set of three consecutive points to check for convexity.
        for (int i = 0; i < numPoints; ++i) {</pre>
           // Obtain three consecutive points (p1, p2, p3) with wrapping.
            List<Integer> point1 = points.get(i);
            List<Integer> point2 = points.get((i + 1) % numPoints);
            List<Integer> point3 = points.get((i + 2) % numPoints);
           // Compute the vectors from point1 to point2 and point1 to point3.
            int vector1X = point2.get(0) - point1.get(0);
            int vector1Y = point2.get(1) - point1.get(1);
            int vector2X = point3.get(0) - point1.get(0);
            int vector2Y = point3.get(1) - point1.get(1);
            // Compute the cross product of the two vectors.
            currentProduct = (long)vector1X * vector2Y - (long)vector2X * vector1Y;
           // If the cross product is non-zero (vectors are not collinear),
           // check if it has the same sign as the previous non-zero cross product.
            if (currentProduct != 0) {
                // If they have opposite signs, the polygon is non-convex.
                if (currentProduct * previousProduct < 0) {</pre>
                    return false;
                previousProduct = currentProduct; // Update previousProduct for the next iteration.
       // All consecutive triplets have the same sign of cross product, so the polygon is convex.
       return true;
```

currentProduct = static_cast<long long>(deltaX1) * deltaY2 - static_cast<long long>(deltaX2) * deltaY1;

// If the current and previous cross products have different signs, the polygon is non-convex

```
type Point = number[][];

// Function to determine if a given set of points constitutes a convex polygon
function isConvex(points: Point): boolean {
    let n: number = points.length; // Number of points in the polygon
```

TypeScript

};

return true;

for (let i = 0; i < n; ++i) {

```
// If the current cross product is non-zero
          if (currentCrossProduct !== 0) {
              // If the current and previous cross products have different signs, the polygon is non-convex
              if (currentCrossProduct * prevCrossProduct < 0) {</pre>
                  return false;
              // Update prevCrossProduct with the last non-zero cross product
              prevCrossProduct = currentCrossProduct;
      // The polygon is convex if no sign change in cross products was detected
      return true;
class Solution:
   def isConvex(self, points: List[List[int]]) -> bool:
       num_points = len(points) # Number of points in the polygon
       prev_cross_product = current_cross_product = 0
       # Loop through each point
        for i in range(num_points):
           # Vector from current point to the next point
            x1 = points[(i + 1) % num_points][0] - points[i][0]
            y1 = points[(i + 1) % num_points][1] - points[i][1]
           # Vector from current point to the point after the next
            x2 = points[(i + 2) % num_points][0] - points[i][0]
            y2 = points[(i + 2) % num_points][1] - points[i][1]
            # Compute cross product of the two vectors to find the orientation
            current cross product = x1 * y2 - x2 * y1
           # If the cross product is not 0, check for convexity
            if current_cross_product != 0:
               # If cross products of adjacent edges have opposite signs, it's not convex
               if current_cross_product * prev_cross_product < 0:</pre>
                    return False
               # Update previous cross product for the next iteration
               prev cross product = current cross product
       # If all cross products have the same sign, the polygon is convex
Time and Space Complexity
  The given Python code is designed to check if a polygon defined by a list of points is convex. Here's an analysis of its
  computational complexity.
```

The time complexity of the algorithm can be summarized as follows: 1. **Loop Over Points:** The code features a for-loop that iterates through the list of polygon vertices, where n is the number of points in the polygon. The loop runs exactly n times.

Time Complexity:

2. **Constant Time Operations:** Inside the loop, the code performs a constant number of operations. These operations include arithmetic calculations and if-conditions that do not depend on the size of the input. Therefore, these operations have a constant time complexity 0(1)

each iteration having O(1) operations. Thus, the total time complexity is O(n).

- constant time complexity 0(1).

 Considering the above points, the overall time complexity of the function is dominated by the for-loop, which runs n times, with
- Space Complexity:

Variables: The code uses a constant number of variables (n, pre, cur, x1, y1, x2, y2) regardless of the size of the input.

Input Storage: The space taken by the input list of points is not included in the space complexity analysis since it is a given

input to the function and not an additional space requirement created by the function.

As a result, the space complexity of the function is constant, 0(1), as it does not allocate any additional space that is dependent

Therefore, the final complexities are:

• Time Complexity: 0(n)

The space complexity of the algorithm is as follows:

Space Complexity: 0(1)

on the size of the input.