CSC282 HW2

Hanzhang Yin

Sep/1/2023

Question 1

Idea

This algorithm computes slopes between each base point and all other points, sorts these slopes to group colinear points, and checks for occurrences of identical slopes. By repeating this process for each base point and applying efficient sorting beforehand, the algorithm ensures all possible lines containing more than two points are identified.

Pseudocode

```
# Helper Functions
function mergeSort(array):
    # Base Case
    if length(array) <= 1:
        return array

    # Divide the array into halves
    mid = len(array) / 2
    leftHalf = mergeSort(array{0:mid})
    rightHalf = mergeSort(array[mid:length(array)])

    # Merge the two sorted halves
    return merge(leftHalf, rightHalf)

function merge(left, right):
    sortedArray = emptyList()
    # Pointers for LEFT and RIGHT halves
    i = 0, j = 0

    # Merge elements from both halves in sorted order
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            sortedArray.append(left[i])
```

```
            i++
        else:
            sortedArray.append(right[j])
            j++

    # Add any remaining elements from left half
    while i < length(left):
        sortedArray.append(left[i])
        i++

    # Add any remaining elements from right half
    while j < length(right):
        sortedArray.append(right[j])
        j++

    return sortedArray

# Main function
function findLinearWithMoreThanTwoPoints(points):
    n = len(points)

    for i = 1 to n:
        # Let base point be points[i] = (x_i, y_i)
        slopes = emptyList()

        # Calculate slopes with respect to the base point (x_i. y_i)
        for j = 1 to n:
            if i == j:
                # Skip if it's the same point
                continue
            # Get coordinates of points
            x_i, y_i = points[i]
            x_j, y_j = points[j]

            # Calculate slope between points[i] and points[j]
            if x_j == x_i:
                # Special case: vertical line
                slope = INFINITE
            else:
                slope = (y_j - y_i) / (x_j - x_i)

            # Store the slope
            slopes.append(slope)

        # Sort the slopes using merge sort
        sortedSlopes = mergeSort(slopes)
```

```
        # Traverse through sorted slopes to count consecutive occurrences
        counter = 1
        for k = 2 to len(sortedSlopes):
            if sortedSlopes[k] == sortedSlopes[k-1]:
                counter++
            else:
                # Reset Counter
                count = 1

    # Case where no line with more than two points found
    return False
```

Complexity Analysis

- For each base point (x_i, y_i), we compute the slop w.r.t. every other point. This takes $O(n)$ for each base point. The overall time complexity is $O(n^2)$.
- Sorting $n-1$ slopes using "merge sort" takes $O(nlogn)$ time. Noticing that we sort the slope for each of the n base points, the total time for sorting is $O(n \cdot (nlogn)) \Rightarrow O(n^2logn)$
- After sorting, counting consecutive occurrences of slopes to check for more than two points on a line takes $O(n)$ time per base point. For all base points, the algorithm takes $O(n^2)$ times.

Overall the time complexity of the algorithm is:

$$O(n^2) + O(n^2logn) + O(n^2) \sim O(n^2logn)$$

Question 2

Idea

In this algorithm, calculating the slope is effective because it allows us to determine the position of the point $p = (x, y)$ relative to each line segment. Hence, we can identify which region the end belongs to. By comparing the point's position w.r.t. the segments, and leveraging their ordered arrangement, we can use binary search to find the correct part efficiently.

Pseudocode

```
function findPartition(a, b, x, y, n):
    low = 1
    high = n
```

```
# Binary search loop
while low <= high:
    mid = (low + high) / 2

    # Find the line defined by (a[mid], 0) and (b[mid], 1)
    # Slope of the line: m = (1 - 0) / (b[mid] - a[mid]) = 1 / (b[mid] - a[mid])
    # Line equation: y_line = m * (x - a[mid])
    # Position of point p relative to this line:
    # y_line = (x - a[mid]) / (b[mid] - a[mid])

    y_line = (x - a[mid]) / (b[mid] - a[mid])

    if y < y_line:
        # Case when point p is below the line segment, updating to lower partition
        high = mid - 1
    else:
        # Case when point p is above the line segment, updating to higher partition
        low = mid + 1

# Invariant: point p is between partitions low and high
return low
```

Complexity Analysis

- Noting that Binary Search has the complexity $O(logn)$ since it divides the search space in half each time.
- Calculating the slope of the line only costs $O(1)$ time.

Overall the time complexity of the algorithm is:

$$O(logn) + O(1) \sim O(logn)$$

Question 3

For this question, we need to find the Gauss's Area Formula. The correct expression for the area of a simple polygon given its vertices in order is:

$$\frac{1}{2}\left|\sum_{i=0}^{n}(x_i y_{i+1} - y_i x_{i+1})\right|,$$

where $x_n = x_0$ and $y_n = y_0$. This is equivalent to the Shoelace formula and correctly computes the polygon's area by summing the signed areas of the trapezoids formed between each edge and the coordinate axes.

- **Expression 1** is correct because it matches Gauss's formula when $x_n = x_0$ and $y_n = y_0$, which properly accounts for the area by summing from 0 to n.
- **Expression 2** is incorrect because it takes the absolute value of each term individually, leading to a potential overestimation of the area.
- **Expression 3** is incorrect because it does not follow the correct mathematical structure for calculating the area of a polygon.

Question 4

Idea

This algorithm uses a binary search combined with the CCW test to find the left-most point of a convex polygon efficiently. By evaluating the orientation of three consecutive points, the algorithm determines whether to move left or right based on the CCW test result, narrowing down the range to locate the left-most point.

Pseudocode

```
# Helper function to compute CCW
function CCW(A, B, C):
    return (B[x] - A[x]) * (C[y] - A[y]) - (B[y] - A[y]) * (C[x] - A[x])

function findLeftMostPoint(points):
    n = len(points)

    # Init. binary search range
    low = 0
    high = n - 1

    # Perform binary search
    while low < high:
        mid = low + (high - low) // 2

        if CCW(points[mid-1], points[mid], points[mid+1]) > 0:
            # If counter-clockwise turn, mid is moving left
            high = mid
        else:
            # If clockwise turn or collinear, move right
            low = mid + 1

    # After the binary search is completed, low points to the left-most point
    return points[low]
```

Complexity Analysis

- By applying binary search in the algorithm, it costs $O(logn)$ to locate the target point.
- CCW algorithm has a time complexity around $O(1)$.

Overall the time complexity of the algorithm is:

$$O(logn) + O(1) \sim O(logn)$$

Extra Credit: Question 5

Idea

This algorithm uses the properties of reflection and perpendicular bisectors. By calculating the midpoint and slope of the perpendicular bisector of a line segment between two points, we can determine the required reflection line that maximized the reflection point pairs.

Pseudocode

```
# Helper Function
Slope_Calculator(dx, dy):
    perp_slope = float()
    if dx == 0:
        return perp_slope = INFINITY
    else if dy == 0:
        return perp_slope = 0
    else:
        return perp_slope = Fraction(-dx, dy)

    return NONE

# Main Function
function findMaxReflectingPairs(points):
    n = len(points)
    maxPairs = 0

    # Iterate all pairs of points
    for i in range(n):
        # Dict. to store the count of lines
        line_count = defaultdict(int)
```

```
        for j in range(n):
            if i == j:
                # Skip if it's the same point
                continue

                # Cal the midpoint between points[i] and points[j]
                mid_x = (point[i][0] - point[j][0]) / 2
                mid_x = (point[i][1] - point[j][1]) / 2

                # Cal. the slope opf the line pq
                dx = points[j][0] - points[i][0]
                dy = points[j][1] - points[i][1]

                perp_slope = Slope_Calculator(dx, dy)

                # Create unique key to represent the line
                line = (mid_x, mid_y, perp_slope)

                # Increment the count for this line
                line_count[line] += 1

            # Find the MAX number of pairs that can be reflected across a single line
            max_pairs = max(max_pairs, max(line_count.values(), default = 0))

    return max_pairs
```

Complexity Analysis

- The outer and inner nested for loop might cost $O(n^2)$ time
- The "Mid point" and "Perpendicular slope" calculation only costs $O(1)$
- In the worst cases of inserting and searching elements in a HashMap, we might need $O(logn)$ time to do so. (NOTE: this is placed inside the nested for loop.)
- The max function is an inherent library in Python for finding the max value which in the worst case might cost $O(n)$ times.

Overall, the time complexity of the algorithm is:

$$O(n^2) \cdot O(1) \cdot O(logn) + \sim O(n^2)$$