CSC282 HW2

Hanzhang Yin

Sep/9/2023

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
def mergeSort(lst):
    # Base Case: If the list has 1 or 0 elements, it is already sorted
    if len(lst) <= 1:
        return lst

    # Find the middle index to divide the list into two halves
    mid = len(lst) // 2
    # Recursively sort both halves
    leftHalf = mergeSort(lst[:mid])
    rightHalf = mergeSort(lst[mid:])

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

def merge(left, right):
    # Init. empty list
    sortedlst = []
    # Pointers for left and right halves
    i, j = 0, 0

    # Merge elements from both halves in sorted order
    while i < i < len(left) and j < len(right):
```

```
        if left[i] <= right[j]:
            sortedlst.append(left[i])
            i += 1
        else:
            sortedlst.append(right[j])
            j += 1

    # Add any remaining elements from the left half
    while i < len(left):
        sortedlst.append(left[i])
        i += 1

    # Add any remaining elements from the right half
    while j < len(right):
        sortedlst.append(right[j])
        j += 1

    return sortedlst

# Main function
function findLinearWithMoreThanTwoPoints(points):
    n = len(points) - 1

    for i = 0 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 = 0 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 = 0
        for k = 1 to len(sortedSlopes) - 1:
            if sortedSlopes[k] == sortedSlopes[k-1]:
                counter += 1
                # Found a line with more than two points
                if counter >= 1:
                    return True

    # 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 cross product (i.e. using CCW algorithm) is effective because it allows us to determine the position of the point $p = (x, y)$ relative to other points. Hence, we can identify which region the target point belongs to. By comparing the point's position value (NOTE: $value < 0$ indicates point is to the left of the line) , we can use binary search to find the correct part efficiently.

Pseudocode

```
# Helper Function
function CCW(Point_a, Point_b, Point_p):
```

```
    (x1, y1) = Point_a
    (x2, y2) = Point_b
    (x, y) = Point_p

    # Compute the cross product to determine orientation
    return (x - x1) * (y2 - y1) - (y - y1) * (x2 - x1)

# Main Function
function findPartition(a, b, x, y):
    Point_p = (x, y)
    low = 0
    high = len(a) - 1

    # Binary search loop
    while low < high:
        mid = (low + high) // 2
        Point_a = (a[mid], 0)
        Point_b = (b[mid], 1)

        if CCW(Point_a, Point_b, Point_p) > 0:
            # Point is to the left of the line, search in the left half
            high = mid
        elif CCW(Point_a, Point_b, Point_p) < 0:
            # Point is to the right of the line, search in the right half
            low = mid + 1
        else:
            # Point is exactly on the line
            return mid

    # point p lies with the interval ({a[low], 0}, {b[low], 1}) AND ({a[high], 0}, {b[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 CCW from point only costs $O(1)$ time.

Overall the time complexity of the algorithm is:

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

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 since it uses sums and differences of coordinates instead of the cross products needed to compute the polygon's area according to the Shoelace formula. This incorrect formula fails to properly account for the geometric properties and orientation of the polygon.

Question 4

Idea

By leveraging the convex nature of the polygon, the algorithm iteratively narrows down the search range by comparing the x-coordinates of the current midpoint with its neighboring points. If the midpoint's x-coordinate is less than both of its "left" and "right" neighbors, it is the left-most point. Otherwise, the algorithm continues searching in the appropriate half based on the x-coordinate comparisons, ensuring a rapid convergence to the solution.

Pseudocode

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

    # Initialize the binary search range
    low = 0
    high = n - 1

    # Perform binary search
    while low < high:
```

```
        mid = (low + high) // 2

        # Compare x-coordinates to find the minimum
        if points[mid].x < points[(mid - 1 + n) % n].x AND points[mid].x <
        points[(mid + 1) % n].x:
            # Found the left-most point
            return points[mid]

        elif points[mid].x > points[(mid - 1 + n) % n].x:
            # If the point to the left has a smaller x-coordinate,
            search the left half
            high = mid - 1
        else:
            # Otherwise, search the right half
            low = mid + 1

    # When low meets high, the left-most point is found
    return points[low]
```

Complexity Analysis

- By applying binary search in the algorithm, it costs $O(logn)$ to locate the target point.
- x-coordinates comparison calculation only costs $O(1)$.

Overall the time complexity of the algorithm is:

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

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):
    if dx == 0:
        return INFINITY
    else if dy == 0:
```

```
        return 0
    else:
        return perp_slope = -dx // dy

    return NONE

# Main Function
function findMaxReflectingPairs(points):
    n = len(points) - 1
    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 = (points[i][0] + points[j][0]) // 2
            mid_y = (points[i][1] + points[j][1]) // 2

            # Cal. the slope of 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 dictionary, we might need $O(logn)$ time to do so. (NOTE: this is operation is placed inside the nested for loop.)
- The max function is from inherent library (e.g. in Python) for finding the max value which in the worst case might cost $O(n)$ times. (NOTE: this is operation is placed inside the outer for loop.)

Overall, the time complexity of the algorithm is:

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