ASSIGNMENT 6 REPORT

Assignment number: 6

Name: Sai Kiran Nandipati

ULID: snandip

Secret directory name: sai

Summary:

I used divide-and-conquer approach to identify the closest pair of 2D points efficiently. When dealing with only a few points (3 or fewer), I directly find the closest pair using a simple method. For larger sets, it divides the points into two groups based on their x-coordinates and recursively finds the closest pairs within each group. It then calculates the minimum distance between these closest pairs, referred to as 'delta.' Additionally, the algorithm selects a vertical line to separate the points into two sets, creating a list of points (S) that are within a delta distance from this line. Within this list, it efficiently determines the closest pair of points using a heuristic and stores the result as SRes. Finally, the algorithm compares the minimum distances between SRes, QRes, and RRes, selecting the pair with the smallest distance as the overall closest pair. This approach minimizes the number of distance calculations and greatly enhances the efficiency of finding the closest pair of 2D points."

Data structures used:

1. **Point Class:** This is a custom data structure to represent 2D points. It contains two attributes, x and y, which store the coordinates of a point. The Point class also has a method to calculate the distance between two points using the Euclidean distance formula.

2**. ClosestPairResult Class**: Another custom data structure used to store the result of the closest pair search. It contains three attributes: point1 and point2, which represent the closest pair of points found, and distance, which stores the distance between these points.

3. **Arrays:** The code utilizes arrays to store and manipulate sets of Point objects. These arrays are sorted and split to facilitate the divide-and-conquer algorithm's operations.

4. **Lists:** In the code, java.util.ArrayList is used to create a dynamic list of points. This list is employed to collect points within a certain range during the divide-and-conquer algorithm's execution.

Pseudo code:

Class Point

The Point class represents a point with x and y coordinates. It has the following attributes and methods:

- Attributes:
  - x (a double representing the x-coordinate)
  - y (a double representing the y-coordinate)

- Constructor:
  - The constructor initializes a Point object with given x and y coordinates.

- Method distance(other):
  - This method calculates and returns the Euclidean distance between the current point and another point 'other'.

Class ClosestPairResult

The ClosestPairResult class is used to store the result of the closest pair of points. It has the following attributes:

- Attributes:
  - point1 (the first point in the closest pair)
  - point2 (the second point in the closest pair)
  - distance (the distance between the closest pair of points)

- Constructor:
  - The constructor initializes a ClosestPairResult object with the given points and their distance.

Main Algorithm: findClosestPair(P)

The primary algorithm for finding the closest pair of points in a set P is as follows:

1. Sort the points in P by their x-coordinates and store the result in Px. This sorting operation has a runtime of O(n log n).

2. Call the closestPairRecursive function with the sorted P and Px as input to find the closest pair. The recursive algorithm follows a divide-and-conquer approach, similar to merge sort, with a runtime of O(n log n).

3. Return the result, which contains the closest pair of points and their distance.

Recursive Algorithm: closestPairRecursive(P, Px)

The closestPairRecursive function is a recursive algorithm that finds the closest pair of points within a subset of the input points. The algorithm consists of the following steps:

1. If the number of points in the subset is less than or equal to 3, the closest pair is found using a brute force approach, which has a constant runtime of O(1).

2. Divide the subset into two halves, Q and R, based on the x-coordinates. Then sort both Q and R by their x-coordinates.

3. Recursively find the closest pair for Q and R using closestPairRecursive.

4. Calculate the minimum distance between the results obtained for Q and R.

5. Create a list SList to store points within a strip of width 2 * delta centered on the vertical line at xStar. This process has a runtime of O(n).

6. Sort the points in SList by their y-coordinates.

7. Iterate through the sorted points in SList and calculate the distance between each pair of points. Keep track of the closest pair within the strip.

8. Finally, compare the results from SRes, QRes, and RRes to determine the overall closest pair, and return the result.

Summary of methods:

1**. sortByX(Point[] points):** This method takes an array of points and arranges them in ascending order of their x-coordinates, making it easier to process and find the closest pair efficiently.

2. **sortByY(Point[] points):** Similar to the previous method, this one sorts the points, but this time by their y-coordinates, enabling operations based on y-coordinate sorting.

3. **bruteForceClosestPair(Point[] P):** When dealing with a small number of points (3 or less), this method employs a straightforward approach to directly compute and return the closest pair of points.

4. **closestPairRecursive(Point[] P, Point[] Px):** At the heart of the code, this method employs a divide-and-conquer approach. It recursively partitions the points into subsets, identifies the closest pairs within each subset, and ultimately finds the overall closest pair.

5. **findClosestPair(Point[] P):** Serving as a wrapper for the recursive method, this function first sorts the input points by their x-coordinates and then invokes the recursive algorithm. It also measures the runtime for performance evaluation.

6. **main(String[] args):** The central piece of the program, this method handles reading sets of points from an input file, processing each set to find the closest pair, and subsequently printing the results. It relies on the assistance of other methods to accomplish these tasks, resulting in a complete and efficient solution for finding the closest pair of 2D points.

Time and space complexity analysis:

Time Complexity analysis:

1. **Sorting Step:** The code begins by sorting the input points by their x-coordinates using the sortByX method. This step has a time complexity of O(n * log(n)), where 'n' is the number of points. Sorting is often the most time-consuming step in the algorithm.

2. **Recursive Divide-and-Conquer:** The main part of the algorithm is the closestPairRecursive method. When there are more than three points in the set, it recursively divides the points into two halves and finds the closest pair within each half. This recursive division is performed until the base case is reached, where the set is small enough to use a brute-force approach.

- The divide step (splitting the points into two subsets) takes O(n) time since we're iterating through all the points once to create two subsets.

- The conquer step (finding closest pairs in each subset) involves two recursive calls. The total work done in these recursive calls is proportional to T(n) = 2 * T(n/2), where 'T(n)' represents the time taken for 'n' points. By solving this recurrence, it's determined that the time complexity for this part is O(n * log(n)).

3. Merging Step: After the recursive calls, there's a step where the code merges the results from the left and right subsets. This step also takes O(n) time, as each point is examined to ensure no closer pair exists across the subsets.

4. Efficient Pair Identification in S: In the closestPairRecursive method, there's a step where it efficiently identifies the closest pair of points within a delta distance of a vertical line. This step takes O(k^2) time, where 'k' is a small constant (e.g., 15) representing the number of points to consider within the delta range.

Overall, the dominant time complexity comes from the sorting and the recursive divide-and-conquer steps, resulting in a time complexity of O(n * log(n)), which is the standard time complexity for the closest pair problem. The other steps contribute smaller terms and constants to the overall time complexity.

Space Complexity:

1. Arrays and Lists: The code uses arrays and lists to store the points and subsets of points. The space complexity for these data structures is O(n) since they store all the input points.

2. Recursion Stack: During the recursive divide-and-conquer process, a stack is used to keep track of function calls. In the worst case, the maximum depth of the recursion is O(log(n)), leading to a space complexity of O(log(n)) for the recursion stack.

In summary, the space complexity is primarily dominated by the input data structures and the recursion stack, resulting in a space complexity of O(n) + O(log(n)), which can be simplified to O(n) since it is the larger of the two terms.