BLG 336E Analysis of Algorithms 2 Assignment 2

Due: Tuesday, April 9, 2024

Lecturer: Assoc. Prof. Dr. Mehmet Baysan CRN: 21349

Fatih Baskın
150210710

Contents

1 Pseudocode of Functions

1.1 CompareX

```
FUNCTION CompareX(p1, p2):
  // p1, p2: points containing x and y coordinates
  // Only 1 operation, O(1) complexity
  RETURN p1.x < p2.x
```

This function is used to compare points about their x coordinates and for sorting these points about their x coordinates. Time complexity is O(1) since it does only one comparison operation.

1.2 CompareY

```
FUNCTION CompareY(p1, p2)
  // p1, p2: points containing x and y coordinates
  // Only 1 operation, O(1) complexity
  RETURN p1.y < p2.y
```

This function is used to compare points about their y coordinates and for sorting these points about their y coordinates. Time complexity is O(1) since it does only one comparison operation.

1.3 Distance

```
FUNCTION distance(p1, p2):
  // p1, p2: points containing x and y coordinates
  // Only 4 operations, O(1) complexity
  x_diff <- p1.x - p2.x
  y_diff <- p1.y - p2.y
  // Calculate euclidian distance using differences
  squared_diffs <- (x_diff * x_diff) + (y_diff * y_diff)
  RETURN sqrt(squared_diffs)
```

This function is used to compare the distance between two points and has O(1) time complexity since it does only 4 fundamental operations.

1.4 Brute Force Closest Pair

```
ALGORITHM bruteForceClosestPair(Points, start, end):
  // Points: an array containing points
  // start: starting index of this function on this array
  // end: ending
  // Initially, minimum distance and answer is uninitialized
  min_dist <- infinity
  min_pair <- TIL
  // Then, try each combination of points in range [start, end]
  // Time complexity is O(n^2) because of nested loops
  FOR i <- start to end:
    FOR j <- i + 1 to end:
      // Calculate distance of current pair
      curr_dist <- distance(Points[i], Points[j])
      // If current pair distance is smaller than the previous minimum
      // then set current pair as minimum
      IF curr_dist < min_dist:
        // Update min dist
         min_dist <- curr_dist
        // Take current points
        p1 <- points[i]
        p2 <- points[j]
        // While saving the current pair, save smallest y, x first
```

```
      IF p1.y < p2.y:
        min_pair <- makePair(p1, p2)
      ELSE IF p1.y == p2.y:
        IF p1.x < p2.x
          min_pair <- makePair(p1, p2)
        ELSE:
          min_pair <- makePair(p2, p1)
      ELSE:
        min_pair <- makePair(p2, p1)
  // Finally, return minimum pair
  RETURN min_pair
```

This function has $O(n^2)$ time complexity since it has two nested loops and checks every possible pair of points. For n points, there are $\frac{n\times(n-1)}{2}$ combinations of pairs, therefore this function does about $c \times n^2$ operations where $c > 0$.

1.5 Divide and Conquer Closest Pair

```
ALGORITHM closestPair(Points, start, end):
  // Points: an array containing points
  // start: starting index of this function on this array
  // end: ending
  // Recursive algorithm, time complexity: O(n logn) using master theorem
  // T(n) = 2T(n/2) + n logn -> O(n logn)
  // Base case: start > end, there are 0 elements
  IF start > end:
    // Return invalid pair
    RETURN makePair(TIL, TIL)
  // Base case: start == end, there are 1 element
  IF start == end:
    // Return single point and invalid point
    RETURN makePair(Points[start], TIL)
  // Base case: 2 elements
  IF start + 1 == end:
    // Take these points
    p1 <- points[start]
    p2 <- points[end]
    // While saving the current pair, save smallest y, x first
    IF p1.y < p2.y:
      RETURN makePair(p1, p2)
    ELSE IF p1.y == p2.y:
      IF p1.x < p2.x:
        RETURN makePair(p1, p2)
      ELSE:
        RETURN makePair(p2, p1)
    ELSE:
      RETURN makePair(p2, p1)
  // Base case: 3 elements, 3 combinations, use brute force, can be considered O(1)
  IF start + 2 == end:
    // Use brute force to return closest pair of 3
    // Can be considered constant time, since O(3^2) = O(9) = O(1)
    RETURN bruteForceClosestPair(points, start, end)
  // Mid point to divide array into two
  mid <- (start + end) / 2
  // Calculate the closest pair for left half and right half
  pair_l <- closestPair(Points, start, mid)
  delta_l <- distance(pair_l.first, pair_l.second)
  pair_r <- closestPair(Points, mid + 1, end)
  delta_r <- distance(pair_r.first, pair_r.second)
  // Get the minimum delta from left and right halves
  delta_min <- min(delta_l, delta_r)
```

```
// Determine the elements that are delta_min distance away from the
// mid point by considering the x coordinate
mid_point <- points[mid]
INITIALIZE Strip // Array to keep track of points in the strip
// This step has O(n) complexity
FOR i <- start to end:
  p <- points[i]
  // If X coordinate is mid - delta <= x <= mid + delta range, add it to strip
  IF p.x <= (delta_min + mid_point.x) AND p.x >= (mid_point.x - delta_min):
    Strip.add(p)
// Calculate the minimum pair from left and right halves
min_pair <- TIL // Initially, uninitialized
IF delta_l <= delta_r:
  min_pair <- pair_l
ELSE
  min_pair <- pair_r
// Then, determine the minimum pair using the strip.
// If strip consists of only one element, then skip this step.
IF size(Strip) > 1:
  // First, sort the strip about elements' y coordinate using CompareY function.
  // This step has O(n logn) complexity
  Strip <- sort(Strip, CompareY)
  // Then compare a point with its next 7 elements
  // This step has O(n) complexity, since it does 7 operations n times.
  FOR i <- 1 to size(Strip):
    // The limit is the strip size or i + 7
    FOR j <- i + 1 to min(size(Strip), i + 7):
      // Take the current points
      p1 <- Strip[i]
      p2 <- Strip[j]
      // Calculate the distance of the current pair
      curr_dist <- distance(p1, p2)
      // If the current pair distance is smaller than the previous minimum,
      // then set the current pair as the minimum
      IF curr_dist < delta_min:
          // Update delta min
          delta_min <- curr_dist
          // While saving the current pair, save the smallest y, x first
          IF p1.y < p2.y:
            min_pair <- makePair(p1, p2)
          ELSE IF p1.y == p2.y:
            IF p1.x < p2.x:
              min_pair <- makePair(p1, p2)
            ELSE
              min_pair <- makePair(p2, p1)
          ELSE
            min_pair <- makePair(p2, p1)
// Finally, return the minimum pair.
RETURN min_pair
```

This function is used to find the smallest distance pair recursively. It has $O(n \times log(n))$ time complexity and $O(n)$ space complexity (due to strip array). It takes Points array, which is sorted about its axis, then divides this array from the middle and calls itself recursively. These functions return the smallest pair in the left hand side and right hand side. But algorithm did not compare the points close to the border between left and right side.

A distance δ is calculated, from the minimum of δ_L and δ_R, distance between minimum pair of points of left and right hand sides. Then, points that lie δ distance away from the middle x point, $x_{mid} - \delta \leq x_{point} \leq x_{mid} + \delta$, are found and saved in a different array, called **strip**. This has $O(n)$ time complexity and $O(n)$ space complexity since all n points might lie in the strip. Then, this strip array is sorted about its y axis, this step has $O(n \times log(n))$ time complexity.

Then, finally, for each element in the strip, its distance with its 7 next neighbours are calculated and if they are smaller than the previous minimum pair, pair from left hand side and right hand side, they are saved as the minimum pair. The comparison is done with 7 next neighbours because if you think of the strip, it is $2 \times \delta$ wide area, and smaller distance points might lie in the $\delta \times 2\delta$ rectangle area. This rectangle has 4 corners and cuts the middle line in 2 points, then we have 7 points of interest. Also, since it is possible that there might be very close or coincident point to our current point, therefore we need to make 7 checks. Therefore, in total, there are $7 \times n$ operations, with $O(n)$ time complexity.

Overall, the recurrence equation can be written as $T(n) = 2 \times T(\frac{n}{2}) + (O(n) + O(n \times log(n)) + O(n))$ and can be simplified down to $T(n) = 2 \times T(\frac{n}{2}) + O(n \times log(n))$. Using the master theorem, complexity of this recurrence equation is $O(n \times log(n))$. Also, since we are saving a strip array, it has $O(n)$ space complexity.

1.6 Remove Pair

```
FUNCTION removePair(Points, point_pair)
  // O(n) time complexity
  // An array to to save points that are not in the pair.
  INITIALIZE updated_points
  FOR EACH Point p IN Points:
    // If a point is not in the pair, add it to the new vector.
    IF p != point_pair.first AND p != point_pair.second:
      updated_points.add(p)
  RETURN updated_points
```

This algorithm has $O(n)$ time complexity since it traverses the entire points array. It also has $O(n)$ space complexity since it saves all the points that are not in the point pair.

1.7 Find Closest Pair Order

```
ALGORITHM findClosestPairOrder(Points):
  // Time complexity is O(n^2 logn) for divide and conquer, O(n^3) for brute force
  INITIALIZE pairs // An array to save closest pairs
  unconnected <- TIL // Unconnected point if it exists, initially uninitialized
  // Sort points about their x axis, using CompareX function
  Points <-  sort(Points, compareX)
  // Until there are 0 or 1 points left, find closest pair
  while size(Points) > 1:
    // Get the closest pair using recursive divide and conquer algorithm.
    closest_pair <- closestPair(points, 1, size(Points))
    // Save the closest pair and remove it from the elements
    pairs.add(closest_pair)
    Points <- removePair(Points, closest_pair)
  // If there is one point left in the points, set it as unconnected
  if size(Points) == 1
    unconnected <- points[1]
  // Print the results
  Print pairs
  Print unconnected
```

This algorithm is used to find all closest pairs. This algorithm will run divide and conquer or brute force method $\frac{n}{2}$ times therefore it will have $O(n^2 \times log(n))$ or $O(n^3)$ time complexity respectively. This is the overall runtime complexity of this program.

This algorithm will use $O(n)$ space complexity since it requires auxiliary space while sorting, removing from array and during divide & conquer algorithm

2 Questions and Answers

2.1 Time and Space Complexity of Brute Force Algorithm

Brute force approach tries every possible combination of points, therefore it tries $C(n, 2) = \frac{n \times (n-1)}{2}$ combinations. Therefore this algorithm's runtime is proportional to n^2. Then the time complexity is $O(n^2)$.

This algorithm does not use any auxiliary space since it uses nested loops to try all combinations, therefore it has $O(1)$ space complexity.

2.2 Time and Space Complexity of Divide and Conquer Algorithm

This algorithm has $O(n \times log(n))$ time and $O(n)$ space complexity. The time complexity was thoroughly explained in the pseudocode explanation. It has $T(n) = 2 \times T(\frac{n}{2}) + O(n \times log(n)$ recurrence equation and using master theorem, it can be said that it has $O(n \times log(n))$ time complexity.

To store the elements located in the strip, this algorithm uses an auxiliary space, If we assume all the points are located in the strip (worst case) then this algorithm has $O(n)$ space complexity.

2.3 Timelapse of Brute Dorce and Divide and Conquer Algorithms

Using the **chrono** library in C++, timelapse of each algorithm is recorded in different test cases. Note that since this algorithm tries to find all closest pairs possible, it runs $\frac{n}{2}$ times, therefore overall time complexity is $O(n^3)$ for brute force approach and $O(n^2 \times log(n))$ for divide and conquer algorithm.

Test Case	Number of Points	Runtime: Divide & Conquer	Runtime: Brute Force
case0.txt	7	27415 ns	17850 ns
case1.txt	10	73485 ns	34979 ns
case2.txt	100	4397355 ns	5135665 ns
case3.txt	401	92123485 ns	141469689 ns
case4.txt	1000	544355756 ns	1793133142 ns

Table 1: Timelapses of Divide & Conquer and Brute Force algorithms under different test cases

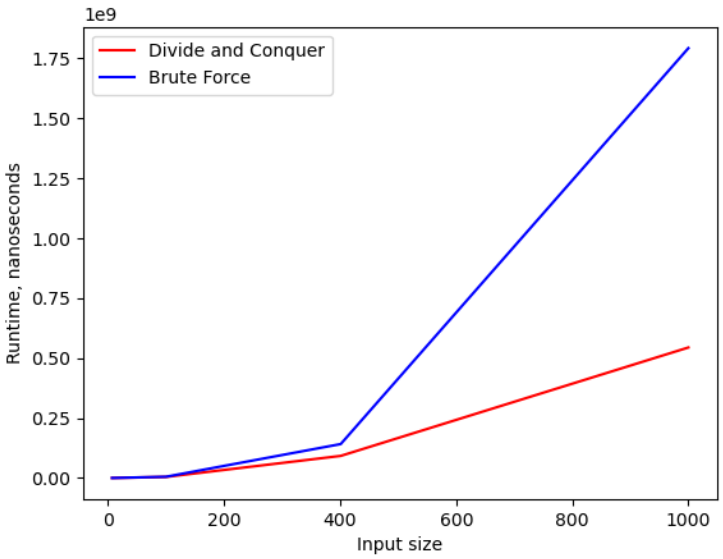

Figure 1: Graph of timelapses of Divide & Conquer and Brute Force algorithms

2.4 Manhattan Distance

If the manhattan distance was used, the results would definitely change, since manhattan distance is the sum of the absolute values of differences of coordinates. Mathematically:

- Absolute value of the differances of x coordinates: a
- Absolute value of the differances of y coordinates: b
- Manhattan distance: $a + b = \sqrt{(a+b)^2} = \sqrt{a^2 + 2 \times a \times b + b^2}$
- Euclidean distance: $\sqrt{a^2 + b^2}$

Manhattan distance has extra term, which is $a \times b$. This favors unproportunate a and b values. For example:

Let's say we got two different pairs, first pair has $a_1 = 6$ and $b_1 = 3$. Second pair has $a_2 = 7$ and $b_2 = 1$. Then manhattan distance of first pair is $a_1 + b_1 = 6 + 3 = 9$ and manhattan distance of second pair is $a_2 + b_2 = 7 + 1 = 8$. According to manhattan distance, second pair is closer than the first pair.

However, with using euclidean distance, first pair is closer than the second pair since euclidean distance of the first pair is $\sqrt{{a_1}^2 + {b_1}^2} = \sqrt{6^2 + 3^2} = \sqrt{36 + 9} = \sqrt{45}$ and the euclidean distance of the second pair is $\sqrt{{a_2}^2 + {b_2}^2} = \sqrt{7^2 + 1^2} = \sqrt{49 + 1} = \sqrt{50}$.

Therefore the results would definitely change if we used manhattan distance instead of euclidean distance.