# BLG 336E- Analysis of Algorithms II Homework 2 Report

Özkan Gezmiş 150200033

**a) pseudo-code:**

```
// Function to calculate the distance between two points
distance(Point p1, Point p2):
   x_diff = AbsoluteValue(p1.x - p2.x)
   y_diff = AbsoluteValue(p1.y - p2.y)
   dist = SquareRoot(x_diff^2 + y_diff^2)
   Return dist

// Function to compare points based on x-coordinate
compareX(Point p1, Point p2):
   Return p1.x <= p2.x

// Function to compare points based on y-coordinate
compareY(Point p1, Point p2):
   Return p1.y <= p2.y

// Function to find the closest pair of points using brute force
bruteForceClosestPair(points, start, end):
   min_distance = Infinity
   min_pair = EmptyPair
   For i from start to end:
      For j from i+1 to end:
         dist = distance(points[i], points[j])
         If dist < min_distance:
            min_distance = dist
            min_pair = (points[i], points[j])
   Return min_pair

// Function to find the closest pair of points recursively using divide and conquer
closestPair(points, start, end):
   n = end - start
   If n <= 3:
      Return bruteForceClosestPair(points, start, end)

   middle = start + n/2
   pair1 = closestPair(points, start, middle)
   pair2 = closestPair(points, middle, end)

   dist1 = distance(pair1.first, pair1.second)
   dist2 = distance(pair2.first, pair2.second)
   min_pair = pair1 if dist1 < dist2 else pair2
   min_dist = Minimum(dist1, dist2)
```

```
    new_vector = []
    For each point in points:
        If AbsoluteValue(point.x - points[middle].x) <= min_dist:
            Add point to new_vector

    Sort new_vector based on y-coordinate

    max_check_num = 7
    min_distance = min_dist
    For i from 0 to size(new_vector):
        For j from i+1 to Minimum(i+max_check_num, size(new_vector)):
            dist = distance(new_vector[i], new_vector[j])
            If dist < min_distance:
                min_distance = dist
                min_pair = (new_vector[i], new_vector[j])

    Return min_pair

// Function to remove a pair of points from the point vector
removePairFromVector(point_vector, point_pair):
    position1 = 0
    position2 = 0
    For i from 0 to size(point_vector):
        If point_vector[i] == point_pair.first:
            position1 = i
        If point_vector[i] == point_pair.second:
            position2 = i

    If position1 > position2:
        Swap position1 and position2

    Erase point_vector[position2]
    Erase point_vector[position1]

    Return point_vector

// Function to find the closest pair of points, add them to pairs vector, and set unconnected
point
findClosestPairOrder(points):
    pairs = EmptyVector
    unconnected = (-1, -1)
    brute_force = False

    Sort points based on x-coordinate

    If brute_force:
        While size of points >= 2:
```

```
            closest_pair = bruteForceClosestPair(points, 0, size of points - 1)
            Add closest_pair to pairs
            points = removePairFromVector(points, closest_pair)
        If size of points == 1:
            Set unconnected as the remaining point

    Else:
        While size of points >= 2:
            closest_pair = closestPair(points, 0, size of points - 1)
            Add closest_pair to pairs
            points = removePairFromVector(points, closest_pair)
        If size of points == 1:
            Set unconnected as the remaining point

    Print pairs
    If unconnected exists:
        Print unconnected

// Function to read coordinates from a file and convert them to a vector
readCoordinatesFromFile(filename):
    points = EmptyVector
    If filename is empty:
        Print "Filename is empty!"
        Exit program
    Open file with filename
    If file cannot be opened:
        Print "File cannot be opened!"
        Exit program

    For each line in file:
        Read x and y coordinates from line
        Add (x, y) to points

    Return points

// Main function
main(argc, argv):
    points = readCoordinatesFromFile(argv[1])

    start = CurrentTime
    findClosestPairOrder(points)
    end = CurrentTime

    total_time_ms = CalculateDuration(start, end)
    Print "CPU total execution time:", total_time_ms, "milliseconds"
    Return 0
```

**b) Complexity of the functions:**

1. **distance**() function calculates the distance between two points using the Euclidean distance formula.
   - Time Complexity: O(1)

2. **compareX**() and **compareY**() functions compare two points based on their x or y coordinates.
   - Time Complexity: O(1)

3. **bruteForceClosestPair**() function finds the closest pair of points within a given range using brute force. It compares every pair of points within that range.
   - Time Complexity: $O(n^2)$, where n is the number of points in the given range.

4. **closestPair**() function finds the closest pair of points within a given range using a divide-and-conquer approach.
   - Time Complexity: O(n log n), where n is the number of points in the given range. This complexity arises due to sorting the points and the divide-and-conquer algorithm.

5. **removePairFromVector**() function removes a pair of points from the given vector.
   - Time Complexity: O(n), where n is the number of points in the vector. Since, it traverses the entire vector and removes the given pairs. Both traversing the vector and removing the pair take O(n) time.

6. **findClosestPairOrder**() function finds all pairs in increasing distance either with brute force or divide and conquer approach.
   - Each call to "**bruteForceClosestPair**" takes $O(n^2)$ and "**closestPair**" takes O(n logn) within the loop. The number of iterations of the loop in `findClosestPairOrder` is also proportional to n because in each call to `bruteForceClosestPair` or `closestPair` reduces the size of the input points by only two. Thus, the total number of iterations is O(n).
   - If brute force approach is used then Time Complexity is $O(n * (n^2)) = O(n^3)$.
   - But if divide and conquer approach is used then Time Complexity would be $O(n * (n \log n)) = O(n^2 * \log n)$.
   - However, the actual time complexity may vary depending on the input data.

7. **readCoordinatesFromFile**() reads coordinates from a file and converts them to a vector.
   - Time Complexity: O(n), where n is the number of coordinates in the file. This is because it needs to read each coordinate from the file.

**time and space complexity of the divide & conquer algorithm:**
**Time complexity:**
- We recursively divide the set of points into halves until the base case is reached.
- We sort the vector before we call brute force or divide and conquer algorithms, therefore, sorting according to x coordinates is done one time. However, sorting with respect to y is done in each iteration, since to be able to calculate the distances in the strip we need sorted list. But, the element in the strip is generally so small so it doesn't take much time.
- At each recursion level, we perform several operations:
    Finding the closest pair of points in the left and right halves (which together take O(n) time).
    Merging the results from the left and right halves.

- At each level of recursion, the work done is O(n) for finding the closest pair of points in each half. The number of levels of recursion is O(log n), as we divide the set of points in half at each level until the base case is reached. Therefore, the total time complexity of the divide-and-conquer algorithm is the product of the work done at each level and the number of levels:
- Since finding the closest pair takes O(n log n) and we call it for n/2 times the **Total Time Complexity = O(n log n) * O(n) = O($n^2$ * logn)** where n is the number of points in the set.

**Space Complexity:**
**The space complexity of the divide-and-conquer algorithm is O(n),** where n is the number of points.
- This space complexity is due to the additional space required for recursion, including the stack space for function calls and any auxiliary data structures used during the computation.

**time and space complexity of the brute force algorithm:**
**Time Complexity:**

- In the brute force approach, we compare every pair of points in the given set to find the closest pair. This results in considering every possible pair of points exactly once. Therefore, the time complexity of the each brute force call is O($n^2$).
- Since finding the closest pair takes O($n^2$) and we call it for n/2 times the **Total Time Complexity = O($n^2$) * O(n) = O($n^3$)** where n is the number of points in the set.

**Space Complexity:**
**The space complexity of the brute force approach is O(1)**, as it doesn't require any additional space proportional to the size of the input.

- The algorithm operates in place, performing comparisons directly on the given set of points without needing any extra memory.

**Performance of the algorithms:**

| Number of points / Algorithm | N=7 | N=10 | N=100 | N=1000 |
|---|---|---|---|---|
| Brute Force | 0ms | 68ms | 2358ms | 35990ms |
| Divide and Conquer | 0ms | 14ms | 249ms | 1811ms |

Since divide and conquer has $O(n^2*\log n)$ and brute force has $O(n^3)$ time complexities, divide and conquer approach is faster than brute force. If we mathematically calculate the difference when n = 100 the rate would be: $(10^3)^3 / ((10^3)^2 * \log 1000)$ which is approximately equal to 100, which is consistent with my results.

**Would the results change if we used Manhattan distance instead of Euclidean distance? How?**

Of courses it would change.
For example, we can think 3 points which are (0,4), (4,0) and (11,0).
If we use Manhattan distance closest pair is (4,0) and (11,0). However, if we use Euclidian distance then closest pair would be (0,4), (4,0).

Since, in the Manhattan distance it just calculate the difference between the numbers, whereas in the Euclidian distance the hypotenuse may be smaller.