Süleyman KORAMAZ
1901042615

CSE321 HOMEWORK 5

(1)
- The algorithm uses a divide-and-conquer approach to efficiently find the minimum distance between any two drones on a 2D plane.
- It sorts the drons based on their x-coordinates to simplify the problem.
- It recursively divides the set of drones into two halves, solves the problem for each half, and then combines the results.
- The crucial step is finding the minimum distance between drones in the two halves and checking for drones close to the dividing line.

```
procedure find_closest_pair(drones):
    n = drones.length
    if n <= 3: // If there are fewer than 3 drones, brute force check (base case)
        return minimum_distance( calculate_distance(drones[i], drones[j]) for i in range(n) for j in range(i+1, n))
    end if
    drones.sort(key=lambda drone: drone.x) // Sort drones based on x-coordinates
    //Recursively find the closest pair in the left and right halves
    mid = n/2
    left_half = drones[:mid]
    right_half = drones[mid:]
    min_left = find_closest_pair(left_half)
    min_right = find_closest_pair(right_half)
    min_distance = minimum(min_left, min_right) //Minimum distance among the left and right halves
    //Check for drones with x-coordinates close to the middle
    strip = [drone for drone in drones if absolute_value(drone.x - drones[mid].x) < min_distance]
    strip.sort(key=lambda drone: drone.y) //Sort the strip based on y-coordinates
    min_strip = minimum_distance(calculate_distance(strip[i], strip[j]) for i in range(strip.length) for j in range(i+1, strip.length))
    return minimum(min_distance, min_strip)
end
procedure calculate_distance(drone1, drone2):
    return square_root((drone1.x - drone2.x)**2 + (drone1.y - drone2.y)**2)
end
procedure closest_pair_of_drones(drones):
    drones.sort(key=lambda drone: drone.x)
    return find_closest_pair(drones)
end
```

Time Complexity:

- Divide and Conquer part: $T(n) = 2*T(n/2) + O(n) \rightarrow a=2,\ b=2,\ f(n)=n,\ d=1 \rightarrow a = b^d \rightarrow O(n \log n)$
- Sorting processing: The initial sorting step of the drones based on x-coordinates takes $O(n \log n)$ time, where n is # of drones

Therefor, the overall time complexity of the given algorithm is $O(n \log n)$

(2)

```
procedure secure_primeter(sensors):
    if sensors.length == 0:
        return 0
    else if sensors.length == 1:
        return 1
    end if
    sensors.sort(key=lambda sensor: sensor.x) // Sort sensors based on x-coordinates
    mid = sensors.length / 2
    left_half = sensors[:mid]
    right_half = sensors[mid:]

    // Recursively find the minimum sensors for each half
    left_min = secure_primeter(left_half)
    right_min = secure_primeter(right_half)

    // find the sensors on the dividing line
    mid_x = sensors[mid].x
    dividing_line = [sensor for sensor in sensors if absolute_value(sensor.x - mid.x) < radius]
    dividing_line.sort(key=lambda sensor: sensor.y) // Sort the dividing line based on y-coordinates
    min_line = min_sensors_dividing_line(dividing_line) // Calculate the minimum sensors for the dividing line

    return left_min + right_min + min_line
end
procedure min_sensors_dividing_line(dividing_line):
    min_sensors = 1
    rightmost_sensor = dividing_line[0]
    for sensor in dividing_line[1:]:
        if sensor.y - radius > rightmost_sensor.y:
            min_sensors += 1
            rightmost_sensor = sensor
        end if
    end for
    return min_sensors
end
```

- The algorithm aims to find the minimum number of sensors needed to establish a secure primeter around a campsite.
- It uses a divide-and-conquer approach, sorting sensors based on their coordinates.
- Recursively, it calculates the minimum sensors needed for the left and right halves and then determines the minimum for the dividing line.

Time Complexity:

- Sorting sensors based on coordinates: $O(n \log n)$
- Recursive calls on halves: $\log n$ levels, and at each level, we do constant work.

Resulting in a time complexity of $O(n \log n)$

③ • The algorithm aligns two DNA sequences with minimum cost, considering insertion, deletion and substitution.
• It uses a dynamic programming table to iteratively calculate the minimum cost for aligning prefixes of the sequence.
• The table entry at dp[i][j] represents the minimum cost to align prefixes seq1[0...i] and seq2[0...j]

Steps:

1 - Initialize a dynamic programming table with dimensions (m+1) x (n+1), m and n are length of sequences.
2 - Fill the table iteratively, considering the costs of insertion, deletion and substitution.
3 - The value in the bottom-right cell (dp[m][n]) represents the minimum cost for aligning the entire sequences.

```
procedure align_dna_sequence(seq1, seq2):
    m, n = seq1.length, seq2.length
    dp = [[0]*(n+1) for i in range(m+1)] // Initialize the dynamic programming table

    // Fill the table iteratively
    for i in range(m+1):
        for j in range(n+1):
            if i==0:
                dp[i][j] = j*3 // Insertion
            else if j==0:
                dp[i][j] = i*3 // Deletion
            else:
                insertion = dp[i][j-1] +3  //Cost of insertion
                deletion = dp[i-1][j] +3 //Cost of deletion
                substitution = dp[i-1][j-1] + ((seq1[i-1] != seq2[j-1]) ? 3 : 0)//Cost of substitution

                dp[i][j] = min(insertion, deletion, substitution) //Minimum cost
            end if
        end for
    end for
    return dp[m][n]
```

Time Complexity:

- The time complexity is directly proportional to the product of the lengths of the input sequences.
- Filling the dynamic programming table iteratively: $O(m*n)$, where m and n are the lengths of the sequences.

$O(m*n)$

(4) procedure MaxDiscount(sequence)

```
n = sequence.length
maxDiscount = array of size n, initialized to 0
maxDiscount[1] = discount(sequence[1]) // Discount of the first store

for i from 2 to n:
    maxDiscount[i] = max(maxDiscount[i-1], 0) + discount(sequence[i])
end for
return max(maxDiscount)
```

'sequence' is the array representing the sequence of stores visited

'discount(store)' gives the discount for a particular store

'maxDiscount[i]' stores the minimum discount achievable up to the ith store in the sequence

- The algorithm iterates through the sequence once, updating maxDiscount at each step by considering the maximum discount up to the current store.

Time complexity:

- The single loop contributes to a time complexity of $O(n)$, where n is the length of the sequence.

(5) . The greedy algorithm selects antennas to maximize coverage without causing interference.

- It sorts antennas based on their coverage endpoints and iteratively selects the antenna with the rightmost endpoint that does not intersect with previously selected antennas.

```
procedure max_antennas(antennas):
    antennas.sort(key=lambda antenna: antenna.end_point) // Sort antennas based on coverage points
    selected_antennas = []
    current_coverage = -∞

    for antenna in antennas:
        if antenna.start_point > current_coverage:
            selected_antennas.append(antenna)
            current_coverage = antenna.end_point
        end if
    end for
    return selected_antennas.length
end
```

Time Complexity:

- Sorting antennas based on coverage endpoints: $O(n \log n)$
- Iterating through sorted antennas: $O(n)$

Resulting in a time complexity of $O(n \log n)$