CSE321 - HW5 1801042614
Elifnur Kabalcı

1) Pseudocode:

1) Sort the points according to their x-coordinates.

2) Divide the set of points into two equal-sized subsets by a vertical line at the median x-coordinate.

3) Conquer by recursively finding the smallest distances in the two subsets. This will give you the left-side and right-side minimum distances, denoted as dLmin an dRmin, respectively.

4) Find the minimal distance dLRmin between the pair of points in which one point lies on the left of the dividing vertical and the second point lies to the right.

5) The final answer is the minimum amoung dLmin, dRmin and dLRmin.

```
def closest_pair(P):
    n = len(P)
    if n<=3:
        return brute_force(P)
    mid = n //2
    Q = P[:mid]
    R = P[mid:]
    (p1, q1, d1) = closest_pair(Q)
    (p2, q2, d2) = closest_pair(R)
    (p3, q3, d3) = closest_split_pair(P, d1 if d1<d2 else d2)
    if d1 <= d2 and d1 <= d3:
        return (p1, q1, d1)
    elif d2 <= d1 and d2 <= d3:
        return (p2, q2, d2)
    else:
        return (p3, q3, d3)
```

=> In terms of time complexity, the sorting operation at the beginning of the algorithm takes $O(n \log n)$ time. After this, the algorithm essentially becomes a recursive divide-and-conquer approach, which has a time complexity of $O(n \log n)$. Therefore the overall time complexity of the algorithm is $O(n \log n)$.

2) QuickHull Pseudocode:

1) Find the leftmost and rightmost points, and make a line segment from these two points.
2) Divide the remaining points into two subsets, where one subset is above the line segment and the other subset is below the line segment.
3) For each subset, find the point that is farthest from the line segment. This point, together with the two endpoints of the line segment, forms a triangle. The points inside this triangle are not part of the Convex Hull, and can therefore be discarded.
4) Repeat the process for the two line segments formed by the triangle.

```
def quickHull(P):
  if len(P) <= 3:
     return P
  convexHull = []
  A = min(P, key = lambda point: point[0])
  B = max(P, key = lambda point: point[0])
  convexHull.extend([A, B])
  S1 = [point for point in P if point not in [A,B] and isLeft(A, B, point)]
  S2 = [point for point in P if point not in [A,B] and not isLeft(A, B, point)]
  hullPoints(S1, A, B, convexHull)
  hullPoints(S2, A, B, convexHull)
  return convexHull
```

=) The QuickHull algorithm has an average-case time complexity is $O(n\log n)$. However, in the worst case (when all the points are part of the Convex Hull), the time complexity can go up to $O(n^2)$.

3) Wagner - Fisher Algorithm - Pseudocode :

1) Initilize a matrix of size (m+1) x (n+1), where m and n are the lengths of the two sequences. The first row and the first column are initialized to 0, ... n.

2) Iterate over the matrix, filling in each cell based on the costs of deletion, insertion, and substitution.

3) The minimum cost of aligning the two sequences is the value in the bottom-right cell of the matrix.

```
def wagner-fisher(seq1, seq2):
    m,n = len(seq1) , len(seq2)
    D=[[0 for_in range(n+1)] for_in range(m+1)]
    for i in range(m+1):
        D[i][0] = i
    for J in range(n+1):
        D[0][J]=J
    for J in range(1,n+1):
        for i in range(1, m+1):
            if seq1[i-1] == seq2[J-1]:
                subCost = 0
            else:
                subCost = 3
            D[i][J]= min(D[i-1][J] + 1, D[i][J-1]+1, D[i-1][J-1] +subCost)
    return D[m][n]
```

=> The wagner-fisher algorithm has a time complexity of O(mn). This is because the algorithm needs to fill in a matrix of size mxn, and each cell requires constant time to compute. The space complexity of the algorithm is also O(mn), due to the need to store the matrix. However, if only the alignment cost is needed (and not the alignment itself), the space complexity can be reduced to O(min(m,n)) by keeping only the current and previous rows (or columns) of the matrix.

4) Pseudocode:

1) Initialize a table of size n, where n is the # of stores. Each entry in the table represents the maximum discount achievable for the subset of stores up to that index.

2) Iterate over the table, and for each entry, calculate the maximum discount by considering all possible combinations of stores up to the index. This can be done by iterating over all the previous entries in the table and calculating the discount for each combination using the calc_discount function.

3) The maximum achievable discount is the maximum value in the table.

```
def max_discount (stores):
    n=len(stores)
    table =[0 for _ in range(n)]
    for i in range (n):
        max_discount=0
        for j in range(i+1):
            discount = calc_discount (stores [j:i+1])
            if discount > max_discount :
                max_discount =discount
        table[i] =max_discount
    return max(table)
```

⇒ The algorithm iterates over the table of size n, and for each entry, it iterates over all the previous entries. Therefore, the time complexity of the algorithm is $O(n^2)$. The space complexity of the algorithm is, due to the need to store the table.

5) Pseudocode:

1) Sort the antennas by their finish points.

2) Initialize a variable current_antenna to the first antenna in the sorted list.

3) Iterate over the sorted list, and for each antenna if its start point is greater than or equal to the finish point of current_antenna, select it and update current_antenna to this antenna.

```
def max_antennas (antennas):
    antennas.sort (key=lambda x : x.finish)
    current_antenna = antennas[0]
    count = 1
    for antenna in antennas:
        if antenna.start >= current_antenna.finish:
            current_antenna = antenna
            count += 1
    return count
```

=) The algorithm first sorts the antennas, which takes $O(n\log n)$ time, where is the # of antennas. Then it iterates over the sorted list of antennas, which takes $O(n)$ time. Therefore, the overall time complexity of the algorithm is. The space complexity of the algorithm is $O(n)$, due to the need to store the sorted list of antennas.