Assignment 4

Problem 1

Consider the problem of sorting an array A[1, ..., n] of integers. We presented an O(n log n)-time algorithm in class and, also, proved a lower bound of Ω(n log n) for any comparison-based algorithm.

1. a. Give an efficient sorting algorithm for an array C[1,...,n] whose elements are taken from the set {1,2,3,4,5,6,7}.

Counting sort is the most efficient algorithm for sorting since its complexity is O(n+k) or it sorts the array in linear time. counting sort works by counting the number of objects having distinct key values. We use counting sort because the range of values was determined in advance and is not too broad. we can also use bucket sort or radix sort to get in O(n).The algorithm of counting sort works as follows

1. First, from the specified array, create a count array of the size of the maximum element +1 and initialize it with zero.
2. We will count the number of occurrences of each element in the array and save the results.
3. Modify the count array so that the total of previous counts is stored in each element at each index.
4. Each object's position in the output sequence is indicated by the modified count array.
5. In the count array, find the index of each element of the original array. This returns the total number of items.
6. Rotate the array once clockwise.
7. Output each object from the input sequence followed by increasing its count by 1.
8. Put data 1 at index 0 in output. Increase count by 1 to place next data 1 at an index 1 greater than this index
9. After each element has been correctly positioned, reduce its count by one

10. After all the operations we can sort the array in O(n) time complexity and so it is the efficient algorithm for this particular case.

psuedo Code:

```
CountingSort(array, max_value):
// Step 1: Create a counting array of size max_value+1 and initialize with zeros
counting_array = [0] * (max_value + 1).               -----------O(n)

// Step 2: Count the frequencies of each element in the input array
for element in array:
    counting_array[element] += 1.                      -----------O(n)

// Step 3: Modify the counting array to represent the cumulative count
for i in range(1,max_value):
    counting_array[i] += counting_array[i-1]           -----------O(n)

// Step 4: Create a new output array
output_array = [0] * len(array)                       -----------O(n)

// Step 5: Place the elements in the output array based on their counts
for element in reversed(array):                        -----------O(n)
    index = counting_array[element] - 1
    output_array[index] = element
    counting_array[element] -= 1

return output_array
```

code:

```
def countingSort(arr):
# Find the maximum element in the array
max_value = max(arr)

# Create a count array to store the count of each element
count = [0] * (max_value + 1)

# Count the occurrences of each element
for num in arr:
    count[num] += 1

# Calculate the cumulative sum of count array
for i in range(1, max_value + 1):
    count[i] += count[i - 1]

# Create an output array to store the sorted elements
output = [0] * len(arr)

# Place the elements in the output array based on their count
```

```
for num in arr:
    output[count[num] - 1] = num
    count[num] -= 1

return output

arr = [1,2,3,4]
ar1=[1,3,4,5,6,5,5,5,6]
sorted_arr1 = countingSort(ar1)
sorted_arr = countingSort(arr)
print(sorted_arr)
print(sorted_arr1)
```

```
[1, 2, 3, 4]
[1, 3, 4, 5, 5, 5, 5, 6, 6]
>
```

1. b. Give an efficient sorting algorithm for an array D[1, ..., n] whose elements are distinct (D[i] D[j], for every ij {1, ..., n}) and are taken from the set for every i≠j {1, 2, ..., 2n}

   We could use the Counting sort which is used in the first subpart a As it is the most efficient algorithm for sorting since its complexity is O(n+k) or it sorts the array in linear time. counting sort works by counting the number of objects having distinct key values. We use counting sort because the range of values was determined in advance and is not too broad. we can also use bucket sort or radix sort to get in O(n). The algorithm of counting sort works as follows.

   1. First, from the specified array, create a count array of the size of the maximum element+ and initialize it with zero.
   2. As every element is unique and distinct there is no use of counting in this case. collect the occurrence of each element in the array and save the results.
   3. Modify the count array so that the total of previous counts is stored in each element at each index.

4. Each object's position in the output sequence is indicated by the modified count array.
5. In the count array, find the index of each element of the original array. This returns the total number of items.
6. Rotate the array once clockwise.
7. Output each object from the input sequence followed by increasing its count by 1.
8. Put data 1 at index 0 in output. Increase count by 1 to place next data 1 at an index 1 greater than this index
9. After each element has been correctly positioned, reduce its count by one
10. So hence after all the operations we can sort the array in O(n) time complexity and so it is the efficient algorithm for this particular case.

```
CountingSort(array, max_value):
// Step 1: Create a counting array of size max_value+1 and initialize with zeros
counting_array = [0] * (max_value + 1)             ----------O(n)

// Step 2: Count the frequencies of each element in the input array
for element in array:
    counting_array[element] += 1                     ----------O(n)

// Step 3: Modify the counting array to represent the cumulative count
for i from 1 to max_value:                            ----------O(n)
    counting_array[i] += counting_array[i-1]

// Step 4: Create a new output array
output_array = [0] * len(array)                     ----------O(n)

// Step 5: Place the elements in the output array based on their counts
for element in reversed(array):                    ----------O(n)
    index = counting_array[element] - 1
    output_array[index] = element
    counting_array[element] -= 1

return output_array
```

code:

```
def countingSort(arr):
# Find the maximum element in the array
max_value = max(arr)

# Create a count array to store the count of each element
count = [0] * (max_value + 1)

# Count the occurrences of each element
for num in arr:
    count[num] += 1

# Calculate the cumulative sum of count array
for i in range(1, max_value + 1):
    count[i] += count[i - 1]

# Create an output array to store the sorted elements
output = [0] * len(arr)

# Place the elements in the output array based on their count
for num in arr:
    output[count[num] - 1] = num
    count[num] -= 1

return output

arr = [1,2,3,4]
ar1=[1,3,4,5,6,5,5,5,6]
sorted_arr1 = countingSort(ar1)
sorted_arr = countingSort(arr)
print(sorted_arr)
print(sorted_arr1)
```

```
[1, 2, 3, 4]
[1, 3, 4, 5, 5, 5, 5, 6, 6]
>
```

Problem 2

In case you designed linear-time sorting algorithms for any subpart of problem 1, does it mean that the lower bound for sorting of Ω(n log n) is wrong? Explain.

If you did not design a linear-time sorting algorithm for any subpart of problem 1, explain your lower bound for both subparts of problem 1.

Ans.

I have used count sort in the first problem which performs sorting operations in O(n+k) where n is the length of the array and k is the range of input values.it doesn't mean that the lower bound for sorting of Ω(n log n) is wrong. counting Sort is a non-comparison based sorting algorithm.This algorithm uses the frequencies of elements in the input array to sort them and doesn't involve in comparing any elements. In the problem 1 and 2 the k and n has been mentioned. The counting sort depends on both k and n as if k=O(n^2) then the count sort will not be efficient as the algorithm takes O(n^2) instead of O(n) only if the condition k<<n is followed then the counting sort holds good and it will perform sorting in Θ(n).

Comparison-based sorting algorithms like Quicksort, Mergesort, and Heapsort have an average time complexity of O(n log n). These algorithms rely on comparing elements to establish their relative order, and the number of comparisons required grows logarithmically with the number of elements, resulting in the O(n log n) time complexity.

Counting Sort is a valuable sorting algorithm in situations where the range of input values is small, leading to a more efficient sorting process compared to comparison-based algorithms. it is important to consider the specific requirements and characteristics of the data set when selecting an appropriate sorting algorithm. However, for most practical purposes, the lower bound of Ω(nlogn) is still considered to be the best time complexity that can be achieved by a comparison-based sorting algorithm

Problem 3 Closest Pair (10 points)

We have learned the algorithm that solves the Closest pair problem in 2D in Θ(n log n) time. (Closest pair problem in 2D: Given n points in the 2D plane, find a pair with smallest Euclidean distance between them.)

Give an algorithm that solves the Closest pair problem in 3D in Θ(n log n) time. (Closest pair problem in 3D: Given n points in the 3D space, find a pair with smallest Euclidean distance between them.)

To solve the Closest Pair problem in 3D in Θ(n log n) time, we can use a modified version of the Divide and Conquer algorithm. Here's the algorithm:

1. Sort the points based on their x-coordinates. This step takes Θ(n log n) time.
2. Divide the sorted array into two equal-sized subarrays based on the x-coordinate of the middle point. This step takes Θ(n) time.
3. Recursively find the closest pair in each subarray using the same algorithm. This step involves two recursive calls and takes Θ(n log n) time.
4. Determine the minimum distance d between the closest pair found in the left subarray and the closest pair found in the right subarray. This can be done in Θ(n) time.
5. Create a strip of points within distance d of the middle line in the x-axis. Sort the strip based on their y-coordinates. This step takes Θ(n log n) time.
6. Iterate through the points in the sorted strip, considering only the points within 6d distance vertically. For each point, calculate the distance to its closest neighbors in the strip and update the minimum distance if necessary. This step takes Θ(n) time.
7. Return the minimum distance found.

The overall time complexity of this algorithm is Θ(n log n), as the most time-consuming steps involve sorting the points, which takes Θ(n log n) time. The algorithm takes advantage of the divide and conquer strategy to efficiently find the closest pair of points in 3D space.

```
import math

def closestPair3D(points):
    # Step 1: Sort points based on x-coordinate
    sortedPoints = sorted(points, key=lambda point: point[0])

    # Step 2: Recursive divide-and-conquer
    return closestPair3DRecursive(sortedPoints)

def closestPair3DRecursive(points):
    n = len(points)

    # Base case: If number of points is small, brute force search
    if n <= 3:
        return bruteForceClosestPair3D(points)
```

```
    # Divide points into two halves
    mid = n // 2
    leftPoints = points[:mid]
    rightPoints = points[mid:]

    # Recursively find closest pair in each half
    closestLeft = closestPair3DRecursive(leftPoints)
    closestRight = closestPair3DRecursive(rightPoints)

    # Determine the minimum distance from both halves
    minDistance = min(closestLeft['distance'], closestRight['distance'])

    # Step 3: Merge step - Check for closest pair across the halves
    closestAcross = closestPairAcrossHalves(points, minDistance)

    # Find the overall closest pair
    return min(closestLeft, closestRight, closestAcross, key=lambda pair: pair['dista
nce'])

def closestPairAcrossHalves(points, minDistance):
    # Create a strip of points within minDistance from the middle line
    midX = points[len(points) // 2][0]
    strip = [point for point in points if abs(point[0] - midX) < minDistance]

    # Sort the points in the strip based on y-coordinate
    sortedStrip = sorted(strip, key=lambda point: point[1])

    # Initialize closest pair with a large distance
    closestPair = {'distance': float('inf'), 'point1': None, 'point2': None}

    # Iterate through the points in the strip
    for i in range(len(sortedStrip)):
        # Check distances only within a vertical range of 6 * minDistance
        for j in range(i + 1, min(i + 7, len(sortedStrip))):
            # Calculate distance between points
            distance = euclideanDistance(sortedStrip[i], sortedStrip[j])

            # If distance is smaller than minDistance, update closestPair
            if distance < closestPair['distance']:
                closestPair['distance'] = distance
                closestPair['point1'] = sortedStrip[i]
                closestPair['point2'] = sortedStrip[j]

    return closestPair

def bruteForceClosestPair3D(points):
    # Initialize closest pair with a large distance
    closestPair = {'distance': float('inf'), 'point1': None, 'point2': None}

    # Check distances between all point pairs
    for i in range(len(points) - 1):
        for j in range(i + 1, len(points)):
            # Calculate distance between points
```

```
            distance = euclideanDistance(points[i], points[j])

            # If distance is smaller than current closestPair, update closestPair
            if distance < closestPair['distance']:
                closestPair['distance'] = distance
                closestPair['point1'] = points[i]
                closestPair['point2'] = points[j]

    return closestPair

def euclideanDistance(point1, point2):
    return math.sqrt((point2[0] - point1[0]) ** 2 + (point2[1] - point1[1]) ** 2 + (p
oint2[2] - point1[2]) ** 2)

    # Test case 1
points1 = [(2, 3, 53), (34, 5, 46), (27, 8, 79)]
result1 = closestPair3D(points1)
print(result1)


# Test case 2
points2 = [(1, 12, 1), (2, 24, 2), (34, 3, 34), (4, 42, 4), (524, 524, 5)]
result2 = closestPair3D(points2)
print(result2)


# Test case 3
points3 = [(1, 10, 10), (20, 2, 20), (30, 30, 3), (40, 40, 40)]
result3 = closestPair3D(points3)
print(result3)
```

```
{'distance': 32.817678162843876, 'point1': (2, 3, 53), 'point2': (34, 5, 46)}
{'distance': 12.083045973594572, 'point1': (1, 12, 1), 'point2': (2, 24, 2)}
{'distance': 22.9128784747792, 'point1': (1, 10, 10), 'point2': (20, 2, 20)}
>
```