Saleh Alkhalifa
10/5/2022

Homework#4

Message to the TA: I was not sure of the level of depth you wanted me to achieve in this assignment as the instructions were not specific, so I wrote out the pseudocode at a similar level of depth shown in the lecture, per the recommendation I received by other TAs. I wanted to make sure that it was specific enough to answer the question.

Problem 1 Sorting Special Arrays (10 points)

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. **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}.**

i. We will first create a new array we shall call count, and instantiate it with values of 0, and a size of 7, given that the input are elements are taken from the set {1,2,3,4,5,6,7}, and index begins at 0.
ii. Next, we iterate over the array and increment the value in the arrays index, so that we essentially count the elements using the index.
iii. Next, we iterate over the array in order to store the cumulative count by adding current and previous counts to the array.
iv. At this point, the cumulative count from step 3 is the element's position.
v. We will once again iterate over this array and reduce the count by 1 as we go over it and ultimately add to a new, and now sorted, array.
vi. Finally, we can replace the input array with the values of the output array.

Ultimately, this will yield a sorting algorithm with time complexity of O(n)

[19]:
```
def count_sort_problem1(in_arr):
    print("Before: ", in_arr)
    # Define the example array to sort
    length = len(in_arr)

    # Create new list to store counts and output:
    count = [0] * 8
    output = [0] * length

    # Count each elemet and increment value in array index:
    for i in range(0, length):
        count[in_arr[i]] +=1

    # Store cumulative count
    for i in range(1, 8):
        count[i] += count[i - 1]

    # Iterate over array, reduce by 1, populate output array
    i = length - 1
    while i >= 0:
        output[count[in_arr[i]] - 1] = in_arr[i]
        count[in_arr[i]] -= 1
        i -= 1
    print("After : ", output)
    return output
```

[21]:
```
x = count_sort_problem1([1, 4, 4, 4, 6, 2, 1, 5, 6, 3, 3, 1, 1, 1, 7])
```
```
Before:  [1, 4, 4, 4, 6, 2, 1, 5, 6, 3, 3, 1, 1, 1, 7]
After :  [1, 1, 1, 1, 1, 2, 3, 3, 4, 4, 4, 5, 6, 6, 7]
```

[24]:
```
x = count_sort_problem1([7,6,5,4,3,2,1])
```
```
Before:  [7, 6, 5, 4, 3, 2, 1]
After :  [1, 2, 3, 4, 5, 6, 7]
```

[25]:
```
x = count_sort_problem1([7,1,1,1,1,1,1,1])
```
```
Before:  [7, 1, 1, 1, 1, 1, 1, 1]
After :  [1, 1, 1, 1, 1, 1, 1, 7]
```

** Please find python code attached.

Homework#4

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

   i. First, we will instantiate a new array we will call count, containing values of 0 and a size of 2n.

   ii. Next, we will count each element in our input array, and increment the count of that element in the count array using the index values. This will essentially allow us to count the elements in the original array.

   iii. Next, we iterate over the array in order to store the cumulative count by adding current and previous counts to the array. Since they are not repetitive, will expect to encounter each value only once.

   iv. At this point, the cumulative count from step 3 is the element's position.

   v. We will once again iterate over this array and reduce the count by 1 as we go over it and ultimately add to a new, and now sorted, array.

   vi. Finally, we can replace the input array with the values of the output array.

Ultimately, this will yield a sorting algorithm with time complexity of O(n)

Problem 2 (10 points)

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.

The algorithm introduced in class was a comparison-based algorithm, and we established a lower bound of n log n. The algorithm used above, count sort which runs in linear time, is not a comparison-based algorithm and therefore that lower bound is not applicable in this case. Unlike merge sort, count sort operations by counting the elements of a list or array based on occurrence and then orders them by filling in a new array, which happens in linear time O(n).

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.

Not applicable since I used linear time sorting algorithm

Homework#4

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.)

For 2D:

i. The first step here is to divide our space S into two spaces S_1 and S_2 using the vertical line L, which is generally defined via the median of the coordinates. This will allow us to find the middle point which is sometimes P[n/2].
ii. We can then recursively compute the closest pair in each of the spaces, and get the closest pair distances
iii. Next, we compute the closest pair in which one point is in each of S_1 and S_2, giving us D_{Left} and D_{Right}
iv. Since complications arise in situations where all points are within the boundary ε which would naively require $n^2/4$ calculations. If we consider the box R, we can determine that the maximum number of points of distance d inside it would be 6.
v. We then determine the six potential points, via the sorted lists of points, which is done in linear time.
vi. Finally, we can determine the final closest pair by calculating the minimum distance between the three options, giving us a recurrence of:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n), which\ is\ solved\ by\ T(n) = O(n\ logn)$$

For 3D:

i. Similarly to the structure we saw in 2D, we can build on this concept of divide and conquer to account for 3D situations as well. The first step is to divide the space S into two spaces S_1 and S_2 using the median to create a hyperplane, H.
ii. We can then recursively compute the closest pair and distances for S_1 and S_2 separately, and determine the minimum of the two.
iii. Given the 3D nature, we introduce a new dimension, and so we shall let S' be the set of all points with the distance d of a hyperplane H.
iv. We can now introduce a sparsity condition such that our cube of interest representing this space with a side of 2L will contain O(1) points within our space S.
    a. Given the recursive nature, its important to note that this sparsity condition concerns points within a slab of thickness s relative to the hyperplane H.
    b. We will need access to sorted lists of the points to achieve O(n)
v. Using the predefined sparsity condition, we can now recursively observe the pairs of points found in the previously defined S', containing O(n) pairs within it.
vi. Finally, we return the minimum, based on distance, to identify the closest pairs, giving us a recurrence of:

$$T(n,d) = 2T\left(\frac{n}{2}, d\right) + U(n, d-1) + O(n)$$
$$T(n,d) = 2T\left(\frac{n}{2}, d\right) + O(n(logn)^{d-2} + O(n)$$
$$T(n,d) = O(n(logn)^{d-2}) = O(n(logn)^{d-1})$$

vii. Given that the problem size m is:

$$m \leq \frac{n}{(logn)^{d-2}}, then\ we\ see\ that\ U(m, d-1) = O(m(logm)^{d-2}) = O(n)$$

viii. Finally, this gives a recurrence of:

$$T(n,d) = 2T\left(\frac{n}{2}, d\right) + O(n) + O(n)$$
$$T(n) = O(n \log n)$$