Homework 3- COMS 311

aryanrao

Part- A

1. (a) T (n) = T (n/4) + cn, T (1) = 1

   T (n/4) = T ($n/4^2$) + c(n/4)

   T (n) = T ($n/4^2$) + c(n/4) + c(n)
   T (n) = T ($n/4^2$) + c($n/4^2$) + c(n/4) + c(n)
   If we keep on doing this recursive step until we get this
   T (n) = T (n/4^k) + c (n/4^(k-1)) +………...+ c($n/4^2$) + c(n/4) + c(n)

   Now as we know T (n/4^k) = 1 we can get k= log_4 (n)
   We can further solve the underlined equation it is a geometric progression sequence.
   c (n/4^(k-1)) +………...+ c(n/4^2) + c(n/4) + c(n)= cn(1+ ¼ +$(1/4)^2$+$(1/4)^3$+ ……. (¼)^k-1)
   That sums up to = 4cn/3(1-(1/4)^k)

   T (n) = T (n/4^k) + 4cn/3(1-(1/4)^k) ….. substitute k = log_4(n).

   T (n) = T (1) + 4cn/3(1-1/n) as we know T (1) = 1
   T(n) = 1 + 4cn/3-4c/3

   (b) T (n) = 3T (n/2) + cn^2, T (2) = 1.

   T (n) = 3(3T(n/4) +c(n/2) ^2) +cn^2=3^2T(n/2^2) +3c(n/2) ^2) +cn^2
   T (n) = 3^3T(n/2^3) +3^2c(n/2^2) ^2) +3c(n/2) ^2) +cn^2
   (If you observe the above sequence is in G.P with difference ¾.)
   So, if you use the formula a(1-r^n)/1-r ------ 1

   Also, T(n/2^k) == 2 so k+1 == log_2 (n)
   T (n) = 3^kT(n/2^k)+ 3^(k-1)T(n/2^(k-1))+……. +3c(n/2) ^2) +cn^2

   Cn^2 (1 + ¾ +(¾)^2 +……….+ (¾)^k-1) + 3^kT(n/2^k))
   4Cn^2(1- (¾)^k) ---- now substitute k = log_2(n)-1
   4Cn^2(1- (¾)^(log_2(n)-1)
   4Cn^2(1- (¾)^(log_2(n))*4/3)
   4Cn^2 – 4c/3*(3^log_2(n))
   Well, the above equation can be written as 4Cn^2 – 4c/3*(n^log_2(3))
   T (n) = 4Cn^2 – 4c/3*(n^log_2(3)).

2. function findAcceptablePoints(points):

```
if size(points) <= 1:
    return points

sort(points) based on x-coordinate

mid = size(points) / 2
S1 = points[0:mid]
S2 = points[mid:end]

acceptablePoints = merge(findAcceptablePoints(S1), findAcceptablePoints(S2))

return acceptablePoints
```

```
function merge(S1, S2):
    i = 0, j = 0
    acceptablePoints = []

    while i < size(S1) and j < size(S2):
        if S1[i].x < S2[j].x and S1[i].y < S2[j].y:
            add S1[i] to acceptablePoints
            increment i
        else:
            increment j

    append remaining points in S1 starting from i to acceptablePoints

    return acceptablePoints
```

This pseudo code uses divide and conquer algorithm. It first sorts the points based on their x-coordinates and then recursively divides the points into two halves. It then merges the acceptable points found in the left and right halves, considering the condition $x < p$ and $y < q$.

Problem: Given a set of two-dimensional points S, find all acceptable points such that there exists a point ⟨p, q⟩ in S where $x < p$ and $y < q$.
Solution:

1. Implement the **findAcceptablePoints** function:
    - Base case: If the size of points is less than or equal to 1, return points.
    - Sort the points based on their x-coordinates.
    - Split the points into two halves, S1 and S2.
    - Recursively find acceptable points in S1 and S2 using **findAcceptablePoints**.
    - Merge the acceptable points from S1 and S2 using **merge**.
    - Return the merged acceptable points.
2. Implement the **merge** function:
    - Initialize pointers i and j to 0.
    - Initialize an empty list **acceptablePoints** to store the merged acceptable points.

  - While i is less than the size of S1 and j is less than the size of S2, do the following:
    - If S1[i].x < S2[j].x and S1[i].y < S2[j].y, add S1[i] to **acceptablePoints** and increment i.
    - Otherwise, increment j.
  - Append the remaining points in S1 starting from index i to **acceptablePoints**.
  - Return **acceptablePoints**.

3. Call **findAcceptablePoints** with the input set of points S.

   Let T(n) be the time complexity of the algorithm when there are n points in the set. The algorithm follows a divide and conquer approach. In each recursive call, the set is divided into two halves. The merge operation takes linear time to merge the acceptable points from the left and right halves. Therefore, the recurrence relation can be stated as:
   $T(n) = 2T(n/2) + O(n)$
   This recurrence relation represents that the algorithm recursively solves two subproblems of size n/2 and performs a merge operation in linear time O(n) to combine the results.

   The time complexity of the algorithm is $O(n \log n)$ since the sorting step takes $O(n \log n)$ time and the recursive calls divide the set into two halves.