CS430 Homework 4

Chapter 5

Problem-1:

Give an algorithm that finds the median value using at most O (log n) queries.

Answer:

We have 2 database DB1 and DB2.

Each database contains n Numerical values so there are total 2n values.

We need to find nth smallest value as the median of these 2n values.

First we query each database for its median- so we query for n/2 element.

Median of database DB1 is m1 and for database DB2 is m2.

We show the median of the joint database must be in between m1 and m2

To see this, there are at least n records in DB1 and DB2 < = max (m1, m2)

Means, median of database is not smaller than Min (m1, m2) and not greater than Max (m1, m2).

Algorithm:

1. FindJointMedian (A, a_{low}, a_{high}, B, b_{low}, b_{high}):
2. if a_{low} == ahigh

    m1= query (DB1, 1)

    m2 = query (DB2, 1)

    return min (m1, m2)

3. m1 = query (DB1, (a_{high} + a_{low} – 1)/2)
4. m2 = query (DB2, (b_{high} + b_{low} – 1)/2)
5. if m1 < m2

    return FindJointMedian (DB1, m1 + 1, a_{high}, DB2, b_{low}, m2)

6. else

    return FindJointMedian (DB1, a_{low}, m1, DB2, m2 + 1, b_{high})

Let T (n) be the total number of queries.

As each round we reduce the problem size by half using two queries, we have T (n) = T (n/2) + 2.

So,T (n) = O (log n).

Problem-2

A = {a1 , a2, a3, ...an-1, an}

Function Count_Inversion (A [1...n])

If n==1

No of inversion N=0 // list has only one element

Else

//divide the list into two part

A contains the first [n/2] elements

B contains the reaming [n/2] elements

Count_Inversion(A)

//return No of counting inversion N1 and sorted Array A1[b1 ..bk]

Count_Inversion(B)

//return No of counting inversion N2 and sorted Array B1[bk+1...bn]

Initialize i=1, j=1, N3=0

// to count the number of significant split inversions

while (i <= length(A1) and j <= length(B1))

if(A1[i] > 2*B1[j])

N3 =N3+ length(A1) – i + 1

j = j + 1;

else

i = i + 1;

//the normal merge-sort process

i = 1, j=1

//the sorted A to be output

Output Array D=[1....n] //initialize with zero

for k = 1 to n

if A1[i] < B1[j]

D[k] = A1[i];

i = i + 1;

else

D[k] = B1[j];

j = j + 1;

return output Array D and Total No of Inversion N= (N1+N2+N3)

Runtime Analysis:

At each level, counting of significant split inversions and the normal merge-sort process take O(n) time

we break the problem into two subproblems and the size of each subproblem is n/2.

Hence, the recurrence relation is T(n) = 2T(n/2) + O(n).

So, Time complexity is O(n log n).

Problem-3

There are n cards.

Let c1,c2,c3….. cn are the equivalence classes of the cards.

Cards i and j are equivalent if ci = cj

We need to search for value x so that more than n/2 of the indices have ci=x

We have total n cards so divide the cards in two /2 parts and apply the algorithm recursively to each part.

So, if there are more than n/2 cards have equivalent class x in whole set, that means one of the 2 parts will have more than half the cards equivalent to x.

Suppose more than half cards are equivalent to x, that means one of the 2 parts will also have at least half of its the cards being equivalent to x.

So, At least one of the 2 recursive call return a card that has equivalent class x

However, reverse is not true : just because more than half the element on one side equivalent to x does not mean more than half of the total elements among both side equivalent to x.

That's why If we get a majority card return from either side ,we have to verify it

Algorithm:

Majority(S)

If |s|=1

    Return the card in S

Let S1 be the first n/2 cards and S2 be the remaining n/2 cards

Call function Majority(S1)

    If function return a card **a** test this against all other cards

    If **a** is strict majority, return **a**

Call function Majority(S2)

    If function return a card **b** test this against all other cards

    If **b** is strict majority, return **b**

Return no strict Majority found

Running Time:

T(n) : Running time with n cards

T(n)=2T(n/2) + 2n

So T(n) is O (n log n)

Problem-4

Algorithm:

Algorithm that takes n lines as input

L= { l1,l2,……ln}

Base Case: if n<=3 we can find visible line in constant time.

Divide the number of lines into 2 parts Lines from L1, L2…..Lm and lines from Lm+1, Lm+2…. Ln

Recursively Compute sequence of visible line L= {li_1, li_2,…. lip} and set of intersection point a1, a2,……. ap-1 (from line L1, L2…Lm) in order of increasing slope

Recursively compute sequence of visible line L'= {Lj_1,Lj_2 ………………….Lj_q} and intersection point b1,b2….bq-1 (from Lm+1,Lm+2……..Ln)

Merge the set of intersection points into a single list of points as per the increasing X-coordinate values -say c1, c2, c3…..cp+q-2

For each k,

We consider the line that is uppermost in both L and L' at X-coordinate ck

The uppermost line Li_{s1} in L lies above the uppermost line Lj_{s2} in L' at x-coordinate Cz

//(z smallest index)

let (x1, y1) is the intersection point of line Li_{s1} and Lj_{s2}

Implied point x1 lies between the x-coordinate of Cz-1 and cz

That means Li_{s1} is uppermost in L +L' immediately to the left of x1 and Lj_{s2} is uppermost in L +L' immediately to the right of x1

Thus,

The sequence of visible lines is Li1, Li2……..Lis1, Ljs2,…………. Ljq and

Sequence of intersection points is ai1, ai2,……ai_{s1-1}, (x1, y1), bj_{s2},……. bjq-1

Running Time:

Sorting n lines in increasing order of slope take O(nlogn) time

Finding visible line with intersection point take and merging sorted list of intersection point takes O(n)

So this Algorithm can be implemented in O(nlogn) time.