CS 5084
Assignment 5

1. To find the median of two sets of numbers without directly seeing all numbers, we can use a recursive algorithm that ask for k-th smallest number in each set & eliminates halves of the sets that cannot contain the median.

a] First we find the median of each set by asking for some i^{th} smallest no. where 'i' is half the size of the set.

b] If one set's median is < other set's median we eliminate the smaller set's first half because it won't have median & then we eliminate larger set's second haf because it cannot have a smaller median.

c] If we repeate this process with the remaining halves of set until we find median.

- Thus, we can find the median with at most O(logn) queries, which is comparatively small no. for big data sets no.s.

So algorithm can be given as:

d_1 - database 1

d_2 - database 2

n - no. of numerical values .... (given)

$d_1 i$ - ith smallest element of d_1

$d_2 i$ - i^{th} smallest element of d_2

let $hf = \frac{n}{2}$ of numerical values

$\therefore d_1 hf$ = median d_1
& $d_2 hf$ = median d_2

for our case lets assume $d_1 hf < d_2 hf$
then we apply steps b] & c] mentioned earlier.
so we can write function 'f' as

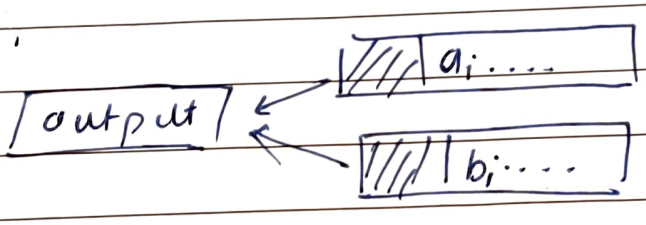
```
f(i1, i2, n)
{
  if d1(i1+hf) < d2(i2+hf)
     then return (d1hf + hf, d2, hf)
      else return (d1ff, d2hf thf, hf)

  if no:of numerical values = 1
     then return min [d1(i1+hf), d2(i2+hf)]
}
```

where, i_1 & i_2 are integer values
n - numerical values in db ... given
to find median of 2 joint data bases
$[d_1(i_1+1, i_1+n)$;
$d_2(i_2+1, i_2+n)]$

2. given,
sequence of 'n' no.s $a_1 \ldots a_n$ all distinct.
an inversion to be a pair $i<j$ such that $a_i > a_j$

let S_n be significant no. of inversions.
& $a_1' \ldots a_n'$ be same sequence sorted in increasing order.

Using same mergecount algorithm (page 223, 224, 225 textbook)



(merging 2 sorted list)

for asking case $a_i > 2a_j$ we can merge twice:
1) merging $b_1 \ldots b_{n/2}$ with $b_{\frac{n}{2}+1} \ldots b_n$
2) merging $b_1 \ldots b_{n/2}$ with $2b_{\frac{n}{2}+1} \ldots 2b_n$

in step 1 we can sort & step 2 we can count significant inversion.

for $n=1$ we will get $S_n = 0$ & a_1 sequence.
if n is greater than 1 then

for $(a_1' \ldots a_{n/2}')$ we will get S_{n_1} & $b_1 \ldots b_{n/2}$ seq.

for seq. $(a'_{n/2+1} \dots a'_n)$ we get $(s_{n2} \in b_{n/2+1} \dots b_n)$

sb total significant inversions s_{n3} $(a_i \dots a_j)$ where $i \le n/2 \le j$

we can calculate as $S_n = S_{n1} + S_{n2} + S_{n3}$ for sequence $(a'_1 \dots a'_n)$ merging with $(b_1 \dots b_{n/2}; b_{n/2+1} \dots b_n)$

we can write algo. as:

```
#   mergesort(arr, left, right):
{   if left < right:
        mid = (left + right) //2
        inversions = mergesort(arr, left, mid)
        inversions+= mergesort(arr, mid+1, right)
        inversions+= merge(arr, left, mid, right)
        return inversions
    else
        return 0
            }

merge(arr, left, mid, right):
    i = left , j = mid+1
    inversions = 0
while( i <= mid && j <= right)
  {   if arr[j] >= arr[i]
          inversions+= mid - i + 1
          j = j+1
      else  i = i+1     return inversions }
```

3. given, 'n' bank cards

let $a_1 \ldots a_n$ & $b_1 \ldots b_n$ be equivalent classes of cards ($a_1 = b_1$ are eqv. cards).

lets divide set of 'n' cards in 2 equal groups $\frac{n}{2}$

If we have majority element in the original collection class, then it must also be the majority element in either of class. (say 'b')
Because, if a card is not in the majority element in either 'a' or 'b' then it cannot be overall majority element, since it occurs in less than half the cards in each half.
∴ we should consider only the 2 halves of their majority elements.

To implement algo. using divide & conquer approach
- we can compare 2 cards for equivalence by invoking eqv. tester once for each pair of cards.
- $O(n^2)$ time can be consider for worst case invokations.

But, if we sort the cards & then only compute adjecet cards, (∵ equivalent cards will be in the adjecent sorted order)
- Thus,
  sorting will take $O(n \log n)$ time.

then we just have to make n-1 invocations for equivalence tester, gives us total of $O(n \log n)$ invocation. as overall Time complexity.

if 'n' is odd then one card will have no match after forming all pairs.

(I) Here we can: for each pair that is not equivale we can discard both cards. For pairs that are equivalent, we will keep one of two.

(II) If there is an equivalence class with more than $n/2$ cards, than the same equiv. class must also have more than half of the cards after step (I)

(III) So, when we discard both cards in pair then atmost 1 of them can be From majority equivalence class, so in step (I) it takes $(\frac{n}{2})$ tests for 'n' cards; thus we only have $\leq \frac{n}{2}$ cards left:

(IV) When we came down to single card, then its equivalence is the only candidate for majority. This is, this method takes $n/2, n/4 \ldots$ tests to reach to end.

∴ Runtime is $O(n/2) + O(n/4) \ldots$ i.e $O(n)$

6. Given, n node complete binary tree $n = 2^d - 1$

To find a local minimum of T using only $O(\log n)$ probes to nodes of T, then we can use divide and conquer algorithm.

- Starting with the root of tree, we compare labels of the root with its 2 children;
  if label of root < labels of both children
  we get local min. & we return label of root
  else:
  we explore the child recursively with smaller label.
- At each, step of the recursion, we explore the subtree rooted at the child with smaller label and repeat the process.
  ∵ T is complete binary tree, each subtree in recursion is complete binary tree with half no. of nodes
  ∴ we can recurse on smaller subtree further with no. of probes required

- Worst case no. of probes required occurs when we reach leaf node. However, T is complete binary tree with $2d-1$ nodes, max. depth of tree is d and hence the worst case no. of probe required is $O(\log n)$.
  ∴ Overall algorithm has $O(\log n)$ runtime.

2. lets consider the grid as chess board itself
- first we will divide it into four parts.
- We can start looking at the squares on border
- If we find square with lowest no. on the border & check its neighbouring square, if one of those neighbouring square has lower no. we move to that square & repeat the process.
- If it does not have lower one then we have our local minimum.
- If the lowest square is in the middle of the chess board, we still divide chess board into 4 parts & repeat the process on the part containing lowest square until we found a local minimum.
- In few repeatations it is guarenteed to find a local minimum.

Given, Graph 'G' with grid $n \times n$

G has n^2 nodes

Node set is the set of all ordered pairs of natural nos (i,j) where $1 \leq i \leq n$ & $1 \leq j \leq n$

nodes (i,j) & (k,l) are joined by an edge only if $|i-k| + |j-l| = 1$

Let us say G has node V that does not belong to set of nodes on the border of graph G, that is adjacent to node in the same set & smaller than that set. — (1)

So, For the grid G the global minimum does not occur on border of G (from eq^n (1))

We can call this min. as internal local minimum

let node N_1 denotes middle rows & colum combined nodes of G.

let N_2 denotes union of border nodes & N_1 if we discard N_2 our chessboard gets divided into 4 parts.

∴ we left with 4 subgrids.

let A be all nodes adjacent to N_2

Using $O(n)$ probes, we can find some node $w \in N_2 \cup N_1$ minimum value. but we know $w \notin$ border elements

∴ $V \in N_2 \cup N_1$ & $V <$ border elements.

∴

1) If $w \in N_1$, then w is internal local min. Since all neighbours of w are in $N_1 \cup N_2$ & it is smaller than all of them.

ii) OR $w \in A$

Let G_2 be smaller grid or subgrid containing w together with portions of nodes (of eq^n ①)
Now, G_2 satisfies / follows eq^n ① as well.
∴ G_2 also have internal local minimum.
which is also internal local minimum of original grid G.

- If we recursively apply same step on G_2.
Thus, it will take O(n) time.