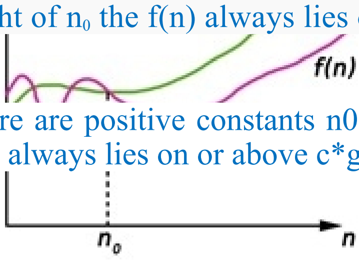
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

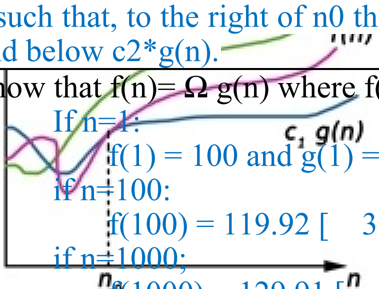
INTERNAL TEST – I

Semester: 4	**Session:** Feb- June 2022
Course Name: DESIGN AND ANALYSIS OF ALGORITHMS	**Course Code:** 18CSI401

Solution

Q#		Questions	Marks	CO's	Bloom's Level
		(15×3 = 45 Marks)			
1.	**a**	Formally Define Big-O, Omega (Ω), and Theta (Θ) notation with respect to two growing functions f(n) and g(n) with a constant factor C. Depict the same graphically and explain. Big - O f(n) = O(g(n)), If there are positive constants n_0 and c such that, to the right of n_0 the f(n) always lies on or below c*g(n). Big-Oh (O) notation gives an upper bound for a function f(n) to within a constant factor. This means that, f(n) = O(g(n)), If there are positive constants n_0 and c such that, to the right of n_0 the f(n) always lies on or below c*g(n).  Omega (Ω) f(n) = Ω(g(n)), If there are positive constants n0 and c such that, to the right of n_0 the f(n) always lies on or above c*g(n). Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor. This means that, f(n) = Ω(g(n)), If there are positive constants n_0 and c such that, to the right of n_0 the f(n) always lies on or above c*g(n). Theta (Θ) f(n) = Θ(g(n)), If there are positive constants n_0 and c1 and c2 such that, to the right of n_0 the f(n) always lies between c1*g(n) and	8	CO1	L2

		c2*g(n) inclusive. Big-Theta(Θ) notation specifies a bound for a function f(n). This means that, f(n) = Θ(g(n)), If there are positive constants n0 and c such that, to the right of n0 the f(n) always lies on or above c1*g(n) and below c2*g(n).			

	b	Show that f(n)= Ω g(n) where f(n) = 3log n + 100 and g(n)=log n.
		
		
		If n=1:
		f(1) = 100 and g(1) = 0
		if n=100:
		f(100) = 119.92 [3*6.64+100] and g(1) = 6.64
		if n=1000;
		f(1000) = 129.91 [3*9.97+100] and g(1) = 9.97
		if n=100000;
		f(100000) = 149.83 [3*16.61+100] and g(1) = 16.61
		
		if n is a positive number f(n) is greater than or equal to g(n).
		i.e. 3log n + 100 ≥ log n
		
		f(n)= Ω g(n), where n_o= 1 and negative will not be considered for logarithmic calculation.
			7	CO1	L3

OR					
2.	**a**	Fermat's little theorem states that if *p* is prime and *a* is not divisible by *p*, then $a^{p-1} \equiv 1 \pmod{p}$ Use this theorem and design the algorithm to test whether the given number is prime or not. **Inputs**: *n*: a value to test for primality, *n*>3; **Output**: *prime* if *n* is *prime*, otherwise *composite*	8	CO1	**L3**

```
Prime0(n, α)
// Returns true if n is a prime and false otherwise.
// α is the probability parameter.
{
    q := n − 1;
    for i := 1 to large do  // Specify large.
    {
        m := q; y := 1;
        a := Random() mod q + 1;
        // Choose a random number in the range [1, n − 1].
        z := a;
        // Compute a^(n−1) mod n.
        while (m > 0) do
        {
            while (m mod 2 = 0) do
            {
                z := z^2 mod n; m := ⌊m/2⌋;
            }
            m := m − 1; y := (y * z) mod n;
        }
        if (y ≠ 1) then return false;
        // If a^(n−1) mod n is not 1, n is not a prime.
    }
    return true;
}
```

b Arrange the following functions in increasing order of their order of growth. Give an example for each of which asymptotic running time belongs to the respective order of growth. (e.g. log n – average case analysis of binary search)

- *log n, 2^n, n , n^3, n^2, nlogn, n!.*

Order of growth	Example
log n	average case analysis of binary search
n	Finding the smallest or largest item in an unsorted array
n log n	Merge sort
n^2	Bubble sort
n^3	Naive multiplication of two n×n matrices
2^n	Traveling salesman problem using a dynamic approach
n!	Breaking passwords using brute-force approach

7 | CO1 | **L2**

3. a Construct the algorithm for search and insert an element in a Binary search tree. Tabulate the best, worst, and average-case complexity of the binary search tree for each operation.

8 | CO1 | **L3**

Searching Algorithm

Input: key elements
Output: return the elements if found, otherwise returns null

```
struct node* search(int data){
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data){
```

```
    if(current != NULL) {
      printf("%d ",current->data);

      //go to left tree
      if(current->data > data){
        current = current->leftChild;
      }  //else go to right tree
      else {
        current = current->rightChild;
      }

      //not found
      if(current == NULL){
        return NULL;
      }
    }
  }

  return current;
}
```

Inserting Algorithm

Input: key elements
Output: insert key elements in the existing BST or create a new BST if the tree is empty

```
void insert(int data) {
     struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;

  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;

    while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;
        //insert to the left

        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      }  //go to right of the tree
      else {
        current = current->rightChild;

        //insert to the right
```

		Search O(log n) O(log n) O(n)			

	b	Construct the algorithm for the *Breadth-First Search (BFS)* traversal of a directed graph. Mention the best, worst, and average-case complexity of the BFS.
		
		Input − The list of vertices, and the start vertex.
		Output − Traverse all of the nodes, if the graph is connected.
		
		BFS(graph, start_node, end_node):
		  frontier = new Queue()
		  frontier.enqueue(start_node)
		  explored = new Set()
		
		  while frontier is not empty:
		    current_node = frontier.dequeue()
		    if current_node in explored: continue
		    if current_node == end_node: return success
		
		    for neighbor in graph.get_neigbhors(current_node):
		      frontier.enqueue(neighbor)
		    explored.add(current_node)
		
		Time complexity will be O(V+E) for all the cases.
			7	CO2	**L3**

OR					

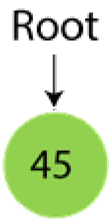
4.	**a**	Construct the algorithm for *Depth-First Search (DFS)* traversal of a directed graph. Mention the best, worst, and average-case complexity of the *DFS*.
		
		Input − The list of vertices, and the start vertex.
		Output − Traverse all of the nodes, if the graph is connected
		
		DFS(graph, start_node, end_node):
		  frontier = new Stack()
		  frontier.push(start_node)
		  explored = new Set()
		  while frontier is not empty:
			8	CO2	**L3**

```
current_node = frontier.pop()
if current_node in explored: continue
if current_node == end_node: return success

for neighbor in graph.get_neigbhors(current_node):
    frontier.push(neighbor)
explored.add(current_node)
```

Time complexity will be O(V+E) for all the cases.

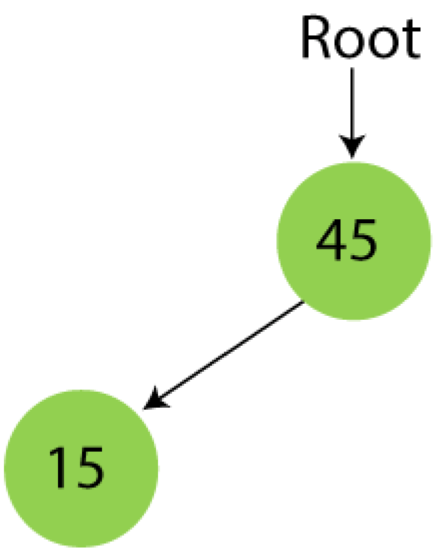
	b	Construct the binary search tree for the following data elements: *45, 15, 79, 90, 10, 55, 12, 20, 50* Trace the algorithm which you have constructed in 3a for every insertion and depict the final tree.	7	CO1	**L3**

Step 1 - Insert 45.



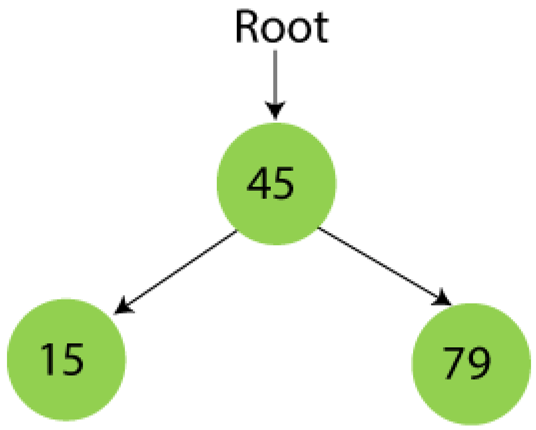
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



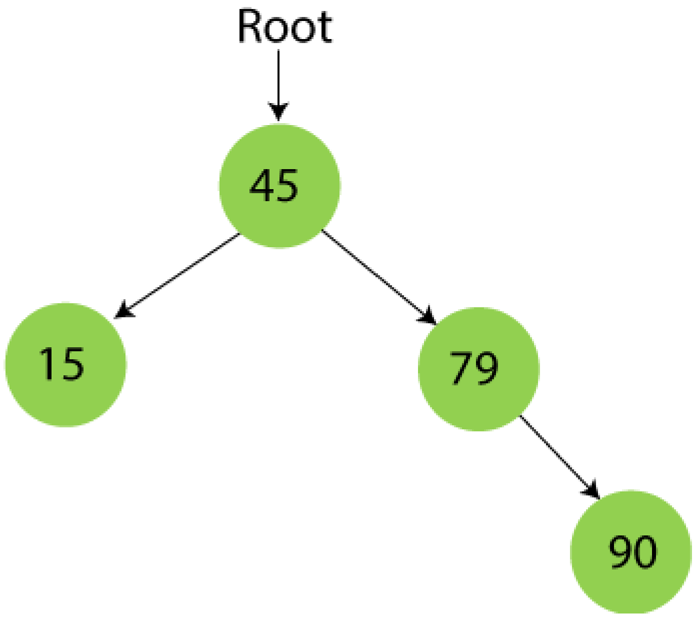
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



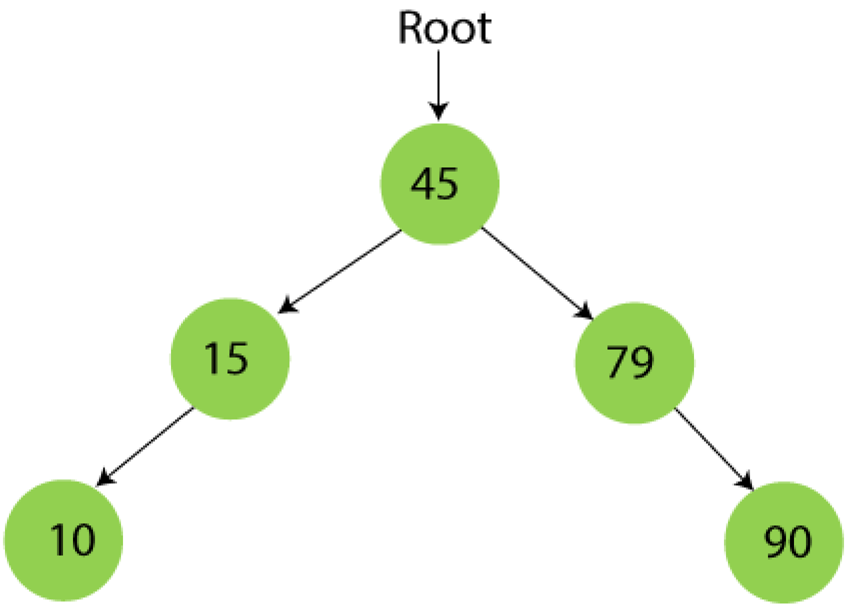
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



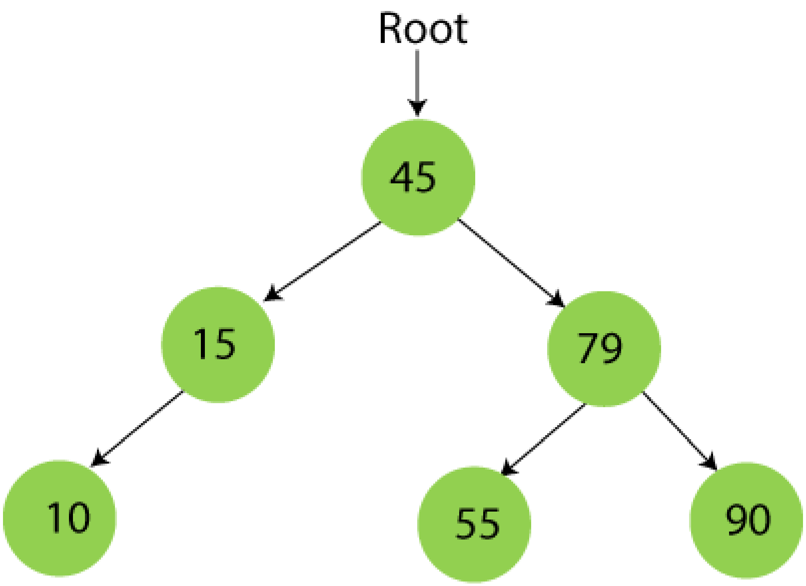
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



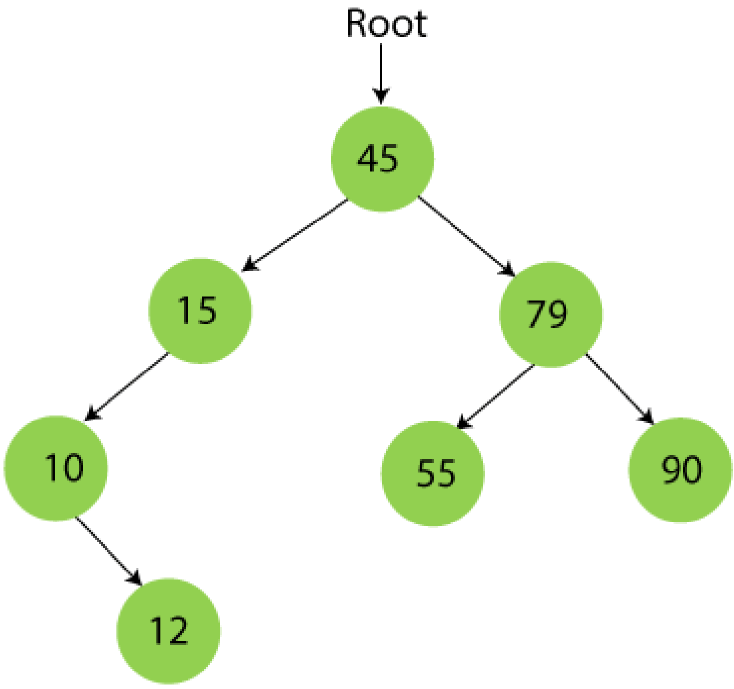
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



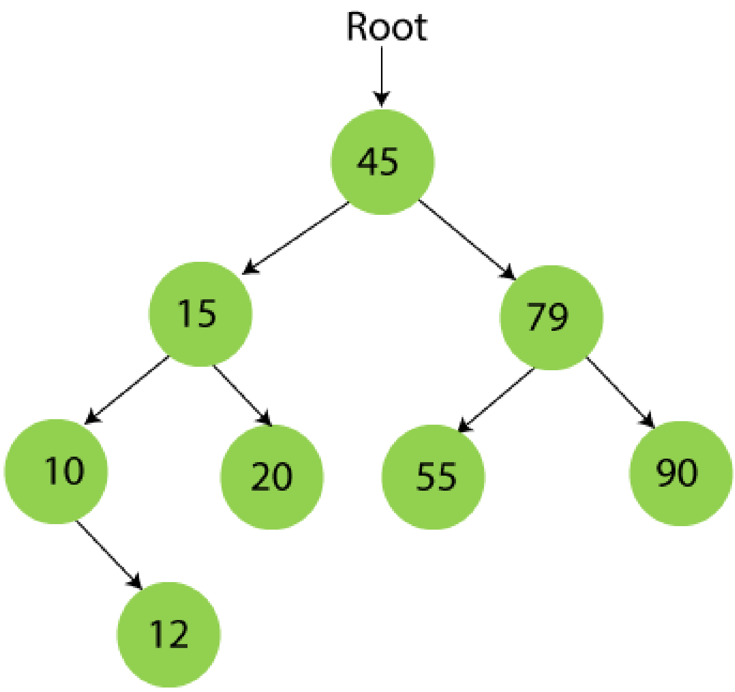
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



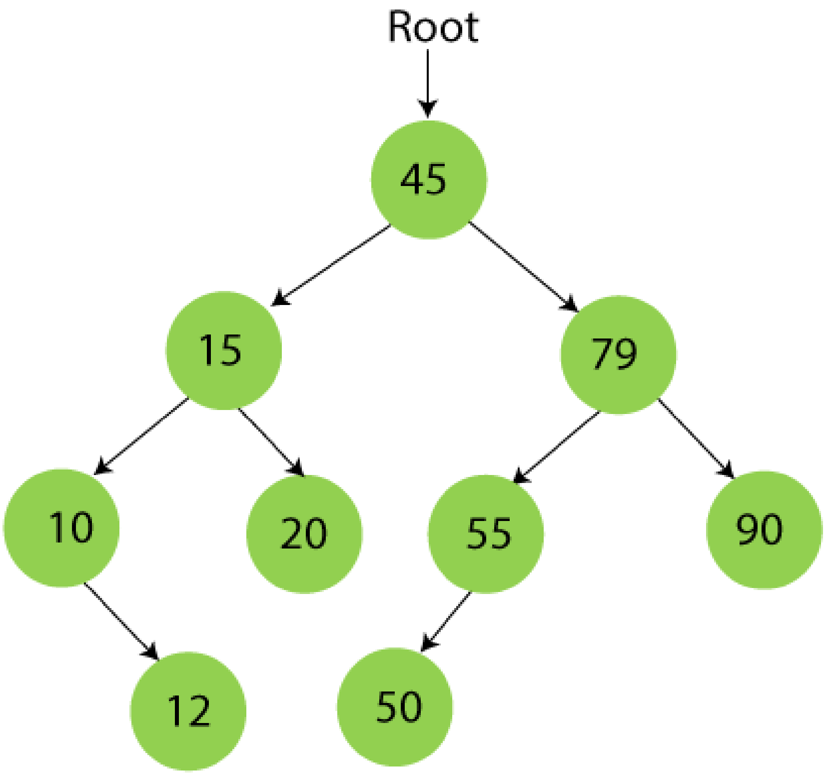
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed.

5.	**a**	Design an algorithm to sort the *n* number of elements using the divide and conquer technique and provide the time complexities for best average, and worst case.	8	CO2	**L3**

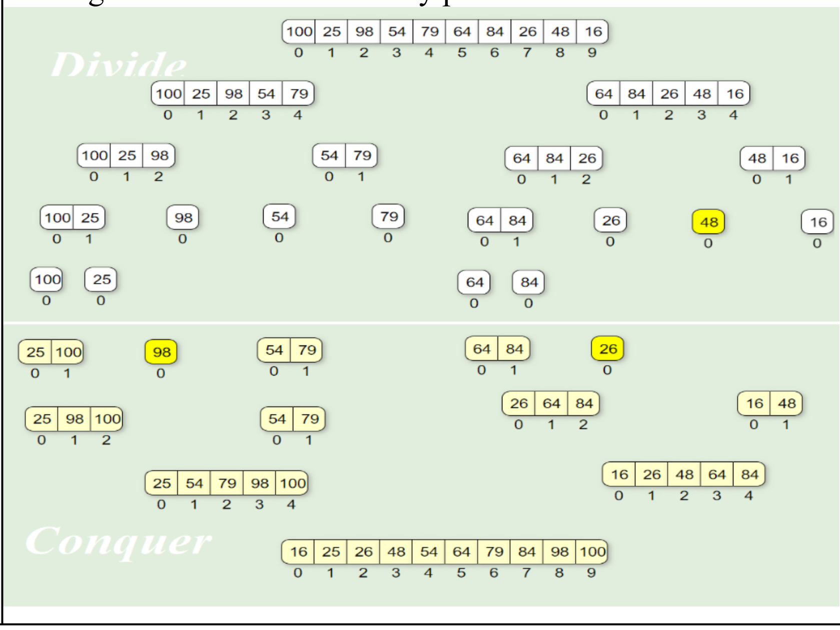
Either we can use Merge sort or Quick sort

Input: unsorted array of elements
Output: sorted array of elements

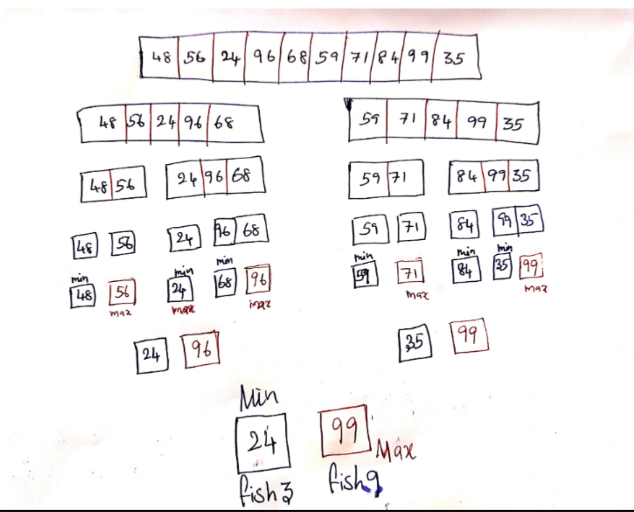
```
void merge(array, low, mid, high )
{
        int temp[MAX];
        int i = low;
        int j = mid +1 ;
        int k = low ;

        while( (i <= mid) && (j <=high) )
        {
                if(array[i] <= array[j])
                          temp[k++] = array[i++] ;
                else
                          temp[k++] = array[j++] ;
        }/*End of while*/

        while( i <= mid )
                temp[k++]=array[i++];

        while( j <= high )
                temp[k++]=array[j++];

        for(i= low; i <= high ; i++)
                array[i]=temp[i];

}/*End of merge()*/

void merge_sort( int low, int high )
```

```
{
        int mid;
        if( low != high )
        {
                mid = (low+high)/2;
                merge_sort( low , mid );
                merge_sort( mid+1, high );
                merge( low, mid, high );
        }
}/*End of merge_sort*/
```

Time complexity of merge sort will be O(n log n) for all the cases.

	b	Sort the following elements using the above-designed divide and conquer algorithm *100, 25, 98, 54, 79, 64, 84, 26, 48 and16.* Trace the algorithm for each and every process. 	7	CO2	L3

OR

6.	a	A man has 10 varieties of *fish*: *fish1, fish2 …… fish10* namely, in his aquarium. The number of fishes from *fish1, fish2…. fish10* are as follows: *48, 56, 24, 96, 68, 59, 71, 84, 99, and 35.* Apply the divide and conquer methodology to find the name of the *fish* which is maximum and minimum in the aquarium. We can apply the Maxmin algorithm to find the minimum and the maximum number of fish in the aquarium in minimum time complexity using the divide and conquer technique. 	8	CO2	L3

	b	Suggest and design the suitable algorithm for the above problem and also provide the time complexity for that algorithm.
		
		Algorithm MAXMIN (A, low, high)
		// Description : Find minimum and maximum element from array using divide and conquer approach
		// Input : Array A of length n, and indices low = 0 and high = n - 1
		// Output : (min, max) variables holding minimum and maximum element of array
		
		if n == 1 then
		return (A[1], A[1])
		else if n == 2 then
		if A[1] < A[2] then
		return (A[1], A[2])
		else
		return (A[2], A[1])
		else
		mid ← (low + high) / 2
		[LMin, LMax] = DC_MAXMIN (A, low, mid)
		[RMin, RMax] = DC_MAXMIN (A, mid + 1, high)
		if LMax > RMax then // Combine solution
		max ← LMax
		else
		max ← RMax
		end
		if LMin < RMin then // Combine solution
		min ← LMin
		else
		min ← RMin
		end
		return (min, max)
		end
			7	CO2	L3