// Q1. List out different OOPS principles and explain with examples. (10marks)
// Ans :
The Different Oops Principal are as Fllows :

Encapsulation: It is the mechanism of hiding the implementation details of an object and providing a public interface to access those details.
Example: A class "Car" can have private member variables like engine and wheels, and public methods like startEngine and stopEngine.

Inheritance: It is the mechanism of creating a new class from an existing class, thereby inheriting its properties and behavior.
Example: A class "SportsCar" can inherit from class "Car" and add new properties like turbo and aerodynamics.

Abstraction: It is the process of hiding the complexity of a system and exposing only the necessary information to the user.
Example: A class "Vehicle" can have abstract methods like move, stop and turn, which need to be implemented in derived classes like "Car" and "Bike".

Polymorphism: It is the ability of objects of different classes to respond to a method call in different ways,
based on the type of object calling the method. Example: A method "drive" can be called for objects of classes "Car" and "Bike" and produce different outputs based on the implementation in each class.

Composition: It is a way of combining simple objects or data types into more complex ones to achieve a desired functionality.
Example: A class "Driver" can have objects of class "Car" and "Bike" as its member variables, thereby achieving a more complex functionality.

Q2. Explain data structures that are mutable versus immutable. (5 marks)
Ans : In computer science, mutable data structures are those that can be changed after they are created, while immutable data structures cannot be changed.

Mutable data structures include arrays, lists, stacks, queues, trees, and hash tables. These data structures allow elements to be added, removed, or modified after they have been created.

Immutable data structures include strings, tuples, and numbers. Once these data structures are created, they cannot be changed. Instead, if you need to modify an immutable data structure, you must create a new instance with the desired changes.

The choice between mutable and immutable data structures depends on the specific use case. Mutable data structures are more flexible and can be more efficient in certain scenarios, but they also require careful management to ensure that their state remains consistent.
Immutable data structures, on the other hand, are simpler and easier to manage, but may not be as efficient in certain scenarios.

// Q3. Construct a binary tree using in-order and post-order traversal given below.
// Inorder Traversal: 9, 3, 15, 20, 7

// HERE WE ARE MAKING A TREE CLASS FOR ACCESS THE PROPERTIES WHILE PASSING THE VALUE TO IT AND MAKING TREES MEANS NODE AND IT LEFT SUB CHILD AND RIGHT SUB CHILD

```
class TreeNode {
    constructor (data , left , right){
        this.data = data;
        this.left = left;
        this.right = right;
    }
```

```
}

// LET CREATE A  ROOT NODE FOR TREE
const root = new TreeNode(9, null, null);
// HERE WE ARE  CREATING 1ST  CHILD FOR ROOT NODE
const node2 = new TreeNode(15, null, null);
// HERE ARE WE ARE CREATING A 2ND CHILD FOR ROOT NODE
const node3 = new TreeNode(3, null, null);
//  WE ARE CONNECTING A NODE 3 AS A LEFT SUB CHILD OF ROOT NODE AND THIS IS THE LEFT C
HILD OF THE ROOT NODE
root.left = node3;
// WE ARE CONNECTING A NODE 2 AS A RIGHT SUB CHILD OF ROOT NODE AND THIS IS THE RIGHT
CHILD OF THE ROOT NODE
root.right = node2;

// NOW CREATING A NEW NODE FOR CONNECTING WITH NODE 3 WHICH IS LEFT  CHILD OF ROOT
NODE
const node4 = new TreeNode(20, null, null);
//  HERE WE ARE CONNECTING TO THE LEFT SIDE OF THE NODE 3 TO NODE 4
node3.left = node4;
// NOW CREATING A NEW NODE FOR CONNECTING WITH NODE 3  WHICH IS LEFT  CHILD OF ROOT
NODE
const node5 = new TreeNode(7, null, null);
//  HERE WE ARE CONNECTING TO THE RIGHT SIDE OF THE NODE 3 TO NODE 5
node3.right = node5;


/**
 * NOW OUR TREES LOOK LIKE THIS
 * //          9
 *        3          15
 *     20    7
 */

// NOW 1ST WE ARE IMPLEMENTING INORDER WHICH IS (LEFT , ROOT , RIGHT)

// HERE ARE CREATING A INORDER FUNCTION TO VISIT (LEFT , ROOT , RIGHT)

function inorder(root) {
   // IMPLEMENTING A BASE CONDITION THAT IF THE ROOT MEANS IF THE NODE BECAME NULL J
UST EMPTY RETURN
   if (root === null) return;

   // VISTING THE LEFT CHILD FIRST BECAUSE THE CONCEPT FOR INORDER IS TO VISIT THE LEFT F
IRST
   inorder(root.left);
   // AFTER VISITING THE LEFT CHILD NOW WE HAVE TO VIST THE ROOT OF IT
   console.log(root.data);
   // AND THEN VISIT THE RIGHT NODE
   inorder(root.right);
 }

 // NOW LET CALL THE INORDER FUNCTION SO THAT WE GOT OUR OUTPUT
 console.log("inorder is 20,3,7,9,15")
 inorder(root)
```

```
//   Q3. Construct a binary tree using in-order and post-order traversal given below.
//       Post-Order Traversal: 9, 15, 7, 20, 3 (10 marks)

// HERE WE ARE MAKING A TREE CLASS FOR ACCESS THE PROPERTIES WHILE PASSING THE VALU
E TO IT AND MAKING TREES MEANS NODE AND IT LEFT SUB CHILD AND RIGHT SUB CHILD
class TreeNode {
    constructor (data , left , right){
        this.data = data;
        this.left = left;
        this.right = right;
    }
}

// LET CREATE A  ROOT NODE FOR TREE
const root = new TreeNode(9, null, null);
// HERE WE ARE  CREATING 1ST  CHILD FOR ROOT NODE
const node2 = new TreeNode(7, null, null);
// HERE ARE WE ARE CREATING A 2ND CHILD FOR ROOT NODE
const node3 = new TreeNode(15, null, null);
//  WE ARE CONNECTING A NODE 3 AS A LEFT SUB CHILD OF ROOT NODE AND THIS IS THE LEFT C
HILD OF THE ROOT NODE
root.left = node3;
// WE ARE CONNECTING A NODE 2 AS A RIGHT SUB CHILD OF ROOT NODE AND THIS IS THE RIGHT
CHILD OF THE ROOT NODE
root.right = node2;

// NOW CREATING A NEW NODE FOR CONNECTING WITH NODE 3 WHICH IS LEFT  CHILD OF ROOT
NODE
const node4 = new TreeNode(20, null, null);
//  HERE WE ARE CONNECTING TO THE LEFT SIDE OF THE NODE 3 TO NODE 4
node3.left = node4;
// NOW CREATING A NEW NODE FOR CONNECTING WITH NODE 3  WHICH IS LEFT  CHILD OF ROOT
NODE
const node5 = new TreeNode(3, null, null);
//  HERE WE ARE CONNECTING TO THE RIGHT SIDE OF THE NODE 3 TO NODE 5
node3.right = node5;


/**
 * NOW OUR TREES LOOK LIKE THIS
 * //          9
 *         15          7
 *     20    3
 */

    // HERE ARE CREATING A POSTORDER FUNCTION TO VISIT (LEFT , RIGHT , ROOT)

  function postorder(root){
    // IMPLEMENTING A BASE CONDITION THAT IF THE ROOT MEANS IF THE NODE BECAME NULL J
UST EMPTY RETURN
        if(root == null)
        return;
        //  VISTING THE LEFT CHILD FIRST BECAUSE THE CONCEPT FOR POSTORDER IS TO VISIT THE L
EFT FIRST
```

```
    postorder(root.left)
    // THEN VISIT THE RIGHT NODE AFTER VISITING LEFT NODE
    postorder(root.right)
    // THEN VISIT NODE ITSELF
    console.log(root.data)
    }

    // CALLING THE POSTORDER FUNCTION FOR PRINTING (LEFT , RIGHT , ROOT)
    console.log ("postorder 4,5,3,6,2,1")
    postorder(root)


// Q4. Construct a binary search tree using pre order traversal given below.
// Pre order Traversal: 50 30 20 40 70 60 80


// HERE WE ARE MAKING A TREE CLASS FOR ACCESS THE PROPERTIES WHILE PASSING THE VALU
E TO IT AND MAKING TREES MEANS NODE AND IT LEFT SUB CHILD AND RIGHT SUB CHILD
class TreeNode {
   constructor (data , left , right){
      this.data = data;
      this.left = left;
      this.right = right;
   }
}

// LET CREATE A  ROOT NODE FOR TREE
const root = new TreeNode(50, null, null);
// HERE WE ARE  CREATING 1ST  CHILD FOR ROOT NODE
const node2 = new TreeNode(20, null, null);
// HERE ARE WE ARE CREATING A 2ND CHILD FOR ROOT NODE
const node3 = new TreeNode(30, null, null);
//  WE ARE CONNECTING A NODE 3 AS A LEFT SUB CHILD OF ROOT NODE AND THIS IS THE LEFT C
HILD OF THE ROOT NODE
root.left = node3;
// WE ARE CONNECTING A NODE 2 AS A RIGHT SUB CHILD OF ROOT NODE AND THIS IS THE RIGHT
CHILD OF THE ROOT NODE
root.right = node2;

// NOW CREATING A NEW NODE FOR CONNECTING WITH NODE 3 WHICH IS LEFT  CHILD OF ROOT
NODE
const node4 = new TreeNode(40, null, null);
//  HERE WE ARE CONNECTING TO THE LEFT SIDE OF THE NODE 3 TO NODE 4
node3.left = node4;
// NOW CREATING A NEW NODE FOR CONNECTING WITH NODE 3  WHICH IS LEFT  CHILD OF ROOT
NODE
const node5 = new TreeNode(70, null, null);
//  HERE WE ARE CONNECTING TO THE RIGHT SIDE OF THE NODE 3 TO NODE 5
node3.right = node5;

const node6 = new TreeNode(60, null, null);
//  HERE WE ARE CONNECTING TO THE RIGHT SIDE OF THE NODE 2 TO NODE 6
node2.left = node6;

const node7 = new TreeNode(80, null, null);
```

```
//  HERE WE ARE CONNECTING TO THE RIGHT SIDE OF THE NODE 2 TO NODE 7
node2.right = node7;


/**
 * NOW OUR TREES LOOK LIKE THIS
 * //          50
 *        30         20
 *     40    70   60    80
 */

 // HERE ARE CREATING A PREORDER FUNCTION TO VISIT (ROOT, LEFT  , RIGHT)

 function preorder(root){
   // IMPLEMENTING A BASE CONDITION THAT IF THE ROOT MEANS IF THE NODE BECAME NULL J
UST EMPTY RETURN
     if(root == null)
     return;
     // VISTING THE ROOT ITSLEF FIRST BECAUSE THE CONCEPT OF THE PREORDER IS TO VISIT RO
OT FIRST
     console.log(root.data)
     // AFTER VISITING THE ROOT WE ARE VISITING THE LEFT SUB TREE NDOE
     preorder(root.left)
     // AND THEN VISIT RIGHT NODE
     preorder(root.right)
     }
     // NOW LET CALL THE PREORDER FUNCTION SO THAT WE GOT OUR OUTPUT
     console.log ("preorder is 1, 3,4,5,2,6")
     preorder(root)


     Q5.
     for(let i = 0;i<n;i++){
     j = 1;
     while(j<n){
     console.log(i)
     j = j*2
     }
     }

     ANS : Time Complexity Is 0(nlogn)

     Q6.
      i = 1;
      while(i2<n){
      i+=1}
     Ans : Time Complexity Is 0(Sqrt(n))

     Q7.
          function bblSort(arr){
          for(var i = 0; i < arr.length; i++){
          break;
          for(var j = 0; j < ( arr.length - i -1 ); j++){
          if(arr[j] > arr[j+1]){
          var temp = arr[j]
```

```
arr[j] = arr[j + 1]
arr[j+1] = temp
}
}
}
console.log(arr);
}
```

Ans : The outer loop runs n times, so it takes O(n) time. The inner loop, on the other hand, runs n-1, n-2, ..., 1 times for each iteration of the outer loop. This gives us a total of n * (n-1) / 2 iterations, which is a quadratic amount of time.

Therefore, the overall time complexity of the code is O(n^2).