Leet Code problems and approach

Note: drawing with https://drawisland.com/

Reverse Integer

```
public int reverseInt(int num){
   int reversed = 0;
   while( num!= 0){
      int digit = num %10; //1234%10 = 4 || 123%10 = 3 ||12%10 =2 ||1%10 =1
      reversed = reversed*10+digit; //4 || 40+3=43 ||430+2 = 432 || 4320+1 = 4321
      num = num/10; //123 ||12 ||1 ||0
   }
}
```

Inorder to handle Integer overflow - result overflow 32-bit integer

```
long reversed = 0;
      while(x!=0){
         int digit = x%10;
         reversed = reversed*10+digit;
         x = x/10;
      }
      if(reversed < Integer.MIN_VALUE || reversed > Integer.MAX_VALUE)
        return 0;
      else
        return (int)reversed;
```

Arrays

Move Zeros

```
class Solution {
   public void moveZeroes(int[] nums) {
      int offset =0;
      for(int j=0; j<nums.length;j++){
         if(nums[j]!=0){
            nums[j-offset] = nums[j];
            if(offset!=0)
               nums[j] = 0;
         }
         else
            offset++;
      }
   }
}
```

//O(N) time and O(1) space complexities

Add Binary

```
class Solution {
    public String addBinary(String a, String b) {
        StringBuilder sb=new StringBuilder();
        int i=a.length()-1;
        int j=b.length()-1;
        int carry =0;
        while(i>=0 || j>=0){  //O(A+B)
            int sum=carry;
            if(i>=0){
                sum+=a.charAt(i--)-'0'; //'1'-'0' ='1' // '0'-'0'='0'
            }
            if(j>=0){
                sum +=b.charAt(j--)-'0';
            }
            sb.insert(0,sum % 2);
            carry= sum/2;
        }
        if(carry>0){
            sb.insert(0,1);
        }
        return sb.toString();
    }
}
```

Intersection of two Arrays II

```
class Solution {
    public int[] intersect(int[] nums1, int[] nums2) {
        if(nums1== null || nums2 == null){
            return null;
        }
        if(nums1.length==0 || nums2.length==0){
            return new int[0];
        }

        Arrays.sort(nums1);//nlogn
        Arrays.sort(nums2);//mlogm
```

```
        int i =0, j=0;
        List<Integer> intersect = new ArrayList<>();
        // Set<Integer> intersect=  new HashSet<Integer>(); // incase to return unique values
        while(nums1.length > i && nums2.length > j){ //O(min(n,m)) worst case is if N=M
            if(nums1[i] == nums2[j])  {
                intersect.add(nums1[i]);
                i++;
                j++;
            }
            else if(nums1[i] < nums2[j]){
                i++;
            }else if(nums1[i] > nums2[j]){
                j++;
            }
        }

        // int j =0;
        // while(nums2.length>j){
        //     int index= Arrays.binarySearch(nums1, nums2[j]);
        //     if(index > -1){
        //         intersect.add(nums2[j]);
        //     }
        //     j++;
        // }

        int size = intersect.size();
        int[] result = new int[size];
        Integer[] temp = intersect.toArray(new Integer[size]);
        for (int n = 0; n < size; ++n) {
            result[n] = temp[n];
        }
        return result;
    }
}
//followup :
//What if elements of nums2 are stored on disk, and the memory is limited such that you cannot load all elements into the memory at once?
// This one is open-ended. But you have to think of divide and conquer. We can always let the memory take care of a segment of our nums2 or nums1.
```

Valid Parentheses

```
class Solution {
   public boolean isValid(String s) {
      // if(s.length()== 0 || s.length()==1)
      //     return true;

      Stack<Character> stack = new Stack<Character>();
      for(char c: s.toCharArray()){
         if(c == '(' || c=='{' || c=='['){
            stack.push(c);
         }else if(!stack.isEmpty() && stack.peek() == '(' && c==')'){
            stack.pop();
         }else if(!stack.isEmpty() && stack.peek() == '{' && c=='}'){
            stack.pop();
         }else if(!stack.isEmpty() && stack.peek() == '[' && c==']'){
            stack.pop();
         }else{
            return false;
         }
      }
      return stack.isEmpty();
```

Linked List

Remove the Nth node from end

```
   public ListNode removeNthFromEnd(ListNode head, int n) {
      ListNode saver= new ListNode(0); //create a dummy node and point to head
      saver.next = head;
      ListNode slow = saver;
      ListNode fast = saver;
      if(head == null || head.next== null)
         return null;
      for(int i=0; i<n ; i++){                    //move fast node n times
         if(fast == null) return head; // if n is greater than the length of the linked list. Edge case
         fast = fast.next;
      }
      while(fast.next!=null){
         slow = slow.next;
         fast = fast.next;
      }
```

```
        slow.next = slow.next.next;  // by this step slow points to node before the Nth node from
last
        return saver.next; // return the head
    }
```

Flatten a binary tree to Doubly Linked list

INorder

- Traverse to left
- Process →
- Traverse to right

  CIrcular doubly linked list . Store the left most child and link it to the right node.

Tree

Basic tree questions:
isSameTree()-

isSymmetricTree() - O(n) time , O(h) space - height

- Check whether the tree when divided at center must look like a mirror of another side
  - Recursively:

```
isSymmetric(TreeNode root){
        if(root == null) return true;
}

isSymmetric(TreeNode left, TreeNode right){
//if both nodes are null - return true
//if either one null - return false
//if node values are not equal - return false
//recursively call the same method for its subtree- palindromic check
if(!isSymmetric(left.left, right.right))
        Return false;
```

```
        if(!isSymmetric(left.right,right.left))
            Return false;
        Return true;
        }
```

- Iteratively: with the help of a Queue

```
    isSymmetric(TreeNode node){
        //create a queue
        Queue<TreeNOde> q = new LinkedList<>();
        q.add(node.left);
        q.add(node.right);
        while(!q.empty()){
            tempLeft = q.remove():
            tempRight = q.remove();
            //if both nodes are null - return true
            //if either one null - return false
            //if node values are not equal - return false
            //add palindromic nodes
            q.add(tempLeft.left);
            q.add(tempRight.right);
            q.add(tempLeft.right);
            q.add(tempRight.left);
        }
        Return true;
    }
```

Inorder : left,root,right
Preorder: left,right,root
Postorder: root,left,right

Inorder traversal of Binary Tree

Recursive:

```
public void inorder(TreeNode node){
    if(node != null)
        inorder(node.left);
    else
        return;
    System.out.println(node);
    inorder(node.right);
}
```

<u>Iterative: (with stack)</u>

```
public void inorder(TreeNode root){
        Stack<TreeNode> s = new Stack<>();
        TreeNode curr = root;
         while(!s.isEmpty() || curr!=null){
              if(curr != null){
                 s.put(curr);
                 curr = curr.left;
              }
              else
              {
                 curr = s.pop();
                  System.out.println(curr.val);
                  curr = curr.right;
              }
        }
}
```

<u>Iterative: (without stack) - Morris Traversal</u>

Algorithm:

```
   current = root
   while current is not NULL
      if not exists current.left
         visit(current);
         current = current.right;
      else
         predecessor = findPredecessor(current);
      if not exists predecessor.right;
         predecessor.right = current;
         current = current.left;
      else
         predecessor.right = null;
         visit(current);
         current = current.right;
```

Code:

```
  public void inorder(TreeNode node){
      TreeNode current = node;
      while(current != null){
         if(current.left == null)
            System.out.println(current);
```

```
            current = current.right;
        else
            TreeNode predecessor = current.left;
            while(predecessor.right !=null && predecessor.right!=current){
                predecessor = predecessor.right;
            }
            if(predecessor.right == null){
                predecessor.right = current;
                current = current.left;
            }
            else{
                predecessor.right = null;
                System.out.println(current);
                current = current.right;
            }
      }
   }
```

Preorder traversal:

Recursive:

```
public void preorder(TreeNode node){
        if(node != null)
        {
                System.out.println(node);
                preorder(node.left);
                preorder(node.right);
        }
    }
```

Iteratively:

Stack - add right before left

POst order traversal:

Recursive:

```
public void postorder(TreeNode node){
        if(node != null)
        {
                postorder(node.left);
                postorder(node.right);
                System.out.println(node);
        }
    }
```

Iterative : 2 Stacks to solve

```
Public void postorder(TreeNode node){
        if(node == null) return;
        Stack<TreeNode> s1 = new Stack<>();
        Stack<TreeNode> s2 = new Stack<>();
        s1.push(node);
        while(!s1.empty()){
                TreeNode curr = s1.pop();
                s2.push(curr);
                if(curr.left != null) s1.push(curr.left);
                if(curr.right !=null) s1.push(curr.right);
        }
        while(!s2.empty()){
                System.out.println(s2.pop());
        }
}
```

Build tree from inorder and preorder (array)

```
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
```

- Get length for each array
  int inStart = 0;
  int inEnd = inorder.length - 1;
  int postStart = 0;
  int postEnd = postorder.length - 1;
- buildtree(iArray,istart,iend,pArray,pstart,pend);
  - if (inStart > inEnd || postStart > postEnd)
    return null;
  - int rootValue = postorder[postEnd]; //get last index value of pArray and create root
    TreeNode root = new TreeNode(rootValue);
  - Check for rootValue in the iArray
    ```
    int k = 0;
        for (int i = 0; i < inorder.length; i++) {
            if (inorder[i] == rootValue) {
                k = i;
                break;
            }
        }
    ```
  - Build left subtree
    - buildtree(iArray,istart,k-1,pArray,pStart+k-(istart+1);

- Build right subtree
  - buildtree(iArray,k+1,iEnd,pArray,pStart+k-iStar,pEnd-1);

Search Binary Tree for a key

Recursive approach:

```
TreeNode  binarySearch(TreeNode root, int key){
   if(root == null)
      return null;
   if(root.val == key)                //key equal to node value
      return root;
   else if(root.val < key)            //key greater than node value → search in right
      return binarySearch(root.right,key);
   else
      return binarySearch(root.left,key); //key lesser than node value → search in left
}
```

Insertion into BST

Iterative approach: (to check) o(N) for unbalance BST / o(log N) for balanced AVL Trees

```
TreeNode  insertBST(TreeNode root, int value){
   TreeNode newNode = new TreeNode(value);
   TreeNode prev = null, curr = null;
   if(root == null)
      return newNode;
   while(current.next!=null){
      if(curr.val > value)
      {
         prev = curr;
         curr = prev.left;
      }else{
         prev = curr;
         curr = prev.right;
      }
   }
   if(curr.val > value)
      curr.left = newNode;
   else{
      TreeNode temp = prev.left;
      prev.left = newNode;
      newNode.left = temp;
```

```
    }
}
```

SameTree validation

Recursive approach:

- Validate root. Both null → return true, Either one null → return false
- Validate child recursively.

```
boolean sameTree(TreeNode root1, TreeNode root2){
    if(root1 == null && root2==null)
        return true;
    if(root1 == null || root2 == null)
        return false;
    return (root1.val == root2.val)
            && sameTree(root1.left, root2.left)
            && sameTree(root1.right, root2.right);
}
```

Size of BST

- Recursive approach

```
int sizeTree(TreeNode root){
    if(root == null)
        return 0;
    int leftSize = sizeTree(root.left);
    int rightSize = sizeTree(root.right);
    return leftSize+rightSize+1;
}
```

Validate BST

Inorder

- Traverse to left

- Process → check min >= node.data >max -> false (for first pass min = Integer.MIN_VALUE, max = Integer.MAX_VALUE)
- Traverse to right

Recursive (validate using inorder traversal)

```
TreeNode prev = null;
Boolean isValidBST(TreeNode node){
        if(node == null)
                Return true;
        if(!isValidBST(node.left))
                Return false;
        if( prev!=null && node.val <= prev.val)
                Return false;
        Prev = node;
        Return (isValidBST(node.right));
}
```

Diameter of a tree

Number of node on the longest path of the binary tree.

Cases:

- Diameter passes throu the root.
  Height of left subtree + 1(for root) + height of the right subtree

- Diameter does not pass thru root
  max(left height, right height)

```
public int diameterOfBinaryTree(TreeNode root) {
      if(root== null) return 0;
/***CaSE1****/
      int lheight= height(root.left);
      int rheight= height(root.right);
/***CaSE2****/
      int ldiameter = diameterOfBinaryTree(root.left);
      int rdiameter = diameterOfBinaryTree(root.right);

      return Math.max(lheight+rheight, Math.max(ldiameter,rdiameter));
   }

   private int height(TreeNode root){
      if(root == null)
```

```
        return 0;
    return 1+Math.max(height(root.left),height(root.right));
}
```

Determine **Height** of Tree BST

Height = 1 + No.of edges in the longest path from root to leaf node.

- Traverse left - call lheight = height(left) recursively
- Traverse right - rheight= height(right)
- if(lheight > rheight) return h =1+lheight;
  Else return h= 1+rheight;
  Return h;

Is Height balanced tree
The absolute difference between left subtree and right subtree should not be more than one level or leaf node

"a binary tree in which the depth of the two subtrees of *every* node never differ by more than 1."

```
    3
   / \
  9  20
    /  \
   15   7
```

Return true

```
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
```

Return false

Balanced Binary tree =(left Subtree height - right subtree height)<=1

public boolean isBalanced(TreeNode root) {

```
    if(checkBalanceHeight(root) > -1)
        return true;
    else
        return false;
}

private int checkBalanceHeight(TreeNode node){
    if(node == null)
        return 0;
    int h1 = checkBalanceHeight(node.left);
    int h2 = checkBalanceHeight(node.right);
    if(h1==-1 || h2==-1)
        return -1;
    if(Math.abs(h1-h2) >1)
        return -1;
    return Math.max(h1,h2)+1;
}
```

Print Order

Level - order

- One line print
    - Queue
        - Enqueue *node*
        - Dequeue node and print
        - Enqueue left and right child of *node*

- LEvel by Level [https://www.youtube.com/watch?v=7uG0gLDbhsI]
    - Queue
        - MEthod 1: (o(n) time o(n) space) with a queue and a delimiter
            - Enqueue root
            - Enqueue *null* (to indicate that level is over) - delimiter
            - Node = dequeue() and print (if encounter null ‘ print \n’)
            - Enqueue NOde.left and Node.right
            - IF node== null
            - Change level
                - ENQUEUE NULL
        - MEthod 2: (o(n) time o(n) space) with a queue and two counters
            - Enqueue root , levelCnt=1 and currCnt=0
            - Loop till (! queue.isEmpty())
                - Curr = queue.poll() → top element, add to result list

- Loop while (levelCnt!=0)
    - check for left node(yes -> queue.offer(left),currCnt++
    - Check for right node (yes-> queue.offer(right),currCnt++
    - levelCnt--
- Add each level to result list
- Set levelCnt = currCnt, currCnt=0; curr = null
- Return result

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if(root == null)
      return result;
    TreeNode curr = null;
    int levelCnt = 1;
    int currCnt = 0;
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);
    while(!queue.isEmpty()){
      List<Integer> levelOrder = new ArrayList<Integer>();
      while(levelCnt!=0){
        curr = queue.poll();
        levelOrder.add(curr.val);
        if(curr.left!=null){
          queue.offer(curr.left);
          currCnt++;
        }
        if(curr.right!=null){
          queue.offer(curr.right);
          currCnt++;
        }
        levelCnt--;
      }
      result.add(levelOrder);
      levelCnt = currCnt;
      currCnt=0;
      curr = null;
    }
    return result;
  }
```

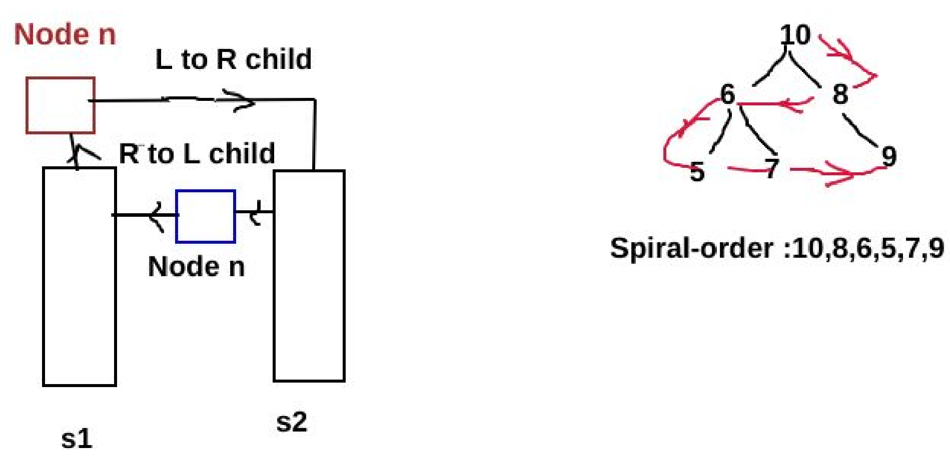
- Method 3: Recursive approach (uses call stack) - DFS

```
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if(root == null)
      return result;
    return dfs(result,root,0);
  }

  public List<List<Integer>> dfs(List<List<Integer>> result, TreeNode node,int level){
    if(result.size() <= level){
      result.add(new ArrayList());
    }
    result.get(level).add(node.val);
```

```
        if(node.left!=null){
            dfs(result,node.left,level+1);
        }
        if(node.right!=null){
            dfs(result,node.right,level+1);
        }
        return result;
    }
```

Spiral order (Zig-Zag level order)

Iterative approach:




```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if(root == null)
            return result;
        Stack<TreeNode> s1 = new Stack<>();
```

```
    Stack<TreeNode> s2 = new Stack<>();
    s1.push(root);
    int level = 0;
    TreeNode node = null;
    while(!s1.isEmpty() || !s2.isEmpty()){
        while(!s1.isEmpty()){
            node = s1.pop();
            if(result.size() <= level){
                result.add(new ArrayList());
                result.get(level).add(node.val);
            }
            else
                result.get(level).add(node.val);
            if(node.left!=null || node.right!=null){
                if(node.left!= null)
                    s2.push(node.left);
                if(node.right!=null)
                    s2.push(node.right);
            }
             if(s1.isEmpty())
                 level++;

        }
        while(!s2.isEmpty()){
            node = s2.pop();
            if(result.size() <= level){
                result.add(new ArrayList());
                result.get(level).add(node.val);
            }
            else
                result.get(level).add(node.val);
            if(node.left!=null || node.right!=null){
                if(node.right!= null)
                    s1.push(node.right);
                if(node.left!=null)
                    s1.push(node.left);
            }
            if(s2.isEmpty())
                level++;

        }

    }
    return result;
}
```

Recursive approach:

1. Add new list to the result list for each new level.
2. If the level is even add the sub tree node value normally.
3. Else if the level is odd add the sub tree node value to the first of existing list in that level.

Data structure:
Use LinkedList for result instead of ArrayList to make use of inserting into first index. addFirst()

```
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if(root == null)
            return result;
        return dfs(result,root,0);
    }

    public List<List<Integer>> dfs(List<List<Integer>> result, TreeNode node,int level){
        if(result.size() <= level){
            result.add(new LinkedList());
        }
        if(level %2 ==0)
            result.get(level).add(node.val);
        else
            ((LinkedList<Integer>)result.get(level)).addFirst(node.val);

        if(node.left!=null){
            dfs(result,node.left,level+1);
        }
        if(node.right!=null){
            dfs(result,node.right,level+1);
        }

        return result;
    }
```

Construct Binary Tree using Preorder and Inorder Traversal

```
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
```

Return the following binary tree:

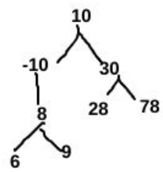
```
    3
```

```
   / \
  9  20
    /  \
   15   7
```

```
public TreeNode buildTree(int[] preorder, int[] inorder) {
        int iSt = 0;
        int iEnd = inorder.length-1;
        int pSt = 0;
        int pEnd = preorder.length-1;
        return buildTree(preorder, inorder, pSt,pEnd,iSt, iEnd);
    }
    public TreeNode buildTree(int[] preorder, int[] inorder, int pSt, int pEnd, int iSt, int iEnd){
        if(iSt > iEnd || pSt>pEnd)
            return null;
        int rootVal = preorder[pSt];
        TreeNode root = new TreeNode(rootVal);
        int k =0;
        for(int i =0; i<inorder.length;i++){
            if(rootVal == inorder[i])
            {
                k=i;
                break;
            }
        }
        root.left = buildTree(preorder,inorder,pSt+1,pSt+(k-iSt),iSt,k-1);
        root.right = buildTree(preorder,inorder,pSt+(k-iSt)+1,pEnd,k+1,iEnd);
        return root;
    }
```

Lowest common ancestor of Binary Search Tree

Lowest common ancestor in BST



Node n1 = 28, Node n2=78

if root < n1 && root <n2
move to right subtree
root > n1 && root >n2
move to left subtree
else
lower_ans = root;

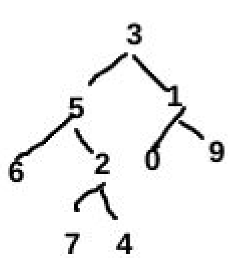
Recursive

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root.val > p.val && root.val > q.val)
            return lowestCommonAncestor(root.left, p, q);
        else if(root.val <  p.val && root.val < q.val)
            return lowestCommonAncestor(root.right, p, q);
        else
            return root;
    }
```

Iterative

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        while (true) {
            if (root.val < p.val && root.val < q.val) {
                root = root.right;
            } else if (root.val > p.val && root.val > q.val) {
                root = root.left;
            } else {
                return root;
            }
        }
    }
```

Lowest common ancestor of Binary Tree



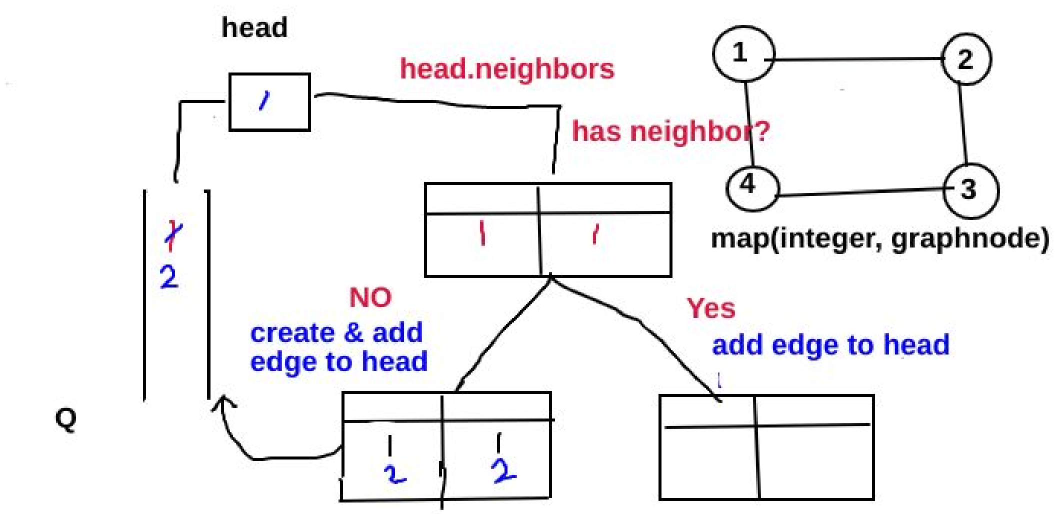
Node n1=7 , n2 =4 → LCA=2

Root == n1 or root == n2
    Return root
Left = LCA(root.left,n1,n2)
Right = LCA(root.right,n1,n2)
Left != null && right!=null
    Return root
Either one null
    Return non-null node

Recursive

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root == null)
           return null;
        if(root.val == p.val || root.val == q.val )
           return root;

        TreeNode left = lowestCommonAncestor(root.left,p,q);
        TreeNode right = lowestCommonAncestor(root.right,p,q);
        if(left!=null && right!=null)
           return root;
        return left!=null ? left:right;
}
```

Clone Graph (Undirected Graph)



Approach : BFS - iterative

map<Node,Node> - track of copies of node and take care of cyclic graph.
Queue <Node> - add neighbors of current node (start with Node passed).

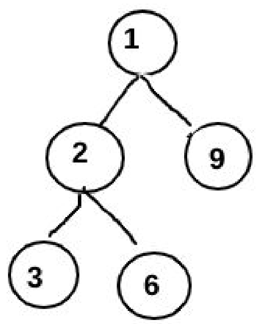
```
/**
// Definition for a Node.
class Node {
    public int val;
    public List<Node> neighbors;

    public Node() {}

    public Node(int _val, List<Node> _neighbors) {
```

```
        val = _val;
        neighbors = _neighbors;
    }
};
*/
public Node cloneGraph(Node node) {
        if(node == null){
            return null;
        }
        Queue<Node> queue = new LinkedList<Node>();
        Map<Node,Node> map = new HashMap<Node,Node>();

        queue.add(node);
        map.put(node,new Node(node.val,new ArrayList<Node>()));

        while(queue.size() > 0){
            Node head = queue.remove();
            for(Node neighbor: head.neighbors){
                if(!map.containsKey(neighbor)){
                    map.put(neighbor,new Node(neighbor.val, new
ArrayList<Node>()));
                    queue.add(neighbor);
                }
                map.get(head).neighbors.add(map.get(neighbor));
            }
        }
        return map.get(node);
    }
```

Approach : DFS

Cartesian Tree for a given sequence

Input [3,2,6,1,9]
Resulting Cartesian Tree




```
public TreeNode buildTree(ArrayList<Integer> A) {
        return buildTree(A,0,A.size()-1);
    }

public TreeNode buildTree(ArrayList<Integer> A, int start, int end){
        if(start > end)
            return null;
        int min_index = minimumIndex(A,start,end);
        TreeNode root = new TreeNode(A.get(min_index));
        root.left = buildTree(A, start, min_index-1);
        root.right = buildTree(A, min_index+1, end);
        return root;
    }

    private int minimumIndex(ArrayList<Integer> list,int start, int end){
        int minIndex = start;
        for (int i = start + 1; i <= end; i++)
        {
            if (list.get(minIndex) > list.get(i)) {
                minIndex = i;
            }
        }
        return minIndex;
    }
```

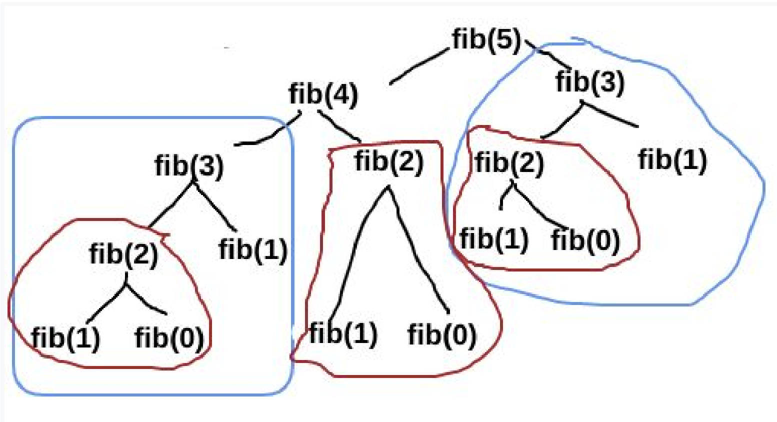
Vertical order
(LEvel - order + HashTable)

- Enqueue root , distance of node(hd) from wall as 0
- Add the Hd to hashtable as key and node as value (0,a)
- Dequeue node and check for left and right child and
- update hashtable
- Left child (hd =hd-1)
- Right child (hd = hd+1)
- Enqueue node and check for left and right child .

Dynamic programming

Fibonacci Series

```
F(0) = 0,   F(1) = 1
F(N) = F(N - 1) + F(N - 2), for N > 1.

Qtn to interviewer : Range of input value : int or long
```




```
public long fibonacci(int x){
    if(x < 0) return -1;
    if(x == 0) return 0;
    long[] cache = new long[x+1];

    for(int i=1; i<cache.length;i++){
        cache[i] = -1; //cache[x] =[0,-1,-1,-1…-1]
        cache[1] = 1;//base case, cache[x] =[0,1,-1,-1…-1]
    }
    return fibonacci(x,cache);

}

private long fibonacci(int x, long[] cache){
    if(cache[x] > -1)
        return cache[x];
    cache[x] = fibonacci(x-1,cache) + fibonacci(x-2,cache);
    return cache[x];
}
```

```
Space complexity : O(X) → trade-off
Time complexity : incrementally it becomes linear time as we calculate and
cache
```

Optimize : linear space
https://www.youtube.com/watch?v=OQ5jsbhAv_M - 30.10

Longest Common Subsequence

https://www.youtube.com/watch?v=NnD96abizww

Given two sequences, find the length of the longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Input1 = **abc**da**f** Input2 = **a**c**bcf**

LCS -> **abcf**

Input1 ---------- input2		a	b	c	d	a	f
	0	0	0	0	0	0	0
a	0	**1**	1	1	1	1	1
c	0	**1**	1	2	2	2	2
b	0	1	**2**	2	2	2	2
c	0	1	2	**3**	**3**	**3**	3
f	0	1	2	3	3	3	**4**

Track all path marked in blue and **bold** from f to a - abcf
Formula:

```
        if(input1[i] == input2[j]) then  T[i][j] = T[i-1][j]+1;
        Else
            T[i][j] = max(T[i-1][j], T[i][j-1]);
```

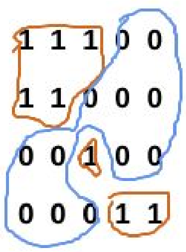
Number of Island

Given a 2d grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands **horizontally or vertically**. You may assume all four edges of the grid are all surrounded by water.

DFS approach: o(m*n) time

```
1 1 1 0 0
1 1 0 0 0
0 0 1 0 0
0 0 0 1 1
```

**island connects if it is '1'
and its top,bottom,left,right is '1**



Number of island: 3

```
public int numIslands(char[][] grid) {
        if(grid == null)
            return 0;
        int count =0;
        for(int i =0; i<grid.length;i++){
            for(int j=0; j<grid[i].length;j++){
                if(grid[i][j] == '1')
                {
                    count +=dfs(i , j, grid);
                }
            }
        }
        return count;
    }

public int dfs(int i, int j, char[][] island){
       if(i <0 || i>= island.length || j< 0 || j>=island[i].length ||
island[i][j] =='0'){
            return 0;
        }
        island[i][j] = '0';
        //bottom
        dfs(i+1,j,island);
        //top
        dfs(i-1,j,island);
        //left
        dfs(i,j-1,island);
        //right
        dfs(i,j+1,island);
        return 1;
    }
```

Backtracking

Generate Parentheses

https://www.youtube.com/watch?v=sz1qaKt0KGQ

The 3 Keys To Backtracking

Our Choice:
Whether we place a left or right paren at a certain decision point in our recursion.

Our Constraints:
We can't place a right paren unless we have left parens to match against.

Our Goal:
Place all k left and all k right parens.

The Key

At each point of constructing the string of length 2k we make a choice.

We can place a "(" and recurse or we can place a ")" and recurse.

But we can't just do that placement, we need 2 critical pieces of information.

The amount of left parens left to place.
The amount of right parens left to place.

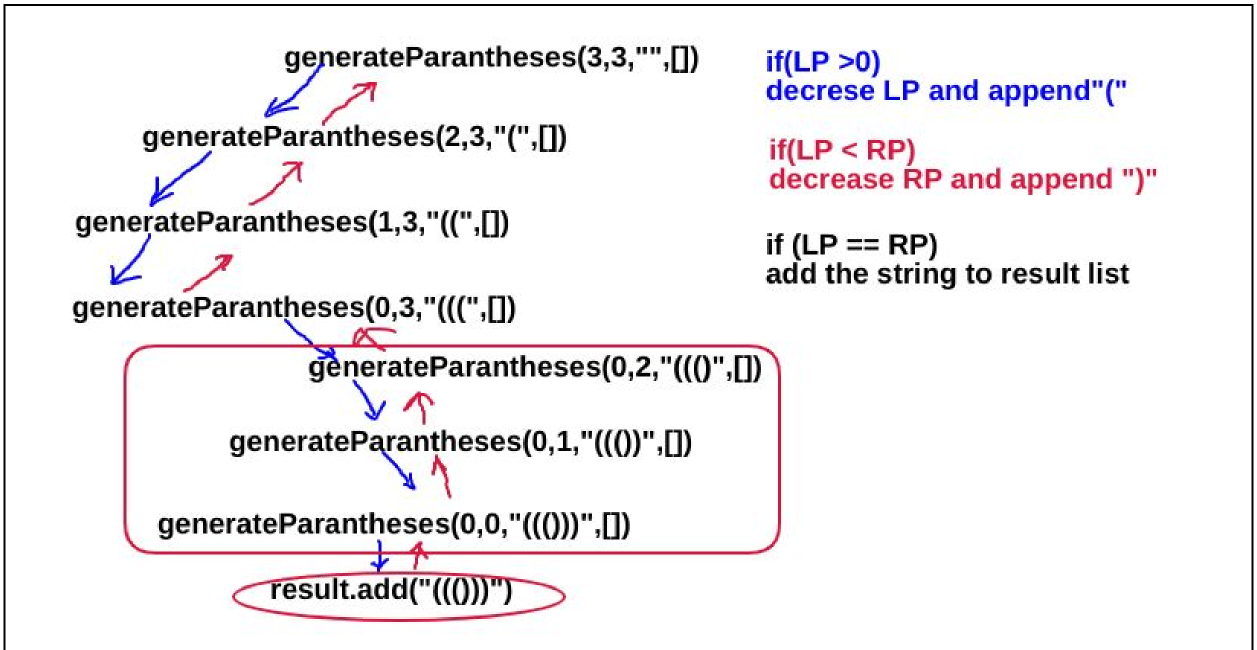
We have 2 critical rules at each placement step.

We can place a left parentheses if we have more than 0 left to place.

We can only place a right parentheses if there are left parentheses that we can match against.

We know this is the case when we have less left parentheses to place than right parentheses to place.

Once we establish these constraints on our branching we know that when we have 0 of both parens to place that we are done, we have an answer in our base case.

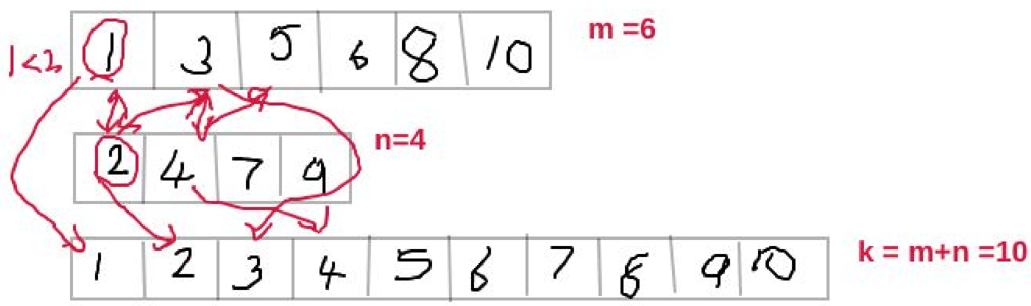



```
public List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<String>();
        generateParanthesis(n, n,"",result);
        return result;
    }

private void generateParanthesis(int lParan,int rParan, String
inProgress,List<String> result){
  //we have used all left and right paranthesis
        if(lParan == 0 && rParan==0){
            result.add(inProgress);
        }
//if i have left paranthesis to use - start using it.
        if(lParan > 0){
            generateParanthesis(lParan-1, rParan, inProgress+"(",result);
        }
//if left paranthesis is less than right paranthesis - start using right
paranthesis
        if(lParan < rParan){
            generateParanthesis(lParan, rParan-1, inProgress+")", result);
        }
    }
```

Sorting and searching

Merge 2 sorted arrays:

- comparing two arrays from beginning and keep adding elements in ascending order
    * time - O(mn) , space - O(mn)



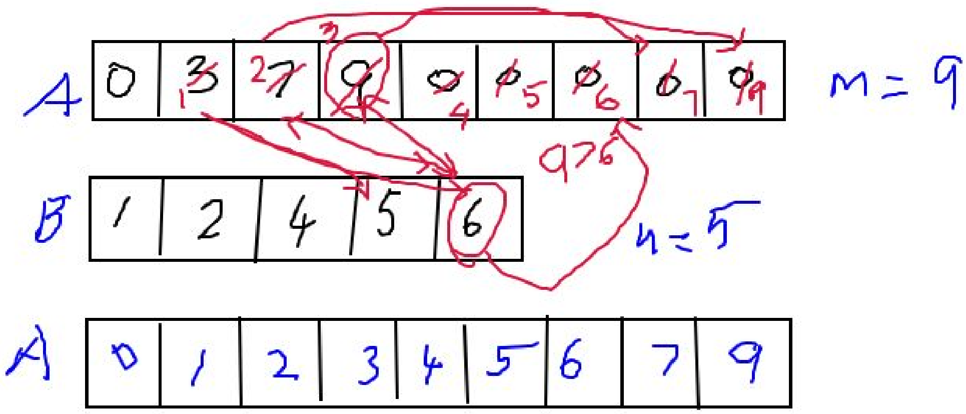
Time : O(mn) Space : O(mn)

```
public void merge(int[] nums1, int m, int[] nums2, int n) {
        int[] mergedArray = new int[m + n];
        /**
        merge in a new array
        */
        int k =0, i=0, j=0;
        while(i < m.length || j < n.length){
            if(nums1[i] < nums2[j])
                mergedArray[k++] = nums1[i++];
            else
                mergedArray[k++] = nums2[j++];
        }
        while(i < m.length){
            mergedArray[k++] = nums1[i++];
        }
        while(j < n.length){
            mergedArray[k++] = nums2[j++];
        }
}
```

-In-place :

<u>if there are enough space in nums1 to hold nums2</u>, compare from last value

*time - O(mn) space - O(1)



Time : O(mn) **Space: O(1)**

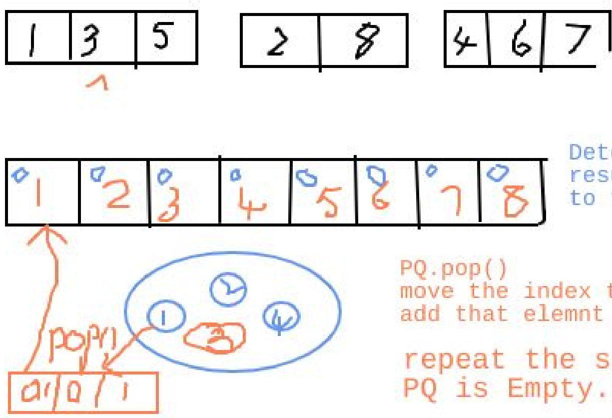
```
public void merge(int[] nums1, int m, int[] nums2, int n) {
       /**
            merge in-place
        */
        int i=n-1; int j=n-1; int k = (m+n)-1;

        while(j>=0){
            if(i >=0 && nums1[i] > nums2[j]){
                nums1[k--] = nums1[i--];
            }else{
                nums1[k--] = nums2[j--];
            }
        }
    }
```

Merge K sorted arrays

merge K sorted array
[2,3,4] [5,6,7] [8,9,10]

- possible 1:
  merge and sort all the array
  any merge is nlogn = kn log(kn)
- possible 2:
  merging one index of all array at a time
  kn * k
- possible 3:
  using a priority queue
  adding to PQ = log n

remove from PQ = constant time
kn log k



k=3 & size = 3+2+3 =8

Determine the size of result array and add all the 1st to the PQ.

PQ.pop()
move the index to next element in the array
add that elemnt to PQ and continue

repeat the same till PQ is Empty.

```
private static class QueueNode implements Comparable<QueueNode>{
        int arrayName , index, value;
        public QueueNode(int arrayName, int index, int value){
            this.arrayName = arrayName;
            this.index = index;
            this.value = value;
        }
        public int compareTo(QueueNode n){
            if(value > n.value) return 1;
            if(value < n.value) return -1;
            return 0;
        }
    }

    public static int[] mergeKSortedArrays(int[][] arrays){
        PriorityQueue<QueueNode> pq = new
PriorityQueue<QueueNode>();
        int size =0;
        for(int i =0; i < arrays.length ; i++){
            size +=arrays[i].length;
            pq.add(new QueueNode(i, 0,arrays[i][0]));
        }
        int[] result = new int[size];
        for(int visit=0; !pq.isEmpty(); visit++){
            QueueNode node = pq.poll();
            result[visit] = node.value;
            int newIndex = node.index+1;
            if(newIndex < arrays[node.arrayName].length){
              pq.add(new QueueNode(node.arrayName, newIndex,
```

```
arrays[node.arrayName][newIndex]));
                }
            }
            return result;
        }
```

Sort color (0,1,2)

```
// 1, 0, 2, 0, 2
//^^        ^
//LM        H

public void sortColors(int[] nums) {
    int low =-1;
    int mid =0;
    int high = nums.length;
    while(mid<high){
        if(nums[mid]==1)
            mid++; // if 1, the mid pointer moves up
        else if(nums[mid]==0){
            ++low; // if 0, low pointer is moved up, swap and increase mid
            swap(nums, mid,low);
            mid++;
        }else{
            high--; // if 2, high is moved down and swap
            swap(nums, mid,high);
        }
    }
}
void swap(int[] nums, int i, int j){
    int temp = nums[i];
    nums[i]= nums[j];
    nums[j] = temp;
}
```

Ascending order sorted and rotated array

Steps to solve without duplicates

A. Find minimum number index / rotation point index in an array
   a. Initialise left = 0 , right = nums.length-1
   b. Loop till left < right (don't check left ==right, that is the break point for this loop as Left and right will point to Min value)
      i. Find pivot mid = (left +right)/2
      ii. If **pivot** index value **> right**, right-side has the shifting point, left = pivot+1; (as we have to move towards the rotation point).
      Else search in Left side. Right = pivot
   c. Return min value.

```
public int findMin(int[] nums) {
        int left =0;
        int right = nums.length -1;
        int pivot = 0;
        while(left < right){
            pivot = (left +right)/2;
            if(nums[pivot] > nums[right]){
                left =pivot+1;
            }
            else{
                right = pivot;
            }
        }

        return nums[left];
    }
```

B. Search in Rotated array for target
   a. Initialise left =0; right = nums.length-1;
   b. Loop until Left <= Right (left == right is included in this case as we are searching for a target value & chances of its presence must be checked till end)
      i. Find the pivot = (left+right)/2
      ii. If nums[pivot] == target return true;
      iii. Determine the rotated side and check the target in two cases
         1. Case 1: nums[pivot] >=nums[left] -> shift point is in right side
         If nums[left] <= target<nums[mid] → chk target lies in the left side
               Right = pivot-1;

Else
Left = pivot+1;

2. Case 2:else -> shift point is in left side
If nums[left] <= target<nums[mid] →chk target lies in the right side
left = pivot+1;
Else
right = pivot-1;

iv. Return false

```
public int search(int[] nums, int target) {
        int left =0;
        int right = nums.length -1;
        while(left <= right){
            int mid = (left+right)/2;
            if(nums[mid] == target){
                return mid;
            }
            if(nums[left]<= nums[mid]){
                if(nums[left] <= target && target < nums[mid]){//[4,5,6,7,0,1,2]
                    right = mid-1;
                }else{
                    left = mid+1;
                }
            }else{
                if(nums[mid] < target && target <= nums[right]){
//[5,6,0,1,2,3,4]
                    left = mid+1;
                }else{
                    right = mid-1;
                }
            }
        }
        return -1;
    }
```

Steps to solve with duplicates

A. Find minimum number index / rotation point index in an array
   a. Initialise left = 0 , right = nums.length-1
   b. Loop till left < right (don't check left ==right, that is the break point for this loop as Left and right will point to Min value)
      i. Loop until nums[left] == nums[right] && left!=right and increment left++ (This takes care of duplicate values)
      ii. Find pivot mid = (left +right)/2

iii. If **pivot** index value **> right**, right-side has the shifting point, left = pivot+1; (as we have to move towards the rotation point). Else search in Left side. Right = pivot

c. Return min value.

```
public int findMin(int[] nums) {
        int left =0;
        int right = nums.length -1;
        int pivot = 0;
        while(left < right){
            //input: [3,3,1,3] will return 3 as answer. to fix that we keep moving the
left pointer if Left and Right has same number
            while(nums[left] == nums[right] && left!=right){
                left++;
            }
            pivot = (left +right)/2;
            if(nums[pivot] > nums[right]){
                left =pivot+1;
            }
            else{
                right = pivot;
            }
        }
        return nums[left];
    }
```

B. Search in Rotated array for target

a. Initialise left =0; right = nums.length-1;

b. Loop until Left <= Right (left == right is included in this case as we are searching for a target value & chances of its presence must be checked till end)

i. Loop until nums[left] == nums[right] && left!=right and increment left++ (This takes care of duplicate values)

ii. Find the pivot = (left+right)/2

iii. If nums[pivot] == target return true;

iv. Determine the rotated side and check the target in two cases

1. Case 1: nums[pivot] >=nums[left] -> shift point is in right side
If nums[left] <= target<nums[mid] → chk target lies in the left side
Right = pivot-1;
Else
Left = pivot+1;

2. Case 2:else -> shift point is in left side
If nums[left] <= target<nums[mid] →chk target lies in the right side
left = pivot+1;

Else

right = pivot-1;

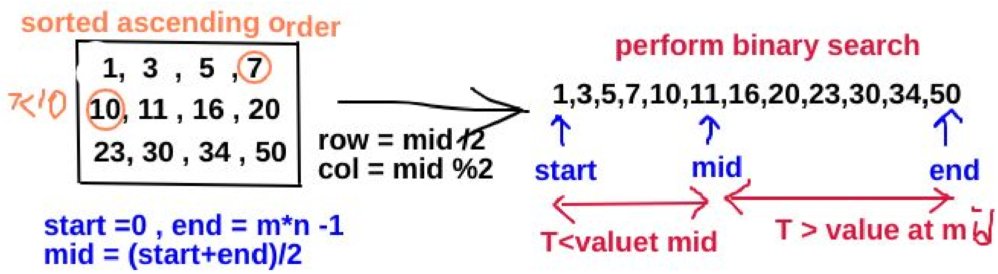
v. Return false

```
public boolean search(int[] nums, int target) {
        int left = 0;
        int right = nums.length-1;
        while(left <= right){
            while(nums[left] == nums[right] && left!=right){
                left++;
            }
            int mid = (left + right)/2;
            if(nums[mid] == target)
                return true;
            //Find the rotation point side
            if(nums[left]<= nums[mid])//right side has the shift point
            {
                if(nums[left]<= target && target < nums[mid])
                    right = mid -1;
                else
                    left = mid+1;
            }
            else{ //left side has the shift point
                if(nums[mid] < target && target <=nums[right])
                    left = mid +1;
                else
                    right = mid-1;
            }
        }
        return false;
    }
```

Search 2D matrix with properties

Matrix properties

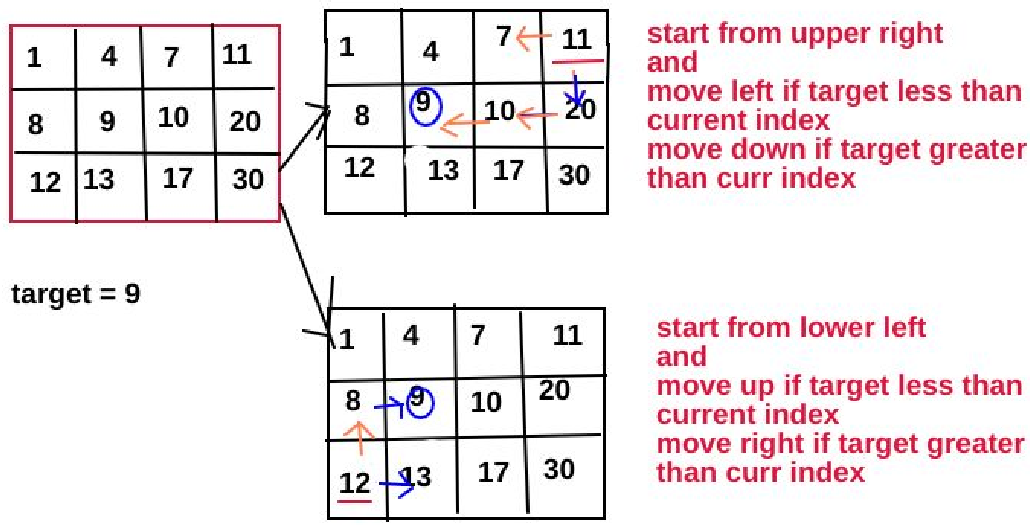
- Integers in each row are sorted from left to right.
- The first integer of each row is greater than the last integer of the previous row.




```
//brute force - o(m*n) if we search for target in the array across
// each item
//o(log(m*n)) - by visually flattening the array to 1D matrix and performing Binary search

public boolean searchMatrix(int[][] matrix, int target) {
   if(matrix==null || matrix.length==0 || matrix[0].length==0)
       return false;

    int m = matrix.length;
    int n = matrix[0].length;

    int start = 0;
    int end = m*n-1;

    while(start<=end){
        int mid=(start+end)/2;
        //the below step helps us to visualise 2D matrix as 1D matrix.
        // we make use of the property that the last element in each row will be
        //greater than the first element in next row
        int midX=mid/n; //go for row
        int midY=mid%n; //go for column

        if(matrix[midX][midY]==target)
            return true;
        // now we perform the binary search
        if(matrix[midX][midY]<target){
            start=mid+1;//go up in value
        }else{
            end=mid-1; //go down in value
        }
    }
    return false;
}
```

```
//brute force - o(m*n) if we search for target in the array across each item
  //o(log(m*n)) - by visually flattening the array to 1D matrix and performing Binary search
```

Search in 2D matrix without properties




```
// o(m+n) complexity
//staircase pattern
public boolean searchMatrix(int[][] matrix, int target) {
    if(matrix==null || matrix.length==0 || matrix[0].length==0)
        return false;
    int m=matrix.length-1; //row
    int n=matrix[0].length-1; //column

    int i=m; //start pointer from lower left element
    int j=0;

    while(i>=0 && j<=n){
        if(target < matrix[i][j]){ //target < matrix[m][0]
            i--; //target lesser than the current element move up
        }else if(target > matrix[i][j]){
            j++; //target greater than current element move right
        }else{
            return true;
        }
    }
    return false;
}
```

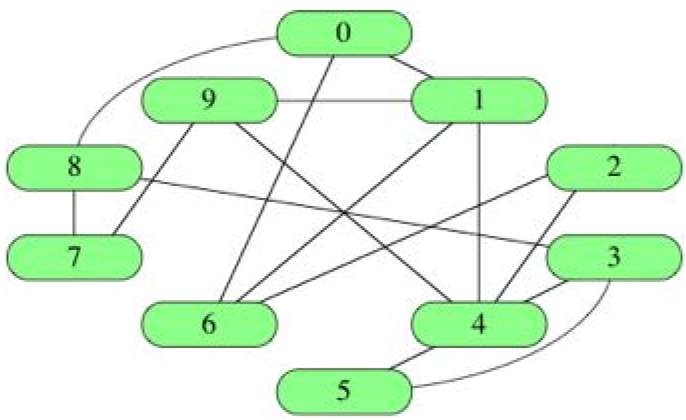
o(m+n) complexity
staircase pattern

Subtree: Maximum Average Node

```
/**
  * @param root the root of binary tree
  * @return the root of the maximum average of subtree
  */

 class ResultType {
    TreeNode node;
    int sum;
    int size;
    public ResultType(TreeNode node, int sum, int size) {
       this.node = node;
       this.sum = sum;
       this.size = size;
    }
 }

 private ResultType result = null;

 public TreeNode findSubtree2(TreeNode root) {
    // Write your code here
    if (root == null) {
       return null;
    }

    ResultType rootResult = helper(root);
    return result.node;
 }

 public ResultType helper(TreeNode root) {
    if (root == null) {
       return new ResultType(null, 0, 0);
    }

    ResultType leftResult = helper(root.left);
    ResultType rightResult = helper(root.right);

    ResultType currResult = new ResultType(
```

```
                root,
                leftResult.sum + rightResult.sum + root.val,
                leftResult.size + rightResult.size + 1);

        if (result == null
           || currResult.sum * result.size > result.sum * currResult.size) {
           result = currResult;
        }

        return currResult;
    }

}
```

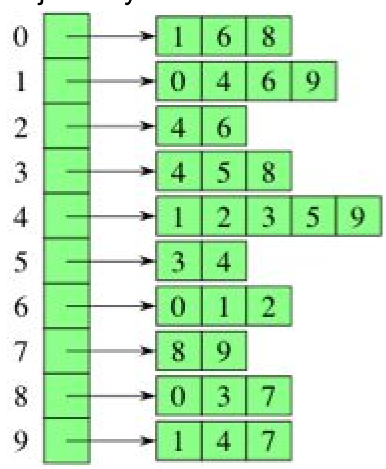
Graph representations:



Graph Representation	Memory/ space to represent graph	Time taken to find an edge in the graph	Time taken to find neighbor of a given vertex

Edge List
`[ [0,1], [0,6], [0,8], [1,4],`
`[1,6], [1,9], [2,4], [2,6],`
`[3,4], [3,5],`
`3,8], [4,5], [4,9], [7,8], [7,9]`
	For E edges
	O(E) space
		O(E) time	O(Ev1*Ev2) time

Adjacency matrix $\begin{array}{c c c c c c c c c c c} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 5 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 6 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 8 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 9 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{array}$ `[0, 1, 0, 0, 0, 0, 1, 0, 1, 0],`	Graph[v][v] V x V matrix - O(V^2)	Constant time O(1) time - Access the edges directly graph[i][j]	O(V) time Even knowing the index of vertex, we have to look for value of all the vertices in the row to see if an edge exist.

1, 0, 0, 0, 1, 0, 1, 0, 0, 1],[0, , 0, 0, 1, 0, 1, 0, 0, 0], [0, 0 , 0, 1, 1, 0, 0, 1, 0], [0, 1, 1 , 0, 1, 0, 0, 0, 1], [0, 0, 0, 1 , 0, 0, 0, 0, 0], [1, 1, 1, 0, 0 , 0, 0, 0, 0], [0, 0, 0, 0, 0, 0 , 0, 1, 1], [1, 0, 0, 1, 0, 0, 0 , 0, 0], [0, 1, 0, 0, 1, 0, 0, 1 , 0]]			
Adjacency List  [[1, 6, 8], [0, 4, 6, 9], [4, 6], [4, 5, 8], [1, 2, 3, 5, 9], [3, 4], [0, 1, 2], [8, 9], [0, 3, 7], [1, 4, 7]]	O(V+E) space	Constant time to lookup ‘i’ in the list and search ‘j’ in ‘i’s’ list. For directed graph O(d) - d is degree of each Vertex in a directed graph	Constant time -O(1) Specify vertex as index to get its adjacency list