MISCELLANEOUS
SET 1

Permutation in String

Given two strings s1 and s2, write a function to return true if s2 contains the permutation of s1. In other words, one of the first string's permutations is the substring of the second string.
Example 1:
Input: s1 = "ab" s2 = "eidbaooo"
Output: True
Explanation: s2 contains one permutation of s1 ("ba").

```
public boolean checkInclusion(String s1, String s2) {
        int n = s1.length();
        int m = s2.length();
        if(n>m)
                return false;
        HashMap<Character, Integer> map = new HashMap<>();
        for(int i=0;i<n;i++){
                char c = s1.charAt(i);
                if(map.containsKey(c)){
                        map.put(c, map.get(c)+1);
                }else
                        map.put(c, 1);
        }
        int i=0;
        while(i<n){
                char c = s2.charAt(i++);
                if(map.containsKey(c)){
                        map.put(c, map.get(c)-1);
                }
        }
        for(i=n;i<=m;i++){
                int flag = 0;
                for(Map.Entry<Character, Integer> obj:map.entrySet()){
                        if(obj.getValue()!=0){
                                flag = 1;
                                break;
                        }
                }
                if(flag==0)
                        return true;
                else if(i==m)
                        break;
                else{
                        char c = s2.charAt(i-n);
```

```
                    if(map.containsKey(c)){
                        map.put(c, map.get(c)+1);
                    }
                    c = s2.charAt(i);
                    if(map.containsKey(c)){
                        map.put(c, map.get(c)-1);
                    }
                }
            }
            return false;
        }
```

Insertion Sort on linked list

```
public ListNode insertionSortList(ListNode head) {
        if(head!=null){
            ListNode curr = head.next,parent = head;
            while(curr!=null){
                ListNode temp = head,par = null;
                while(temp!=curr && temp.val<=curr.val){
                    par = temp;
                    temp = temp.next;
                }
                if(temp!=curr){
                    parent.next = curr.next;
                    if(par==null){
                        curr.next = head;
                        head = curr;
                    }else{
                        curr.next = par.next;
                        par.next = curr;
                    }
                    curr = parent.next;
                }else{
                    parent = curr;
                    curr = curr.next;
                }
            }
        }
        return head;
    }
```

Product of the Last K Numbers

Implement the class `ProductOfNumbers` that supports two methods:

1. `add(int num)`

   - Adds the number `num` to the back of the current list of numbers.

2. `getProduct(int k)`

   - Returns the product of the last `k` numbers in the current list.
   - You can assume that always the current list has **at least** `k` numbers.

At any time, the product of any contiguous sequence of numbers will fit into a single 32-bit integer without overflowing.

Approach : Using Product Prefix array.

```
class ProductOfNumbers {
        List<Integer> list;
        public ProductOfNumbers() {
                list = new ArrayList<>();
                list.add(1);
        }

        public void add(int num) {
                if(num==0){
                                list.clear();
                                list.add(1);
                }else{
                                list.add(list.get(list.size()-1)*num);
                }
        }

        public int getProduct(int k) {
                int pos = list.size() - k - 1;
                if(pos<0)
                                return 0;
                return list.get(list.size()-1) / list.get(pos);
        }
}
```

Unique Binary Search Trees

Given *n*, how many structurally unique **BST's** (binary search trees) that store values 1 ... *n*?

Example:

```
Input: 3
```

```
Output: 5

Explanation:

Given n = 3, there are a total of 5 unique BST's:


   1         3     3      2      1

    \       /     /      / \      \

     3     2     1      1   3      2

    /     /       \                 \

   2     1         2                 3
```

Approach : find catalan of n.
C(0) = C(1) = 1;
C(2) = C(0) * C(1) + C(1) * C(0)
C(3) = C(0) * C(2) + C(1) * C(1) + C(2) * C(0)
C(4) = C(0) * C(3) + C(1) * C(2) + C(2) * C(1) + C(3) * C(0)
…….
……..so on

```
public int numTrees(int n) {
        //calculate catalan(n)
        if(n==0)
                return 1;
        int dp[] = new int[n+1];
        dp[0] = dp[1] = 1;
        for(int i=2;i<=n;i++){
                dp[i] = 0;
                for(int j=0;j<i;j++){
                        dp[i] += dp[j] * dp[i-j-1];
                }
        }
        return dp[n];
}
```

Delete Node in a BST

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

Note: Time complexity should be O(height of tree).

Example:

```
root = [5,3,6,2,4,null,7]

key = 3



    5

   / \

  3   6

 / \   \

2   4   7
```

```
private TreeNode smallCal(TreeNode root){
        if(root.left==null)
                return root;
        return smallCal(root.left);
}
public TreeNode deleteNode(TreeNode root, int key) {
        if(root==null)
                return null;
        TreeNode temp = root,par = null;
        while(temp!=null){
                if(temp.val==key){
                        //leaf node
                        if(temp.left==null && temp.right==null){
                                if(par==null)
                                        return null;
                                else if(par.left == temp)
                                        par.left = null;
                                else
                                        par.right = null;
                                return root;
                        }
                        //two child node
                        if(temp.left!=null && temp.right!=null){
                                TreeNode small = smallCal(temp.right);
```

```
                int t = temp.val;
                temp.val = small.val;
                small.val = t;
                temp.right = deleteNode(temp.right,key);
                return root;
            }
            //one child node
            if(temp.left!=null){
                if(par==null)
                    return temp.left;
                else if(par.left==temp)
                    par.left = temp.left;
                else
                    par.right = temp.left;
                return root;
            }
            if(temp.right!=null){
                if(par==null)
                    return temp.right;
                else if(par.left==temp)
                    par.left = temp.right;
                else
                    par.right = temp.right;
                return root;
            }
        }else{
            par = temp;
            if(key<temp.val)
                temp = temp.left;
            else
                temp = temp.right;
        }
    }
    return root;
}
```

Insertion in AVL tree

We can maintain height of every node to make speed better to calculate height again.

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

```
            return 0;
        return 1 + Math.max(height(root.left), height(root.right));
}
private TreeNode leftRotation(TreeNode x){
        TreeNode y = x.right;
        TreeNode T2 = y.left;

        y.left = x;
        x.right = T2;

        return y;
}
private TreeNode rightRotation(TreeNode x){
        TreeNode y = x.left;
        TreeNode T2 = y.right;

        y.right = x;
        x.left = T2;

        return y;
}
private TreeNode insert(TreeNode root, int key){
        if(root==null){
            return new TreeNode(key);
        }
        if(key<root.val)
            root.left = insert(root.left, key);
        else if(key>root.val)
            root.right = insert(root.right, key);
        else  //duplicate node
            return null;

        //check for balance this node
        int lH = height(root.left);
        int rH = height(root.right);

        int balance = lH-rH;

        if(balance>1){ //left subtree height is more
            if(key<root.left.val){ //left - left case
                return rightRotation(root);
            }else{ //left - right case
```

```
                        root.left = leftRotation(root.left);
                        return rightRotation(root);
                    }
                }else if(balance<1){ //right subtree height is more
                    if(key>root.right.val){ //right - right case
                        return leftRotation(root);
                    }else{ //right - left case
                        root.right = rightRotation(root.right);
                        return leftRotation(root);
                    }
                }

                return root;
        }
        public TreeNode sortedListToBST(ListNode head) {
                TreeNode root = null;
                while(head!=null){
                    root = insert(root, head.val);
                }
                return root;
        }
}
```

Add 1 to a number represented as linked list

A number **N** is represented in Linked List such that each digit corresponds to a node in linked list. You need to add 1 to it.

Example:
Input:
4
456
123
999
1879
Output:
457
124
1000
1880

```
private static int cal(Node head){
        if(head.next==null){
```

```
            if(head.data+1==10){
                head.data = 0;
                return 1;
            }else{
                head.data += 1;
                return 0;
            }
        }
        int val = cal(head.next);
        if(val==1){
            if(head.data+1==10){
                head.data = 0;
                return 1;
            }else{
                head.data += 1;
            }
        }
        return 0;
}
public static Node addOne(Node head) {
        if(head==null)
            return head;
        int val = cal(head);
        if(val==1){
            Node node = new Node(1);
            node.next = head;
            head = node;
        }
        return head;
}
```

M-Coloring Problem

Given an undirected graph and an integer **M**. The task is to determine if the graph can be colored with at most C colors such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices. Print 1 if it is possible to color vertices and 0 otherwise.

```
class solve {
        static boolean isSafe(List<Integer>[] G, int color[], int s, int c){
        List<Integer> l = G[s];
        for(int i=0;i<l.size();i++){
            int a = l.get(i);
```

```
            if(color[a]==c)
                return false;
            }
            return true;
        }
        // initial value s = 0 and color[i] = 0
        public static boolean graphColoring(List<Integer>[] G, int[] color, int s, int C) {
            if(s==G.length){
                return true;
            }
            for(int i=1;i<=C;i++){
                if(isSafe(G, color, s, i)){
                    color[s] = i;
                    if(graphColoring(G, color, s+1, C))
                        return true;
                    color[s] = 0;  //Backtrack
                }
            }
            return false;
        }
}
```

Prerequisite Tasks

There are a total of N tasks, labeled from 0 to N-1. Some tasks may have prerequisites, for example to do task 0 you have to first complete task 1, which is expressed as a pair: [0, 1]

Given the total number of tasks and a list of prerequisite pairs, find if it is possible to finish all tasks.

Note: If there is any **cycle** exist in graph, then no topological sort can be find i.e. impossible to complete all tasks. If order of tasks has to print, then print **topological sort** of graph.

```
class Solution {
    boolean isCycle(int s, ArrayList<ArrayList<Integer>> graph, boolean vis[],
boolean temp[]){
            if(temp[s])
                return true;
            if(vis[s])
                return false;
            vis[s] = true;
            temp[s] = true;
            ArrayList<Integer> l = graph.get(s);
            for(int i=0;i<l.size();i++){
                if(isCycle(l.get(i), list, vis, temp))
```

```
                return true;
            }
            temp[s] = false;
            return false;
        }
        public boolean canFinish(int n, int[][] pre){
            ArrayList<ArrayList<Integer>> graph = new ArrayList<>();
            for(int i=0;i<n;i++){
                graph.add(new ArrayList<>());
            }
            int m = pre.length;
            for(int i=0;i<m;i++){
                graph.get(pre[i][1]).add(pre[i][0]);
            }
            boolean vis[] = new boolean[n];
            boolean temp[] = new boolean[n];
            for(int i=0;i<n;i++){
                if(!vis[i]){
                    if(isCycle(i, graph, vis, temp))
                        return false;
                }
            }
            return true;
        }
}
```

Word Search

Given a 2D board of letters and a word. Check if the word exists in the board. The word can be constructed from letters of adjacent cells only. ie - horizontal or vertical neighbors. The same letter cell can not be used more than once.

```
class Solution {
    boolean dfs(int x,int y,int n,int m,char board[][], boolean vis[][], String word, int
l){
            if(l==word.length()){
                return true;
            }
            vis[x][y] = true;
            int row[] = {-1,1,0,0};
            int col[] = {0,0,-1,1};
            for(int k=0;k<4;k++){
                int i = x + row[k];
```

```
                int j = y + col[k];
                if(i>=0 && j>=0 && i<n && j<m && !vis[i][j] &&
word.charAt(l)==board[i][j]){
                        if(dfs(i,j,n,m,board,vis,word,l+1))
                                return true;
                }
        }
        vis[x][y] = false; //Backtrack
        return false;
    }
    public int wordSearch(char[][] board, String word) {
        int n = board.length;
        int m = board[0].length;
        boolean vis[][] = new boolean[n][m];
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(word.charAt(0)==board[i][j]){
                    if(dfs(i,j,n,m,board,vis,word,1))
                        return 1;
                }
            }
        }
        return 0;
    }
}
```