1. Search a 2D matrix (74)

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        for (int i = 0; i < matrix.length; i++){
            for (int j = 0; j < matrix[i].length; j++){
                if (matrix[i][j] == target)
                    return true;
                if (matrix[i][j] > target)
                    return false;
            }
        }
        return false;
    }
}
```

2. Valid parentheses (20)

```
import java.util.*;
public class Solution {
    public boolean isValid(String s) {
        if (s.length() == 0)  return true;
        if (s.length() == 1)  return false;
        Stack<Character> st = new Stack<Character>();
        for (int i = 0; i < s.length(); i++){
            if (s.charAt(i) == '(' || s.charAt(i) == '[' || s.charAt(i) == '{')
                st.push(s.charAt(i));
            else if (s.charAt(i) == ')' && !st.empty() && st.peek() == '(')
                st.pop();
            else if (s.charAt(i) == '}' && !st.empty() && st.peek() == '{')
                st.pop();
            else if (s.charAt(i) == ']' && !st.empty() && st.peek() == '[')
                st.pop();
            else
                return false;
        }
        if (st.empty())
            return true;
        else
            return false;
    }
}
```

3. Merge Two Sorted Lists (21)

```
import java.util.*;
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if (l1 == null && l2 == null) return l1;
        if (l1 == null && l2 != null) return l2;
        if (l1 != null && l2 == null) return l1;
        ListNode result = new ListNode(0);
        ListNode current = result;
        while (l1 != null && l2 != null){
            if (l1.val < l2.val){
                current.next = l1;
                l1 = l1.next;
                current = current.next;
            }
            else{
                current.next = l2;
                l2 = l2.next;
                current = current.next;
            }
        }
        current.next = (l1 == null ? l2 : l1);
        return result.next;
    }
}
```

4. Rectangle Overlap

```
import java.util.*;
class Node
{
    int x;
    int y;
    public Node(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
public class Solution {
    // A & C is bottom-left, B & D is top-right
    public static boolean hasOverlap(Node A, Node B, Node C, Node D){
        if (B.x <= C.x || A.x >= D.x || A.y >= D.y || C.y >= B.y)
            return false;
        else
            return true;
    }
}
```

5. Sliding Window Maximum 239 /Minimum

```
import java.util.*;
public class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        if (nums == null || nums.length < 2) return nums;
        int[] result = new int[nums.length - k + 1];
        for (int i = 0; i < nums.length - k + 1; i++){
            int window = 0;
            int max = nums[i];
            while (window < k){
                if (max < nums[i+window])
                    max = nums[i+window];
                window++;
            }
            result[i] = max;
        }
        return result;
    }
}
```

6. Search a 2D Matrix II (efficient)

```
public class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0) return false;
        int m = matrix.length, n = matrix[0].length;
        int i = 0, j = n - 1;
        while(i < m && j >= 0)
        {
            if(matrix[i][j] == target) return true;
            else if(matrix[i][j] > target) j--;
            else i++;
        }
        return false;
    }
}
```

7. Gray Code

```
public class Solution {
    public static boolean isConsecutive(byte a, byte b)
    {
        byte c = (byte)(a ^ b);
        int count = 0;
        while(c != 0)
        {
            c &= (c - 1);
            count++;
        }
        return count == 1;
    }
    public static void main(String[] args) {
        byte a = 0x31, b = 0x33;
        System.out.println(isConsecutive(a, b));
    }
}
```

8. Rotate String

```
import java.util.*;
public class Solution{
    public static boolean isRotated(String s, String t){
        if (s == null && t == null) return true;
        else if (s == null || t = null) return false;
        else return (s.length() == t.length()) && ((s + s).indexOf(t) != -1);
    }
}
```

9. Remove Vowel

```
import java.util.*;
public class Solution{
    public static String removeVowel(String s){
        StringBuilder result = new StringBuilder();
        String t = "aeiouAEIOU";
        for (int i = 0; i < s.length(); i++){
            if (t.indexOf(s.charAt(i)) < 0)
                result.append(s.charAt(i));
        }
        return result.toString();
    }
}
```

10. Find Optimal Weights (Close Two Sum)

```
import java.util.*;
class Container {
    public double first;
    public double second;
    public Container(double first, double second)
    {
        this.first = first;
        this.second = second;
    }
}
public class Solution {
    public static Container findOptimalWeights(double capacity, double[] weights){
        if (weights == null || weights.length < 2) return null;
        Arrays.sort(weights);
        if(weights[0] + weights[1] > capacity) return null;
        Container result = new Container(weights[0], weights[1]);
        for (int i = 0; i < weights.length; i++){
            for (int j = i+1; j < weights.length; j++){
                if ((weights[i] + weights[j]) <= capacity && (weights[i] + weights[j]) > (result.first
+ result.second)){
                    result.first = weights[i];
                    result.second = weights[j];
                }
                if ((weights[i] + weights[j]) > capacity)
                    break;
            }
        }
        return result;
    }
    public static void main(String[] args)
    {
        Container res = findOptimalWeights(35, new double[]{10, 24, 30, 9, 19, 23, 7});
        System.out.println(res.first+" "+res.second);
    }
}
```

11. Reverse Second Half of Linked List

```
public static ListNode reverseSecondHalfList(ListNode head) {
    if (head == null || head.next == null)      return head;
    ListNode fast = head;
    ListNode slow = head;
    while (fast.next != null && fast.next.next != null) {
        fast = fast.next.next;
        slow = slow.next;
    }
    ListNode pre = slow.next;
    ListNode cur = pre.next;
    while (cur != null) {
        pre.next = cur.next;
        cur.next = slow.next;
        slow.next = cur;
        cur = pre.next;
    }
    return head;
}
```

12. GCD Greatest Common Divisor

```
public class solution {
    public static int GCD(int[] input)
    {
        if(input.length == 1) return input[0];
        int res = input[0];
        for(int i = 1; i < input.length; i++)
        {
            res = helper(res, input[i]);
        }
        return res;
    }
    private static int helper(int a, int b)
    {
        if(b == 0) return a;
        return helper(b, a%b);
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int[] input = {9,6,12,24};
        System.out.println(GCD(input));
    }

}
```

13. Same Tree (100)

```
import java.util.*;
public class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) return true;
        if ((p != null && q == null) || (p == null && q != null)) return false;
        return (p.val == q.val) && isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}
```

14. Subtree Check

```
import java.util.*;
public class Solution {
    public boolean isSubTree(TreeNode a, TreeNode b){
        if ((a == null && b == null) || (b == null && a != null)) return true;
        if (a == null && b != null) return false;
        return isSameTree(a,b) || isSameTree(a.left, b) || isSameTree(a.right, b);

    }
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) return true;
        if ((p != null && q == null) || (p == null && q != null)) return false;
        return (p.val == q.val) && isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}
```

15. K Closest Points

```
import java.util.*;
class Point
{
    double  x;
    double y;
    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
    }
}
public class Solution {
    private static double distance(Point a, Point b)
    {
        return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
    }

    public static Point[] closestPoint(Point[] array, final Point origin, int k)
    {
        if(k > array.length) return array;
        Point[] res = new Point[k];
        Arrays.sort(array, new Comparator<Point>()
        {
            @Override
            public int compare(Point a, Point b)
            {
                return Double.compare(distance(a, origin), distance(b, origin));
            }

        });
        for(int i = 0; i < k; i++) res[i] = array[i];
        return res;
    }

}
```

16. Two Sum Count

```
import java.util.*;

public class Solution {
    public static int[] twoSum(int[] nums, int target){
        if (nums == null) return null;
        Arrays.sort(nums);
        int[] result = new int[2];
        int start = 0, end = nums.length - 1;
        while (start <= end){
            if (nums[start] + nums[end] == target){
                result[0] = nums[start];
                result[1] = nums[end];
                return result;
            }
            else if (nums[start] + nums[end] > target)
                end--;
            else
                start++;
        }
        return null;
    }
}
```

17. Window Sum

```
import java.util.*;
public class Solution{
    public static List<Integer> windowSum(List<Integer> input, int k){
        List<Integer> result = new ArrayList<>();
        if (input == null || input.size() == 0 || k <= 0) return result;
        for (int i = 0; i < input.size() - k + 1; i++){
            int window = 0;
            int sum = 0;
            while (window < k){
                sum += input.get(i+window);
                window++;
            }
            result.add(sum);
        }
        return result;
    }

    public static void main(String[] args) {
        List<Integer> input = new ArrayList<>();
        input.addAll(Arrays.asList(2,3,4,2,5,7,8,9,6));
        List<Integer> output = windowSum(input, 4);
        for(int i: output) System.out.print(i + " ");
    }
}
```

18. Tree Amplitude

```
import java.util.*;
class TreeNode
{
   public TreeNode left, right;
   public int  val;
   public TreeNode(String val)
   {
      this.val = Integer.parseInt(val);
   }
}
public class Solution {
   public static int amplitude(TreeNode root){
      if (root == null) return 0;
      int max = root.val, min = root.val;
      return helper(root, max, min);
   }
   private static int helper(TreeNode root, int max, int min){
      if (root == null) return max - min;
      if (root.val < min) min = root.val;
      if (root.val > max) max = root.val;
      return Math.max(helper(root.left, max, min), helper(root.right, max, min));
   }
}
```

19. Arithmetic Sequence (413)

(A sequence of number is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.)

```
import java.util.*
public class Solution {
    public int numberOfArithmeticSlices(int[] A) {
        if (A == null || A.length < 2) return 0;
        int diff = 0;
        int result = 0;
        for (int i = 0; i < A.length - 2; i++){
            int count = 0;
            diff = A[i+1] - A[i];
            int current = i+1;
            while(current < A.length-1){
                if (A[current+1]-A[current] == diff){
                    count ++;
                    current ++;
                }
                else{
                    break;
                }
            }
            result += count;
        }
        return result > 1000000000 ? -1: result;
    }
}
```

20. BST Minimum Path Sum

```
import java.util.*;
public class Solution {
    public static int minPathSum(TreeNode root){
        if (root == null)
            return 0;
        else if (root.left == null && root.right != null)
            return root.val + minPathSum(root.right);
        else if (root.left != null && root.right == null)
            return root.val + minPathSum(root.left);
        else
            return root.val + Math.min(minPathSum(root.left), minPathSum(root.right));
    }
}
```

21. Day Change (Cell Growth)

```
import java.util.*;
public class Solution {
    public static int[] dayChange(int[] input, int day){
        if(input == null || input.length == 0 || day <= 0) return input;
        for (int i = 0; i < day; i++){
            int pre = input[0];
            if (input[1] == 0) input[0] = 0;
            else input[0] = 1;
            for (int j = 1; j < input.length - 1; j ++){
                if (pre == input[j+1]){
                    pre = input[j];
                    input[j] = 0;
                }
                else{
                    pre = input[j];
                    input[j] = 1;
                }
            }
            if (pre == 0) input[input.length-1] = 0;
            else input[input.length-1] = 1;
        }
        return input;
    }
    public static void main(String[] args) {
        int[] input = {1,0,0,0,0,1,0,0};
        int[] output = dayChange(input, 4);
        for(int i: output) System.out.print(i);
    }
}
```

22. Insert Into Cycle Linked List

```
import java.util.*;
public class Solution {
    public static LsitNode insertIntoCycleLinkedList(ListNode head, int val){
        ListNode newNode = new ListNode(val);
        if (head == null){
            newNode.next = newNode;
            return newNode;
        }

        ListNode current = head;

        do{
            if (current.val <= val && current.next.val >= val){
                newNode.next = current.next;
                current.next = newNode;
                break;
            }
            if (cur.val > cur.next.val && (val <= cur.next.val || val >= cur.val))){
                newNode.next = current.next;
                current.next = newNode;
                break;
            }
            current = current.next;
        } while (current != head)
        return newNode;
    }
}
```

23. Loop in Linked List (142)

```
import java.util.*;
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if (head == null) return null;
        ListNode fast = head, slow = head;
        while (fast.next != null && fast.next.next != null){
            slow = slow.next;
            fast = fast.next.next;
            if (fast == slow){
                while (head != slow){
                    head = head.next;
                    slow = slow.next;
                }
                return slow;
            }
        }
        return null;
    }
}
```

24. LRU Cache Count Miss

```
public class Solution {
    public static int countMiss(int[] input, int size) {
        if (input == null) return 0;
        int miss = 0;
        List<Integer> cache = new LinkedList<Integer>();
        for (int i = 0; i < input.length; i++){
            if (cache.contains(input[i])){
                cache.remove(new Integer(i));
                cache.add(input[i]);
            }
            else if (cache.size() >= size){
                cache.remove(0);
                cache.add(input[i]);
                miss++;
            }
            else{
                cache.add(input[i]);
                miss++;
            }
        }
        return miss;
    }
}
```

25. Round Robin

```
import java.util.*;
class process{
    int arrTime;
    int exeTime;
    process(int arrTime, int exeTime){
        this.arrTime = arrTime;
        this.exeTime = exeTime;
    }
}
public class Solution {
    public static double roundRobin(int[] Atime, int[] Etime, int q){
        if (Atime == null || Etime == null || Atime.length != Etime.length) return 0;
        int length = Atime.length;
        Queue<process> queue = new LinkedList<process>();
        int curTime = 0, waitTime = 0, index = 0;
        while (!queue.isEmpty() || index < length) {
            if (!queue.isEmpty()) {
                process cur = queue.poll();
                waitTime += curTime - cur.arrTime;
                curTime += Math.min(cur.exeTime, q);
                for (; index < length && Atime[index] <= curTime; index++)
                    queue.add(new process(Atime[index], Etime[index]));
                if (cur.exeTime > q)
                    queue.add(new process(curTime, cur.exeTime - q));
            }
            else {
                queue.add(new process(Atime[index], Etime[index]));
                curTime = Atime[index++];
            }
        }
        return (float) waitTime / length;
    }
    public static void main(String[] args) {
        int[] arriveTime = {0, 1, 3, 9};
        int[] runTime = {2, 1, 7, 5};
        System.out.println(roundRobin(arriveTime, runTime, 2));
    }
}
```

26. Rotate Matrix

```
import java.util.*;
public class Solution{
    public static int[][] rotate(int[][] matrix, int flag){
        if (matrix == null || matrix.length == 0) return null;
        int [][] transposed = transpose(matrix);
        if (flag == 1){
            for (int i = 0; i < transposed.length; i++){
                int begin = 0, end = transposed[i].length - 1;
                while (begin < end){
                    int temp = transposed[i][begin];
                    transposed[i][begin] = transposed[i][end];
                    transposed[i][end] = temp;
                    begin ++;
                    end --;
                }
            }
        }
        if (flag == 0){
            for (int i = 0; i < transposed[0].length; i++){
                int begin = 0, end = transposed.length - 1;
                while (begin < end){
                    int temp = transposed[begin][i];
                    transposed[begin][i] = transposed[end][i];
                    transposed[end][i] = temp;
                    begin ++;
                    end --;
                }
            }
        }
        return transposed;
    }
    private static int[][] transpose(int[][] matrix){
        int[][] transposed = new int[matrix[0].length][matrix.length];
        for (int i = 0; i < matrix.length; i++){
            for (int j = 0; j < matrix[i].length; j++){
                transposed[j][i] = matrix[i][j];
            }
        }
        return transposed;
    }
}
```

27. Shortest Job First

```
import java.util.*;
public class Solution {
public static double SJF(int[] req, int[] dur)
  {
    if(req == null || req.length == 0) return 0;
    PriorityQueue<Process> queue = new PriorityQueue<>(new Comparator<Process>()
    {
      @Override
      public int compare(Process a, Process b)
      {
        if(a.exeTime == b.exeTime) return a.arrTime - b.arrTime;
        else return a.exeTime - b.exeTime;
      }
    });
    int t = 0, sum = 0, i = 0;
    while(i < req.length || !queue.isEmpty())
    {
      if(queue.isEmpty())
      {
        queue.offer(new Process(req[i], dur[i]));
        t = req[i];
        i++;
      }
      else
      {
        Process p = queue.poll();
        sum += (t - p.arrTime);
        t += p.exeTime;
        while(i < req.length && req[i] <= t)
        {
          queue.offer(new Process(req[i], dur[i]));
          i++;
        }
      }
    }
    return (sum + 0.0) / req.length;
  }
}
```

28. Maze

```
import java.util.*;
public class Solution {
    static final int[][] dir = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    public static int checkMaze(int[][] maze)
    {
        if(maze == null || maze.length == 0 || maze[0][0] == 0) return 0;
        if(maze[0][0] == 9) return 1;
        int m = maze.length, n = maze[0].length;
        Queue<int[]> queue = new LinkedList<>();
        queue.offer(new int[]{0, 0});
        maze[0][0] = 0;
        while(!queue.isEmpty())
        {
            int[] p = queue.poll();
            for(int k = 0; k < 4; k++)
            {
                int x = p[0] + dir[k][0];
                int y = p[1] + dir[k][1];
                if(x >=0 && x < m && y >= 0&& y < n)
                {
                    if(maze[x][y] == 9) return 1;
                    else if(maze[x][y] == 1)
                    {
                        queue.offer(new int[]{x, y});
                        maze[x][y] = 0;
                    }
                }
            }
        }
        return 0;
    }
}
```

29. Four Integer

```
public static int[] fourInteger(int A, int B, int C, int D)
{
    int[] nums = new int[]{A, B, C, D};
    Arrays.sort(nums);
    swap(nums, 0, 1);
    swap(nums, 2, 3);
    swap(nums, 0, 3);
    return nums;
}
private static void swap(int[] nums, int i, int j)
{
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}
```

30. Copy List with Random Pointer (138)

```
public RandomListNode copyRandomList(RandomListNode head) {
  if (head == null) return null;
  Map<RandomListNode, RandomListNode> map = new HashMap<RandomListNode,
RandomListNode>();

  RandomListNode node = head;
  while (node != null) {
    map.put(node, new RandomListNode(node.label));
    node = node.next;
  }

node = head;
  while (node != null) {
    map.get(node).next = map.get(node.next);
    map.get(node).random = map.get(node.random);
    node = node.next;
  }

  return map.get(head);
}
```

31. Order Dependency

```
import java.util.*;
class Order{
    String orderName;
    public Order(String orderName)
    {
        this.orderName = orderName;
    }
}
class OrderDependency{
    Order order;
    Order dependent;
    public OrderDependency(Order order, Order dependent)
    {
        this.order = order;
        this.dependent = dependent;
    }
}
public class Solution {
    public static List<Order> findOrder(List<OrderDependency> depenency)
    {
        Map<String, Integer> inmap = new HashMap<>();
        Map<String, List<String>> outmap = new HashMap<>();
        for(OrderDependency i: depenency)
        {
            if(!inmap.containsKey(i.dependent.orderName)) inmap.put(i.dependent.orderName, 0);
            if(!inmap.containsKey(i.order.orderName)) inmap.put(i.order.orderName, 0);
            inmap.put(i.order.orderName, inmap.get(i.order.orderName) + 1);
            if(!outmap.containsKey(i.dependent.orderName)) outmap.put(i.dependent.orderName,
new ArrayList<String>());
            outmap.get(i.dependent.orderName).add(i.order.orderName);
        }
        List<Order> res = new ArrayList<>();
        Queue<String> queue = new LinkedList<>();
        for(String i: inmap.keySet())
            if(inmap.get(i) == 0) queue.offer(i);
        while(!queue.isEmpty())
        {
            String s = queue.poll();
            res.add(new Order(s));
            if(outmap.containsKey(s))
            {
                for(String o: outmap.get(s))
                {
                    inmap.put(o, inmap.get(o) - 1);
                    if(inmap.get(o) == 0) queue.offer(o);
                }
            }
            outmap.remove(s);
        }
        return res;
    }
    public static void main(String[] args)
    {
        List<OrderDependency> input = new ArrayList<>();
        input.add(new OrderDependency(new Order("A"), new Order("E")));
        input.add(new OrderDependency(new Order("D"), new Order("E")));
        input.add(new OrderDependency(new Order("A"), new Order("C")));
        input.add(new OrderDependency(new Order("B"), new Order("D")));
        List<Order> output = findOrder(input);
        for(Order i: output) System.out.print(i.orderName + " ");
    }
}
```

32. Maximum Minimum Path

```
public static int find(int[][] input)
    {
        int m = input.length, n = input[0].length;
        for(int i = 0; i < m; i++)
        {
            for(int j = 0; j < n; j++)
            {
                if(i == 0 && j == 0) continue;
                int a = Integer.MIN_VALUE, b = Integer.MIN_VALUE;
                if(i - 1 >= 0) a = Math.min(input[i][j], input[i - 1][j]);
                if(j - 1 >= 0) b = Math.min(input[i][j], input[i][j - 1]);
                input[i][j] = Math.max(a, b);
            }
        }
        return input[m - 1][n - 1];
    }
```

33. Minimum Spanning Tree

```java
import java.util.*;
class Connection{
    String node1;
    String node2;
    int cost;
    public Connection(String a, String b, int c){
        node1 = a;
        node2 = b;
        cost = c;
    }
}
public class Solution {
    static class DisjointSet
    {
        Set<String> set;
        Map<String, String> map;
        int count;
        public DisjointSet()
        {
            count = 0;
            set = new HashSet<>();
            map = new HashMap<>();
        }
        public void MakeSet(String s)
        {
            if(!set.contains(s))
            {
                count++;
                set.add(s);
                map.put(s, s);
            }
        }
        public String Find(String s)
        {
            if(!set.contains(s)) return null;
            if(s.equals(map.get(s))) return s;
            String root = this.Find(map.get(s));
            map.put(s, root);
            return root;
        }
        public void Union(String s, String t)
        {
            if(!set.contains(s) || !set.contains(t)) return;
            if(s.equals(t)) return;
```

```
      count--;
      map.put(s, t);
   }
}
static class ConnectionComparator1 implements Comparator<Connection>
{
   @Override
   public int compare(Connection a, Connection b)
   {
      return a.cost - b.cost;
   }
}
static class ConnectionComparator2 implements Comparator<Connection>
{
   @Override
   public int compare(Connection a, Connection b)
   {
      if(a.node1.equals(b.node1)) return a.node2.compareTo(b.node2);
      else return a.node1.compareTo(b.node1);
   }
}
public static List<Connection> getMST(List<Connection> connections)
{

   Comparator<Connection> comparator1 = new ConnectionComparator1();
   Comparator<Connection> comparator2 = new ConnectionComparator2();
   Collections.sort(connections, comparator1);
   DisjointSet set = new DisjointSet();
   List<Connection> res = new ArrayList<>();
   for(Connection itr: connections)
   {
      set.MakeSet(itr.node1);
      set.MakeSet(itr.node2);
   }
   for(Connection itr: connections)
   {
      String s = set.Find(itr.node1);
      String t = set.Find(itr.node2);
      if(!s.equals(t))
      {
         set.Union(s, t);
         res.add(itr);
         if(set.count == 1) break;
      }
   }
   if(set.count == 1)
```

```
    {
      Collections.sort(res, comparator2);
      return res;
    }
    else return new ArrayList<Connection>();
  }
}
```

34. Maximum Subtree of Average

```java
class Node {
    int val;
    ArrayList<Node> children;
    public Node(int val){
        this.val = val;
        children = new ArrayList<Node>();
    }
}
public class Solution {
    static class SumCount
    {
        int sum;
        int count;
        public SumCount(int sum, int count)
        {
            this.sum = sum;
            this.count = count;
        }
    }
    static Node ans;
    static double max = 0;
    public static Node find(Node root)
    {
        ans = null;
        max = 0;
        DFS(root);
        return ans;
    }
    private static SumCount DFS(Node root)
    {
        if(root == null) return new SumCount(0, 0);
        if(root.children == null || root.children.size() == 0) return new SumCount(root.val, 1);
        int sum = root.val;
        int count = 1;
        for(Node itr: root.children)
        {
            SumCount sc = DFS(itr);
            sum += sc.sum;
            count += sc.count;
        }
        if(count > 1 && (sum + 0.0) / count > max)
        {
            max = (sum + 0.0) / count;
            ans = root;
```

```
              }
              return new SumCount(sum, count);
          }
}
```

35. Five Scores

```
import java.util.*;
class Result{
    int id;
    int value;
    public Result(int id, int value){
        this.id = id;
        this.value = value;
    }
}
public class Solution {
    public static Map<Integer, Double> getHighFive(Result[] results)
    {
        Map<Integer, PriorityQueue<Integer>> map = new HashMap<>();
        for(Result itr: results)
        {
            if(!map.containsKey(itr.id))
            {
                map.put(itr.id, new PriorityQueue<Integer>());
                map.get(itr.id).offer(itr.value);
            }
            else
            {
                if(map.get(itr.id).size() < 5) map.get(itr.id).offer(itr.value);
                else if(itr.value > map.get(itr.id).peek())
                {
                    map.get(itr.id).poll();
                    map.get(itr.id).offer(itr.value);
                }
            }
        }
        Map<Integer, Double> res = new HashMap<>();
        for(int id: map.keySet())
        {
            int sum = 0;
            PriorityQueue<Integer> q = map.get(id);
            while(!q.isEmpty()) sum += q.poll();
            res.put(id, (sum + 0.0) / 5);
        }
        return res;}}
```

36. Longest Palindromic Substring (5)

```
public class Solution {
    int max, start, end;
    public String longestPalindrome(String s) {
        max = 0;
        start = 0;
        end = 0;
        if(s == null || s.length() < 2) return s;
        for(int i = 0; i < s.length(); i++)
        {
            findPalindrome(s, i, i);
            if(i + 1 < s.length() && s.charAt(i) == s.charAt(i + 1)) findPalindrome(s, i, i + 1);
        }
        return s.substring(start, end + 1);
    }
    private void findPalindrome(String s, int l, int r)
    {
        while(l - 1 >= 0 && r + 1 < s.length() && s.charAt(l - 1) == s.charAt(r + 1))
        {
            l--;
            r++;
        }
        if(r - l + 1 > max)
        {
            max = r - l + 1;
            start = l;
            end = r;
        }
    }
}
```

https://segmentfault.com/a/1190000007519772
http://wdxtub.com/interview/14520850399861.html
http://www.amethlex.com/archives/117
http://www.amethlex.com/archives/139
http://www.amethlex.com/archives/150