Search a 2D Matrix

```
public static boolean searchMatrix(int[][] matrix, int target) {
        if(matrix == null || matrix.length == 0) return false;
        int m = matrix.length, n = matrix[0].length;
        int left = 0, right = m * n - 1;
        while(left <= right)
        {
            int middle = left + (right - left) / 2;
            int i = middle / n;
            int j = middle % n;
            if(matrix[i][j] == target) return true;
            else if(matrix[i][j] > target) right = middle - 1;
            else left = middle + 1;
        }
        return false;
}
```

Overlap Rectangle

```
    // A & C is bottom-left, B & D is top-right
        public static boolean hasOverlap(Node A, Node B, Node C, Node D)
    {
        int bottomleft1_x = Math.min(A.x, B.x);
        int bottomleft1_y = Math.min(A.y, B.y);
        int topright1_x = Math.max(A.x, B.x);
        int topright1_y = Math.max(A.y, B.y);

        int bottomleft2_x = Math.min(C.x, D.x);
        int bottomleft2_y = Math.min(C.y, D.y);
        int topright2_x = Math.max(C.x, D.x);
        int topright2_y = Math.max(C.y, D.y);

        if(bottomleft1_x >= topright2_x ||bottomleft2_x >= topright1_x || bottomleft1_y >=
topright2_y || bottomleft2_y >= topright1_y) return false;
        return true;
    }
```

Search a 2D Matrix II

```
    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;
    }
```

GCD

```
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);
     }
```

K Closest Points

```
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;
    }
```

Window Sum

```
public static List<Integer> windowSum(List<Integer> input, int k) {
        List<Integer> res = new ArrayList<>();
        if(input == null || input.size() == 0 || k <= 0) return res;
        if(k >= input.size()){
            int sum = 0;
            for(int i: input) sum += i;
            res.add(sum);
            return res;
        } else {
            int sum = 0;
            for(int i = 0; i < input.size(); i++){
                sum += input.get(i);
                if(i >= k) sum -= input.get(i - k);
                if(i >= k - 1) res.add(sum);
            }
        }
        return res;
    }
```

Arithmetic Sequence

```
public static int count(int[] input) {
        if(input == null || input.length < 3) return 0;
        int sum = 0, diff = Integer.MAX_VALUE, i = 0;
        for(int j = 1; j < input.length; j++)
        {
            int curdiff = input[j] - input[j - 1];
            if(curdiff == diff) sum += (j - i - 1);
            else { i = j - 1; diff = curdiff; }
        }
        return sum > 1000000000 ? -1: sum;
    }
```

Day change

```
public static int[] dayChange(int[] input, int day) {
```

```
        if(input == null || input.length == 0 || day <= 0) return input;
        for(int k = 0; k < day; k++){
            for(int i = 0; i < input.length; i++){
                if(i - 1 >= 0 && i + 1 < input.length && (input[i - 1] & 1) != (input[i + 1] &
1)) input[i] += 2;
            }
            for(int i = 0; i < input.length; i++) input[i] >>= 1;
        }
        return input;
    }
```

Five Scores

```
class Result{
    int id, 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; }}
```

insert value into a circle linked-list

```
public static CNode insert7(CNode myList, int n) {


           if (myList == null) return new CNode(n);

           } else if (myList.next == myList) {

                       myList.next = new CNode(n);

                       myList.next.next = myList;

                       if (myList.val < n) return myList;

                       else return myList.next;

           } else if (n < myList.val) {

                CNode cur = myList;
```

```
            while (cur.next != myList) cur = cur.next;
            cur.next = new CNode(n);
            cur.next.next = myList;
            return cur.next; }
        CNode cur = myList;
        while (cur.next != myList && cur.next.val <= n) cur = cur.next;
        CNode curNext = cur.next;
        cur.next = new CNode(n);
        cur.next.next = curNext;
        return myList; }
```

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;
```

```
    }
}
```

Minimum Spanning Tree

```
import java.util.*;
class Connection{
    String node1, 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>();
    }
}
```

Maximum Subtree of Average

```
import java.util.ArrayList;
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, 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, 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);
    }
}
```

```java
import java.util.Stack;

public class Baseball {
        public static double cal(String[] input) {
                Stack<Double> stack = new Stack<>();
                double res = 0;
                for (String s : input) {
                        if (s.charAt(0) == 'X') {
                                if (!stack.isEmpty()) res += stack.peek() + stack.peek();
                        } else if (s.charAt(0) == 'Z') {
                                if (!stack.isEmpty()) res -= stack.peek();
                        } else if (s.charAt(0) == '+') {
                                if (stack.isEmpty()) continue;
                                double d1 = stack.pop();
                                if (stack.isEmpty()) {
                                        stack.push(d1);
                                        continue;
                                }
                                double d2 = stack.pop();
                                res += d1 + d2;
                                stack.push(d2);
                                stack.push(d1);
                                stack.push(d1 + d2);
                        } else {
                                double d = Double.parseDouble(s);
                                stack.push(d);
                        }
                }
                return res;
        }
}
```