1. Convert a CDLL to a Balanced BT

```
Public Node transfer(Node head){
        If (head == null){
                Return null;
        }
        int count = 0;
        Node cur = head;
        Map<Integer, Node> map = new HashMap();
        While (cur != head){
                Map.put(count++, cur);
                Cur = cur.right;
        }
        count--;
        Node tail = head.left;
        Tail.right = null;
        Head.left = null;//先把他变成非 circular
        return helper(map, 0, count);
}

private Node helper(HashMap, int start, int end){
        if (start > end){
                return null;
        }
        if (start == end){
                Node root = map.get(start);
                Root.left = null;
                Root.right right = null;
                Return root;
        }
        int mid = start + (end - start)/2;
        Node root = map.get(mid);
        if (mid – 1 < start){
                root.right.left = null;
                root.right = null;
                root.right = helper(map, mid + 1, end);
                return root;
        }
        if (mid + 1 > end){
                root.left.right = null;
                root.left = null;
                root.left = helper(map, start, mid - 1);
                return root;
        }
        root.right.left = null;
        root.right = null;
```

```
        root.left.right = null;
        root.left = null;
        root.left = helper(map, start, mid - 1);
        root.right = helper(map, mid + 1, end);
        return root;
}
```

2. Word break 返回最小的分段数量

```
    private static int wordbreak(String s, List<String> words){
        if (s.length() == 0){
            return 0;
        }
        if (words.size() == 0){
            return -1;
        }
        int n = s.length();
        Set<String> set = new HashSet();
        for (int i = 0; i < words.size(); i++){
            set.add(words.get(i));
        }
        int[] DP = new int[n + 1];//表示从 0 开始长度为 i 的 substring 所需要的最小分段数
        Arrays.fill(DP, -1);
        DP[0] = 0;
        for (int i = 1; i <= n; i++){
            int record = Integer.MAX_VALUE;
            for (int j = i - 1; j >= 0; j--){
                if (set.contains(s.substring(j, i)) && DP[j] != -1){//注意 index 的设置
                    record = Math.min(record, DP[j] + 1);
                }
            }
            DP[i] = record == Integer.MAX_VALUE ? -1 : record;
        }
        return DP[n];
    }
```

3. Longest Arithmetic Progression 返回最长长度

```
private int LongestAP(int[] nums){
    if (nums.length <= 1){
        return nums.length;
    }
    Arrays.sort(nums);
    int n = nums.length;
    int[][] DP = new int[n][n];
    int result = 2;
    for (int i = 0; i < n - 1; i++){
        DP[i][n - 1] = 2;
    }
    for (int j = n - 2; j >= 1; j--){
        int k = j + 1;
        int t = j - 1;
        while (k < n && t >= 0){
            if (nums[k] - nums[j] == nums[j] - nums[t]){
```

```
                DP[t][j] = 1 + DP[j][k];
                result = Math.max(result, DP[t][j]);
                t--;
            } else if (nums[k] - nums[j] > nums[j] - nums[t]){
                DP[t][j] = 2;//每次 t 移动的时候必须伴随 DP 相应位置的更新
                t--;
            } else {
                k++;
            }
        }
        while (t >= 0){
            DP[t][j] = 2;
            t--;
        }
    }
    return result;
}// time O(n^2), space O(n^2)
```

4. Longest Arithmetic Progression 返回最长的序列

```
private static List<Integer> LongestAP(int[] nums){
    List<Integer> result = new ArrayList<>();
    if (nums.length == 0){
        return result;
    }
    if (nums.length == 1){
        result.add(nums[0]);
        return result;
    }
    Arrays.sort(nums);
    int n = nums.length;
    int[][] DP = new int[n][n];
    int start = 0;
    int end = 1;
    int count = 2;
    for (int i = 0; i < n - 1; i++){
        DP[i][n - 1] = 2;
    }
    for (int j = n - 2; j >= 1; j--){
        int k = j + 1;
        int t = j - 1;
        while (k < n && t >= 0){
            if (nums[k] - nums[j] == nums[j] - nums[t]){
                DP[t][j] = 1 + DP[j][k];
                if (DP[t][j] > count){
                    count = DP[t][j];
                    start = t;
                    end = j;
                }
                t--;
            } else if (nums[k] - nums[j] > nums[j] - nums[t]){
                DP[t][j] = 2;
                t--;
            } else {
                k++;
            }
        }
        while (t >= 0){
            DP[t][j] = 2;
            t--;
```

```
        }
    }
    result.add(nums[start]);
    result.add(nums[end]);
    count -= 2;
    for (int i = end  + 1; i < n && count > 0; i++){
        if (DP[end][i] == DP[start][end] - 1){
            result.add(nums[i]);
            start = end;
            end = i;
            count--;
        }
    }
    return result;
}// time O(n^2) space O(n^2)
```

给定正整数 N，返回差值为 k 的等差数列的个数，使得等差数列的和为 N

```
private int countConsecutive(int N, int k)
{
    // constraint on values of L gives us the
    // time Complexity as O(N^0.5)
    int count = 0;
    for (int L = 1; k * L * (L + 1) < 2 * N; L++)
    {
        float a = (float) ((1.0 * N-(k*L * (L + 1)) / 2) / (L + 1));
        if (a-(int)a == 0.0)
            count++;
    }
    return count;
}// time O(n^0.5) space O(1)
```

BT maximum path sum:
(这是至少有一个 node 的情况)

```
public int maxPathSum(TreeNode root) {
        int[] result = new int[1];
        result[0] = Integer.MIN_VALUE;
        helper(root, result);
        return result[0];
    }
    private int helper(TreeNode root, int[] result){
        if (root == null){
            return 0;
        }
        int temp = root.val;
        int left = helper(root.left, result);
        int right = helper(root.right, result);
        left = Math.max(0, left);
        right = Math.max(0, right);
        result[0] = Math.max(result[0], temp + left + right);
        return temp + Math.max(left, right);
    }
```

这是可以为空的情况

```
public int maxPathSum(TreeNode root) {
        int[] result = new int[1];
 //       result[0] = 0;//只需改这个
```

```
    helper(root, result);
    return result[0];
  }
  private int helper(TreeNode root, int[] result){
    if (root == null){
      return 0;
    }
    int temp = root.val;
    int left = helper(root.left, result);
    int right = helper(root.right, result);
    left = Math.max(0, left);
    right = Math.max(0, right);
    result[0] = Math.max(result[0], temp + left + right);
    return temp + Math.max(left, right);
  }
```

sum of BT

```
 1
/ \
2  3
```

返回 12 + 13 = 25
假设 root 不为 null,并且所有的 node.val 非负,且 root.val > 0。

```
public int pathsum(TreeNode root){
        List<Integer> record = new ArrayList<>();
        Helper(root, 0, record);
        Int sum = 0;
        For (int I = 0; I < record.size(); i++){
                Sum += record.get(i);
        }
        return sum;
}


private void helper(TreNode root, int val, List<Integer> record){
        int digit = Count(root.val);
        val *= (int) Math.pow(10, digit);
        val += root.val;
        if (root.left == null && root.right == null){
                record.add(val);
                return;
        }
        if (root.right == null){
                helper(root.left, val, record);
                return;
        }
        if (root.left == null){
                helper(root.right, val, record);
                return;
        }
        helper(root.left, val, record);
        helper(root.right, val, record);
```

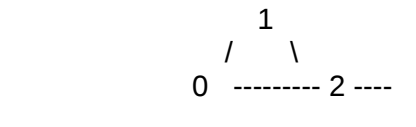
```
            return;
}

private int Count(int val){
            if (val == 0){
                        return 1;
            }
            int result = 0;
            while (val > 0){
               result++;
               val /= 10;
            }
            return result;
}
```

Basic Calculator 2 只有+ -

```
public static int calculate(String s) {
      if (s == null || s.length() == 0){
         return 0;
      }
//       s = s.trim();
      int result = 0;
 //      int now = 1;//为在加减间隔中，现在的乘除操作已经积累到多大了
      int cur = 0;//记录的是现在的这个数是多大了
      int temp = 0;//记录的是现在的这个char是多大
      int n = s.length();
      boolean sign = true;//记录的是上次一的加减符号是什么，初始默认为加
  //     boolean mul = true;//记录的是上一次的乘除符号是什么，默认为乘
      for (int i = 0; i < n; i++){
         if (i == 0 && s.charAt(i) == '+'){
            continue;
         }
         if (s.charAt(i) == ' '){
            continue;
         }
         if (s.charAt(i) >= '0' && s.charAt(i) <= '9'){
            temp = s.charAt(i) - '0';
            cur = 10*cur + temp;
            continue;
         }
         if (sign) {
            result = result + cur;
         } else {
            result = result - cur;
         }
         sign = s.charAt(i) == '+' ? true : false;
         cur = 0;
      }
      if (sign){
         result = result + cur;
      } else {
         result = result - cur;
      }
      return result;
   }
```

Basic Calculator 只有+,*情况

Pubic int




```
Public Node copy(Node node){
        If (node == null){
                Return null;
        }
        Map<Node, Node> map = new HashMap();
        Deque<Node> queue = new LinkedList<>();
        Set<Node> set = new HashSet();
        queue.offerLast(node);
        Node root = new Node(node.val);
        map.put(node, root);
        While (!queue.isEmpty()){
                Node temp = queue.pollFirst();
                If (set.add(temp)){
                        Node newnode = map.get(temp);
                        For (int i = 0; i < temp.neighbors.size(); i++){
                                Node re = temp.neighbors.get(i);
                                If (map.containsKey(re)){
                                        newnode.neighbors.add(map.get(re));
                                } else {
                                        Node ree = new Node(re.val);
                                        map.put(re, ree);
                                        newnode.neighbors.add(map.get(re));
                                }
                                queue.offerLast(re);
                        }
                }
        }
        Return root;
}


Public Node copy(Node node){
        Map<Node, Node> map = new HashMap();
        Return helper(node, map);
}
```

```
Private Node helper(Node node, Map<Node, Node> map){
        If (node == null){
                Return null;
        }
        If (map.containsKey(node)){
                Return map.get(node);
        }
        Node temp = new Node(node.val);
        map.put(node, temp);
        For (Node no : node.neighbors){
                temp.neighbors.add(helper(no, map));
        }
        Return temp;
}
```

[[0, 1] [1, 0], [1,1]] top k elements that's closest to (0, 0)

```
Public Point find(List<Point> list, int k){
        PriorityQueue<Point> maxheap = new PriorityQueue<Point>(new Comparator<Point>(){
        Public int compare (Point o1, Point o2){
                Return Count(o1) < Count(o2) ? 1 : -1;
        }
});
        For (int i = 0; i < k; i++){
                maxheap.offer(list.get(i));
        }
        For (int i = k; i < list.size(); i++){
                If (Count(list.get(i)) < Count(maxheap.peek())){
                        mapheap.poll();
                        maxheap.offer(list.get(i));
                }
        }
        Return maxheap.peek();
}

Private long Count(Point o1){
        Return (long) o1.x*o1.x + (long) o1.y*o1.y;
}
```

给一个 list of node,每个 node 都有一个 parent 指针，并且有一个 val 值，让你返回一个 List<Integer> 其中每一个元素对应 list of node 里相应位置的 node 的 tree 里元素总和。

```
Pulic List<Integer> find(List<Node> list){
        List<Integer> result = new ArrayList<>();
```

```
        If (list.size() == 0){
                Return result;
        }
        Map<Node, List<Node>> map = new HashMap();
        For (int I = 0; I < list.size(); i++){
                If (!map.containsKey(list.get(i))){
                        List<Node> cur1 = new ArrayList<>();
                        Map.put(list.get(i), cur1);
                }
                If (list.get(i).parent != null && !
        map.containsKey(list.get(i).parent)){
                        List<Node> cur = new ArrayList<>();
                        Map.put(list.get(i).parent, cur);
                }
                map.get(list.get(i).parent).add(list.get(i));
        }
        Map<Node, Integer> vals = new HashMap();
        For (Node node : list){
                If (vals.containsKey(node)){
                        Result.add(vals.get(node));
                } else {
                        int val = DFS(node, map, vals);
                        result.add(val);
                }
        }
        return result;
}

private int DFS(Node node, Map<Node, List<Node>> map, Map<Node,
Integer> vals){
        if (vals.containsKey(node)){
                return vals.get(node);
        }
        int sum = 0;
        int temp = 0;
        for (int I = 0; I < map.get(node).size(); i++){
                if (vals.containsKey(map.get(node).get(i))){
                        sum += vals.get(map.get(node).get(i));
                } else {
                        temp = DFS(map.get(node).get(i), map, vals);
                        sum += temp;
                }
        }
        vals.put(node, sum + node.val);
        return sum + node.val;
}
```

follow up: 如果有一个点的值改变了，那么我们应该如何变化这个 result list.

题目是给一个 Array of Nodes，每个 node 有 left 和 right 的 child，求问这些 node 能不能组成一个二叉树

```
public Boolean test(List<Node> list){
        if (list.size() == 0){
                return true;
        }
        Set<Node> set = new HashSet();
        Map<Node, Integer> map = new HashMap();
        For (int I = 0; I < list.size(); i++){
                If (!map.containsKey(list.get(i))){
                        Map.put(list.get(i), 0);
                        Set.add(list.get(i));
                }
                if (list.get(i).left != null){
                        if (!map.containsKey(list.get(i).left)){
                                map.put(list.get(i).left, 1);
                        } else {
                                if (map.get(list.get(i).left) == 1){
                                        return false;
                                }
                                map.put(list.get(i).left, 1);
                        }
                        set.remove(list.get(i).left);
                }
                if (list.get(i).right != null){
                        if (!map.containsKey(list.get(i).right)){
                                map.put(list.get(i).right, 1);
                        } else {
                                if (map.get(list.get(i).right) == 1){
                                        return false;
                                }
                                map.put(list.get(i).right, 1);
                        }
                        set.remove(list.get(i).right);
                }
        }
        if (set.size() != 1){
                return false;
        }
        Node root = set.iterator().next();
```

```
        Deque<Node> queue = new LinkedList<>();
        Queue.offerLast(root);
        While (!queue.isEmpty()){
                Node temp = queue.pollFirst();
                If (temp.left != null){
                        If (set.add(temp.left)){
                                Queue.offerLast(temp.left);
                        } else {
                                return false;
                        }
                }
                If (temp.right != null){
                        If (set.add(temp.right)){
                                Queue.offerLast(temp.right);
                        } else {
                                return false;
                        }
                }
        }
        return set.size() == list.size();
}


import java.io.*;
import java.util.*;

/*
 * To execute Java, please define "static void main" on a class
 * named Solution.
 *
 * If you need more classes, simply define them inline.
 **/
merge k sorted array


public int[] merge(int[][] arrays){
  if (arrays.length == 0){
    return new int[0];
  }
  int n = arrays.length;
  List<Integer> result = new ArrayList<>();
  PriorityQueue<Cell> minheap = new PriorityQueue<Cell>(new Comparator<Cell>(){
    public int compare(Cell o1, Cell o2){
      return o1.val - o2.val;
    }
  });
```

```
  for (int i = 0; i < arrays.length; i++){
    if (arrays[i].length > 0){
      minheap.offer(new Cell(arrays[i][0], i, 0));
    }
  }
  while (!minheap.isEmpty()){
    Cell temp = minheap.poll();
    int val = temp.val;
    int row = temp.row;
    int col = temp.col;
    result.add(val);
    if (col < arrays[row].length - 1){
      minheap.offer(new Cell(arrays[row][col + 1], row, col + 1));
    }
  }
  int[] fin = new int[result.size()];
  for (int i = 0; i < result.size(); i++){
    fin[i] = result.get(i);
  }
  return fin;
}

class Cell{
  int val;
  int row;
  int col;
  public Cell(int val, int row, int col){
    this.val = val;
    this.row = row;
    this.col = col;
  }
}
```

498. Diagonal Traverse

左上到右下

```
public int[] traverse(int[][] matrix){
  if (matrix.length == 0){
    return new int[0];
  }
  int n = matrix.length;
  int m = matrix[0].length;
  int[] result = new int[n*m];
  boolean[][] visited = new boolean[n][m];
  Deque<Cell> queue = new LinkedList<>();
```

```
  queue.offerLast(new Cell(matrix[0][0], 0, 0));
 // result[0] = matrix[0][0];
  int left = 0;
  while (!queue.isEmpty()){
    Cell temp = queue.pollFirst();
    int val = temp.val;
    int row = temp.row;
    int col = temp.col;
    result[left++] = val;
    if (col < m - 1 && !visited[row][col + 1]){
      queue.offerLast(new Cell(matrix[row][col + 1], row, col + 1));
      visited[row][col + 1] = true;
    }
    if (row < n - 1 && !visited[row + 1][col]){
      queue.offerLast(new Cell(matrix[row + 1][col], row + 1, col));
      visited[row + 1][col] = true;
    }
  }
  return result;
}

class Cell{
  int val;
  int row;
  int col;
  public Cell(int val, int row, int col){
    this.val = val;
    this.row = row;
    this.col = col;
  }
}
```

Q: 顺时针给出二维坐标系内三角形的三个顶点的坐标，求三角形面积之和。

思路，首先我们可以通过坐标的平行转移实现所有的点都保证在横坐标上方。之后，对于每一条边的两个端点都做垂直于 x 轴的辅助线，这样我们就会产生三个梯形。三角形的面积就等于三个梯形当中，由上边引出的梯形面积之和减去由下边引出的梯形面积之和。同时注意，因为点是顺时针给出的，所以如果两个相邻的点的横坐标是递增的，我们就可以认定他是上边，其对应的梯形面积需要加。而如果两个相邻的点的坐标是递减的那么这条边就是下边，需要减。注意，点三到点一也是一条边。（注意，如果出现两个点的横坐标一样，那么那个对应的梯形面积就是 0，不影响这个方法的使用）。

让我们实现吧

```
public double Area(int[][] points){
```

```
//先进性坐标平移，确保所有的点都在横坐标上方
  int miny = Integer.MAX_VALUE;
  for (int i = 0; i < 3; i++){
    miny = Math.min(miny, points[i][1]);
  }
  if (miny < 0){
    for (int i = 0; i < 3; i++){
      points[i][1] -= miny;
    }
  }
  //完了之后，我们就开始探讨边是上边还是下边的问题。
  double sum = 0;

  for (int i = 0; i < 3; i++){
    if (i == 2){
      if (points[2][0] <= points[0][0]){//后一个大于等于前一个则为上边
    sum += (double) (points[2][1] + points[0][1])* (double) (points[0][0] - points[2][0])/ 2.0;
  } else {
    sum -= (double) (points[2][1] + points[0][1])* (double) (points[2][0] - points[0][0])/ 2.0;
  }
    } else {
      if (points[i][0] <= points[i + 1][0]){//后一个大于等于前一个则为上边
    sum += (double) (points[i][1] + points[i + 1][1])* (double) (points[i + 1][0] - points[i][0])/ 2.0;
  } else {
    sum -= (double) (points[i][1] + points[i + 1][1])* (double) (points[i][0] - points[i + 1][0])/ 2.0;
  }
    }
  }
  return sum;
}

// follow up: 多边形，也是一样


//字符串自然顺序比较，比如"a9"比"a12"小，字母比数字小。大写字母在小写字母之前

public static List<String> Sort(List<String> list){
        if (list.size() <= 1){
            return list;
        }
        Collections.sort(list, new Comparator<String>(){
            public int compare(String s1, String s2){
                if (s1.equals(s2)){
                    return 0;
                }
                int left1 = 0;
```

```
            int left2 = 0;
            while (left1 < s1.length() && left2 < s2.length()){
                if (Character.isLetter(s1.charAt(left1)) && Character.isLetter(s2.charAt(left2))){
                    if (s1.charAt(left1) == s2.charAt(left2)){
                        left1++;
                        left2++;
                    } else {
                        return s1.charAt(left1) - s2.charAt(left2);
                    }
                } else if (Character.isLetter(s1.charAt(left1))){
                    return -1;
                } else if (Character.isLetter(s2.charAt(left2))){
                    return 1;
                } else {
                    int left11 = left1;
                    int left22 = left2;
                    while (left1 < s1.length()){
                        if (Character.isLetter(s1.charAt(left1))){
                            break;
                        }
                        left1++;
                    }
                    while (left2 < s2.length()){
                        if (Character.isLetter(s2.charAt(left2))){
                            break;
                        }
                        left2++;
                    }
                    int l = Integer.parseInt(s1.substring(left11, left1));
                    int r = Integer.parseInt(s2.substring(left22, left2));
                    if (l == r){
                        continue;
                    }
                    return l - r;
                }
            }
            if (left1 == s1.length()){
                return -1;
            }
            return 1;
        }
    });
    return list;
}
```

//像个 string 表示的整数相加。有负数。假设已知，s2 为负，s1 为正

```
public static String sum(String s1, String s2){
    if (s1.length() == 0){
        return s2;
    }
    if (s2.length() == 0){
        return s1;
    }
    int carry = 0;
    s2 = s2.substring(1);
    boolean flag = s1.compareTo(s2) >= 0;
    String temp1 = flag ? s1 : s2;
    String temp2 = flag ? s2 : s1;
    int left1= temp1.length() - 1;
    int left2 = temp2.length() - 1;
    StringBuilder sb = new StringBuilder();
    while (left1 >= 0 && left2 >= 0){
        int plus = temp1.charAt(left1) - '0';
        plus += carry;
        carry = 0;
        int minus = temp2.charAt(left2) - '0';
        if (plus >= minus){
            sb.append(plus - minus);
        } else {
            carry = -1;
            int rec = plus + 10 - minus;
            sb.append(rec);
        }
        left1--;
        left2--;
    }
    if (left2 == -1){
        while (left1 >= 0){
            int plus = temp1.charAt(left1) - '0';
            plus += carry;
            carry = 0;
            int minus = 0;
            if (plus >= minus){
                sb.append(plus - minus);
            } else {
                carry = -1;
                int rec = plus + 10 - minus;
                sb.append(rec);
            }
            left1--;
        }
    }
    while (sb.length() > 0){
        if (sb.charAt(sb.length() - 1) == '0'){
            sb.deleteCharAt(sb.length() - 1);
        } else {
            break;
        }
    }
    if (sb.length() == 0){
        return "0";
    }
    if (!flag){
        sb.append('-');
    }
```

```
    return sb.reverse().toString();
}


//两个正数相加，可能有小数
public String sumdot(String s1, String s2){//思路就是找到小数部分，然后分开处理。注意，小数部分相加不是从最后以为开始的，而是以左往右的共同位相加的
  if (s1.length() == 0){
    return s2;
  }
  if (s2.length() == 0){
    return s1;
  }
  String[] sa1 = s1.split("\\.");
  String[] sa2 = s2.split("\\.");
  if (sa1.length == 1 && sa2.length == 1){
    return sumint(s1, s2, 0);//表示在 s1, s2 都代表非负整数的情况下，carry 初始值为 0 的时候，最终相加的和
  }
  if (sa1.length == 1){
    String temp = sumint(s1, sa2[0], 0);
    return temp + "." + sa2[1];
  }
  if (sa2.length == 1){
    String temp = sumint(sa1[0], s2, 0);
    return temp + "." + sa1[1];
  }
  String dig1 = sa1[1];
  String dig2 = sa2[1];
  boolean flag = dig1.length() >= dig2.length();
  String dig1rec = dig1;
  String dig2rec = dig2;
  String record = "";
  if (flag){
    dig1rec = dig1.substring(0, dig2.length());
    record = dig1.substring(dig2.length());
  } else {
    dig2rec = dig2.substring(0, dig1.length());
    record = dig2.substring(dig1.length());
  }
  int carry = 0;
  int left1 = dig1rec.length() - 1;
  int left2 = dig2rec.lenth() - 1;
  StringBuilder sb = new StringBuilder();
  while (left1 >= 0 && left2 >= 0){
    int num = dig1rec.charAt(left1) - '0' + dig2rec.charAt(left2) - '0';
    carry += num;
```

```
    sb.append(carry%10);
    carry /= 10;
    left1--;
    left2--;
   }
   String digit = sb.reverse().toString() + record;
   String integer = sumint(sa1[0], sa2[0], 1);
   return integer + "." + digit;
}

//两个稀疏矩阵,做一个 class，使得更有效的进行存储。
```