Given a non-negative integer `num`, repeatedly add all its digits until the result has only one digit.

Example:

```
Input: 38
Output: 2
Explanation: The process is like: 3 + 8 = 11, 1 + 1 = 2.
             Since 2 has only one digit, return it.
```

```
class Solution {
    public int addDigits(int num) {
        if (num == 0) {
            return 0;
        }
        return num % 9 == 0 ? 9 : num % 9;
    }
}
```

Given two integers representing the numerator and denominator of a fraction, return the fraction in string format.

If the fractional part is repeating, enclose the repeating part in parentheses.

```
class Solution {
    public String fractionToDecimal(int n, int d) {
        if (n == 0) {
            return "0";
        }
        StringBuilder res = new StringBuilder();

        res.append(((n > 0) ^ (d > 0)) ? "-" : "");

        long num = Math.abs((long)n);
        long den = Math.abs((long)d);

        res.append(num / den);  // integer part
        long r = num % den;

        if (r == 0) return res.toString();

        res.append(".");        // decimal point
        Map<Long, Integer> map = new HashMap<>();
        map.put(r, res.length());
        while (r != 0) {
            r *= 10;
            res.append(r / den);
```

```
            r %= den;
            if (map.containsKey(r)) {
                int index = map.get(r);
                res.insert(index, "(");
                res.append(")");
                break;
            } else {
                map.put(r, res.length());
            }
        }

        return res.toString();
    }
}
```

If you add periods (`'.'`) between some characters in the **local name** part of an email address, mail sent there will be forwarded to the same address without dots in the local name.
If you add a plus (`'+'`) in the **local name**, everything after the first plus sign will be **ignored**.

```
class Solution {
    public int numUniqueEmails(String[] emails) {
        int res = 0;
        Set<String> set = new HashSet<>();
        for (String s : emails) {
            String[] strs = s.split("@");
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < strs[0].length(); i++) {
                if (strs[0].charAt(i) == '+') {
                    break;
                } else if (strs[0].charAt(i) != '.') {
                    sb.append(strs[0].charAt(i));
                }
            }
            sb.append("@" + strs[1]);
            set.add(sb.toString());
        }
        return set.size();
    }
}
```

A rectangle is represented as a list `[x1, y1, x2, y2]`, where `(x1, y1)` are the coordinates of its bottom-left corner, and `(x2, y2)` are the coordinates of its top-right corner.

Two rectangles overlap if the area of their intersection is positive. To be clear, two rectangles that only touch at the corner or edges do not overlap.

Given two (axis-aligned) rectangles, return whether they overlap.

```
class Solution {
    public boolean isRectangleOverlap(int[] rec1, int[] rec2) {
        return rec1[0] < rec2[2] && rec1[2] > rec2[0] && rec2[3] > rec1[1] && rec1[3] > rec2[1];
    }
}
```

Algorithm of Insertion Sort:

1. Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list.
2. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.
3. It repeats until no input elements remain.

```
class Solution {
    public ListNode insertionSortList(ListNode head) {
        ListNode dh = new ListNode(-1);
        ListNode itr = dh;

        while (head != null) {
            ListNode tmp = head.next;
            head.next = null;
            itr = dh;
            while (itr.next != null && itr.next.val <= head.val) {
                itr = itr.next;
            }
            head.next = itr.next;
            itr.next = head;
            head = tmp;
        }
        return dh.next;
    }
}
```

Given the root node of a binary search tree (BST) and a value to be inserted into the tree, insert the value into the BST. Return the root node of the BST after the insertion. It is guaranteed that the new value does not exist in the original BST.

Note that there may exist multiple valid ways for the insertion, as long as the tree remains a BST after insertion. You can return any of them.

```
class Solution {
    public TreeNode insertIntoBST(TreeNode root, int val) {
        if (root == null) {
            return new TreeNode(val);
        }
        if (root.val > val) {
            if (root.left == null) {
                root.left = new TreeNode(val);
            } else {
                insertIntoBST(root.left, val);
            }
        } else {
            if (root.right == null) {
                root.right = new TreeNode(val);
            } else {
                insertIntoBST(root.right, val);
            }
        }

        return root;
    }
}
```

There are *N* gas stations along a circular route, where the amount of gas at station *i* is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1.

Note:

- If there exists a solution, it is guaranteed to be unique.
- Both input arrays are non-empty and have the same length.
- Each element in the input arrays is a non-negative integer.

```
class Solution {
    public int canCompleteCircuit(int[] gas, int[] cost) {
        int sum = 0, total = 0, start = 0;
        for (int i = 0; i < gas.length; i++) {
            sum += gas[i] - cost[i];
            total += gas[i] - cost[i];
            if (sum < 0) {
                sum = 0;
                start = i+1;
            }
        }
        if (total < 0) {
            return -1;
        }
        return start;
    }
}
```

Given an integer array *nums*, find the sum of the elements between indices *i* and *j* (*i* ≤ *j*), inclusive.

Example:

```
Given nums = [-2, 0, 3, -5, 2, -1]

sumRange(0, 2) -> 1
sumRange(2, 5) -> -1
sumRange(0, 5) -> -3
```

```
class NumArray {

    ArrayList<Integer> sum = new ArrayList<>();
    public NumArray(int[] nums) {
        int s = 0;
        for (int i = 0; i < nums.length; i++){
            s += nums[i];
            sum.add(s);
        }
    }

    public int sumRange(int i, int j) {
        if (i == 0) return sum.get(j);
        else {
            return sum.get(j) - sum.get(i-1);
        }
    }
}


class NumMatrix {
    int[][] res;
    public NumMatrix(int[][] matrix) {
        if (matrix.length == 0) {
            res = new int[0][0];
            return;
        }
        int row = matrix.length;
        int col = matrix[0].length;
        res = new int[row+1][col+1];
        for (int i = 1; i < row+1; i++){
            for (int j = 1; j < col+1; j++){
                res[i][j] = res[i][j-1] + res[i-1][j] - res[i-1][j-1] + matrix[i-1][j-1];
            }
```

```
        }
    }

    public int sumRegion(int row1, int col1, int row2, int col2) {
        return res[row2+1][col2+1] + res[row1][col1] - res[row2+1][col1] - res[row1][col2+1];
    }
}
```

You are given a *m x n* 2D grid initialized with these three possible values.

1. `-1` - A wall or an obstacle.
2. `0` - A gate.
3. `INF` - Infinity means an empty room. We use the value $2^{31} - 1 = 2147483647$ to represent `INF` as you may assume that the distance to a gate is less than `2147483647`.

Fill each empty room with the distance to its *nearest* gate. If it is impossible to reach a gate, it should be filled with `INF`.

```
class Solution {
    public void wallsAndGates(int[][] rooms) {
        for (int i = 0; i < rooms.length; i++) {
            for (int j = 0; j < rooms[0].length; j++) {
                if (rooms[i][j] == 0) {
                    h(rooms, i, j, 0);
                }
            }
        }
    }

    private void h(int[][] rooms, int r, int c, int d) {
        if (r < 0 || c < 0 || r >= rooms.length || c >= rooms[r].length
            || d > rooms[r][c]) {
            return;
        }
        rooms[r][c] = Math.min(d, rooms[r][c]);
        h(rooms, r+1, c, d+1);
        h(rooms, r-1, c, d+1);
        h(rooms, r, c+1, d+1);
        h(rooms, r, c-1, d+1);
    }
}
```

Given a list of **non-negative** numbers and a target **integer** k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to the multiple of **k**, that is, sums up to n*k where n is also an **integer**.

Example 1:

```
Input: [23, 2, 4, 6, 7],  k=6
Output: True
Explanation: Because [2, 4] is a continuous subarray of size 2 and sums up to 6.
```

```
class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        Map<Integer, Integer> mp = new HashMap<>();
        mp.put(0, -1);
        int sum = 0;

        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
            if (k != 0) {
                sum %= k;
            }
            if (mp.containsKey(sum)) {
                if (i - mp.get(sum) > 1)
                    return true;
            } else {
                mp.put(sum, i);
            }
        }

        return false;
    }
}
```

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.

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

```
class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        TreeNode cur = root;
        TreeNode pre = null;
        while (cur != null && cur.val != key) {
            pre = cur;
            if (cur.val > key) {
                cur = cur.left;
            } else if (cur.val < key) {
                cur = cur.right;
            }
        }
        if (pre == null) {
            return delete(cur);
        } else if (pre.left == cur) {
            pre.left = delete(cur);
        } else {
            pre.right = delete(cur);
        }
        return root;
    }

    public TreeNode delete(TreeNode node) {
        if (node == null)
            return null;
        if (node.left == null)
            return node.right;
        if (node.right == null)
            return node.left;
        TreeNode tmp = node.right;
        TreeNode pre = null;
        while (tmp.left != null) {
            pre = tmp;
            tmp = tmp.left;
        }
        tmp.left = node.left;
        if (node.right != tmp) {
            pre.left = tmp.right;
            tmp.right = node.right;
        }
        return tmp;
    }
}
```

```
class Solution {
    private static final int[] values = {
    1000, 900, 500, 400,
    100,  90,  50,  40,
    10,   9,   5,   4,
    1 };
    private static final String[] symbols = {
    "M", "CM", "D", "CD",
    "C", "XC", "L", "XL",
    "X", "IX", "V", "IV",
    "I"
    };

    public String intToRoman(int num) {
        String res = "";
        for (int i = 0; i < values.length; i++){
            if (num / values[i] > 0) {
                for (int j = 0; j < num / values[i]; j++) {
                    res += symbols[i];
                }
                num %= values[i];
            }
        }
        return res;
    }
}
```

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.

Example 1:

```
Input: "(()"
Output: 2
Explanation: The longest valid parentheses substring is "()"
```

Example 2:

```
Input: ")()())"
Output: 4
Explanation: The longest valid parentheses substring is "()()"
```

```
class Solution {
    public int longestValidParentheses(String s) {
        Stack<Integer> stack = new Stack<>();
        int max = 0, left = -1;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(') {
                stack.push(i);
            } else {
                if (stack.isEmpty()) {
                    left = i;
                } else {
                    stack.pop();
                    max = Math.max(max, stack.isEmpty() ? i - left : i - stack.peek());
                }
            }
        }
        return max;
    }
}
```

Return any binary tree that matches the given preorder and postorder traversals.

Values in the traversals `pre` and `post` are distinct positive integers.

Example 1:

```
Input: pre = [1,2,4,5,3,6,7], post = [4,5,2,6,7,3,1]
Output: [1,2,3,4,5,6,7]
```

```
class Solution {
    public TreeNode constructFromPrePost(int[] pre, int[] post) {
        Deque<TreeNode> s = new ArrayDeque<>();
        s.offer(new TreeNode(pre[0]));
        for (int i = 1, j = 0; i < pre.length; ++i) {
            TreeNode node = new TreeNode(pre[i]);
            while (s.getLast().val == post[j]) {
                s.pollLast(); j++;
            }
            if (s.getLast().left == null)
                s.getLast().left = node;
            else
                s.getLast().right = node;
            s.offer(node);
        }
        return s.getFirst();
    }
}
```

There is a **ball** in a maze with empty spaces and walls. The ball can go through empty spaces by rolling **up**, **down**, **left** or **right**, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the ball's **start position**, the **destination** and the **maze**, determine whether the ball could stop at the destination.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

```
class Solution {
    class Point {
        int x,y;
        public Point(int _x, int _y) {x=_x;y=_y;}
    }

    public boolean hasPath(int[][] maze, int[] start, int[] destination) {
        int m = maze.length, n = maze[0].length;
        if (start[0] == destination[0] && start[1] == destination[1])
            return true;
        int[][] dir = new int[][] {{-1,0}, {0,1}, {1,0}, {0,-1}};
        boolean[][] visited = new boolean[m][n];
        LinkedList<Point> list = new LinkedList<>();
        visited[start[0]][start[1]] = true;
        list.offer(new Point(start[0], start[1]));
        while (!list.isEmpty()) {
            Point p = list.poll();
            int x = p.x, y = p.y;
            for (int i = 0; i < 4; i++) {
                int xx = x, yy = y;
                while (xx >= 0 && xx < m && yy >= 0 && yy < n && maze[xx][yy] == 0) {
                    xx += dir[i][0];
                    yy += dir[i][1];
                }
                xx -= dir[i][0];
                yy -= dir[i][1];
                if (visited[xx][yy])
                    continue;
                visited[xx][yy] = true;
                if (xx == destination[0] && yy == destination[1])
                    return true;
                list.offer(new Point(xx, yy));
            }
        }
        return false;

    }
}
```

Return the shortest distance from start to destination

```
class Solution {
    class Point {
        int x,y,l;
        public Point(int _x, int _y, int _l) {x=_x;y=_y;l=_l;}
    }
    public int shortestDistance(int[][] maze, int[] start, int[] destination) {
        int m = maze.length, n = maze[0].length;
        int[][] length=new int[m][n]; // record length
        for (int i = 0; i < m * n; i++)
            length[i / n][i % n] = Integer.MAX_VALUE;
        int[][] dir = new int[][] {{-1,0},{0,1},{1,0},{0,-1}};
        PriorityQueue<Point> list = new PriorityQueue<>((o1,o2) -> o1.l - o2.l); // using priority
queue
        list.offer(new Point(start[0], start[1], 0));
        while (!list.isEmpty()) {
            Point p = list.poll();
            if (length[p.x][p.y] <= p.l)
                continue; // if we have already found a route shorter
            length[p.x][p.y] = p.l;
            for (int i = 0; i < 4; i++) {
                int xx = p.x, yy = p.y, l = p.l;
                while (xx >= 0 && xx < m && yy >= 0 && yy < n && maze[xx][yy]==0) {
                    xx += dir[i][0];
                    yy += dir[i][1];
                    l++;
                }
                xx -= dir[i][0];
                yy -= dir[i][1];
                l--;
                list.offer(new Point(xx, yy, l));
            }
        }
        return length[destination[0]][destination[1]] == Integer.MAX_VALUE ? -1 :
length[destination[0]][destination[1]];
    }
}
```

Given a node from a cyclic linked list which is sorted in ascending order, write a function to insert a value into the list such that it remains a cyclic sorted list. The given node can be a reference to *any* single node in the list, and may not be necessarily the smallest value in the cyclic list.

If there are multiple suitable places for insertion, you may choose any place to insert the new value. After the insertion, the cyclic list should remain sorted.

If the list is empty (i.e., given node is `null`), you should create a new single cyclic list and return the reference to that single node. Otherwise, you should return the original given node.

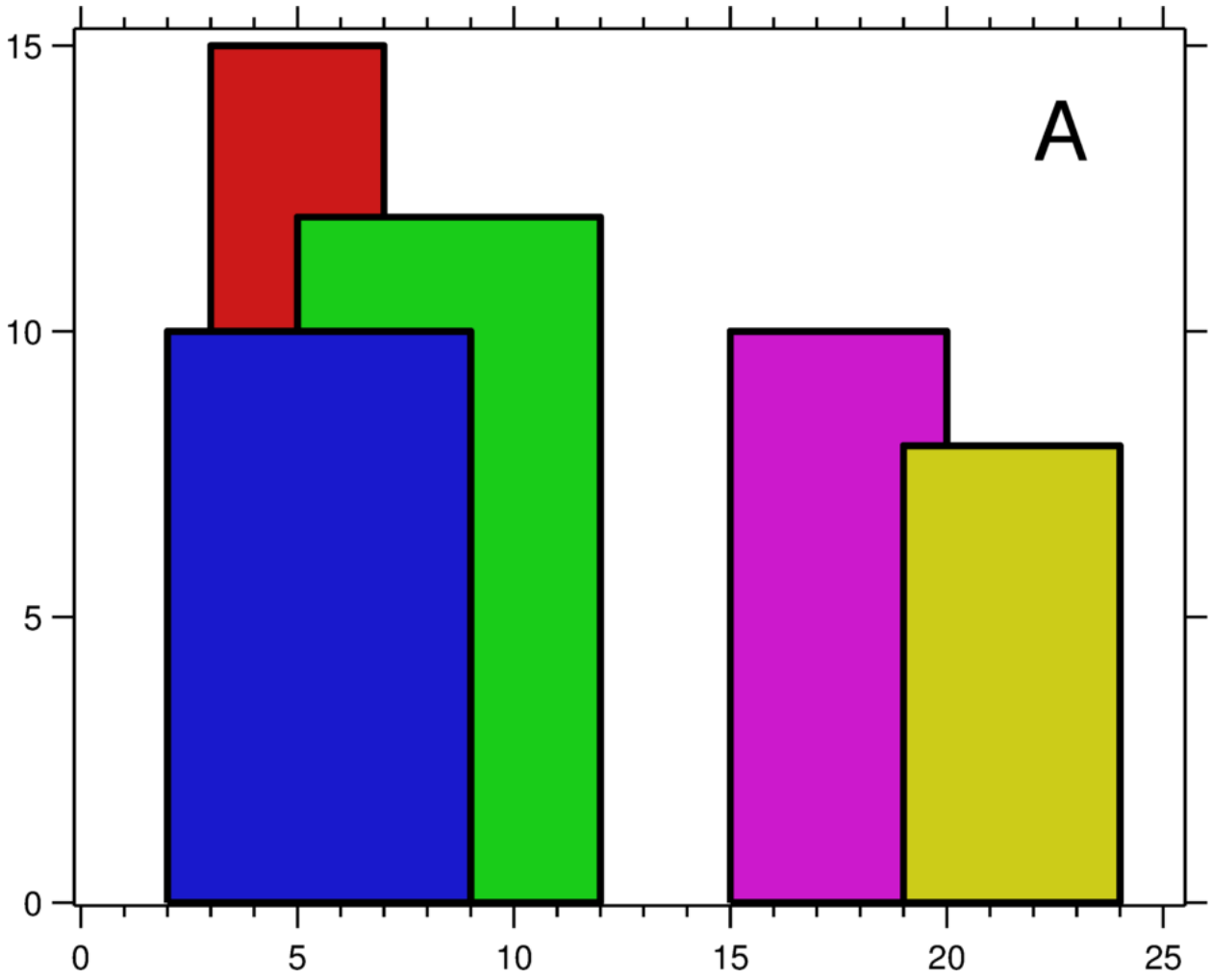
```
class Solution {
    public Node insert(Node start, int x) {
        if (start == null) {
            Node node = new Node(x, null);
            node.next = node;
            return node;
        }
        // is start is NOT null, try to insert it into correct position
        Node cur = start;
        while (true) {
            // case 1A: has a tipping point, still climbing
            if (cur.val < cur.next.val) {
                if (cur.val <= x && x <= cur.next.val) { // x in between cur and next
                    insertAfter(cur, x);
                    break;
                }
            // case 1B: has a tipping point, about to return back to min node
            } else if (cur.val > cur.next.val) {
                if (cur.val <= x || x <= cur.next.val) { // cur is the tipping point, x is max or min val
                    insertAfter(cur, x);
                    break;
                }
            // case 2: NO tipping point, all flat
            } else {
                if (cur.next == start) {  // insert x before we traverse all nodes back to start
                    insertAfter(cur, x);
                    break;
                }
            }
            // None of the above three cases met, go to next node
            cur = cur.next;
        }
        return start;
    }
    private void insertAfter(Node cur, int x) {
        cur.next = new Node(x, cur.next);
    }
}
```

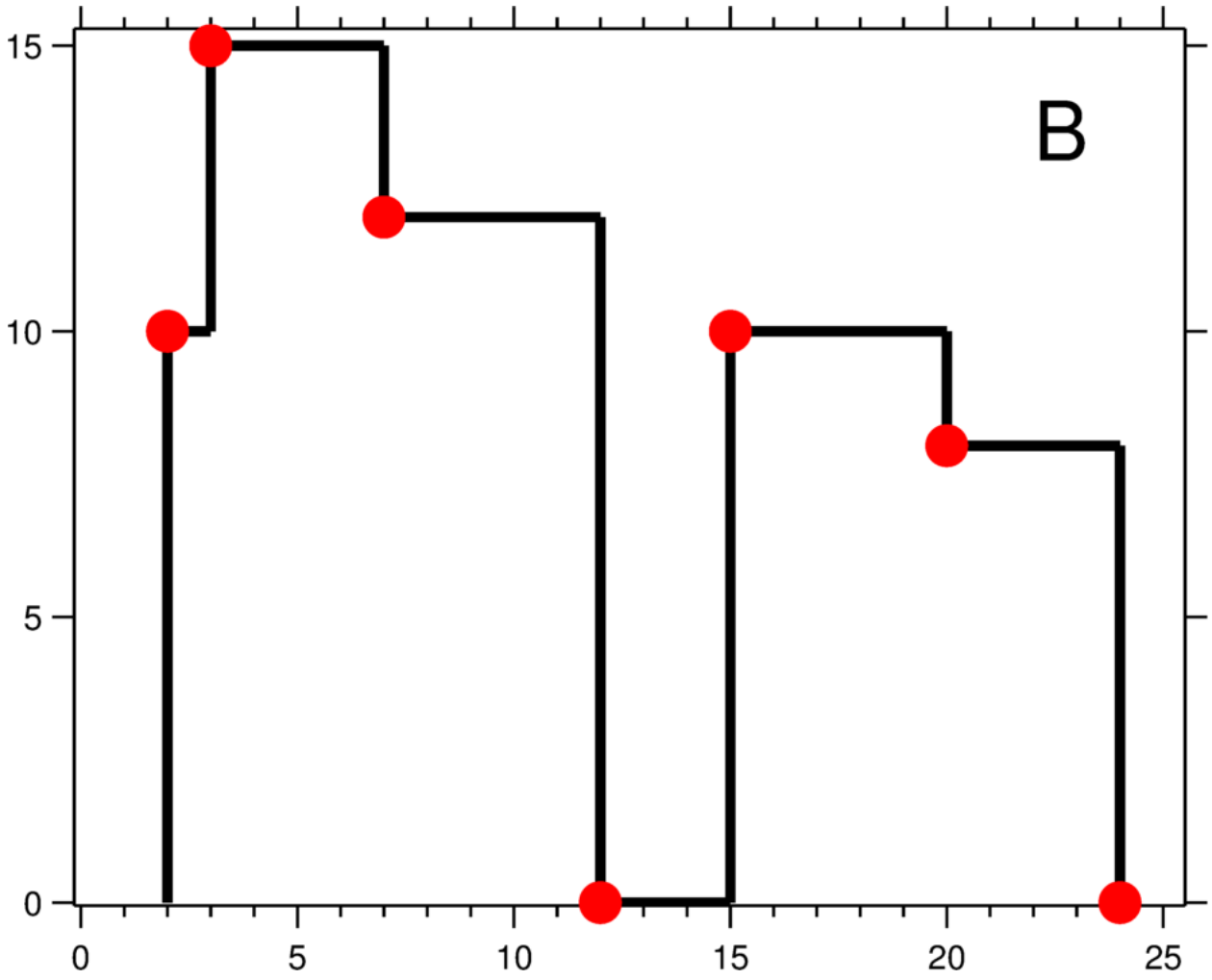
There are **N** students in a class. Some of them are friends, while some are not. Their friendship is transitive in nature. For example, if A is a **direct** friend of B, and B is a **direct** friend of C, then A is an **indirect**friend of C. And we defined a friend circle is a group of students who are direct or indirect friends.

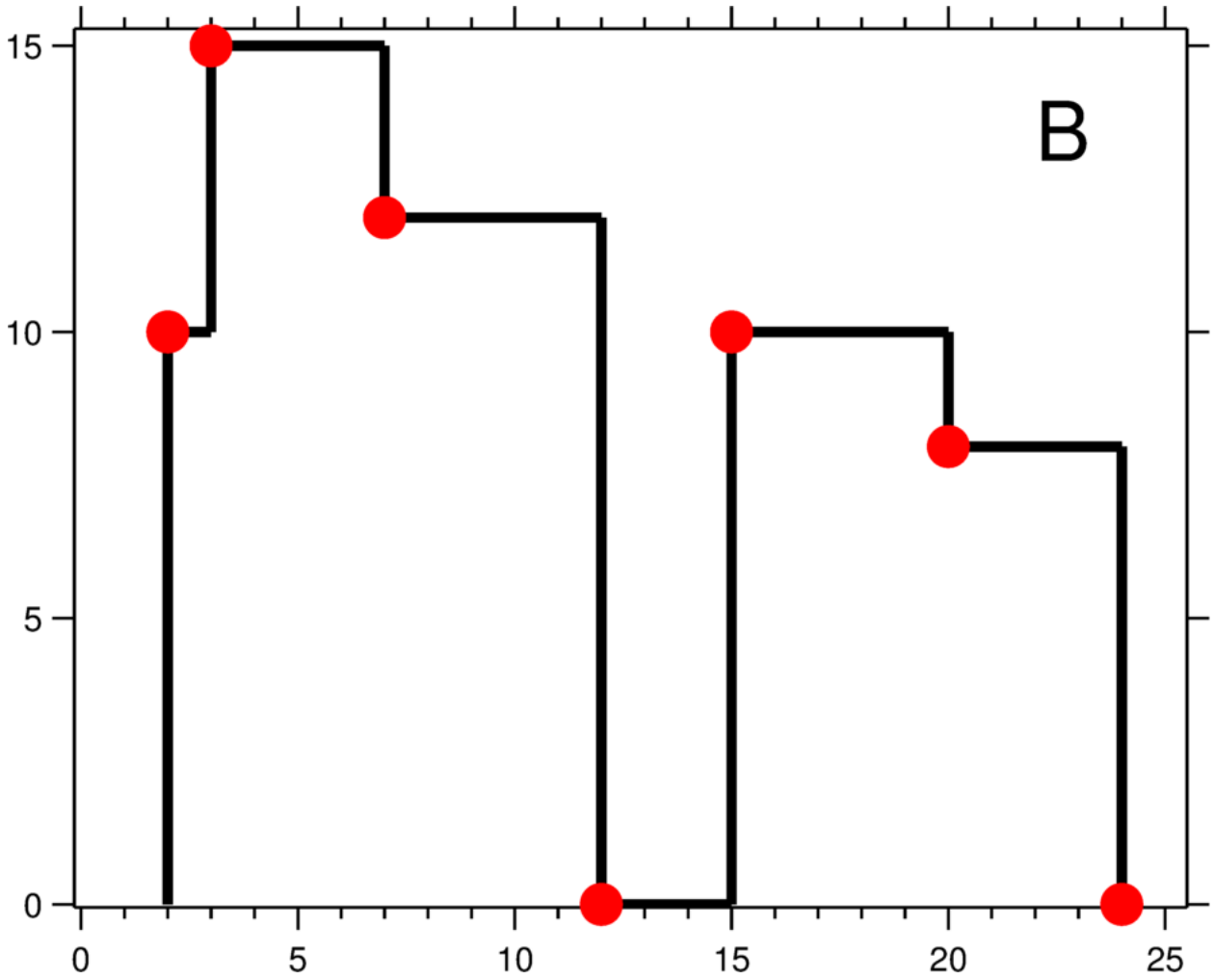
Given a **N*N** matrix **M** representing the friend relationship between students in the class. If M[i][j] = 1, then the i_{th} and j_{th} students are **direct** friends with each other, otherwise not. And you have to output the total number of friend circles among all the students.

```
class Solution {
    public void dfs(int[][] M, int[] visited, int i) {
        for (int j = 0; j < M.length; j++) {
            if (M[i][j] == 1 && visited[j] == 0) {
                visited[j] = 1;
                dfs(M, visited, j);
            }
        }
    }
    public int findCircleNum(int[][] M) {
        int[] visited = new int[M.length];
        int count = 0;
        for (int i = 0; i < M.length; i++) {
            if (visited[i] == 0) {
                dfs(M, visited, i);
                count++;
            }
        }
        return count;
    }
}
```

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Now suppose you are **given the locations and height of all the buildings** as shown on a cityscape photo (Figure A), write a program to **output the skyline**formed by these buildings collectively (Figure B).




B
15
10
5
0
0
5
10
15
20
25



The geometric information of each building is represented by a triplet of integers `[Li, Ri, Hi]`, where `Li` and `Ri` are the x coordinates of the left and right edge of the ith building, respectively, and `Hi` is its height. It is guaranteed that `0 ≤ Li, Ri ≤ INT_MAX`, `0 < Hi ≤ INT_MAX`, and `Ri - Li > 0`. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

For instance, the dimensions of all buildings in Figure A are recorded as: `[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]` .

The output is a list of "**key points**" (red dots in Figure B) in the format of `[ [x1,y1], [x2, y2], [x3, y3], ... ]` that uniquely defines a skyline. **A key point is the left endpoint of a horizontal line segment**. Note that the last key point, where the rightmost building ends, is merely used to mark the termination of the skyline, and always has zero height.

Also, the ground in between any two adjacent buildings should be considered part of the skyline contour.

For instance, the skyline in Figure B should be represented as:`[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`.

Notes:

- The number of buildings in any input list is guaranteed to be in the range `[0, 10000]`.
- The input list is already sorted in ascending order by the left x position `Li`.
- The output list must be sorted by the x position.
- There must be no consecutive horizontal lines of equal height in the output skyline. For instance, `[...[2 3], [4 5], [7 5], [11 5], [12 7]...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: `[...[2 3], [4 5], [12 7], ...]`

```
class Solution {
    public List<List<Integer>> getSkyline(int[][] buildings) {
        List<List<Integer>> res = new ArrayList<List<Integer>>();

        List<int[]> heightPoints = new ArrayList<int[]>();
        for (int[] b : buildings) {
            heightPoints.add(new int[]{b[0], -b[2]});
            heightPoints.add(new int[]{b[1], b[2]});
        }

        Collections.sort(heightPoints, (o1, o2) -> o1[0] == o2[0] ? o1[1] - o2[1] : o1[0] - o2[0]);

        PriorityQueue<Integer> queue = new PriorityQueue<>((o1, o2) -> o2 - o1);
        int pre = 0;
        queue.offer(0);
        for (int[] hp : heightPoints) {
                        if (hp[1] < 0){
                                queue.offer(-hp[1]);
                        } else {
                                queue.remove(hp[1]);
                        }
                        int cur = queue.peek();
                        if(cur != pre){
                                List<Integer> list = new ArrayList<Integer>();
                                list.add(hp[0]);
                                list.add(cur);
                                res.add(list);
                                pre = cur;
                        }
                }
        return res;
    }
}
```

```
class Solution {
    public List<List<Integer>> getSkyline(int[][]
buildings) {
        List<List<Integer>> res = new
ArrayList<List<Integer>>();

        List<int[]> heightPoints = new
ArrayList<int[]>();//装的是每个矩形上面的两个点
        for (int[] b : buildings) {
```

```
            heightPoints.add(new int[]{b[0], -b[2]});//矩形左上角的点
            heightPoints.add(new int[]{b[1], b[2]});//矩形右上角的点，用-，+区分左右
        }
        //把heightPoints中的点按照横坐标从小到大排列，如果横坐标相同，纵坐标小的放在前面，目的是方便从左到右依次去看这些点
        Collections.sort(heightPoints, new Comparator<int[]>() {
            public int compare(int o1[], int o2[]){
                if(o1[0] != o2[0])
                    return o1[0]-o2[0];
                else
                    return o1[1]-o2[1];
            }
            });

        //放当前看到的楼的高度,从大到小排列，也就是最高的楼在最前面
        PriorityQueue<Integer> queue = new PriorityQueue<>((o1, o2) -> (o2 - o1));
        int pre = 0;
        queue.offer(0);
        for (int[] hp : heightPoints) {
            if(hp[1] < 0){//这是一个楼的最左上角的点，也就是这个楼第一次被看见
                queue.offer(-hp[1]);//第一次看见时候给他加入到当前正在比较的楼当中
            } else { //这是一个楼的右上角的点，也就是这个楼第二次被看见
                queue.remove(hp[1]);//第二次看见时候给他移除当前正在比较的楼
            }
```

```
                int cur = queue.peek();//当前最高的楼
                if(cur != pre){//当前最高的楼不是上一次加入结果集的楼;如果cur==pre,说明当前hp[1]的高度没有queue中最高的高或者当前的hp表示的是一个矩形的右上角点
                    List<Integer> list = new ArrayList<Integer>();
                    list.add(hp[0]);
                    list.add(cur);
                    res.add(list);
                    pre = cur;
                }
            }
            return res;
        }
}
```

```
/**
 * // This is the interface that allows for creating nested lists.
 * // You should not implement it, or speculate about its implementation
 * public interface NestedInteger {
 *    // Constructor initializes an empty nested list.
 *    public NestedInteger();
 *
 *    // Constructor initializes a single integer.
 *    public NestedInteger(int value);
 *
 *    // @return true if this NestedInteger holds a single integer, rather than a nested list.
 *    public boolean isInteger();
 *
 *    // @return the single integer that this NestedInteger holds, if it holds a single integer
 *    // Return null if this NestedInteger holds a nested list
 *    public Integer getInteger();
 *
 *    // Set this NestedInteger to hold a single integer.
 *    public void setInteger(int value);
 *
 *    // Set this NestedInteger to hold a nested list and adds a nested integer to it.
 *    public void add(NestedInteger ni);
 *
 *    // @return the nested list that this NestedInteger holds, if it holds a nested list
 *    // Return null if this NestedInteger holds a single integer
 *    public List<NestedInteger> getList();
 * }
 */
class Solution {
  public int depthSum(List<NestedInteger> nestedList) {
    return h(nestedList, 1);
  }

  private int h(List<NestedInteger> nl, int w) {
    int sum = 0;
    for (NestedInteger ni : nl) {
      if (ni.isInteger()) {
        sum += w * ni.getInteger();
      } else if (ni.getList() != null) {
        sum += h(ni.getList(), w+1);
```

```
        }
      }
      return sum;
    }
}
```

Insert into a BST

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public TreeNode insertIntoBST(TreeNode root, int val) {
        if (root == null) {
            return new TreeNode(val);
        }
        if (root.val > val) {
            root.left = insertIntoBST(root.left, val);
        } else {
            root.right = insertIntoBST(root.right, val);
        }

        return root;
    }
}
```

Binary Search

```
class Solution {
  public int search(int[] nums, int target) {
    int pivot, left = 0, right = nums.length - 1;
    while (left <= right) {
      pivot = (left + right) / 2;
      if (nums[pivot] == target)
        return pivot;
      else {
        if (target < nums[pivot])
          right = pivot - 1;
        else
          left = pivot + 1;
      }
    }
    return -1;
  }
}
```

Given two strings ***s*** and ***t***, determine if they are isomorphic.

Two strings are isomorphic if the characters in ***s*** can be replaced to get ***t***.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character but a character may map to itself.

Example 1:

```
Input: s = "egg", t = "add"
Output: true
```

Example 2:

```
Input: s = "foo", t = "bar"
Output: false
```

```
class Solution {
  public boolean isIsomorphic(String s, String t) {
    if (s.length() != t.length()) return false;
    HashMap<Character, Integer> sMap = new HashMap<>();
    HashMap<Character, Integer> tMap = new HashMap<>();

    for (int i = 0; i < s.length(); i++){
      if (sMap.containsKey(s.charAt(i)) == tMap.containsKey(t.charAt(i))){
        if (!sMap.containsKey(s.charAt(i))){
          sMap.put(s.charAt(i), i);
          tMap.put(t.charAt(i), i);
        } else {
          if (sMap.get(s.charAt(i)) != tMap.get(t.charAt(i)))
            return false;
        }
      } else
        return false;
    }
    return true;
  }
}
```

The **Fibonacci numbers**, commonly denoted `F(n)` form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from `0` and `1`

```
class Solution {
  public int fib(int N) {
    if (N == 0) {
      return 0;
    }
    if (N == 1) {
      return 1;
    }
    return h(1, 0, N-2);
  }

  private int h(int a, int b, int cnt) {
    if (cnt == 0) {
      return a + b;
    }
    return h(a + b, a, cnt-1);
  }
}
```

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree `[1,2,2,3,4,4,3]` is symmetric:

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

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isSymmetric(TreeNode root) {
        if (root == null) {
            return true;
        }
        return h(root.left, root.right);
    }

    private boolean h(TreeNode n1, TreeNode n2) {
        if (n1 == null && n2 == null) {
            return true;
        }
        if (n1 != null && n2 != null) {
            return n1.val == n2.val && h(n1.left, n2.right) && h(n2.left, n1.right);
        }
        return false;
    }
}
```

Given an array of integers, 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear **twice** and others appear **once**.

Find all the elements that appear **twice** in this array.

Could you do it without extra space and in O(*n*) runtime?

Example:

Input:
```
[4,3,2,7,8,2,3,1]
```

Output:
```
[2,3]
```

```
// when find a number i, flip the number at position i-1 to negative.
    // if the number at position i-1 is already negative, i is the number that occurs twice.
```

```
class Solution {
  public List<Integer> findDuplicates(int[] nums) {
    List<Integer> res = new ArrayList<>();
    for (int i = 0; i < nums.length; i++) {
      int idx = Math.abs(nums[i]) - 1;
      if (nums[idx] < 0) {
        res.add(idx+1);
      } else {
        nums[idx] = -nums[idx];
      }
    }
    return res;
  }
}
```

Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.

Find all the elements of [1, *n*] inclusive that do not appear in this array.

Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.

Example:

Input:
```
[4,3,2,7,8,2,3,1]
```

Output:
```
[5,6]
```

```
class Solution {
  // The basic idea is that we iterate through the input array
  // and mark elements as negative using nums[nums[i] -1] = -nums[nums[i]-1].
  // In this way all the numbers that we have seen will be marked as negative.
  // In the second iteration, if a value is not marked as negative,
  // it implies we have never seen that index before, so just add it to the return list.
  public List<Integer> findDisappearedNumbers(int[] nums) {
    List<Integer> list = new ArrayList<>();

    for (int i = 0; i < nums.length; i++){
      int index = Math.abs(nums[i]) - 1;
      if (nums[index] > 0) {
        nums[index] = -nums[index];
      }
    }

    for (int i = 0; i < nums.length; i++){
      if (nums[i] > 0) list.add(i+1);
    }

    return list;
  }
}
```

A website domain like "discuss.leetcode.com" consists of various subdomains. At the top level, we have "com", at the next level, we have "leetcode.com", and at the lowest level, "discuss.leetcode.com". When we visit a domain like "discuss.leetcode.com", we will also visit the parent domains "leetcode.com" and "com" implicitly.

Now, call a "count-paired domain" to be a count (representing the number of visits this domain received), followed by a space, followed by the address. An example of a count-paired domain might be "9001 discuss.leetcode.com".

We are given a list `cpdomains` of count-paired domains. We would like a list of count-paired domains, (in the same format as the input, and in any order), that explicitly counts the number of visits to each subdomain.

Example 1:
Input:
`["9001 discuss.leetcode.com"]`
Output:
`["9001 discuss.leetcode.com", "9001 leetcode.com", "9001 com"]`
Explanation:
We only have one website domain: "discuss.leetcode.com". As discussed above, the subdomain "leetcode.com" and "com" will also be visited. So they will all be visited 9001 times.

```
class Solution {
  public List<String> subdomainVisits(String[] cpdomains) {
    Map<String, Integer> mp = new HashMap<>();
    for (String str : cpdomains) {
      int idx = str.indexOf(" ");
      String f = str.substring(0, idx);
      String sec = str.substring(idx+1);
      int val = Integer.valueOf(f);

      mp.put(sec, mp.getOrDefault(sec, 0) + val);
      int i = 0;
      while (i < sec.length()) {
        if (sec.charAt(i) == '.') {
          String tmp = sec.substring(i+1);
          mp.put(tmp, mp.getOrDefault(tmp, 0) + val);
        }
        i++;
      }
    }
    List<String> res = new ArrayList<>();
    for (String key : mp.keySet()) {
      res.add(mp.get(key) + " " + key);
    }
    return res;
  }
}
```