# Search in 2D matrix

**class** **Point** {

**int** x;

**int** y;

}

Point **isInMatrix**(**int**[][] matrix, **int** target){

**int** row = matrix.length;

**int** column = matrix[0].length;

**int** r = 0;

**int** c = column - 1;

**while** (r < row && c >= 0){

**if** (matrix[r][c] == target){

**return** **new** Point(r,c);

}

**if** (matrix[r][c] > target){

c--;

} **else** {

r++;

}

}

**return** **new** Point(-1,-1);

}

# Overlap Rectangle

*// Overlap Rectangle*

*// Rect 1: top-left(A, B), bottom-right(C, D)*

*// Rect 2: top-left(E, F), bottom-right(G, H)*

**public** **int** **computeArea**(**int** A, **int** B, **int** C, **int** D, **int** E, **int** F, **int** G, **int** H) {

**int** innerL = Math.max(A, E);

**int** innerR = Math.max(innerL, Math.min(C, G));

**int** innerT = Math.max(B, F);

**int** innerB = Math.max(innerT, Math.min(D, H));

**return** (C - A) \* (B - D) - (innerR - innerL) \* (innerT - innerB) + (G -E) \* (F - H);

}

# 20. Valid Parentheses

**public** boolean **isValid**(String s) {

Stack<Character> stack = **new** Stack<Character>();

**for** (**char** c : s.toCharArray()) {

**if** (c == '(')

stack.push(')');

**else** **if** (c == '{')

stack.push('}');

**else** **if** (c == '[')

stack.push(']');

**else** **if** (stack.isEmpty() || stack.pop() != c)

**return** false;

}

**return** stack.isEmpty();

}

# 438. Find All Anagrams in a String

**public** **List**<Integer> findAnagrams(String s, String p) {

**List**<Integer> **list** = **new** ArrayList<>();

**if** (s == **null** || s.length() == 0 || p == **null** || p.length() == 0) **return** **list**;

int[] hash = **new** int[256]; *//character hash*

*//record each character in p to hash*

**for** (char c : p.toCharArray()) {

hash[c]++;

}

*//two points, initialize count to p's length*

int left = 0, right = 0, count = p.length();

**while** (right < s.length()) {

*//move right everytime, if the character exists in p's hash, decrease the count*

*//current hash value >= 1 means the character is existing in p*

**if** (hash[s.charAt(right++)]-- >= 1) count--;

*//when the count is down to 0, means we found the right anagram*

*//then add window's left to result list*

**if** (count == 0) **list**.add(left);

*//if we find the window's size equals to p, then we have to move left (narrow the window) to find the new match window*

*//++ to reset the hash because we kicked out the left*

*//only increase the count if the character is in p*

*//the count >= 0 indicate it was original in the hash, cuz it won't go below 0*

**if** (right - left == p.length() && hash[s.charAt(left++)]++ >= 0) count++;

}

**return** **list**;

}

# //3. Find the min steps of maze

//意思是说有一个M\*N的maze，0代表可以通过，1代表不可以通过，然后给你一个出口（x,y），找从（0,0）到出口的最少steps，如果找不到path就返回-1

static int minlen = Integer.MAX\_VALUE;

public static void bfsfind(int[][] mazeinit) {

if(mazeinit == null || mazeinit.length == 0) return;

int row = mazeinit.length;

int col = mazeinit[0].length;

boolean[][] visited = new boolean[row][col];

int[] direcx = {-1,0,1,0};

int[] direcy = {0,-1,0,1};

bfs(mazeinit, visited, 0, 0, 0, direcx, direcy);

return;

}

public static void bfs(int[][] mazeinit, boolean[][] visited, int row, int col, int pathlen, int[] direcx, int[] direcy) {

if(!isValid(mazeinit, visited, row, col)) return;

if(row == mazeinit.length-1 && col == mazeinit[0].length-1) {

minlen = Math.min(minlen, pathlen);

return;

}

for(int i = 0; i < 4; i++) {

visited[row][col] = true;

bfs(mazeinit, visited, row+direcx[i], col+direcy[i], pathlen+1, direcx, direcy);

visited[row][col] = false;

}

}

public static boolean isValid(int[][] mazeinit, boolean[][] visited, int row, int col) {

int m = mazeinit.length;

int n = mazeinit[0].length;

return row >=0 && row < m && col >= 0 && col < n && mazeinit[row][col] == 0 && visited[row][col] == false;

}

public static int bdfsfind(int[][] mazeinit, Point src, Point tar) {

if(mazeinit == null || mazeinit.length == 0) return 0;

int row = mazeinit.length;

int col = mazeinit[0].length;

boolean[][] visited = new boolean[row][col];

if( !isBfsValid(mazeinit, visited, src.x, src.y) ) return 0;

if( !isBfsValid(mazeinit, visited, tar.x, tar.y) ) return 0;

int[] direcx = {-1,0,1,0};

int[] direcy = {0,-1,0,1};

PointNode root = new PointNode(new Point(src.x, src.y), 0);

visited[src.x][src.y] = true;

Queue<PointNode> queue = new LinkedList<PointNode>();

queue.add(root);

while(!queue.isEmpty()) {

int size = queue.size();

for(int i = 0; i < size; i++) {

PointNode temp = queue.poll();

//find a matched solution

if(temp.pt.x == tar.x && temp.pt.y == tar.y) return temp.dist;

for(int j = 0; j < 4; j++) {

// tranverse point is valid

if(isBfsValid(mazeinit, visited, temp.pt.x+direcx[j], temp.pt.y+direcy[j])) {

int rowtemp = temp.pt.x+direcx[j];

int coltemp = temp.pt.y+direcy[j];

visited[rowtemp][coltemp] = true;

PointNode qadd = new PointNode(new Point(rowtemp, coltemp), temp.dist+1);

queue.offer(qadd);

}

}

}

}

return 0;

}

public static boolean isBfsValid(int[][] mazeinit, boolean[][] visited, int row, int col) {

int m = mazeinit.length;

int n = mazeinit[0].length;

return row >= 0 && row < m && col >= 0 && col < n

&& mazeinit[row][col] == 0 && visited[row][col] == false;

}

class Point {

int x;

int y;

Point(int x, int y) {

this.x = x;

this.y = y;

}

}

class PointNode {

Point pt;

int dist;

PointNode(Point pt, int dist) {

//this.x = x;

this.pt = pt;

this.dist = dist;

}

}

# BallScore

// ball score

public static int ballscore(String[] strarr) {

int score = 0;

if(strarr == null || strarr.length == 0) return score;

Stack<Integer> stack = new Stack<Integer>();

for(int i = 0; i < strarr.length; i++) {

char ch = strarr[i].charAt(0);

if(ch != 'X' && ch != '+' && ch != 'Z') {

int ps = Integer.parseInt(strarr[i]);

stack.push(ps);

score += ps;

} else {

switch(ch) {

case 'Z' :

if(!stack.isEmpty()) {

int curr = stack.pop();

score -= curr;

}

break;

case 'X' :

if(!stack.isEmpty()) {

int curr = stack.peek();

score += 2\*curr;

stack.push(2\*curr);

}

break;

case '+' :

if(!stack.isEmpty()) {

int i1 = stack.pop();

if(!stack.isEmpty()) {

int i2 = stack.pop();

score += i1;

score += i2;

stack.push(i2);

stack.push(i1);

stack.push(i1+i2);

} else {

score += i1;

stack.push(i1);

stack.push(i1);

}

}

break;

default :

break;

} // end switch

} // end first if-else

} // end for

return score;

}

# find-min-bst

// find distance between two node of BST

// http://algorithms.tutorialhorizon.com/find-the-distance-between-two-nodes-of-a-binary-tree/ （find distance between two node of Binary Tree）

public static TreeNode LCABST(TreeNode root, TreeNode p, TreeNode q) {

if(root == null || p == root || q == root) return root;

if((root.val > p.val && root.val < q.val) ||

(root.val < p.val && root.val > q.val)) {

return root;

} else if(root.val > p.val && root.val > q.val) {

return LCABST(root.left, p, q);

} else if(root.val < p.val && root.val < q.val) {

return LCABST(root.right, p, q);

} else {

return null;

}

}

public static int findlen(TreeNode root, int n) {

return finddistancebst(root, n)-1;

}

public static int finddistancebst(TreeNode root, int n) {

if(root == null) return 0;

int x = 0;

if(root.val == n) return x+1;

if(root.val > n) {

x = finddistancebst(root.left, n);

} else {

x = finddistancebst(root.right, n);

}

if(x > 0)

return x+1;

else return 0;

}

public static int bstdistance(int[] values, int n, int node1, int node2) {

// create the tree use values array and n

if(values == null || values.length == 0) return 0;

TreeNode root = new TreeNode(values[0]);

for(int i = 1; i < values.length; i++) {

createbst(root, values[i]);

}

//TreeNode root = createbst(values); // this is a func

int len1 = findlen(root, node1);

if(len1==-1) return 0;

int len2 = findlen(root, node2);

if(len2==-1) return 0;

int lcaval = LCABST1(root, node1, node2).val;

int lenmid = findlen(root, lcaval);

return len1+len2-2\*lenmid;

}

public static void createbst(TreeNode root, int value) {

if(value < root.val) {

if(root.left == null) {

root.left = new TreeNode(value);

} else {

createbst(root.left, value);

}

} else {

if(root.right == null) {

root.right = new TreeNode(value);

} else {

createbst(root.right, value);

}

}

}

public static TreeNode LCABST1(TreeNode root, int i1, int i2) {

if(root == null || i1 == root.val || i2 == root.val) return root;

if((root.val > i1 && root.val < i2) ||

(root.val < i1 && root.val > i2)) {

return root;

} else if(root.val > i1 && root.val > i2) {

return LCABST1(root.left, i1, i2);

} else if(root.val < i1 && root.val < i2) {

return LCABST1(root.right, i1, i2);

} else {

return null;

}

}

# common\_manager

// LCA of n-ary tree

//Employee firstEmployee = ceo.getReports().get(0).getReports().get(0).getReports().get(0).getReports().get(0);

//Employee secondEmployee = ceo.getReports().get(0).getReports().get(0).getReports().get(0).getReports().get(1);

public static Employee commonmanager(Employee ceo, Employee firstEmployee, Employee secondEmployee) {

Stack<Employee> first = new Stack<Employee>();

Stack<Employee> second = new Stack<Employee>();

Employee root = ceo;

dfs(root, firstEmployee, first);

dfs(root, secondEmployee, second);

if( (!first.isEmpty() && first.peek().getId() == firstEmployee.getId())

&& (!second.isEmpty() && second.peek().getId() == secondEmployee.getId())) {

int size1 = first.size();

int size2 = second.size();

int diff = Math.abs(size1-size2);

if(size1 > size2) {

// first stack pop element until size equals the second

moveUp(first, diff);

} else {

moveUp(second, diff);

}

while(first.peek().getId() != second.peek().getId()) {

first.pop();

second.pop();

}

if(first.size() > 0) {

return first.pop();

}

}

return null;

}

public static void moveUp(Stack<Employee> stack, int diff) {

while(diff > 0 && !stack.isEmpty()) {

stack.pop();

diff--;

}

}

public static boolean dfs(Employee root, Employee curr, Stack<Employee> stack) {

stack.push(root);

if(root.getId() == curr.getId()) {

return true;

}

for(Employee em : root.getReports()) {

boolean res = dfs(em, curr, stack);

if(res == true) {

return true;

}

}

stack.pop();

return false;

}

public static Employee commonmanageroftree(Employee ceo, Employee firstEmployee, Employee secondEmployee) {

if(ceo==null) return ceo;

if(ceo==firstEmployee) return ceo;

if(ceo==secondEmployee) return ceo;

boolean isEmployee1 = false;

boolean isEmployee2 = false;

for(Employee em : ceo.getReports()) {

Employee res = commonmanageroftree(em, firstEmployee, secondEmployee);

if(res == firstEmployee) {

isEmployee1 = true;

} else if(res == secondEmployee) {

isEmployee2 = true;

} else if(res != null) {

return res;

}

}

if(isEmployee1==true && isEmployee2==true) {

return ceo;

} else if(isEmployee1==true) {

return firstEmployee;

} else if(isEmployee2==true) {

return secondEmployee;

}

return null;

}

# Movie Network

private static void bfsSearchMovieNetwork(Movie movie, PriorityQueue<Movie> queue) {

for (Movie m : movie.getSimilarMovies()) {

if (!queue.contains(m)) {

queue.offer(m);.

bfsSearchMovieNetwork(m, queue);

}

}

}

PriorityQueue<Movie> queue = new PriorityQueue<>(new Comparator<Movie>() {

@Override

public int compare(Movie m1, Movie m2) {

return new Float(m2.getRating()).compareTo(m1.getRating());

}

});

private static void bfsSearchMovieNetwork(Movie movie, PriorityQueue<Movie> queue) {

Queue que = new LinkedList;

que.offer(movie);

while(!que.isEmpty()) {

int size = que.size();

for(int i = 0; i < size; i++) {

Movie temp = que.poll();

for (Movie m : temp.getSimilarMovies()) {

if(queue.size() < k) {

queue.offer(m);

} else if(queue.peek().ratings < m.ratings) {

queue.poll();

queue.offer(m);

}

}

}

}

}