# 

[**Default interview problem**](#_ujepbmiwebbe) **6**

[Longest Palindrome](#_6eaqwxdojo8t) 6

[Implement strStr()](#_mlyiydhyd23y) 6

[Valid Palindrome](#_1ctneur4vmkn) 7

[Longest Palindromic Substring](#_121pfisj4wuf) 8

[LRU](#_cquky4c9roz) 9

[常规写法](#_hl1o28mk4gc) 9

[LinkedHashMap](#_iqxnl97cptgm) 10

[**BFS**](#_28qm9vcb3cid) **11**

[Number of Islands](#_7u47yrt47hmo) 11

[BFS](#_eva2s4fdxi29) 11

[Union Find](#_3pvwx9f4mqi4) 13

[Binary Tree Level Order Traversal](#_gajtkqe84ny6) 14

[Course Schedule](#_egzr4e8agt32) 15

[Course Schedule II](#_u7tc6sru0a0r) 17

[Knight Shortest Path](#_atl7p0hwnqsw) 18

[Sequence Reconstruction](#_d7uvnlwps6od) 19

[Clone Graph](#_11zrtv5prd44) 20

[Topological Sorting](#_vy8fr3gx5svv) 21

[Serialize and Deserialize Binary Tree](#_l102w51e0t06) 22

[推荐方法](#_jmjd6duq6e1q) 22

[Word Ladder](#_b57kze5x6zw7) 24

[**Binary Search**](#_m5vtg1o8lgse) **25**

[Last Position of Target](#_m2jnk8ja95s1) 25

[Maximum Number in Mountain Sequence](#_6g1j45ujfmg7) 25

[Find K Closest Elements](#_iyw81yulypn2) 26

[Search in a Big Sorted Array](#_qhuh2rwm59vy) 27

[Fast Power](#_4qibmzi1sr24) 28

[Find Peak Element](#_a46b0zbig96d) 28

[First Bad Version](#_a7bcw1s53qhq) 29

[Search in Rotated Sorted Array](#_3pkp3qsc27rd) 29

[**Binary Tree Divide & Conquer**](#_6d73u227w07c) **31**

[Closest Binary Search Tree Value](#_f5baiksxqo84) 31

[Vertical Order Traversal of a Binary Tree](#_64tci5gg8wf6) 32

[Minimum Subtree](#_sb04gsfvag0q) 33

[Binary Tree Paths](#_4h136uom7m8e) 34

[Flatten Binary Tree to Linked List](#_ueos06kbw5h6) 34

[Balanced Binary Tree](#_ws0hvil4zn93) 35

[Kth Smallest Element in a BST](#_ryki1kn1u26d) 36

[Lowest Common Ancestor III](#_bhtd7lpxsgdi) 37

[Validate Binary Search Tree](#_uwmcr54x5axf) 38

[Closest Binary Search Tree Value II](#_7qvm4by37rce) 39

[Binary Search Tree Iterator](#_l1mntp2ev1ke) 41

[**Two Pointers**](#_lfccab8bjoog) **42**

[Middle of Linked List](#_90fnq2orza52) 42

[Two Sum III - Data structure design](#_co22tq99gyms) 43

[Move Zeroes](#_8ux1wekptmmu) 44

[Remove Duplicate Numbers in Array](#_epvf2aie7kba) 44

[Sort Integers II](#_lteax4j6ctus) 45

[Merge Sort](#_mwrvwwc9pv1s) 45

[Quick Sort](#_5w4kemmw314f) 46

[Heap Sort](#_94vrc92d43b6) 47

[Two Sum II - Input array is sorted](#_kzq2s8o1jqss) 49

[Sort Colors II](#_o2x5sbz57oey) 49

[3Sum](#_1uaodsw6uvf) 51

[Partition Array](#_eacm22mhiq4f) 52

[Kth Largest Element](#_biahsgvx6fij) 53

[**DFS**](#_f05e72yxvxso) **53**

[Split String](#_7c29lo8hy5x6) 54

[Letter Combinations of a Phone Number](#_xkvwgoxy76qe) 55

[Combination Sum II](#_wt3zq0f3jczb) 56

[Combination Sum](#_ifnhnc5ugca1) 57

[N-Queens](#_63gxtdvylj1v) 58

[Subsets II](#_36gj5v3qs9o8) 59

[Subsets](#_na2bhz9jjrqo) 60

[Permutations](#_ueduskbylqf7) 61

[Word Pattern II](#_es0id7mql9at) 62

[Word Ladder II](#_a4cevnjsku) 63

[**Hash & Heap**](#_rpxysmw1g61n) **65**

[Moving Average from Data Stream](#_fz7otlqcprlt) 65

[Implement Stack by Two Queues](#_p8khr5pxv95a) 66

[First Unique Character in a String](#_xn3tnlourup8) 67

[Insert Delete GetRandom O(1)](#_5v4x6g4nrmgq) 68

[K Closest Points](#_10eas031qnem) 69

[Top k Largest Numbers](#_d2mkwgv602ue) 70

[Merge K Sorted Lists](#_lox16meoa123) 71

[Implement Queue by Two Stacks](#_pe1lr2m0a7bi) 72

[Ugly Number II](#_elei586niyaf) 73

[**Memonization Search & Dynamic Programming**](#_obj8zvtzpule) **74**

[Word Break III](#_fch5z8selh4x) 74

[Triangle](#_9ol78cdjp61f) 75

[Word Break](#_ciwns3cp96xp) 76

[Word Break II](#_98fv99e4k4ck) 77

[Wildcard Matching](#_tvwvdua71a7a) 78

[Regular Expression Matching](#_b7gdlfh1ro6r) 79

[**4 key points DP and coordinate DP**](#_bcrvnv9k70tx) **81**

[Unique Paths II](#_4vak4s2ujvqh) 81

[Unique Paths](#_ez4seikbche9) 82

[Climbing Stairs](#_61j6xbg93a65) 83

[Largest Divisible Subset](#_d8uet0y1qs2h) 84

[Knight Shortest Path](#_lfar8o90sncs) 85

[Longest Increasing Subsequence](#_wfsqb3nlwp1h) 86

[Russian Doll Envelopes](#_4awt3tlpi695) 87

[Merge Two Sorted Interval Lists](#_6zs0j0h9bjsf) 88

[Intersection of Two Arrays](#_iczupe2gu1c8) 89

[Subarray Sum](#_s18sozt64gtc) 90

[Merge Sorted Array](#_5f8r37ij8l0a) 91

[Maximum Subarray](#_7eyb2joeoe9e) 92

[Maximum Submatrix](#_u44jfjfoo5k9) 92

[Range Sum Query - Mutable](#_nk1wwsc7tee) 93

[Sparse Matrix Multiplication](#_b9lnaoml9ao) 94

[Merge K Sorted Interval Lists](#_u2i6wnr2w9lm) 95

[Merge K Sorted Arrays](#_enjrlalm4268) 96

[Median of K Sorted Arrays](#_xf3gguh9zcbq) 98

[Median of two Sorted Arrays](#_umozt8vorcdy) 100

[**Homework**](#_w9wskh20kg64) **100**

[Add Two Numbers II](#_gdtlf0oiv2na) 100

[Search a 2D Matrix](#_2klsnn2o4a3f) 102

[The Maze](#_bowmy10l402) 102

[Binary Tree Zigzag Level Order Traversal](#_rmymyhloclfb) 104

[01 Matrix](#_ufgsl9jnvrg0) 104

[Binary Tree Right Side View](#_8r3ocg267hk1) 105

[The Maze II](#_p6nw8c1kuhm4) 106

[Post order traversal](#_qmt71xd6tz81) 107

[Jump Game](#_qmt71xd6tz81) 108

[DFS:](#_fr176gazfg10) 108

[DP:](#_6u2d3qwte2mf) 108

[Greedy:](#_oxpno0bgv2uc) 108

[Pow(x, n)](#_qmt71xd6tz81) 109

[位运算](#_v0i3x2kqfr07) 109

[普通除法](#_xp3fo6bsjyk1) 109

[递归](#_40hdntcvjxv8) 110

[Symmetric Tree / Same Tree](#_qmt71xd6tz81) 110

[Permutations II](#_qmt71xd6tz81) 111

[Remove Linked List Elements](#_qjndsumjkftl) 112

[Remove Duplicates from Sorted List](#_qjndsumjkftl) 112

[Remove Duplicates from Sorted List II](#_qjndsumjkftl) 113

[Intersection of Two Arrays](#_qjndsumjkftl) 113

[Intersection of Two Arrays II](#_qjndsumjkftl) 114

[Next Greater Node In Linked List](#_qjndsumjkftl) 114

[Reverse Nodes in k-Group](#_nk3uewopexaa) 115

[LFU](#_pmrnkxms5cem) 115

[构树](#_pmrnkxms5cem) 116

[Number of Airplanes in the Sky](#_pmrnkxms5cem) 119

[Meeting Rooms](#_pmrnkxms5cem) 120

[Meeting Rooms II](#_pmrnkxms5cem) 121

[Factor Combinations](#_pmrnkxms5cem) 121

[Binary Tree Paths](#_pmrnkxms5cem) 122

[Inorder Successor in BST](#_pmrnkxms5cem) 123

[Game of Life](#_pmrnkxms5cem) 123

[Increasing Order Search Tree](#_pmrnkxms5cem) 124

[Non-recursive](#_feah4b9wjs4r) 124

[Recursive](#_ioky3fkylsu6) 124

[Symmetric Tree](#_pmrnkxms5cem) 125

[Recursive](#_2mt3zvgemvr1) 125

[Non-Recursive](#_ify9dfxozb8w) 126

[Same Tree](#_pacz7vk3xswz) 127

[Non-Recursive](#_urpakg7ocsh5) 127

[Recursive](#_bilv4l7s9woy) 127

[Quick Select](#_ypgg41n3ijk9) 128

[Union Find](#_kgzlwuajsbl8) 129

[Number of Matching Subsequences](#_nzm13jdfm7nr) 129

[Flood Fill](#_27u8544eu2f2) 130

[Keys and Rooms](#_hawu7zlimg3r) 130

[H-Index](#_daw7e1hppu84) 131

[Find the Celebrity](#_vl69f72r40df) 131

[Range Sum Query - Immutable](#_uol6n1qsbskj) 132

[Maximum Size Subarray Sum Equals k](#_zaba4ymogm41) 132

[Verify Preorder Serialization of a Binary Tree](#_8g4a49frudn) 133

[Largest BST Subtree](#_7amum9v9od1a) 133

[Increasing Triplet Subsequence](#_xl31snt8j1kb) 134

[Top K Frequent Elements](#_wijz96bmh3i9) 134

[Surrounded Regions](#_fwavjvheg4s7) 135

[Kth Smallest Element in a Sorted Matrix](#_fwzouli0jbol) 136

[答案二分](#_tedjal2kb6ja) 136

[Priority Queue](#_x4xgdu212noa) 137

[Implement Trie (Prefix Tree)](#_4shbixjhumhm) 137

[Design Twitter](#_4shbixjhumhm) 139

# 

# 

# Default interview problem

## Longest Palindrome

|  |
| --- |
| class Solution {  public int longestPalindrome(String s) {  int[] count = new int[256];  for (char c : s.toCharArray()) {  count[c]++;  }  int total = 0;  int odd = 0;  for (int c : count) {  if (c % 2 == 0) {  total += c;  } else {  total += c - 1;  odd = 1;  }  }  return total + odd;  } } |

## Implement strStr()

|  |
| --- |
| public class Solution {  public int strStr(String haystack, String needle) {  for (int i = 0; ; i++) {  for (int j = 0; ; j++) {  if (j == needle.length()) {  return i;  }  if (i + j == haystack.length()) {  return -1;  }  if (needle.charAt(j) != haystack.charAt(i + j)) {  break;  }  }  }  } } |

## Valid Palindrome

|  |
| --- |
| class Solution {  public boolean isPalindrome(String s) {  if (s == null) {  return true;  }    int i = 0;  int j = s.length() - 1;    while (i < j) {  char left = s.charAt(i);  char right = s.charAt(j);    if (!isAlphaNumeric(left)) {  i++;  } else if (!isAlphaNumeric(right)) {  j--;   } else if (Character.toLowerCase(left) != Character.toLowerCase(right)) {  return false;  } else {  i++;  j--;  }  }  return true;  }    private boolean isAlphaNumeric(char c) {  return (c >= '0' && c <= '9') || (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');  } } |

## Longest Palindromic Substring

|  |
| --- |
| class Solution {  public String longestPalindrome(String s) {  char[] chars = convertString(s);  int[] counts = new int[chars.length];  int center = 0;  int rightIndex = 0;  for (int i = 1; i < chars.length - 1; i++) {  int leftIndex = 2 \* center - rightIndex;  counts[i] = rightIndex > i ? Math.min(rightIndex - i, counts[leftIndex]) : 0;  while (chars[i + counts[i] + 1] == chars[i - 1 - counts[i]]) {  counts[i]++;  }  if (i + counts[i] > rightIndex) {  center = i;  rightIndex = i + counts[i];  }  }  int maxLen = 0;  for (int i = 1; i < counts.length - 1; i++) {  if (counts[i] > maxLen) {  maxLen = counts[i];  center = i;  }  }  return s.substring((center - 1 - maxLen) / 2, (center - 1 - maxLen) / 2 + maxLen);  }    private char[] convertString(String s) {  int len = s.length();  char[] result = new char[len \* 2 + 3];  result[0] = '^';  for (int i = 1, j = 0; i < result.length - 1; i++) {  result[i] = (i & 1) == 1 ? '#' : s.charAt(j++);  }  result[result.length - 1] = '$';  return result;  } } |

## LRU

### 常规写法

|  |
| --- |
| public class LRUCache {  private class Node{  Node prev;  Node next;  int key;  int value;   public Node(int key, int value) {  this.key = key;  this.value = value;  this.prev = null;  this.next = null;  }  }   private int capacity;  private HashMap<Integer, Node> hs = new HashMap<Integer, Node>();  private Node head = new Node(-1, -1);  private Node tail = new Node(-1, -1);   public LRUCache(int capacity) {  this.capacity = capacity;  tail.prev = head;  head.next = tail;  }   public int get(int key) {  if( !hs.containsKey(key)) { //key找不到  return -1;  }   // remove current  Node current = hs.get(key);  current.prev.next = current.next;  current.next.prev = current.prev;   // move current to tail  move\_to\_tail(current); //每次get，使用次数+1，最近使用，放于尾部   return hs.get(key).value;  }   public void set(int key, int value) { //数据放入缓存  // get 这个方法会把key挪到最末端，因此，不需要再调用 move\_to\_tail  if (get(key) != -1) {  hs.get(key).value = value;  return;  }   if (hs.size() == capacity) { //超出缓存上限  hs.remove(head.next.key); //删除头部数据  head.next = head.next.next;  head.next.prev = head;  }   Node insert = new Node(key, value); //新建节点  hs.put(key, insert);  move\_to\_tail(insert); //放于尾部  }   private void move\_to\_tail(Node current) { //移动数据至尾部  current.prev = tail.prev;  tail.prev = current;  current.prev.next = current;  current.next = tail;  } } |

### LinkedHashMap

|  |
| --- |
| class LRUCache {  private int capacity;  private Map<Integer, Integer> map;  public LRUCache(int capacity) {  this.capacity = capacity;  map = new LinkedHashMap<>();  }    public int get(int key) {  if (!map.containsKey(key)) {  return -1;  }  int val = map.get(key);  map.remove(key);  map.put(key, val);  return val;  }    public void put(int key, int value) {  if (map.containsKey(key)) {  map.put(key, value);  get(key);  return;  }  map.put(key, value);  if (map.size() > capacity) {  int victim = map.entrySet().iterator().next().getKey();  map.remove(victim);  }  } } |

# BFS

## Number of Islands

### BFS

|  |
| --- |
| // version 2: BFS class Coordinate {  int x, y;  public Coordinate(int x, int y) {  this.x = x;  this.y = y;  } }  public class Solution {  /\*\*  \* @param grid a boolean 2D matrix  \* @return an integer  \*/  public int numIslands(boolean[][] grid) {  if (grid == null || grid.length == 0 || grid[0].length == 0) {  return 0;  }    int n = grid.length;  int m = grid[0].length;  int islands = 0;    for (int i = 0; i < n; i++) {  for (int j = 0; j < m; j++) {  if (grid[i][j]) {  markByBFS(grid, i, j);  islands++;  }  }  }    return islands;  }    private void markByBFS(boolean[][] grid, int x, int y) {  // magic numbers!  int[] directionX = {0, 1, -1, 0};  int[] directionY = {1, 0, 0, -1};    Queue<Coordinate> queue = new LinkedList<>();    queue.offer(new Coordinate(x, y));  grid[x][y] = false;    while (!queue.isEmpty()) {  Coordinate coor = queue.poll();  for (int i = 0; i < 4; i++) {  Coordinate adj = new Coordinate(  coor.x + directionX[i],  coor.y + directionY[i]  );  if (!inBound(adj, grid)) {  continue;  }  if (grid[adj.x][adj.y]) {  grid[adj.x][adj.y] = false;  queue.offer(adj);  }  }  }  }    private boolean inBound(Coordinate coor, boolean[][] grid) {  int n = grid.length;  int m = grid[0].length;    return coor.x >= 0 && coor.x < n && coor.y >= 0 && coor.y < m;  } } |

### Union Find

|  |
| --- |
| public class Solution {  int[] father;  /\*\*  \* @param grid: a boolean 2D matrix  \* @return: an integer  \*/  public int numIslands(boolean[][] grid) {  if (grid.length == 0) {  return 0;  }  int[] xBias = new int[] {0, 0, -1, 1};  int[] yBias = new int[] {-1, 1, 0, 0};  father = new int[grid.length \* grid[0].length];  Arrays.fill(father, -1);    Queue<int[]> queue = new LinkedList<>();  queue.offer(new int[]{0, 0});  father[0] = grid[0][0] ? 0 : -2;    while (!queue.isEmpty()) {  int[] current = queue.poll();  for (int i = 0; i < xBias.length; i++) {  int newX = current[0] + xBias[i];  int newY = current[1] + yBias[i];  if (!isValid(grid, newX, newY)) {  continue;  }  int oneDPoint = convert2dTo1d(newX, newY, grid[0].length);  if (father[oneDPoint] == -1) {  father[oneDPoint] = -2;  queue.offer(new int[] {newX, newY});  }  if (grid[newX][newY]) {  if (father[oneDPoint] < 0) {  father[oneDPoint] = oneDPoint;  }  if (grid[current[0]][current[1]]) {  connect(convert2dTo1d(current[0], current[1], grid[0].length), oneDPoint);  }  }  }  }    int result = 0;  for (int i = 0; i < father.length; i++) {  if (father[i] == i) {  result++;  }  }  return result;  }    private boolean isValid(boolean[][] grid, int newX, int newY) {  return newX >= 0 && newY >= 0 && newX < grid.length && newY < grid[0].length;  }    private int convert2dTo1d(int x, int y, int colLength) {  return x \* colLength + y;  }    private void connect(int a, int b) {  int rootA = find(a);  int rootB = find(b);  if (rootA != rootB) {  father[rootA] = rootB;  }  }    private int find(int x) {  if (father[x] == x) {  return x;  }  return father[x] = find(father[x]);  } } |

## Binary Tree Level Order Traversal

|  |
| --- |
| /\*\*  \* Definition of TreeNode:  \* public class TreeNode {  \* public int val;  \* public TreeNode left, right;  \* public TreeNode(int val) {  \* this.val = val;  \* this.left = this.right = null;  \* }  \* }  \*/   public class Solution {  /\*  \* @param root: A Tree  \* @return: Level order a list of lists of integer  \*/  public List<List<Integer>> levelOrder(TreeNode root) {  // write your code here  Queue<TreeNode> queue = new LinkedList<>();  List<List<Integer>> result = new ArrayList<>();  if (root == null){  return result;  }  queue.offer(root);  while (!queue.isEmpty()){  int size = queue.size();  List<Integer> level = new ArrayList<>();  for (int i = 0; i < size; i++){  TreeNode vertices = queue.poll();  if (vertices.left != null){  queue.add(vertices.left);  }  if (vertices.right != null){  queue.add(vertices.right);  }  level.add(vertices.val);  }  result.add(level);  }  return result;  } } |

## Course Schedule

|  |
| --- |
| class Solution {  public boolean canFinish(int numCourses, int[][] prerequisites) {  if (prerequisites == null || prerequisites.length == 0 || numCourses == 0) {  return true;  }  int[] inDegree = getInDegree(numCourses, prerequisites);  Map<Integer, List<Integer>> dependencies = getDependencies(prerequisites);  int selectedCourse = 0;    Queue<Integer> queue = new ArrayDeque<>();  for (int i = 0; i < inDegree.length; i++) {  if (inDegree[i] == 0) {  queue.offer(i);  }  }    while (!queue.isEmpty()) {  int course = queue.poll();  selectedCourse++;  if (!dependencies.containsKey(course)) {  continue;  }    for (int related : dependencies.get(course)) {  inDegree[related]--;  if (inDegree[related] == 0) {  queue.offer(related);  }  }  }    return selectedCourse == numCourses;  }    private int[] getInDegree(int numCourses, int[][] prerequisites) {  int[] inDegree = new int[numCourses];  for (int[] prerequisite : prerequisites) {  inDegree[prerequisite[0]]++;  }  return inDegree;  }    private Map<Integer, List<Integer>> getDependencies(int[][] prerequisites) {  Map<Integer, List<Integer>> dependencies = new HashMap<>();  for (int[] prerequisite : prerequisites) {  dependencies.putIfAbsent(prerequisite[1], new ArrayList<>());  dependencies.get(prerequisite[1]).add(prerequisite[0]);  }  return dependencies;  } } |

## Course Schedule II

|  |
| --- |
| public class Solution {  /\*  \* @param numCourses: a total of n courses  \* @param prerequisites: a list of prerequisite pairs  \* @return: the course order  \*/  public int[] findOrder(int numCourses, int[][] prerequisites) {  // write your code here  int[] result = new int[numCourses];  Map<Integer, Set<Integer>> map = new HashMap<>();  int[] degree = new int[numCourses];  for (int [] p : prerequisites) {  if (!map.containsKey(p[1]) || !map.get(p[1]).contains(p[0])) {  degree[p[0]]++;  }  map.putIfAbsent(p[1], new HashSet<>());  map.get(p[1]).add(p[0]);  }    Queue<Integer> queue = new ArrayDeque<>();  for (int i = 0; i < numCourses; i++) {  if (degree[i] == 0) {  queue.offer(i);  }  // System.out.println(degree[i]);  }      int i = 0;  while (!queue.isEmpty()) {  int current = queue.poll();  result[i] = current;  i++;  if (!map.containsKey(current)) {  continue;  }    for (int course : map.get(current)) {  degree[course]--;  if (degree[course] == 0) {  queue.offer(course);  }  }  }  // System.out.println(unSelected);  return i == numCourses ? result : new int[0];  } } |

## Knight Shortest Path

|  |
| --- |
| /\*\*  \* Definition for a point.  \* class Point {  \* int x;  \* int y;  \* Point() { x = 0; y = 0; }  \* Point(int a, int b) { x = a; y = b; }  \* }  \*/  public class Solution {  /\*\*  \* @param grid: a chessboard included 0 (false) and 1 (true)  \* @param source: a point  \* @param destination: a point  \* @return: the shortest path   \*/    int[] dx = {1, 1, -1, -1, 2, 2, -2, -2};  int[] dy = {2, -2, 2, -2, 1, -1, 1, -1};  public int shortestPath(boolean[][] grid, Point source, Point destination) {      Queue<Point> queue = new LinkedList<>();  boolean[][] v = new boolean [grid.length + 1][grid[0].length + 1];  queue.offer(source);  int res =0;  while(!queue.isEmpty()){    int size = queue.size();   res++;  for(int i = 0; i < size; i++){  Point cur = queue.poll();  int x = cur.x, y = cur.y;    for(int k = 0; k < 8; k ++){  Point nextPoint = new Point(  x + dx[k],  y + dy[k]);  if(nextPoint.x < 0 || nextPoint.x >= grid.length || nextPoint.y<0 || nextPoint.y >= grid[0].length || grid[nextPoint.x][nextPoint.y] || v[nextPoint.x][nextPoint.y]){  continue;  }   if(nextPoint.x == destination.x && nextPoint.y == destination.y){  return res;  }  queue.offer(nextPoint);  v[nextPoint.x][nextPoint.y] = true;  }  }  }  return -1;  } } |

## Sequence Reconstruction

|  |
| --- |
| public class Solution {  public boolean sequenceReconstruction(int[] org, int[][] seqs) {  Map<Integer, Set<Integer>> map = new HashMap<>();  Map<Integer, Integer> indegree = new HashMap<>();    for(int[] seq: seqs) {  if(seq.length == 1) {  if(!map.containsKey(seq[0])) {  map.put(seq[0], new HashSet<>());  indegree.put(seq[0], 0);  }  } else {  for(int i = 0; i < seq.length - 1; i++) {  if(!map.containsKey(seq[i])) {  map.put(seq[i], new HashSet<>());  indegree.put(seq[i], 0);  }   if(!map.containsKey(seq[i + 1])) {  map.put(seq[i + 1], new HashSet<>());  indegree.put(seq[i + 1], 0);  }   if(map.get(seq[i]).add(seq[i + 1])) {  indegree.put(seq[i + 1], indegree.get(seq[i + 1]) + 1);  }  }  }  }   Queue<Integer> queue = new LinkedList<>();  for(Map.Entry<Integer, Integer> entry: indegree.entrySet()) {  if(entry.getValue() == 0) queue.offer(entry.getKey());  }   int index = 0;  while(!queue.isEmpty()) {  int size = queue.size();  if(size > 1) return false;  int curr = queue.poll();  if(index == org.length || curr != org[index++]) return false;  for(int next: map.get(curr)) {  indegree.put(next, indegree.get(next) - 1);  if(indegree.get(next) == 0) queue.offer(next);  }  }  return index == org.length && index == map.size();  } } |

## Clone Graph

|  |
| --- |
| public class Solution {  private HashMap<Integer, UndirectedGraphNode> map = new HashMap<>();  public UndirectedGraphNode cloneGraph(UndirectedGraphNode node) {  return clone(node);  }   private UndirectedGraphNode clone(UndirectedGraphNode node) {  if (node == null) return null;    if (map.containsKey(node.label)) {  return map.get(node.label);  }  UndirectedGraphNode clone = new UndirectedGraphNode(node.label);  map.put(clone.label, clone);  for (UndirectedGraphNode neighbor : node.neighbors) {  clone.neighbors.add(clone(neighbor));  }  return clone;  } } |

## Topological Sorting

|  |
| --- |
| public class Solution {  public ArrayList<DirectedGraphNode> topSort(ArrayList<DirectedGraphNode> graph) {  HashMap<DirectedGraphNode, Integer> map = new HashMap();  for (DirectedGraphNode node : graph) {  if(!map.containsKey(node)) map.put(node, 0);  for (DirectedGraphNode neighbor : node.neighbors) {  if (map.containsKey(neighbor)) {  map.put(neighbor, map.get(neighbor) + 1);  } else {  map.put(neighbor, 1);   }  }  }  ArrayList<DirectedGraphNode> result = new ArrayList<DirectedGraphNode>();  for(DirectedGraphNode node : graph){  if(!map.containsKey(node) || map.get(node) == 0)dfs(node, map, graph, result);  }  return result;  }    public void dfs(DirectedGraphNode node,   HashMap<DirectedGraphNode, Integer> map,   ArrayList<DirectedGraphNode> graph,   ArrayList<DirectedGraphNode> result){  map.put(node, -1);  result.add(node);  for(DirectedGraphNode neighbor : node.neighbors){  map.put(neighbor, map.get(neighbor) - 1);  if(map.get(neighbor) == 0) dfs(neighbor, map, graph, result);  }  return ;  } } |

## Serialize and Deserialize Binary Tree

|  |
| --- |
| public class Codec {  private static final String spliter = ",";  private static final String NN = "X";   // Encodes a tree to a single string.  public String serialize(TreeNode root) {  StringBuilder sb = new StringBuilder();  buildString(root, sb);  return sb.toString();  }   private void buildString(TreeNode node, StringBuilder sb) {  if (node == null) {  sb.append(NN).append(spliter);  } else {  sb.append(node.val).append(spliter);  buildString(node.left, sb);  buildString(node.right,sb);  }  }  // Decodes your encoded data to tree.  public TreeNode deserialize(String data) {  Deque<String> nodes = new LinkedList<>();  nodes.addAll(Arrays.asList(data.split(spliter)));  return buildTree(nodes);  }    private TreeNode buildTree(Deque<String> nodes) {  String val = nodes.remove();  if (val.equals(NN)) return null;  else {  TreeNode node = new TreeNode(Integer.valueOf(val));  node.left = buildTree(nodes);  node.right = buildTree(nodes);  return node;  }  } } |

## 推荐方法

|  |
| --- |
| public class Seri {   String result="";  int id=0;  private int serialWork(TreeNode root){  if (root==null) return -1;  id++;  int nowId=id;  int leftId=serialWork(root.left);  int rightId=serialWork(root.right);  result+=root.val+","+nowId+","+leftId+","+rightId+";";  return nowId;  }    public String serialize(TreeNode root) {  // write your code here  int rootId=serialWork(root);  return result;  }   public TreeNode deserialize(String data) {  // write your code here  String[] nodes=data.split(";");  HashMap<Integer,TreeNode> nodeHashMap=new HashMap<Integer, TreeNode>();  for (int i=0;i<nodes.length;i++){  String[] datas=nodes[i].split(",");  if (datas.length!=4) continue;  if (!nodeHashMap.containsKey(Integer.parseInt(datas[1]))) {  TreeNode node = new TreeNode(Integer.parseInt(datas[0]));  nodeHashMap.put(Integer.parseInt(datas[1]), node);  }  else{  nodeHashMap.get(Integer.parseInt(datas[1])).val=Integer.parseInt(datas[0]);  }  if (!nodeHashMap.containsKey(Integer.parseInt(datas[2])) && Integer.parseInt(datas[2])!=-1){  nodeHashMap.put(Integer.parseInt(datas[2]),new TreeNode(-1));  }  if (!nodeHashMap.containsKey(Integer.parseInt(datas[3])) && Integer.parseInt(datas[3])!=-1){  nodeHashMap.put(Integer.parseInt(datas[3]),new TreeNode(-1));  }  nodeHashMap.get(Integer.parseInt(datas[1])).left=nodeHashMap.get(Integer.parseInt(datas[2]));  nodeHashMap.get(Integer.parseInt(datas[1])).right=nodeHashMap.get(Integer.parseInt(datas[3]));  }   return nodeHashMap.get(1);  } } |

## Word Ladder

|  |
| --- |
| public int ladderLength(String beginWord, String endWord, Set<String> wordDict) {  Set<String> reached = new HashSet<String>();  reached.add(beginWord);  wordDict.add(endWord);  int distance = 1;  while (!reached.contains(endWord)) {  Set<String> toAdd = new HashSet<String>();  for (String each : reached) {  for (int i = 0; i < each.length(); i++) {  char[] chars = each.toCharArray();  for (char ch = 'a'; ch <= 'z'; ch++) {  chars[i] = ch;  String word = new String(chars);  if (wordDict.contains(word)) {  toAdd.add(word);  wordDict.remove(word);  }  }  }  }  distance++;  if (toAdd.size() == 0) return 0;  reached = toAdd;  }  return distance;  } |

# Binary Search

## Last Position of Target

|  |
| --- |
| public class Solution {  /\*  \* @param nums: An integer array sorted in ascending order  \* @param target: An integer  \* @return: An integer  \*/  public int lastPosition(int[] nums, int target) {  // write your code here  if (nums == null || nums.length == 0) {  return -1;  }  int start = 0;  int end = nums.length - 1;  while (start + 1 < end) {  int mid = start + (end - start) / 2;  if (nums[mid] == target) {  start = mid;  } else if (nums[mid] < target) {  start = mid;  } else {  end = mid;  }  }  if (nums[end] == target) {  return end;  } else if (nums[start] == target) {  return start;  } else {  return -1;  }  } } |

## Maximum Number in Mountain Sequence

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums a mountain sequence which increase firstly and then decrease  \* @return then mountain top  \*/  public int mountainSequence(int[] nums) {  // Write your code here  int l = 0, r = nums.length - 1;  int mid = (l + r) / 2;    int lst = -1;    while(l < r){  if(nums[mid] < nums[mid + 1])  l = mid;   else r = mid;    mid = (l + r) / 2;  if(mid == lst){  break;  }  lst = mid;  }  if(nums[l] > nums[r]){  return nums[l];  }  else{  return nums[r];  }  } } |

## Find K Closest Elements

|  |
| --- |
| class Solution {  public List<Integer> findClosestElements(int[] arr, int k, int x) {  List<Integer> result = new ArrayList<>(k);  int position = Arrays.binarySearch(arr, x);  int left;  int right;  if(position < 0){  position = -position - 1;  left = position - 1;  right = position;  }  else{  left = position - 1;  right = position + 1;  }    while(right - left - 1 != k){  if(left < 0){  right ++;  }  else if(right > arr.length - 1){  left --;  }  else if(arr[right] - x < x - arr[left]){  right ++;  }else{  left --;  }  }    for(int i = 0; i < k; i ++){  result.add(arr[left + i + 1]);  }  return result;  } } |

## Search in a Big Sorted Array

|  |
| --- |
| public class Solution {  /\*  \* @param reader: An instance of ArrayReader.  \* @param target: An integer  \* @return: An integer which is the first index of target.  \*/  public int searchBigSortedArray(ArrayReader reader, int target) {  int firstElement = reader.get(0);  if (firstElement == target)   return 0;  else if (firstElement > target)  return -1;    int idx = 0, jump = 1;  while (jump != 0) {  while (jump != 0 && reader.get(idx + jump) < target)   idx += jump;  jump >>= 1;  jump <<= 1;   }    if (reader.get(idx + 1) == target)  return idx + 1;  else  return -1;  } } |

## Fast Power

|  |
| --- |
| class Solution {  /\*  \* @param a, b, n: 32bit integers  \* @return: An integer  \*/  public int fastPower(int a, int b, int n) {  long ans = 1 % b, tmp = a;  while(n > 0) {  if (n % 2 == 1) {  ans = ans \* tmp % b;  }  tmp = tmp \* tmp % b;  n = n / 2;  }  return (int)ans;  } } |

## Find Peak Element

|  |
| --- |
| class Solution {  public int findPeakElement(int[] nums) {  int l = 0;  int r = nums.length - 1;  while(l + 1 < r){  int mid = (l+r) / 2;  if(nums[mid] > nums[mid-1]){  l = mid;  }else{  r = mid;  }  }  return nums[l] > nums[r] ? l:r;  } } |

## First Bad Version

|  |
| --- |
| class Solution {  /\*\*  \* @param n: An integers.  \* @return: An integer which is the first bad version.  \*/  public int findFirstBadVersion(int n) {  int start = 1, end = n;  while (start + 1 < end) {  int mid = start + (end - start) / 2;  if (SVNRepo.isBadVersion(mid)) {  end = mid;  } else {  start = mid;  }  }    if (SVNRepo.isBadVersion(start)) {  return start;  }  return end;  } |

## Search in Rotated Sorted Array

|  |
| --- |
| class Solution {  public int search(int[] nums, int target) {  if(nums.length == 0) return -1;  int l = 0;  int r = nums.length - 1;  while((l+1) < r){  int mid = (l+ r) / 2;  if(nums[mid] > nums[l]){  if(nums[mid] > target && nums[l] <= target){  r = mid;  }else{  l = mid;  }  }else{  if(nums[mid] < target && nums[r] >= target){  l = mid;  }else{  r = mid;  }  }  }  if(nums[l] == target) return l;  if(nums[r] == target) return r;  return -1;  } } |

G. add two numbers II

|  |
| --- |
| public class Solution {  /\*\*  \* @param l1: the first list  \* @param l2: the second list  \* @return: the sum list of l1 and l2   \*/  public ListNode addLists2(ListNode l1, ListNode l2) {  l1 = reverse(l1);  l2 = reverse(l2);    return reverse(addList1(l1, l2));  }     private ListNode reverse(ListNode head) {  ListNode prev = null;  while (head != null) {  ListNode temp = head.next;  head.next = prev;  prev = head;  head = temp;  }  return prev;  }    private ListNode addList1(ListNode l1, ListNode l2) {  ListNode dummy = new ListNode(0);  ListNode tail = dummy;  int carry = 0;    while (l1 != null && l2 != null) {  tail.next = new ListNode((l1.val + l2.val + carry) % 10);  tail = tail.next;  carry = (l1.val + l2.val + carry) / 10;    l1 = l1.next;  l2 = l2.next;  }    while (l1 != null) {  tail.next = new ListNode((l1.val + carry) % 10);  tail = tail.next;  carry = (l1.val + carry) / 10;    l1 = l1.next;  }  while (l2 != null) {  tail.next = new ListNode((l2.val + carry) % 10);  tail = tail.next;  carry = (l2.val + carry) / 10;    l2 = l2.next;  }    while (carry != 0) {  tail.next = new ListNode(carry % 10);  tail = tail.next;  carry = carry / 10;  }    return dummy.next;  } } |

# Binary Tree Divide & Conquer

## Closest Binary Search Tree Value

|  |
| --- |
| public class Solution {  /\*\*  \* @param root: the given BST  \* @param target: the given target  \* @return: the value in the BST that is closest to the target  \*/  public int ans;  public int closestValue(TreeNode root, double target) {  // write your code here  ans = root.val;  dfs(root, target);  return ans;  }  public void dfs(TreeNode cur, double target){  if(cur == null) return ;  if(Math.abs(cur.val - target) < Math.abs(ans - target)) ans = cur.val;  if(target < cur.val) dfs(cur.left, target);  else dfs(cur.right, target);  } } |

## Vertical Order Traversal of a Binary Tree

|  |
| --- |
| class Solution {  static class Pair implements Comparable<Pair> {  int val;  int pos;  public Pair(int val, int pos) {  this.val = val;  this.pos = pos;  }    @Override  public int compareTo(Pair o) {  int compare = Integer.compare(o.pos, pos);  if (compare != 0) {  return compare;  }  return Integer.compare(val, o.val);  }  }    Map<Integer, PriorityQueue<Pair>> map;  int min;  int max;  public List<List<Integer>> verticalTraversal(TreeNode root) {  map = new HashMap<>();  min = Integer.MAX\_VALUE;  max = Integer.MIN\_VALUE;  helper(root, 0, 0);  List<List<Integer>> result = new ArrayList<>(max - min);  for (int i = min; i <= max; i++) {  PriorityQueue<Pair> pq = map.get(i);  List<Integer> list = new ArrayList<>(pq.size());  while (!pq.isEmpty()) {  list.add(pq.poll().val);  }  result.add(list);  }  return result;  }    private void helper(TreeNode node, int verticalLevel, int horizontalLevel) {  if (node == null) {  return;  }  min = Math.min(min, verticalLevel);  max = Math.max(max, verticalLevel);  map.computeIfAbsent(verticalLevel, k -> new PriorityQueue<>()).offer(new Pair(node.val, horizontalLevel));  helper(node.left, verticalLevel - 1, horizontalLevel - 1);  helper(node.right, verticalLevel + 1, horizontalLevel - 1);  } } |

## Minimum Subtree

|  |
| --- |
| public class Solution {  private TreeNode subtree = null;  private int subtreeSum = Integer.MAX\_VALUE;  public TreeNode findSubtree(TreeNode root) {  // write your code here  dfs(root);  return subtree;  }    private int dfs(TreeNode root){  if(root == null){  return 0;  }  int sum = dfs(root.left) + dfs(root.right) + root.val;  if(sum < subtreeSum){  subtreeSum = sum;  subtree = root;  }  return sum;  } } |

## Binary Tree Paths

|  |
| --- |
| public class Solution {  /\*\*  \* @param root the root of the binary tree  \* @return all root-to-leaf paths  \*/  public List<String> binaryTreePaths(TreeNode root) {  // Write your code here  if(root != null) findPaths(root,String.valueOf(root.val));  return res;  }   private void findPaths(TreeNode n, String path){  if(n.left == null && n.right == null) res.add(path);  if(n.right != null) findPaths(n.right, path+"->"+n.right.val);  if(n.left != null) findPaths(n.left, path+"->"+n.left.val);    }   private List<String> res = new ArrayList(); } |

## Flatten Binary Tree to Linked List

|  |
| --- |
| public class Solution {  /\*\*  \* @param root: a TreeNode, the root of the binary tree  \* @return: nothing  \*/  public void flatten(TreeNode root) {  // write your code here  dfs(root);  }    public TreeNode dfs(TreeNode root) {  if (root == null) {  return null;  }    TreeNode leftlast = dfs(root.left);  TreeNode rightlast = dfs(root.right);    if (leftlast != null) {  leftlast.right = root.right;  root.right = root.left;  root.left = null;  }  if (rightlast != null) {  return rightlast;  }  if (leftlast != null) {  return leftlast;  }    return root;  } } |

## Balanced Binary Tree

|  |
| --- |
| public class Solution {  /\*\*  \* @param root: The root of binary tree.  \* @return: True if this Binary tree is Balanced, or false.  \*/    boolean balanced = true;    private int check(TreeNode p) { // return height of subtree rooted at p  if (p == null) {  return 0;  }    int l = check(p.left);  int r = check(p.right);  if (Math.abs(l - r) > 1) { // unbalanced  balanced = false;  }    return Math.max(l, r) + 1;  }    public boolean isBalanced(TreeNode root) {  check(root);  return balanced;  } } |

## Kth Smallest Element in a BST

|  |
| --- |
| public class Solution {  /\*\*  \* @param root: the given BST  \* @param k: the given k  \* @return: the kth smallest element in BST  \*/    public int quickSelectOnTree(TreeNode root, int k,HashMap<TreeNode, Integer> size) {  if(root == null) {  return -1;  }  int left = 0;  if(root.left != null) {  left = size.get(root.left);  }  if(left == k - 1) {  return root.val;  }   if(left >= k) {  return quickSelectOnTree(root.left, k, size);  } else {  return quickSelectOnTree(root.right, k - 1 - left, size);  }  }      public int getSize(TreeNode root, HashMap<TreeNode, Integer> size) {  if(root == null) {  return 0;  }  int sum = 1;  sum += getSize(root.left, size);  sum += getSize(root.right, size);  size.put(root, sum);  return sum;  }    public int kthSmallest(TreeNode root, int k) {  // write your code here  HashMap<TreeNode, Integer> size = new HashMap<>();  getSize(root, size);    return quickSelectOnTree(root, k, size);  } } |

## Lowest Common Ancestor III

|  |
| --- |
| public class Solution {  /\*  \* @param root: The root of the binary tree.  \* @param A: A TreeNode  \* @param B: A TreeNode  \* @return: Return the LCA of the two nodes.  \*/    class resultType{  boolean A, B;  TreeNode LCA;  resultType(boolean A, boolean B, TreeNode LCA){  this.A = A;  this.B = B;  this.LCA = LCA;  }  }  public TreeNode lowestCommonAncestor3(TreeNode root, TreeNode A, TreeNode B) {  // write your code here      resultType res = find(root, A, B);  TreeNode LCA = res.LCA;  return LCA;  }    private resultType find(TreeNode root, TreeNode A, TreeNode B){  resultType res = new resultType(false, false, null);  if(root == null){  return res ;  }    resultType left = find(root.left, A, B);  resultType right = find(root.right, A, B);    if(left.LCA != null){  res.LCA = left.LCA;  return res;  }    if(right.LCA != null){  res.LCA = right.LCA;  return res;  }      res.A = left.A | right.A | root == A;  res.B = left.B | right.B | root == B;      if(res.A && res.B ){  res.LCA = root;  return res;  }    return res;  } |

## Validate Binary Search Tree

|  |
| --- |
| public class Solution {  /\*\*  \* @param root: The root of binary tree.  \* @return: True if the binary tree is BST, or false  \*/    public boolean isValidBST(TreeNode root) {  // write your code here    return divConq(root, Long.MIN\_VALUE, Long.MAX\_VALUE);  }  private boolean divConq(TreeNode root, long min, long max){  if(root == null){  return true;  }    if(root.val <= min || root.val >= max){  return false;  }    return divConq(root.left, min, root.val) &&  divConq(root.right, root.val, max);  } } |

## Closest Binary Search Tree Value II

|  |
| --- |
| public class Solution {  /\*\*  \* @param root: the given BST  \* @param target: the given target  \* @param k: the given k  \* @return: k values in the BST that are closest to the target  \*/  private void findTarget(Stack<TreeNode> stack,TreeNode root,double target){  while (root!=null){  stack.push(root);  if (target<root.val){  root=root.left;  }  else {  root=root.right;  }  }  }     private void getNext(Stack<TreeNode> stack){  TreeNode node=stack.peek();  if (node.right!=null){  node=node.right;  while (node!=null){  stack.push(node);  node=node.left;  }  }  else {  while (!stack.isEmpty()){  node=stack.pop();  if (!stack.isEmpty() && stack.peek().right!=node){  break;  }  }  }  }    private void getPrev(Stack<TreeNode> stack){  TreeNode node=stack.peek();  if (node.left!=null){  node=node.left;  while (node!=null){  stack.push(node);  node=node.right;  }  }  else {  while (!stack.isEmpty()){  node=stack.pop();  if (!stack.isEmpty() && stack.peek().left!=node){  break;  }  }  }  }    public List<Integer> closestKValues(TreeNode root, double target, int k) {  // write your code here  Stack<TreeNode> nextStack=new Stack<>();  Stack<TreeNode> prevStack=new Stack<>();  findTarget(nextStack,root,target);  prevStack.addAll(nextStack);  getPrev(prevStack);    List<Integer> answer=new ArrayList<>();  for (int i=0;i<k;i++){  if (prevStack.isEmpty()){  answer.add(nextStack.peek().val);  getNext(nextStack);  }  else if (nextStack.isEmpty()){  answer.add(prevStack.peek().val);  getPrev(prevStack);  }  else if (target-prevStack.peek().val<nextStack.peek().val-target){  answer.add(prevStack.peek().val);  getPrev(prevStack);   }  else {  answer.add(nextStack.peek().val);  getNext(nextStack);   }  }  return answer;  } } |

## Binary Search Tree Iterator

* + 1. LintCode答案

|  |
| --- |
| public class BSTIterator {    private Stack<TreeNode> stack = new Stack<>();   public BSTIterator(TreeNode root) {  TreeNode cur = root;  while (cur != null) {  stack.push(cur);  cur = cur.left;  }  }   public boolean hasNext() {  return !stack.isEmpty();  }    public TreeNode next() {  TreeNode ans = stack.pop();  TreeNode cur = ans.right;  while (cur != null) {  stack.push(cur);  cur = cur.left;  }  return ans;  } } |

* + 1. LeetCode答案

|  |
| --- |
| public class BSTIterator {    private Stack<TreeNode> stack = new Stack<>();   public BSTIterator(TreeNode root) {  TreeNode cur = root;  while (cur != null) {  stack.push(cur);  cur = cur.left;  }  }   public boolean hasNext() {  return !stack.isEmpty();  }    public int next() {  TreeNode ans = stack.pop();  TreeNode cur = ans.right;  while (cur != null) {  stack.push(cur);  cur = cur.left;  }  return ans.val;  } } |

# Two Pointers

## Middle of Linked List

|  |
| --- |
| public class Solution {  /\*  \* @param head: the head of linked list.  \* @return: a middle node of the linked list  \*/  public ListNode middleNode(ListNode head) {  // write your code here  if(head == null || head.next == null)  return head;  ListNode slow = head;  ListNode fast = head.next;  while(fast != null && fast.next != null){  slow = slow.next;  fast = fast.next.next;  }  return slow;  } } |

## Two Sum III - Data structure design

|  |
| --- |
| public class TwoSum {  /\*\*  \* @param number: An integer  \* @return: nothing  \*/    private int[] nums = new int[20000];  private int count = 0;    public void add(int number) {  nums[count++] = number;  for (int i = count - 1; i > 0; --i) {  if (nums[i - 1] <= nums[i]) break;  int tmp = nums[i - 1];  nums[i - 1] = nums[i];  nums[i] = tmp;  }   }   /\*\*  \* @param value: An integer  \* @return: Find if there exists any pair of numbers which sum is equal to the value.  \*/  public boolean find(int value) {  if (count < 2) return false;  int left = 0, right = count - 1;  while (left < right) {  int sum = nums[left] + nums[right];  if (sum == value) return true;  if (sum < value) {  ++left;  } else {  --right;  }  }  return false;  } } |

## Move Zeroes

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums: an integer array  \* @return: nothing  \*/  public void moveZeroes(int[] nums) {  // write your code here    // // version 1  int slow = 0, fast = 0;  while(fast < nums.length){  if(nums[fast] != 0) nums[slow++] = nums[fast++];  else fast++;  }  while(slow < nums.length) nums[slow++] = 0;  } } |

## Remove Duplicate Numbers in Array

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums an array of integers  \* @return the number of unique integers  \*/  public int deduplication(int[] nums) {  // Write your code here  if (nums.length == 0) {  return 0;  }    Arrays.sort(nums);  int i, j = -1;  for (i = 0; i < nums.length; ++i) {  if (j == -1 || nums[i] != nums[j]) {  ++j;  nums[j] = nums[i];  }  }    return j + 1;  } } |

## Sort Integers II

### Merge Sort

|  |
| --- |
| public class Solution {  /\*\*  \* @param A an integer array  \* @return void  \*/  public void sortIntegers2(int[] A) {  // use a shared temp array, the extra memory is O(n) at least  int[] temp = new int[A.length];  mergeSort(A, 0, A.length - 1, temp);  }    private void mergeSort(int[] A, int start, int end, int[] temp) {  if (start >= end) {  return;  }    int left = start, right = end;  int mid = (start + end) / 2;   mergeSort(A, start, mid, temp);  mergeSort(A, mid+1, end, temp);  merge(A, start, mid, end, temp);  }    private void merge(int[] A, int start, int mid, int end, int[] temp) {  int left = start;  int right = mid+1;  int index = start;    // merge two sorted subarrays in A to temp array  while (left <= mid && right <= end) {  if (A[left] < A[right]) {  temp[index++] = A[left++];  } else {  temp[index++] = A[right++];  }  }  while (left <= mid) {  temp[index++] = A[left++];  }  while (right <= end) {  temp[index++] = A[right++];  }    // copy temp back to A  for (index = start; index <= end; index++) {  A[index] = temp[index];  }  } } |

### Quick Sort

|  |
| --- |
| public class Solution {  /\*\*  \* @param A: an integer array  \* @return: nothing  \*/  public void sortIntegers2(int[] A) {  // write your code here  quickSort(A, 0, A.length - 1);  }    private void quickSort(int[] A, int start, int end) {  if (start >= end) {  return;  }  int mid = partition(A, start, end);  quickSort(A, start, mid - 1);  quickSort(A, mid + 1, end);  }    private int partition(int[] A, int start, int end) {  int pivot = A[start];  int i = start + 1;  int j = end;  while (i <= j) {  while (i <= j && A[i] <= pivot) {  i++;  }  while (i <= j && A[j] > pivot) {  j--;  }  if (i > j) {  break;  }   int temp = A[i];  A[i] = A[j];  A[j] = temp;  j--;  i++;  }  A[start] = A[j];  A[j] = pivot;  return j;  } } |

### Heap Sort

|  |
| --- |
| public class Solution {  /\*\*  \* @param A: an integer array  \* @return: nothing  \*/  public void sortIntegers2(int[] A) {  // write your code here  if (A.length < 2) {  return;  }  for (int i = A.length / 2; i >= 0; i--) {  reHeap(A, i, A.length - 1);  }  int temp = A[0];  A[0] = A[A.length - 1];  A[A.length - 1] = temp;  for (int i = A.length - 2; i > 0; i--) {  reHeap(A, 0, i);  temp = A[0];  A[0] = A[i];  A[i] = temp;  }  }    private void reHeap(int[] A, int start, int end) {  int val = A[start];  for (int i = 2 \* start + 1; i <= end; start = i, i = 2 \* start + 1) {  if (i < end && A[i + 1] > A[i]) {  i++;  }  if (val >= A[i]) {  break;  }  A[start] = A[i];  }  A[start] = val;  } }  public class Solution {  /\*  \* @param A: an integer array  \* @return:  \*/  public Random rand;  public void sortIntegers2(int[] A) {  rand = new Random();  // write your code here  quickSort(A, 0, A.length - 1);  }    public void quickSort(int[] A, int start, int end) {  if (start >= end) {  return;  }  int index = rand.nextInt(end - start + 1) + start;  int pivot = A[index];  int left = start;  int right = end;    while (left <= right) {  while (left <= right && A[left] < pivot) {  left ++;  }  while (left <= right && A[right] > pivot) {  right --;  }    if (left <= right) {  int temp = A[left];  A[left] = A[right];  A[right] = temp;    left ++;  right --;  }  }  // A[start... right]  quickSort(A, start, right);  // A[left ... end]  quickSort(A, left, end);  }  } |

## Two Sum II - Input array is sorted

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums: an array of Integer  \* @param target: target = nums[index1] + nums[index2]  \* @return: [index1 + 1, index2 + 1] (index1 < index2)  \*/  public int[] twoSum(int[] nums, int target) {  // write your code here  if (nums == null || nums.length < 2) {  return new int[0];  }    for (int l = 0, r = nums.length - 1; l < r; r--) {  while (l < r && nums[l] + nums[r] < target) {  l++;  }    if (l != r && nums[l] + nums[r] == target) {  return new int[] {l + 1, r + 1};  }  }  return new int[0];  } } |

## Sort Colors II

|  |
| --- |
| class Solution {  /\*\*  \* @param colors: A list of integer  \* @param k: An integer  \* @return: nothing  \*/  public void sortColors2(int[] colors, int k) {  if (colors == null || colors.length == 0) {  return;  }  rainbowSort(colors, 0, colors.length - 1, 1, k);  }    public void rainbowSort(int[] colors,  int left,  int right,  int colorFrom,  int colorTo) {  if (colorFrom == colorTo) {  return;  }    if (left >= right) {  return;  }    int colorMid = (colorFrom + colorTo) / 2;  int l = left, r = right;  while (l <= r) {  while (l <= r && colors[l] <= colorMid) {  l++;  }  while (l <= r && colors[r] > colorMid) {  r--;  }  if (l <= r) {  int temp = colors[l];  colors[l] = colors[r];  colors[r] = temp;    l++;  r--;  }  }    rainbowSort(colors, left, r, colorFrom, colorMid);  rainbowSort(colors, l, right, colorMid + 1, colorTo);  } } |

## 3Sum

|  |
| --- |
| public class Solution {  /\*\*  \* @param numbers: Give an array numbers of n integer  \* @return: Find all unique triplets in the array which gives the sum of zero.  \*/  public List<List<Integer>> threeSum(int[] numbers) {  // write your code here  List<List<Integer>> result = new ArrayList<>();  if(numbers == null || numbers.length < 3){  return result;  }  Arrays.sort(numbers);    for(int i = 2; i < numbers.length; i++){  if(i + 1 < numbers.length && numbers[i + 1] == numbers[i]){  continue;  }  int tgt = -1 \* numbers[i];  twoSum(numbers, tgt, i, result);  }  return result;  }  public void twoSum(int[] nums, int target, int r, List<List<Integer>> result) {  // write your code here  int left = 0;  int right = r - 1;  while(left < right){  if(nums[left] + nums[right] == target){  List<Integer> temp = new ArrayList<>();  temp.add(-1\*target);  temp.add(nums[left]);  temp.add(nums[right]);  Collections.sort(temp);  result.add(temp);  left++;  right--;  while(left < right && nums[left] == nums[left-1]){  left++;  }  while(left < right && nums[right] == nums[right+1]){  right--;  }  }else if(nums[left] + nums[right] < target){  left++;  }else{  right--;  }  }  } } |

## Partition Array

|  |
| --- |
| public class Solution {  /\*\*   \*@param nums: The integer array you should partition  \*@param k: As description  \*return: The index after partition  \*/  public int partitionArray(int[] nums, int k) {  if(nums == null || nums.length == 0){  return 0;  }    int left = 0, right = nums.length - 1;  while (left <= right) {   while (left <= right && nums[left] < k) {  left++;  }   while (left <= right && nums[right] >= k) {  right--;  }   if (left <= right) {  int temp = nums[left];  nums[left] = nums[right];  nums[right] = temp;    left++;  right--;  }  }  return left;  } } |

## Kth Largest Element

|  |
| --- |
| class Solution {  /\*  \* @param k : description of k  \* @param nums : array of nums  \* @return: description of return  \*/  public int kthLargestElement(int k, int[] nums) {  if (nums == null || nums.length == 0 || k < 1 || k > nums.length){  return -1;  }  return partition(nums, 0, nums.length - 1, nums.length - k);  }    private int partition(int[] nums, int start, int end, int k) {  if (start >= end) {  return nums[k];  }    int left = start, right = end;  int pivot = nums[(start + end) / 2];    while (left <= right) {  while (left <= right && nums[left] < pivot) {  left++;  }  while (left <= right && nums[right] > pivot) {  right--;  }  if (left <= right) {  swap(nums, left, right);  left++;  right--;  }  }    if (k <= right) {  return partition(nums, start, right, k);  }  if (k >= left) {  return partition(nums, left, end, k);  }  return nums[k];  }     private void swap(int[] nums, int i, int j) {  int tmp = nums[i];  nums[i] = nums[j];  nums[j] = tmp;  } }; |

# DFS

## Split String

|  |
| --- |
| public class Solution {  /\*  \* @param : a string to be split  \* @return: all possible split string array  \*/  public List<List<String>> splitString(String s) {  List<List<String>> result = new ArrayList<>();  helper(s, 0, new ArrayList<>(), result);  return result;  }    private void helper(String s, int index, List<String> current, List<List<String>> result) {  if (index == s.length()) {  result.add(new ArrayList<>(current));  return;  }    // Option 1  current.add(String.valueOf(s.charAt(index)));  helper(s, index + 1, current, result);  current.remove(current.size() - 1);    // Option 2  if (index < s.length() - 1) {  current.add(s.substring(index, index + 2));  helper(s, index + 2, current, result);  current.remove(current.size() - 1);  }  } } |

## Letter Combinations of a Phone Number

|  |
| --- |
| public class Solution {  /\*\*  \* @param digits: A digital string  \* @return: all posible letter combinations  \*/  public List<String> letterCombinations(String digits) {  // write your code here  /\*for (int i = 'a'; i <= 'c'; i++) {  for (int j = 'j'; j <= 'l'; j++) {  for (int k = 'd'; k <= 'f'; k++) {  print();  }  }  }\*/    String[] phone = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"};    List<String> ans = new ArrayList<>();  if (digits == null || digits.length() == 0) {  return ans;  }    dfs(0, "", digits, phone, ans);  return ans;  }    private void dfs(int x, String str, String digits, String[] phone, List<String> ans) {  if (x == digits.length()) {  ans.add(str);  return;  }    int d = digits.charAt(x) - '0';  for (char c : phone[d].toCharArray()) {  dfs(x + 1, str + c, digits, phone, ans);  }  } } |

## Combination Sum II

|  |
| --- |
| public class Solution {  /\*\*  \* @param num: Given the candidate numbers  \* @param target: Given the target number  \* @return: All the combinations that sum to target  \*/  public List<List<Integer>> combinationSum2(int[] candidates,  int target) {  List<List<Integer>> results = new ArrayList<>();  if (candidates == null || candidates.length == 0) {  return results;  }   Arrays.sort(candidates);  List<Integer> combination = new ArrayList<Integer>();  helper(candidates, 0, combination, target, results);   return results;  }   private void helper(int[] candidates,  int startIndex,  List<Integer> combination,  int target,  List<List<Integer>> results) {  if (target == 0) {  results.add(new ArrayList<Integer>(combination));  return;  }   for (int i = startIndex; i < candidates.length; i++) {  if (i != startIndex && candidates[i] == candidates[i - 1]) {  continue;  }  if (target < candidates[i]) {  break;  }  combination.add(candidates[i]);  helper(candidates, i + 1, combination, target - candidates[i], results);  combination.remove(combination.size() - 1);  }  } } |

## Combination Sum

|  |
| --- |
| public class Solution {  /\*\*  \* @param candidates: A list of integers  \* @param target: An integer  \* @return: A list of lists of integers  \*/  public List<List<Integer>> combinationSum(int[] candidates, int target) {  Arrays.sort(candidates);    List<List<Integer>> results = new ArrayList<>();  dfs(candidates, 0, target, new ArrayList<Integer>(), results);  return results;  }    private void dfs(int[] candidates, int index, int target, List<Integer> combination, List<List<Integer>> results) {  if (index == candidates.length) {  if (target == 0) {  results.add(new ArrayList<Integer>(combination));  }  return;  }    if (target < 0) {  return;  }    dfs(candidates, index + 1, target, combination, results);    if (index > 0 && candidates[index] == candidates[index - 1]) {  return;  }    combination.add(candidates[index]);  dfs(candidates, index, target - candidates[index], combination, results);  combination.remove(combination.size() - 1);  } } |

## N-Queens

|  |
| --- |
| class Solution {  private Set<Integer> vertical;  private Set<Integer> cross1;  private Set<Integer> cross2;  List<List<String>> result;    public List<List<String>> solveNQueens(int n) {  result = new ArrayList<>();  vertical = new HashSet<>();  cross1 = new HashSet<>();  cross2 = new HashSet<>();    helper(n, 0, new ArrayList<>());    return result;  }    private void helper(int n, int row, List<Integer> path) {  if (row == n) {  result.add(convert(path));  return;  }    for (int col = 0; col < n; col++) {  int cross1Val = row + col;  int cross2Val = row - col;  if (notValid(col, cross1Val, cross2Val)) {  continue;  }  vertical.add(col);  cross1.add(cross1Val);  cross2.add(cross2Val);  path.add(col);    helper(n, row + 1, path);    vertical.remove(col);  cross1.remove(cross1Val);  cross2.remove(cross2Val);  path.remove(path.size() - 1);  }  }    private boolean notValid(int col, int cross1Val, int cross2Val) {  return vertical.contains(col) || cross1.contains(cross1Val) || cross2.contains(cross2Val);  }    private List<String> convert(List<Integer> path) {  char[] board = new char[path.size()];  Arrays.fill(board, '.');   return path.stream()  .map(val -> {  board[val] = 'Q';  String newBoard = new String(board);  board[val] = '.';  return newBoard;  })  .collect(Collectors.toList());  } } |

## Subsets II

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums: A set of numbers.  \* @return: A list of lists. All valid subsets.  \*/  public List<List<Integer>> subsetsWithDup(int[] nums) {  // write your code here  List<List<Integer>> result = new ArrayList<>();  List<Integer> path = new ArrayList<>();  Arrays.sort(nums);  dfs(nums, result, path, 0, -1);  return result;  }    private void dfs(int[] nums, List<List<Integer>> result, List<Integer> path, int index, int lastSelected) {  if (index == nums.length) {  result.add(new ArrayList<>(path));  return;  }    // not select  dfs(nums, result, path, index + 1, lastSelected);      // select  if (index > 0 && nums[index] == nums[index - 1] && lastSelected != index - 1) {  return;  }  path.add(nums[index]);  dfs(nums, result, path, index + 1, index);  path.remove(path.size() - 1);  } } |

## Subsets

|  |
| --- |
| public class Solution {    /\*  \* @param nums: A set of numbers  \* @return: A list of lists  \*/  public List<List<Integer>> subsets(int[] nums) {  // List vs ArrayList （google）  List<List<Integer>> results = new ArrayList<>();    if (nums == null) {  return results; // 空列表  }    Arrays.sort(nums);    // BFS  Queue<List<Integer>> queue = new LinkedList<>();  queue.offer(new LinkedList<Integer>());    while (!queue.isEmpty()) {  List<Integer> subset = queue.poll();  results.add(subset);    for (int i = 0; i < nums.length; i++) {  if (subset.size() == 0 || subset.get(subset.size() - 1) < nums[i]) {  List<Integer> nextSubset = new LinkedList<Integer>(subset);  nextSubset.add(nums[i]);  queue.offer(nextSubset);  }  }  }  return results;  } } |

## Permutations

|  |
| --- |
| public class Solution {  /\*  \* @param nums: A list of integers.  \* @return: A list of permutations.  \*/  public List<List<Integer>> permute(int[] nums) {  // write your code here  List<List<Integer>> result = new ArrayList<>();  List<Integer> path = new ArrayList<>();  boolean[] visited = new boolean[nums.length];  dfs(nums, result, path, visited);  return result;  }    private void dfs(int[] nums, List<List<Integer>> result, List<Integer> path, boolean[] visited) {  if (path.size() == nums.length) {  result.add(new ArrayList<>(path));  return;  }  for (int i = 0; i < nums.length; i++) {  if (visited[i]) {  continue;  }  visited[i] = true;  path.add(nums[i]);  dfs(nums, result, path, visited);  // backtrack  path.remove(path.size() - 1);  visited[i] = false;  }  } } |

## Word Pattern II

|  |
| --- |
| public class Solution {  /\*\*  \* @param pattern: a string,denote pattern string  \* @param str: a string, denote matching string  \* @return: a boolean  \*/  public boolean wordPatternMatch(String pattern, String str) {  Map<Character, String> map = new HashMap<>();  Set<String> set = new HashSet<>();  return match(pattern, str, map, set);  }    private boolean match(String pattern,  String str,  Map<Character, String> map,  Set<String> set) {  if (pattern.length() == 0) {  return str.length() == 0;  }    Character c = pattern.charAt(0);  if (map.containsKey(c)) {  if (!str.startsWith(map.get(c))) {  return false;  }    return match(  pattern.substring(1),  str.substring(map.get(c).length()),  map,  set  );  }    for (int i = 0; i < str.length(); i++) {  String word = str.substring(0, i + 1);  if (set.contains(word)) {  continue;  }  map.put(c, word);  set.add(word);  if (match(pattern.substring(1),  str.substring(i + 1),  map,  set)) {  return true;   }  set.remove(word);  map.remove(c);  }    return false;  } } |

## Word Ladder II

|  |
| --- |
| public class Solution {  /\*  \* @param start: a string  \* @param end: a string  \* @param dict: a set of string  \* @return: a list of lists of string  \*/  List<List<String>> ans = new ArrayList<>();  Map<String, List<String>> graph = new HashMap<>();  Map<String, Integer> lb = new HashMap<>();    private void dfs(int limit, int x, String word, String end, List<String> path) {  if (x == limit + 1) {  if (word.equals(end)) {  ans.add(new ArrayList<>(path));  }  return;  }    if (x - 1 + lb.get(word) > limit) {  return;  }    for (String next : graph.get(word)) {  path.add(next);  dfs(limit, x + 1, next, end, path);  path.remove(path.size() - 1);  }    if (ans.isEmpty()) {  lb.put(word, Math.max(lb.get(word), limit - (x - 1) + 1));  }  }    private int getDiff(String a, String b) {  int ret = 0;  for (int i = 0; i < a.length(); i++) {  if (a.charAt(i) != b.charAt(i)) {  ret++;  }  }  return ret;  }    private List<String> getNext(String word, Set<String> dict) {  List<String> ret = new ArrayList<>();    for (int i = 0; i < word.length(); i++) {  char[] sc = word.toCharArray();  for (char c = 'a'; c <= 'z'; c++) {  sc[i] = c;  String next = String.valueOf(sc);  if (dict.contains(next) && !word.equals(next)) {  ret.add(next);  }  }  }  return ret;  }    public List<List<String>> findLadders(String start, String end, Set<String> dict) {  // write your code here  dict.add(start);  dict.add(end);  for (String word : dict) {  graph.put(word, getNext(word, dict));  lb.put(word, getDiff(word, end));  }      int limit = 0;  List<String> path = new ArrayList<>();  path.add(start);    while(ans.isEmpty()) {  dfs(limit, 1, start, end, path);  limit++;  }  return ans;  } } |

# 

# Hash & Heap

## Moving Average from Data Stream

|  |
| --- |
| public class MovingAverage {  /\*\*  \* Initialize your data structure here.  \*/  int id, size;  double[] sum;   MovingAverage(int s) {  id = 0;  size = s;  sum = new double[1000000]; //this is not final version  }   public double next(int val) {  // Write your code here  id++;  sum[id] = sum[id - 1] + val;  if (id - size >= 0) {  return (sum[id] - sum[id - size]) / size;  } else {  return sum[id] / id;  }  } } |

## Implement Stack by Two Queues

|  |
| --- |
| class Stack {  private Queue<Integer> queue1;  private Queue<Integer> queue2;    public Stack() {  queue1 = new LinkedList<Integer>();  queue2 = new LinkedList<Integer>();  }    private void moveItems() {  while (queue1.size() != 1) {  queue2.offer(queue1.poll());  }  }    private void swapQueues() {  Queue<Integer> temp = queue1;  queue1 = queue2;  queue2 = temp;  }    /\*\*  \* push a new item into the stack  \*/  public void push(int value) {  queue1.offer(value);  }    /\*\*  \* return the top of the stack  \*/  public int top() {  moveItems();  int item = queue1.poll();  swapQueues();  queue1.offer(item);  return item;  }    /\*\*  \* pop the top of the stack and return it  \*/  public void pop() {  moveItems();  queue1.poll();  swapQueues();  }    /\*\*  \* check the stack is empty or not.  \*/  public boolean isEmpty() {  return queue1.isEmpty();  } } |

## First Unique Character in a String

|  |
| --- |
| public class Solution {  /\*\*  \* @param str: str: the given string  \* @return: char: the first unique character in a given string  \*/  public char firstUniqChar(String str) {  // Write your code here  HashMap<Character, Integer> map = new HashMap<>();  Queue<Character> queue = new LinkedList<>();    for (char c : str.toCharArray()) {  if (map.containsKey(c)) {  map.put(c, map.get(c) + 1);  } else {  map.put(c, 1);  queue.offer(c);  }  }  while (!queue.isEmpty() && map.get(queue.peek()) != 1) {  queue.poll();  }  return queue.peek();  } } |

## Insert Delete GetRandom O(1)

|  |
| --- |
| public class RandomizedSet {  ArrayList<Integer> nums;  HashMap<Integer, Integer> nums2Int;  Random rand;  public RandomizedSet() {  nums = new ArrayList<>();  nums2Int = new HashMap<>();  rand = new Random();// do intialization if necessary  }   /\*  \* @param val: a value to the set  \* @return: true if the set did not already contain the specified element or false  \*/  public boolean insert(int val) {  if(nums.contains(val)){  return false;  }    nums2Int.put(val,nums.size());  nums.add(val);  return true;    }   /\*  \* @param val: a value from the set  \* @return: true if the set contained the specified element or false  \*/  public boolean remove(int val) {  if(nums2Int.containsKey(val) == false){  return false;  }  int index = nums2Int.get(val);  if(index < nums.size() - 1){  int last = nums.get(nums.size()-1);  nums2Int.put(last, index);  nums.set(index , last);    }  nums2Int.remove(nums.size() - 1);  nums.remove(nums.size() - 1);  return true;    }   /\*  \* @return: Get a random element from the set  \*/  public int getRandom() {  return nums.get(rand.nextInt(nums.size()));// write your code here  } } |

## K Closest Points

|  |
| --- |
| public class Solution {  /\*\*  \* @param points: a list of points  \* @param origin: a point  \* @param k: An integer  \* @return: the k closest points  \*/  public Point[] kClosest(Point[] points, Point origin, int k) {  // write your code here  PriorityQueue<Point> pq = new PriorityQueue<>(k, (a, b) -> {  int diff = getDistance(b, origin) - getDistance(a, origin);  if (diff == 0) {  diff = b.x - a.x;  }  if (diff == 0) {  diff = b.y - a.y;  }  return diff;  });    for (int i = 0; i < points.length; i++) {  pq.offer(points[i]);  if (pq.size() > k) {  pq.poll();  }  }    k = pq.size();  Point[] res = new Point[k];  while (!pq.isEmpty()) {  k--;  res[k] = pq.poll();  }  return res;  }    public int getDistance(Point a, Point b) {  return (a.x-b.x)\*(a.x-b.x) + (a.y-b.y)\*(a.y-b.y);  } } |

## Top k Largest Numbers

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums: an integer array  \* @param k: An integer  \* @return: the top k largest numbers in array  \*/  public int[] topk(int[] nums, int k) {  // write your code here  quickSort(nums, 0, nums.length - 1, k);    int[] topk = new int[k];  for(int i = 0; i < k; i++) {  topk[i] = nums[i];  }  return topk;  }    private void quickSort(int[] A, int start, int end, int k) {  if(start >= end) {  return ;  }    if(start >= k) {  return;  }    int left = start, right = end;  // pivot   Random rand = new Random(end - start + 1);  int index = rand.nextInt(end - start + 1) + start;  int pivot = A[index];    while(left <= right) {  while(left <= right && A[left] > pivot) {  left++;  }    while(left <= right && A[right] < pivot) {  right--;  }    if(left <= right) {  int temp = A[left];  A[left] = A[right];  A[right] = temp;    left++;  right--;  }  }    quickSort(A, start, right, k);  quickSort(A, left, end, k);  } } |

## Merge K Sorted Lists

|  |
| --- |
| public class Solution {  /\*\*  \* @param lists: a list of ListNode  \* @return: The head of one sorted list.  \*   private Comparator<ListNode> ListNodecomparator = new Comparator<ListNode>(){  public int compare(ListNode left, ListNode right){  return left.val - right.val;  }  };\*/  public ListNode mergeKLists(List<ListNode> lists) {     Queue<ListNode> heap = new PriorityQueue<ListNode>(lists.size(), new Comparator<ListNode> () {  @Override  public int compare(ListNode n1, ListNode n2) {  return Integer.compare(n1.val, n2.val);  }  });   for(int i = 0; i < lists.size(); i++){  if(lists.get(i) != null){  heap.add(lists.get(i));  }  }  ListNode dummy = new ListNode(0);  ListNode point = dummy;  while(!heap.isEmpty()){  ListNode cur = heap.poll();  point.next = cur;  point = point.next;  if(cur.next != null){  heap.add(cur.next);   }  }  return dummy.next;  } } |

## Implement Queue by Two Stacks

|  |
| --- |
| public class MyQueue {  private Stack<Integer> stack1;  private Stack<Integer> stack2;    public MyQueue() {  // do intialization if necessary  stack1 = new Stack<Integer>();  stack2 = new Stack<Integer>();  }   /\*  \* @param element: An integer  \* @return: nothing  \*/  public void push(int element) {  stack1.push(element);  }   /\*  \* @return: An integer  \*/    private void stack1Tostack2(){  while(!stack1.empty()){  stack2.push(stack1.peek());  stack1.pop();  }  }    public int pop() {  // write your code here  if(stack2.empty()){  stack1Tostack2();  }  return stack2.pop();  }   /\*  \* @return: An integer  \*/  public int top() {  // write your code here  if(stack2.empty()){  stack1Tostack2();  }  return stack2.peek();  } } |

## Ugly Number II

|  |
| --- |
| public class Solution {  /\*\*  \* @param n: An integer  \* @return: return a integer as description.  \*/  public int nthUglyNumber(int n) {  // write your code here  PriorityQueue<Long> minHeap;  minHeap = new PriorityQueue<Long>();  minHeap.offer(new Long(1));  int count = 0;  long lst = -1;  while(count < n){  count ++;  while(minHeap.peek() == lst) minHeap.poll();  Long now = minHeap.poll();  minHeap.offer(now \* 2);  minHeap.offer(now \* 3);  minHeap.offer(now \* 5);  lst = now;  }  return (int)lst;  } } |

# Memonization Search & Dynamic Programming

## Word Break III

|  |
| --- |
| public class Solution {  /\*  \* @param : A string  \* @param : A set of word  \* @return: the number of possible sentences.  \*/  public int wordBreak3(String s, Set<String> dict) {  int n = s.length();  String lowerS = s.toLowerCase();    Set<String> lowerDict = new HashSet<String>();  for(String str : dict) {  lowerDict.add(str.toLowerCase());  }  int[][] dp = new int[n][n];  for(int i = 0; i < n; i++){  for(int j = i; j < n;j++){  String sub = lowerS.substring(i, j + 1);  if(lowerDict.contains(sub)){  dp[i][j] = 1;  }  }  }  for(int i = 0; i < n; i++){  for(int j = i; j < n; j++){  for(int k = i; k < j; k++){  dp[i][j] += (dp[i][k] \* dp[k + 1][j]);  }  }  }  return dp[0][n - 1];  } } |

## Triangle

|  |
| --- |
| public class Solution {  /\*\*  \* @param triangle: a list of lists of integers  \* @return: An integer, minimum path sum  \*/  public int minimumTotal(int[][] triangle) {  // write your code here  if (triangle == null || triangle.length == 0) {  return 0;  }  if (triangle[0] == null || triangle[0].length == 0) {  return 0;  }    int n = triangle.length;  int[][] dp = new int[n][n];    dp[0][0] = triangle[0][0];  for (int i = 1; i < n; i++) {  dp[i][0] = dp[i-1][0] + triangle[i][0];  dp[i][i] = dp[i-1][i-1] + triangle[i][i];  }    for (int i = 1; i < n; i++) {  for (int j = 1; j < i; j++) {  dp[i][j] = Math.min(dp[i-1][j], dp[i-1][j-1]) + triangle[i][j];  }  }    int ans = dp[n-1][0];  for (int i = 1; i < n; i++) {  ans = Math.min(ans, dp[n-1][i]);  }  return ans;  } } |

## Word Break

|  |
| --- |
| public class Solution {  private int getMaxLength(Set<String> dict) {  int maxLength = 0;  for (String word : dict) {  maxLength = Math.max(maxLength, word.length());  }  return maxLength;  }   public boolean wordBreak(String s, Set<String> dict) {  if (s == null || s.length() == 0) {  return true;  }   int maxLength = getMaxLength(dict);  boolean[] canSegment = new boolean[s.length() + 1];   canSegment[0] = true;  for (int i = 1; i <= s.length(); i++) {  canSegment[i] = false;  for (int j = 1; j <= maxLength && j <= i; j++) {  if (!canSegment[i - j]) {  continue;  }  String word = s.substring(i - j, i);  if (dict.contains(word)) {  canSegment[i] = true;  break;  }  }  }   return canSegment[s.length()];  } } |

## Word Break II

|  |
| --- |
| public class Solution {  /\*  \* @param s: A string  \* @param wordDict: A set of words.  \* @return: All possible sentences.  \*/  public List<String> wordBreak(String s, Set<String> wordDict) {  // write your code here  Map<String, List<String>> memo = new HashMap<>();  memo.put("", new ArrayList<>());  memo.get("").add("");    return dfs(s, wordDict, memo);  }    private List<String> dfs(String s, Set<String> dict, Map<String, List<String>> memo) {  if (memo.containsKey(s)) {  return memo.get(s);  }    List<String> ans = new ArrayList<>();    for (int i = 1; i <= s.length(); i++) {  String s1 = s.substring(0, i);  String s2 = s.substring(i);    if (dict.contains(s1)) {  List<String> s2\_res = dfs(s2, dict, memo);  for (String item : s2\_res) {  if (item == "") {  ans.add(s1);  } else {  ans.add(s1 + " " + item);  }  }  }  }  memo.put(s, ans);  return ans;  } } |

## Wildcard Matching

|  |
| --- |
| public class Solution {  /\*\*  \* @param s: A string   \* @param p: A string includes "?" and "\*"  \* @return: is Match?  \*/  public boolean isMatch(String s, String p) {  if (s == null || p == null) {  return false;  }    boolean[][] memo = new boolean[s.length()][p.length()];  boolean[][] visited = new boolean[s.length()][p.length()];  return isMatchHelper(s, 0, p, 0, memo, visited);  }  // s[sidnex:] and p[pIndex:]  // memo[sIndex][pIndex] = true / false  // visited[sIndex][pIndex] = true  private boolean isMatchHelper(String s, int sIndex, String p, int pIndex, boolean[][] memo, boolean[][] visited) {  // 如果 p 从pIdex开始是空字符串了，那么 s 也必须从 sIndex 是空才能匹配上  if (pIndex == p.length()) {  return sIndex == s.length();  }    // 如果 s 从 sIndex 是空，那么p 必须全是 \*   if (sIndex == s.length()) {  return allStar(p, pIndex);  }    if (visited[sIndex][pIndex]) {  return memo[sIndex][pIndex];  }    char sChar = s.charAt(sIndex);  char pChar = p.charAt(pIndex);  boolean match = false;  int nextsIndex = sIndex;    if (pChar == '\*') {  // int nextpIndex = pIndex + 1;  // while(nextpIndex < p.length() && p.charAt(nextpIndex) == '\*') nextpIndex++;  // while(match == false && nextsIndex <= s.length()) {  // match = isMatchHelper(s, nextsIndex, p, nextpIndex, memo, visited);  // nextsIndex ++;  // }  match = isMatchHelper(s, sIndex, p, pIndex + 1, memo, visited) || isMatchHelper(s, sIndex + 1, p, pIndex, memo, visited);  } else {  match = charMatch(sChar, pChar) &&  isMatchHelper(s, sIndex + 1, p, pIndex + 1, memo, visited);  }    visited[sIndex][pIndex] = true;  memo[sIndex][pIndex] = match;  return match;  }    private boolean charMatch(char sChar, char pChar) {  return (sChar == pChar || pChar == '?');  }    private boolean allStar(String p, int pIndex) {  for (int i = pIndex; i < p.length(); i++) {  if (p.charAt(i) != '\*') {  return false;  }  }  return true;  } } |

## Regular Expression Matching

|  |
| --- |
| public class Solution {  /\*\*  \* @param s: A string   \* @param p: A string includes "." and "\*"  \* @return: A boolean  \*/  public boolean isMatch(String s, String p) {  if (s == null || p == null) {  return false;  }    boolean[][] memo = new boolean[s.length()][p.length()]; //记忆搜索结果  boolean[][] visited = new boolean[s.length()][p.length()]; //标记是否访问    return isMatchHelper(s, 0, p, 0, memo, visited);  }    private boolean isMatchHelper(String s, int sIndex,  String p, int pIndex,  boolean[][] memo,  boolean[][] visited) {  // "" == ""  if (pIndex == p.length()) { //如果p已经匹配完毕  return sIndex == s.length(); //根据s是否匹配完毕即可  }    if (sIndex == s.length()) { //如果s匹配完毕  return isEmpty(p, pIndex);  }    if (visited[sIndex][pIndex]) {  return memo[sIndex][pIndex];  }    char sChar = s.charAt(sIndex);  char pChar = p.charAt(pIndex);  boolean match;    // consider a\* as a bundle  if (pIndex + 1 < p.length() && p.charAt(pIndex + 1) == '\*') { //如果为'\*'，有两种方案  match = isMatchHelper(s, sIndex, p, pIndex + 2, memo, visited) || //'\*'不去匹配字符  charMatch(sChar, pChar) && isMatchHelper(s, sIndex + 1, p, pIndex, memo, visited); //'\*'重复前面一个字符去匹配s  } else {  match = charMatch(sChar, pChar) && //如果当前两字符匹配  isMatchHelper(s, sIndex + 1, p, pIndex + 1, memo, visited); //继续下一个字符匹配  }    visited[sIndex][pIndex] = true; //搜索完成就标记  memo[sIndex][pIndex] = match; //存储搜索结果  return match;  }    private boolean charMatch(char sChar, char pChar) { //判断两字符是否匹配  return sChar == pChar || pChar == '.';  }    private boolean isEmpty(String p, int pIndex) { //形如"x\*x\*"形式  for (int i = pIndex; i < p.length(); i += 2) {  if (i + 1 >= p.length() || p.charAt(i + 1) != '\*') { //如果不是'\*'，无法匹配  return false;  }  }  return true;  } } |

# 4 key points DP and coordinate DP

## Unique Paths II

|  |
| --- |
| public class Solution {  /\*\*  \* @param obstacleGrid: A list of lists of integers  \* @return: An integer  \*/  public int uniquePathsWithObstacles(int[][] obstacleGrid) {  // write your code here  if (obstacleGrid == null || obstacleGrid.length == 0) {  return 0;  }  if (obstacleGrid[0] == null || obstacleGrid[0].length == 0) {  return 0;  }  int n = obstacleGrid.length;  int m = obstacleGrid[0].length;  int[][] dp = new int[n][m];    for (int i = 0; i < n; i++) {  if (obstacleGrid[i][0] != 1) {  dp[i][0] = 1;  } else {  break;  }  }    for (int i = 0; i < m; i++) {  if (obstacleGrid[0][i] != 1) {  dp[0][i] = 1;  } else {  break;  }  }    for (int i = 1; i < n; i++) {  for (int j = 1; j < m; j++) {  if (obstacleGrid[i][j] != 1) {  dp[i][j] = dp[i-1][j] + dp[i][j-1];  } else {  dp[i][j] = 0;  }  }  }  return dp[n - 1][m - 1];    } } |

## Unique Paths

|  |
| --- |
| public class Solution {  /\*\*  \* @param m: positive integer (1 <= m <= 100)  \* @param n: positive integer (1 <= n <= 100)  \* @return: An integer  \*/  public int uniquePaths(int m, int n) {  // write your code here  if (m == 0 || n == 0) {  return 1;  }    int[][] dp = new int[m][n];    for (int i = 0; i < m; i++) {  dp[i][0] = 1;  }  for (int i = 0; i < n; i++) {  dp[0][i] = 1;  }  for (int i = 1; i < m; i++) {  for (int j = 1; j < n; j++) {  dp[i][j] = dp[i-1][j] + dp[i][j-1];  }  }    return dp[m - 1][n - 1];  } } |

## Climbing Stairs

|  |
| --- |
| public class Solution {  /\*\*  \* @param n: An integer  \* @return: An integer  \*/  int[] result = null;   void f(int X) {  if (result[X] != -1) return;   if (X == 0 || X == 1) {  result[X] = 1;  return;  }    f(X - 1);  f(X - 2);  result[X] = result[X - 1] + result[X - 2];  }   public int climbStairs(int n) {  if (n == 0) {  return 0;  }    result = new int[n + 1];  for (int i = 0; i <= n; ++i) {  result[i] = -1;  }    f(n);  return result[n];  } } |

## Largest Divisible Subset

|  |
| --- |
| public class Solution {  /\*  \* @param nums: a set of distinct positive integers  \* @return: the largest subset   \*/  public List<Integer> largestDivisibleSubset(int[] nums) {  // write your code here  Arrays.sort(nums);  int n = nums.length;    int[] dp = new int[n];  int[] pre = new int[n];    for(int i = 0; i < n; i ++){  dp[i] = 1;  pre[i] = -1;  }    int max = 0;  int index = -1;  for(int i = 0; i < n; i ++){  for(int j = 0; j < i; j ++){  if(nums[i] % nums[j] == 0 && dp[i] < dp[j] + 1){  dp[i] = dp[j] + 1;  pre[i] = j;  }  }  if(dp[i] >= max){  max = dp[i];  index = i;  }  }  List<Integer> res = new ArrayList<>();  for(int i = 0; i < max; i ++){  res.add(nums[index]);  index = pre[index];  }    return res;  } } |

## Knight Shortest Path

|  |
| --- |
| public class Solution {  /\*\*  \* @param grid: a chessboard included 0 (false) and 1 (true)  \* @param source: a point  \* @param destination: a point  \* @return: the shortest path   \*/    int[] dx = {1, 1, -1, -1, 2, 2, -2, -2};  int[] dy = {2, -2, 2, -2, 1, -1, 1, -1};  public int shortestPath(boolean[][] grid, Point source, Point destination) {      Queue<Point> queue = new LinkedList<>();  boolean[][] v = new boolean [grid.length + 1][grid[0].length + 1];  queue.offer(source);  int res =0;  while(!queue.isEmpty()){    int size = queue.size();   res++;  for(int i = 0; i < size; i++){  Point cur = queue.poll();  int x = cur.x, y = cur.y;      for(int k = 0; k < 8; k ++){    Point nextPoint = new Point(  x + dx[k],  y + dy[k]);  if(nextPoint.x < 0 || nextPoint.x >= grid.length || nextPoint.y<0 || nextPoint.y >= grid[0].length || grid[nextPoint.x][nextPoint.y] || v[nextPoint.x][nextPoint.y]){  continue;  }   if(nextPoint.x == destination.x && nextPoint.y == destination.y){  return res;  }  queue.offer(nextPoint);  v[nextPoint.x][nextPoint.y] = true;  }  }  }  return -1;    } } |

## Longest Increasing Subsequence

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums: The integer array  \* @return: The length of LIS (longest increasing subsequence)  \*/  public int longestIncreasingSubsequence(int[] nums) {  int[] minLast = new int[nums.length + 1];  minLast[0] = Integer.MIN\_VALUE;  for (int i = 1; i <= nums.length; i++) {  minLast[i] = Integer.MAX\_VALUE;  }    for (int i = 0; i < nums.length; i++) {  // find the first number in minLast >= nums[i]  int index = binarySearch(minLast, nums[i]);  minLast[index] = nums[i];  }    for (int i = nums.length; i >= 1; i--) {  if (minLast[i] != Integer.MAX\_VALUE) {  return i;  }  }    return 0;  }    // find the first number > num  private int binarySearch(int[] minLast, int num) {  int start = 0, end = minLast.length - 1;  while (start + 1 < end) {  int mid = (end - start) / 2 + start;  if (minLast[mid] < num) {  start = mid;  } else {  end = mid;  }  }    return end;  } } |

## Russian Doll Envelopes

|  |
| --- |
| public class Solution {  /\*\*  \* @param envelopes a number of envelopes with widths and heights  \* @return the maximum number of envelopes  \*/  public int maxEnvelopes(int[][] envelopes) {  // Write your code here  if(envelopes == null || envelopes.length == 0   || envelopes[0] == null || envelopes[0].length != 2)  return 0;  Arrays.sort(envelopes, new Comparator<int[]>(){  public int compare(int[] arr1, int[] arr2){  if(arr1[0] == arr2[0])  return arr2[1] - arr1[1];  else  return arr1[0] - arr2[0];  }   });  int dp[] = new int[envelopes.length];  int len = 0;  for(int[] envelope : envelopes){  int index = Arrays.binarySearch(dp, 0, len, envelope[1]);  if(index < 0)  index = -index - 1;  dp[index] = envelope[1];  if (index == len)  len++;  }  return len;  } } |

## Merge Two Sorted Interval Lists

|  |
| --- |
| public class Solution {  /\*\*  \* @param list1: one of the given list  \* @param list2: another list  \* @return: the new sorted list of interval  \*/  public List<Interval> mergeTwoInterval(List<Interval> list1, List<Interval> list2) {  List<Interval> results = new ArrayList<>();  if (list1 == null || list2 == null) {  return results;  }    Interval last = null, curt = null;  int i = 0, j = 0;  while (i < list1.size() && j < list2.size()) {  if (list1.get(i).start < list2.get(j).start) {  curt = list1.get(i);  i++;  } else {  curt = list2.get(j);  j++;  }    last = merge(results, last, curt);  }    while (i < list1.size()) {  last = merge(results, last, list1.get(i));  i++;  }    while (j < list2.size()) {  last = merge(results, last, list2.get(j));  j++;  }    if (last != null) {  results.add(last);  }  return results;  }    private Interval merge(List<Interval> results, Interval last, Interval curt) {  if (last == null) {  return curt;  }    if (curt.start > last.end) {  results.add(last);  return curt;  }    last.end = Math.max(last.end, curt.end);  return last;  } } |

## Intersection of Two Arrays

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums1: an integer array  \* @param nums2: an integer array  \* @return: an integer array  \*/  public int[] intersection(int[] nums1, int[] nums2) {  // write your code here    if(nums1 == null || nums2 == null) {  return null;  }    HashSet<Integer> resultHash = new HashSet<>();  Arrays.sort(nums2);    for(int i = 0; i < nums1.length; i++) {  if(binarySearch(nums1[i], nums2) && !resultHash.contains(nums1[i])) {  resultHash.add(nums1[i]);  }  }    int[] result = new int[resultHash.size()];  int index = 0;  for(Integer num: resultHash) {  result[index++] = num;  }  return result;  }    private boolean binarySearch(int target, int[] nums) {  if(nums == null || nums.length == 0) {  return false;  }    int start = 0, end = nums.length - 1;  while(start + 1 < end) {  int mid = (end - start) / 2 + start;  if(nums[mid] == target) {  return true;  }else if (nums[mid] < target) {  start = mid;  }else {  end = mid;  }  }    if (nums[start] == target || nums[end] == target) {  return true;  }else {  return false;  }  } } |

## Subarray Sum

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums: A list of integers  \* @return: A list of integers includes the index of the first number  \* and the index of the last number  \*/  public ArrayList<Integer> subarraySum(int[] nums) {  // write your code here    int len = nums.length;    ArrayList<Integer> ans = new ArrayList<Integer>();  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();    map.put(0, -1);    int sum = 0;  for (int i = 0; i < len; i++) {  sum += nums[i];    if (map.containsKey(sum)) {  ans.add(map.get(sum) + 1);  ans.add(i);  return ans;  }    map.put(sum, i);  }    return ans;  } } |

## Merge Sorted Array

|  |
| --- |
| public class Solution {  /\*  \* @param A: sorted integer array A which has m elements, but size of A is m+n  \* @param m: An integer  \* @param B: sorted integer array B which has n elements  \* @param n: An integer  \* @return: nothing  \*/  public void mergeSortedArray(int[] A, int m, int[] B, int n) {  // write your code here  int i = m - 1, j = n - 1, k = m + n - 1;  while (i >= 0 && j >= 0) {  if (A[i] < B[j]) {  A[k] = B[j];  j--;  k--;  } else {  A[k] = A[i];  i--;  k--;  }  }  while (i >= 0) {  A[k] = A[i];  i--;  k--;  }  while (j >= 0) {  A[k] = B[j];  j--;  k--;  }  } } |

## Maximum Subarray

|  |
| --- |
| public class Solution {  public int maxSubArray(int[] A) {  if (A == null || A.length == 0){  return 0;  }  int max = A[0], sum = 0, minSum = 0;  for (int i = 0; i < A.length; i++) {  sum += A[i];  max = Math.max(max, sum - minSum);  minSum = Math.min(minSum, sum);  }   return max;  } } |

## Maximum Submatrix

|  |
| --- |
| public class Solution {  /\*\*  \* @param matrix: the given matrix  \* @return: the largest possible sum  \*/  public int maxSubmatrix(int[][] matrix) {  // write your code here  if(matrix.length == 0 || matrix[0].length == 0){  return 0;  }  int m = matrix.length;  int n = matrix[0].length;  int[][] sum = new int[m+1][n+1];  for(int i = 1; i <= m; i++){  for(int j = 1; j <= n; j++){  sum[i][j] = sum[i-1][j] + matrix[i-1][j-1];  }  }  int max = Integer.MIN\_VALUE;  for(int i = 0; i <= m; i++){  for(int j = i+1; j <= m; j++){  int cur = 0;  for(int k = 1; k <= n; k++){  cur += (sum[j][k] - sum[i][k]);  max = Math.max(cur, max);  cur = Math.max(0, cur);  }  }  }  return max;  } } |

## Range Sum Query - Mutable

|  |
| --- |
| public class NumArray {  private int[] arr, bit;    /\*\*  \* @return: nothing  \*/  public NumArray(int[] nums) {  arr = new int[nums.length];  bit = new int[nums.length + 1];    for (int i = 0; i < nums.length; i++) {  update(i, nums[i]);  }  }    public void update(int index, int val) {  int delta = val - arr[index];  arr[index] = val;    for (int i = index + 1; i <= arr.length; i = i + lowbit(i)) {  bit[i] += delta;  }  }    public int getPrefixSum(int index) {  int sum = 0;  for (int i = index + 1; i > 0; i = i - lowbit(i)) {  sum += bit[i];  }  return sum;  }    private int lowbit(int x) {  return x & (-x);  }   public int sumRange(int left, int right) {  return getPrefixSum(right) - getPrefixSum(left - 1);  } } |

## Sparse Matrix Multiplication

|  |
| --- |
| public class Solution {  /\*\*  \* @param A a sparse matrix  \* @param B a sparse matrix  \* @return the result of A \* B  \*/  public int[][] multiply(int[][] A, int[][] B) {  // Write your code here  int n = A.length;  int m = B[0].length;  int t = A[0].length;  int[][] C = new int[n][m];   List<List<Integer>> col = new ArrayList<>();  for (int i = 0; i < t; i++) {  col.add(new ArrayList<>());  for (int j = 0; j < m; j++) {  if (B[i][j] != 0) {  col.get(i).add(j);  }  }  }  for (int i = 0; i < n; i++) {  for (int k = 0; k < t; k++) {  if (A[i][k] == 0) {  continue;  }  for (int j: col.get(k)) {  C[i][j] += A[i][k] \* B[k][j];  }  }  }  return C;  } } |

## Merge K Sorted Interval Lists

|  |
| --- |
| class Pair {  int row, col;  public Pair(int row, int col) {  this.row = row;  this.col = col;  } }  public class Solution {  /\*\*  \* @param intervals: the given k sorted interval lists  \* @return: the new sorted interval list  \*/  public List<Interval> mergeKSortedIntervalLists(List<List<Interval>> intervals) {  int k = intervals.size();  PriorityQueue<Pair> queue = new PriorityQueue<>(  k,  new Comparator<Pair>() {  public int compare(Pair e1, Pair e2) {  return intervals.get(e1.row).get(e1.col).start -   intervals.get(e2.row).get(e2.col).start;  }  }  );    for (int i = 0; i < intervals.size(); i ++) {  if (intervals.get(i).size() > 0) {  queue.add(new Pair(i, 0));  }  }    List<Interval> result = new ArrayList<>();  while (!queue.isEmpty()) {  Pair pair = queue.poll();  result.add(intervals.get(pair.row).get(pair.col));  pair.col++;  if (pair.col < intervals.get(pair.row).size()) {  queue.add(pair);  }  }    return merge(result);  }    private List<Interval> merge(List<Interval> intervals) {  if (intervals.size() <= 1) {  return intervals;  }    List<Interval> result = new ArrayList<>();  int start = intervals.get(0).start;  int end = intervals.get(0).end;  for (Interval interval : intervals) {  if (interval.start <= end) {  end = Math.max(end, interval.end);  } else {  result.add(new Interval(start, end));  start = interval.start;  end = interval.end;  }  }  // kickoff the last interval  result.add(new Interval(start, end));    return result;  } } |

## Merge K Sorted Arrays

|  |
| --- |
| class Element{  public int row, col, val;  Element(int row, int col, int val) {  this.row = row;  this.col = col;  this.val = val;  } }   public class Solution {  /\*\*  \* @param arrays: k sorted integer arrays  \* @return: a sorted array  \*/    private Comparator<Element> ElementComparator = new Comparator<Element> () {  public int compare(Element left, Element right) {  return left.val - right.val;  }  };    public int[] mergekSortedArrays(int[][] arrays) {  // write your code here  if(arrays == null) {  return new int[0];  }    int total\_size = 0;  Queue<Element> Q = new PriorityQueue<Element> (arrays.length, ElementComparator);    for(int i = 0; i < arrays.length; i++) {  if(arrays[i].length > 0) {  Element elem = new Element(i, 0, arrays[i][0]);  Q.add(elem);  total\_size += arrays[i].length;  }  }    int[] result = new int[total\_size];  int index = 0;  while(!Q.isEmpty()) {  Element elem = Q.poll();  result[index++] = elem.val;  if(elem.col + 1 < arrays[elem.row].length) {  elem.col += 1;  elem.val = arrays[elem.row][elem.col];  Q.add(elem);  }  }    return result;  } } |

## Median of K Sorted Arrays

|  |
| --- |
| public class Solution {  /\*\*  \* @param nums: the given k sorted arrays  \* @return: the median of the given k sorted arrays  \*/  public double findMedian(int[][] nums) {  int n = getTotal(nums);  if (n == 0) {  return 0;  }    if (n % 2 != 0) {  return findKth(nums, n / 2 + 1);  }    return findKth(nums, n / 2) / 2.0 + findKth(nums, n / 2 + 1) / 2.0;  }    private int getTotal(int[][] nums) {  int sum = 0;  for (int i = 0; i < nums.length; i++) {  sum += nums[i].length;  }  return sum;  }    // k is not zero-based, it starts from 1.  private int findKth(int[][] nums, int k) {  int start = 0, end = Integer.MAX\_VALUE;    // find the last number x that >= k numbers are >= x.   while (start + 1 < end) {  int mid = start + (end - start) / 2;  if (getGTE(nums, mid) >= k) {  start = mid;  } else {  end = mid;  }  }    if (getGTE(nums, end) >= k) {  return end;  }    return start;  }    // get how many numbers greater than or equal to val in 2d array  private int getGTE(int[][] nums, int val) {  int sum = 0;  for (int i = 0; i < nums.length; i++) {  sum += getGTE(nums[i], val);  }  return sum;  }    // get how many numbers greater than or equal to val in an array  private int getGTE(int[] nums, int val) {  if (nums == null || nums.length == 0) {  return 0;  }    int start = 0, end = nums.length - 1;    // find first element >= val   while (start + 1 < end) {  int mid = start + (end - start) / 2;  if (nums[mid] >= val) {  end = mid;  } else {  start = mid;  }  }    if (nums[start] >= val) {  return nums.length - start;  }    if (nums[end] >= val) {  return nums.length - end;  }    return 0;  } } |

## Median of two Sorted Arrays

|  |
| --- |
| public double findMedianSortedArrays(int[] A, int[] B) {  int m = A.length, n = B.length;  int l = (m + n + 1) / 2;  int r = (m + n + 2) / 2;  return (getkth(A, 0, B, 0, l) + getkth(A, 0, B, 0, r)) / 2.0;  }  public double getkth(int[] A, int aStart, int[] B, int bStart, int k) {  if (aStart > A.length - 1) return B[bStart + k - 1];   if (bStart > B.length - 1) return A[aStart + k - 1];   if (k == 1) return Math.min(A[aStart], B[bStart]);    int aMid = Integer.MAX\_VALUE, bMid = Integer.MAX\_VALUE;  if (aStart + k/2 - 1 < A.length) aMid = A[aStart + k/2 - 1];   if (bStart + k/2 - 1 < B.length) bMid = B[bStart + k/2 - 1];     if (aMid < bMid)   return getkth(A, aStart + k/2, B, bStart, k - k/2);// Check: aRight + bLeft   else   return getkth(A, aStart, B, bStart + k/2, k - k/2);// Check: bRight + aLeft } |

# Homework

## Add Two Numbers II

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* public class ListNode {  \* int val;  \* ListNode next;  \* ListNode(int x) { val = x; }  \* }  \*/ class Solution {  public ListNode addTwoNumbers(ListNode l1, ListNode l2) {  ListNode t1 = reverse(l1);  ListNode t2 = reverse(l2);  return add(t1, t2);  }    public ListNode reverse(ListNode head){  ListNode next = null;  ListNode originalNext = null;    while(head != null){  originalNext = head.next;  head.next = next;  next = head;  head = originalNext;    }  return next;  }  public ListNode add(ListNode l1, ListNode l2){  // ListNode dummy = new ListNode(-1);  ListNode temp = null;  ListNode t1 = l1;  ListNode t2 = l2;  int carry = 0;  while(t1 != null || t2 != null){  int val1 = t1 == null ? 0 : t1.val;  int val2 = t2 == null ? 0 : t2.val;  int sum = carry + val1 + val2;  carry = sum / 10;  sum = sum % 10;  ListNode newNode = new ListNode(sum);  newNode.next = temp;  temp = newNode;  if(t1 != null) t1 = t1.next;  if(t2 != null) t2 = t2.next;  }  if(carry != 0){  ListNode newNode = new ListNode(carry);  newNode.next = temp;  temp = newNode;  }  return temp;  } } |

## Search a 2D Matrix

|  |
| --- |
| public class Solution {  /\*\*  \* @param matrix, a list of lists of integers  \* @param target, an integer  \* @return a boolean, indicate whether matrix contains target  \*/  public boolean searchMatrix(int[][] matrix, int target) {  // write your code here  if (matrix == null || matrix.length == 0 || matrix[0].length == 0)  return false;  int start = 0;  int end = matrix.length \* matrix[0].length - 1;  int row = 0;  int col = 0;  while (start + 1 < end){  int mid = start + (end - start) / 2;  row = mid / matrix[0].length;  col = mid % matrix[0].length;  if (matrix[row][col] == target)  return true;  if (matrix[row][col] < target)  start = mid;  else  end = mid;  }    if (matrix[start / matrix[0].length][start % matrix[0].length] == target ||   matrix[end / matrix[0].length][end % matrix[0].length] == target)  return true;  return false;  } } |

## The Maze

|  |
| --- |
| class Solution {  int[] xBias = {0, 0, 1, -1};  int[] yBias = {1, -1, 0, 0};  public boolean hasPath(int[][] maze, int[] start, int[] destination) {  if (maze[start[0]][start[1]] == 1) {  return false;  }  Set<Integer> visited = new HashSet<>();  Queue<Integer> queue = new ArrayDeque<>();  queue.offer(start[0] \* 100 + start[1]);  int dest = destination[0] \* 100 + destination[1];  int m = maze.length;  int n = maze[0].length;    while (!queue.isEmpty()) {  int pos = queue.poll();  if (pos == dest) {  return true;  }  visited.add(pos);  int row = pos / 100;  int col = pos % 100;  for (int i = 0; i < 4; i++) {  int prevRow = row;  int prevCol = col;  while (isValid(prevRow, prevCol, m, n) && maze[prevRow][prevCol] == 0) {  prevRow += xBias[i];  prevCol += yBias[i];  }  prevRow -= xBias[i];  prevCol -= yBias[i];  int newDest = prevRow \* 100 + prevCol;  if (isValid(prevRow, prevCol, m, n) && !visited.contains(newDest)) {  queue.offer(newDest);  }  }  }  return false;  }    private boolean isValid(int newRow, int newCol, int r, int n) {  return newRow >= 0 && newRow < r && newCol >= 0 && newCol < n;  } } |

## Binary Tree Zigzag Level Order Traversal

|  |
| --- |
| class Solution {  public List<List<Integer>> zigzagLevelOrder(TreeNode root) {  List<List<Integer>> result = new ArrayList<>();  if(root == null) return result;  Queue<TreeNode> curr = new LinkedList<>();  boolean order = true;    curr.offer(root);    while(!curr.isEmpty()){  int size = curr.size();  List<Integer> level = new ArrayList<>();  for(int i = 0; i < size; i ++){  TreeNode temp = curr.poll();  if(temp == null) continue;  curr.offer(temp.left);   curr.offer(temp.right);    if(order){  level.add(temp.val);  }else{  level.add(0, temp.val);  }    }  if(level.isEmpty()) continue;  result.add(level);  order = !order;  }  return result;  } } |

## 01 Matrix

|  |
| --- |
| class Solution {  public int[][] updateMatrix(int[][] matrix) {  int[] xBias = new int[]{0,0,-1,1};  int[] yBias = new int[]{-1, 1, 0, 0};  Queue<int[]> queue = new LinkedList<>();  int[][] result = new int[matrix.length][matrix[0].length];  for (int i = 0; i < matrix.length; i++) {  for (int j = 0; j < matrix[0].length; j++) {  if (matrix[i][j] == 0) {  queue.offer(new int[] {i, j});  } else {  result[i][j] = -1;  }  }  }  int level = 1;  while (!queue.isEmpty()) {  int size = queue.size();  for (int i = 0; i < size; i++) {  int[] current = queue.poll();  for (int j = 0; j < xBias.length; j++) {  int newX = current[0] + xBias[j];  int newY = current[1] + yBias[j];  if (isValid(matrix, newX, newY)) {  if (result[newX][newY] == -1) {  queue.offer(new int[]{newX, newY});  result[newX][newY] = level;  }  }  }  }  level++;  }  return result;  }    private boolean isValid(int[][] matrix, int newX, int newY) {  return newX >= 0 && newY >= 0 && newX < matrix.length && newY < matrix[0].length;  } } |

## Binary Tree Right Side View

|  |
| --- |
| class Solution {  List<Integer> result;  public List<Integer> rightSideView(TreeNode root) {  result = new ArrayList<>();    helper(root, 0);  return result;  }    private void helper(TreeNode node, int level) {  if (node == null) {  return;  }    if (result.size() <= level) {  result.add(node.val);  } else {  result.set(level, node.val);  }  helper(node.left, level + 1);  helper(node.right, level + 1);  } } |

## The Maze II

|  |
| --- |
| class Solution {  int[] xBias = {0, 0, 1, -1};  int[] yBias = {1, -1, 0, 0};  public int shortestDistance(int[][] maze, int[] start, int[] destination) {  if (maze[start[0]][start[1]] == 1) {  return -1;  }  Queue<Integer> queue = new ArrayDeque<>();  queue.offer(start[0] \* 100 + start[1]);  int dest = destination[0] \* 100 + destination[1];  int m = maze.length;  int n = maze[0].length;  int[][] distance = new int[m][n];  for (int[] d : distance) {  Arrays.fill(d, -1);  }  distance[start[0]][start[1]] = 0;    while (!queue.isEmpty()) {  int pos = queue.poll();  int row = pos / 100;  int col = pos % 100;  for (int i = 0; i < 4; i++) {  int prevRow = row;  int prevCol = col;  int count = distance[row][col];  while (isValid(prevRow + xBias[i], prevCol + yBias[i], maze)) {  prevRow += xBias[i];  prevCol += yBias[i];  count++;  }  int newDest = prevRow \* 100 + prevCol;  if (distance[prevRow][prevCol] == -1 || count < distance[prevRow][prevCol]) {  queue.offer(newDest);  distance[prevRow][prevCol] = count;  }  }  }  return distance[destination[0]][destination[1]];  }    private boolean isValid(int newRow, int newCol, int[][] maze) {  return newRow >= 0 && newRow < maze.length && newCol >= 0 && newCol < maze[0].length && maze[newRow][newCol] == 0;  } } |

## Post order traversal

|  |
| --- |
| public class Solution {  /\*\*  \* @param root: A Tree  \* @return: Postorder in ArrayList which contains node values.  \*/  public List<Integer> postorderTraversal(TreeNode root) {  List<Integer> res = new ArrayList<>();  Stack<TreeNode> stack = new Stack<>();  stack.push(root);    while(!stack.isEmpty()) {  TreeNode node = stack.pop();    if(node != null) {  if(node.left == null && node.right == null) {  res.add(node.val);  } else {  stack.push(new TreeNode(node.val));  }  stack.push(node.right);  stack.push(node.left);  }  }    return res;   } } |

## Jump Game

### DFS:

|  |
| --- |
| public class Solution {  public boolean canJumpFromPosition(int position, int[] nums) {  if (position == nums.length - 1) {  return true;  }   int furthestJump = Math.min(position + nums[position], nums.length - 1);  for (int nextPosition = position + 1; nextPosition <= furthestJump; nextPosition++) {  if (canJumpFromPosition(nextPosition, nums)) {  return true;  }  }   return false;  }   public boolean canJump(int[] nums) {  return canJumpFromPosition(0, nums);  } } |

### DP:

|  |
| --- |
| class Solution {  public boolean canJump(int[] nums) {  boolean[] jump = new boolean[nums.length];  jump[0] = true;  for(int i = 0; i < nums.length; i ++){  if(!jump[i]) return false;  for(int j = 1; j <= nums[i]; j ++){  if((i + j) >= nums.length) return true;  jump[i + j] = true;  }  }  return jump[nums.length - 1];  } } |

### Greedy:

|  |
| --- |
| class Solution {  public boolean canJump(int[] nums) {  int max = nums[0];  for (int i = 0; i < nums.length; i++) {  if (i > max) {  return false;  }  max = Math.max(max, nums[i] + i);  }  return true;  } } |

## Pow(x, n)

### 位运算

|  |
| --- |
| class Solution {  public double myPow(double x, int n) {  double result = 1.0;  long longN = Math.abs((long)n);   while (longN != 0) {  if ((longN & 1) == 1) {  result \*= x;  }  longN >>= 1;  x \*= x;  }  return n < 0 ? 1 / result : result;  } } |

### 普通除法

|  |
| --- |
| class Solution {  public double myPow(double x, int n) {  long longN = Math.abs((long)n);  double result = 1.0;  while (longN != 0) {  if (longN % 2 == 1) {  result \*= x;  }  x \*= x;  longN /= 2;  }  return n < 0 ? 1 / result : result;  } } |

### 递归

|  |
| --- |
| class Solution {  public double myPow(double x, int n) {  long longN = Math.abs((long)n);  double result = 1.0;  while (longN != 0) {  if (longN % 2 == 1) {  result \*= x;  }  x \*= x;  longN /= 2;  }  return n < 0 ? 1 / result : result;  } } |

## Symmetric Tree / Same Tree

|  |
| --- |
| public boolean isSymmetric(TreeNode root) {  return root==null || isSymmetricHelp(root.left, root.right); }  private boolean isSymmetricHelp(TreeNode left, TreeNode right){  if(left==null || right==null)  return left==right;  if(left.val!=right.val)  return false;  return isSymmetricHelp(left.left, right.right) && isSymmetricHelp(left.right, right.left); } |

## Permutations II

|  |
| --- |
| class Solution {  List<List<Integer>> result;  public List<List<Integer>> permuteUnique(int[] nums) {  result = new ArrayList<>();  Arrays.sort(nums);  dfs(nums, new HashSet<>(), new ArrayList<>());  return result;  }   private void dfs(int[] nums, Set<Integer> visited, List<Integer> list) {  if (visited.size() == nums.length) {  result.add(new ArrayList<>(list));  return;  }  for (int i = 0; i < nums.length; i++) {  if (visited.contains(i) || (i > 0 && nums[i] == nums[i - 1] && !visited.contains(i - 1))) {  continue;  }  visited.add(i);  list.add(nums[i]);  dfs(nums, visited, list);  visited.remove(i);  list.remove(list.size() - 1);  }  } } |

## 

## Remove Linked List Elements

|  |
| --- |
| class Solution {  public ListNode removeElements(ListNode head, int val) {  ListNode dummy = new ListNode(-1);  dummy.next = head;  ListNode prev = dummy;    while(head != null){  if(head.val == val){  prev.next = head.next;  }else{  prev = head;  }  head = head.next;  }  return dummy.next;  } } |

## Remove Duplicates from Sorted List

|  |
| --- |
| class Solution {  public ListNode deleteDuplicates(ListNode head) {   ListNode iterator = head;  ListNode prev = head;  while (iterator != null) {  if (iterator.val == prev.val) {  prev.next = iterator.next;  } else {  prev = prev.next;  }  iterator = iterator.next;  }  return head;  } } |

## Remove Duplicates from Sorted List II

|  |
| --- |
| class Solution {  public ListNode deleteDuplicates(ListNode head) {  if (head == null) {  return null;  }  ListNode dummy = new ListNode(head.val - 1);  dummy.next = head;  ListNode prev = dummy;    while (head != null && head.next != null) {  if (head.val == head.next.val) {  while (head != null && head.next != null && head.val == head.next.val) {  head = head.next;  }  prev.next = head.next;  } else {  prev = prev.next;  }  head = head.next;  }  return dummy.next;  } } |

## Intersection of Two Arrays

|  |
| --- |
| public class Solution {  public int[] intersection(int[] nums1, int[] nums2) {  Set<Integer> set = new HashSet<>();  Set<Integer> intersect = new HashSet<>();  for (int i = 0; i < nums1.length; i++) {  set.add(nums1[i]);  }  for (int i = 0; i < nums2.length; i++) {  if (set.contains(nums2[i])) {  intersect.add(nums2[i]);  }  }  int[] result = new int[intersect.size()];  int i = 0;  for (Integer num : intersect) {  result[i++] = num;  }  return result;  } } |

## Intersection of Two Arrays II

|  |
| --- |
| class Solution {  public int[] intersect(int[] nums1, int[] nums2) {  Map<Integer, Integer> map = new HashMap<>();  for (int num : nums1){  if (map.containsKey(num)){  map.put(num, map.get(num) + 1);  } else {  map.put(num, 1);  }  }  List<Integer> list = new ArrayList<>();  Integer value;  for (int num : nums2){  value = map.get(num);  if (value != null && value > 0){  list.add(num);  map.put(num, value - 1);  }  }  int i = 0;  int[] result = new int[list.size()];  for (int num : list){  result[i++] = num;  }  return result;  } } |

## Next Greater Node In Linked List

|  |
| --- |
| class Solution {  public int[] nextLargerNodes(ListNode head) {  ArrayList<Integer> A = new ArrayList<>();  for (ListNode node = head; node != null; node = node.next)  A.add(node.val);  int[] res = new int[A.size()];  Stack<Integer> stack = new Stack<>();  for (int i = 0; i < A.size(); ++i) {  while (!stack.isEmpty() && A.get(stack.peek()) < A.get(i))  res[stack.pop()] = A.get(i);  stack.push(i);  }  return res;  } } |

## Reverse Nodes in k-Group

|  |
| --- |
| class Solution {  public ListNode reverseKGroup(ListNode head, int k) {  ListNode dummy = new ListNode(-1);  dummy.next = head;  ListNode temp = dummy;  boolean reachEnd = head == null;  ListNode next = null;  ListNode originalNext = null;  while (true) {  ListNode iterator = temp.next;  for (int i = 0; i < k; i++) {  if (iterator == null) {  reachEnd = true;  // System.out.println("Reach end");  break;  }  iterator = iterator.next;  }  if (reachEnd) {  break;  }  ListNode previousHeader = temp;  iterator = temp.next;  ListNode nextHeader = null;  for (int i = 0; i < k; i++) {  // System.out.println(iterator.val);  originalNext = iterator.next;  iterator.next = next;  next = iterator;  if (nextHeader == null) {  nextHeader = iterator;  }  iterator = originalNext;  }  temp = nextHeader;  nextHeader.next = iterator;  previousHeader.next = next;  }  return dummy.next;  } } |

## LFU

|  |
| --- |
| class LFUCache {   private int capacity;  private int minFrequency;  private Map<Integer, Integer> map;  private Map<Integer, Integer> mapFrequency;  private Map<Integer, Set<Integer>> frequencies;   public LFUCache(int capacity) {  this.capacity = capacity;  map = new HashMap<>();   mapFrequency = new HashMap<>();   frequencies = new HashMap<>();   minFrequency = 1;  }    public int get(int key) {  if (!map.containsKey(key)) {  return -1;  }  int frequency = mapFrequency.get(key);  Set<Integer> set = frequencies.get(frequency);  set.remove(key);  if (frequency == minFrequency && set.isEmpty()) {  minFrequency++;  }  frequency++;  mapFrequency.put(key, frequency);  frequencies.computeIfAbsent(frequency, k -> new LinkedHashSet<>()).add(key);  return map.get(key);  }    public void put(int key, int value) {  if (map.containsKey(key)) {  map.put(key, value);  get(key);  return;  }  map.put(key, value);  mapFrequency.put(key, 1);  frequencies.computeIfAbsent(1, k -> new LinkedHashSet<>()).add(key);  if (map.size() > capacity) {  Set<Integer> set = frequencies.get(minFrequency);  int victim = set.iterator().next();  set.remove(victim);  map.remove(victim);  mapFrequency.remove(victim);  }  minFrequency = 1;  } } |

## 构树

|  |
| --- |
| public class InPre {   private TreeNode preIn(int[] preOrder,int preLeft,int preRight,int[] inOrder,int inLeft,int inRight){  if (preLeft==preRight){  return new TreeNode(preOrder[preLeft]);  }  if (preLeft>preRight){  return null;  }   int rootIndex=-1;  int root=preOrder[preLeft];  for (int i=inLeft;i<=inRight;i++){  if (inOrder[i]==root){  rootIndex=i;  break;  }  }   int leftCount=rootIndex-inLeft;  int rightCount=inRight-rootIndex;  TreeNode left=genTree(preOrder,preLeft+1,preLeft+leftCount,inOrder,inLeft,rootIndex-1);  TreeNode right=genTree(preOrder,preLeft+leftCount+1,preRight,inOrder,rootIndex+1,inRight); // TreeNode right=genTree(preOrder,preRight-rightCount+1,preRight,inOrder,rootIndex+1,inRight);  TreeNode newRoot=new TreeNode(root);  newRoot.left=left;  newRoot.right=right;  return newRoot;  }   private TreeNode PostIn(int[] postOrder,int postLeft,int postRight,int[] inOrder,int inLeft,int inRight){  if (postLeft==postRight){  return new TreeNode(postOrder[postLeft]);  }  if (postLeft>postRight){  return null;  }   int rootIndex=-1;  int root=postOrder[postRight];  for (int i=inLeft;i<=inRight;i++){  if (inOrder[i]==root){  rootIndex=i;  break;  }  }   int leftCount=rootIndex-inLeft;  int rightCount=inRight-rootIndex;  TreeNode left=genTree1(postOrder,postLeft,postLeft+leftCount -1,inOrder,inLeft,rootIndex-1);  TreeNode right=genTree1(postOrder,postLeft+leftCount,postRight-1,inOrder,rootIndex+1,inRight); // TreeNode right=genTree(preOrder,preRight-rightCount,postRight-1,inOrder,rootIndex+1,inRight);  TreeNode newRoot=new TreeNode(root);  newRoot.left=left;  newRoot.right=right;  return newRoot;  }   public TreeNode buildTree(int[] inorder, int[] preorder) {  return genTree(preorder,0,preorder.length-1,inorder,0,inorder.length-1);  } } |

## Number of Airplanes in the Sky

|  |
| --- |
| /\*\*  \* Definition of Interval:  \* public classs Interval {  \* int start, end;  \* Interval(int start, int end) {  \* this.start = start;  \* this.end = end;  \* }  \* }  \*/    class Point{  int time;  int flag;   Point(int t, int s) {  this.time = t;  this.flag = s;  }  public static Comparator<Point> PointComparator = new Comparator<Point>() {  public int compare(Point p1, Point p2) {  if(p1.time == p2.time)   return p1.flag - p2.flag;  else   return p1.time - p2.time;  }  }; }    public class Solution {  /\*\*  \* @param airplanes: An interval array  \* @return: Count of airplanes are in the sky.  \*/  public int countOfAirplanes(List<Interval> airplanes) {    if (airplanes == null || airplanes.size() == 0) {  return 0;  }    List<Point> list = new ArrayList<>(airplanes.size() \* 2);  for (Interval i : airplanes) {  list.add(new Point(i.start, 1));  list.add(new Point(i.end, 0));  }   Collections.sort(list, Point.PointComparator);  int count = 0, ans = 1;  for (Point p : list) {  if(p.flag == 1)   count++;  else   count--;  ans = Math.max(ans, count);  }   return ans;  } } |

## Meeting Rooms

|  |
| --- |
| class Solution {  public boolean canAttendMeetings(int[][] intervals) {  Map<Integer, Integer> map = new TreeMap<>();  for (int[] interval : intervals) {  map.put(interval[0], map.getOrDefault(interval[0], 0) + 1);  map.put(interval[1], map.getOrDefault(interval[1], 0) - 1);  }    int max = 0;  for (int count : map.values()) {  max += count;  if (max > 1) {  return false;  }  }  return true;  } } |

## Meeting Rooms II

|  |
| --- |
| class Solution {  public int minMeetingRooms(int[][] intervals) {  Map<Integer, Integer> map = new TreeMap<>();  for(int[]interval : intervals){  map.put(interval[0], map.getOrDefault(interval[0], 0) + 1);  map.put(interval[1], map.getOrDefault(interval[1], 0) - 1);  }    int result = 0;  int sum = 0;    for(int value : map.values()){  sum += value;  result = Math.max(result, sum);  }  return result;  } } |

## Factor Combinations

|  |
| --- |
| class Solution {  List<List<Integer>> result;  public List<List<Integer>> getFactors(int n) {  result = new ArrayList<>();  helper(n, 2, new ArrayList<>());  return result;  }    private void helper(int n, int diviser, List<Integer> list) {  if (n == 1 && list.size() > 1) {  result.add(new ArrayList<>(list));  return;  }  for (int i = diviser; i <= n; i++) {  if (n % i != 0) {  continue;  }  list.add(i);  helper(n / i, i, list);  list.remove(list.size() - 1);  }  } } |

## Binary Tree Paths

|  |
| --- |
| class Solution {  List<List<Integer>> result;  public List<String> binaryTreePaths(TreeNode root) {  result = new ArrayList<>();  helper(root, new ArrayList<>());  return convert();  }    private void helper(TreeNode node, List<Integer> list) {  if (node == null) {  return;  }  list.add(node.val);    if (node.left == null && node.right == null) {  result.add(new ArrayList<>(list));  } else {  helper(node.left, list);  helper(node.right, list);   }   list.remove(list.size() - 1);  }    private List<String> convert() {  return result.stream()  .map(list -> {  StringBuilder sb = new StringBuilder();  for (int element : list) {  sb.append(element);  sb.append("->");  }  sb.setLength(sb.length() - 2);  return sb.toString();  })  .collect(Collectors.toList());  } } |

## Inorder Successor in BST

|  |
| --- |
| class Solution {  public TreeNode inorderSuccessor(TreeNode root, TreeNode p) {  // left root right  Stack<TreeNode> stack = new Stack<>();  stack.push(root);  TreeNode previous = null;    while (!stack.isEmpty()) {  TreeNode node = stack.pop();  if (node == null) {  continue;  }  TreeNode actualLeft = node.left;  stack.push(node.right);  if (node.right == null && node.left == null) {  if (previous != null && previous.val == p.val) {  return node;  }  previous = node;  } else {  node.left = null;  node.right = null;  stack.push(node);  }  stack.push(actualLeft);  }  return null;  } } |

## Game of Life

|  |
| --- |
| class Solution {  // 0 - current die - future die  // 1 - current live - future die  // 2 - current die - future live  // 3 - current live - future live  public void gameOfLife(int[][] board) {  if (board == null || board.length == 0) {  return;  }  for (int i = 0; i < board.length; i++) {  for (int j = 0; j < board[0].length ; j++) {  int lives = countNeighbors(board, i, j);  if (board[i][j] == 1 && (lives == 2 || lives == 3)) {  board[i][j] = 3;  }  if (board[i][j] == 0 && lives == 3) {  board[i][j] = 2;  }  }  }  for (int i = 0; i < board.length; i++) {  for (int j = 0; j < board[0].length; j++) {  board[i][j] >>= 1;  }  }  }    private int countNeighbors(int[][] board, int i, int j) {  int m = board.length;  int n = board[0].length;    int lives = 0;  for (int x = Math.max(i - 1, 0); x <= Math.min(i + 1, m - 1); x++) {  for (int y = Math.max(j - 1, 0); y <= Math.min(j + 1, n - 1); y++) {  lives += board[x][y] & 1;  }  }  lives -= board[i][j] & 1;  return lives;  } } |

## Increasing Order Search Tree

### Non-recursive

|  |
| --- |
| class Solution {  public TreeNode increasingBST(TreeNode root) {  TreeNode dummy = new TreeNode(-1);  TreeNode iterator = dummy;  Stack<TreeNode> stack = new Stack<>();  stack.push(root);    while (!stack.isEmpty()) {  TreeNode node = stack.pop();  if (node == null) {  continue;  }  stack.push(node.right);  if (node.left == null && node.right == null) {  iterator.right = node;  iterator = iterator.right;  } else {  stack.push(new TreeNode(node.val));  }  stack.push(node.left);  }  return dummy.right;  } } |

### Recursive

|  |
| --- |
| public TreeNode increasingBST(TreeNode root) {  return increasingBST(root, null);  }   public TreeNode increasingBST(TreeNode root, TreeNode tail) {  if (root == null) return tail;  TreeNode res = increasingBST(root.left, root);  root.left = null;  root.right = increasingBST(root.right, tail);  return res;  } |

## Symmetric Tree

### Recursive

|  |
| --- |
| class Solution {  public boolean isSymmetric(TreeNode root) {  return root == null || helper(root.left, root.right);  }    private boolean helper(TreeNode left, TreeNode right) {  if (left == null || right == null) {  return left == right;  }  if (left.val != right.val) {  return false;  }  return helper(left.left, right.right) && helper(left.right, right.left);  } } |

### 

### Non-Recursive

|  |
| --- |
| class Solution {  public boolean isSymmetric(TreeNode root) {  if (root == null) {  return true;  }  Stack<TreeNode> stack = new Stack<>();  stack.push(root.left);  stack.push(root.right);    while (!stack.isEmpty()) {  TreeNode n1 = stack.pop();  TreeNode n2 = stack.pop();  if (n1 == null && n2 == null) {  continue;  }  if (n1 == null || n2 == null || n1.val != n2.val) {  return false;  }  stack.push(n1.left);  stack.push(n2.right);  stack.push(n1.right);  stack.push(n2.left);  }  return true;  } } |

### 

## Same Tree

### Non-Recursive

|  |
| --- |
| class Solution {  public boolean isSameTree(TreeNode p, TreeNode q) {  Stack<TreeNode> stack = new Stack<>();  stack.push(p);  stack.push(q);    while (!stack.isEmpty()) {  TreeNode n1 = stack.pop();  TreeNode n2 = stack.pop();  if (n1 == null && n2 == null) {  continue;  }  if (n1 == null || n2 == null || n1.val != n2.val) {  return false;  }  stack.push(n1.left);  stack.push(n2.left);    stack.push(n1.right);  stack.push(n2.right);  }  return true;  } } |

### Recursive

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

## Quick Select

|  |
| --- |
| class Solution {  public int findKthLargest(int[] nums, int k) {  return quickSelect(nums, 0, nums.length - 1, k);  }    private int quickSelect(int[] nums, int left, int right, int k) {   int pivot = nums[left];  int i = left, j = right;  while (i <= j) {  while (i <= j && nums[i] > pivot) {  i++;  }  while (i <= j && nums[j] < pivot) {  j--;  }  if (i <= j) {  int tmp = nums[i];  nums[i] = nums[j];  nums[j] = tmp;  i++;  j--;  }  }   if (left + k - 1 <= j) {  return quickSelect(nums, left, j, k);  }  if (left + k - 1 >= i) {  return quickSelect(nums, i, right, k - (i - left));  }  return nums[j + 1];  } } |

## 

## Union Find

|  |
| --- |
| public class UnionFind {  private int[] father=null;   public UnionFind(int n) {  father = new int[n];  for(int i = 0; i < n; ++i) {  father[i] = i;  }  }   public int find(int x) {  return father[x] == x ? x : ( father[x] = find(father[x]) );  }   public void union(int a,int b) {  int root\_a = find(a);  int root\_b = find(b);  if(root\_a != root\_b) {  father[root\_a] = root\_b;  }  } } |

## Number of Matching Subsequences

|  |
| --- |
| class Solution {  public int numMatchingSubseq(String S, String[] words) {  Map<Character, Deque<String>> map = new HashMap<>();  for (char c = 'a'; c <= 'z'; c++) {  map.putIfAbsent(c, new LinkedList<String>());  }  for (String word : words) {  map.get(word.charAt(0)).addLast(word);  }   int count = 0;  for (char c : S.toCharArray()) {  Deque<String> queue = map.get(c);  int size = queue.size();  for (int i = 0; i < size; i++) {  String word = queue.removeFirst();  if (word.length() == 1) {  count++;  } else {  map.get(word.charAt(1)).addLast(word.substring(1));  }  }  }  return count;  } } |

## 

## Flood Fill

|  |
| --- |
| class Solution {  public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {  if (image[sr][sc] == newColor) return image;  fill(image, sr, sc, image[sr][sc], newColor);  return image;  }    private void fill(int[][] image, int sr, int sc, int color, int newColor) {  if (sr < 0 || sr >= image.length || sc < 0 || sc >= image[0].length || image[sr][sc] != color) return;  image[sr][sc] = newColor;  fill(image, sr + 1, sc, color, newColor);  fill(image, sr - 1, sc, color, newColor);  fill(image, sr, sc + 1, color, newColor);  fill(image, sr, sc - 1, color, newColor);  } } |

## Keys and Rooms

|  |
| --- |
| class Solution {  public boolean canVisitAllRooms(List<List<Integer>> rooms) {  if (rooms == null || rooms.isEmpty()) {  return true;  }  Set<Integer> set = new HashSet<>(rooms.get(0));  set.add(0);    Deque<Integer> stack = new ArrayDeque<>(rooms.get(0));    while (!stack.isEmpty()) {  int room = stack.pop();  for (int key : rooms.get(room)) {  if (set.contains(key)) {  continue;  }  set.add(key);  if (set.size() == rooms.size()) {  return true;  }  stack.push(key);  }  }  return set.size() == rooms.size();  } } |

## H-Index

|  |
| --- |
| public class Solution {  public int hIndex(int[] citations) {  int n = citations.length;  int[] papers = new int[n + 1];  // counting papers for each citation number  for (int c: citations)  papers[Math.min(n, c)]++;  // finding the h-index  int k = n;  for (int s = papers[n]; k > s; s += papers[k])  k--;  return k;  } } |

## Find the Celebrity

|  |
| --- |
| public class Solution extends Relation {  public int findCelebrity(int n) {  int candidate = 0;  for(int i = 1; i < n; i++){  if(knows(candidate, i))  candidate = i;  }  for(int i = 0; i < n; i++){  if(i != candidate && (knows(candidate, i) || !knows(i, candidate))) return -1;  }  return candidate;  } } |

## Range Sum Query - Immutable

|  |
| --- |
| class NumArray {  int[] sums;  public NumArray(int[] nums) {  if (nums == null || nums.length == 0) {  return;  }  sums = new int[nums.length + 1];  sums[0] = 0;  for (int i = 1; i <= nums.length; i++) {  sums[i] = nums[i - 1] + sums[i - 1];  }  }    public int sumRange(int i, int j) {  return sums[j + 1] - sums[i];  } } |

## Maximum Size Subarray Sum Equals k

|  |
| --- |
| public int maxSubArrayLen(int[] nums, int k) {  int sum = 0, max = 0;  HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();  for (int i = 0; i < nums.length; i++) {  sum = sum + nums[i];  if (sum == k) max = i + 1;  else if (map.containsKey(sum - k)) max = Math.max(max, i - map.get(sum - k));  if (!map.containsKey(sum)) map.put(sum, i);  }  return max; } |

## Verify Preorder Serialization of a Binary Tree

|  |
| --- |
| public boolean isValidSerialization(String preorder) {  String[] nodes = preorder.split(",");  int diff = 1;  for (String node: nodes) {  if (--diff < 0) return false;  if (!node.equals("#")) diff += 2;  }  return diff == 0; } |

## Largest BST Subtree

|  |
| --- |
| public class Solution {    // return array for each node:   // [0] --> min  // [1] --> max  // [2] --> largest BST in its subtree(inclusive)    public int largestBSTSubtree(TreeNode root) {  int[] ret = largestBST(root);  return ret[2];  }    private int[] largestBST(TreeNode node){  if(node == null){  return new int[]{Integer.MAX\_VALUE, Integer.MIN\_VALUE, 0};  }  int[] left = largestBST(node.left);  int[] right = largestBST(node.right);  if(node.val > left[1] && node.val < right[0]){  return new int[]{Math.min(node.val, left[0]), Math.max(node.val, right[1]), left[2] + right[2] + 1};  }else{  return new int[]{Integer.MIN\_VALUE, Integer.MAX\_VALUE, Math.max(left[2], right[2])};  }  } } |

## Increasing Triplet Subsequence

|  |
| --- |
| public boolean increasingTriplet(int[] nums) {  // start with two largest values, as soon as we find a number bigger than both, while both have been updated, return true.  int small = Integer.MAX\_VALUE, big = Integer.MAX\_VALUE;  for (int n : nums) {  if (n <= small) { small = n; } // update small if n is smaller than both  else if (n <= big) { big = n; } // update big only if greater than small but smaller than big  else return true; // return if you find a number bigger than both  }  return false;  } |

## Top K Frequent Elements

|  |
| --- |
| class Solution {  public List<Integer> topKFrequent(int[] nums, int k) {  Map<Integer, Integer> map = new HashMap<>();  PriorityQueue<Integer> pq = new PriorityQueue<>((n1, n2) -> Integer.compare(map.get(n1), map.get(n2)));    for (int n : nums) {  map.put(n, map.getOrDefault(n, 0) + 1);  }   for (int key : map.keySet()) {  pq.offer(key);  if (pq.size() > k) {  pq.poll();  }  }   LinkedList<Integer> result = new LinkedList<>();  while (!pq.isEmpty()) {  result.addFirst(pq.poll());  }  return result;  } } |

## Surrounded Regions

|  |
| --- |
| public class Solution {    int[] unionSet; // union find set  boolean[] hasEdgeO; // whether an union has an 'O' which is on the edge of the matrix    public void solve(char[][] board) {  if(board.length == 0 || board[0].length == 0) return;    // init, every char itself is an union  int height = board.length, width = board[0].length;  unionSet = new int[height \* width];  hasEdgeO = new boolean[unionSet.length];  for(int i = 0;i<unionSet.length; i++) unionSet[i] = i;  for(int i = 0;i<hasEdgeO.length; i++){  int x = i / width, y = i % width;  hasEdgeO[i] = (board[x][y] == 'O' && (x==0 || x==height-1 || y==0 || y==width-1));  }    // iterate the matrix, for each char, union it + its upper char + its right char if they equals to each other  for(int i = 0;i<unionSet.length; i++){  int x = i / width, y = i % width, up = x - 1, right = y + 1;  if(up >= 0 && board[x][y] == board[up][y]) union(i,i-width);  if(right < width && board[x][y] == board[x][right]) union(i,i+1);  }    // for each char in the matrix, if it is an 'O' and its union doesn't has an 'edge O', the whole union should be setted as 'X'  for(int i = 0;i<unionSet.length; i++){  int x = i / width, y = i % width;  if(board[x][y] == 'O' && !hasEdgeO[findSet(i)])   board[x][y] = 'X';   }  }    private void union(int x,int y){  int rootX = findSet(x);  int rootY = findSet(y);  // if there is an union has an 'edge O',the union after merge should be marked too  boolean hasEdgeO = this.hasEdgeO[rootX] || this.hasEdgeO[rootY];  unionSet[rootX] = rootY;  this.hasEdgeO[rootY] = hasEdgeO;  }    private int findSet(int x){  if(unionSet[x] == x) return x;  unionSet[x] = findSet(unionSet[x]);  return unionSet[x];  } } |

## Kth Smallest Element in a Sorted Matrix

### 答案二分

|  |
| --- |
| class Solution {  public int kthSmallest(int[][] matrix, int k) {  int len = matrix.length;  int start = matrix[0][0];  int end = matrix[len - 1][len - 1];  while (start < end) {  int mid = start + (end - start) / 2;  int count = countAll(matrix, mid);  if (count < k) {  start = mid + 1;  } else {  end = mid;  }  }  return start;  }   private int countAll(int[][] matrix, int target) {  int count = 0;  int i = 0;  int j = matrix.length - 1;  while (i < matrix.length && j >= 0) {  if (matrix[i][j] > target) {  j--;  } else {  count += j + 1;  i++;  }  }  return count;  } } |

### Priority Queue

|  |
| --- |
| class Solution {  public int kthSmallest(int[][] matrix, int k) {  int len = matrix.length - 1;  PriorityQueue<int[]> pq = new PriorityQueue<>((i1, i2) -> Integer.compare(matrix[i1[0]][i1[1]], matrix[i2[0]][i2[1]]));  pq.offer(new int[]{0, 0});  int[] index = null;  int count = 1;  while (!pq.isEmpty()) {  index = pq.poll();  if (count == k) {  break;  }  if (index[0] < len) {  pq.offer(new int[]{index[0] + 1, index[1]});  }  if (index[0] == 0 && index[1] < len) {  pq.offer(new int[]{index[0], index[1] + 1});  }  count++;  }  return matrix[index[0]][index[1]];  } } |

## Implement Trie (Prefix Tree)

|  |
| --- |
| class Trie {  static class Node {  Map<Character, Node> nodes = new HashMap<>();  boolean isWord;  }   private Node root;  /\*\* Initialize your data structure here. \*/  public Trie() {  root = new Node();  }    /\*\* Inserts a word into the trie. \*/  public void insert(String word) {  Node iterator = root;  char[] strs = word.toCharArray();  for (char str : strs) {  iterator = iterator.nodes.computeIfAbsent(str, k -> new Node());  }  iterator.isWord = true;  }    /\*\* Returns if the word is in the trie. \*/  public boolean search(String word) {  Node iterator = root;  char[] strs = word.toCharArray();  for (char str : strs) {  iterator = iterator.nodes.get(str);  if (iterator == null) {  return false;  }  }  return iterator.isWord;  }    /\*\* Returns if there is any word in the trie that starts with the given prefix. \*/  public boolean startsWith(String prefix) {  Node iterator = root;  char[] strs = prefix.toCharArray();  for (char str : strs) {  iterator = iterator.nodes.get(str);  if (iterator == null) {  return false;  }  }  return true;  } } |

## Design Twitter

|  |
| --- |
| class Twitter {  static class Tweet {  int tweetId;  long timeStamp;   public Tweet(int tweetId, long timeStamp) {  this.tweetId = tweetId;  this.timeStamp = timeStamp;  }  }  private Map<Integer, List<Tweet>> tweets;  private Map<Integer, Set<Integer>> follows;  private int messageCount;  private long timeStamp;  /\*\* Initialize your data structure here. \*/  public Twitter() {  tweets = new HashMap<>();  follows = new HashMap<>();  messageCount = 10;  timeStamp = 0;  }    /\*\* Compose a new tweet. \*/  public void postTweet(int userId, int tweetId) {  tweets.computeIfAbsent(userId, k -> new ArrayList<>()).add(new Tweet(tweetId, timeStamp++));  }    /\*\* Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user herself. Tweets must be ordered from most recent to least recent. \*/  public List<Integer> getNewsFeed(int userId) {  PriorityQueue<Tweet> pq = new PriorityQueue<>((t1, t2) -> Long.compare(t1.timeStamp, t2.timeStamp));    Set<Integer> followees = follows.getOrDefault(userId, new HashSet<>());  followees.add(userId);  for (int followee : followees) {  if (!tweets.containsKey(followee)) {  continue;  }  for (Tweet tweet : tweets.get(followee)) {  pq.offer(tweet);  if (pq.size() > messageCount) {  pq.poll();  }  }  }   List<Integer> result = new ArrayList<>();  while (!pq.isEmpty()) {  result.add(pq.poll().tweetId);  }  Collections.reverse(result);  return result;  }    /\*\* Follower follows a followee. If the operation is invalid, it should be a no-op. \*/  public void follow(int followerId, int followeeId) {  follows.computeIfAbsent(followerId, k -> new HashSet<>()).add(followeeId);  }    /\*\* Follower unfollows a followee. If the operation is invalid, it should be a no-op. \*/  public void unfollow(int followerId, int followeeId) {  if (!follows.containsKey(followerId)) {  return;  }  follows.get(followerId).remove(followeeId);  } } |

## 

## 

## 

## 

## 

## 

## 

## 