Bloomberg LeetCode Tagged Questions : [Grind 75 - A better Blind 75 you can customize, by the author of Blind 75](https://www.techinterviewhandbook.org/grind75/?weeks=1)

138 Copy List with Random Pointer (/problems/copy-listwith-

*class Node {*

*int val;*

*Node next;*

*Node random;*

*public Node(int val) {*

*this.val = val;*

*this.next = null;*

*this.random = null;*

*}*

*}*

*public class Solution {*

*public Node copyRandomList(Node head) {*

*if (head == null) return null;*

*// Step 1: Interleave original and copied nodes*

*Node curr = head;*

*while (curr != null) {*

*Node copy = new Node(curr.val);*

*copy.next = curr.next;*

*curr.next = copy;*

*curr = copy.next;*

*}*

*// Step 2: Set random pointers for copied nodes*

*curr = head;*

*while (curr != null) {*

*if (curr.random != null) {*

*curr.next.random = curr.random.next;*

*}*

*curr = curr.next.next;*

*}*

*// Step 3: Separate the interleaved lists*

*Node dummy = new Node(0);*

*Node copyCurr = dummy;*

*curr = head;*

*while (curr != null) {*

*copyCurr.next = curr.next;*

*curr.next = curr.next.next;*

*curr = curr.next;*

*copyCurr = copyCurr.next;*

*}*

*return dummy.next;*

*}*

*}*

**Time and Space Complexity**

* **Time Complexity:** O(n), where n is the number of nodes in the list. Each step (interleaving, setting random pointers, and separating) takes O(n) time.
* **Space Complexity:** O(1) extra space (not counting the space needed for the output). We only use a few pointers, and the interleaved list reuses the original structure temporarily.

146 LRU Cache (/problems/lru-cache) 25.0% Hard

*import java.util.HashMap;*

*import java.util.Map;*

*public class LRUCache {*

*class Node {*

*int key, value;*

*Node prev, next;*

*Node(int key, int value) {*

*this.key = key;*

*this.value = value;*

*}*

*}*

*private Map<Integer, Node> cache;*

*private Node head, tail;*

*private int capacity;*

*public LRUCache(int capacity) {*

*this.capacity = capacity;*

*this.cache = new HashMap<>();*

*// Initialize dummy head and tail nodes*

*this.head = new Node(0, 0);*

*this.tail = new Node(0, 0);*

*head.next = tail;*

*tail.prev = head;*

*}*

*public int get(int key) {*

*Node node = cache.get(key);*

*if (node == null) return -1;*

*// Move to front (most recently used)*

*moveToFront(node);*

*return node.value;*

*}*

*public void put(int key, int value) {*

*Node node = cache.get(key);*

*if (node != null) {*

*// Update existing node*

*node.value = value;*

*moveToFront(node);*

*} else {*

*// Add new node*

*Node newNode = new Node(key, value);*

*cache.put(key, newNode);*

*addToFront(newNode);*

*if (cache.size() > capacity) {*

*// Remove least recently used*

*Node lru = tail.prev;*

*removeNode(lru);*

*cache.remove(lru.key);*

*}*

*}*

*}*

*private void addToFront(Node node) {*

*node.next = head.next;*

*node.prev = head;*

*head.next.prev = node;*

*head.next = node;*

*}*

*private void removeNode(Node node) {*

*node.prev.next = node.next;*

*node.next.prev = node.prev;*

*}*

*private void moveToFront(Node node) {*

*removeNode(node);*

*addToFront(node);*

*}*

*public static void main(String[] args) {*

*LRUCache cache = new LRUCache(2);*

*cache.put(1, 1); // cache is {1=1}*

*cache.put(2, 2); // cache is {1=1, 2=2}*

*System.out.println(cache.get(1)); // returns 1*

*cache.put(3, 3); // evicts key 2, cache is {1=1, 3=3}*

*System.out.println(cache.get(2)); // returns -1 (not found)*

*cache.put(4, 4); // evicts key 1, cache is {4=4, 3=3}*

*System.out.println(cache.get(1)); // returns -1 (not found)*

*System.out.println(cache.get(3)); // returns 3*

*System.out.println(cache.get(4)); // returns 4*

*}*

*}*

**Time Complexity:** O(1) for both get and put.  
**Space Complexity:** O(capacity) to store the cache.

394 Decode String (/problems/decode-string) 44.5% Medium

*import java.util.Stack;*

*public class DecodeString {*

*public String decodeString(String s) {*

*Stack<Integer> numStack = new Stack<>();*

*Stack<StringBuilder> strStack = new Stack<>();*

*StringBuilder currentString = new StringBuilder();*

*int currentNum = 0;*

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

*if (Character.isDigit(c)) {*

*currentNum = currentNum \* 10 + (c - '0');*

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

*numStack.push(currentNum);*

*strStack.push(currentString);*

*currentNum = 0;*

*currentString = new StringBuilder();*

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

*int num = numStack.pop();*

*StringBuilder prevString = strStack.pop();*

*StringBuilder repeated = new StringBuilder();*

*for (int i = 0; i < num; i++) {*

*repeated.append(currentString);*

*}*

*currentString = prevString.append(repeated);*

*} else {*

*currentString.append(c);*

*}*

*}*

*return currentString.toString();*

*}*

*public static void main(String[] args) {*

*DecodeString solution = new DecodeString();*

*String s = "3[a]2[bc]";*

*System.out.println("Decoded String: " + solution.decodeString(s)); // Output: "aaabcbc"*

*}*

*}*

**Time Complexity:** O(n \* m), where n is the length of the string and m is the maximum nested depth of brackets (due to repeated string construction).  
**Space Complexity:** O(n) for the stacks.

451 Sort Characters By Frequency (/problems/sortcharacters-

*import java.util.\*;*

*public class SortCharactersByFrequency {*

*public String frequencySort(String s) {*

*// Step 1: Count frequency of each character*

*Map<Character, Integer> freqMap = new HashMap<>();*

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

*freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);*

*}*

*// Step 2: Bucket sort based on frequency*

*int maxFreq = Collections.max(freqMap.values());*

*List<List<Character>> buckets = new ArrayList<>();*

*for (int i = 0; i <= maxFreq; i++) {*

*buckets.add(new ArrayList<>());*

*}*

*for (char c : freqMap.keySet()) {*

*int freq = freqMap.get(c);*

*buckets.get(freq).add(c);*

*}*

*// Step 3: Build result string from highest to lowest frequency*

*StringBuilder result = new StringBuilder();*

*for (int freq = maxFreq; freq > 0; freq--) {*

*for (char c : buckets.get(freq)) {*

*for (int i = 0; i < freq; i++) {*

*result.append(c);*

*}*

*}*

*}*

*return result.toString();*

*}*

*public static void main(String[] args) {*

*SortCharactersByFrequency solution = new SortCharactersByFrequency();*

*String s = "tree";*

*System.out.println("Sorted by Frequency: " + solution.frequencySort(s)); // Output: "eetr"*

*}*

*}*

**Time Complexity:** O(n + k log k), where n is the length of the string and k is the number of unique characters (due to finding the max frequency).  
**Space Complexity:** O(k) for the HashMap and buckets.

430 Flatten a Multilevel Doubly Linked List

*// Definition for a Node.*

*class Node {*

*public int val;*

*public Node prev;*

*public Node next;*

*public Node child;*

*public Node(int val) {*

*this.val = val;*

*this.prev = null;*

*this.next = null;*

*this.child = null;*

*}*

*}*

*public class FlattenMultilevelDoublyLinkedList {*

*public Node flatten(Node head) {*

*if (head == null) return null;*

*Node curr = head;*

*while (curr != null) {*

*if (curr.child == null) {*

*curr = curr.next; // Move to the next node if there's no child*

*} else {*

*// Save the next node and process the child*

*Node child = curr.child;*

*Node next = curr.next;*

*// Connect the child to the current node*

*curr.next = child;*

*child.prev = curr;*

*curr.child = null; // Clear the child pointer*

*// Find the tail of the flattened child*

*Node tail = child;*

*while (tail.next != null) {*

*tail = tail.next;*

*}*

*// Connect the tail of the child back to the next node*

*tail.next = next;*

*if (next != null) {*

*next.prev = tail;*

*}*

*curr = child; // Continue from the child*

*}*

*}*

*return head;*

*}*

*}*

**Time Complexity**: O(n), where n is the total number of nodes in the multilevel doubly linked list.

**Space Complexity**: O(1), since the solution uses in-place operations without extra data structures.

200 Number of Islands (/problems/number-of-islands) 41.1% Medium

*public class NumberOfIslands {*

*public int numIslands(char[][] grid) {*

*if (grid == null || grid.length == 0) return 0;*

*int islands = 0;*

*int rows = grid.length;*

*int cols = grid[0].length;*

*for (int r = 0; r < rows; r++) {*

*for (int c = 0; c < cols; c++) {*

*if (grid[r][c] == '1') {*

*islands++;*

*dfs(grid, r, c);*

*}*

*}*

*}*

*return islands;*

*}*

*private void dfs(char[][] grid, int r, int c) {*

*int rows = grid.length;*

*int cols = grid[0].length;*

*// Check boundaries and if the cell is water or visited*

*if (r < 0 || r >= rows || c < 0 || c >= cols || grid[r][c] != '1') {*

*return;*

*}*

*// Mark the cell as visited by changing it to '0'*

*grid[r][c] = '0';*

*// Explore all four directions*

*dfs(grid, r - 1, c); // up*

*dfs(grid, r + 1, c); // down*

*dfs(grid, r, c - 1); // left*

*dfs(grid, r, c + 1); // right*

*}*

*public static void main(String[] args) {*

*NumberOfIslands solution = new NumberOfIslands();*

*char[][] grid = {*

*{'1', '1', '1', '1', '0'},*

*{'1', '1', '0', '1', '0'},*

*{'1', '1', '0', '0', '0'},*

*{'0', '0', '0', '0', '0'}*

*};*

*System.out.println("Number of Islands: " + solution.numIslands(grid)); // Output: 1*

*}*

*}*

**Time Complexity:** O(rows \* cols), as we visit each cell at most once.  
**Space Complexity:** O(rows \* cols) in the worst case due to the recursion stack (if the entire grid is an island).

42 Trapping Rain Water (/problems/trapping-rain-water) 42.7% Hard

*public class TrappingRainWater {*

*public int trap(int[] height) {*

*if (height == null || height.length == 0) return 0;*

*int left = 0, right = height.length - 1;*

*int leftMax = 0, rightMax = 0;*

*int waterTrapped = 0;*

*while (left <= right) {*

*if (height[left] < height[right]) {*

*if (height[left] >= leftMax) {*

*leftMax = height[left];*

*} else {*

*waterTrapped += leftMax - height[left];*

*}*

*left++;*

*} else {*

*if (height[right] >= rightMax) {*

*rightMax = height[right];*

*} else {*

*waterTrapped += rightMax - height[right];*

*}*

*right--;*

*}*

*}*

*return waterTrapped;*

*}*

*public static void main(String[] args) {*

*TrappingRainWater solution = new TrappingRainWater();*

*int[] height = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};*

*System.out.println("Water Trapped: " + solution.trap(height));*

*}*

*}*

**Time Complexity**: O(n), where n is the length of the array, as we traverse the array once.

**Space Complexity**: O(1), as no additional data structures are used.

692 Top K Frequent Words (/problems/top-k-frequentwords)

98 Validate Binary Search Tree (/problems/validatebinary-

20 Valid Parentheses (/problems/valid-parentheses) 36.3% Easy

*import java.util.Stack;*

*public class ValidParentheses {*

*public boolean isValid(String s) {*

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

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

*if (c == '(' || c == '{' || c == '[') {*

*stack.push(c); // Push opening brackets onto the stack*

*} else {*

*// If the stack is empty or the top of the stack doesn't match, return false*

*if (stack.isEmpty() || !isMatching(stack.pop(), c)) {*

*return false;*

*}*

*}*

*}*

*// If the stack is empty, all brackets are matched correctly*

*return stack.isEmpty();*

*}*

*private boolean isMatching(char open, char close) {*

*return (open == '(' && close == ')') ||*

*(open == '{' && close == '}') ||*

*(open == '[' && close == ']');*

*}*

*public static void main(String[] args) {*

*ValidParentheses solution = new ValidParentheses();*

*System.out.println(solution.isValid("()[]{}")); // Output: true*

*System.out.println(solution.isValid("(]")); // Output: false*

*System.out.println(solution.isValid("([{}])")); // Output: true*

*}*

*}*

**Time Complexity**: O(n), where n is the length of the string s. Each character is processed once.

**Space Complexity**: O(n) in the worst case, when all characters are opening brackets

56 Merge Intervals (/problems/merge-intervals) 35.4% Medium

*import java.util.ArrayList;*

*import java.util.Arrays;*

*import java.util.List;*

*public class MergeIntervals {*

*public int[][] merge(int[][] intervals) {*

*if (intervals == null || intervals.length == 0) return new int[][]{};*

*// Sort intervals by start time*

*Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));*

*List<int[]> merged = new ArrayList<>();*

*int[] currentInterval = intervals[0];*

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

*if (currentInterval[1] >= intervals[i][0]) {*

*// Overlapping: merge by updating the end*

*currentInterval[1] = Math.max(currentInterval[1], intervals[i][1]);*

*} else {*

*// Non-overlapping: add current interval to result and update current*

*merged.add(currentInterval);*

*currentInterval = intervals[i];*

*}*

*}*

*// Add the last interval*

*merged.add(currentInterval);*

*return merged.toArray(new int[merged.size()][]);*

*}*

*public static void main(String[] args) {*

*MergeIntervals solution = new MergeIntervals();*

*int[][] intervals = {{1,3},{2,6},{8,10},{15,18}};*

*int[][] result = solution.merge(intervals);*

*System.out.println("Merged Intervals:");*

*for (int[] interval : result) {*

*System.out.println("[" + interval[0] + ", " + interval[1] + "]");*

*}*

*}*

*}*

**Time Complexity:** O(n log n) due to sorting.  
**Space Complexity:** O(n) to store the result.

554 Brick Wall (/problems/brick-wall) 47.5% Medium

3 Longest Substring Without Repeating Characters

114 Flatten Binary Tree to Linked List (/problems/flattenbinary-

301 Remove Invalid Parentheses (/problems/removeinvalid-

1 Two Sum (/problems/two-sum) 43.5% Easy

*import java.util.HashMap;*

*import java.util.Map;*

*public class TwoSum {*

*public int[] twoSum(int[] nums, int target) {*

*Map<Integer, Integer> map = new HashMap<>();*

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

*int complement = target - nums[i];*

*if (map.containsKey(complement)) {*

*return new int[] { map.get(complement), i };*

*}*

*map.put(nums[i], i);*

*}*

*return new int[] {}; // No solution found*

*}*

*public static void main(String[] args) {*

*TwoSum solution = new TwoSum();*

*int[] nums = {2, 7, 11, 15};*

*int target = 9;*

*int[] result = solution.twoSum(nums, target);*

*System.out.println("Indices: [" + result[0] + ", " + result[1] + "]");*

*}*

*}*

**Time Complexity:** O(n), where n is the length of the array.  
**Space Complexity:** O(n) to store the HashMap.

253 Meeting Rooms II (/problems/meeting-rooms-ii)  42.7% Medium

79 Word Search (/problems/word-search) 31.0% Medium

139 Word Break (/problems/word-break) 35.0% Medium

2 Add Two Numbers (/problems/add-two-numbers) 31.0% Medium

560 Subarray Sum Equals K (/problems/subarray-sumequals-

23 Merge k Sorted Lists (/problems/merge-k-sorted-lists) 34.0% Hard

121 Best Time to Buy and Sell Stock (/problems/best-timeto-

*public class BestTimeToBuyAndSellStock {*

*public int maxProfit(int[] prices) {*

*if (prices == null || prices.length == 0) return 0;*

*int minPrice = Integer.MAX\_VALUE;*

*int maxProfit = 0;*

*for (int price : prices) {*

*if (price < minPrice) {*

*minPrice = price; // Update the minimum price seen so far*

*} else {*

*maxProfit = Math.max(maxProfit, price - minPrice); // Calculate potential profit*

*}*

*}*

*return maxProfit;*

*}*

*public static void main(String[] args) {*

*BestTimeToBuyAndSellStock solution = new BestTimeToBuyAndSellStock();*

*int[] prices = {7, 1, 5, 3, 6, 4};*

*System.out.println("Maximum Profit: " + solution.maxProfit(prices)); // Output: 5*

*}*

*}*

**Time Complexity**: O(n), where n is the length of the prices array. The algorithm processes each price once.  
**Space Complexity**: O(1), as no extra space is used beyond a few variables.

33 Search in Rotated Sorted Array (/problems/search-inrotated-

54 Spiral Matrix (/problems/spiral-matrix) 30.1% Medium

236 Lowest Common Ancestor of a Binary Tree

15 3Sum (/problems/3sum) 23.8% Medium

322 Coin Change (/problems/coin-change) 29.9% Medium

88 Merge Sorted Array (/problems/merge-sorted-array) 35.4% Easy

270 Closest Binary Search Tree Value (/problems/closestbinary-

10 Regular Expression Matching (/problems/regularexpression-

92 Reverse Linked List II (/problems/reverse-linked-list-ii) 34.6% Medium

426 Convert Binary Search Tree to Sorted Doubly Linked

329 Longest Increasing Path in a Matrix

22 Generate Parentheses (/problems/generateparentheses)

41 First Missing Positive (/problems/first-missing-positive) 28.6% Hard

46 Permutations (/problems/permutations) 54.6% Medium

76 Minimum Window Substring (/problems/minimumwindow-

123 Best Time to Buy and Sell Stock III (/problems/besttime-

94 Binary Tree Inorder Traversal (/problems/binary-treeinorder-

122 Best Time to Buy and Sell Stock II (/problems/besttime-

210 Course Schedule II (/problems/course-schedule-ii) 34 4% Medium

210 Course Schedule II (/problems/course schedule ii) 34.4% Medium

55 Jump Game (/problems/jump-game) 31.7% Medium

215 Kth Largest Element in an Array (/problems/kth-largestelement-

238 Product of Array Except Self (/problems/product-ofarray-

17 Letter Combinations of a Phone Number

28 Implement strStr() (/problems/implement-strstr)

141 Linked List Cycle (/problems/linked-list-cycle)

21 Merge Two Sorted Lists

*// Definition for singly-linked list.*

*class ListNode {*

*int val;*

*ListNode next;*

*ListNode() {}*

*ListNode(int val) { this.val = val; }*

*ListNode(int val, ListNode next) { this.val = val; this.next = next; }*

*}*

*public class MergeTwoSortedLists {*

*public ListNode mergeTwoLists(ListNode list1, ListNode list2) {*

*// Create a dummy node to start the merged list*

*ListNode dummy = new ListNode(-1);*

*ListNode current = dummy;*

*// Traverse both lists and merge them in sorted order*

*while (list1 != null && list2 != null) {*

*if (list1.val <= list2.val) {*

*current.next = list1;*

*list1 = list1.next;*

*} else {*

*current.next = list2;*

*list2 = list2.next;*

*}*

*current = current.next;*

*}*

*// Attach the remaining nodes, if any*

*if (list1 != null) {*

*current.next = list1;*

*} else if (list2 != null) {*

*current.next = list2;*

*}*

*return dummy.next; // Return the merged list (skipping the dummy node)*

*}*

*public static void main(String[] args) {*

*// Example usage*

*ListNode list1 = new ListNode(1, new ListNode(2, new ListNode(4)));*

*ListNode list2 = new ListNode(1, new ListNode(3, new ListNode(4)));*

*MergeTwoSortedLists solution = new MergeTwoSortedLists();*

*ListNode mergedList = solution.mergeTwoLists(list1, list2);*

*// Print the merged list*

*while (mergedList != null) {*

*System.out.print(mergedList.val + " ");*

*mergedList = mergedList.next;*

*}*

*}*

*}*

**Time Complexity**: O(n + m), where n is the length of list1 and m is the length of list2, because each node is visited once.

**Space Complexity**: O(1), as the merging is done in place without extra data structures.

125 Valid Palindrome

*public class ValidPalindrome {*

*public boolean isPalindrome(String s) {*

*if (s == null) return false;*

*// Use two pointers*

*int left = 0, right = s.length() - 1;*

*while (left < right) {*

*// Skip non-alphanumeric characters*

*while (left < right && !Character.isLetterOrDigit(s.charAt(left))) {*

*left++;*

*}*

*while (left < right && !Character.isLetterOrDigit(s.charAt(right))) {*

*right--;*

*}*

*// Compare characters*

*if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right))) {*

*return false;*

*}*

*left++;*

*right--;*

*}*

*return true;*

*}*

*public static void main(String[] args) {*

*ValidPalindrome solution = new ValidPalindrome();*

*System.out.println(solution.isPalindrome("A man, a plan, a canal: Panama")); // Output: true*

*System.out.println(solution.isPalindrome("race a car")); // Output: false*

*System.out.println(solution.isPalindrome(" ")); // Output: true*

*}*

*}*

**Time Complexity**: O(n), where n is the length of the string, as each character is processed at most once.

**Space Complexity**: O(1), since the comparison is done in place without extra data structures.

226 Invert Binary Tree

*// Definition for a binary tree node.*

*class TreeNode {*

*int val;*

*TreeNode left;*

*TreeNode right;*

*TreeNode() {}*

*TreeNode(int val) {*

*this.val = val;*

*}*

*TreeNode(int val, TreeNode left, TreeNode right) {*

*this.val = val;*

*this.left = left;*

*this.right = right;*

*}*

*}*

*public class InvertBinaryTree {*

*public TreeNode invertTree(TreeNode root) {*

*if (root == null) return null; // Base case: if the tree is empty, return null*

*// Swap the left and right subtrees*

*TreeNode temp = root.left;*

*root.left = root.right;*

*root.right = temp;*

*// Recursively invert the left and right subtrees*

*invertTree(root.left);*

*invertTree(root.right);*

*return root;*

*}*

*public static void main(String[] args) {*

*// Example usage*

*TreeNode root = new TreeNode(4,*

*new TreeNode(2, new TreeNode(1), new TreeNode(3)),*

*new TreeNode(7, new TreeNode(6), new TreeNode(9))*

*);*

*InvertBinaryTree solution = new InvertBinaryTree();*

*TreeNode invertedRoot = solution.invertTree(root);*

*// Simple printing to verify the tree*

*printTree(invertedRoot); // Custom function to traverse and print the tree*

*}*

*public static void printTree(TreeNode root) {*

*if (root != null) {*

*System.out.print(root.val + " ");*

*printTree(root.left);*

*printTree(root.right);*

*}*

*}*

*}*

**Time Complexity**: O(n), where n is the number of nodes in the tree. Each node is visited once.

**Space Complexity**: O(h), where h is the height of the tree. This accounts for the recursive stack during traversal.

242 valid anagram

*import java.util.Arrays;*

*public class ValidAnagram {*

*public boolean isAnagram(String s, String t) {*

*// If lengths are different, they cannot be anagrams*

*if (s.length() != t.length()) {*

*return false;*

*}*

*// Convert both strings to character arrays, sort them, and compare*

*char[] sArray = s.toCharArray();*

*char[] tArray = t.toCharArray();*

*Arrays.sort(sArray);*

*Arrays.sort(tArray);*

*return Arrays.equals(sArray, tArray);*

*}*

*public static void main(String[] args) {*

*ValidAnagram solution = new ValidAnagram();*

*// Example cases*

*System.out.println(solution.isAnagram("anagram", "nagaram")); // Output: true*

*System.out.println(solution.isAnagram("rat", "car")); // Output: false*

*}*

*}*

**Time Complexity**: O(n log n), where n is the length of the strings, due to the sorting step.

**Space Complexity**: O(n), as we use character arrays for both strings.