LEET CODE PRACTICE QUESTIONS WITH CODES

1. Two Sum (LC 1):
2. **Input:** nums = [2,7,11,15], target = 9
3. **Output:** [0,1]
4. **Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

Program:

import java.util.HashMap;

class Solution {

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

HashMap<Integer, Integer> numMap = new HashMap<>();

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

int complement = target - nums[i];

if (numMap.containsKey(complement)) {

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

}

numMap.put(nums[i], i);

}

return new int[]{};

}

}

2. Best Time To Buy and Sell Stock(LC 121):

**Input:** prices = [7,1,5,3,6,4]

**Output:** 5

**Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Program:

class Solution {

public int maxProfit(int[] prices) {

        int minPrice = Integer.MAX\_VALUE;

        int maxProfit = 0;

for (int price : prices) {

            if (price < minPrice) {

                minPrice = price; // Update minimum price

            } else if (price - minPrice > maxProfit) {

                maxProfit = price - minPrice; // Update max profit

            }

        }

return maxProfit;

}

}

3. Move Zero’s (LC 283):

**Input:** nums = [0,1,0,3,12]

**Output:** [1,3,12,0,0]

Program:

import java.util.Scanner;

class Solution {

public void moveZeroes(int[] nums) {

int insertPos = 0;

for (int num : nums) {

if (num != 0) {

nums[insertPos++] = num;

}

}

while (insertPos < nums.length) {

nums[insertPos++] = 0;

}

}

}

4.Maximum Subarray (LC 53):

1. **Input:** nums = [-2,1,-3,4,-1,2,1,-5,4]
2. **Output:** 6
3. **Explanation:** The subarray [4,-1,2,1] has the largest sum 6.

Program:

import java.util.Scanner;

public class Solution {

public int maxSubArray(int[] nums) {

int currentSum = nums[0];

int maxSum = nums[0];

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

currentSum = Math.max(nums[i], currentSum + nums[i]);

maxSum = Math.max(maxSum, currentSum);

}

return maxSum;

}

}

5. Set Matrix Zero’s (LC 73):

1. **Input:** matrix = [[1,1,1],[1,0,1],[1,1,1]]
2. **Output:** [[1,0,1],[0,0,0],[1,0,1]]

Program:

class Solution {

public void setZeroes(int[][] matrix) {

        int m = matrix.length;

        int n = matrix[0].length;

        boolean firstRowZero = false;

        boolean firstColZero = false;

        for (int j = 0; j < n; j++) {

            if (matrix[0][j] == 0) {

                firstRowZero = true;

                break;

            }

        }

        for (int i = 0; i < m; i++) {

            if (matrix[i][0] == 0) {

                firstColZero = true;

                break;

            }

        }

        for (int i = 1; i < m; i++) {

            for (int j = 1; j < n; j++) {

                if (matrix[i][j] == 0) {

                    matrix[i][0] = 0;

                    matrix[0][j] = 0;

                }

            }

        }

        for (int i = 1; i < m; i++) {

            for (int j = 1; j < n; j++) {

                if (matrix[i][0] == 0 || matrix[0][j] == 0) {

                    matrix[i][j] = 0;

                }

            }

        }

        if (firstRowZero) {

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

                matrix[0][j] = 0;

            }

        }

        if (firstColZero) {

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

                matrix[i][0] = 0;

            }

        }

}

}

1. Product of array Except Self (LC 238):
2. **Input:** nums = [1,2,3,4]
3. **Output:** [24,12,8,6]

Program:

class Solution {

    public int[] productExceptSelf(int[] nums) {

          int n = nums.length;

        int[] result = new int[n];

        result[0] = 1;

        for (int i = 1; i < n; i++) {

            result[i] = result[i - 1] \* nums[i - 1];

        }

        int suffix = 1;

        for (int i = n - 1; i >= 0; i--) {

            result[i] \*= suffix;

            suffix \*= nums[i];

        }

return result;

    }

}

7. 3Sum (LC 15):

1. **Input:** nums = [-1,0,1,2,-1,-4]
2. **Output:** [[-1,-1,2],[-1,0,1]]
3. **Explanation:**
4. nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
5. nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
6. nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
7. The distinct triplets are [-1,0,1] and [-1,-1,2].
8. Notice that the order of the output and the order of the triplets does not matter.

Program:

class Solution {

    public List<List<Integer>> threeSum(int[] nums) {

        List<List<Integer>> result = new ArrayList<>();

        Arrays.sort(nums);

        for (int i = 0; i < nums.length - 2; i++) {

            if (i > 0 && nums[i] == nums[i - 1]) continue;

            int target = -nums[i];

            int left = i + 1;

            int right = nums.length - 1;

            while (left < right) {

                int sum = nums[left] + nums[right];

                if (sum == target) {

                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));

                    while (left < right && nums[left] == nums[left + 1]) left++;

                    while (left < right && nums[right] == nums[right - 1]) right--;

                    left++;

                    right--;

                } else if (sum < target) {

                    left++;

                } else {

                    right--;

                }

            }

        }

        return result;

    }

}

1. Find the Duplicate number (LC 287):
2. **Input:** nums = [1,3,4,2,2]
3. **Output:** 2

Program:

class Solution {

    public int findDuplicate(int[] nums) {

       int slow = nums[0];

        int fast = nums[0];

        do {

            slow = nums[slow];

            fast = nums[nums[fast]];

        } while (slow != fast);

        slow = nums[0];

        while (slow != fast) {

            slow = nums[slow];

            fast = nums[fast];

        }

        return slow;

    }

}

9.Merge Intervals (LC 56):

1. **Input:** intervals = [[1,3],[2,6],[8,10],[15,18]]
2. **Output:** [[1,6],[8,10],[15,18]]
3. **Explanation:** Since intervals [1,3] and [2,6] overlap, merge them into [1,6].

Program:

class Solution {

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

        if (intervals.length <= 1) return intervals;

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

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

        int[] current = intervals[0];

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

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

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

            } else {

                merged.add(current);

                current = intervals[i];

            }

        }

        merged.add(current);

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

    }

}

10.Kth Largest Element in an Array (LC 215):

**Input:** nums = [3,2,1,5,6,4], k = 2

1. **Output:** 5

Program:

class Solution {

    public int findKthLargest(int[] nums, int k) {

        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {

            minHeap.offer(num);

            if (minHeap.size() > k) {

                minHeap.poll();

            }

        }

        return minHeap.peek();

    }

}

11.Subarray Sum Equals K (LC 560):

1. **Input:** nums = [1,1,1], k = 2
2. **Output:** 2

Program:

class Solution {

    public int subarraySum(int[] nums, int k) {

        int count = 0, sum = 0;

        HashMap<Integer, Integer> prefixSumFreq = new HashMap<>();

        prefixSumFreq.put(0, 1);

        for (int num : nums) {

            sum += num;

            if (prefixSumFreq.containsKey(sum - k)) {

                count += prefixSumFreq.get(sum - k);

            }

            prefixSumFreq.put(sum, prefixSumFreq.getOrDefault(sum, 0) + 1);

        }

        return count;

    }

}

12.Rotate Image (LC 48):

1. **Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]
2. **Output:** [[7,4,1],[8,5,2],[9,6,3]]

Program:

class Solution {

    public void rotate(int[][] matrix) {

                int n = matrix.length;

        for (int i = 0; i < n; i++) {

            for (int j = i + 1; j < n; j++) {

                int temp = matrix[i][j];

                matrix[i][j] = matrix[j][i];

                matrix[j][i] = temp;

            }

        }

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n / 2; j++) {

                int temp = matrix[i][j];

                matrix[i][j] = matrix[i][n - 1 - j];

                matrix[i][n - 1 - j] = temp;

            }

        }

    }

}

1. Longest consecutive Sequence (LC 128):
2. **Input:** nums = [100,4,200,1,3,2]
3. **Output:** 4
4. **Explanation:** The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

Program:

class Solution {

    public int longestConsecutive(int[] nums) {

         HashSet<Integer> set = new HashSet<>();

        for (int num : nums) set.add(num);

        int longest = 0;

        for (int num : set) {

            if (!set.contains(num - 1)) {

                int currentNum = num;

                int length = 1;

                while (set.contains(currentNum + 1)) {

                    currentNum++;

                    length++;

                }

                longest = Math.max(longest, length);

            }

        }

        return longest;

    }

}

1. Trapping Rain water (LC 42):
2. **Input:** height = [0,1,0,2,1,0,1,3,2,1,2,1]
3. **Output:** 6
4. **Explanation:** The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Program:

class Solution {

    public int trap(int[] height) {

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

        int leftMax = 0, rightMax = 0, water = 0;

        while (left < right) {

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

                if (height[left] >= leftMax) {

                    leftMax = height[left];

                } else {

                    water += leftMax - height[left];

                }

                left++;

            } else {

                if (height[right] >= rightMax) {

                    rightMax = height[right];

                } else {

                    water += rightMax - height[right];

                }

                right--;

            }

        }

        return water;

    }

}

15.Sliding Window Maximum (LC 239):

1. **Input:** nums = [1,3,-1,-3,5,3,6,7], k = 3
2. **Output:** [3,3,5,5,6,7]
3. **Explanation:**
4. Window position Max
5. --------------- -----
6. [1 3 -1] -3 5 3 6 7 **3**
7. 1 [3 -1 -3] 5 3 6 7 **3**
8. 1 3 [-1 -3 5] 3 6 7  **5**
9. 1 3 -1 [-3 5 3] 6 7 **5**
10. 1 3 -1 -3 [5 3 6] 7 **6**
11. 1 3 -1 -3 5 [3 6 7] **7**

Program:

class Solution {

    public int[] maxSlidingWindow(int[] nums, int k) {

        if (nums == null || k <= 0) return new int[0];

        int n = nums.length;

        int[] result = new int[n - k + 1];

        Deque<Integer> deque = new ArrayDeque<>();

        for (int i = 0; i < n; i++) {

            while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {

                deque.pollFirst();

            }

            while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {

                deque.pollLast();

            }

            deque.offerLast(i);

            if (i >= k - 1) {

                result[i - k + 1] = nums[deque.peekFirst()];

            }

        }

        return result;

    }

}