Practice No. 11

Topic : Stack and Queue

Date : 12-07-2024

Solve the following problems

|  |  |  |
| --- | --- | --- |
| **Q.**  **No.** | **Question Detail** | **Level** |
| **1** | **Next Greater Element I**  **Problem Statement:** The next greater element of some element x in an array is the first greater element that is to the right of x in the same array.  You are given two distinct 0-indexed integer arrays nums1 and nums2, where nums1 is a subset of nums2.  For each 0 <= i < nums1.length, find the index j such that nums1[i] == nums2[j] and determine the next greater element of nums2[j] in nums2. If there is no next greater element, then the answer for this query is -1.  Return an array ans of length nums1.length such that ans[i] is the next greater element as described above.  **Example 1:**  **Input:** nums1 = [4,1,2], nums2 = [1,3,4,2]  **Output:** [-1,3,-1]  **Explanation:**  The next greater element for each value of nums1 is as follows:   * 4 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answer is -1. * 1 is underlined in nums2 = [1,3,4,2]. The next greater element is 3. * 2 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answer is -1.   **Example 2:** | Easy |

|  |  |  |
| --- | --- | --- |
|  | **Input:** nums1 = [2,4], nums2 = [1,2,3,4]  **Output:** [3,-1]  **Explanation:**  The next greater element for each value of nums1 is as follows:   * 2 is underlined in nums2 = [1,2,3,4]. The next greater element is 3. * 4 is underlined in nums2 = [1,2,3,4]. There is no next greater element, so the answer is -1.   **Constraints:**  1 <= nums1.length <= nums2.length <= 1000 0 <= nums1[i], nums2[i] <= 10^4  All integers in nums1 and nums2 are unique. All the integers of nums1 also appear in nums2.  Program:  package com.SelfPractice;  import java.util.\*;  class NextGreaterElementI {  public int[] nextGreaterElement(int[] nums1, int[] nums2) {  Map<Integer, Integer> nextGreater = new HashMap<>();  Deque<Integer> stack = new ArrayDeque<>();  for (int i = nums2.length - 1; i >= 0; i--) {  while (!stack.isEmpty() && stack.peek() <= nums2[i]) {  stack.pop();  }  if (!stack.isEmpty()) {  nextGreater.put(nums2[i], stack.peek());  } else {  nextGreater.put(nums2[i], -1);  }  stack.push(nums2[i]);  }  int[] result = new int[nums1.length];  for (int i = 0; i < nums1.length; i++) {  result[i] = nextGreater.get(nums1[i]);  }  return result;  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.print("Enter the size of nums1: ");  int n1 = scanner.nextInt();  int[] nums1 = new int[n1];  System.***out***.println("Enter elements of nums1:");  for (int i = 0; i < n1; i++) {  nums1[i] = scanner.nextInt();  }  System.***out***.print("Enter the size of nums2: ");  int n2 = scanner.nextInt();  int[] nums2 = new int[n2];  System.***out***.println("Enter elements of nums2:");  for (int i = 0; i < n2; i++) {  nums2[i] = scanner.nextInt();  }  // Compute next greater elements  NextGreaterElementI solution = new NextGreaterElementI();  int[] result = solution.nextGreaterElement(nums1, nums2);  // Output results  System.***out***.println("Next Greater Elements for nums1:");  System.***out***.println(Arrays.*toString*(result));  scanner.close();  }  }  Output:  Enter the size of nums1: 3  Enter elements of nums1:  4 1 2  Enter the size of nums2: 4  Enter elements of nums2:  1 3 4 2  Next Greater Elements for nums1:  [-1, 3, -1] |  |
| **2** | **Minimum Cost of ropes**  **Problem statement** : There are given N ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths.  The task is to connect the ropes with minimum cost. Given N size array arr[] contains the lengths of the ropes.  **Example 1:**  **Input:**  n = 4  arr[] = {4, 3, 2, 6}  **Output:**  29  **Explanation:**  We can connect the ropes in following ways.   1. First connect ropes of lengths 2 and 3. Which makes the array   {4, 5, 6}. Cost of  this operation 2+3 = 5.   1. Now connect ropes of lengths 4 and 5. Which makes the array   {9, 6}. Cost of  this operation 4+5 = 9.   1. Finally connect the two ropes and all ropes have connected. Cost | Easy |

|  |  |  |
| --- | --- | --- |
|  | of this  operation 9+6 =15  Total cost for connecting all ropes is 5+ 9 + 15 = 29. This is the optimized cost  for connecting ropes.  Other ways of connecting ropes would always have same or more cost. For example, if we connect 4 and 6 first (we get three rope of 3,2 and 10), then connect 10 and 3 (we get two rope of 13 and 2).  Finally we connect 13 and 2. Total cost in this way is 10 + 13 + 15 = 38.  **Example 2:**  **Input:**  n = 5  arr[] = {4, 2, 7, 6, 9}  **Output:**  62  **Explanation:**  First, connect ropes 4 and 2, which makes the array {6,7,6,9}.  Cost of  this operation 4+2 = 6. Next, add ropes 6 and 6, which results in  {12,7,9}.  Cost of this operation 6+6 = 12.Then, add 7 and 9, which makes the array {12,16}.  Cost of this operation 7+9 = 16. And finally, add these two which gives {28}.  Hence, the total cost is 6 + 12 + 16 + 28 = 62.  **Constraints:**  1 ≤ N ≤ 200000  1 ≤ arr[i] ≤ 10^6  Program:  package com.SelfPractice;  import java.util.PriorityQueue;  class MinimumCostOfRopes {  public static long minCostToConnectRopes(int[] ropes) {  // Edge case: if there's only one rope, no cost needed  if (ropes.length <= 1) {  return 0;  }  // Create a min-heap priority queue  PriorityQueue<Integer> minHeap = new PriorityQueue<>();  // Add all ropes to the min-heap  for (int rope : ropes) {  minHeap.offer(rope);  }  long totalCost = 0;  // While there are more than one rope in the heap  while (minHeap.size() > 1) {  // Extract two smallest ropes  int first = minHeap.poll();  int second = minHeap.poll();  int currentCost = first + second;  totalCost += currentCost;  minHeap.offer(currentCost);  }  return totalCost;  }  public static void main(String[] args) {  // Example usage  int[] arr1 = {4, 3, 2, 6};  System.***out***.println("Minimum cost for arr1: " + *minCostToConnectRopes*(arr1)); // Output: 29  int[] arr2 = {4, 2, 7, 6, 9};  System.***out***.println("Minimum cost for arr2: " + *minCostToConnectRopes*(arr2)); // Output: 62  }  }  Output:  Minimum cost for arr1: 29  Minimum cost for arr2: 62 |  |
| **3** | **Implement two stacks in an array**  **Problem Statement:** Your task is to implement 2 stacks in one array efficiently. You need to implement 4 methods. | Easy |

|  |  |  |
| --- | --- | --- |
|  | **twoStacks :** Initialize the data structures and variables to be used to implement 2 stacks in one array.  **push1** : pushes element into first stack.  **push2** : pushes element into second stack.  **pop1** : pops element from first stack and returns the popped element. If first stack is empty, it should return -1.  **pop2** : pops element from second stack and returns the popped element. If second stack is empty, it should return -1.  **Example 1:**  **Input:**  push1(2) push1(3) push2(4) pop1()  pop2()  pop2()  **Output:**  3 4 -1  **Explanation:**  push1(2) the stack1 will be {2} push1(3) the stack1 will be {2,3} push2(4) the stack2 will be {4}  pop1() the poped element will be 3 from stack1 and stack1 will be  {2}  pop2() the poped element will be 4 from stack2 and now stack2 is empty  pop2() the stack2 is now empty hence returned -1.  **Example 2: Input:** push1(1) push2(2) pop1() push1(3) pop1()  pop1() |  |

|  |  |  |
| --- | --- | --- |
|  | **Output:**  1 3 -1  **Explanation:**  push1(1) the stack1 will be {1} push2(2) the stack2 will be {2}  pop1() the poped element will be 1 from stack1 and stack1 will be empty  push1(3) the stack1 will be {3}  pop1() the poped element will be 3 from stack1 and stack1 will be empty  pop1() the stack1 is now empty hence returned -1.  **Constraints:**  1 <= Number of queries <= 10^4  1 <= Number of elements in the stack <= 100  The sum of count of elements in both the stacks < size of the given array  Program:  package com.SelfPractice;  import java.util.Scanner;  class TwoStacks {  private int[] array;  private int top1; // Top of stack 1 (starts from left)  private int top2; // Top of stack 2 (starts from right)  public TwoStacks(int size) {  array = new int[size];  top1 = -1; // Initially empty  top2 = size; // Initially empty  }  public void push1(int x) {  if (top1 + 1 < top2) { // Check if there's space in the array  array[++top1] = x;  System.***out***.println("Pushed " + x + " into Stack 1");  } else {  System.***out***.println("Stack 1 Overflow");  }  }  public void push2(int x) {  if (top2 - 1 > top1) { // Check if there's space in the array  array[--top2] = x;  System.***out***.println("Pushed " + x + " into Stack 2");  } else {  System.***out***.println("Stack 2 Overflow");  }  }  public int pop1() {  if (top1 >= 0) {  int popped = array[top1--];  System.***out***.println("Popped " + popped + " from Stack 1");  return popped;  }  System.***out***.println("Stack 1 Underflow");  return -1; // Stack 1 is empty  }  public int pop2() {  if (top2 < array.length) {  int popped = array[top2++];  System.***out***.println("Popped " + popped + " from Stack 2");  return popped;  }  System.***out***.println("Stack 2 Underflow");  return -1; // Stack 2 is empty  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.***in***);  System.***out***.print("Enter the size of the array: ");  int size = scanner.nextInt();  TwoStacks twoStacks = new TwoStacks(size);  while (true) {  System.***out***.println("\nEnter your choice:");  System.***out***.println("1. Push into Stack 1");  System.***out***.println("2. Push into Stack 2");  System.***out***.println("3. Pop from Stack 1");  System.***out***.println("4. Pop from Stack 2");  System.***out***.println("5. Exit");  int choice = scanner.nextInt();  int x;  switch (choice) {  case 1:  System.***out***.print("Enter element to push into Stack 1: ");  x = scanner.nextInt();  twoStacks.push1(x);  break;  case 2:  System.***out***.print("Enter element to push into Stack 2: ");  x = scanner.nextInt();  twoStacks.push2(x);  break;  case 3:  int popped1 = twoStacks.pop1();  if (popped1 != -1) {  System.***out***.println("Popped element from Stack 1: " + popped1);  }  break;  case 4:  int popped2 = twoStacks.pop2();  if (popped2 != -1) {  System.***out***.println("Popped element from Stack 2: " + popped2);  }  break;  case 5:  System.***out***.println("Exiting program...");  scanner.close();  return;  default:  System.***out***.println("Invalid choice. Please enter again.");  }  }  }  }  Output:  Enter the size of the array: 2  Enter your choice:  1. Push into Stack 1  2. Push into Stack 2  3. Pop from Stack 1  4. Pop from Stack 2  5. Exit  1  Enter element to push into Stack 1: 3  Pushed 3 into Stack 1  Enter your choice:  1. Push into Stack 1  2. Push into Stack 2  3. Pop from Stack 1  4. Pop from Stack 2  5. Exit  2  Enter element to push into Stack 2: 4  Pushed 4 into Stack 2  Enter your choice:  1. Push into Stack 1  2. Push into Stack 2  3. Pop from Stack 1  4. Pop from Stack 2  5. Exit  3  Popped 3 from Stack 1  Popped element from Stack 1: 3  Enter your choice:  1. Push into Stack 1  2. Push into Stack 2  3. Pop from Stack 1  4. Pop from Stack 2  5. Exit  4  Popped 4 from Stack 2  Popped element from Stack 2: 4  Enter your choice:  1. Push into Stack 1  2. Push into Stack 2  3. Pop from Stack 1  4. Pop from Stack 2  5. Exit  4  Stack 2 Underflow  Enter your choice:  1. Push into Stack 1  2. Push into Stack 2  3. Pop from Stack 1  4. Pop from Stack 2  5. Exit  5  Exiting program... |  |
| **4** | **Max Stack**  **Problem Statement:** You have to implement a special data structure “MAX\_STACK” it would be a hybrid data structure of max heap and stack. Basically, it will have all the functionality of a stack in addition to it the max stack should also give max element in a stack in O(1). you have to implement the following functions: specialPush(value): should push the value in the stack in O(1). specialPop( ) : should pop the last element from the stack in O(1). specialTop( ): should give the element at the top of the stack in O(1).  specialMax( ): should give the maximum element from all the elements that are currently present in the stack in O(1).  In addition it tries to construct it only using the stack data structure.  Four types of queries denote these operations: Type 1 : for specialPush(value) operation.  Type 2 : for specialPop( ) operation. | Easy |

|  |  |  |
| --- | --- | --- |
|  | Type 3 : for specialTop( ) operation. Type 4 : for specialMax( ) operation. **Sample Input 1 :**  10  1 5  1 4  1 6  1 1  3  4  2  2  3  4  **Sample Output 1 :**  1  6  4  5  **Explanation for Sample Input 1:**  Initialising the new stack : MaxStack Stack = new MaxStack(). Then each operation is performed as shown below.  Stack.specialPush(5) // stack = [5] Stack.specialPush(4) // stack = [5,4] Stack.specialPush(6) // stack = [5,4,6] Stack.specialPush(1) // stack = [5,4,6,1] Stack.specialTop() // returns 1, stack = [5,4,6,1] Stack.specialMax() // returns 6, stack = [5,4,6,1] Stack.specialPop() // stack = [5,4,6] Stack.specialPop() // stack = [5,4] Stack.specialTop() // returns 4, stack = [5,4] Stack.specialMax() // returns 5, stack = [5,4]  **Sample Input 2 :**  10  1 8  1 7 |  |

|  |  |  |
| --- | --- | --- |
|  | 1 13  1 10  4  2  4  2  4  3  **Sample Output 2 :**  13  13  8  7  **Explanation of Sample Input 2 :**  Initialising the new stack : MaxStack Stack = new MaxStack(). Then each operation is performed as shown below.  Stack.specialPush(8) // stack = [8] Stack.specialPush(7) // stack = [8,7] Stack.specialPush(13) // stack = [8,7,13] Stack.specialPush(10) // stack = [8,7,13,10] Stack.specialMax() // returns 13, stack = [8,7,13,10] Stack.specialPop() // stack = [8,7,13] Stack.specialMax() // return 13, stack = [8,7,13] Stack.specialPop() // stack = [8,7] Stack.specialMax() // returns 8, stack = [8,7] Stack.specialTop() // returns 7, stack = [8,7]  Program:  package com.SelfPractice;  import java.util.Stack;  class MaxStack {  private Stack<Integer> stack;  private Stack<Integer> maxStack;  public MaxStack() {  stack = new Stack<>();  maxStack = new Stack<>();  }  // Function to push element onto the stack  public void specialPush(int value) {  stack.push(value);  if (maxStack.isEmpty() || value >= maxStack.peek()) {  maxStack.push(value);  }  }  // Function to pop element from the stack  public void specialPop() {  if (!stack.isEmpty()) {  int popped = stack.pop();  if (popped == maxStack.peek()) {  maxStack.pop();  }  }  }  // Function to get the top element of the stack  public int specialTop() {  if (!stack.isEmpty()) {  return stack.peek();  }  return -1; // Stack is empty  }  // Function to get the maximum element in the stack  public int specialMax() {  if (!maxStack.isEmpty()) {  return maxStack.peek();  }  return -1; // Stack is empty  }  public static void main(String[] args) {  MaxStack stack = new MaxStack();  // Sample Input  int[][] queries = {  {1, 5},  {1, 4},  {1, 6},  {1, 1},  {3},  {4},  {2},  {2},  {3},  {4}  };  for (int[] query : queries) {  int type = query[0];  if (type == 1) {  int value = query[1];  stack.specialPush(value);  System.***out***.println("Pushed " + value + " into stack");  } else if (type == 2) {  stack.specialPop();  System.***out***.println("Popped from stack");  } else if (type == 3) {  int top = stack.specialTop();  System.***out***.println("Top of stack: " + top);  } else if (type == 4) {  int max = stack.specialMax();  System.***out***.println("Max element in stack: " + max);  }  }  }  }  Output:  Pushed 5 into stack  Pushed 4 into stack  Pushed 6 into stack  Pushed 1 into stack  Top of stack: 1  Max element in stack: 6  Popped from stack  Popped from stack  Top of stack: 4  Max element in stack: 5 |  |
| **5** | **Generate Binary Numbers**  **Problem Statement:** Given a number N. The task is to generate and print all binary numbers with decimal values from 1 to N.  **Example 1:**  Input: N = 2  Output:  1 10 | Easy |

|  |  |  |
| --- | --- | --- |
|  | Explanation:  Binary numbers from 1 to 2 are 1 and 10.  **Example 2:**  Input:  N = 5  Output:  1 10 11 100 101  Explanation:  Binary numbers from  1 to 5 are 1 , 10 , 11 , 100 and 101.  **Constraints:**  1 ≤ N ≤ 10^6  Program:  package com.SelfPractice;  import java.util.\*;  public class GenerateBinaryNumbers {  public static void generateBinaryNumbers(int N) {  Queue<String> queue = new LinkedList<>();  queue.offer("1"); // Start with "1"  for (int i = 0; i < N; i++) {  String current = queue.poll();  System.***out***.print(current + " ");  // Append '0' and '1' to current binary string and enqueue them  queue.offer(current + "0");  queue.offer(current + "1");  }  }  public static void main(String[] args) {  int N = 5; // Example: N = 5  *generateBinaryNumbers*(N);  }  }  Output:  1 10 11 100 101 |  |
| **6** | **Exclusive Time of Functions**  **Problem Statement:** On a single-threaded CPU, we execute a program containing n functions. Each function has a unique ID between 0 and n-1.  Function calls are stored in a call stack: when a function call starts, its ID is pushed onto the stack, and when a function call ends, its ID is popped off the stack. The function whose ID is at the top of the stack is the current function being executed. Each time a function starts or ends, we write a log with the ID, whether it started or ended, and the timestamp.  You are given a list logs, where logs[i] represents the ith log message formatted as a string "{function\_id}:{"start" | "end"}:{timestamp}". For example, "0:start:3" means a function call with function ID 0 started at the beginning of timestamp 3, and "1:end:2" means a function call with function ID 1 ended at the end of timestamp 2. Note that a function can be called multiple times, possibly recursively.  A function's exclusive time is the sum of execution times for all  function calls in the program. For example, if a function is called | Medium |

|  |  |  |
| --- | --- | --- |
|  | twice, one call executing for 2 time units and another call executing for 1 time unit, the exclusive time is 2 + 1 = 3.  Return the exclusive time of each function in an array, where the value at the ith index represents the exclusive time for the function with ID i.  **Example 1:**  **Input:** n = 2, logs = ["0:start:0","1:start:2","1:end:5","0:end:6"]  **Output**: [3,4]  **Explanation:**  Function 0 starts at the beginning of time 0, then it executes 2 for units of time and reaches the end of time 1.  Function 1 starts at the beginning of time 2, executes for 4 units of time, and ends at the end of time 5.  Function 0 resumes execution at the beginning of time 6 and executes for 1 unit of time.  So function 0 spends 2 + 1 = 3 units of total time executing, and function 1 spends 4 units of total time executing.  **Example 2:**  **Input:** n = 1, logs = ["0:start:0","0:start:2","0:end:5","0:start:6","0:end:6","0:end:7"] **Output:** [8]  **Explanation:**  Function 0 starts at the beginning of time 0, executes for 2 units of time, and recursively calls itself.  Function 0 (recursive call) starts at the beginning of time 2 and executes for 4 units of time.  Function 0 (initial call) resumes execution then immediately calls itself again.  Function 0 (2nd recursive call) starts at the beginning of time 6 and executes for 1 unit of time.  Function 0 (initial call) resumes execution at the beginning of time 7 and executes for 1 unit of time.  So function 0 spends 2 + 4 + 1 + 1 = 8 units of total time executing. |  |

|  |  |  |
| --- | --- | --- |
|  | **Example 3:**  **Input:** n = 2, logs = ["0:start:0","0:start:2","0:end:5","1:start:6","1:end:6","0:end:7"] **Output**: [7,1]  **Explanation:**  Function 0 starts at the beginning of time 0, executes for 2 units of time, and recursively calls itself.  Function 0 (recursive call) starts at the beginning of time 2 and executes for 4 units of time.  Function 0 (initial call) resumes execution then immediately calls function 1.  Function 1 starts at the beginning of time 6, executes 1 unit of time, and ends at the end of time 6.  Function 0 resumes execution at the beginning of time 6 and executes for 2 units of time.  So function 0 spends 2 + 4 + 1 = 7 units of total time executing, and function 1 spends 1 unit of total time executing.  **Constraints:**  1 <= n <= 100  1 <= logs.length <= 500 0 <= function\_id < n  0 <= timestamp <= 10^9  No two start events will happen at the same timestamp. No two end events will happen at the same timestamp. Each function has an "end" log for each "start" log.  Program:  package com.SelfPractice;  import java.util.\*;  public class ExclusiveTimeOfFunctions {  public static int[] exclusiveTime(int n, List<String> logs) {  int[] exclusiveTimes = new int[n];  Deque<Integer> stack = new ArrayDeque<>();  int parevTime = 0;  for (String log : logs) {  String[] parts = log.split(":");  int functionId = Integer.*parseInt*(parts[0]);  String action = parts[1];  int timestamp = Integer.*parseInt*(parts[2]);  if (action.equals("start")) {  if (!stack.isEmpty()) {  exclusiveTimes[stack.peek()] += timestamp - prevTime;  }  stack.push(functionId);  prevTime = timestamp;  } else if (action.equals("end")) {  exclusiveTimes[stack.pop()] += timestamp - prevTime + 1;  prevTime = timestamp + 1;  }  }  return exclusiveTimes;  }  public static void main(String[] args) {  int n1 = 2;  List<String> logs1 = Arrays.*asList*("0:start:0", "1:start:2", "1:end:5", "0:end:6");  int[] result1 = *exclusiveTime*(n1, logs1);  System.***out***.println(Arrays.*toString*(result1)); // Output: [3, 4]  int n2 = 1;  List<String> logs2 = Arrays.*asList*("0:start:0", "0:start:2", "0:end:5", "0:start:6", "0:end:6", "0:end:7");  int[] result2 = *exclusiveTime*(n2, logs2);  System.***out***.println(Arrays.*toString*(result2)); // Output: [8]  int n3 = 2;  List<String> logs3 = Arrays.*asList*("0:start:0", "0:start:2", "0:end:5", "1:start:6", "1:end:6", "0:end:7");  int[] result3 = *exclusiveTime*(n3, logs3);  System.***out***.println(Arrays.*toString*(result3)); // Output: [7, 1]  }  }  Output:  [3, 4]  [8]  [7, 1] |  |
| **7** | **Next Greater Element II**  **Problem Statement:** Given a circular integer array nums (i.e., the next element of nums[nums.length - 1] is nums[0]), return the next greater number for every element in nums.  The next greater number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return -1 for this number. | Medium |

|  |  |  |
| --- | --- | --- |
|  | **Example 1:**  **Input:** nums = [1,2,1]  **Output:** [2,-1,2]  **Explanation:** The first 1's next greater number is 2; The number 2 can't find next greater number.  The second 1's next greater number needs to search circularly, which is also 2.  **Example 2:**  **Input:** nums = [1,2,3,4,3]  **Output:** [2,3,4,-1,4]  **Constraints:**  1 <= nums.length <= 10^4  -10^9 <= nums[i] <= 10^9  Program:  package com.SelfPractice;  import java.util.\*;  public class NextElementII {  public static int[] nextGreaterElements(int[] nums) {  int n = nums.length;  int[] result = new int[n];  Arrays.*fill*(result, -1);  Deque<Integer> stack = new ArrayDeque<>();  // Double the array to handle circular nature  int[] doubledNums = new int[2 \* n];  for (int i = 0; i < 2 \* n; i++) {  doubledNums[i] = nums[i % n];  }  // Process the doubled array to find next greater elements  for (int i = 0; i < 2 \* n; i++) {  while (!stack.isEmpty() && doubledNums[i] > doubledNums[stack.peek()]) {  int index = stack.pop();  if (index < n) {  result[index] = doubledNums[i];  }  }  if (i < n) {  stack.push(i);  }  }  return result;  }  public static void main(String[] args) {  int[] nums1 = {1, 2, 1};  int[] result1 = *nextGreaterElements*(nums1);  System.***out***.println(Arrays.*toString*(result1)); // Output: [2, -1, 2]  int[] nums2 = {1, 2, 3, 4, 3};  int[] result2 = *nextGreaterElements*(nums2);  System.***out***.println(Arrays.*toString*(result2)); // Output: [2, 3, 4, -1, 4]  }  }  Output:  [2, -1, 2]  [2, 3, 4, -1, 4] |  |
| **8** | **Basic Calculator II**  **Problem Statement:** Given a string s which represents an expression, evaluate this expression and return its value.  The integer division should truncate toward zero.  You may assume that the given expression is always valid. All intermediate results will be in the range of [-231, 231 - 1].  Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().  **Example 1:**  **Input:** s = "3+2\*2"  **Output:** 7  **Example 2: Input:** s = " 3/2 " **Output**: 1  **Example 3:**  **Input:** s = " 3+5 / 2 "  **Output:** 5 | Medium |

|  |  |  |
| --- | --- | --- |
|  | **Constraints:**  1 <= s.length <= 3 \* 10^5  s consists of integers and operators ('+', '-', '\*', '/') separated by some number of spaces.  s represents a valid expression.  All the integers in the expression are non-negative integers in the range [0, 231 - 1].  The answer is guaranteed to fit in a 32-bit integer.  Program:  package com.SelfPractice;  import java.util.\*;  public class BasicCalculatorII {  public static int calculate(String s) {  if (s == null || s.length() == 0) {  return 0;  }  Stack<Integer> stack = new Stack<>();  char sign = '+';  int num = 0;  for (int i = 0; i < s.length(); i++) {  char c = s.charAt(i);  if (Character.*isDigit*(c)) {  num = num \* 10 + (c - '0');  }  if (!Character.*isDigit*(c) && c != ' ' || i == s.length() - 1) {  if (sign == '+') {  stack.push(num);  } else if (sign == '-') {  stack.push(-num);  } else if (sign == '\*') {  stack.push(stack.pop() \* num);  } else if (sign == '/') {  stack.push(stack.pop() / num);  }  sign = c;  num = 0;  }  }  int result = 0;  while (!stack.isEmpty()) {  result += stack.pop();  }  return result;  }  public static void main(String[] args) {  String s1 = "3+2\*2";  String s2 = " 3/2 ";  String s3 = " 3+5 / 2 ";  System.***out***.println("Result 1: " + *calculate*(s1)); // Output: 7  System.***out***.println("Result 2: " + *calculate*(s2)); // Output: 1  System.***out***.println("Result 3: " + *calculate*(s3)); // Output: 5  }  }  Output:  Result 1: 7  Result 2: 1  Result 3: 5 |  |
| **9** | **Break Number** | Medium |
|  | **Problem statement** |  |
|  | Given a number 'N', you need to find all possible unique ways to |  |
|  | represent this number as the sum of positive integers. |  |
|  | **Note** |  |
|  | 1. By unique it is meant that no other composition can be |  |
|  | expressed as a permutation of the generated composition. For eg. |  |
|  | [1, 2, 1] and [1, 1, 2] are not unique. |  |
|  | 2. You need to print all combinations in non-decreasing order for |  |
|  | eg. [1, 2, 1] or [1, 1, 2] will be printed as [1, 1, 2], however, the |  |
|  | order of printing all the sequences can be random. |  |
|  | **Sample Input 1:** |  |
|  | 4 |  |
|  | **Sample Output 1:** |  |
|  | 4 |  |
|  | 1 1 1 1 |  |
|  | 1 1 2 |  |
|  | 2 2 |  |
|  | 1 3 |  |
|  | **Explanation For Sample Input 1:** |  |
|  | Here notice that all combinations are sorted in non-decreasing |  |
|  | order and [1, 1, 2] and [1, 2, 1] are the same and printed as [1, 1, |  |
|  | 2]. |  |

|  |  |  |
| --- | --- | --- |
|  | **Note:**  1 1 1 1  2 2  4  1 3  1 1 2 is also a valid output as the order of different sequences  doesn’t matter.  **Sample Input 2:**  1  **Sample Output 2:**  1  **Constraints:**  1 <= N <= 50  Program:  package com.SelfPractice;  import java.util.\*;  public class BreakNumber {  public static void main(String[] args) {  int N = 4; // Example input  List<List<Integer>> result = *findCombinations*(N);  // Printing the result as specified  System.***out***.println(N);  for (List<Integer> combination : result) {  for (int i = 0; i < combination.size(); i++) {  System.***out***.print(combination.get(i));  if (i < combination.size() - 1) {  System.***out***.print(" ");  }  }  System.***out***.println();  }  }  public static List<List<Integer>> findCombinations(int N) {  List<List<Integer>> result = new ArrayList<>();  *backtrack*(result, new ArrayList<>(), N, 1);  return result;  }  private static void backtrack(List<List<Integer>> result, List<Integer> current, int remaining, int start) {  if (remaining == 0) {  result.add(new ArrayList<>(current));  return;  }  for (int i = start; i <= remaining; i++) {  current.add(i);  *backtrack*(result, current, remaining - i, i);  current.remove(current.size() - 1);  }  }  }  Output:  4  1 1 1 1  1 1 2  1 3  2 2  4 |  |
| **10** | **Daily Temperatures**  **Problem Statement:** Given an array of integers temperatures represents the daily temperatures, return an array answer such that answer[i] is the number of days you have to wait after the ith day to get a warmer temperature. If there is no future day for which this is possible, keep answer[i] == 0 instead.  **Example 1:**  **Input:** temperatures = [73,74,75,71,69,72,76,73]  **Output:** [1,1,4,2,1,1,0,0]  **Example 2:**  **Input:** temperatures = [30,40,50,60]  **Output**: [1,1,1,0]  **Example 3:**  **Input:** temperatures = [30,60,90]  **Output:** [1,1,0] | Medium |

|  |  |  |
| --- | --- | --- |
|  | **Constraints:**  1 <= temperatures.length <= 10^5 30 <= temperatures[i] <= 100  Program:  package com.SelfPractice;  import java.util.\*;  public class DailyTemperatures {  public static void main(String[] args) {  int[] temperatures = {73, 74, 75, 71, 69, 72, 76, 73};  int[] result = *dailyTemperatures*(temperatures);  System.***out***.println(Arrays.*toString*(result)); // Expected output: [1, 1, 4, 2, 1, 1, 0, 0]  }  public static int[] dailyTemperatures(int[] temperatures) {  int n = temperatures.length;  int[] answer = new int[n];  Deque<Integer> stack = new ArrayDeque<>();  for (int i = n - 1; i >= 0; i--) {  while (!stack.isEmpty() && temperatures[i] >= temperatures[stack.peek()]) {  stack.pop();  }  if (!stack.isEmpty()) {  answer[i] = stack.peek() - i;  }  stack.push(i);  }  return answer;  }  }  Output:  [1, 1, 4, 2, 1, 1, 0, 0] |  |
| **11** | **Number of Visible People in a Queue**  **Problem Statement:** There are n people standing in a queue, and they numbered from 0 to n - 1 in left to right order. You are given an array heights of distinct integers where heights[i] represents the height of the ith person.  A person can see another person to their right in the queue if everybody in between is shorter than both of them. More formally, the ith person can see the jth person if i < j and min(heights[i], heights[j]) > max(heights[i+1], heights[i+2], ..., heights[j-1]).  Return an array answer of length n where answer[i] is the number of people the ith person can see to their right in the queue.  **Example 1:**    **Input:** heights = [10,6,8,5,11,9]  **Output:** [3,1,2,1,1,0]  **Explanation:**  Person 0 can see person 1, 2, and 4.  Person 1 can see person 2.  Person 2 can see person 3 and 4.  Person 3 can see person 4.  Person 4 can see person 5.  Person 5 can see no one since nobody is to the right of them.  **Example 2:** | Hard |

|  |  |  |
| --- | --- | --- |
|  | **Input:** heights = [5,1,2,3,10]  **Output:** [4,1,1,1,0]  **Constraints:**  n == heights.length 1 <= n <= 10^5  1 <= heights[i] <= 10^5  All the values of heights are unique.  Program:  import java.util.\*;  public class NumberOfVisiblePeople {  public static void main(String[] args) {  int[] heights1 = {10, 6, 8, 5, 11, 9};  int[] result1 = *canSeePersonsCount*(heights1);  System.***out***.println(Arrays.*toString*(result1)); // Expected output: [3, 1, 2, 1, 1, 0]  int[] heights2 = {5, 1, 2, 3, 10};  int[] result2 = *canSeePersonsCount*(heights2);  System.***out***.println(Arrays.*toString*(result2)); // Expected output: [4, 1, 1, 1, 0]  }  public static int[] canSeePersonsCount(int[] heights) {  int n = heights.length;  int[] answer = new int[n];  Deque<Integer> stack = new ArrayDeque<>();  for (int i = n - 1; i >= 0; i--) {  while (!stack.isEmpty() && heights[i] > heights[stack.peek()]) {  answer[i]++;  stack.pop();  }  if (!stack.isEmpty()) {  answer[i]++;  }  stack.push(i);  }  return answer;  }  }  Output:  [3, 1, 2, 1, 1, 0]  [4, 1, 1, 1, 0] |  |
| **12** | **Basic Calculator III**  **Problem Statement:** Implement a basic calculator to evaluate a simple expression string.  The expression string contains only non-negative integers, '+', '-', '\*', '/' operators, and open '(' and closing parentheses ')'. The integer division should truncate toward zero.  You may assume that the given expression is always valid. All intermediate results will be in the range of [-2^31, 2^31 - 1]. Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().  **Example 1:**  **Input**: s = "1+1"  **Output:** 2  **Example 2:**  **Input:** s = "6-4/2"  **Output:** 4  **Example 3:**  **Input**: s = "2\*(5+5\*2)/3+(6/2+8)"  **Output: 2**1  **Constraints:**  1 <= s <= 10^4  s consists of digits, '+', '-', '\*', '/', '(', and ')'. s is a valid expression.  Program:  package com.SelfPractice;  import java.util.\*;  public class BasicCalculatorIII {  public static void main(String[] args) {  BasicCalculatorIII calculator = new BasicCalculatorIII();  String s1 = "1+1";  System.***out***.println(calculator.calculate(s1)); // Output: 2  String s2 = "6-4/2";  System.***out***.println(calculator.calculate(s2)); // Output: 4  String s3 = "2\*(5+5\*2)/3+(6/2+8)";  System.***out***.println(calculator.calculate(s3)); // Output: 21  }  public int calculate(String s) {  List<String> tokens = tokenize(s);  List<String> rpn = shuntingYard(tokens);  return evaluateRPN(rpn);  }  private List<String> tokenize(String s) {  List<String> tokens = new ArrayList<>();  StringBuilder sb = new StringBuilder();  for (char c : s.toCharArray()) {  if (c == ' ') {  continue;  }  if (Character.*isDigit*(c)) {  sb.append(c);  } else {  if (sb.length() > 0) {  tokens.add(sb.toString());  sb.setLength(0);  }  tokens.add(String.*valueOf*(c));  }  }  if (sb.length() > 0) {  tokens.add(sb.toString());  }  return tokens;  }  private List<String> shuntingYard(List<String> tokens) {  List<String> rpn = new ArrayList<>();  Deque<String> stack = new ArrayDeque<>();  Map<String, Integer> precedence = new HashMap<>();  precedence.put("+", 1);  precedence.put("-", 1);  precedence.put("\*", 2);  precedence.put("/", 2);  for (String token : tokens) {  if (Character.*isDigit*(token.charAt(0))) {  rpn.add(token);  } else if (token.equals("(")) {  stack.push(token);  } else if (token.equals(")")) {  while (!stack.isEmpty() && !stack.peek().equals("(")) {  rpn.add(stack.pop());  }  stack.pop(); // Pop the '('  } else {  while (!stack.isEmpty() && precedence.containsKey(stack.peek())  && precedence.get(stack.peek()) >= precedence.get(token)) {  rpn.add(stack.pop());  }  stack.push(token);  }  }  while (!stack.isEmpty()) {  rpn.add(stack.pop());  }  return rpn;  }  private int evaluateRPN(List<String> rpn) {  Deque<Integer> stack = new ArrayDeque<>();  for (String token : rpn) {  if (Character.*isDigit*(token.charAt(0))) {  stack.push(Integer.*parseInt*(token));  } else {  int operand2 = stack.pop();  int operand1 = stack.pop();  switch (token) {  case "+":  stack.push(operand1 + operand2);  break;  case "-":  stack.push(operand1 - operand2);  break;  case "\*":  stack.push(operand1 \* operand2);  break;  case "/":  stack.push(operand1 / operand2);  break;  }  }  }  return stack.pop();  }  }  Output:  2  4  21 | Hard |