Queue Problem Solving:

Problem Statement 1:

Given an integer k and a queue of integers, reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.

Constraints:

1 <= k <= queue.size()

The queue will have at most 10^5 elements.

Each element of the queue is an integer between -10^9 and 10^9.

Input:

An integer k

A queue of integers

Output:

The modified queue with the first k elements reversed

Example:

Example 1:

Input:

k = 3

queue = [10, 20, 30, 40, 50]

Output:

queue = [30, 20, 10, 40, 50]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Statement 2:

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

void push(int x) Pushes element x to the top of the stack.

int pop() Removes the element on the top of the stack and returns it.

int top() Returns the element on the top of the stack.

boolean empty() Returns true if the stack is empty, false otherwise.

Notes:

You must use only standard operations of a queue, which means that only push to back, peek/pop from front, size and is empty operations are valid.

Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

Input

["MyStack", "push", "push", "top", "pop", "empty"]

[[], [1], [2], [], [], []]

Output

[null, null, null, 2, 2, false]

Explanation

MyStack myStack = new MyStack();

myStack.push(1);

myStack.push(2);

myStack.top(); // return 2

myStack.pop(); // return 2

myStack.empty(); // return False

Constraints:

1 <= x <= 9

At most 100 calls will be made to push, pop, top, and empty.

All the calls to pop and top are valid.

Follow-up: Can you implement the stack using only one queue?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Statement 3:

The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1, respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

The number of sandwiches in the cafeteria equals the number of students. The sandwiches are placed in a stack. At each step:

If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.

Otherwise, they will leave it and go to the queue's end.

This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays, students and sandwiches, where sandwiches[i] is the type of the i​​​​​​th sandwich in the stack (i = 0 is the top of the stack) and students[j] is the preference of the j​​​​​​th student in the initial queue (j = 0 is the front of the queue). Return the number of students that are unable to eat.

Example 1:

Input: students = [1,1,0,0], sandwiches = [0,1,0,1]

Output: 0

Explanation:

- Front student leaves the top sandwich and returns to the end of the line making students = [1,0,0,1].

- Front student leaves the top sandwich and returns to the end of the line, making students = [0,0,1,1].

The front student takes the top sandwich and leaves the line making students = [0,1,1] and sandwiches = [1,0,1].

- Front student leaves the top sandwich and returns to the end of the line making students = [1,1,0].

- Front student takes the top sandwich and leaves the line making students = [1,0] and sandwiches = [0,1].

- Front student leaves the top sandwich and returns to the end of the line making students = [0,1].

- Front student takes the top sandwich and leaves the line making students = [1] and sandwiches = [1].

- Front student takes the top sandwich and leaves the line making students = [] and sandwiches = [].

Hence all students are able to eat.

Example 2:

Input: students = [1,1,1,0,0,1], sandwiches = [1,0,0,0,1,1]

Output: 3

Constraints:

1 <= students.length, sandwiches.length <= 100

students.length == sandwiches.length

sandwiches[i] is 0 or 1.

students[i] is 0 or 1.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Statement 4:

There are n people in a line queuing to buy tickets, where the 0th person is at the front of the line and the (n - 1)th person is at the back of the line.

You are given a 0-indexed integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) to buy more tickets. If a person has no tickets left to buy, the person will leave the line.

Return the time taken for the person at position k (0-indexed) to finish buying tickets.

Example 1:

Input: tickets = [2,3,2], k = 2

Output: 6

Explanation:

- In the first pass, everyone in the line buys a ticket and the line becomes [1, 2, 1].

- In the second pass, everyone in the line buys a ticket and the line becomes [0, 1, 0].

The person at position 2 has successfully bought 2 tickets and it took 3 + 3 = 6 seconds.

Example 2:

Input: tickets = [5,1,1,1], k = 0

Output: 8

Explanation:

- In the first pass, everyone in the line buys a ticket and the line becomes [4, 0, 0, 0].

- In the next 4 passes, only the person in position 0 is buying tickets.

The person at position 0 has successfully bought 5 tickets and it took 4 + 1 + 1 + 1 + 1 = 8 seconds.

Constraints:

n == tickets.length

1 <= n <= 100

1 <= tickets[i] <= 100

0 <= k < n

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Statement 5:

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Implement the MovingAverage class:

MovingAverage(int size) Initializes the object with the size of the window size.

double next(int val) Returns the moving average of the last size values of the stream.

Example 1:

Input

["MovingAverage", "next", "next", "next", "next"]

[[3], [1], [10], [3], [5]]

Output

[null, 1.0, 5.5, 4.66667, 6.0]

Explanation

MovingAverage movingAverage = new MovingAverage(3);

movingAverage.next(1); // return 1.0 = 1 / 1

movingAverage.next(10); // return 5.5 = (1 + 10) / 2

movingAverage.next(3); // return 4.66667 = (1 + 10 + 3) / 3

movingAverage.next(5); // return 6.0 = (10 + 3 + 5) / 3

Constraints:

1 <= size <= 1000

-105 <= val <= 105

At most 104 calls will be made to next.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Problem Statement 6 : Printer Job Scheduling**

A printer can handle multiple print jobs but has limited memory to store these jobs. The jobs are processed in the order they arrive. Once a job is printed, it is removed from the queue, and the next job is printed. If the printer's memory is full, no new jobs can be added until there is space.

Implement a circular queue to manage the print jobs with the following operations:

* enqueue(int job): Adds a job to the queue.
* dequeue(): Removes and returns the front job from the queue.
* front(): Returns the front job without removing it.
* isEmpty(): Checks if the queue is empty.
* isFull(): Checks if the queue is full.

#### **Constraints:**

* The maximum number of jobs in the queue is n (1 <= n <= 1000).
* Job IDs are positive integers.

#### **Examples:**

**Example 1:**

**Input:**

**["CircularQueue", "enqueue", "enqueue", "enqueue", "front", "dequeue", "enqueue", "isFull", "dequeue", "dequeue", "dequeue", "isEmpty"]**

**[[3], [1], [2], [3], [], [], [4], [], [], [], [], []]**

**Output:**

**[null, true, true, true, 1, 1, true, true, 2, 3, 4, true]**

**Example 2:**

**Input:**

**["CircularQueue", "enqueue", "enqueue", "front", "dequeue", "enqueue", "front", "dequeue", "dequeue", "isEmpty"]**

**[[2], [10], [20], [], [], [30], [], [], [], []]**

**Output:**

**[null, true, true, 10, 10, true, 20, 20, 30, true]**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Priority Queus:**

**Problem Statement 7:**

You are given a 0-indexed integer array nums of even length and there is also an empty array arr. Alice and Bob decided to play a game where in every round Alice and Bob will do one move. The rules of the game are as follows:

Every round, first Alice will remove the minimum element from nums, and then Bob does the same.

Now, first Bob will append the removed element in the array arr, and then Alice does the same.

The game continues until nums becomes empty.

Return the resulting array arr.

Example 1:

Input: nums = [5,4,2,3]

Output: [3,2,5,4]

Explanation: In round one, first Alice removes 2 and then Bob removes 3. Then in arr firstly Bob appends 3 and then Alice appends 2. So arr = [3,2].

At the begining of round two, nums = [5,4]. Now, first Alice removes 4 and then Bob removes 5. Then both append in arr which becomes [3,2,5,4].

Example 2:

Input: nums = [2,5]

Output: [5,2]

Explanation: In round one, first Alice removes 2 and then Bob removes 5. Then in arr firstly Bob appends and then Alice appends. So arr = [5,2].

Constraints:

2 <= nums.length <= 100

1 <= nums[i] <= 100

nums.length % 2 == 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Statement 8:

Given the array of integers nums, you will choose two different indices i and j of that array. Return the maximum value of (nums[i]-1)\*(nums[j]-1).

Example 1:

Input: nums = [3,4,5,2]

Output: 12

Explanation: If you choose the indices i=1 and j=2 (indexed from 0), you will get the maximum value, that is, (nums[1]-1)\*(nums[2]-1) = (4-1)\*(5-1) = 3\*4 = 12.

Example 2:

Input: nums = [1,5,4,5]

Output: 16

Explanation: Choosing the indices i=1 and j=3 (indexed from 0), you will get the maximum value of (5-1)\*(5-1) = 16.

Example 3:

Input: nums = [3,7]

Output: 12

Constraints:

2 <= nums.length <= 500

1 <= nums[i] <= 10^3

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Solving: 9

Design a system that manages the reservation state of n seats that are numbered from 1 to n.

Implement the SeatManager class:

SeatManager(int n) Initializes a SeatManager object that will manage n seats numbered from 1 to n. All seats are initially available.

int reserve() Fetches the smallest-numbered unreserved seat, reserves it, and returns its number.

void unreserve(int seatNumber) Unreserves the seat with the given seatNumber.

Example 1:

Input

["SeatManager", "reserve", "reserve", "unreserve", "reserve", "reserve", "reserve", "reserve", "unreserve"]

[[5], [], [], [2], [], [], [], [], [5]]

Output

[null, 1, 2, null, 2, 3, 4, 5, null]

Explanation

SeatManager seatManager = new SeatManager(5); // Initializes a SeatManager with 5 seats.

seatManager.reserve(); // All seats are available, so return the lowest numbered seat, which is 1.

seatManager.reserve(); // The available seats are [2,3,4,5], so return the lowest of them, which is 2.

seatManager.unreserve(2); // Unreserve seat 2, so now the available seats are [2,3,4,5].

seatManager.reserve(); // The available seats are [2,3,4,5], so return the lowest of them, which is 2.

seatManager.reserve(); // The available seats are [3,4,5], so return the lowest of them, which is 3.

seatManager.reserve(); // The available seats are [4,5], so return the lowest of them, which is 4.

seatManager.reserve(); // The only available seat is seat 5, so return 5.

seatManager.unreserve(5); // Unreserve seat 5, so now the available seats are [5].

Constraints:

1 <= n <= 105

1 <= seatNumber <= n

For each call to reserve, it is guaranteed that there will be at least one unreserved seat.

For each call to unreserve, it is guaranteed that seatNumber will be reserved.

At most 105 calls in total will be made to reserve and unreserve.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Statement 10:

Problem: Find the maximum sum subarray of a fixed size k. Description: Given an array of integers and a positive integer k, find the maximum sum of any contiguous subarray of size k.

Example: Input: [2, 1, 5, 1, 3, 2], k = 3.

Output: 9 (corresponding to subarray [5, 1, 3]).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problem Statement 11:

You are given an integer array of nums and two integers, minK and maxK.

A fixed-bound subarray of nums is a subarray that satisfies the following conditions:

The minimum value in the subarray is equal to minK.

The maximum value in the subarray is equal to maxK.

Return the number of fixed-bound subarrays.

A subarray is a contiguous part of an array.

Example 1:

Input: nums = [1,3,5,2,7,5], minK = 1, maxK = 5

Output: 2

Explanation: The fixed-bound subarrays are [1,3,5] and [1,3,5,2].

Example 2:

Input: nums = [1,1,1,1], minK = 1, maxK = 1

Output: 10

Explanation: Every subarray of nums is a fixed-bound subarray. There are 10 possible subarrays.

Constraints:

2 <= nums.length <= 105

1 <= nums[i], minK, maxK <= 106

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_