Q0: Queue

1. Problem Statement

- Queue is a linear data structure that follows FIFO order:
- First In First Out
  - The element added first is removed first
- Think of a line at a counter:
  - first person → served first
- **Core Terminology**
- Enqueue → Insert element at the rear
- Dequeue → Remove element from the front
- Front / Peek → Element at the front without removing it
- Rear → Last inserted element
- isEmpty → Check if queue is empty
- size → Number of elements
- **Intuition Behind the Concept**
- Queue is used when order matters
- Useful when tasks must be processed in the same order they arrive
- Unlike stack (LIFO), queue ensures fair processing
- **Basic Operations (Conceptually)**
- Enqueue → push from one side
- Dequeue → pop from the other side
- Access is restricted:
  - No random access like arrays
- 💡 **Types of Queue (Theory)**
- Simple Queue
- Circular Queue
- Deque (Double Ended Queue)

- Priority Queue
- **Queue Declaration in Java (Most Important)**
- ✅ Using LinkedList (Most Common)
  - Queue q = new LinkedList<>();
  - Why?
  - LinkedList implements Queue
  - Allows dynamic size
  - Fast enqueue & dequeue → O(1)
- ✅ Using ArrayDeque (Recommended for DSA)
  - Queue q = new ArrayDeque<>();
  - Why better?
  - Faster than LinkedList
  - No unnecessary node overhead
  - Preferred in interviews & competitive coding
- ❌ Using PriorityQueue (Different behavior)
  - Queue q = new PriorityQueue<>();
  - ❌ Does NOT follow FIFO
  - Elements come out by priority, not order

```
q.add(10);        // Enqueue (throws exception if fails)
q.offer(20);      // Enqueue (returns false if fails)

q.remove();       // Dequeue (throws exception if empty)
q.poll();         // Dequeue (returns null if empty)

q.peek();         // Front element (null if empty)
q.element();      // Front element (exception if empty)

q.isEmpty();      // Check empty
q.size();         // Queue size
```

- **add vs offer (Interview Favorite)**
- add() → throws exception on failure
- offer() → safe, returns false
- **remove vs poll**
- remove() → exception if empty
- poll() → returns null

- **When to Use Queue in DSA**
- BFS (Breadth First Search)
- Level order traversal (Trees)
- Sliding window problems
- Task scheduling
- Producer–Consumer problems
- **Complexity**
- ⏱️ Time Complexity (Core Operations)
  - Enqueue → O(1)
  - Dequeue → O(1)
  - Peek → O(1)
  - Why?
  - Because insertion & removal happen only at ends
- 💾 Space Complexity
  - O(n) — stores n elements

---

2. Pitfalls

- Using PriorityQueue when FIFO is required
- Using remove() instead of poll() without empty check
- Confusing Queue vs Deque
- Expecting random access like array

Q198: Implement Queue Using Arrays

---

1. Problem Statement

- Implement a First-In-First-Out (FIFO) queue using an array.
- The queue must support the following operations:
  - push(int x) → add element at the end
  - pop() → remove and return the front element
  - peek() → return the front element without removing it
  - isEmpty() → check if queue is empty

---

2. Problem Understanding

- Queue follows FIFO principle

- Insertion happens at the rear
- Deletion happens from the front
- Array is used as the underlying data structure
- We must track:
- front → index of first element
- rear → index of last inserted element

---

3. Constraints

- 1 <= number of operations <= 100
- 1 <= x <= 100
- Fixed-size array is sufficient

---

4. Edge Cases

- pop() when queue is empty
- peek() when queue is empty
- isEmpty() immediately after initialization
- Single element enqueue → dequeue

---

5. Examples

```
Example 1

Input:

["ArrayQueue", "push", "push", "peek", "pop", "isEmpty"]
[[], [5], [10], [], [], []]


Output:

[null, null, null, 5, 5, false]

Example 2

Input:

["ArrayQueue", "isEmpty"]
[[]]


Output:

[null, true]

Example 3
```

```
Input:

["ArrayQueue", "push", "pop", "isEmpty"]
[[], [1], [], []]


Output:

[null, null, 1, true]
```

---

6. Approaches

Approach 1: Queue Using Array (Simple Linear Queue)

Idea:

- Use an integer array
- Maintain two pointers:
  - front → points to current front
  - rear → points to last inserted element
- Queue is empty when front > rear

Steps:

- Initialize:
  - front = 0
  - rear = -1
  - push(x):
  - Increment rear
  - Store x at arr[rear]
- pop():
  - If empty, return -1
  - Return arr[front]
  - Increment front
- peek():
  - If empty, return -1
  - Return arr[front]
- isEmpty():
  - Return front > rear

Java Code:

```
class ArrayQueue {
    int[] arr;
    int front, rear;

    ArrayQueue() {
        arr = new int[100];
```

```java
        front = 0;
        rear = -1;
    }

    void push(int x) {
        arr[++rear] = x;
    }

    int pop() {
        if (isEmpty()) return -1;

        int val = arr[front++];

        // 🔥 reset when queue becomes empty
        if (front > rear) {
            front = 0;
            rear = -1;
        }
        return val;
    }

    int peek() {
        if (isEmpty()) return -1;
        return arr[front];
    }

    int size() {
        return rear - front + 1;
    }

    boolean isEmpty() {
        return size() == 0;
    }
}
```

💭 **Intuition Behind the Approach:**

- Queue processes elements in arrival order
- front always represents the oldest element
- rear tracks the most recent insertion
- Array provides constant-time access using indices

Complexity (Time & Space):

- Time: O(1) — direct index access for all operations
- Space: O(N) — array storage

---

7. Justification / Proof of Optimality

- This approach correctly simulates FIFO behavior using an array with constant-time operations and minimal overhead, making it suitable for small constraints.

---

8. Variants / Follow-Ups

- Queue using Linked List
- Circular Queue (to reuse freed space)
- Queue using Two Stacks
- Deque (Double Ended Queue)

---

9. Tips & Observations

- Linear array queue wastes space after many dequeues
- Circular queue solves space wastage
- FIFO problems often appear in:
  - BFS
  - Sliding Window
  - Scheduling problems

---

10. Pitfalls

- Forgetting empty condition before pop() / peek()
- Confusing queue with stack (FIFO vs LIFO)
- Not resetting pointers correctly
- Ignoring space wastage in linear queue

Q199: Implement Stack Using Queue (Single Queue)

---

1. Problem Statement

- Implement a Last-In-First-Out (LIFO) stack using a single queue.
- The stack must support the following operations:
  - push(int x) → push element onto stack
  - pop() → remove and return top element
  - top() → return top element without removing
  - isEmpty() → check if stack is empty

---

2. Problem Understanding

- Stack follows LIFO
- Queue follows FIFO

- Only one queue is allowed
- Stack behavior must be simulated using queue operations
- The most recently pushed element must always be removed first

---

3. Constraints

- 1 <= number of calls <= 100
- 1 <= x <= 100
- Only one queue can be used

---

4. Edge Cases

- pop() on empty stack
- top() on empty stack
- isEmpty() immediately after initialization
- Single element push → pop

---

5. Examples

```
Example 1

Input:

["QueueStack", "push", "push", "pop", "top", "isEmpty"]
[[], [4], [8], [], [], []]


Output:

[null, null, null, 8, 4, false]

Example 2

Input:

["QueueStack", "isEmpty"]
[[]]


Output:

[null, true]

Example 3

Input:

["QueueStack", "push", "pop", "isEmpty"]
```

```
[[], [6], [], []]


Output:

[null, null, 6, true]
```

---

6. Approaches

Approach 1: Stack Using Single Queue (Push Costly)

Idea:

- Use one queue
- After every push(x), rotate the queue so that:
    - x comes to the front
- This ensures:
    - Front of queue = top of stack

Steps:

- Maintain one queue q
- push(x):
    - Add x to queue
    - Rotate the queue size - 1 times
- pop():
    - If empty, return -1
    - Remove and return front element
- top():
    - If empty, return -1
    - Return front element
- isEmpty():
    - Return true if queue is empty

Java Code:

```
class QueueStack {
    Queue<Integer> q = new ArrayDeque<>();

    void push(int x) {
        q.offer(x);
        int size = q.size();
        while (size-- > 1) {
            q.offer(q.poll());
        }
    }

    int pop() {
```

```
        if (q.isEmpty()) return -1;
        return q.poll();
    }

    int top() {
        if (q.isEmpty()) return -1;
        return q.peek();
    }

    boolean isEmpty() {
        return q.isEmpty();
    }
}
```

💭 **Intuition Behind the Approach:**

- Queue gives FIFO access
- Rotating after push moves the newest element to the front
- Front of queue always behaves like stack top
- Sacrifices push efficiency to simplify pop and top

Complexity (Time & Space):

- Time: O(N) — push rotates all elements
- Time: O(1) — pop, top, isEmpty
- Space: O(N) — queue storage

---

7. Justification / Proof of Optimality

- Rotating the queue after each push ensures correct LIFO behavior while using only one queue, satisfying the problem constraints.

---

8. Variants / Follow-Ups

- Stack using two queues (push O(1))
- Stack using two queues (pop O(1))
- Stack using array
- Stack using linked list

---

9. Tips & Observations

- Single-queue solution always makes push costly
- If push must be O(1), two queues are required
- Rotation count is always queue_size - 1
- Front of queue represents stack top

---

10. Pitfalls

- Forgetting to rotate after push
- Rotating incorrect number of times
- Confusing FIFO behavior during pop
- Not checking empty before pop/top

Q228: Circular Queue

---

1. Problem Statement

- Implement a Circular Queue using an array that supports the following operations:
  - push(x) → Insert element x into the queue
  - pop() → Remove and return the front element of the queue
  - front() → Return the front element without removing it
  - size() → Return the number of elements currently in the queue
- Key Requirements
  - The queue must use a fixed-size array
  - Efficiently utilize space by reusing empty positions
  - All operations should work in O(1) time

---

2. Problem Understanding

- A Circular Queue is an improved version of a normal array-based queue.
- Why not a normal queue?
- In a linear queue:
  - After several pop() operations, front moves forward
  - Empty spaces at the beginning cannot be reused
  - This causes space wastage
- 👉 Circular Queue solves this by:
  - Treating the array as circular
  - Using modulo (%) to wrap indices

---

3. Constraints

- Fixed-size array
- Maximum size is known beforehand
- 0 <= number of elements <= size
- All operations must be constant time

---

4. Edge Cases

- Queue is empty
- Queue is full
- First insertion

- Last deletion (queue becomes empty)
- Wrap-around when rear reaches end

---

5. Examples

```
push(10) → [10, _, _, _, _]
push(20) → [10, 20, _, _, _]
push(30) → [10, 20, 30, _, _]
pop()    → [_, 20, 30, _, _]
push(40) → [40, 20, 30, _, _]   ← wrap-around
```

---

6. Approaches

Approach 1: Circular Queue using Array (Optimal)

Idea:

- Use % size to reuse array positions
- Queue is full when the next position of rear hits front
- Queue is empty when front == -1
- Reset pointers when queue becomes empty
- ◇ Conditions (Very Important)
    - Empty → front == -1
    - Full → (rear + 1) % size == front

Java Code:

```java
class CircularQueue {
    int[] arr;
    int front, rear, size;

    CircularQueue(int size) {
        this.size = size;
        arr = new int[size];
        front = -1;
        rear = -1;
    }

    // enqueue
    void push(int x) {
        if ((rear + 1) % size == front) {
            return; // queue full
        }

        if (front == -1) {
            front = 0;
            rear = 0;
```

```
        } else {
            rear = (rear + 1) % size;
        }

        arr[rear] = x;
    }

    // dequeue
    int pop() {
        if (front == -1) return -1;

        int val = arr[front];

        if (front == rear) {
            front = -1;
            rear = -1;
        } else {
            front = (front + 1) % size;
        }

        return val;
    }

    // peek
    int front() {
        if (front == -1) return -1;
        return arr[front];
    }

    // size
    int size() {
        if (front == -1) return 0;
        if (rear >= front)
            return rear - front + 1;
        return size - (front - rear - 1);
    }
}
```

Complexity (Time & Space):

- Time: O(1) — index updates use constant-time modulo
- Space: O(N) — fixed-size array storage

---

7. Justification / Proof of Optimality

- Prevents space wastage present in linear queue
- Supports all queue operations in constant time
- Standard interview-expected implementation

---

8. Variants / Follow-Ups

- Circular Queue using Linked List
- Deque (Double Ended Queue)
- Dynamic Circular Queue (resizable)

---

9. Tips & Observations

- Circular Queue is conceptually important even if not directly coded
- % size is the heart of the solution
- Always reset pointers after last removal
- Deque problems are extensions of circular queue logic

---

10. Pitfalls

- Forgetting modulo while incrementing
- Wrong full condition
- Not resetting front and rear
- Mixing linear and circular queue logic

Q229: Queue Using Linked List

---

1. Problem Statement

- Implement a Queue using a Linked List that supports the following operations:
  - push(x) → Insert integer x at the rear of the queue
  - pop() → Remove and return the front element of the queue
  - front() → Return the front element without removing it
  - size() → Return the number of elements currently in the queue
- All operations must work efficiently.

---

2. Problem Understanding

- A Queue follows the FIFO (First In First Out) principle.
- Using a Linked List:
- Each node stores data and a reference to the next node
- We maintain:
  - front → points to first element
  - rear → points to last element
  - Insertions happen at rear
  - Deletions happen at front
- 👉 Unlike array-based queues, no fixed size and no shifting needed.

---

3. Constraints

- At most 1000 elements in the queue
- Operations belong to {1, 2, 3, 4}
- All operations should run in O(1)

---

4. Edge Cases

- Pop on empty queue
- Front on empty queue
- Queue becomes empty after pop
- First insertion into empty queue

---

5. Examples

```
Operations:
push(1) → [1]
push(2) → [1, 2]
push(3) → [1, 2, 3]
front() → 1
pop()   → removes 1 → [2, 3]
front() → 2
size()  → 2
```

---

6. Approaches

Approach 1: Queue Using Linked List (Optimal)

Idea:

- Maintain front and rear pointers
- Insert at rear in O(1)
- Remove from front in O(1)
- Update both pointers correctly when queue becomes empty

Java Code:

```java
class Queue {
    class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    Node front, rear;
```

```
    int size;

    Queue() {
        front = null;
        rear = null;
        size = 0;
    }

    void push(int x) {
        Node newNode = new Node(x);

        if (rear == null) {
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
        size++;
    }

    int pop() {
        if (front == null) return -1;

        int val = front.data;
        front = front.next;

        if (front == null) {
            rear = null;
        }

        size--;
        return val;
    }

    int front() {
        if (front == null) return -1;
        return front.data;
    }

    int size() {
        return size;
    }
}
```

Complexity (Time & Space):

- Time: O(1) — insert and delete using pointers only
- Space: O(N) — one node per element

---

7. Justification / Proof of Optimality

- No space wastage as in array queue
- No overflow until memory is exhausted
- Constant time for all queue operations
- Clean and interview-preferred implementation

---

8. Variants / Follow-Ups

- Queue using Array
- Circular Queue
- Deque using Linked List

---

9. Tips & Observations

- Always maintain both front and rear
- When queue becomes empty, set both to null
- Linked List queue is better when size is unknown

---

10. Pitfalls

- Forgetting to update rear when last element is removed
- Returning wrong value on empty pop
- Not maintaining size correctly

Q230: Reverse First K Elements of Queue

1. Problem Statement

- Given an integer K and a queue of N integers, reverse the order of the first K elements of the queue, while keeping the remaining elements in the same relative order.

---

2. Problem Understanding

- You are given:
  - A queue (FIFO structure)
  - An integer K
- Your task:
  - Reverse only the first K elements
  - Do not disturb the order of the remaining N - K elements
- Important points:
  - Only the prefix of length K is affected
  - Queue nature must be preserved after modification

---

3. Constraints

- 1 <= K <= N <= 10000
- 1 <= elements <= 10000
- Efficient solution required

---

4. Edge Cases

- K = 1 → queue remains unchanged
- K = N → entire queue is reversed
- Queue with only one element
- Large N → avoid unnecessary operations

---

5. Examples

```
Input:
N = 5, K = 3
Queue = [1, 2, 3, 4, 5]

Output:
[3, 2, 1, 4, 5]
Explanation:

First 3 elements → [1, 2, 3]

Reverse → [3, 2, 1]

Remaining elements stay same → [4, 5]
```

---

6. Approaches

Approach 1: Brute Force using Extra Array (Conceptual)

Idea:

- Copy queue to array
- Reverse first K elements in array
- Rebuild queue
- ⚠ Not preferred in interviews, but helps understanding.

Steps:

- Dequeue all elements into array
- Reverse array indices [0..K-1]
- Enqueue elements back

Java Code:

```java
static void reverseFirstK_Array(Queue<Integer> q, int k) {
    int n = q.size();
    int[] arr = new int[n];

    for (int i = 0; i < n; i++) {
        arr[i] = q.poll();
    }

    int l = 0, r = k - 1;
    while (l < r) {
        int temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
        l++; r--;
    }

    for (int x : arr) {
        q.add(x);
    }
}
```

💭 **Intuition Behind the Approach:**

- Treat queue like a normal array
- Straightforward but breaks queue abstraction

Complexity (Time & Space):

- Time: O(N) — full traversal and rebuild
- Space: O(N) — extra array used

Approach 2: Using Recursion (Better, but Risky)

Idea:

- Use recursion to reverse first K elements
- Stack frame acts as implicit stack

Steps:

- Pop first element
- Recursively reverse next K-1
- Insert popped element at rear

Java Code:

```java
static void reverseFirstK_Recursion(Queue<Integer> q, int k) {
    if (k == 0) return;

    int x = q.poll();
```

```
        reverseFirstK_Recursion(q, k - 1);
        q.add(x);
    }

    After recursion, rotate remaining N-K elements.

    static void reverseK_WithRotation(Queue<Integer> q, int k) {
        int n = q.size();
        reverseFirstK_Recursion(q, k);

        for (int i = 0; i < n - k; i++) {
            q.add(q.poll());
        }
    }
```

💭 **Intuition Behind the Approach:**

- Recursion reverses order naturally
- Queue rotation maintains remaining elements

Complexity (Time & Space):

- Time: O(N) — recursion + rotation
- Space: O(K) — recursion stack

Approach 3: Stack + Queue (Optimal & Interview-Preferred)

Idea:

- Stack reverses first K elements
- Queue rotation preserves remaining order

Steps:

- Push first K elements into stack
- Pop stack back into queue
- Rotate remaining N-K elements

Java Code:

```java
static void reverseFirstK(Queue<Integer> q, int k) {
    Stack<Integer> st = new Stack<>();

    for (int i = 0; i < k; i++) {
        st.push(q.poll());
    }

    while (!st.isEmpty()) {
        q.add(st.pop());
    }
```

```
    int rem = q.size() - k;
    for (int i = 0; i < rem; i++) {
        q.add(q.poll());
    }
}


// In accio boiler plate solution
import java.util.*;
import java.io.*;

public class Main {
    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        int n = input.nextInt(), k = input.nextInt();
        Queue<Integer> q = new LinkedList<>();

        for (int i = 0; i < n; i++) {
            q.add(input.nextInt());
        }

        Stack<Integer> st = new Stack<>();

        for (int i = 0; i < k; i++) {
            st.push(q.poll());
        }

        while (!st.isEmpty()) {
            q.add(st.pop());
        }

        int rem = q.size() - k;
        for (int i = 0; i < rem; i++) {
            q.add(q.poll());
        }

        while (q.size() > 0) {
            System.out.print(q.poll() + " ");
        }
    }
}
```

💭 Intuition Behind the Approach:

- Stack = reversal
- Queue = order preservation
- Clean separation of responsibilities

Complexity (Time & Space):

- Time: O(N) — each element moved once
- Space: O(K) — stack usage

7. Justification / Proof of Optimality

- Stack-based solution is clean and interview-expected
- Respects queue abstraction
- Efficient and safe for large inputs

8. Variants / Follow-Ups

- Reverse every K elements
- Reverse first K using Deque
- Reverse queue using only recursion

9. Tips & Observations

- Stack is the natural choice for reversal
- Queue problems often mix stack usage
- Always rotate remaining elements

10. Pitfalls

- Forgetting to rotate N-K elements
- Reversing entire queue
- Mishandling K = N

Q231: Rotting Oranges

1. Problem Statement

- You are given an m x n grid where each cell can have one of three values:
  - 0 → empty cell
  - 1 → fresh orange
  - 2 → rotten orange
- Every minute, any fresh orange that is 4-directionally adjacent (up, down, left, right) to a rotten orange becomes rotten.
- Return the minimum number of minutes required so that no fresh orange remains.
- If it is impossible, return -1.

2. Problem Understanding

- Rot spreads simultaneously from all rotten oranges
- This is a level-by-level spread over time
- Each level represents 1 minute

- If even one fresh orange is isolated, answer is -1
- 👉 This is a multi-source BFS problem.

---

3. Constraints

- 1 <= m, n <= 10
- Grid size is small, but logic must be correct
- Only 4-directional movement allowed

---

4. Edge Cases

- No fresh oranges initially → answer 0
- Fresh oranges but no rotten orange → answer -1
- Single cell grid
- Fresh orange completely isolated

---

5. Examples

```
Example 1
2 1 1
1 1 0
0 1 1


Output: 4

Example 2
2 1 1
0 1 1
1 0 1


Output: -1
```

---

6. Approaches

Approach 1: Brute Force Simulation (Not Recommended)

Idea:

- Simulate the rotting process minute by minute by scanning the entire grid repeatedly and rotting fresh oranges adjacent to rotten ones.

Steps:

- Count total fresh oranges

- Repeat:
    - Traverse entire grid
    - For every rotten orange, mark adjacent fresh oranges as “to be rotten”
- Apply all changes at once (to simulate simultaneous spread)
- Increment minute counter
- Stop when:
    - No fresh oranges remain → return time
    - No change occurs but fresh still exist → return -1

Java Code:

```
static int orangesRottingBrute(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    int fresh = 0;

    for (int[] row : grid)
        for (int cell : row)
            if (cell == 1) fresh++;

    int minutes = 0;
    int[][] dir = {{1,0},{-1,0},{0,1},{0,-1}};

    while (fresh > 0) {
        boolean changed = false;
        List<int[]> toRot = new ArrayList<>();

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 2) {
                    for (int[] d : dir) {
                        int ni = i + d[0], nj = j + d[1];
                        if (ni >= 0 && ni < m && nj >= 0 && nj < n && grid[ni][nj]
== 1) {
                            toRot.add(new int[]{ni, nj});
                        }
                    }
                }
            }
        }

        for (int[] cell : toRot) {
            if (grid[cell[0]][cell[1]] == 1) {
                grid[cell[0]][cell[1]] = 2;
                fresh--;
                changed = true;
            }
        }

        if (!changed) return -1;
        minutes++;
    }
```

```
        return minutes;
    }
```

💭 Intuition Behind the Approach:

- We imitate the real-world process directly
- Each loop represents one minute
- Inefficient because we re-scan the whole grid every time

Complexity (Time & Space):

- Time: O((m*n)^2) — full grid scan per minute
- Space: O(m*n) — temporary list for changes
- 👉 Avoid in interviews

Approach 2: Multi-Source BFS (Optimal & Interview-Standard)

Idea:

- Treat every rotten orange as a BFS source
- Spread rot level by level
- Each BFS level = 1 minute
- Count fresh oranges to verify final state

Steps:

- Add all rotten oranges to queue
- Count fresh oranges
- BFS:
    - Process one level (1 minute)
    - Rot adjacent fresh oranges
- If fresh oranges remain → return -1
- Else return time

Java Code:

```java
static int orangesRotting(int[][] grid) {
    int m = grid.length, n = grid[0].length;
    Queue<int[]> q = new ArrayDeque<>();
    int fresh = 0;

    // Step 1: push all rotten oranges
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 2) {
                q.add(new int[]{i, j});
            } else if (grid[i][j] == 1) {
                fresh++;
            }
        }
```

```
    }

    // Edge case
    if (fresh == 0) return 0;

    int minutes = 0;
    int[][] dir = {{1,0},{-1,0},{0,1},{0,-1}};

    // Step 2: BFS
    while (!q.isEmpty()) {
        int size = q.size();
        boolean rotted = false;

        for (int i = 0; i < size; i++) {
            int[] cell = q.poll();
            int r = cell[0], c = cell[1];

            for (int[] d : dir) {
                int nr = r + d[0];
                int nc = c + d[1];

                if (nr >= 0 && nr < m && nc >= 0 && nc < n && grid[nr][nc] == 1) {
                    grid[nr][nc] = 2;
                    fresh--;
                    q.add(new int[]{nr, nc});
                    rotted = true;
                }
            }
        }

        if (rotted) minutes++;
    }

    return fresh == 0 ? minutes : -1;
}
```

💭 **Intuition Behind the Approach:**

- Rot spreads simultaneously → BFS
- Multiple rotten oranges → multi-source BFS
- Queue naturally handles time layers
- Stop when no more spread is possible

Complexity (Time & Space):

- Time: O(m*n) — each cell processed once
- Space: O(m*n) — queue in worst case

---

7. Justification / Proof of Optimality

- BFS models time-based spread perfectly

- Multi-source handles simultaneous rotting
- Guarantees minimum time
- Industry-standard solution

---

8. Variants / Follow-Ups

- Rotting Oranges II (different spread rules)
- Zombie infection problems
- Fire spread simulation
- Shortest path in matrix

---

9. Tips & Observations

- Always count fresh oranges first
- Increment time per BFS level
- Use a flag to check if any rotting happened
- 4-direction only (no diagonals)

---

10. Pitfalls

- Incrementing time when no rotting happens
- Forgetting multi-source initialization
- Missing edge case where no fresh oranges exist
- Treating it as DFS instead of BFS

Q232: Sliding Window Maximum

1. Problem Statement

- You are given an array of integers nums of size N and an integer K representing the window size.
- A sliding window of size K moves from left to right by one position at a time.
- At each position, return the maximum element present in the current window.
- You need to return an array containing the maximum of each window.

---

2. Problem Understanding

- Window size is fixed (K)
- Window moves one step at a time
- For each window, we need the maximum
- Brute force would recompute max for every window
- We need to reuse information from previous windows

---

3. Constraints

- 1 <= N <= 20000
- 1 <= K <= N
- -10^4 <= arr[i] <= 10^4

---

4. Edge Cases

- K = 1 → answer is the array itself
- K = N → answer has one element (max of whole array)
- All elements equal
- Negative numbers

---

5. Examples

```
Input:
nums = [1,3,-1,-3,5,3,6,7], K = 3

Output:
[3, 3, 5, 5, 6, 7]
```

---

6. Approaches

Approach 1: Brute Force (Naive)

Idea:

- For every window of size K, scan all K elements and find the maximum.

Steps:

- For i from 0 to N-K
- For each window [i ... i+K-1], find max
- Store result

Java Code:

```
static int[] slidingWindowMaxBrute(int[] arr, int n, int k) {
    int[] ans = new int[n - k + 1];

    for (int i = 0; i <= n - k; i++) {
        int max = arr[i];
        for (int j = i; j < i + k; j++) {
            max = Math.max(max, arr[j]);
        }
        ans[i] = max;
```

```java
    }
    return ans;
}
```

💭 Intuition Behind the Approach:

- Direct simulation of the problem
- Simple but repetitive work
- Does not reuse previous window information

Complexity (Time & Space):

- Time: O(N*K) — each window scans K elements
- Space: O(1) — excluding output

Approach 2: Using Max Heap (Better)

Idea:

- Use a max heap to always get the maximum element of the current window.

Steps:

- Push (value, index) into heap
- Remove elements that are outside the window
- Heap top gives current max

Java Code:

```java
static int[] slidingWindowMaxHeap(int[] arr, int n, int k) {
    PriorityQueue<int[]> pq =
        new PriorityQueue<>((a, b) -> b[0] - a[0]);

    int[] ans = new int[n - k + 1];
    int idx = 0;

    for (int i = 0; i < n; i++) {
        pq.offer(new int[]{arr[i], i});

        while (pq.peek()[1] <= i - k) {
            pq.poll();
        }

        if (i >= k - 1) {
            ans[idx++] = pq.peek()[0];
        }
    }
    return ans;
}
```

💭 Intuition Behind the Approach:

- Heap keeps max accessible
- Old elements are lazily removed
- Better than brute force but still not optimal

Complexity (Time & Space):

- Time: O(N log N) — heap operations
- Space: O(N) — heap storage

Approach 3: Monotonic Deque (Optimal & Interview-Expected)

Idea:

- Maintain a deque of indices such that:
  - Elements are in decreasing order
  - Front of deque is always the maximum

Steps:

- Remove indices from front that are outside window
- Remove smaller elements from back
- Add current index
- Front of deque is max for current window

Java Code:

```java
static int[] slidingWindowMaximum(int[] arr, int n, int k) {
    Deque<Integer> dq = new ArrayDeque<>();
    int[] ans = new int[n - k + 1];
    int idx = 0;

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

        // remove out-of-window indices
        while (!dq.isEmpty() && dq.peekFirst() <= i - k) {
            dq.pollFirst();
        }

        // maintain decreasing order
        while (!dq.isEmpty() && arr[dq.peekLast()] <= arr[i]) {
            dq.pollLast();
        }

        dq.addLast(i);

        // record answer
        if (i >= k - 1) {
            ans[idx++] = arr[dq.peekFirst()];
        }
    }
```

```
        return ans;
    }
```

💭 **Intuition Behind the Approach:**

- Deque stores useful candidates only
- Smaller elements are discarded early
- Each element enters and leaves deque once
- Front always holds max

Complexity (Time & Space):

- Time: O(N) — each index processed once
- Space: O(K) — deque stores window indices

---

7. Justification / Proof of Optimality

- Brute force is too slow
- Heap improves but still logarithmic
- Monotonic deque achieves linear time
- This is the industry-standard solution

---

8. Variants / Follow-Ups

- Sliding Window Minimum
- First negative number in every window
- Count distinct elements in window
- Maximum of minimums of every window size

---

9. Tips & Observations

- Always store indices, not values
- Deque problems are about order + validity
- Remove out-of-window elements first
- Monotonic structures appear frequently

---

10. Pitfalls

- Storing values instead of indices
- Forgetting to remove out-of-window indices
- Incorrect comparison (<= vs <)
- Using heap when O(N) solution exists

Q233: Circular Tour

---

1. Problem Statement

- There are N petrol pumps arranged in a circular manner.
- You are given:
  - An array petrol[] where petrol[i] is the amount of petrol at pump i
  - An array distance[] where distance[i] is the distance from pump i to the next pump (i+1)
- Assume:
  - 1 unit of petrol allows the truck to travel 1 unit distance
- Find the starting petrol pump index from where the truck can complete the entire circular tour without running out of petrol.
- If no such starting point exists, return -1.

---

2. Problem Understanding

- The truck starts with 0 petrol
- At each pump:
  - It gains petrol[i]
  - It must travel distance[i] to the next pump
- The tour is circular → after last pump, it must return to the first
- If petrol ever becomes negative → journey fails
- This is a feasibility + optimization problem.

---

3. Constraints

- 2 <= N <= 10000
- 1 <= petrol[i], distance[i] <= 1000

---

4. Edge Cases

- Total petrol < total distance → impossible
- Single valid starting point
- Multiple pumps but only one feasible start
- All pumps individually fail but total still sufficient

---

5. Examples

```
Input

4
4 6 7 4
6 5 3 5
Output
```

```
1
Explanation

There are 4 petrol pumps with amount of petrol and distance to next

petrol pump value pairs as {4, 6}, {6, 5}, {7, 3} and {4, 5}. The first point from

where truck can make a circular tour is 2nd petrol pump. Output in this case is 1

(index of 2nd petrol pump).

Example 2
Input

2
1 1 2 3
Output

-1
Explanation

No solution exists.
```

---

6. Approaches

Approach 1: Brute Force (Try Every Starting Point)

Idea:

- Try starting the tour from each petrol pump and simulate the entire circular journey.

Steps:

- For each index i from 0 to N-1
- Start with fuel = 0
- Traverse all pumps circularly
- If fuel becomes negative → break
- If full circle completes → return i

Java Code:

```java
static int circularTourBrute(int[] petrol, int[] dist, int n) {
    for (int start = 0; start < n; start++) {
        int fuel = 0;
        boolean possible = true;

        for (int i = 0; i < n; i++) {
            int idx = (start + i) % n;
            fuel += petrol[idx] - dist[idx];
```

```
            if (fuel < 0) {
                possible = false;
                break;
            }
        }

        if (possible) return start;
    }
    return -1;
}
```

💭 Intuition Behind the Approach:

- Direct simulation
- Simple and easy to reason
- Inefficient because it repeats work for every start

Complexity (Time & Space):

- Time: O(N^2) — full traversal for each start
- Space: O(1)

Approach 2: Greedy / Queue Elimination (Optimal)

Idea:

- Instead of trying all starts:
  - Track current fuel
  - Track total petrol vs total distance
  - If fuel becomes negative at index i, then no pump between previous start and i can be a valid start

Steps:

- Initialize:
- totalFuel = 0
- currFuel = 0
- start = 0
- Traverse pumps from 0 to N-1
- Update:
  - currFuel += petrol[i] - distance[i]
  - totalFuel += petrol[i] - distance[i]
- If currFuel < 0:
  - Reset currFuel = 0
  - Set start = i + 1
- After loop:
  - If totalFuel < 0 → return -1
  - Else return start

Java Code:

```java
static int circularTour(int[] petrol, int[] dist, int n) {
    int totalFuel = 0;
    int currFuel = 0;
    int start = 0;

    for (int i = 0; i < n; i++) {
        int gain = petrol[i] - dist[i];
        totalFuel += gain;
        currFuel += gain;

        if (currFuel < 0) {
            start = i + 1;
            currFuel = 0;
        }
    }

    return totalFuel >= 0 ? start : -1;
}

Alternative Greedy Implementation

int tour(int petrol[], int distance[]) {
    int n = petrol.length;

    int totalPetrol = 0;
    int totalDistance = 0;

    for (int i = 0; i < n; i++) {
        totalPetrol += petrol[i];
        totalDistance += distance[i];
    }

    if (totalPetrol < totalDistance) return -1;

    int currFuel = 0;
    int start = 0;

    for (int i = 0; i < n; i++) {
        currFuel += petrol[i] - distance[i];

        if (currFuel < 0) {
            start = i + 1;
            currFuel = 0;
        }
    }

    return start;
}
```

💭 **Intuition Behind the Approach:**

- If you fail at pump i, any start before i also fails
- So we safely skip all those starts
- Total fuel check ensures feasibility
- One pass is enough
- Alternateive approach
- If total petrol is insufficient, no solution exists
- If the truck fails at index i, any start before i is invalid
- Resetting start skips all impossible pumps
- One full pass is enough to find the answer

Complexity (Time & Space):

- Time: O(N) — single traversal
- Space: O(1)

---

7. Justification / Proof of Optimality

- Brute force is too slow for large N
- Greedy eliminates impossible starts early
- Optimal solution is clean, fast, and interview-standard

---

8. Variants / Follow-Ups

- Gas Station (LeetCode)
- Circular queue feasibility
- Resource allocation problems
- Scheduling with wrap-around constraints

---

9. Tips & Observations

- Always check total petrol vs total distance
- Reset start only when current fuel becomes negative
- Greedy works because failure region is provably invalid

---

10. Pitfalls

- Returning start without checking total fuel
- Off-by-one errors in start update
- Trying BFS/DP unnecessarily
- Overthinking circular traversal

Q249: Class chocolate distribution

---

1. Problem Statement

- There are n classes standing in a queue to receive chocolates.
- Each class has a certain number of students, and each student needs exactly 1 chocolate.
- You are given:
  - A 0-indexed integer array classes where classes[i] represents the number of students in the i-th class.
  - An integer k, representing the index of a specific class.
- Rules:
  - A class can take only 1 chocolate at a time.
  - Taking 1 chocolate takes 1 second.
  - After taking a chocolate:
    - If the class still has students left → it goes to the end of the queue instantly.
    - If the class has no students left → it leaves the queue.
  - The queue order is maintained strictly.
- Task:
  - Return the total time (in seconds) taken for the class at index k to finish receiving chocolates for all its students.

---

2. Problem Understanding

- This is a round-robin distribution problem.
- Each class gets turns cyclically.
- Time increases by exactly 1 second per chocolate given.
- We stop counting the moment class k gets its last chocolate.
- 👉 Core idea: How many times does each class get to take a chocolate before class k finishes?

---

3. Constraints

- 1 <= n <= 100
- 1 <= classes[i] <= 100
- 0 <= k < n

---

4. Edge Cases

- classes[k] == 1 → finishes in the first cycle.
- n == 1 → time = classes[0]
- Classes before k may finish early and leave the queue.
- Classes after k do not contribute in the final second.

---

5. Examples

```
Input:
3
```

```
2 3 2
2

Output:
6


Input:
4
5 1 1 1
0

Output:
8
```

---

6. Approaches

Approach 1: Brute Force Queue Simulation

Idea:

- Simulate the process exactly as described using a queue.

Steps:

- Push (index, remainingStudents) into a queue.
- While queue is not empty:
  - Pop front.
  - Give 1 chocolate → time++.
  - If this class becomes empty:
    - If it is class k, stop.
- Else push it back.

Java Code:

```java
static int timeRequired(int[] classes, int k) {
    Queue<int[]> q = new ArrayDeque<>();
    for (int i = 0; i < classes.length; i++) {
        q.offer(new int[]{i, classes[i]});
    }

    int time = 0;

    while (!q.isEmpty()) {
        int[] curr = q.poll();
        time++;
        curr[1]--;

        if (curr[1] == 0) {
            if (curr[0] == k) return time;
```

```
        } else {
            q.offer(curr);
        }
    }
    return time;
}
```

💭 Intuition Behind the Approach:

- We literally execute the rules step by step.
- Queue preserves order, and every chocolate consumes 1 second.

Complexity (Time & Space):

- Time: O(sum(classes)) — one operation per chocolate
- Space: O(n) — queue storage

Approach 2: Optimized Mathematical Counting (Optimal)

Idea:

- Class k needs classes[k] chocolates → classes[k] rounds.
    - Classes before or at k can get chocolates in all those rounds
    - Classes after k can get chocolates in only classes[k] - 1 rounds

Steps:

- Let target = classes[k]
- For every class i:
    - If i <= k → add min(classes[i], target)
    - If i > k → add min(classes[i], target - 1)
- Sum is the answer.

Java Code:

```java
static int timeRequired(int[] classes, int k) {
    int time = 0;
    int target = classes[k];

    for (int i = 0; i < classes.length; i++) {
        if (i <= k) {
            time += Math.min(classes[i], target);
        } else {
            time += Math.min(classes[i], target - 1);
        }
    }
    return time;
}
```

💭 Intuition Behind the Approach:

- The last chocolate of class k ends everything.
- Classes after k cannot delay that last second.
- So we count only the effective blocking time.

Complexity (Time & Space):

- Time: O(n) — single pass
- Space: O(1) — no extra data structures

---

7. Justification / Proof of Optimality

- Queue simulation matches the problem definition exactly.
- Optimized approach counts how many seconds each class blocks class k.
- Both produce identical results.

---

8. Variants / Follow-Ups

- Ticket Buying (LeetCode 2073)
- Round-Robin CPU Scheduling
- Printer Queue problems

---

9. Tips & Observations

- "Go to end of the line" → Queue
- "One unit per second" → Time simulation
- Always check if last round can be excluded for optimization

---

10. Pitfalls

- Counting classes after k in the last round
- Forgetting that re-queueing is instantaneous
- Over-simulating when constraints are small

Q250: Design Ring Buffer

1. Problem Statement

- Design a Circular Queue (Ring Buffer) that supports the following operations without using any built-in queue data structure.
- You are required to implement the AccioQueue class with the following methods:
  - AccioQueue(k) → Initialize the queue with size k
  - int Front() → Return the front element, or -1 if empty
  - int Rear() → Return the rear element, or -1 if empty

- boolean enQueue(int value) → Insert an element, return true if successful
  - boolean deQueue() → Remove an element, return true if successful
  - boolean isEmpty() → Check if queue is empty
  - boolean isFull() → Check if queue is full
- Input Queries
  - 0 X → Create queue of size X
  - 1 → Front
  - 2 → Rear
  - 3 X → enQueue(X)
  - 4 → deQueue
  - 5 → isEmpty
  - 6 → isFull

---

2. Problem Understanding

- A circular queue is a FIFO structure where the last position is connected back to the first.
- Unlike a normal queue, empty space at the front can be reused.
- Indices wrap around using modulo arithmetic.
- We track:
  - front → first valid element
  - rear → last valid element

---

3. Constraints

- 1 <= k <= 1000
- 0 <= value <= 1000
- 0 <= number of queries <= 300

---

4. Edge Cases

- Dequeue on empty queue
- Enqueue on full queue
- Single element queue (front == rear)
- Wrap-around condition ((rear + 1) % size)
- Calling operations before initialization

---

5. Examples

```
Input:
10
0 3
3 1
3 2
3 3
3 4
```

```
2
6
4
3 4
2

Output:
null true true true false 3 true true true 4
```

---

6. Approaches

Approach 1: Brute Force Using Array + Shifting (NOT OPTIMAL)

Idea:

- Use an array
- On dequeue, shift elements left
- 🚫 Why Not Used
  - Shifting costs O(n) per operation
  - Fails time efficiency requirement

💭 **Intuition Behind the Approach:**

- This mimics a normal queue but wastes time by shifting elements.

Complexity (Time & Space):

- Time: O(n) — shifting on every deQueue
- Space: O(n)

Approach 2: Circular Queue Using Modulo (OPTIMAL)

Idea:

- Use a fixed size array
- Maintain front and rear
- Use modulo % size to wrap around

Steps:

- Initialize:
  - front = -1, rear = -1
- enQueue
  - If full → return false
  - First insert → set front = rear = 0
  - Else → rear = (rear + 1) % size
- deQueue
  - If empty → return false
  - If only one element → reset both to -1
  - Else → front = (front + 1) % size

- isEmpty
    - front == -1
- isFull
    - (rear + 1) % size == front

Java Code:

```
import java.util.*;

public class Main {

    static AccioQueue curr = null; // MUST be static

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();

        for (int i = 0; i < n; i++) {
            int call = sc.nextInt();

            if (call == 0 || call == 3) {
                int value = sc.nextInt();
                FunctionToBeCalled(call, value);
            } else {
                FunctionToBeCalled(call);
            }
        }
    }

    static void FunctionToBeCalled(int call) {
        FunctionToBeCalled(call, 0);
    }

    static void FunctionToBeCalled(int call, int value) {
        switch (call) {
            case 0:
                curr = new AccioQueue(value);
                System.out.print("null ");
                break;
            case 1:
                System.out.print(curr.Front() + " ");
                break;
            case 2:
                System.out.print(curr.Rear() + " ");
                break;
            case 3:
                System.out.print(curr.enQueue(value) + " ");
                break;
            case 4:
                System.out.print(curr.deQueue() + " ");
                break;
            case 5:
```

```java
                System.out.print(curr.isEmpty() + " ");
                break;
            case 6:
                System.out.print(curr.isFull() + " ");
                break;
        }
    }

    static class AccioQueue {
        int[] arr;
        int front, rear, size;

        AccioQueue(int size) {
            this.size = size;
            arr = new int[size];
            front = -1;
            rear = -1;
        }

        int Front() {
            if (isEmpty()) return -1;
            return arr[front];
        }

        int Rear() {
            if (isEmpty()) return -1;
            return arr[rear];
        }

        boolean enQueue(int value) {
            if (isFull()) return false;
            if (front == -1) {
                front = 0;
                rear = 0;
            } else {
                rear = (rear + 1) % size;
            }
            arr[rear] = value;
            return true;
        }

        boolean deQueue() {
            if (isEmpty()) return false;
            if (front == rear) {
                front = -1;
                rear = -1;
                return true;
            }
            front = (front + 1) % size;
            return true;
        }

        boolean isEmpty() {
            return front == -1;
```

```
        }

        boolean isFull() {
            return (rear + 1) % size == front;
        }
    }
}
```

💭 **Intuition Behind the Approach:**

- We reuse space instead of shifting elements.
- Modulo ensures indices stay within bounds.
- Resetting when front == rear handles the last element case cleanly.

Complexity (Time & Space):

- Time: O(1) — each operation is constant time
- Space: O(n) — fixed array size

---

7. Justification / Proof of Optimality

- Circular queue eliminates wasted space.
- Modulo arithmetic avoids shifting.
- All operations meet optimal time complexity.

---

8. Variants / Follow-Ups

- Deque (Double-Ended Queue)
- Round-Robin CPU Scheduling
- Producer–Consumer buffer
- Ring buffer in OS / networking

---

9. Tips & Observations

- Circular queue always keeps one empty slot logically
- Full condition is next rear hits front
- Reset pointers when queue becomes empty
- Very common interview + system design topic

---

10. Pitfalls

- Forgetting modulo % size
- Not resetting front and rear
- Using = instead of == in isFull
- Mishandling single element case
