Round 1 - Two Sum (HashMap)

Spot the bug in the code below.

```
// returns indices of two numbers that add to target
int[] twoSum(int[] nums, int target) {
  Map<Integer, Integer> map = new HashMap<>();
  for (int i = 0; i < nums.length; i++) {
    map.put(nums[i], i);
    int need = target - nums[i];
    if (map.containsKey(need)) {
      return new int[]{ map.get(need), i };
    }
  }
  return new int[]{-1, -1};
}
```

Round 1 - Two Sum (HashMap)

Spot the bug in the code below.

```
// returns indices of two numbers that add to target
int[] twoSum(int[] nums, int target) {
  Map<Integer, Integer> map = new HashMap<>();
  for (int i = 0; i < nums.length; i++) {
    map.put(nums[i], i);
    int need = target - nums[i];
    if (map.containsKey(need)) {
      return new int[]{ map.get(need), i };
    }
  }
  return new int[]{-1, -1};
}
```

Bug

Inserts nums[i] into the map before checking for its complement. If need == nums[i], it can match the same index you just inserted (same element twice), and it also overwrites earlier indices for duplicate values.

Fix

Check for the complement first, then insert the current value.

```
for (int i = 0; i < nums.length; i++) {
  int need = target - nums[i];
  if (map.containsKey(need)) {
    return new int[]{ map.get(need), i };
  }
  map.put(nums[i], i);
}
```

Round 2 - Sliding Window (Longest substring without repeating)

Spot the bug in the code below.

```
int lengthOfLongestSubstring(String s) {
  Set<Character> set = new HashSet<>();
  int l = 0, best = 0;
  for (int r = 0; r < s.length(); r++) {
    while (set.contains(s.charAt(r))) {
      set.remove(s.charAt(r));
      l++;
    }
    set.add(s.charAt(r));
    best = Math.max(best, r - l + 1);
  }
  return best;
}
```

Round 2 - Sliding Window (Longest substring without repeating)

Spot the bug in the code below.

```
int lengthOfLongestSubstring(String s) {
  Set<Character> set = new HashSet<>();
  int l = 0, best = 0;
  for (int r = 0; r < s.length(); r++) {
    while (set.contains(s.charAt(r))) {
      set.remove(s.charAt(r));
      l++;
    }
    set.add(s.charAt(r));
    best = Math.max(best, r - l + 1);
  }
  return best;
}
```

Bug

When shrinking the window, it removes s.charAt(r) instead of removing the leftmost character s.charAt(l). That does not correctly slide the window and can keep duplicates in the set.

Fix

Remove s.charAt(l) while moving l forward.

```
while (set.contains(s.charAt(r))) {
  set.remove(s.charAt(l));
  l++;
}
```

Round 3 - Reverse Linked List (iterative)

Spot the bug in the code below.

```
ListNode reverse(ListNode head) {
  ListNode prev = null;
  ListNode curr = head;
  while (curr != null) {
    curr.next = prev;
    prev = curr;
    curr = curr.next;
  }
  return prev;
}
```

Round 3 - Reverse Linked List (iterative)

Spot the bug in the code below.

```
ListNode reverse(ListNode head) {
  ListNode prev = null;
  ListNode curr = head;
  while (curr != null) {
    curr.next = prev;
    prev = curr;
    curr = curr.next;
  }
  return prev;
}
```

Bug

It overwrites curr.next before saving the original next node. Then curr = curr.next moves backward (to prev), causing an infinite loop and losing the rest of the list.

Fix

Store the original next pointer first.

```
while (curr != null) {
  ListNode next = curr.next;
  curr.next = prev;
  prev = curr;
  curr = next;
}
```

Round 4 - Cycle Detection (Floyd)

Spot the bug in the code below.

```
boolean hasCycle(ListNode head) {
  ListNode slow = head, fast = head;
  while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next;
    if (slow == fast) return true;
  }
  return false;
}
```

Round 4 - Cycle Detection (Floyd)

Spot the bug in the code below.

```
boolean hasCycle(ListNode head) {
  ListNode slow = head, fast = head;
  while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next;
    if (slow == fast) return true;
  }
  return false;
}
```

Bug

fast only moves one step. That makes slow and fast move at the same speed, so they meet immediately in most non-empty lists (false positives).

Fix

Move fast two steps per iteration.

```
fast = fast.next.next;
```

Round 5 - Validate BST (range logic)

Spot the bug in the code below.

```
boolean isValidBST(TreeNode root) {
  return valid(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

boolean valid(TreeNode node, int low, int high) {
  if (node == null) return true;
  if (node.val < low || node.val > high) return false;
  return valid(node.left, low, node.val) &&
         valid(node.right, node.val, high);
}
```

Round 5 - Validate BST (range logic)

Spot the bug in the code below.

```
boolean isValidBST(TreeNode root) {
  return valid(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

boolean valid(TreeNode node, int low, int high) {
  if (node == null) return true;
  if (node.val < low || node.val > high) return false;
  return valid(node.left, low, node.val) &&
         valid(node.right, node.val, high);
}
```

Bug

Uses int bounds (can fail at extremes) and non-strict comparisons (duplicates can slip through). Also, passing node.val as an inclusive bound allows equal values on the wrong side.

Fix

Use long bounds and strict inequality: low < val < high.

```
boolean isValidBST(TreeNode root) {
  return valid(root, Long.MIN_VALUE, Long.MAX_VALUE);
}

boolean valid(TreeNode node, long low, long high) {
  if (node == null) return true;
  if (node.val <= low || node.val >= high) return false;
  return valid(node.left, low, node.val) &&
         valid(node.right, node.val, high);
}
```

Round 6 - BFS (visited placement)

Spot the bug in the code below.

```
void bfs(int start, List<List<Integer>> adj) {
  Queue<Integer> q = new ArrayDeque<>();
  boolean[] vis = new boolean[adj.size()];
  q.add(start);

  while (!q.isEmpty()) {
    int u = q.remove();
    vis[u] = true;
    for (int v : adj.get(u)) {
      if (!vis[v]) q.add(v);
    }
  }
}
```

Round 6 - BFS (visited placement)

Spot the bug in the code below.

```
void bfs(int start, List<List<Integer>> adj) {
  Queue<Integer> q = new ArrayDeque<>();
  boolean[] vis = new boolean[adj.size()];
  q.add(start);

  while (!q.isEmpty()) {
    int u = q.remove();
    vis[u] = true;
    for (int v : adj.get(u)) {
      if (!vis[v]) q.add(v);
    }
  }
}
```

Bug

Marks visited when dequeuing, so the same node can be enqueued multiple times (especially in cyclic graphs).

Fix

Mark visited when enqueuing (when discovered), including the start node.

```
vis[start] = true;
q.add(start);

while (!q.isEmpty()) {
  int u = q.remove();
  for (int v : adj.get(u)) {
    if (!vis[v]) {
      vis[v] = true;
      q.add(v);
    }
  }
}
```

Round 7 - Connected Components (loop boundary)

Spot the bug in the code below.

```
int countComponents(List<List<Integer>> adj) {
  boolean[] vis = new boolean[adj.size()];
  int count = 0;
  for (int i = 0; i <= adj.size(); i++) {
    if (!vis[i]) {
      count++;
      dfs(i, adj, vis);
    }
  }
  return count;
}
```

Round 7 - Connected Components (loop boundary)

Spot the bug in the code below.

```
int countComponents(List<List<Integer>> adj) {
  boolean[] vis = new boolean[adj.size()];
  int count = 0;
  for (int i = 0; i <= adj.size(); i++) {
    if (!vis[i]) {
      count++;
      dfs(i, adj, vis);
    }
  }
  return count;
}
```

Bug

Loop condition uses i <= adj.size(), so i reaches adj.size() and causes ArrayIndexOutOfBounds on vis[i].

Fix

Use i < adj.size().

```
for (int i = 0; i < adj.size(); i++) { ... }
```

Round 8 - Binary Search (loop update)

Spot the bug in the code below.

```
int binarySearch(int[] a, int target) {
  int l = 0, r = a.length - 1;
  while (l < r) {
    int m = (l + r) / 2;
    if (a[m] == target) return m;
    if (a[m] < target) l = m;
    else r = m - 1;
  }
  return -1;
}
```

Round 8 - Binary Search (loop update)

Spot the bug in the code below.

```
int binarySearch(int[] a, int target) {
  int l = 0, r = a.length - 1;
  while (l < r) {
    int m = (l + r) / 2;
    if (a[m] == target) return m;
    if (a[m] < target) l = m;
    else r = m - 1;
  }
  return -1;
}
```

Bug

When a[m] < target, it sets l = m (not m + 1), which can get stuck when m == l. Also the loop ends at l == r without checking that final position.

Fix

Use a standard while (l <= r) template and move l to m + 1 when going right.

```
int l = 0, r = a.length - 1;
while (l <= r) {
  int m = l + (r - l) / 2;
  if (a[m] == target) return m;
  if (a[m] < target) l = m + 1;
  else r = m - 1;
}
return -1;
```