# Edge Cases Checklist - CRITICAL for Citadel

## WHY EDGE CASES MATTER AT CITADEL

Citadel is **strict** about edge cases. According to recent reports:

- Hidden test cases designed to expose incomplete logic
- Partial credit is uncommon
- A solution that fails corner cases is treated as **incorrect**
- First submission should be correct - no time to debug

**BEFORE YOU SUBMIT:** Go through this checklist EVERY TIME.

---

## UNIVERSAL EDGE CASES (Check These Always)

### 1. Empty Input

```
# Arrays/Lists
if not arr:
    return []  # or 0, or -1, depending on problem

if len(arr) == 0:
    return default_value

# Strings
if not s:
    return ""

# Trees
if not root:
    return None

# Graphs
if not graph or n == 0:
    return 0
```

## 2. Single Element

```
# Arrays
if len(arr) == 1:
    return arr[0]

# Trees
if not root.left and not root.right:
    return root.val  # Leaf node

# Strings
if len(s) == 1:
    return s
```

## 3. Two Elements (Minimum for Pairs/Comparisons)

```
if len(arr) == 2:
    return max(arr[0], arr[1])
```

## 4. All Elements Same

```
# This breaks many algorithms
arr = [5, 5, 5, 5, 5]

# Check if all same
if len(set(arr)) == 1:
    handle_all_same()

# Or
if all(x == arr[0] for x in arr):
    handle_all_same()
```

## 5. Maximum Constraints

```
# Arrays
n = 10^5  # Maximum size
val = 10^9  # Maximum value

# Check if your solution handles these
# Will it overflow? (Not usually in Python, but be aware)
# Will it timeout with O(n²)?
```

---

# ARRAY-SPECIFIC EDGE CASES

## Duplicates

```
# Array with duplicates
arr = [1, 2, 2, 3, 3, 3]

# All duplicates
arr = [5, 5, 5, 5]

# No duplicates
arr = [1, 2, 3, 4, 5]

# Check if problem assumes unique elements
if len(arr) != len(set(arr)):
    # Has duplicates
```

## Sorted vs Unsorted

```
# Is array sorted?
sorted_arr = [1, 2, 3, 4, 5]
unsorted_arr = [3, 1, 4, 1, 5]

# Reverse sorted
reverse_sorted = [5, 4, 3, 2, 1]

# Partially sorted
partially = [1, 2, 3, 5, 4]
```

## Negative Numbers

```
# All negative
arr = [-5, -3, -1]

# Mix of positive and negative
arr = [-2, -1, 0, 1, 2]

# Zeros
arr = [0, 0, 0]

# Does your algorithm handle negatives correctly?
```

### Index Boundaries

```
# First element
arr[0]

# Last element
arr[-1] or arr[len(arr)-1]

# Out of bounds check
if 0 <= i < len(arr):
    safe_access = arr[i]

# Window at boundaries
# Start: i=0, j=k-1
# End: i=n-k, j=n-1
```

---

## STRING-SPECIFIC EDGE CASES

### Empty String

```
s = ""
if not s:
    return default
```

### Single Character

```
s = "a"
if len(s) == 1:
    return s
```

### All Same Character

```
s = "aaaaa"
if len(set(s)) == 1:
    handle_all_same()
```

### Special Characters

```
# Spaces
```

```
s = "hello world"
s = "   "  # Only spaces

# Punctuation
s = "hello, world!"

# Numbers
s = "123"

# Mixed
s = "Hello123!@#"
```

### Case Sensitivity

```
# Different cases
s1 = "Hello"
s2 = "hello"

# Does problem care about case?
# If not, convert first
s = s.lower()
```

### Palindromes

```
# Odd length palindrome
s = "racecar"  # Center at 'e'

# Even length palindrome
s = "abba"  # No single center

# Single character (always palindrome)
s = "a"

# Not a palindrome
s = "hello"
```

---

## TREE-SPECIFIC EDGE CASES

### Empty Tree

```
root = None
```

```
if not root:
    return 0  # or None, or []
```

## Single Node

```
#     1
if not root.left and not root.right:
    return root.val
```

## Linear Tree (Linked List)

```
# Right-skewed
#     1
#      \
#       2
#        \
#         3

# Left-skewed
#       3
#      /
#     2
#    /
#   1
```

## Complete Binary Tree

```
#       1
#      / \
#     2   3
#    / \
#   4   5
```

## Perfect Binary Tree

```
#       1
#      / \
#     2   3
#    / \ / \
#   4  5 6  7
```

### Unbalanced Tree

```
#     1
#    /
#   2
#  /
# 3
#  \
#   4
```

---

## GRAPH-SPECIFIC EDGE CASES

### No Edges

```
# n nodes, 0 edges
graph = {0: [], 1: [], 2: []}
edges = []
```

### Disconnected Graph

```
# Multiple components
#  0---1    2---3
#           |
#           4
```

### Single Node

```
graph = {0: []}
n = 1
```

### Self-Loop

```
# Node points to itself
edges = [(0, 0)]
graph = {0: [0]}
```

### Cycles

```
# Simple cycle
#  0---1
#  |   |
#  3---2

# No cycles (tree/DAG)
```

### Complete Graph

```
# Every node connected to every other
# n nodes → n(n-1)/2 edges
```

---

## LINKED LIST EDGE CASES

### Empty List

```
head = None
if not head:
    return None
```

### Single Node

```
# 1 → None
if not head.next:
    return head
```

### Two Nodes

```
# 1 → 2 → None
```

### Cycle Detection

```
# No cycle
# 1 → 2 → 3 → None

# Cycle at end
# 1 → 2 → 3 → 2 (cycles back)
```

```
# Cycle at start
# 1 → 2 → 1 (cycles back)
```

---

# NUMBER-SPECIFIC EDGE CASES

## Zero

```
n = 0
# Division by zero
if n != 0:
    result = x / n

# Multiplication by zero
result = x * 0  # Always 0

# Powers of zero
0 ** 0  # Undefined, but Python returns 1
```

## Negative Numbers

```
n = -5
# Absolute value
abs(n)

# Division with negatives
-7 // 2  # -4 in Python (floor division)
-7 % 2   # 1 in Python

# Comparison
-1 < 0  # True
```

## Large Numbers

```
# Maximum int (problem constraints)
n = 10**9
n = 2**31 - 1  # Max 32-bit int

# Python handles arbitrary precision
# But check if problem expects overflow behavior
```

### Floating Point

```
# Precision issues
0.1 + 0.2 == 0.3  # False!
abs(a - b) < 1e-9  # Use epsilon for comparison

# Division
7 / 2   # 3.5 (float division)
7 // 2  # 3 (integer division)
```

---

## MATRIX/2D ARRAY EDGE CASES

### Empty Matrix

```
matrix = []
matrix = [[]]

if not matrix or not matrix[0]:
    return []
```

### Single Element

```
matrix = [[5]]
```

### Single Row

```
matrix = [[1, 2, 3, 4]]
```

### Single Column

```
matrix = [[1], [2], [3], [4]]
```

### Square vs Rectangle

```
# Square
matrix = [[1,2], [3,4]]  # 2x2

# Rectangle
```

```
matrix = [[1,2,3], [4,5,6]]  # 2x3
matrix = [[1,2], [3,4], [5,6]]  # 3x2
```

### Boundary Elements

```
# Corners
matrix[0][0]  # Top-left
matrix[0][n-1]  # Top-right
matrix[m-1][0]  # Bottom-left
matrix[m-1][n-1]  # Bottom-right

# Edges
# Top row: matrix[0][j]
# Bottom row: matrix[m-1][j]
# Left column: matrix[i][0]
# Right column: matrix[i][n-1]
```

---

## INTERVAL-SPECIFIC EDGE CASES

### Empty Intervals

```
intervals = []
if not intervals:
    return []
```

### Single Interval

```
intervals = [[1, 3]]
```

### Non-Overlapping

```
intervals = [[1,2], [3,4], [5,6]]
```

### Fully Overlapping

```
intervals = [[1,5], [2,3]]  # [2,3] inside [1,5]
```

### Touching Intervals

```
intervals = [[1,2], [2,3]]  # Share endpoint
# Does problem consider these overlapping?
```

### Same Start/End

```
intervals = [[1,3], [1,3]]  # Identical
intervals = [[1,3], [1,4]]  # Same start
intervals = [[1,3], [2,3]]  # Same end
```

---

## BINARY SEARCH EDGE CASES

### Target Not in Array

```
arr = [1, 3, 5, 7]
target = 4  # Not present
# Should return -1 or insertion position?
```

### Target at Boundaries

```
arr = [1, 3, 5, 7]
target = 1  # First element
target = 7  # Last element
```

### Duplicates

```
arr = [1, 2, 2, 2, 3]
target = 2
# Return first occurrence? Last? Any?
```

### All Elements Same

```
arr = [5, 5, 5, 5, 5]
target = 5
```

---

# SUBARRAY/SUBSTRING EDGE CASES

## Empty Subarray

```
# Is empty subarray valid?
# Some problems include it, some don't
```

## Single Element Subarray

```
arr = [5]
# Subarray is just [5]
```

## Entire Array

```
arr = [1, 2, 3, 4]
subarray = [1, 2, 3, 4]  # Entire array is valid subarray
```

## Prefix/Suffix

```
arr = [1, 2, 3, 4]
prefix = [1, 2, 3]
suffix = [2, 3, 4]
```

---

# CHECKLIST TEMPLATE FOR EACH PROBLEM

Before submitting, verify:

```
☐ Empty input (n=0, arr=[], s="", root=None)
☐ Single element (n=1)
☐ Two elements (n=2)
☐ All elements same
☐ Maximum constraints (n=10^5, val=10^9)
☐ Negative numbers (if applicable)
☐ Zero (if applicable)
☐ Duplicates
☐ Sorted vs unsorted
☐ Boundaries (first/last element, 0/n-1 indices)
☐ Special characters (for strings)
☐ Cycles (for graphs/linked lists)
```

```
☐ Disconnected components (for graphs)
☐ Overflow/underflow (rare in Python, but check)
☐ Division by zero
☐ Off-by-one errors in loops
☐ Correct inequality operators (< vs <=, > vs >=)
```

---

# COMMON OFF-BY-ONE ERRORS

## Loop Bounds

```
# WRONG
for i in range(n):
    if i + 1 < n:  # Redundant, range already handles this
        arr[i+1]

# RIGHT
for i in range(n - 1):
    arr[i+1]

# WRONG
for i in range(1, n):
    arr[i-1]  # Processes arr[0] to arr[n-2], misses arr[n-1]

# RIGHT
for i in range(n):
    if i > 0:
        arr[i-1]
```

## Slicing

```
# arr[start:end] includes start, excludes end
arr = [0, 1, 2, 3, 4]
arr[1:3]  # [1, 2], NOT [1, 2, 3]

# To include end
arr[1:4]  # [1, 2, 3]
```

## Range Queries

```
# Sum from index i to j (inclusive)
# WRONG
```

```
sum(arr[i:j])  # Excludes j

# RIGHT
sum(arr[i:j+1])  # Includes j
```

---

## TESTING STRATEGY

1. **Test empty input first**
2. **Test single element**
3. **Test two elements**
4. **Test all same elements**
5. **Test with negative numbers**
6. **Test maximum constraints**
7. **Test boundary conditions**
8. **Test example from problem statement**

## SAMPLE TEST CASES TO ALWAYS TRY

For array problems:

```
[]
[1]
[1, 1]
[1, 2]
[1, 1, 1, 1]
[-1, -2, -3]
[0, 0, 0]
[1, 2, 3, 4, 5]  # Sorted
[5, 4, 3, 2, 1]  # Reverse sorted
[3, 1, 4, 1, 5]  # Unsorted
```

For string problems:

```
""
"a"
"aa"
"ab"
```

```
"aaaa"
"abc"
"racecar"  # Palindrome
"Hello World"  # Spaces
"123"  # Numbers
```

For tree problems:

```
None  # Empty
Single node
Linear (left or right skewed)
Complete binary tree
Unbalanced tree
```

---

## FINAL REMINDER

**Citadel's hidden test cases are STRICT.**

A solution that handles 90% of cases but fails on edge cases = **WRONG**.

Spend 2-3 minutes going through this checklist before submitting. Those 2-3 minutes could be the difference between passing and failing the assessment.

**Think like a tester, not just a coder.**