Edge Cases Checklist - C++ - 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

```
// Vectors
if (arr.empty()) {
    return {}; // or 0, or -1, depending on problem
}
if (arr.size() == 0) {
    return default_value;
}

// Strings
if (s.empty()) {
    return "";
}

// Trees
if (!root) {
    return nullptr;
}

// Graphs
if (graph.empty() || n == 0) {
    return 0;
}
```

2. Single Element

```
// Vectors
if (arr.size() == 1) {
    return arr[0];
}

// Trees
if (!root->left && !root->right) {
    return root->val; // Leaf node
}

// Strings
if (s.length() == 1) {
    return s;
}
```

3. Two Elements (Minimum for Pairs/Comparisons)

```
if (arr.size() == 2) {
    return max(arr[0], arr[1]);
}
```

4. All Elements Same

```
// This breaks many algorithms
vector<int> arr = {5, 5, 5, 5, 5};

// Check if all same
set<int> unique(arr.begin(), arr.end());
if (unique.size() == 1) {
    handle_all_same();
}

// Or
if (all_of(arr.begin(), arr.end(), [&](int x) { return x == arr[0]; })) {
    handle_all_same();
}
```

5. Maximum Constraints

```
// Arrays
int n = 1e5;  // Maximum size
int val = 1e9; // Maximum value

// Check if your solution handles these
// Will it overflow? Use long long if needed
// Will it timeout with O(n²)?
```

ARRAY-SPECIFIC EDGE CASES

Duplicates

```
// Array with duplicates
vector<int> arr = {1, 2, 2, 3, 3, 3};

// All duplicates
vector<int> arr = {5, 5, 5, 5};

// No duplicates
vector<int> arr = {1, 2, 3, 4, 5};

// Check if problem assumes unique elements
set<int> unique(arr.begin(), arr.end());
if (arr.size() != unique.size()) {
    // Has duplicates
}
```

Sorted vs Unsorted

```
// Is array sorted?
vector<int> sorted_arr = {1, 2, 3, 4, 5};
vector<int> unsorted_arr = {3, 1, 4, 1, 5};

// Reverse sorted
vector<int> reverse_sorted = {5, 4, 3, 2, 1};

// Partially sorted
vector<int> partially = {1, 2, 3, 5, 4};
```

Negative Numbers

```
// All negative
```

```
vector<int> arr = {-5, -3, -1};

// Mix of positive and negative
vector<int> arr = {-2, -1, 0, 1, 2};

// Zeros
vector<int> arr = {0, 0, 0};

// Does your algorithm handle negatives correctly?
```

Index Boundaries

```
// First element
arr[0];

// Last element
arr[arr.size() - 1];
// Or using back()
arr.back();

// Out of bounds check
if (0 <= i && i < arr.size()) {
    int 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

```
string s = "";
if (s.empty()) {
    return default_value;
}
```

Single Character

```
string s = "a";
if (s.length() == 1) {
```

```
    return s;
}
```

All Same Character

```
string s = "aaaaa";
set<char> unique(s.begin(), s.end());
if (unique.size() == 1) {
    handle_all_same();
}
```

Special Characters

```
// Spaces
string s = "hello world";
string s = " "; // Only spaces

// Punctuation
string s = "hello, world!";

// Numbers
string s = "123";

// Mixed
string s = "Hello123!@#";
```

Case Sensitivity

```
// Different cases
string s1 = "Hello";
string s2 = "hello";

// Does problem care about case?
// If not, convert first
transform(s.begin(), s.end(), s.begin(), ::tolower);
```

Palindromes

```
// Odd length palindrome
string s = "racecar"; // Center at 'e'

// Even length palindrome
```

```
string s = "abba"; // No single center

// Single character (always palindrome)
string s = "a";

// Not a palindrome
string s = "hello";
```

TREE-SPECIFIC EDGE CASES

Empty Tree

```
TreeNode* root = nullptr;
if (!root) {
    return 0; // or nullptr, or {}
}
```

Single Node

```
//     1
if (!root->left && !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
vector<vector<int>> graph(n);
vector<pair<int,int>> edges;
```

Disconnected Graph

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

Single Node

```
vector<vector<int>> graph(1);
int n = 1;
```

Self-Loop

```
// Node points to itself
vector<pair<int,int>> edges = {{0, 0}};
graph[0].push_back(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

```
ListNode* head = nullptr;
if (!head) {
    return nullptr;
}
```

Single Node

```
// 1 → nullptr
if (!head->next) {
    return head;
```

```
}
```

Two Nodes

```
// 1 → 2 → nullptr
```

Cycle Detection

```
// No cycle
// 1 → 2 → 3 → nullptr

// Cycle at end
// 1 → 2 → 3 → 2 (cycles back)

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

NUMBER-SPECIFIC EDGE CASES

Zero

```
int n = 0;

// Division by zero
if (n != 0) {
    int result = x / n;
}

// Multiplication by zero
int result = x * 0; // Always 0

// Powers of zero
pow(0, 0); // Undefined, but C++ returns 1
```

Negative Numbers

```
int n = -5;

// Absolute value
abs(n);
```

```
// Division with negatives
-7 / 2;  // -3 in C++ (truncates towards zero)
-7 % 2;  // -1 in C++ (sign matches dividend)

// Comparison
-1 < 0;  // true
```

Large Numbers

```
// Maximum int (problem constraints)
int n = 1e9;
int n = INT_MAX;  // 2^31 - 1

// Use long long for larger values
long long n = 1e18;
long long n = LLONG_MAX; // 2^63 - 1

// Check for overflow
if (a > INT_MAX - b) {
    // a + b would overflow
}
```

Floating Point

```
// Precision issues
double a = 0.1 + 0.2;
a == 0.3; // False!

// Use epsilon for comparison
const double EPS = 1e-9;
abs(a - b) < EPS;

// Division
7 / 2;   // 3 (integer division)
7.0 / 2; // 3.5 (float division)
```

MATRIX/2D ARRAY EDGE CASES

Empty Matrix

```
vector<vector<int>> matrix;
vector<vector<int>> matrix = {{}};
```

```
if (matrix.empty() || matrix[0].empty()) {
    return {};
}
```

Single Element

```
vector<vector<int>> matrix = {{5}};
```

Single Row

```
vector<vector<int>> matrix = {{1, 2, 3, 4}};
```

Single Column

```
vector<vector<int>> matrix = {{1}, {2}, {3}, {4}};
```

Square vs Rectangle

```
// Square
vector<vector<int>> matrix = {{1,2}, {3,4}}; // 2x2

// Rectangle
vector<vector<int>> matrix = {{1,2,3}, {4,5,6}}; // 2x3
vector<vector<int>> matrix = {{1,2}, {3,4}, {5,6}}; // 3x2
```

Boundary Elements

```
int m = matrix.size();
int n = matrix[0].size();

// 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

```
vector<vector<int>> intervals;
if (intervals.empty()) {
    return {};
}
```

Single Interval

```
vector<vector<int>> intervals = {{1, 3}};
```

Non-Overlapping

```
vector<vector<int>> intervals = {{1,2}, {3,4}, {5,6}};
```

Fully Overlapping

```
vector<vector<int>> intervals = {{1,5}, {2,3}}; // [2,3] inside [1,5]
```

Touching Intervals

```
vector<vector<int>> intervals = {{1,2}, {2,3}}; // Share endpoint
// Does problem consider these overlapping?
```

Same Start/End

```
vector<vector<int>> intervals = {{1,3}, {1,3}}; // Identical
vector<vector<int>> intervals = {{1,3}, {1,4}}; // Same start
vector<vector<int>> intervals = {{1,3}, {2,3}}; // Same end
```

BINARY SEARCH EDGE CASES

Target Not in Array

```
vector<int> arr = {1, 3, 5, 7};
int target = 4; // Not present
// Should return -1 or insertion position?
```

Target at Boundaries

```
vector<int> arr = {1, 3, 5, 7};
int target = 1; // First element
int target = 7; // Last element
```

Duplicates

```
vector<int> arr = {1, 2, 2, 2, 3};
int target = 2;
// Return first occurrence? Last? Any?
```

All Elements Same

```
vector<int> arr = {5, 5, 5, 5, 5};
int target = 5;
```

SUBARRAY/SUBSTRING EDGE CASES

Empty Subarray

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

Single Element Subarray

```
vector<int> arr = {5};
// Subarray is just {5}
```

Entire Array

```
vector<int> arr = {1, 2, 3, 4};
```

```
// Entire array is valid subarray
```

Prefix/Suffix

```
vector<int> 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=nullptr)
- 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
- Division by zero
- Off-by-one errors in loops
- Correct inequality operators (< vs <=, > vs >=)

COMMON OFF-BY-ONE ERRORS

Loop Bounds

```
// WRONG
for (int i = 0; i < n; i++) {
    if (i + 1 < n) { // Redundant, range already handles this
        arr[i+1];
    }
}

// RIGHT
for (int i = 0; i < n - 1; i++) {
    arr[i+1];
}

// WRONG
for (int i = 1; i < n; i++) {
    arr[i-1]; // Processes arr[0] to arr[n-2], misses arr[n-1]
}

// RIGHT
for (int i = 0; i < n; i++) {
    if (i > 0) {
        arr[i-1];
    }
}
```

Slicing/Subvectors

```
// vector(begin, end) includes begin, excludes end
vector<int> arr = {0, 1, 2, 3, 4};
vector<int> sub(arr.begin() + 1, arr.begin() + 3); // {1, 2}, NOT {1, 2, 3}

// To include end
vector<int> sub(arr.begin() + 1, arr.begin() + 4); // {1, 2, 3}
```

Range Queries

```
// Sum from index i to j (inclusive)
// WRONG
int sum = 0;
for (int k = i; k < j; k++) { // Excludes j
    sum += arr[k];
}
```

```
// RIGHT
int sum = 0;
for (int k = i; k <= j; k++) { // Includes j
    sum += arr[k];
}

// Or using accumulate
int sum = accumulate(arr.begin() + i, arr.begin() + j + 1, 0);
```

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:

```
nullptr           // 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.