1. Notable Obstacles

One of the obstacles I had was with the CountRuns. This was because I initially had issues with nested loops causing incorrect counting. I fixed this by simplifying it to a single loop that compared it with the previous element. This worked better because it only compared each element to the one immediately preceding and incremented well.

Another obstacle I had was with PositionOfMax. I had to fix the comparison logic to handle identical elements by adding proper handling of the n = 0 case to return -1. This worked better because it properly handled edge cases and when elements were equal, it kept the earliest position.

2. Test Cases and Reasoning:

appendToAll Tests:

// Tests adding a string to end of each array element

string arr1[5] = { "a", "b", "c", "d", "e" };

assert(appendToAll(arr1, 5, "!") == 5); // Normal case: append "!" to all elements

assert(arr1[0] == "a!" && arr1[4] == "e!"); // Verify first and last elements

assert(appendToAll(arr1, -1, "x") == -1); // Error case: negative array size

assert(appendToAll(arr1, 0, "x") == 0); // Edge case: empty array

Reasoning: Tests basic appending, negative size, empty array, and verifies array contents

lookup Tests:

// Tests finding position of a target string in array

string arr2[5] = { "kamala", "donald", "jd", "tim", "jill" };

assert(lookup(arr2, 5, "tim") == 3); // Normal case: find existing element

assert(lookup(arr2, 5, "joe") == -1); // Normal case: element not in array

assert(lookup(arr2, 3, "jill") == -1); // Normal case: element outside search range

assert(lookup(arr2, -1, "tim") == -1); // Error case: negative array size

assert(lookup(arr2, 0, "tim") == -1); // Edge case: empty array

Reasoning: Tests finding existing element, non-existent element, partial array search, invalid size

positionOfMax Tests:

// Tests finding position of largest string

string arr3[6] = { "doug", "kamala", "melania", "usha", "gwen", "donald" };

assert(positionOfMax(arr3, 6) == 3); // Normal case: find max in full array

assert(positionOfMax(arr3, 2) == 1); // Normal case: find max in partial array

assert(positionOfMax(arr3, -1) == -1); // Error case: negative array size

assert(positionOfMax(arr3, 0) == -1); // Edge case: empty array

string maxArr[4] = { "tim", "tim", "tim", "tim" };

assert(positionOfMax(maxArr, 4) == 0); // Edge case: all elements equal

assert(positionOfMax(maxArr, 1) == 0); // Edge case: single element

Reasoning: Tests finding max, identical elements, empty array, negative size

rotateLeft Tests:

// Tests rotating array elements left from given position

string arr4[5] = { "kamala", "donald", "jd", "tim", "joe" };

assert(rotateLeft(arr4, 5, 1) == 1); // Normal case: rotate from position 1

assert(arr4[1] == "jd" && arr4[4] == "donald"); // Verify rotation results

assert(rotateLeft(arr4, -1, 1) == -1); // Error case: negative array size

assert(rotateLeft(arr4, 5, -1) == -1); // Error case: negative position

assert(rotateLeft(arr4, 5, 5) == -1); // Error case: position >= array size

Reasoning: Tests rotation, array modification verification, invalid position/size

countRuns Tests:

// Tests counting sequences of consecutive identical elements

string arr5[9] = { "melania", "doug", "gwen", "gwen", "jill", "jill", "jill", "gwen", "gwen" };

assert(countRuns(arr5, 9) == 5); // Normal case: multiple runs of different lengths

string arr5a[3] = { "tim", "tim", "tim" };

assert(countRuns(arr5a, 3) == 1); // Edge case: single run

assert(countRuns(arr5, -1) == -1); // Error case: negative array size

assert(countRuns(arr5, 0) == 0); // Edge case: empty array

string singleRun[1] = { "a" };

assert(countRuns(singleRun, 1) == 1); // Edge case: single element

Reasoning: Tests multiple runs, single run, single element, consecutive duplicates

flip Tests:

// Tests reversing elements in array

string arr6[6] = { "kamala", "doug", "", "jill", "jd", "donald" };

assert(flip(arr6, 4) == 4); // Normal case: flip first 4 elements

assert(arr6[0] == "jill" && arr6[3] == "kamala"); // Verify flip results

assert(flip(arr6, -1) == -1); // Error case: negative array size

assert(flip(arr6, 0) == 0); // Edge case: empty array

Reasoning: Tests array reversal, content verification, empty array, negative size

differ Tests:

// Tests finding first position where two arrays differ

string arr7a[6] = { "kamala", "doug", "", "jill", "jd", "donald" };

string arr7b[5] = { "kamala", "doug", "donald", "", "jd" };

assert(differ(arr7a, 6, arr7b, 5) == 2); // Normal case: arrays differ at position 2

assert(differ(arr7a, 2, arr7b, 1) == 1); // Normal case: shorter length comparison

assert(differ(arr7a, -1, arr7b, 5) == -1); // Error case: negative array size

Reasoning: Tests arrays of different lengths, finding first difference, invalid size

subsequence Tests:

// Tests finding a sequence within another sequence

string arr8a[6] = { "kamala", "tim", "usha", "gwen", "donald", "jd" };

string arr8b[3] = { "tim", "usha", "gwen" };

assert(subsequence(arr8a, 6, arr8b, 3) == 1); // Normal case: subsequence exists

string arr8c[2] = { "kamala", "gwen" };

assert(subsequence(arr8a, 5, arr8c, 2) == -1); // Normal case: subsequence doesn't exist

assert(subsequence(arr8a, 6, arr8b, 0) == 0); // Edge case: empty subsequence

assert(subsequence(arr8a, -1, arr8b, 3) == -1); // Error case: negative array size

Reasoning: Tests finding subsequence, non-existent subsequence, empty subsequence

lookupAny Tests:

// Tests finding first element that appears in both arrays

string arr9a[6] = { "kamala", "tim", "usha", "gwen", "donald", "jd" };

string arr9b[4] = { "donald", "melania", "gwen", "tim" };

assert(lookupAny(arr9a, 6, arr9b, 4) == 1); // Normal case: common element exists

string arr9c[2] = { "jill", "joe" };

assert(lookupAny(arr9a, 6, arr9c, 2) == -1); // Normal case: no common elements

assert(lookupAny(arr9a, -1, arr9b, 4) == -1); // Error case: negative array size

Reasoning: Tests finding first match, no matches, multiple possibilities

separate Tests:

string sepArr[6] = { "joe", "joe", "joe", "joe", "joe", "joe" };

int sepResult = separate(sepArr, 6, "joe");

assert(sepResult >= 0 && sepResult <= 6); // Edge case: all elements equal to separator

string arr10[6] = { "doug", "kamala", "melania", "usha", "gwen", "donald" };

int pos = separate(arr10, 6, "joe");

assert(pos >= 0 && pos <= 6); // Normal case: mixed elements

// Verify partition: elements before pos are < "joe"

for (int i = 0; i < pos; i++)

assert(arr10[i] < "joe");

// Verify partition: elements after pos are >= "joe"

for (int i = pos; i < 6; i++)

assert(arr10[i] >= "joe");

Reasoning: Tests separation around value, all equal elements, verifies ordering