**Name:** Eshan Uniyal **UID:** 205-172-354

# Obstacles

1. **Dealing with edge cases**: In several functions, the expected behaviour when given edge inputs (such as zero n) was not immediately obvious. Understanding what the functions were expected to return in such cases required a thorough inspection of the specification and an intuitive grasp of the goals of whichever user might’ve called that function from the library.
2. **Writing correct functions without creating additional arrays**: It was understandably ideal to not create additional arrays in functions such as moveToFront(), moveToBack(), deleteDups(), meld(), and split(), as their usage would required the allocation of extra memory. It was challenging to write such functions without creating additional arrays, especially in the case of meld() and split().
3. **Writing useful test cases**: Given the number of functions, several of which had a number of parameters, writing comprehensive test cases was tricky and often arduous. For instance, making sure the arrays were modified correctly (in functions deleteDups(), meld() and split(), for example) required writing long assert statements as C++ does not have an inbuilt function for array comparison.

# Tests

**Testing countMatches()**

string test\_countMatches[7] =

{ "romanoff", "thor", "rogers", "banner", "", "danvers", "rogers" };

assert(countMatches(test\_countMatches, -4, "stark") == -1); // negative n, returns -1

assert(countMatches(test\_countMatches, 0, "romanoff") == 0); // zero n, returns 0

assert(countMatches(test\_countMatches, 7, "rogers") == 2);

// two occurrences of "rogers" in first two elements

assert(countMatches(test\_countMatches, 7, "") == 1); // one occurrence of "" in first seven elements

assert(countMatches(test\_countMatches, 7, "rhodes") == 0); // no occurrences of "rhodes" in first seven elements

**Testing detectMatch()**

string test\_detectMatches[7] =

{ "romanoff", "thor", "rogers", "banner", "", "danvers", "rogers" };

assert(detectMatch(test\_detectMatches, -2, "romanoff") == -1); // negative n

assert(detectMatch(test\_detectMatches, 0, "romanoff") == -1);

// zero n, no "romanoff" found

assert(detectMatch(test\_detectMatches, 7, "rogers") == 2);

// "rogers" in index 2 in first seven elements

assert(detectMatch(test\_detectMatches, 2, "rogers") == -1);

// no occurrences of "roger" in first two elements

**Testing detectSequence()**

string test\_detectSequence[9] = { "thor", "romanoff", "danvers", "danvers", "stark",

"stark", "stark", "banner", "banner" };

int begin = 0;

int end = 0;

assert(detectSequence(test\_detectSequence, -2, "danvers", begin, end) == false

&& begin == 0 && end == 0);

// negative n; returns false and leaves begin and end unchanged

assert(detectSequence(test\_detectSequence, 0, "danvers", begin, end) == false

&& begin == 0 && end == 0);

// zero n; returns false and leaves begin and end unchanged

assert(detectSequence(test\_detectSequence, 9, "romanoff", begin, end) == true

&& begin == 1 && end == 1);

// sequence of length 1: returns true and sets begin to 1 and end to 1

assert(detectSequence(test\_detectSequence, 9, "rogers", begin, end) == false

&& begin == 1 && end == 1);

// sequence not found: returns false and leaves begin and end unchanged

assert(detectSequence(test\_detectSequence, 9, "banner", begin, end) == true

&& begin == 7 && end == 8);

// sequence of length 2: returns true and sets begin to 7 and end to 8

assert(detectSequence(test\_detectSequence, 9, "thor", begin, end) == true

&& begin == 0 && end == 0);

// sequence of length 1, first element in array: returns true and sets begin to 0 and end to 0

assert(detectSequence(test\_detectSequence, 6, "stark", begin, end) == true

&& begin == 4 && end == 5);

// sequence of length 2 (with more occurrences after interesting part of array)

// returns true and sets begin to 4 and end to 5

**Testing detectMin()**

string test\_detectMin[5] = { "danvers", "banner", "stark", "banner", "" };

assert(detectMin(test\_detectMin, -2) == -1); // negative n, returns -1

assert(detectMin(test\_detectMin, 0) == -1); // n equals zero, returns -1

assert(detectMin(test\_detectMin, 1) == 0);

// "danvers" at index 0 minimum in first element

assert(detectMin(test\_detectMin, 3) == 1);

// "banner" at index 1 minimum in first three elements

assert(detectMin(test\_detectMin, 4) == 1); // "banner" at index 1 minimum (there is

another at index 3, but the first one takes precedence)

assert(detectMin(test\_detectMin, 5) == 4);

// "" at index 4 minimum in all five elements

**Testing moveToBack()**

string test\_moveToBack[5] = { "danvers", "banner", "stark", "banner", "" };

assert(moveToBack(test\_moveToBack, -1, -2) == -1); // negative n, returns -1

assert(moveToBack(test\_moveToBack, -1, -2) == -1); // zero n, returns -1

assert(moveToBack(test\_moveToBack, 1, -2) == -1); // negative pos, returns -1

assert(moveToBack(test\_moveToBack, 0, 0) == -1);

// position index not in interesting indices, returns -1

assert(moveToBack(test\_moveToBack, 1, 1) == -1);

// position index not in interesting indices, returns -1

assert(moveToBack(test\_moveToBack, 5, 0) == 0 && test\_moveToBack[0] == "banner"

&& test\_moveToBack[1] == "stark" && test\_moveToBack[2] == "banner"

&& test\_moveToBack[3] == "" && test\_moveToBack[4] == "danvers");

// "danvers" at index 0 moved to end, returns 0

assert(moveToBack(test\_moveToBack, 5, 1) == 1 && test\_moveToBack[0] == "banner"

&& test\_moveToBack[1] == "banner" && test\_moveToBack[2] == ""

&& test\_moveToBack[3] == "danvers" && test\_moveToBack[4] == "stark");

// "stark" at index 1 moved to end, returns 1

assert(moveToBack(test\_moveToBack, 5, 2) == 2 && test\_moveToBack[0] == "banner"

&& test\_moveToBack[1] == "banner" && test\_moveToBack[2] == "danvers"

&& test\_moveToBack[3] == "stark" && test\_moveToBack[4] == "");

// "" at index 2 moved to end, returns 2

assert(moveToBack(test\_moveToBack, 5, 3) == 3 && test\_moveToBack[0] == "banner"

&& test\_moveToBack[1] == "banner" && test\_moveToBack[2] == "danvers"

&& test\_moveToBack[3] == "" && test\_moveToBack[4] == "stark");

// "stark" at index 3 moved to end, returns 3

assert(moveToBack(test\_moveToBack, 5, 4) == 4 && test\_moveToBack[0] == "banner"

&& test\_moveToBack[1] == "banner" && test\_moveToBack[2] == "danvers"

&& test\_moveToBack[3] == "" && test\_moveToBack[4] == "stark");

// "stark" at index 4 already at end, returns 4

**Testing moveToFront()**

string test\_moveToFront[4] = { "danvers", "banner", "thor", "rogers" };

assert(moveToFront(test\_moveToFront, -1, -2) == -1); // negative n, returns -1

assert(moveToFront(test\_moveToFront, 1, -2) == -1); // negative pos, returns -1

assert(moveToFront(test\_moveToFront, 0, 0) == -1);

// zero n, position index not in interesting indices; returns -1

assert(moveToFront(test\_moveToFront, 1, 1) == -1);

// position index not in interesting indices, returns -1

assert(moveToFront(test\_moveToFront, 3, 0) == 0 && test\_moveToFront[0] == "danvers"

&& test\_moveToFront[1] == "banner" && test\_moveToFront[2] == "thor");

// "danvers" at index 0 already at front, returns 0

assert(moveToFront(test\_moveToFront, 3, 1) == 1 && test\_moveToFront[0] == "banner"

&& test\_moveToFront[1] == "danvers" && test\_moveToFront[2] == "thor");

// "banner" at index 1 moved to front, returns 1

assert(moveToFront(test\_moveToFront, 3, 2) == 2 && test\_moveToFront[0] == "thor"

&& test\_moveToFront[1] == "banner" && test\_moveToFront[2] == "danvers");

// "thor" at index 2 moved to front, returns 2

**Testing detectDifference()**

string test\_dD1[5] = { "danvers", "banner", "stark", "banner", "" };

string test\_dD2[4] = { "danvers", "banner", "thor", "rogers" };

string test\_dD3[6] = { "romanoff", "banner", "thor", "rogers", "", "danvers" };

assert(detectDifference(test\_dD1, -1, test\_dD2, 0) == -1); // negative n1, returns -1

assert(detectDifference(test\_dD1, 0, test\_dD2, -1) == -1); // negative n2, returns -1

assert(detectDifference(test\_dD1, 0, test\_dD2, 3) == -1);

// cannot compare any elements, as n1 equals zero; returns -1

assert(detectDifference(test\_dD1, 3, test\_dD2, 0) == -1);

// cannot compare any elements, as n2 equals zero; returns -1

assert(detectDifference(test\_dD1, 1, test\_dD2, 3) == 1);

// equal up to index 1, returns 1 as indices ran out

assert(detectDifference(test\_dD1, 5, test\_dD2, 4) == 2);

// index 2 is unequal, returns 2

assert(detectDifference(test\_dD1, 5, test\_dD3, 5) == 0);

// index 0 is unequal, returns 0

**Testing deleteDups()**

string test\_deleteDups[6] =

{ "thor", "romanoff", "romanoff", "danvers", "stark", "stark" };

assert(deleteDups(test\_deleteDups, -2) == -1); // negative n, returns -1

assert(deleteDups(test\_deleteDups, 3) == 2

&& test\_deleteDups[0] == "thor" && test\_deleteDups[1] == "romanoff");

// deleted "romanoff" at index 3, only two elements remaining from first three elements; returns 2

assert(deleteDups(test\_deleteDups, 6) == 4

&& test\_deleteDups[0] == "thor" && test\_deleteDups[1] == "romanoff"

&& test\_deleteDups[2] == "danvers" && test\_deleteDups[3] == "stark");

// moved romanoff at index 3 and stark at index 5 to end, only four elements remaining from first six elements; returns 4

assert(deleteDups(test\_deleteDups, 6) == 6

&& test\_deleteDups[0] == "thor" && test\_deleteDups[1] == "romanoff"

&& test\_deleteDups[2] == "danvers" && test\_deleteDups[3] == "stark"

&& test\_deleteDups[4] == "romanoff" && test\_deleteDups[5] == "stark");

// all six elements remain ("romanoff" and "stark" were pushed to back in previous assert call); returns 6

**Testing contains()**

string contains\_test[10] =

{ "danvers", "thor", "stark", "banner", "romanoff", "stark" };

// main array against which the other arrays will be tested

string contains\_test\_2[10] = { "thor", "banner", "romanoff" };

string contains\_test\_3[10] = { "stark", "thor" };

string contains\_test\_4[10] = { "thor", "stark", "stark" };

string contains\_test\_5[10] = { "thor", "thor", "stark" };

assert(contains(contains\_test, -1, contains\_test\_2, 3) == false);

// negative n1, returns false

assert(contains(contains\_test, 6, contains\_test\_2, -1) == false);

// negative n2, returns false

assert(contains(contains\_test, 0, contains\_test\_2, 0) == true);

// empty array contains empty array; returns true

assert(contains(contains\_test, 6, contains\_test\_5, 0) == true);

// empty array contained in contains\_test; returns true

assert(contains(contains\_test, 0, contains\_test\_2, 3) == false);

// n2 > n1, returns false

assert(contains(contains\_test, 6, contains\_test\_2, 3) == true);

// "thor", "banner", "romanoff" appear in contains\_test in order; returns true

assert(contains(contains\_test, 6, contains\_test\_3, 2) == false);

// "stark" does not appear before "thor" in contains\_test; returns false

assert(contains(contains\_test, 6, contains\_test\_4, 3) == true);

// "thor", "stark", "stark" appear in contains\_test in order; returns true

assert(contains(contains\_test, 6, contains\_test\_5, 3) == false);

// No "thor" appears after "thor" in contains\_test; returns false

**Testing meld()**

string meld\_result[20];

string meld\_test\_1[5] = { "banner", "rhodes", "rogers", "stark", "tchalla" };

string meld\_test\_2[4] = { "danvers", "rogers", "rogers", "thor" };

string meld\_test\_3[3] = { "rogers", "danvers", "thor" };

assert(meld(meld\_test\_1, -1, meld\_test\_2, 3, meld\_result, 9) == -1);

// negative n1, returns -1

assert(meld(meld\_test\_1, 1, meld\_test\_2, -3, meld\_result, 9) == -1);

// negative n2, returns -1

assert(meld(meld\_test\_1, 2, meld\_test\_3, 3, meld\_result, 9) == -1);

// meld\_test\_3 is not nondecreasing

assert(meld(meld\_test\_1, 2, meld\_test\_2, 3, meld\_result, 4) == -1);

// n1 + n2 exceeds max, returns -1

assert(meld(meld\_test\_1, 0, meld\_test\_2, 4, meld\_result, 5) == 4

&& detectDifference(meld\_result, 4, meld\_test\_2, 4) == 4); // n1 equals zero, first four elements of meld\_result mirror meld\_test\_2; returns 4

assert(meld(meld\_test\_1, 5, meld\_test\_2, 0, meld\_result, 5) == 5

&& detectDifference(meld\_result, 5, meld\_test\_1, 5) == 5); // n2 equals zero, first five elements of meld\_result mirror meld\_test\_1; returns 5

assert(meld(meld\_test\_1, 5, meld\_test\_2, 4, meld\_result, 20) == 9 // usual case

&& meld\_result[0] == "banner" && meld\_result[1] == "danvers"

&& meld\_result[2] == "rhodes" && meld\_result[3] == "rogers"

&& meld\_result[4] == "rogers" && meld\_result[5] == "rogers"

&& meld\_result[6] == "stark" && meld\_result[7] == "tchalla");

// both arrays fully merged in meld\_test; returns 9

**Testing split()**

string split\_test\_1[6] = { "rhodes", "banner", "stark", "danvers", "thor", "rogers" };

assert(split(split\_test\_1, -1, "romanoff") == -1); // negative n, returns -1

assert(split(split\_test\_1, 0, "romanoff") == 0);

// returns n = 0 since no interesting elements >= splitter

assert(split(split\_test\_1, 6, "winter soldier") == 6);

// returns n = 6 since no elements >= splitter

assert(split(split\_test\_1, 6, "romanoff") == 4

&& split\_test\_1[0] == "rhodes" && split\_test\_1[1] == "banner"

&& split\_test\_1[2] == "rogers" && split\_test\_1[3] == "danvers"

&& split\_test\_1[4] == "thor" && split\_test\_1[5] == "stark");

// returns 4 (index of "thor" >= "romanoff"),

// array rearranged as { "rhodes","banner","rogers","danvers","thor","stark" }

string split\_test\_2[4] = { "romanoff", "rogers", "thor", "banner" };

assert(split(split\_test\_2, 4, "thor") == 3 && split\_test\_2[0] == "romanoff" && split\_test\_2[1] == "rogers"

&& split\_test\_2[2] == "banner" && split\_test\_2[3] == "thor");

// returns 3 (index of "thor" >= "thor")

// array rearranged as { "romanoff", ""rogers", "banner", "thor" }