The two main obstacles that I overcame were counting the number of duplicates in the *deleteDups* function correctly and figuring out how to not lose data in the *moveToFront* function when shifting values to the right. I really learned how valuable tracing through code was in this project. I overcame both of these challenges by tracking all the different variables on a notepad and seeing where the program went wrong. For the first problem, I was originally incrementing my “dups” variable incorrectly because I was shifting an element of the array to a temporary variable too early. Then, I still had a problem because I did not account for the case where the last element of the array was a duplicate. Once I realized these missteps, I fixed the error. For the second problem, I at first just copied the logic of my *moveToBack* function without realizing that the task was trickier than that. I had to preserve the data of an element before copying over it, so I ended up creating two temporary variables that allowed me to shift strings around. Again, using pen and paper to draw a diagram and visualize the data in temporary variables after each iteration of a loop was incredibly helpful.

Another small obstacle that I faced was getting caught in a while loop in my *contains* function. I did not account for the possibility that the condition for the while loop to break would not be fulfilled, so a couple of my *assert* statements did not pass. Once I added an *if* statement followed by a *break* statement, the error was fixed.

A list of the test data that could be used to thoroughly test your functions, along with the reason for each test, follows:

string h[7] = { "romanoff", "thor", "rogers", "banner", "", "danvers", "rogers" };

1. assert(countMatches(h, 7, "rogers") == 2); //straightforward test of the function
2. assert(countMatches(h, 7, "") == 1); //see if it treats empty string properly
3. assert(countMatches(h, 7, "rhodes") == 0); //returns 0 if no matches
4. assert(countMatches(h, 0, "rogers") == 0); //see if it handles n = 0 correctly
5. assert(detectMatch(h, 7, "rogers") == 2); //straightforward test of the function
6. assert(detectMatch(h, 2, "rogers") == -1); //see what function does when there is no match

**int** bg;

**int** en;

1. assert(detectSequence(h, 7, "banner", bg, en) && bg == 3 && en == 3); // see what function does when there is only 1 appearance of target string

string zz[5] ={"romanoff", "romanoff", "romanoff", "thor", "romanoff" };

1. assert(detectSequence(zz, 5, "romanoff", bg, en) && bg == 0 && en == 2); //checks if function works for a consecutive sequence and ends when the scanned value is not the target

string g[4] = { "romanoff", "thor", "banner", "danvers" };

1. assert(detectMin(g, 4) == 2); //straightforward test of the function
2. assert(detectDifference(h, 4, g, 4) == 2); //n1 and n2 are equal to each other but arrays differ
3. assert(contains(h, 7, g, 4)); //straightforward test of the function
4. assert(moveToBack(g, 4, 1) == 1 && g[1] == "banner" && g[3] == "thor"); //straightforward test of the function

string f[4] = { "danvers", "banner", "thor", "rogers" };

1. assert(moveToFront(f, 4, 2) == 2 && f[0] == "thor" && f[2] == "banner"); //straightforward test of the function

string e[5] = { "danvers", "danvers", "danvers", "thor", "thor" };

1. assert(deleteDups(e, 5) == 2 && e[1] == "thor"); //straightforward test of the function

string x[4] = { "rhodes", "rhodes", "tchalla", "thor" };

string y[4] = { "banner", "danvers", "rhodes", "rogers" };

string z[10];

1. assert(meld(x, 4, y, 4, z, 10) == 8 && z[5] == "rogers"); //straightforward test of the function
2. assert(split(h, 7, "rogers") == 3); //straightforward test of the function

string cast[5] = { "danvers", "thor", "stark", "banner", "romanoff" };

string roles[4] = { "danvers", "thor", "barton", "rhodes" };

1. assert(detectDifference(cast, 5, roles, 4) == 2); // n1 and n2 are unequal and arrays differ
2. assert(detectDifference(cast, 2, roles, 1) == 1); //tests case where one string runs out before there are even any differences

1. assert(deleteDups(d, 9) == 5 && d[0] == "thor" && d[1] == "romanoff" && d[2] == "danvers" && d[3] == "stark" && d[4] == "danvers"); //heavy duty array, tests nested for loops better

string big[10] = { "danvers", "thor", "stark", "banner", "romanoff", "stark" };

string little1[10] = { "thor", "banner", "romanoff" };

1. assert(contains(big, 6, little1, 3) == **true**); //makes sure we are not just looking at consecutive containment

string little2[10] = { "stark", "thor" };

1. assert(contains(big, 6, little2, 2) == **false**); //makes sure we can exit the loop when there fails to be a match

string little3[10] = { "thor", "stark", "stark" };

1. assert(contains(big, 6, little3, 3) == **true**); //see if program can handle repeat value in little string when big string contains little string

string little4[10] = { "thor", "thor", "stark" };

1. assert(contains(big, 6, little4, 3) == **false**); //see if program can handle repeat value in little string when big string does not contain little string
2. assert(contains(big, 6, little4, 0) == **true**); // 0 strings must be contained in the first array

string xx[5] = { "banner", "rhodes", "rogers", "stark", "tchalla" };

string yy[4] = { "danvers", "rogers", "rogers", "thor" };

string zzz[20];

string aa[3] = {"thor", "banner", "stark" };

1. assert(meld(aa, 3, yy, 4, zzz, 9 ) == -1); //tests if array is not in non-decreasing order
2. assert(meld(xx, 5, yy, 4, zzz, 7) == -1) // tests if max is less than n1 + n2

string ff[6] = { "rhodes", "banner", "stark", "danvers", "thor", "rogers" };

1. assert(split(ff, 6, "romanoff") == 4); //verifying correctness of function

My progam handles all 27 of these test cases correctly.