**Notable obstacles overcome:**

I had to make sure I wasn’t accessing any out of bound positions when writing my loops as well as figure out how to effectively loop through multiple arrays or when sorting. In addition, when writing each separate function, I had to make sure I was accounting for the basic error returns each time, with -1 being returned for invalid parameters. For this project, it was especially difficult to process the spec, since there were so many different aspects within each function alongside considering special cases, such as when int n = 0 and returning the correct response for each respective function.

**Test cases:**

appendToAll

string people[5] = { "donald", "joe", "mike", "lindsey", "kamala" };

int j = appendToAll(people, 5, "!!!");

* Test case with no special circumstances

string people[5] = { "donald", "joe", "mike", "lindsey", "kamala" };

int j = appendToAll(people, -1, "!!!");

* Tests Negative n that should return -1 from the function

string people[5] = { "donald", "joe", "mike", "lindsey", "kamala" };

int j = appendToAll(people, 0, "!!!");

* Appending to 0 elements // should return an unchanged array

string people[5] = { "donald", "joe", "mike", "lindsey", "kamala" };

int j = appendToAll(people, 5, "");

* Appending the empty string

string people[5] = { "donald", "joe", "mike", "lindsey", "kamala" };

int j = appendToAll(people, 10, "!!!");

* When N is greater than number of elements in array
* Code wouldn’t handle this correctly b/c it’d be accessing out of bounds

lookup

string a [5] = {“mango”, “orange”, “apple”, “pineapple”, “pear”};

Int x = lookup (a, 5, “apple”);

* Case with no special circumstances

string a [5] = {“mango”, “orange”, “apple”, “pineapple”, “pear”};

Int x = lookup (a, -4,” apple”);

* Tests a negative number as n// tests that function recognizes n < 0// would return -1

string a [5] = {“mango”, “orange”, “apple”, “pineapple”, “pear”};

Int x = lookup (a, 0, “apple”);

* Looking at 0 of the elements

string a [5] = {“mango”, “orange”, “apple”, “pineapple”, “pear’};

Int x = lookup (a, 10,” apple”);

* More elements than is in array // code wouldn’t handle correctly b/c it’s accessing out of bounds elements

string a [5] = {“mango”, “orange”, “apple”, “pineapple”, “pear’};

Int x = lookup (a, 5, “”);

* Searching for an empty string

string a [5] = {“apple”, “orange”, “apple”, “pineapple”, “pear’};

Int x = lookup (a, 5,” apple”);

* When target shows up more than once // should return first position where it’s found

string a [5] = {“mango”, “orange”, “orange”, “pineapple”, “pear’};

Int x = lookup (a, 5,” apple”);

* Target is not present in the array // should return -1

string a [5] = {“mango”, “orange”, “orange”, “pineapple”, “pear’};

Int x = lookup (a, 5,” apple”);

* Fewer elements (int n) than in array

string a [5] = {“mango”, “orange”, “orange”, “pineapple”, “pear’};

Int x = lookup (a, 0,” apple”);

* Checking 0 elements

positionOfMax

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int k = positionOfMax(candidate, 6);

* Case with no special circumstances

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int k = positionOfMax(candidate, 0);

* 0 interesting elements //should return -1

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int k = positionOfMax(candidate, -6);

* Negative number of elements // should return -1

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int k = positionOfMax(candidate, 9);

* Out of bounds with n > the size of the array // out of bounds access

string candidate[6] = { "susan", "lindsey", "susan", "susan", "joe", "donald" };

int k = positionOfMax(candidate, 6);

* More than one greatest string// should return position of first occurrence

string candidate[6] = { "susan", "lindsey", "susan", "susan", "joe", "donald" };

int k = positionOfMax(candidate, 4);

* Fewer elements (int n) than in the array

rotateLeft

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, 5, 1);

* Case with no special circumstances

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, 5, 4);

* Position is last element// would not change the array

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, -1, 1);

* Negative int n // would return -1

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, 5, -6);

* Negative position //would return -1

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, -5, -1);

* Negative int n and position //would return -1

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, 5, 15);

* Out of bounds position

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, 2, 1);

* Fewer elements (int n) than in array

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, 15, 1);

* Int n would be out of bounds

string politician[5] = { "kamala", "jamie", "lindsey", "sara", "mark" };

int m = rotateLeft(politician, 0, 6);

* Rotating 0 elements // would be an error

countRuns

string d[9] = {

"susan", "donald", "mike", "mike", "joe", "joe", "joe", "mike", "mike"};

int p = countRuns(d, 9);

* Case with no special circumstances

string d[9] = {

"susan", "donald", "mike", "mike", "joe", "joe", "joe", "mike", "mike"};

int p = countRuns(d, 10);

* Case with out of bounds access // int n is greater than array size

string d[9] = {

"susan", "donald", "mike", "mike", "joe", "joe", "joe", "mike", "mike"};

int p = countRuns(d, -4);

* Negative int n// would return -1

string d[9] = {

"susan", "donald", "mike", "mike", "joe", "joe", "joe", "mike", "mike"};

int p = countRuns(d, 4);

* Int n is smaller than size of array // looking at fewer elements

string d[9] = {

"susan", "susan", "susan", "susan", "susan", "susan", "susan", "susan", "susan"};

int p = countRuns(d, 9);

* All identical items in a

string d[9] = {

"susan", "donald", "mike", "ann", "joe", "jeffrey", "fred", "isabel", "lisa"};

int p = countRuns(d, 9);

* No identical items in a

string d[9] = {

"susan", "donald", "mike", "ann", "joe", "jeffrey", "fred", "isabel", "lisa"};

int p = countRuns(d, 0);

* Looking at 0 elements in the array

flip

string folks[6] = { "donald", "mike", "", "susan", "sara", "jamie" };

int q = flip(folks, 6);

* Case with no special circumstances

string folks[6] = { "donald", "mike", "", "susan", "sara", "jamie" };

int q = flip(folks, -9);

* Negative int n

string folks[6] = { "donald", "mike", "", "susan", "sara", "jamie" };

int q = flip(folks, 2);

* Accessing fewer elements

string folks[7] = { "donald", "mike", "", "susan", "sara", "jamie" };

int q = flip(folks, 7);

* Odd number of elements to flip

string folks[6] = { "donald", "mike", "", "susan", "sara", "jamie" };

int q = flip(folks, 15);

* Out of bounds accessing when int n is greater than array size

string folks[6] = { "donald", "mike", "", "susan", "sara", "jamie" };

int q = flip(folks, 0);

* Accessing 0 elements

Differ

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "d", "e" , “f”};

int r = differ(folks, 6, group, 6);

* Equal arrays in all elements // Equal number of interesting elements, no special circumstances

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, 6, group, -6);

* Negative n2 //should return -1

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, -6, group, 6);

* Negative n1//should return -1

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, -6, group, -6);

* Negative n1 and n2// should return -1

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, 2, group, 2);

* Fewer interesting elements (n1 & n2) than actually in array

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, 0, group, 0);

* 0 interesting elements

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, 10, group, 6);

* N1 greater than number of elements// would go out of bounds

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, 6, group, 10);

* N2 greater than number of elements /// would go out of bounds

string folks[6] = { "a", "b", "c", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, 3, group, 6);

* Different number of interesting elements (n1 and n2)

string folks[6] = { "a", "b", "c", "d", "e", "f" };

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

int r = differ(folks, 6, group, 1);

* Different size arrays / different number of elements

string folks[6] = { "a", "b", "e", "d", "e", "f" };

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, 6, group, 6);

* Unequal elements in same size arrays

string folks[4] = { "a", "b", "e", "d"};

string group[6] = { "a", "b", "c", "e", "d" , “f”};

int r = differ(folks, 4, group, 6);

* Unequal elements in diff size arrays

string folks[4] = { "a", "b", "c", "d"};

string group[6] = { "a", "b", "c", "d", "d" , “d”};

int r = differ(folks, 4, group, 6);

* Different elements in elements past length of first array

Subsequence

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string names1[10] = { "mark", "sara", "martha" };

int t = subsequence(names, 6, names1, 3);

* Case with no special circumstances // different lengths

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string names1[10] = { "mark", "sara", "martha" };

int t = subsequence(names1, 3, names2, 6);

* Looking for a longer array within shorter

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string names1[10] = { "mark", "sara", "martha" };

int t = subsequence(names, -1, names1, 3);

* Negative n1 // should return -1

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string names1[10] = { "mark", "sara", "martha" };

int t = subsequence(names, 6, names1, -3);

* Negative n2 // should return -1

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string names1[10] = { " ", " ", " " };

int t = subsequence(names, 6, names1, 3);

* Doesn’t contain the sequence // should return -1

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string names1[10] = { "mark", "sara", "martha" };

int t = subsequence(names, 6, names1, 0);

* Subsequence with 0 elements, would start at 0

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string names1[10] = { "mark", "sara", "martha" };

int t = subsequence(names, 6, names1, 15);

* N2 greater than size of array// would go out of bounds

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string names1[10] = { "mark", "sara", "martha" };

int t = subsequence(names, 15, names1, 6);

* N1 greater than size of array// would go out of bounds

lookupAny

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string set1[10] = { "jamie", "donald", "martha", "mark" };

int v = lookupAny(names, 6, set1, 4);

* Case with no special circumstances

string names[10] = { "kamala", "xyz", "sara", "xy", "donald", "lindsey" };

string set1[10] = { "jamie", "abc", "martha", "mark" };

int v = lookupAny(names, 6, set1, 4);

* No matching elements // should return -1

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string set1[10] = { "jamie", "donald", "martha", "mark" };

int v = lookupAny(names, -5, set1, 4);

* Negative n1 // should return -1

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string set1[10] = { "jamie", "donald", "martha", "mark" };

int v = lookupAny(names, 6, set1, -2);

* Negative n2 // should return -1

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string set1[10] = { "jamie", "donald", "martha", "mark" };

int v = lookupAny(names, 6, set1, 0);

* 0 interesting elements in a2

string names[10] = { "kamala", "mark", "sara", "martha", "donald", "lindsey" };

string set1[10] = { "jamie", "donald", "martha", "mark" };

int v = lookupAny(names, 0, set1, 4);

* 0 interesting elements in a1

Divide

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int x = divide(candidate, 6, "kamala");

* Case with no special circumstances

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int x = divide(candidate, 6, "");

* Empty string as divider

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int x = divide(candidate, -6, "kamala");

* Negative n // would return -1

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int x = divide(candidate, 6, "zzzzzzz");

* No elements greater than divider // would return n

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int x = divide(candidate, 6, "a");

* All elements greater than divider

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int x = divide(candidate, 7, "kamala");

* Out of bounds n, greater than actual # of elements in array

string candidate[6] = { "jamie", "lindsey", "mark", "susan", "joe", "donald" };

int x = divide(candidate, 0, "kamala");

* Looking at no elements // will not find > divider