1. Notable Obstacles

While doing this project, I had to take my time to plan out most of my functions before I began writing them because there were so many different functions to implement. I also practiced testing out each function as I went along so that I wouldn’t have major issues at the end I didn’t know how to fix because I rushed through everything.

The main obstacle I had to overcome was thinking about the algorithmic part of the functions. I had to plan out the logic in how each function would implement the required task successfully. This took some extra time, but I managed to get everything done well, I believe.

1. Test Data

**int appendToAll(string a[], int n, string value);**

Arrays:

string names[5] = {“chris”, “matthew”, “Jessica”, “james”, “jim”};

Test data + reasons:

|  |  |
| --- | --- |
| Data (a[], n, value) | Why |
| (names, 5, “!!”) | Tests to see if it goes through the whole array and appends correctly |
| (names, 3, “!?”) | Tests to see if it just hits three elements |
| (names, 0, “??”) | Tests when number of indexes is 0 because 0 is a special case |
| (names, -3, “?!”) | Tests when n is below 0 because indexes of an array can’t be negative |

**int lookup(const string a[], int n, string target);**

Arrays:

string lookupnames[6] = {“chris”, “matthew”, “Jessica”, “james”, “jim”, “james”};

Test data + reasons:

|  |  |
| --- | --- |
| Data (a[], n, target) | Why |
| (lookupnames, 6, “james”) | Tests the case where a target appears more than once; it should return the lower index |
| (lookupnames, 5, “jeff”) | Tests where the target isn’t found |
| (lookupnames, 0, “chris”) | Tests when n is 0 |
| (lookupnames, -1, “Jessica”) | Tests when n is less than 0 |

**int positionOfMax(const string a[], int n);**

Arrays:

string posnames[9] = {“chris”, “matthew”, “jessica”, “james”, “jim”, “james”, “zeppelin”, “zeppelin”, “xavier”};

string posnames2[9] = {“chris”, “matthew”, “Jessica”, “james”, “jim”, “james”, “zeppelin”, “zeppelin”, “Xavier”};

Test data + reasons:

|  |  |
| --- | --- |
| Data (a[], n) | Why |
| (posnames, 9) | Tests for all lowercase and for when the largest string appears more than once. |
| (posnames2, 9) | Tests for a mix of uppercase and lowercase |
| (posnames2, 7) | Just tests for a subset rather than the whole array |
| (posnames, 0) | Test for n=0 |
| (posnames, -1) | Test for n<0 |

**int rotateLeft(string a[], int n, int pos);**

Arrays:

string candidates[5] = { “mike”, “donald”, “lindsey”, “nancy”, “adam” };

Test data + reasons:

|  |  |
| --- | --- |
| Data (a[], n, pos) | Why |
| (candidates, 5, 1) | Test a normal case |
| (candidates, 5, 0) | Tests to see pos at the 0th index |
| (candidates, 5, 4) | Tests the last index for pos |
| (candidates, 3, 1) | Tests a subset of the array |
| (candidates, 0, 1) | Tests for n=0 |
| (candidates, -1, 1) | Tests for n<0 |

**int countRuns(const string a[], int n);**

Arrays:

string manynames[9].= { “rudy”, “adam”, “mike”, “mike”, “fiona”, “fiona”, “fiona”, “mike”, “mike”};

Test data + reasons:

|  |  |
| --- | --- |
| Data (a[], n) | Why |
| (manynames, 9) | Test for a normal case |
| (manynames, 5) | Tests a subset of that case |
| (manynames, 0) | Tests for n=0 |
| (manynames, -1) | Tests for n<0 |

**int flip(string a[], int n);**

Arrays:

string flipeven[6] = {“jack”, “judy”, “rudy”, ““, “stephanie”, “alyssa”};

string flipodd[7] = {“jack”, “judy”, “rudy”, “bobby”, “joey”, “amy”, “alphonse”};

Test data + reasons:

|  |  |
| --- | --- |
| Data (a[], n) | Why |
| (flipeven, 6) | Tests a normal case for an array with an even number of elements |
| (flipodd, 7) | Tests a normal case for an array with an odd number of elements |
| (flipodd, 6) | Tests a subset even amount for an odd array |
| (flipodd, 5) | Tests a subset odd amount |
| (flipeven, 0) | Tests for n=0 |
| (flipeven, -1) | Tests for n<0 |

**int differ(const string a1[], int n1, const string a2[], int n2);**

Return the position of the first corresponding elements of a1 and a2 that are not equal. n1 is the number of interesting elements in a1, and n2 is the number of interesting elements in a2. If the arrays are equal up to the point where one or both runs out, return whichever value of n1 and n2 is less than or equal to the other. For example,

string folks[7] = { “adam”, ““, “fiona” “mike”, “rudy”, “nancy”, “donald” };

string group[6] = { “adam”, ““, “fiona” “donald”, “mike”, “rudy” };

int r = differ(folks, 7, group, 6); // returns 3

int s = differ(folks, 2, group, 3); // returns 2

Arrays:

string folks[7] = { “adam”, ““, “fiona”, “mike”, “rudy”, “nancy”, “donald” };

string group1[6] = { “adam”, ““, “fiona”, “donald”, “mike”, “rudy” };

string group2[3] = {“adam, ““, “fiona”};

string group3[3] = {“jim”, “jack,” “bob”};

Test data + reasons:

|  |  |
| --- | --- |
| Data (a1[], n1, a2[], n2) | Why |
| (folks, 7, group1, 6) | Normal test for whole arrays |
| (folks, 2, group1, 3) | Tests with smaller values of n1 than n2 |
| (folks, 5, group2, 3) | Tests for when the entire group is the same |
| (folks, 5, group3, 3) | Tests for when the groups are different |
| (folks, 0, group3, 0) | Tests for when both n’s are 0 |
| (folks, 0, group3, 2) | Tests for when only n1 is 0 |
| (folks, -1, group2, 1) | Tests for when n1 is < 0 |
| (folks, 2, group2, -1) | Tests for when n2 is < 0 |

**int subsequence(const string a1[], int n1, const string a2[], int n2);**

Arrays:

string villagers[7] = {“Jackie”, “jimmie”, “joey”, “jeff,” “mike”, “andy”, “smelly”};

string sub1[3] = {“jimmie, joey”, “jeff”};

string sub2[3] = {“andy”, “smelly”, “mike”};

string sub3[3] = {“Zimbabwe”, “somewhere”, “a name”};

Test data + reasons:

|  |  |
| --- | --- |
| Data (a1[], n1, a2[], n2) | Why |
| (villagers, 7, sub1, 3) | Tests a normal case that works |
| (villagers, 7, sub2, 3) | Tests for a case that might go out of bounds if code is written incorrectly |
| (villagers, 7, sub3, 3) | Tests for a case that just doesn’t work |
| (villagers, 7, sub3, 0) | Tests for when the sequence is 0 elements. Should return 0. |
| (villagers, 0, sub 3, 3) | Tests when n1 = 0 |
| (villagers, -1, sub3, 3) | Tests when n1<0 |
| (villagers, 7, sub3, -1) | Tests when n2<0 |

**int lookupAny(const string a1[], int n1, const string a2[], int n2);**

Arrays:

string namestosearch[6] = { “gordon”, “marie”, “nancy”, “mick”, “adam”, “lindsey” };

string set1[4] = { “donald”, “adam”, “mick”, “marie” };

string set2[2] = { “rudy”, “fiona” };

Test data + reasons:

|  |  |
| --- | --- |
| Data (a1[], n1, a2[], n2) | Why |
| (namestosearch, 6, set1, 4) | Basic test case with multiple of set1 existing in namestosearch |
| (namestosearch, 6, set2, 2) | Test case that doesn’t work |
| (namestosearch, 0, set1, 4) | Testing when n1=0 |
| (namestosearch, 6, set1, 0) | Testing when n2=0 |
| (namestosearch, -1, set1, 4) | Testing when n1<0 |
| (namestosearch, 6, set1, -1) | Testing when n2<0 |

**int separate(string a[], int n, string separator);**

Arrays:

string people[6] = { “donald”, “lindsey”, “marie”, “rudy”, “fiona”, “adam” };

string people2[4] = { “marie”, “nancy”, “lindsey”, “mike” };

Test data + reasons:

|  |  |
| --- | --- |
| Data (a[], n, separator) | Why |
| (people, 6, “gordon”) | Tests for a case that the array doesn’t contain |
| (people2, 4, “mike”) | Tests for a case that the array does contain |
| (people, 0, “john”) | Tests for n=0 |
| (people, -1, “john”) | Tests for n<0 |
| (people, 6, “zebra”) | Tests for when separator is greater than everything |
| (people, 6, “abbie”) | Tests for when separator is less than everything |