1. One notable obstacle that I encountered was figuring out how to incorporate a temporary string within my program so I can store necessary information when needed. At first, I wasn’t exactly sure when I needed to use a temporary string and how I was supposed to store information in the temporary string. However, as I worked through the different arrays, there were multiple functions where I realized I needed to utilize a temporary string. With more and more applications, I began to understand that a temporary string is required in situations where I need to flip the positions of different strings. I familiarized myself with the general format of storing information into a temp, swapping the positions of desired strings, and then calling the temp into the later position and overall, I was able to overcome this obstacle of incorporating a temporary string. Another notable obstacle that I overcame was figuring out how to iterate through two different items. I realized that if I use two for loops, I can iterate through the first array with the first for loop and then compare(or do whatever the task called for) with all the strings in the second array using a second for loop. This came especially handy when there were two different arrays of strings or when I just needed to compare two different incrementing values.
2. **Test Cases**

string c[4] = { "ma", "can", "tu", "do" };

string d[5] = { "mulan", "mulan", "mulan", "belle", "belle" };

string e[4] = { "ariel", "tiana", "", "belle" };

string f[3] = { "tiana", "ariel", "raya" };

string g[4] = { "moana", "mulan", "belle", "raya" };

string h[7] = { "moana", "mulan", "ariel", "tiana", "", "belle", "elsa" };

1. int reduplicate(string a[], int n)

* assert(reduplicate(c, 4) == 4 && c[0] == "mama" && c[3] == "dodo") tests a single correct case
* assert(reduplicate(c, 2) == 2 && c[0] == "mama" && c[1] == "cancan") tests another correct case
* assert(reduplicate(c, 0) == 0 && c[0] == "ma" && c[1] == "can") tests when n = 0 and nothing changes within the array
* assert(reduplicate(c, -3) == -1 && c[0] == "ma" && c[1] == "can") tests when n is negative

1. int locate(const string a[], int n, string target)

* assert(locate(h, 7, "belle") == 5) tests a single correct case
* assert(locate(h, 7, "ariel") == 2) tests another correct case
* assert(locate(h, 0, "ariel") == -1) tests an incorrect case where “ariel” is not at position 0
* assert(locate(h, -5, "ariel") == -1) tests when n is negative
* assert(locate(h, 5, "belle") == -1) tests when “belle” is not part of the array
* assert(locate(d, 5, "belle") == 3) tests when there is more than one string that matches target

1. int locationOfMax(const string a[], int n)

* assert(locationOfMax(h, 7) == 3) tests a single correct case
* assert(locationOfMax(g, 4) == 3) tests another correct case
* assert(locationOfMax(h, 0) == -1) tests when n is 0
* assert(locationOfMax(h, -1) == -1) tests when n is a negative value
* assert(locationOfMax(d, 5) == 0) tests when more than one string is >= every string in the array

1. int circleLeft(string a[], int n, int pos)

* assert(circleLeft(g, 4, 1) == 1 && g[1] == "belle" && g[3] == "mulan") tests a single correct case
* assert(circleLeft(h, 7, 2) == 2 && h[1] == "mulan" && h[3] == "" && h[5] == "elsa" && h[6] == "ariel") tests another correct case
* assert(circleLeft(g, 0, 1) == -1) tests when n is 0
* assert(circleLeft(g, 0, 1) == -1) tests when n is negative
* assert(circleLeft(c, 4, 0) == 0 && c[0] == "can" && c[1] == "tu" && c[2] == "do" && c[3] == "ma") tests when pos is 0
* assert(circleLeft(c, 4, -2) == -1) tests when pos is negative
* assert(circleLeft(c, 4, 5) == -1) tests when pos is greater than n

1. int enumerateRuns(const string a[], int n)

* assert(enumerateRuns(d, 5) == 2) tests a single correct case
* assert(enumerateRuns(g, 4) == 4) tests another correct case
* assert(enumerateRuns(g, -3) == -1) returns -1 if n is negative
* assert(enumerateRuns(g, 0) == 0) returns 0 if n is 0

1. int flip(string a[], int n)

* assert(flip(f, 3) == 3 && f[0] == "raya" && f[2] == "tiana") tests a single correct case
* assert(flip(e, 3) == 3 && e[0] == "" && e[1] == "tiana" && e[2] == "ariel" && e[3] == "belle") tests another correct case
* assert(flip(e, -2) == -1) tests when n is negative
* assert(flip(e, 0) == 0) tests when n is 0

1. int locateDifference(const string a1[], int n1, const string a2[], int n2)

* assert(locateDifference(h, 4, g, 4) == 2) tests a single correct case
* assert(locateDifference(d, 5, g, 4) == 0) tests when there is a difference at index 0
* assert(locateDifference(g, 4, h, 2) == 2) tests when the second array runs out
* assert(locateDifference(g, 2, h, 4) == 2) tests when the first array runs out
* assert(locateDifference(g, -1, h, 4) == -1) tests when n1 is negative
* assert(locateDifference(g, 0, h, 4) == 0) tests when one of the arrays has size 0

1. int subsequence(const string a1[], int n1, const string a2[], int n2)

* assert(subsequence(h, 7, e, 4) == 2) tests a single correct case
* assert(subsequence(h, 3, e, 1) == 2) tests when the second array only has one string
* assert(subsequence(h, 7, g,4) == -1) tests when there are no contiguous subsequences
* assert(subsequence(h, -3, g, 4) == -1) tests when n1 is negative
* assert(subsequence(h, 7, e, 2) == 2) tests a fragment of an array
* assert(subsequence(i, 6, f, 3) == 0) tests when the subsequence appears more than once

1. int locateAny(const string a1[], int n1, const string a2[], int n2)

* assert(locateAny(h, 7, f, 3) == 2) tests a single correct case
* assert(locateAny(h, 0, f, 3) == -1) tests when the size of array a1 is 0
* assert(locateAny(h, 7, c, 4) == -1) tests when there are no matches
* assert(locateAny(h, -4, f, 3) == -1) tests when size of array is negative

1. int separate(string a[], int n, string separator)

* assert(separate(h, 7, "elsa") == 3) tests a single correct case
* assert(separate(h, 0, "elsa") == 0) tests when the size of an array is 0
* assert( separate(cast, 6, "merida") == 3) tests another correct case
* assert(separate(cast2, 4, "raya") == 2) tests a third correct case where n is an even number
* assert(separate(h, -2, "elsa") == -1) tests when size of array is less than 0