Kelly Yu

Dr. Smallberg

CS 31

October 29, 2021

**CS 31 Project 4, Fall 2021**

**1. Notable Obstacles**

One notable obstacle I faced was planning out my code and starting with simple code first to incrementally debug and build the entire function. My mind tends to run in many what-if’s, such as what would the outcome be if n == 1, or n == 0, or n > 1, or n < 0. I would run through many scenarios in my mind, most of which I still feel unsure about since I can’t really check my work if my work is all in my head. What I tried to this time was be patient, write out pseudocode in a document, and make sure my ideas are just about thorough and reasonable enough before implementing them with code. I still need to work on building code incrementally, because even when planning things out, I run off in one direction all the way to the end and then realize a portion of my idea does not meet the project specifications. Either way, the pseudocode helped me practice understanding the why’s behind code. Another obstacle I faced was during debugging. I printed out different values to check if my code was working properly, but I wasn’t careful in choosing what to print out, so I confused myself for a few hours. Next time, I’ll check one test case at a time when the test cases involve printing out values. My code worked properly but printing out the wrong them caused me to think there must be a bug in my code. I also need to work on taking breaks and switching gears. Coming back to the code with fresh eyes is better than searching for that one bug for hours.

**2. Test Code**

**int** reduplicate(string a[], **int** n)

|  |  |
| --- | --- |
| Cannot work with negative number of elements in array | string a[3] = { "a", "b", "c" };  assert(reduplicate(a, -1) == -1); |
| Work with 0 elements in array;  returns 0 and nothing is changed in array | string a[3] = { "a", "b", "c" };  assert(reduplicate(a, 0) == 0 && a[0]=="a" && a[1]=="b" && a[2]=="c"); |
| Work with 1 out of 3 elements in array;  returns 1 and reduplicates only a portion of array | string a[3] = { "a", "b", "c" };  assert(reduplicate(a, 1) == 1 && a[0]=="aa" && a[1]=="b" && a[2]=="c"); |
| Work with 3 out of 3 elements in array;  returns 3 and reduplicates all elements in array | string a[3] = { "a", "b", "c" };  assert(reduplicate(a, 3) == 3 && a[0]=="aa" && a[1]=="bb" && a[2]=="cc"); |
| Nothing is performed with an empty array | string a[0] = {};  assert(reduplicate(a, 0) == 0); |

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

|  |  |
| --- | --- |
| Checks all elements in the array and returns position of string that matches target “c” | string a[3] = { "a", "b", "c" };  assert(locate(a, 3, "c") == 2); |
| Checks 2 out of 3 elements in the array and returns position of string that matches target “b” | string a[3] = { "a", "b", "c" };  assert(locate(a, 2, "b") == 1); |
| Checks 2 out of 3 elements in the array for target “c”;  returns -1 since range of elements checked does not contain “c” | string a[3] = { "a", "b", "c" };  assert(locate(a, 2, "c") == -1); |
| Checks all elements in string and returns smallest position of string that matches target “c” | string a[4] = { "a", "b", "c", "c" };  assert(locate(a, 4, "c") == 2); |
| Check entire array; return -1 since matching element is not in array at all | string a[4] = { "a", "b", "c", "c" };  assert(locate(a, 4, "d") == -1); |
| Cannot work with negative number of elements in array | string a[4] = { "a", "b", "c", "c" };  assert(locate(a, -1, "d") == -1); |

**int** locationOfMax(**const** string a[], **int** n)

|  |  |
| --- | --- |
| Return smallest position number of biggest item in array | string a[4] = { “a”, “b”, “c”, “c” };  assert(locationOfMax(a, 4) == 2); |
| Return position number of biggest item in array | string a[4] = { “a”, “b”, “c”, “d” };  assert(locationOfMax(a, 4) == 3); |
| Return -1 if we examine no elements in array | string a[4] = { “a”, “b”, “c”, “d” };  assert(locationOfMax(a, 0) == -1); |
| Array has no items; nothing is examined so return -1 | string b[0]={};  assert(locationOfMax(b, 0) == -1); |
| Cannot work with negative number of elements in array | string b[0]={};  assert(locationOfMax(b, -1) == -1); |

**int** circleLeft(string a[], **int** n, **int** pos)

|  |  |
| --- | --- |
| If there are no elements in array, then all positions are invalid inputs so return -1 | string a[4] = { "a", "b", "c", "c" };  assert(circleLeft(a, 0, 3)==-1); |
| Last element number is n-1; position cannot be greater than that | string a[4] = { "a", "b", "c", "c" };  assert(circleLeft(a, 4, 4)==-1); |
| Last element number is n-1; position cannot be greater than that | string a[4] = { "a", "b", "c", "c" };  assert(circleLeft(a, 4, 5)==-1); |
| Smallest number is 0; position cannot be less than that | string a[4] = { "a", "b", "c", "c" |
| Move the element at position 0 to the end | string a[3] = { "a", "b", "c" };  assert(circleLeft(a, 3, 0)==0 && a[0] == "b" && a[1] == "c" && a[2] == "a"); |
| Cannot work with negative number of elements in array | string a[3] = { "a", "b", "c" };  assert(circleLeft(a, -1, 0)==-1); |

**int** enumerateRuns(**const** string a[], **int** n)

|  |  |
| --- | --- |
| 3 sequences in array – return 3 | string a[4] = { "a", "b", "c", "c"};  assert(enumerateRuns(a, 4) == 3); |
| 1 element in the array indicates only 1 sequence exists – return 1 | string a[4] = { "a", "b", "c", "c"};  assert(enumerateRuns(a, 1) == 1); |
| 0 elements in array means no sequences exist – return 0 | string a[4] = { "a", "b", "c", "c"};  assert(enumerateRuns(a, 0) == 0); |
| Cannot work with negative number of elements in array | string a[4] = { "a", "b", "c", "c"};  assert(enumerateRuns(a, -1) == -1); |

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

|  |  |
| --- | --- |
| Reverses 4 out of 4 elements in array | string a[4] = { "a", "b", "c", "c"};  assert(flip(a, 4) == 4 && a[0] == "c" && a[1] == "c" && a[2] == "b" && a[3] == "a"); |
| Reverses 3 out of 4 elements in array | string a[4] = { "a", "b", "c", "c"};  assert(flip(a, 3) == 3 && a[0] == "c" && a[1] == "b" && a[2] == "a" && a[3] == "c"); |
| Cannot work with negative number of elements in array | string a[4] = { "a", "b", "c", "c"};  assert(flip(a, -1) == -1); |
| Array is unchanged since 0 elements are processed | string a[4] = { "a", "b", "c", "c"};  assert(flip(a, 0) == 0 && a[0] == "a" && a[1] == "b" && a[2] == "c" && a[3] == "c"); |

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

|  |  |
| --- | --- |
| First array is larger than second array; difference at element 2; “c” ≠ “e” | string h[4] = { "a", "b", "c", "c"};  string b[3] = { "a", "b", "e"};  assert(locateDifference(h, 4, b, 3) == 2); |
| Cannot work with negative number of elements in first array | string h[4] = { "a", "b", "c", "c"};  string b[3] = { "a", "b", "e"};  assert(locateDifference(h, -1, b, 3) == -1); |
| Cannot work with negative number of elements in second array | string h[4] = { "a", "b", "c", "c"};  string b[3] = { "a", "b", "e"};  assert(locateDifference(h, 1, b, -3) == -1); |
| Of the elements observed, the elements are equal; so return the smaller of n1 and n2 which is 3 | string h[4] = { "a", "b", "c", "c"};  string b[3] = { "a", "b", "c" };  assert(locateDifference(h, 4, b, 3) == 3); |
| Second array is larger than first array; difference at element 3; “c” ≠ “d” | string h[4] = { "a", "b", "c", "c"};  string b[5] = { "a", "b", "c", "d" , "e"};  assert(locateDifference(h, 4, b, 5) == 3); |

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

|  |  |
| --- | --- |
| if a1[] is empty, subsequence is never found in a1[] – return -1 | string b[2] = { "a", "b"};  string c[0] = {};  assert(subsequence(c, 0, b, 2)==-1); |
| No matter the size of a1[], if a2[] has no elements, then a2[] is found at position 0 of a1[] | string b[2] = { "a", "b"};  string c[0] = {};  assert(subsequence(b, 2, c, 0)==0); |
| Cannot work with negative number of elements in first array | string b[2] = { "a", "b"};  string c[0] = {};  assert(subsequence(b, -1, c, 0)==-1); |
| Cannot work with negative number of elements in second array | string b[2] = { "a", "b"};  string c[0] = {};  assert(subsequence(b, 2, c, -1)==-1); |
| Second array is never found in first array because second array is larger than first array | string b[2] = { "a", "b"};  string c[1] = {"A"};  assert(subsequence(c, 1, b, 2)==-1); |
| First appearance of second array is found at position 0 of first array | string b[2] = { "a", "b"};  string c[1] = {"a"};  assert(subsequence(b, 2, c, 1)==0); |

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

|  |  |
| --- | --- |
| Cannot work with negative number of elements in first array | string b[2] = { "a", "b"};  string c[0] = {};  assert(subsequence(b, -1, c, 0)==-1); |
| Cannot work with negative number of elements in second array | string b[2] = { "a", "b"};  string c[0] = {};  assert(subsequence(b, 2, c, -1)==-1); |
| One match is found with element 0 in first array  -- first array is larger than second array | string b[2] = { "a", "b"};  string c[1] = {"a"};  assert(locateAny(b, 2, c, 1) == 0); |
| Two matches found with element 0 and element 2 in first array, but only the smaller of the two elements is returned  -- first array is larger than second array | string b[3] = { "a", "b", "a"};  string c[1] = {"a"};  assert(locateAny(b, 3, c, 1) == 0); |
| If no elements are observed in first array, no matches are found | string b[3] = { "a", "b", "a"};  string c[1] = {"a"};  assert(locateAny(b, 0, c, 1) == -1); |
| If no elements are observed in second array, no matches are found | string b[3] = { "a", "b", "a"};  string c[1] = {"a"};  assert(locateAny(b, 3, c, 0) == -1); |
| One match is found with element 0 in first array  -- second array is larger than first array | string c[1] = {"a"};  string b[3] = { "a", "b", "a"};  assert(locateAny(c, 1, b, 3) == 0); |
| Two matches found with element 0 and element 1 in first array, but only the smaller of the two elements is returned  -- second array is larger than first array | string c[2] = {"d", "d"};  string b[4] = { "a", "b", "c", "d"};  assert(locateAny(c, 2, b, 4) == 0); |

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

|  |  |
| --- | --- |
| Cannot work with negative number of elements in array | string b[4] = { "a", "b", "x", "y"};  assert(separate(b, -1, "c") == -1); |
| If there are no elements in the array, then no elements are not less than separator; return 0, the number of elements observed | string b[4] = { "a", "b", "x", "y"};  assert(separate(b, 0, "c") == 0); |
| element at position 2 is the first element >= separator | string b[4] = { "x", "y", "a", "b"};  assert(separate(b, 4, "c") == 2); |
| Return n = 0 when none of the elements in c[] are >= the separator “b” | string c[1]={"a"};  assert(separate(c, 1, "b") == 1); |
| element at position 0 is the first element equal to the separator, so is returned | string c[1]={"b"};  assert(separate(c, 1, "b") == 0); |