Joshua Liu

105136031

November 17, 2018

CS 31

Description of Obstacles I Overcame

While working on the program, it was hard for me find a place to start. I thought the instructions were a little hard to understand at first and finding an approach to finding the solution was not as obvious as previous projects. Initially, working with C strings was difficult for me because I could not figure out the correct way to deal with and read through a two-dimensional character array. I had to spend a lot of time planning and deciding how to remove a row of incorrect patterns and keeping the remaining ones that were correct. After completing the program, I had a lot of errors while running test cases, so I had to go back and print out all the cases that were producing an error and fix up the algorithms in the code. For example, I forgot to consider negative values for nPatterns, and was able to fix the issue quickly once I knew what was wrong.

Description of the design of my program

The general concept of the my makeProper function

Include all the needed libraries(<iostream>, <cassert>, <cctype>, <cstring>, as well as “using namespace std;”. Before these statements, I defined \_CRT\_SECURE\_NO\_WARNINGS since I am running the code on Visual C++.

We declare the MAX\_WORD\_LENGTH for the length of the words in word1 and word2 as stated in the specs.

int makeProper(char word1[][MAX\_WORD\_LENGTH + 1], char word2[][MAX\_WORD\_LENGTH + 1], int separation[], int nPatterns);

In this function, I first remove all the invalid strings in the character arrays, as well as the negative values in separation. If nPatterns is less than 0, return 0. Then, processed the strings by matching a word in word1 with a word in word2 to see if there are duplicates. The number of patters is kept track of every time a pattern is subtracted from the code. Lastly, I returned the value of nPatterns.

Pseudocode:

*if nPatters is zero or less than zero*

*return 0*

*remove all strings with non-alphabetical characters for word1 and word2(decrease nPatterns)*

*convert all the strings to lowercase in both word1 and word2*

*for word1 and word2, if there is an emtpy string*

*remove that row of elements for word1 and word2(decrease nPatterns)*

*if separation is negative*

*remove that row of elements(decrease nPatterns)*

*if a string in word1 is equal to a string in word1*

*if a string in word2 of the same element is the same as a string in word2 of the corresponding element*

*if the first row has a greater separation, remove the second row(decrease nPatterns)*

*else, remove the first row(decrease nPatterns)*

*if a string in word1 is equal to a string in word2*

*if a string in word1 of the same element is the same as a string in word2 of the corresponding element*

*if the first row has a greater separation, remove the second row(decrease nPatterns)*

*else, remove the first row(decrease nPatterns)*

*if a string in word2 is equal to a string in word2*

*if a string in word1 of the same element is the same as a string in word1 of the corresponding element*

*if the first row has a greater separation, remove the second row(decrease nPatterns)*

*else, remove the first row(decrease nPatterns)*

*if a string in word2 is equal to a string in word1*

*if a string in word2 of the same element is the same as a string in word1 of the corresponding element*

*if the first row has a greater separation, remove the second row(decrease nPatterns)*

*else, remove the first row(decrease nPatterns)*

*return nPatterns*

int rate(const char document[], const char word1[][MAX\_WORD\_LENGTH + 1], const char word2[][MAX\_WORD\_LENGTH + 1], const int separation[], int nPatterns);

In this function, I began by copying the data into another array. Then, I checked the string for nonalphabetical characters. Next, I placed all the words into a character array in order to manipulate the data easier. Cycling through the document, I incremented the integer “matches” that I returned at the end that represents the number of matches found in the document.

Pseudocode:

*declare array size for STORE\_CHARS*

*declare int variable matches to count the number of matches*

*copy document string into an array that can be manipulated*

*remove all nonalphabetical characters in the array*

*copy the same string into another array and add spaces before and after the string in order for the program to process the string*

*cycle through the document, looking for a character that matches the first word in word1*

*if word1 does not match a word in the document, move onto next word in word1*

*if there is a match, create a lower and upper bound to check for word2 that corresponds to word1*

*if lower bound is less than zero, then lower bound is equal to zero*

*if upper bound is greater than the number of words in the document, then upper bounds is equal to this number*

*if a word in word2 matches with the word1 within the separation distance, increment matches and break out of loop to investigate next pattern until reaches end of word1*

*return matches*

The main function consists of testing data for the functions.

Data to Test Program

// Test cases for makeProper

const int TEST1\_NRULES = 8;

char test1w1[TEST1\_NRULES][MAX\_WORD\_LENGTH + 1] = { "mad", "deranged", "NEFARIOUS", "half-witted", "robot", "plot", "have", "NeFaRiOuS" };

char test1w2[TEST1\_NRULES][MAX\_WORD\_LENGTH + 1] = { "scientist", "robot", "PLOT", "assistant", "deranged", "Nefarious", "mad", "pLoT" };

int test1dist[TEST1\_NRULES] = { 1, 2, 3, 4, 5, 6, -12, 7 };

assert(makeProper(test1w1, test1w2, test1dist, TEST1\_NRULES) == 3); // remove rows with negative separation and nonalphabetical words

//Words that appear twice intercahgnably in word1 and word2 are listed only once with the greatest separation

//All upper case letters are changed to lower case letters

const int TEST3\_NRULES = 3;

char test3w1[TEST3\_NRULES][MAX\_WORD\_LENGTH + 1] = { "", "", "" };

char test3w2[TEST3\_NRULES][MAX\_WORD\_LENGTH + 1] = { "hello", " ", " " };

int test3dist[TEST3\_NRULES] = { 1, 2, 3 };

assert(makeProper(test3w1, test3w2, test3dist, TEST3\_NRULES) == 0); // blank char array and spaces return 0 for nPatterns

const int TEST4\_NRULES = 1;

char test4w1[TEST4\_NRULES][MAX\_WORD\_LENGTH + 1] = { };

char test4w2[TEST4\_NRULES][MAX\_WORD\_LENGTH + 1] = { };

int test4dist[TEST4\_NRULES] = { };

assert(makeProper(test4w1, test4w2, test4dist, TEST4\_NRULES) == 0); // blank char array and spaces return 0 for nPatterns

const int TEST5\_NRULES = 1;

char test5w1[TEST5\_NRULES][MAX\_WORD\_LENGTH + 1] = { "chicken" };

char test5w2[TEST5\_NRULES][MAX\_WORD\_LENGTH + 1] = { "turkey" };

int test5dist[TEST5\_NRULES] = { 15 };

assert(makeProper(test5w1, test5w2, test5dist, TEST5\_NRULES) == 1); // one correct row of elements only, return nPatters = 1

const int TEST6\_NRULES = 1;

char test6w1[TEST6\_NRULES][MAX\_WORD\_LENGTH + 1] = { "chicken" };

char test6w2[TEST6\_NRULES][MAX\_WORD\_LENGTH + 1] = { "turkey" };

int test6dist[TEST6\_NRULES] = { -15 };

assert(makeProper(test6w1, test6w2, test6dist, TEST6\_NRULES) == 0); // separation is negative, return 0

assert(makeProper(test6w1, test6w2, test6dist, -1) == 0); // negative nPatterns, return 0

// Test cases for rate

const int TEST2\_NRULES = 4;

char test2w1[TEST2\_NRULES][MAX\_WORD\_LENGTH + 1] = { "mad", "deranged", "nefarious", "have" };

char test2w2[TEST2\_NRULES][MAX\_WORD\_LENGTH + 1] = { "scientist", "robot", "plot", "mad" };

int test2dist[TEST2\_NRULES] = { 1, 3, 0, 12 };

assert(rate("The mad UCLA scientist unleashed a deranged evil giant robot.", test2w1, test2w2, test2dist, TEST2\_NRULES) == 2); // count correct number of words, return 2

assert(rate("The mad UCLA scientist unleashed a deranged robot.", test2w1, test2w2, test2dist, TEST2\_NRULES) == 2); // extra spaces in document string, ignore and return 2

assert(rate("\*\*\*\* 2018 \*\*\*\*", test2w1, test2w2, test2dist, TEST2\_NRULES) == 0); // blank string when nonalphabetical chars are ignored, return 0

assert(rate(" That plot: NEFARIOUS!", test2w1, test2w2, test2dist, TEST2\_NRULES) == 1); // ignore nonalphabetical characters and count matches, return 1

assert(rate("That scientist said two mad scientists suffer from deranged-robot fever.", test2w1, test2w2, test2dist, TEST2\_NRULES) == 0); // no mathces, return 0

const int TEST7\_NRULES = 4;

char test7w1[TEST7\_NRULES][MAX\_WORD\_LENGTH + 1] = { "chicken", "good", "golden", "sauce" };

char test7w2[TEST7\_NRULES][MAX\_WORD\_LENGTH + 1] = { "nuggets", "taste", "crispy", "hot" };

int test7dist[TEST7\_NRULES] = { 4, 1, 3, 2 };

assert(rate(" \*The9 chi5cken nu\*ggets taste so sooo good# with sauce w%hen they a#re hot.", test7w1, test7w2, test7dist, TEST7\_NRULES) == 1); // ignore nonalphabetical chars, return 1

assert(rate(" \*The9 nu\*ggets chi5cken good# so sooo taste with sauce w%hen they a#re hot.", test7w1, test7w2, test7dist, TEST7\_NRULES) == 1); // swap words from previous case, return 1

assert(rate("chicken golden crispy golden green nuggets ", test7w1, test7w2, test7dist, TEST7\_NRULES) == 2); // only count matches once, return 2

assert(rate(" [purple chicken golde%n c3rispy go&&lden gre\*e()n nug!@#$gets ", test7w1, test7w2, test7dist, TEST7\_NRULES) == 2); // only count matches once, random spaces beofre and after, return 2

assert(rate(" [purple chicken golde%n c3rispy go&&lden nug!@#$gets ", test7w1, test7w2, test7dist, TEST7\_NRULES) == 2); // make sure program does not check out of bounds, return 2

assert(rate(" [purple nug!@#$gets golde%n c3rispy go&&lden gre\*e()n chicken ", test7w1, test7w2, test7dist, TEST7\_NRULES) == 2); // same as previous test but swapped, ensure it works as well, return 2

assert(rate(" \*\*\*7&&&883285%$^#74567%\*%$&%", test7w1, test7w2, test7dist, TEST7\_NRULES) == 0); // empty string after nonalphabetical characters and spaces are considered, return 0

assert(rate(" good ", test7w1, test7w2, test7dist, TEST7\_NRULES) == 0); // one word that has is a word in word1 but not word2, return 0

assert(rate("chicken golden blue crispy golden green nuggets ", test7w1, test7w2, test7dist, TEST7\_NRULES) == 1); // ensure separation works at beginning of string wihtout accessing invalid spots, return 1

assert(rate("nuggets golden blue crispy golden green chicken", test7w1, test7w2, test7dist, TEST7\_NRULES) == 1); // ensure separation works at end of string without accessing invalid sports, return 1