Report

1. In the case of functions that involved rearranged elements (e.g. flip, split), I initially wanted to create new arrays to copy the rearranged elements of the original array. I found it was it was easeir to directly assign elements than create a whole new array. For the split function, I had difficulty figuring out to move the elements equal to split. I realized if I just moved the elements for one of the case (such as moving to the front), then I would just have to make the splitter elements after those. The elements greater than the splitterwould still end up at the end of the array.
2. Here is my list of test data per function.
   * int appendToAll(string a[], int n, string value);

a[5] = { "hillary", "jeb", "rand", "ben", "john" }

(a, 5, "!!!") normal case, should work.

(a, 0, “yes”) testing if defining 0 elements, should not change original array.

(a, -1, “yes”) (a, -5, “yes”) testing if n is negative, should return -1.

(a, 5, “”) testing if empty string value, should not change original array.

(a, 5, “89430”) (a, 5, “4”) testing if string value accepts numbers, should append each number to each element.

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

a[5] = { "hillary", "jeb", "rand", "ben", "no" }

b[5] = { "hillary", "", "4", "ben", "" }

(a, 5, “no”) testing for normal case, should work.

(a, 2, “no”) testing within example array, should not find target.

(a, 0, “yes”) testing if defining 0 elements, should not find target.

(a, -1, “yes”) (a, -5, “yes”) testing if defining negative elements, should return -1 and not find target.

(b, 3, “”) testing if will find empty string, should return found index or n if not found.

(a, 3, “89430”) (a, 3, “4”) testing if will find numbers, should return found index or n if not found.

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

a[6] = { "bernie", "hillary", "donald", "marco", "carly", "ben" }

b[6] = { "bernie", "", "donald", "marco", "carly", "ben" }

c[6] = { "bernie", "hillary", "donald", "marco", "ben", "ben" }

d[6] = { "bernie", "hillary", "marco", "marco", "carly", "ben" }

(a, 6) normal case, should return correct max position

(a, 5) testing elements within the array, should return correct max position.

(a, 0) testing if no elements in array, should return -1.

(a, -1) (a, -35) testing if negative definition of elements in array, should return -1.

(b, 6) testing when empty element, should return correct max positon.

(c, 6) testing if multiple same elements, should return correct max position.

(d, 6) testing if max appears multiple times, should return first max position.

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

//example array should have an empty string

a[5] = { "carly", "mike", "", "bernie", "jeb" }

(a, 5, 1) normal case, should displace first element to the end.

(a, 3, 1) checking within example array, should displace first element to the end of this segment of the array.

(a, 2, 1) checking if can properly use empty elements, should displace first element ot eh end.

(a, 0, 1) (a, 2, 4) checking if position is greater than n, should return -1.

(a, -1, 1) (a, 5 -1) checking if negative integer accepted, should return -1.

(a, 5, 4) checking end bound of n, should displace first element to the end.

(a, 5, 0) checking beginning bound of n, should displace first element to end.

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

d[9] = {"ben", "chris", "marco", "marco", "donald", "donald", "donald", "marco","marco"}

a[4] = {“a”, “b”, “c”, “d”,}

b[4] = {“a”, “a”, “a”, “a”,}

c[5] = {“a”, “a”, “c”, “a”, “a”}

e[4] = {“a”, “b”, “”, “d”,}

(d, 9) normal case, should find correct number of consecutive sequences.

(d, 8) checking within example array, should find correct number of consecutive sequences.

(d, 0) checking if no elements defined in array, should return 0.

(d, -1) (a, -343) checking if defining negative elements in array, should return -1.

(a, 4) checking if array with consecutive unique elements, should return number of elements in array.

(b, 4) checking if array with same consecutive element, should return 1.

(c, 5) checking if array with same element spaced out, should return correct number of consecutive sequences.

(e, 4) checking if array with an empty string, should return correct number of consecutive sequences.

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

f [6] = { "chris", "marco", "", "ben", "donald", "john" }

a [6] = { "chris", "marco", "", "ben", "donald", "" }

b [6] = { "", "marco", "", "ben", "donald", "john" }

(f, 6) normal case, should return n and flips array elements properly.

(f, 4) (f, 3) checking if within example array, should return n and flips array elements properly.

(f, 0) checking if no elements, should return n and do nothing to array.

(f, -1) (f, -34) checking if negative elements, should return -1 and do nothing to the array.

(a, 6) checking if array has empty string at end, should return n and flip array elements properly.

(b, 4) checking if array has empty string at beginning, should return n and flip array elements properly.

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

f[6] = { "chris", "marco", "", "ben", "donald", "john" };

g[5] = { "chris", "marco", "john", "", "carly" };

(f, 6, g, 5) normal case, should return correct differ positon.

(f, 2, g, 1) checking within example array, should return correct differ positon.

(f, 0, g, 1) (f, 2, g, 0) checking if no elements in either array, should return correct differ positon.

(f, -5, g, 1) (f, 2, g, -4) checking if negative elemetns in either array, should return correct differ positon.

(f, 4, g, 5) checking if n1 < n2, should return correct differ positon.

(f, 6, f, 6) checking if both arrays have same elements, should return smaller of n1 and n2.

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

a[10] = { "ted", "hillary", "rand", "bernie", "mike", "jeb" };

b[10] = { "hillary", "rand", "bernie" };

c[10] = { "ted", "hillary", "hillary", "bernie", "mike", "jeb" };

d[10] = { "hillary", "rand", "bernie", "mike", "jeb" };

e[10] = { "ted", "hillary", "rand", "bernie"};

f[10] = { "ted", "hillary", "rand", "mike", "jeb" };

(a, 6, b, 3) normal case, should return correct starting position of match in first array.

(a, 5, b, 2) checking within example array, should return correct starting position of match in first array or -1 if not found.

(a, 0, b, 0) (a, 2, b, 0) checking if 0 elements within subsequence, should return -1.

(a, 0, b, 3) (a, 2, b, 4) checking if n2 > n1, should return -1.

(a, -3, b, 3) (a, 6, b, -3) checking if negative definitions, should return -1.

(a, 6, c, 3) checking if consecutive elements of the first position of first array, should return correct starting position of match in first array or -1 if not found.

(d, 6, b, 3) checking if a2 sequence is in the very beginning of a1, should return correct starting position of match in first array or -1 if not found.

(e, 4, b, 4) checking if a2 seuqence is at the very end of a1, should return correct starting position of match in first array or -1 if not found.

(f, 5, b, 3) checking if a2 sequence is almost in a1, should return correct starting position of match in first array or -1 if not found.

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

n[10] = { "ted", "hillary", "rand", "bernie", "mike", "jeb" };

set1[10] = { "carly", "mike", "bernie", "hillary" };

b[10] = { "ted", "mike", "bernie", "hillary" };

c[10] = { "yo”, "si", "kl", "jeb" };

d[10] = { "yo”, "yo", "kl", "jeb" };

e[10] = { "yo”, "kl", "kl", "jeb" };

(n, 6, s, 4) (n, 6, s, 2) normal case, should return found a1 position or 0 if not found.

(n, 5, s, 3) (n, 4, s, 1) checking within arrays, should return found a1 position or -1 if not found.

(n, 0, s, 1) (n, 1, s, 3) checking if n1 < n2, should return -1.

(n, 6, s, 0) checking if no defined elements in second array, should return -1.

(n, -3, s, 1) (n, 6, s, -5) checking if negative elements in either arrays, should return -1.

(n, 6, b, 4) checking if first matching element is first element in both arrays, should return correct position in a1.

(n, 6, c, 4) checking if last matching element is last element in both arrays, should return correct position in a1.

(d, 4, e, 4) checking if element is seen multiple times in either array, should return correct positon in a1.

* + int split(string a[], int n, string splitter);

c[6] = { "bernie", "hillary", "donald", "marco", "carly", "ben" }

a[4] = { "donald", "hillary", "jeb", "ben" }

b[4] = {“a”, “b”, “c”, “d”]

d[4] = { “d”, “a”, “c”, “b”]

e[3] = {“”, “a”, “b”]

f[3] = (“a”, “b”, “”]

(c, 6, "chris")(a, 4, "donald") normal case, should return the position of the first element that, after the rearrangement, is not < splitter.

(c, 5, “chris”) (a, 4, “donald”) checking within the array, (c, 6, "chris")(a, 4, "donald") normal case, should return the position of the first element that, after the rearrangement, is not < splitter, or n if there are no such elements

(c, 0, “a”) checking if no elmeents in array, should return n and not change array.

(c, -5, “a”) checking if negative elements in array, should return -1.

(b, 4, d) checking if split element is at end, should return the position of the first element that, after the rearrangement, is not < splitter, and change array accordingly.

(b, 4, a) checking if split element is at beginning, should return the position of the first element that, after the rearrangement, is not < splitter, and change array accordingly.

(d, 4, a) checking if greater than elements are at beginning or less than elements are at the end, should return the position of the first element that, after the rearrangement, is not < splitter, and change array accordingly.

(b, 4, a) checking if less than elements are at the beginning

or if greater than elements are at the end or if all in order, should return the position of the first element that, after the rearrangement, is not < splitter, and change array accordingly.

(e, 3, a) checking if empty element at the beginning of the array, should return the position of the first element that, after the rearrangement, is not < splitter, and change array accordingly.

(f, 3, a) checking if empty element is at the end of the array, should return the position of the first element that, after the rearrangement, is not < splitter, and change array accordingly.

(f, 3, 2) checking if array element is equal to the splitter, should return the position of the first element that, after the rearrangement, is not < splitter, and change array accordingly.