Nathan Tung
Com Sci 31 (Lec 2)
October 31, 2011

# CS31 Project 4 Report

a. For the most part, this project was straightforward. However, there were too many opportunities to cause off-by-one errors! I often had to work through each function several times in order to prevent any incorrect integers from being returned. This was particularly complex because the size of the array is not equal to the last index number in that array. Therefore, each time a variable n is given to represent the array size, I had to make sure the index integer returned wasn't directly based on the array size (which would provide an off-by-one error).

   I also nearly screwed up my rotateRight function because I overlooked something simple. I had copied my rotateLeft function since the same idea and mechanism is used. But instead of starting from the front of the array, my rotateRight function starts at the end. Unfortunately, I had forgotten to decrement the index variable rather than increment it, hence creating an array out-of-bounds error. A simple change to k-- fixed this problem.

   Lastly, for coding the split function, I had to resort to creating a temporary array with a size of 9999 in order to arrange and store each string (based on the splitter string). After the required n strings are arranged, they are copied back from the temporary array into the original one. Though this might present errors for arrays with millions of strings, it should work correctly in most other cases.

b. The finished program is able to handle all my test cases listed below. However, prior to completion, the only test cases that did not pass were those for the function rotateRight. None of the test cases passed; rather, I essentially received an out-of-bounds error. I had forgotten to change to k++ to k--, thus causing the first index increment to look beyond the array. I have written a function to test each required function specified in the specs; the test cases are listed below:

**appendToAll:**

```
	string blanksAndMore[] = {"", "asdf", "!"};
	string blanksAndMoreAns[] = {"!!!!!", "asdf!!!!!", "!!!!!!"};
	assert(appendToAll(blanksAndMore, 3, "!!!!!")==3);
	for(int k=0; k<3; k++)
	{
		assert(blanksAndMore[k]==blanksAndMoreAns[k]); //Make sure different strings are all concatenated correctly
	}

	string bounds[] = {"a", "b", "c", "d"};
	string boundsAns[] = {"a123z", "b123z", "c", "d"};
	assert(appendToAll(bounds, 2, "123z")==2);
	for(int k=0; k<4; k++)
	{
		assert(bounds[k]==boundsAns[k]); //Make sure only the first n elements are appended with the specified string
	}
```

```
	string hiThere[] = {""};
	assert(appendToAll(hiThere, -5, "asdf")==-1); //Illegal array size testing!
```

**lookup:**

```
	string greetings[] = {"hi", "hello", "bonjour", "great to see you", "hello"};
	assert(lookup(greetings, 5, "hello there")==-1); //Check to see if -1 is returned for a string not found
	assert(lookup(greetings, 5, "Hello")==-1); //Check to verify that the function is case-sensitive
	assert(lookup(greetings, 5, "hello")==1); //Make sure the first index is given if more than one instance is found
	assert(lookup(greetings, 0, "boo!")==-1); //See that -1 is returned if there is no array (hence no strings)
	assert(lookup(greetings, 2, "bonjour")==-1); //Make sure only the first n elements are checked
	assert(lookup(greetings, -1, "bonjour")==-1); //Illegal array size testing!
```

**positionOfMax:**

```
	string inOrder[] = {"a", "b", "c", "d", "e"};
	assert(positionOfMax(inOrder, 5)==4); //Test to see if the index of "e" is returned, since it's latest in alphabet
	assert(positionOfMax(inOrder, 4)==3); //Test to see if the function is limited to the first n elements

	string backwardsOrder[] = {"e", "d", "c", "b", "a"};;
	assert(positionOfMax(backwardsOrder, 5)==0); //Test to see if the index of "e" is returned, since it's latest in alphabet
	assert(positionOfMax(backwardsOrder, 2)==0); //Test to see if bounds will throw the program off

	string randomOrder[] = {"great", "cool", "zillion", "ton", "massive", "zillion", "billion"};
	assert(positionOfMax(randomOrder, 2)==0); //See if only the first n elements are compared
	assert(positionOfMax(randomOrder, 7)==2); //See if only the index of the first max string is returned
	assert(positionOfMax(randomOrder, -2)==-1); //Illegal array size testing!
```

**rotateLeft:**

```
	string alpha[] = {"a", "b", "c", "d", "e", "f"};
	string alphaRotatedLeftOnA[] = {"b", "c", "d", "e", "f", "a"};
	string alphaRotatedLeftOnC[] = {"a", "b", "d", "e", "f", "c"};
	string alphaRotatedLeftOnF[] = {"a", "b", "c", "d", "e", "f"};
	string alphaRotatedLeftOnCFour[] = {"a", "b", "d", "c", "e", "f"};

	assert(rotateLeft(alpha, 6, 0)==0); //test if rotateLeft is performed
	for(int k=0; k<6; k++)
	{
		assert(alpha[k]==alphaRotatedLeftOnA[k]); //See if all elements of the new array are correct as predicted (first string targetted)
	}

	string alpha2[] = {"a", "b", "c", "d", "e", "f"};
	assert(rotateLeft(alpha2, 6, 2)==2);
	for(int k=0; k<6; k++)
	{
		assert(alpha2[k]==alphaRotatedLeftOnC[k]); //See if all elements of the new array are correct as predicted (string "c" targetted)
	}

	string alpha3[] = {"a", "b", "c", "d", "e", "f"};
	assert(rotateLeft(alpha3, 6, 5)==5);
```

```
	for(int k=0; k<6; k++)
	{
		assert(alpha3[k]==alphaRotatedLeftOnF[k]); //See if all elements of the new array are
correct as predicted (last string targetted)
	}

	string alpha4[] = {"a", "b", "c", "d", "e", "f"};
	assert(rotateLeft(alpha4, 4, 2)==2); //test if rotateLeft is performed
	for(int k=0; k<6; k++)
	{
		assert(alpha4[k]==alphaRotatedLeftOnCFour[k]); //See if only the first n elements (4 in
this case) are considered by the function
	}

	string blank[] = {""};
	assert(rotateLeft(blank, 0, 0)==-1); //test if a -1 is given when n is an array of zero size
	assert(rotateLeft(blank, -5, 0)==-1); //test if a -1 is given when n is an array of negative
size
	assert(rotateLeft(blank, 1, 0)==0); //test if rotateLeft is performed
	assert(blank[0]=="");

	string blankAlpha[] = {"a", "", "b", "c"};
	string blankAlphaAnswer[] = {"a", "b", "c", ""};
	assert(rotateLeft(blankAlpha, 0, 0)==-1); //test if a -1 is given when n is an array of zero
size
	assert(rotateLeft(blankAlpha, -5, 0)==-1); //test if a -1 is given when n is an array of
negative size
	assert(rotateLeft(blankAlpha, 4, 1)==1); //test if rotateLeft is performed

	for (int k=0; k<4; k++)
	{
		assert(blankAlpha[k]==blankAlphaAnswer[k]); //check if all the elements are rotated as
expected
	}
```

**rotateRight:**

```
	string alpha[] = {"a", "b", "c", "d", "e", "f"};
	string alphaRotatedRightOnA[] = {"a", "b", "c", "d", "e", "f"};
	string alphaRotatedRightOnC[] = {"c", "a", "b", "d", "e", "f"};
	string alphaRotatedRightOnF[] = {"f", "a", "b", "c", "d", "e"};

	assert(rotateRight(alpha, 6, 0)==0); //test if rotateRight is performed
	for(int k=0; k<6; k++)
	{
		assert(alpha[k]==alphaRotatedRightOnA[k]); //See if all elements of the new array are
correct as predicted (first string targetted)
	}

	string alpha2[] = {"a", "b", "c", "d", "e", "f"};
	assert(rotateRight(alpha2, 6, 2)==2);
	for(int k=0; k<6; k++)
	{
		assert(alpha2[k]==alphaRotatedRightOnC[k]); //See if all elements of the new array are
correct as predicted (string "c" targetted)
	}

	string alpha3[] = {"a", "b", "c", "d", "e", "f"};
	assert(rotateRight(alpha3, 6, 5)==5);
	for(int k=0; k<6; k++)
```

```
	{
		assert(alpha3[k]==alphaRotatedRightOnF[k]); //See if all elements of the new array are
correct as predicted (last string targetted)
	}

	string alpha4[] = {"a", "b", "c", "d", "e", "f"};
	assert(rotateRight(alpha4, 4, 2)==2); //test if rotateRight is performed
	for(int k=0; k<6; k++)
	{
		assert(alpha4[k]==alphaRotatedRightOnC[k]); //See if only the first n elements (4 in this
case) are considered by the function
	}

	string blank[] = {""};
	assert(rotateRight(blank, 0, 0)==-1); //test if a -1 is given when n is an array of zero size
	assert(rotateRight(blank, -5, 0)==-1); //test if a -1 is given when n is an array of negative
size
	assert(rotateRight(blank, 1, 0)==0); //test if rotateRight is performed
	assert(blank[0]==""); //make sure array did not change

	string blankAlpha[] = {"a", "", "b", "c"};
	string blankAlphaAnswer[] = {"", "a", "b", "c"};
	assert(rotateRight(blankAlpha, 0, 0)==-1); //test if a -1 is given when n is an array of zero
size
	assert(rotateRight(blankAlpha, -5, 0)==-1); //test if a -1 is given when n is an array of
negative size
	assert(rotateRight(blankAlpha, 4, 1)==1);  //test if rotateRight is performed

	for (int k=0; k<4; k++)
	{
		assert(blankAlpha[k]==blankAlphaAnswer[k]); //check if all the elements are rotated as
expected
	}
```

**flip:**

```
	string blank[] = {""};
	assert(flip(blank, 0)==0); //nothing should happen to a blank array
	assert(flip(blank, -5)==-1); //check to make sure it catches impossible array sizes
	assert(flip(blank, 1)==1); //check to see if flip was performed
	assert(blank[0]==""); //check to make sure the only string element is "flipped" with itself

	string alpha[] = {"a", "b", "c", "d"};
	string alphaFlipped[] = {"d", "c", "b", "a"};
	assert(flip(alpha, 4)==4); //check to see if flip was performed

	for(int k=0; k<4; k++)
	{
		assert(alpha[k]==alphaFlipped[k]); //verify that all elements have been flipped
	}

	string random[] = {"apple", "cookie", "", "pie", "123"};
	string randomFlipped[] = {"123", "pie", "", "cookie", "apple"};
	assert(flip(random, 5)==5); //check to see if flip was performed

	for(int k=0; k<5; k++)
	{
		assert(random[k]==randomFlipped[k]); //verify that more-complicated elements can be
flipped
	}
```

```
	string subRandom[] = {"apple", "cookie", "", "pie", "123"};
	string subRandomFlipped[] = {"", "cookie", "apple", "pie", "123"};
	assert(flip(subRandom, 3)==3); //check to see if flip was performed

	for(int k=0; k<5; k++)
	{
		assert(subRandom[k]==subRandomFlipped[k]); //verify that flip works even when not all element indexes are targetted
	}
```

**differ:**

```
	string blank[] = {""};
	assert(differ(blank, 1, blank, 1)==1); //since array is entirely equal to itself, the "smaller" n should be given: 1

	string test1[] = {"hello", "no thanks", "nice to meet you"};
	string test2[] = {"hello", "NO THANKS", "nice to meet you"};

	assert(differ(test1, 1, test2, 1)==1); //since the sub-array is entirely equal, the "smaller" n should be given: 1
	assert(differ(test1, 2, test2, 1)==1); //verifies that an irrelevant change in size makes no difference
	assert(differ(test1, 1, test2, 2)==1); //verifies that an irrelevant change in size makes no difference
	assert(differ(test1, 2, test2, 2)==1); //since a difference has been found, the index 1 where it occurs is returned
	assert(differ(test1, 3, test2, 3)==1); //since a difference is found halfway in, the rest of the array doesn't matter

	string stuff1[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff2[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff3[] = {"animals", "bagels", "camels", "dolphins", "nothing"};
	assert(differ(stuff1, 1, stuff2, 1)==1); //since array is entirely equal to itself, the "smaller" n should be given
	assert(differ(stuff1, 3, stuff2, 1)==1); //verifies that an irrelevant change in size makes no difference
	assert(differ(stuff1, 5, stuff2, 5)==5); //since the sub-array is entirely equal, the "smaller" n should be given: 5
	assert(differ(stuff1, 3, stuff2, 5)==3); //since the sub-array is entirely equal, the "smaller" n should be given: 3

	assert(differ(stuff1, 3, stuff3, 5)==3); //one array has run out, so its n is returned
	assert(differ(stuff1, 5, stuff3, 5)==4); //the fifth element is different
	assert(differ(stuff1, 3, stuff3, 3)==3); //since we do not reach the last element, we are told the arrays match
```

**subsequence:**

```
	string blank[] = {""};
	assert(subsequence(blank, 1, blank, 1)==0); //since array is entirely equal to itself, it starts matching at index 0

	string test1[] = {"hello", "no thanks", "nice to meet you"};
	string test2[] = {"hello", "NO THANKS", "nice to meet you"};
	assert(subsequence(test1, 3, test2, 3)==-1); //although the fronts match up, test2 is not entirely a subsequence of test1
	assert(subsequence(test1, 1, test2, 1)==0); //since "hello" is at the first index of the first array, it's always 0
```

```
	assert(subsequence(test1, 2, test2, 1)==0); //since "hello" is at the first index of the first array, it's always 0
	assert(subsequence(test1, 1, test2, 2)==-1); //test2 is larger than test1, so subsequence is not possible

	string stuff1[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff2[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff3[] = {"animals", "bagels", "camels", "dolphins", "nothing"};
	assert(subsequence(stuff1, 1, stuff2, 1)==0); //since "animals" is at the first index of the first array, it's always 0
	assert(subsequence(stuff1, 5, stuff2, 2)==0); //since "animals" and "bagels" is at the first index of the first array, it's always 0
	assert(subsequence(stuff1, 5, stuff3, 5)==-1); //the element "nothing" isn't in stuff1
	assert(subsequence(stuff1, 3, stuff3, 3)==0); //since we don't see the part where the array differs, 0 is returned

	string sub1[] = {"here", "there", "nowhere", "somewhere", "anywhere"};
	string sub2[] = {"there", "nowhere", "somewhere", "anywhere"};
	assert(subsequence(sub1, 5, sub2, 4)==1); //sub2 is found in sub1 starting at index 1
	assert(subsequence(sub1, 4, sub2, 4)==-1); //sub2 is found in sub1, but we can't see the part where they completely coincide
	assert(subsequence(sub1, 4, sub2, 3)==1); //see if sub2 can shift correctly and be found in sub1 at index 1
	assert(subsequence(sub1, 2, sub2, 1)==1); //see if sub2 can shift correctly and be found in sub1 at index 1
```

**lookupAny:**

```
	string blank[] = {""};
	assert(lookupAny(blank, 1, blank, 1)==0); //since array is entirely equal to itself, the first match is at 0

	string test1[] = {"hello", "no thanks", "nice to meet you"};
	string test2[] = {"hello", "NO THANKS", "nice to meet you"};
	assert(lookupAny(test1, 1, test2, 1)==0); //since the first one matches, index 0 is always returned
	assert(lookupAny(test1, 3, test2, 3)==0); //since the first one matches, index 0 is always returned

	string test3[] = {"NO THANKS", "nice to meet you"};
	string test4[] = {"hello", "nice to meet you"};
	assert(lookupAny(test3, 2, test4, 2)==1); //see if two matching strings at the end can be found
	assert(lookupAny(test3, 1, test4, 1)==-1); //see if different single-string arrays return -1
	assert(lookupAny(test3, 2, test4, 1)==-1); //see if two arrays without any common strings return -1
	assert(lookupAny(test3, 1, test4, 2)==-1); //see if two arrays without any common strings return -1
	assert(lookupAny(test1, 3, test3, 1)==-1); //verify that only first n elements are checked, and function is case sensitive
	assert(lookupAny(test1, 3, test3, 2)==2); //test that matches can be found without going out-of-bounds

	string stuff1[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff2[] = {"earwax", "camels"};
	string stuff3[] = {"animals", "bagels", "camels", "dolphins", "nothing"};
	assert(lookupAny(stuff1, 4, stuff2, 1)==-1); //see if only the first n elements are considered
	assert(lookupAny(stuff1, 5, stuff2, 1)==4); //see if matches can be found at the end without bounds problems
	assert(lookupAny(stuff1, 5, stuff2, 2)==2); //see if matches with lower indexes are returned first
```

```
	assert(lookupAny(stuff1, 3, stuff2, 2)==2); //see if only the first n are considered

	string sub1[] = {"here", "there", "nowhere", "somewhere", "anywhere"};
	string sub2[] = {"there", "nowhere", "somewhere", "anywhere"};
	assert(lookupAny(sub1, 5, sub2, 4)==1); //check that only the first match at earliest index is
returned
	assert(lookupAny(sub1, 5, sub2, 1)==1); //check that only the first match at earliest index is
returned
```

**split:**

```
	string stuffAns[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff1[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff2[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff3[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff4[] = {"animals", "bagels", "camels", "dolphins", "earwax"};
	string stuff5[] = {"animals", "bagels", "camels", "dolphins", "earwax"};

	assert(split(stuff1, 5, "camels")==2); //test if a sorted array (target in the middle) returns
the right index
	assert(split(stuff2, 5, "animals")==0); //test if a sorted array (target in the front) returns
the right index
	assert(split(stuff3, 5, "az")==1); //test if a sorted array (target nonexistent but at index 1)
returns the right index
	assert(split(stuff4, 5, "ear")==4); //test if a sorted array (target one before the end) returns
the right index
	assert(split(stuff5, 5, "ez")==5); //test if n is returned if all strings are less than "ez"

	for(int k=0; k<5; k++) //check that no arrays are changed, since they were already sorted
	{
		assert(stuff1[k]==stuffAns[k]);
		assert(stuff2[k]==stuffAns[k]);
		assert(stuff3[k]==stuffAns[k]);
		assert(stuff4[k]==stuffAns[k]);
		assert(stuff5[k]==stuffAns[k]);
	}


	string stuffAns6[] = {"c", "b", "a", "q", "d", "z"};
	string stuffAns7[] = {"c", "d", "q", "b", "a", "z"};
	string stuff6[] = {"c", "q", "d", "b", "a", "z"};
	string stuff7[] = {"c", "q", "d", "b", "a", "z"};

	assert(split(stuff6, 6, "ce")==3); //see if correct position is returned in an unsorted array

	for(int k=0; k<5; k++)
	{
		assert(stuff6[k]==stuffAns6[k]); //see if the array is sorted as expected
	}

	assert(split(stuff7, 3, "darnit")==2); //see if correct position is returned in an unsorted
array

	for(int k=0; k<5; k++)
	{
		assert(stuff7[k]==stuffAns7[k]); //see if the array is sorted as expected
	}
```