**CS Report**

**Notable problems in creating the program include:**

**The incorrect implementation of for-loop bounds:** With many of the functions being products of comparisons, it became a serious problem when I got my bounds wrong. Loops would either end too soon, never start, or become an infinite loop. It took drawing out the problem to solve this issue. It was important to not create undefined behaviors that were impossible to implement. For the most part, using n+1/n-1 distinctions instead of “ < n” in the beginnings of for-loops fixed these problems.

**Creating comparisons:**

I had a hard time figuring out how to compare parts of an array without making several different loops and arrays. At one point, it became a question of whether a for or a while loop would solve the problem. Most of the time I resorted to for-loops because they seemed simpler to implement.

**Overcomplicated loops:** While many of my functions worked, many of my loops became very complicated, but I had to simplify them for my understanding and for the graders understanding; it also helped in that some of the strategies had to be carried over to other functions, so easy comprehension made that part easier.

**Logically handling the 0 case:**

Because 0 was not treated as undefined behavior, I had to look at every function and determine what the 0 value would return in each individual case. It required knowing the rules of arrays, but I think I was able to sort out the problem (you can’t have positions to look at in an array with size 0, etc.).

**Possible data to put in:**

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

**The 0 case:** to make sure that it doesn’t find any values as opposed to undefined behavior. Should return 0.

**Upper vs. Lower case**: Make sure it doesn’t treat strings with upper and lower case the same (“Barry” and “barry” should not be the same).

**Target that is not in array:** Shows that it returns 0 as opposed to a -1 or another miscellaneous return value.

**N < 0:** Should return -1, checks to see that your program cannot deal with a negative array.

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

**The 0 case:** Show that it cannot be located, returns -1.

**Upper vs lower case:** Checks to see that the program does not treat these certain strings as identical (“Barry” vs. “barry”)

**N < 0:** Should return -1, checks to see that your program cannot deal with a negative array

**Target not in the array:** Checks that it returns -1 because it could not be found.

**bool locateSequence(const string a[], int n, string target, int& begin, int& end);**

**Testing a string that appears only once:** tests to check that the program recognizes this as a sequence of 1, b and e the same

**N < 0:** Should return false, checks to see that your program cannot deal with a negative array.

**Target not in the array:** Checks that the function does not work (returns false) and does not make any changes to the beginning and end points being referenced.

**Upper vs lower case:** Checks to see that the program does not treat these certain strings as identical (“Barry” vs. “barry”).

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

**Backwards order:** Makes sure the min is not being looked at in terms of an alphabetical order.

**Random order:** Similar to above, but shows that it is even more independent of the order of an array.

**Putting min in last and first position:** Tests that the loops for checking do not end too early/late, risking a wrong answer or undefined behavior.

**N < 0:** Should return -1, checks to see that your program cannot deal with a negative array.

**Upper vs lower case:** Checks to see that the program treats ‘B’ as less than ‘b’.

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

**Pos >= n, < 0:** Checks to make sure that the pos being looked at is within the array.

**N < 0:** Should return -1, checks to see that your program cannot deal with a negative array.

**The end value:** Checks to see if the array changes while the last value of the array is already set.

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

**Pos >= n, < 0:** Checks to make sure that the pos being looked at is within the array.

**N < 0:** Should return -1, checks to see that your program cannot deal with a negative array

**The beginning value:** Checks to see if the array changes while the first value of the array is already set.

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

**One string ending before another:** Reviews that the difference comes one position after the end of one string (shorter one).

**Difference of the same size:** Shows that the difference comes one position after the end of both strings.

**n1, n2 < 0:** Should return -1, checks to see that your program cannot deal with a negative array.

**int eliminateDups(string a[], int n);**

**Multiple consecutive strings of the same string:** Checks to see that the program deletes all of the duplicates **AFTER** the first occurrences in their consecutive string, as opposed to the next instance of it (i.e: “a”,”a”, “b”,”b”, “a”, “a” should only delete the ‘a’ twice).

**N < 0:** Should return -1, checks to see that your program cannot deal with a negative array.

**Upper vs lower case:** Checks to see that the program does not treat these certain strings as identical and doesn’t delete one or the other as a duplicate of one another. (if “b”,”B”, “B” should not be deleted).

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

**n2 > n1:** Checks that the function does not consider a subsequence that is bigger than the sequence.

**The 0 case:** Should return false because it is not an array with empty strings, but an array with size 0.

**Upper vs. lower case letters:** Checks that the computer does not see upper and lower case as the same in a sequence.

**n1, n2 < 0:** Should return false, checks to see that your program cannot deal with a negative array.

**Consecutive order vs. In order but not consecutively:** Checks that the function still runs successfully if not in consecutive order.

**int makeMerger(const string a1[], int n1, const string a2[], int n2, string result[], int max);**

**Two strings that are out of alphabetical order:** Should return -1. Makes sure that the only strings considered are those already in order.

**Array with all identical strings:** Tests that it nondecreasing order is being validated (only terminates if it is > than the next, not >=).

**n1, n2 < 0:** Should return -1, checks to see that your program cannot deal with a negative array.

**N1 + n2 <=, > max:** Checks to see that the program does not consider this function with a size larger than the one provided.

**Overlapping identical strings from both arrays:** Makes sure that duplicates are not deleted when making the result array.

**int divide(string a[], int n, string divider);**

**Including cases that are equal to the divider:** Makes sure that these strings are always in the middle, as they are the divider/everything should be less than or greater than it.

**N < 0:** Should return -1, checks to see that your program cannot deal with a negative array.

**Upper and lower case:** Checks that strings comparing between “b” and “B” are separately compared to the divider, not identical.

**General statements about data cases:**

1. **Upper vs. lower case letters:**

Tests if the program can determine if the greater than/less than commands are case-sensitive.

1. **Numbers/other characters in strings:**

Tests to see if the programs obey order rules with characters outside the obvious ones (upper and lower case).

1. **0 case:**

Makes sure that the 0 is treated as input and not undefined behavior, returns the right results for the given circumstances.

1. **N < 0**

Should return false/-1; you can’t have a negative array because you can’t check for those cases; can use this to avoid undefined behavior.

1. **Positions >= n:**

This is to check that a position given doesn’t go outside the positions in the array. For example, if the array size is 6 and the position to be looked at is 6, the function would work (goes 1-5).

1. **Numbers whose n1+n2 > max:**

Should also return false/-1, reflects that an array cannot contain more numbers than it is intended to hold.

1. **Targeting the first positions of strings:**

Makes sure that these cases do not skip over the first position in a string; points out any looping problems that occurred through wrong starting points.

1. **Targeting the last positions of strings:**

Reveals if there is a problem with the endpoint of a loop. Sorts out a common problem that occur if the for loop compares two different parts of an array (a vs. a+1, etc.). Can help patch problems with leaving a loop too early.

1. **Utilizing strings that are in alphabetical order**

Makes sure that while coding, I didn’t have a tendency to make functions based on their orientation; for some of the functions, an alphabetically ordered array and a randomized array should yield the same results.

1. **Strings in backwards order:**

The order should not change the result. Points out if I created functions that cannot be used backwards/any other type of order. Especially important for finding the minimum.

1. **Strings with overlapping items:**

If there is a function in which all content needs to be used, duplicate items can test whether or not the functions consider this fact. (i.e – an array with a,b,c,d,e and another array with d,e,f,g,h)

1. **Content not found in the array:**

When putting in content not found in the array, it should return 0 or false as opposed to a value (results vary with each case). Highlights if there is a tendency for a loop to act before checking cases that cannot be done.