**CS Project 4 report**

1. **A brief description of notable obstacles you overcame:** 
   1. The biggest obstacle I faced was figuring out how to implement removeDups(). I started by trying to combine the findRun() and moveToEnd() functions. My intention was to implement in such a way that once a run of more than 1 element was found (i.e. findRun returned true and beg != end) then move all the elements in that run (except the first) to the end of the known array. For some reason, this kept giving me errors when duplicates occurred in different runs. I spent roughly 5 hours debugging this error but eventually realized that since findRun() returns only the earliest occurring error, my approach was flawed. For example in the array {“1”, “2”, “1”, “1”} the if condition: if(findRun(“1”) && beg != end), we want the array to locate the second occurrence of 1 and remove the third one. However, findRun() will set beg and end to 0 and since beg!=end, so it does not enter the if condition at all. I ended up implementing the function without using findRun() at all, which was actually relatively simple.
   2. I also struggled a bit for the last function divide() but eventually I was able to create a function only using moveToBeginning and very few lines of code.
   3. Finally, I found it a bit tedious to write all the assert statements because a lot of the functions changed the inputs as well so I needed to create a new array for every new test case.
2. **List of test cases using assert statement. The comments have been included in the code itself, so the test cases have simply been copy pasted here (with comments) for convenience.**

string h[7] = { "neil", "sonia", "john", "amy", "", "elena", "john" };

string g[0];

string d[9] = {

"clarence", "neil", "amy", "amy", "ketanji", "ketanji", "ketanji", "amy", "amy"

};

string p[5] = { "john", "sonia", "samuel", "elena", "neil" };

string gg[4] = { "neil", "sonia", "amy", "elena" };

string r[4] = { "nakul", "sana", "angad", "angad" };

//declaring some arrays to be used for testing (of functions which don't modify arrays)

assert(enumerate(h, 7, "john") == 2); //checking for multiple occurences

assert(enumerate(h, 7, "") == 1); // checking for empty string

assert(enumerate(h, 7, "brett") == 0); // checking for missing element

assert(enumerate(h, 0, "john") == 0); // look at the first 0 elmts

assert(enumerate(g, 0, "john") == 0); // output 0 for empty (unitialized) array

assert(enumerate(g, 0, "") == 0); // output 0 for empty (unitialized) array

assert(enumerate(g, 0, "brett") == 0); // output 0 for empty (unitialized) array

assert(enumerate(g, 0, "john") == 0); // output 0 for empty (unitialized) array

assert(enumerate(g, -1, "nakul") == -1); //error works for empty array

assert(findMatch(d, 9, "ketanji") == 4); // mutliple occurences returns first

assert(findMatch(d, 4, "ketanji") == -1); // not found when looking at first n elmts

assert(findMatch(h, 7, "john") == 2); // checking different array to be sure

assert(findMatch(h, 2, "john") == -1); // not found when looking at first n elmts

assert(findMatch(g, 0, "nakul") == -1); // can't return a position if no elmts in array

int bg = -999;

int en = -999;

assert(! findRun(h, 7, "nakul", bg, en) && bg == -999 && en == -999); // false if elmt not present

assert(! findRun(h, -1, "nakul", bg, en) && bg == -999 && en == -999); // false if n <0

assert(! findRun(d, 2, "amy", bg, en) && bg == -999 && en == -999); // only looks at first n elts

assert(! findRun(d, 0, "clarence", bg, en) && bg == -999 && en == -999); // false because it won't be found in 0 elements

assert(! findRun(d, 9, "samuel", bg, en) && bg == -999 && en == -999); // false if not found at all

assert(findRun(h, 7, "amy", bg, en) && bg == 3 && en == 3); // checking that it finds first run

assert(findRun(d, 9, "ketanji", bg, en) && bg == 4 && en == 6); // works for multiple consecutive

assert(findRun(d, 4, "amy", bg, en) && bg == 2 && en == 3); // check with a shorter version of array (checking for some one-off error)

assert(findRun(d, 3, "amy", bg, en) && bg == 2 && en == 2); // should say 2 and 2, not 2 and 3 because it's only checking the first 3 elements of the array

assert(findRun(h, 7, "", bg, en) && bg == 4 && en == 4); // works for empty string as well

assert(findRun(d, 9, "amy", bg, en) && bg == 2 && en == 3); // in spec

assert(findRun(d, 9, "neil", bg, en) && bg == 1 && en == 1); // in spec

assert(findRun(r, 4, "angad", bg, en) && bg == 2 && en == 3); // checking one-off error again

assert(findRun(r, 3, "angad", bg, en) && bg == 2 && en == 2); // checking one-off error again

assert(findMin(d, 9) == 2); //finds first occurence of the min

assert(findMin(d, 2) == 0); // returns 0 as a position (no error)

assert(findMin(h, 7) == 4); // empty string is the minimum

assert(findMin(g, 0) == -1); // can't return a position for 0 elmts

assert(findMin(p, 5) == 3); // extra check with different array

string mte\_1[4] = { "neil", "sonia", "amy", "elena" };

assert(moveToEnd(mte\_1, 0, 0) == -1); //can't find a position in a 0 array elmt

assert(moveToEnd(mte\_1, 1, 0) == 0); // with one element, it will always return 0

assert(moveToEnd(mte\_1, 4, -1) == -1); // position can't be negative

assert(moveToEnd(mte\_1, 4, 4) == -1); // position n doesn't exist in n elt array

assert(moveToEnd(mte\_1, 4, 1) == 1 && mte\_1[1] == "amy" && mte\_1[3] == "sonia"); // checking for proper functionality of moveToEnd for normal case

assert(moveToEnd(mte\_1, 0, 1) == -1); // checking of n = 0 error even when pos != 0

string mtb\_1[4] = { "elena", "amy", "sonia", "john" };

assert(moveToBeginning(mtb\_1, 0, 0) == -1); //can't find a position in a 0 array elmt

assert(moveToBeginning(mtb\_1, 1, 0) == 0); // with one element, it will always return 0

assert(moveToBeginning(mtb\_1, 4, -1) == -1); // position can't be negative

assert(moveToBeginning(mtb\_1, 4, 4) == -1); // position n doesn't exist in n elt array

assert(moveToBeginning(mtb\_1, 4, 2) == 2 && mtb\_1[0] == "sonia" && mtb\_1[2] == "amy"); // checking for proper functionality of moveToEnd for normal case

assert(findDifference(h, 4, gg, 4) == 2); // normal functioning

assert(findDifference(h, 2, gg, 2) == 2); // return length if equal until end

assert(findDifference(h, 2, gg, 3) == 2); // return length if equal until end

assert(findDifference(h, 3, gg, 3) == 2); // shortened array not equal to return length

assert(findDifference(h, 0, gg, 0) == -1); // can't return a position for a zero elt array

assert(findDifference(h, 0, gg, 1) == -1); // can't return a position for a zero elt array

assert(findDifference(h, 5, p, 5) == 0); // returns 0 if first elmt is different

string dups1[4] = {"nakul", "arnav", "nakul", "nakul"};

assert(removeDups(dups1, 4) == 3 && dups1[0] == "nakul" && dups1[1] == "arnav" && dups1[2] == "nakul"); // check normal functioning for dup elements occuring non-consecutively

string dups2[10] = {"a", "b", "b", "c", "c", "c", "a", "b", "a", "a"};

assert(removeDups(dups2, 10) == 6 && dups2[0] == "a" && dups2[1] == "b" && dups2[2] == "c" && dups2[3] == "a" && dups2[4] == "b" && dups2[5] == "a"); // works with complicated combinations of duplicate sequences as well

string dups3[20] = {"1", "", "", "", "1", "2", "2"};

assert(removeDups(dups3, 7) == 4 && dups3[0] == "1" && dups3[1] == "" && dups3[2] == "1" && dups3[3] == "2"); // works for the empty string as well

string dups4[10] = {"", "", "", "", ""};

assert(removeDups(dups4, 4) == 1 && dups4[0] == ""); // works with all elmts equal

assert(removeDups(dups2, 0) == 0); // no elmts there so no elmts are dups so no elmts are removed so 0 elmts are retained

assert(removeDups(dups2, -5) == -1); // error if n < 0

string big[10] = { "elena", "john", "amy", "ketanji", "neil", "amy" };

string little1[10] = { "john", "ketanji", "neil" };

assert(subsequence(big, 6, little1, 3)); // returns true (from spec)

string little2[10] = { "amy", "john" };

assert(! subsequence(big, 6, little2, 2)); // returns false (from spec)

string little3[10] = { "john", "amy", "amy" };

assert(subsequence(big, 6, little3, 3)); // returns true (from spec)

string little4[10] = { "john", "john", "amy" };

assert(! subsequence(big, 6, little4, 3)); // returns false (from spec)

assert(subsequence(big, 6, little4, 0)); // returns true (from spec)

string rlittle[3] = {"nakul", "sana", "angad"};

assert(subsequence(r, 4, rlittle, 3)); // normal functioning for my own example

assert(subsequence(big, 0, little1, 0)); // a 0 length substring is a subsequence even of the 0 length array

assert(! subsequence(big, 2, little1, 3)); // false if little is bigger than big

for (int i = 0; i <= 6; i++) assert(subsequence(big, 6, big, i)); // checking that all subarrays are subsequences

string big2[5] = {"a", "a", "a", "a", "b"};

string little21[4] = {"a", "a", "a", "a"};

string little22[5] = {"a", "a", "a", "a", "a"};

assert(subsequence(big2, 5, little21, 4)); // checking for one-off errors

assert(!subsequence(big2, 5, little22, 5)); // checking for one-off errors

string x1[5] = { "amy", "elena", "elena", "ketanji", "samuel" };

string y1[4] = { "clarence", "elena", "john", "sonia" };

string z1[20];

string result1[20] = {"amy",

"clarence",

"elena",

"elena",

"elena",

"john",

"ketanji",

"samuel",

"sonia"};

for (int i = 0; i < makeMerger(x1, 5, y1, 4, z1, 20);i++){ // checking all elmts are as desired by creating a results array that it should match

assert(z1[i] == result1[i]);

} // returns 9

string x2[4] = { "john", "john", "samuel", "sonia" };

string y2[4] = { "amy", "elena", "john", "ketanji" };

string z2[10];

assert(makeMerger(x2, 4, y2, 4, z2, 10) == 8 && z2[5] == "ketanji"); // returns the right number

// for (int i = 0; i < 8; i++){

// cout << z2[i] << endl;;

// }

string x3[4] = { "john", "john", "sonia", "samuel" };

string y3[4] = { "amy", "elena", "john", "ketanji" };

string z3[10];

assert(makeMerger(x3, 4, y3, 4, z3, 10) == -1); // error because inputs not sorted

string x4[4] = { "john", "john", "samuel", "sonia" };

string y4[4] = { "amy", "elena", "john", "ketanji" };

string z4[10];

assert(makeMerger(x4, 4, y4, 4, z4, 7) == -1); // error because max is too small

string x5[5] = {"1", "3", "5", "7", "9"};

string y5[5] = {"2", "4", "6", "8", "9"};

string z5[20];

string result5[10] = {"1", "2", "3", "4", "5", "6", "7", "8", "9", "9"};

assert(makeMerger(x5, 5, y5, 5, z5, 10) == 10); // works for numbers (easy to see order)

for (int i = 0; i < 10; i++){

assert(z5[i] == result5[i]); // works for numbers (easy to see order)

}

//unsure about whether 0 should return an error here or not.

//current logic is spec says "or return n if no elements not less than divider" and since there are no elements at all in the array this is a vacuous truth. n is 0 so return 0.

string sc[6] = { "john", "amy", "samuel", "elena", "sonia", "neil" };

assert(divide(sc, 6, "ketanji") == 3); // sorts correctly for an array of names (without the divider)

assert(divide(sc, 0, "ketanji") == 0); // returns 0 for n = 0 but does not return an error

string sc2[4] = { "john", "sonia", "amy", "neil" };

assert(divide(sc2, 4, "neil") == 2 && sc2[2] == "neil"); // returns position of divider if it is present

string sc3[6] = {"c", "c", "c", "c", "a", "b"};

assert(divide(sc3, 6, "b") == 1); // works with repeats as well

string sc4[7] = {"aanya", "nakul", "sana", "saachi", "anirudh", "shaurya", "mihir"};

assert(divide(sc4, 7, "arnav") == 2); // dividing with a name not present

assert(divide(sc4, 7, "tahaa") == 7); // dividing with a name not present but greater than all

assert(divide(sc4, 7, "ronit") == 4); // dividing with a name not present in the middle

assert(divide(sc4, 7, "nakul") == 3); // dividing with a name that is present