CS Lecture 1 – Professor Smallberg Leo Gretzinger

Project 4 Report 11/5/18

**Obstacles:**

Most of the functions in this project were not overly difficult and after thinking about each one for a couple minutes, I discovered a solution that solved the problem efficiently. The notable exceptions to this were the functions ‘countRuns’ and ‘divide.’ CountRuns was challenging because I could not figure out how to account for the fact that a run might occur in the last position of the array. I continued to get overflow and out of bounds errors, and the function continued to access indexes outside of the specified array. To fix this, I changed the for loop range to one less, and then added one to runs if the last object of the array is different. The second, harder function was ‘divide.’ This function can be very complicated based on the sorting method used, but after trying a few different strategies, I realized that I could utilize the rotateLeft function that I had previously created to implement the most efficient way to solve the problem – with only three for loops.

**Test Cases:**

// List of arrays of strings

string a[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string b[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string c[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Zarry", "Pairy", "Zarry" };

string d[10] = { "Zzrry", "Perry", "Larry", "Gary", "Harry", "Zarry", "Pairy", "Zarry" };

string e[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Carry", "Pairy", "Zarry" };

string f[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Zarry", "Pairy", "Zarry" };

string g[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Zarry", "Pairy", "Zarry" };

string h[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Zarry", "Pairy", "Zarry" };

string i[10] = { "Jerry", "Jerry", "Jerry", "Gary", "Harry", "Zarry", "Zarry", "Larry" };

string ia[10] = { "Jerry", "Jerry", "Jerry", "Gary", "Harry", "Zarry", "Zarry", "Zarry"};

string j[10] = { "Jerry", "Harry", "Jerry", "Gary", "Harry", "Zarry", "Zarry", “Larry" };

string k[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string l[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry", Larry"};

string m[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string n[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry", “Larry" };

string o[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string p[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string q[10] = { "Jerry", "Perry", "Jerwy", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string r[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string s[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string t[10] = { "Perry", "Larry", "Gary", "Gary", "Harry", "Jerry", "Perry", "Larry" };

string u[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string v[10] = { "Hairy", "Pairy", "Carry", "Gary", "Harry", "Hairy", "Pairy", "Jerry" };

string w[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string x[10] = { "Carry", "Pwrry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string y[10] = { "Jwrry", "Pwrry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string z[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string aa[10] = { "Jerry", "Larry", "Gary", "Cherry", "Larry", "Gary", "Harry", "Carry"};

string ab[10] = { "Larry", "Gary", "Harry", "Gary", "Harry", "Hairy", "Pairy", "Carry" };

string names[10] = { "eleni", "gavin", "kevin", "greg", "betty", "fiona" };

string set1[10] = { "dianne", "betty", "greg", "gavin" };

string set2[10] = { "xavier", "ed" };

string ac[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Zarry", "Pairy", "Zarry"};

string ad[10] = { "Jwrry", "Pwrry", "Lwrry", "Gwry", "Hwrry", "Zwrry", "Pwiry", "Zwrry"};

string ae[10] = { "Jerry", "Perry", "Larry", "Gary", "Harry", "Zarry", "Pairy", "Zarry"};

string af[10] = { "Jwrry", "Pwrry", "Lwrry", "Gwry", "Hwrry", "Zwrry", "Pwiry", "Zarry"};

string ag[10] = { "array", "zrray", "crray", "wrray", "orray", "zrray" };

string ah[10] = { "zrray", "zrray", "zrray", "zrray", "zrray", "zrray" };

string ai[10] = { "zrray", "zrray", "zrray", "array", "zrray" };

string candidate[6] = { "dianne", "fiona", "gavin", "xavier", "ed", "betty" };

string candidate2[4] = { "gavin", "kevin", "fiona", "john" };

// Checks if the appendToAll function adds exclamation mark to the first and last indexes

assert(appendToAll(a, 3, "!!!") == 3 && a[0] == "Jerry!!!" && a[2] == "Larry!!!");

// Checks if the appendToAll function works with a size 0 array

assert(appendToAll(a, 0, "!!!") == 0);

// Checks if appendToAll returns -1 if size is invalid

assert(appendToAll(a, -1, "!!!") == -1);

// Checks if the lookup function finds an object at the end of an array

assert(lookup(b, 7, "Pairy") == 6);

// Checks if the lookup function finds an object in the middle of an array

assert(lookup(b, 7, "Larry") == 2);

// Checks if the lookup function identifies if an object is not in the array

assert(lookup(b, 2, "Larry") == -1);

// Checks if lookup finds an object at the beginning of an array

assert(lookup(b, 7, "Jerry") == 0);

// Checks if lookup can’t find an object in a 0 array

assert(lookup(b, 0, "Larry") == -1);

// Checks if lookup gives an error for negative size

assert(lookup(b, -1, "Larry") == -1);

// Checks if the positionOfMax function identifies a max in the middle of the array

assert(positionOfMax(c, 8) == 5);

// Checks if the positionOfMax function identifies if the max is the first index

assert(positionOfMax(d, 8) == 0);

// Checks if the positionOfMax function identifies if the max in the middle

assert(positionOfMax(e, 8) == 7);

// Checks if the positionOfMax function identifies a max in a 0 array

assert(positionOfMax(f, 0) == 0);

// Checks if the position is negative the function gives the error code

assert(positionOfMax(g, -1) == -1);

// Checks if the rotateLeft function rotates a specified index from the middle

assert(rotateLeft(h, 8, 4) == 4 && h[1] == "Perry" && h[5] == "Pairy" && h[7] == "Harry");

// Checks if rotateLeft rotates a specified index from the beginning

assert(rotateLeft(h, 8, 0) == 0 && h[0] == "Perry" && h[5] == "Zarry" && h[7] == "Jerry");

// Checks if rotateLeft rotates a specified index from the end

assert(rotateLeft(h, 8, 7) == 7 && h[0] == "Perry" && h[5] == "Zarry" && h[7] == "Jerry");

// Checks if rotateLeft gives an error for a negative size

assert(rotateLeft(h, 0, 0) == -1);

// Checks if the countRuns function counts the runs in an array

assert(countRuns(i, 8) == 5);

// Checks if the countRuns function counts the runs in a 0 array.

assert(countRuns(i, 0) == 0);

// Checks if the countRuns function correctly accesses a not full array

assert(countRuns(i, 7) == 4);

// Checks if the countRuns function counts the runs in an array of size 1

assert(countRuns(j, 1) == 1);

// Checks if the countRuns function counts runs with the last two objects the same

assert(countRuns(ia, 8) == 4);

// Checks if the flip function flips an array of even size

assert(flip(k, 8) == 8 && k[0] == "Carry" && k[7] == "Jerry");

// Checks if the flip function flips an array of odd size

assert(flip(l, 9) == 9 && l[0] == "Larry" && l[8] == "Jerry");

// Checks if flip flips an array of size 2

assert(flip(l, 2) == 2 && l[0] == "Carry" && l[1] == "Larry");

// Checks if differ identifies if two arrays are equal until the first one runs out

assert(differ(m, 8, n, 9) == 8);

// Checks if differ identifies if two arrays are equal until the second one runs out

assert(differ(o, 2, p, 1) == 1);

// Checks if differ identifies if two arrays differ in the middle

assert(differ(q, 8, r, 8) == 2);

// Checks if differ returns 0 if one array is size 0

assert(differ(q, 0, r, 1) == 0);

// Checks if subsequence identifies a subsequence in the beginning

assert(subsequence(s, 8, t, 3) == 1);

// Checks if subsequence identifies a subsequence in the end

assert(subsequence(u, 8, v, 3) == 5);

// Checks if subsequence identifies a subsequence of length 1 at the end

assert(subsequence(w, 8, x, 1) == 7);

// Checks if subsequence identifies if a subsequence is not present

assert(subsequence(y, 8, z, 2) == -1);

// Checks if subsequence identifies a subsequence in the middle

assert(subsequence(aa, 8, ab, 3) == 4);

// Checks if lookupAny finds a common object in beginning

assert(lookupAny(names, 6, set1, 4) == 1);

// Checks if lookupAny identifies if there are no common objects

assert(lookupAny(names, 6, set2, 2) == -1);

// Checks if lookupAny identifies if there are no common objects in long arrays

assert(lookupAny(ac, 8, ad, 8) == -1);

// Checks if lookupAny identifies if there is a common object in the middle

assert(lookupAny(ae, 8, af, 8) == 5);

//Checks if divide divides an array of varying objects

assert(divide(ag, 6, "orray") == 2 && ag[1] == "crray" && ag[2] == "orray" && ag[5] == "wrray");

// Checks if divide divides an array of all the same objects

assert(divide(ah, 6, "orray") == 0);

// Checks if divide divides an array with only one different object

assert(divide(ai, 5, "orray") == 1 && ag[0] == "array" && ag[4] == "zrray");

// Checks if divide divides a shorter array correctly

assert(divide(candidate, 6, "eleni") == 3);

// Checks if divide divides an array with one of the objects equaling the divider.

assert(divide(candidate2, 4, "john") == 2);