1.

Actually those functions are much easier than the functions in project 3, but I still struggled with some minor obstacles. For example, whether to return -1 or 0 when n equals to 0 is really tricky, for it depends on different conditions for each function. I spent half an hour check it one by one to make sure I get all the return value right.

2. TEST DATA

string a[] = {"blake", "steven", "tony"};

string b[] = {"blake", "steven", "", "", "tony", "shawn"};

string c[] = {"jack", "steven"};

string d[] = {"blake", "steven", "tony", "shawn"};

string e[] = {"blake", "steven", "", "tony", "shawn", "!"};

string f[] = {"blake", "steven", "", "tony", "shawn", ""};

string g[] = {"blake", "steven", "jack", "tony", "shawn", "thomas"};

assert(appendToAll(d, -2, "!!!") == -1 && d[0] == “blake");

n is negative

assert(appendToAll(d, 0, "!!!") == 0 && d[0] == “blake")

n equals to 0

assert(appendToAll(d, 1, "!!!") == 1 && d[0] == "blake!!!" && d[2] == “tony");

normal case

assert(lookup(a, -1, "blake") == -1);

n is negative

assert(lookup(a, 0, "tony") == -1);

n equals to 0

assert(lookup(a, 2, "tony") == -1);

there’s no such string

assert(lookup(a, 2, "steven") == 1);

normal case

assert(lookup(a, 3, "jack") == -1);

there’s no such string

assert(positionOfMax(a, -2) == -1);

n is negative

assert(positionOfMax(a, 0) == -1);

n equals to 0

assert(positionOfMax(a, 1) == 0);

the first is the max

assert(positionOfMax(a, 3) == 2);

normal case

assert(rotateLeft(a, -1, 1) == -1);

n is negative

assert(rotateLeft(a, 2, -1) == -1);

pos is negative

assert(rotateLeft(a, 2, 3) == -1);

pos is larger than n

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

when n equals to 0

assert(rotateLeft(a, 2, 1) == 1 && a[1] == “steven");

normal case

assert(countRuns(a, -2) == -1);

n is negative

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

n equals to 0

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

only one string

assert(countRuns(b, 4) == 3);

normal case

assert(countRuns(b, 6) == 5);

normal case

assert(flip(b, -2) == -1);

n is negative

assert(flip(b, 0) == 0);

n equals to 0

assert(flip(b, 3) == 3 && b[0] == "" && b[4] == “tony");

normal case

assert(differ(e, -1, f, 3) == -1);

n1 is negative

assert(differ(e, 3, f, -2) == -1);

n2 is negative

assert(differ(e, 0, f, 3) == 0);

n1 runs out first

assert(differ(e, 2, f, 2) == 2);

n1 and n2 run out at the same time

assert(differ(e, 3, f, 5) == 3);

n1 runs out first, but n1 not equals to 0

assert(differ(e, 6, f, 6) == 5);

normal case

assert(subsequence(e, -1, f, 3) == -1);

n1 is negative

assert(subsequence(e, 1, f, -2) == -1);

n2 is negative

assert(subsequence(e, 3, f, 2) == 0);

normal case

assert(subsequence(e, 3, f, 0) == 0);

a sequence of 0 elements to be a subsequence of any sequence

assert(subsequence(e, 0, f, 0) == 0);

a sequence of 0 elements to be a subsequence of any sequence

assert(subsequence(e, 3, f, 5) == -1);

n2 is larger than n1

assert(subsequence(e, 6, f, 6) == -1);

a1 does not contain a2 as a contiguous subsequence

assert(lookupAny(c, -1, f, 3) == -1);

n1 is negative

assert(lookupAny(c, 1, f, -2) == -1);

n2 is negative

assert(lookupAny(c, 3, f, 2) == 1);

normal case

assert(lookupAny(c, 3, f, 0) == -1);

no element of a1 is equal to any element of a2

assert(lookupAny(c, 0, f, 2) == -1);

no element of a1 is equal to any element of a2

assert(split(g, -2, "tony") == -1);

n is negative

assert(split(g, 0, "tony") == 0);

n equals to 0

assert(split(g, 3, "tony") == 3);

there are no such elements

assert(split(g, 6, "shawn") == 2);

normal case