CSCI 3104
Spring 2018
Problem Set 7

Allison, George
02/17

March 12, 2018

1.(i) The "alignStrings()" function creates a 2D array as a matrix of the values to store the optimal costs of aligning each input string. The function initializes the first row and column of the matrix, because they will be the same regardless of what the input strings are. Next, a loop is used to match all combinations of lengths of substrings and sends the matches to the helper function "cost()" for the cost to be calculated. The cost function uses a loop to iterate through all characters of input string i and compare each with the characters of input string j. The first thing that is checked is if the current position in i is empty; if it is, the corresponding letter in j can be inserted for a cost of 1. If the position is not empty, the function checks to see if it is equal to the corresponding position in j, in which case no action needs to be taken. If the characters in i and j are not equal the function checks to see if the same letter appears later in string i. If so, the characters in between are deleted. The function only checks 13 characters ahead, because any larger amounts of character deletion would be less efficient than using a substitution operation. If the needed letter is indeed more that 13 characters away, or does not appear at all, a substitution operation is used for a cost of 12.

(ii) The extractAlignment() method starts at element [i][j] in the matrix of optimal operations, and compares all of the adjacent operations to find the one with the lowest cost. Once the lowest cost is found it is added to the myVec[] vector, which represents the vector of optimal operations and is returned at the end of the funciton. The algorithm moves to the optimal element that is in range of its current position, and then it continues until it reaches element [0][0]. Once element [0][0] is reached, the matrix will have been successfully traversed and myVec[] will contain each step of the optimal route.

(iii) commonSubstrings() finds the common substrings in x by looping through its list of operations and checking if they are nops or not. If there are at least L consecutive nops then the characters of x that correspond with the consecutive nops are added to an array of common substrings. Once each character in x has been checked then the array of common substrings is returned.

The following pages contain the implementation of these functions, along with some helper functions, programmed in Java, and written and tested in IntelliJ IDEA.

```java
import java.util.Scanner;//Used to get user input in main() allowing easier testing of
    this program.

public class algorithmsStringAlign {

    //printMatrix() is a helper function that prints the matrix
    //@param String x: the first string input, on the y-axis of the matrix
    //@param String y: the second string input, on the x-axis of the matrix
    //@param matrix: the 2D array of operation costs for the two strings
    private static void printMatrix(String x, String y, int[][] matrix) {
        System.out.println("\n=====================================");
        System.out.printf("%5c ", ' ');
        for (int i = 0; i < y.length(); i++) {
            System.out.printf("%5c ", y.charAt(i));
        }
        System.out.println();

        for (int i = 0; i < x.length(); i++) {
            System.out.printf("%5c ", x.charAt(i));
            for (int j = 0; j < y.length(); j++) {
                System.out.printf("%5d ", matrix[i][j]);
            }
            System.out.println();
        }

        System.out.println("=====================================\n");
    }


    //cost() function is a helper function for alignStrings()
    //@param String x: the string that is being aligned (edited)
    //@param String y: the string that is being aligned to (unedited)
    //@returns an integer representing the optimal cost to be added to the matrix
    private static int cost(String x, String y) {
        int myCost = 0;
        int xlen = x.length();
        int ylen = y.length();

        for (int indx = 0; indx < xlen; indx++) {//looping through x
            if (indx < ylen) { //making sure we don't go out of bounds on y
                if (x.charAt(indx) != y.charAt(indx)) {//if letters are inequal
                    int k = indx;
                    while (k < 12 && k < xlen) {//search ahead for matching letters
                        //only searches 13 ahead because after that point, a
                        //substitution operation would be more efficient
                        if (x.charAt(k) == y.charAt(indx)) {
                            //delete the previous letters in x
                            myCost += (k - indx);
                            indx = k;
                        }
                        k++;
```

```
                }
                myCost += 12;
            }
        } else {//once we do go out of bounds on y, delete the rest of x
            myCost++;//cost of deletion operation
        }
    }
    return myCost;
}


//alignStrings() function for problem (1)(i)
//@param String x: the string that is being aligned (edited)
//@param String y: the string that is being aligned to (unedited)
//@returns int[][] S: the matrix of the optimal costs for each subproblem
private static int[][] alignStrings(String x, String y) {
    int xlen = x.length();
    int ylen = y.length();
    int[][] S = new int[xlen][ylen]; //initialization of our matrix

    //initialization for first row and column
    for (int i = 0; i < xlen; i++) {
        S[i][0] = i;
    }
    for (int i = 0; i < ylen; i++) {
        S[0][i] = i;
    }

    //initialization for second column
    int del = 0;//number of insertions
    int ins = 0;//number of deletions
    boolean eq = false;//signifies when there is an equivalent letter present
    for (int i = 1; i < xlen; i++) {
        if (x.charAt(i) != y.charAt(1)) {
            if (i == 1)
                ins++;
            del++;
            S[i][1] = ins + del;
        } else {
            if (eq)//if there is already a matching letter
                del++;
            eq = true;
            S[i][1] = del;
        }
    }

    //initialization for second row
    del = 0;//number of insertions
    ins = 0;//number of deletions
    eq = false;
    for (int i = 1; i < ylen; i++) {
        if (x.charAt(1) != y.charAt(i)) {
```

```
            if (i == 1)
                ins++;
            del++;
            S[1][i] = ins + del;
        } else {
            if (eq)//if there is already a matching letter
                del++;
            eq = true;
            S[1][i] = del;
        }
    }


    //filling out the rest of the table by using the cost() function
    for (int i = 2; i < xlen; i++) {
        for (int j = 2; j < ylen; j++) {
            //passing substrings of original string to cost()
            S[i][j] = cost(x.substring(0, i + 1), y.substring(0, j + 1));
        }
    }

    printMatrix(x, y, S);
    return S;
}


//alignStrings() function for problem (1)(ii)
//@param 2D array S is the optimal cost matrix
//@param String x: the string that is being aligned (edited)
//@param String y: the string that is being aligned to (unedited)
//@returns an array as a vector of optimal sequence of edit operations
private static int[] extractAlignment(int[][] S, String x, String y) {
    int xlen = x.length();
    int ylen = y.length();
    int[] myVec = new int[x.length()];
    if (xlen == 0 && ylen == 0) {
        int i = 1;
        myVec[myVec.length - i] = S[xlen][ylen];

    }
    //else ...
    return myVec;
}


//commonSubstrings() Finds substrings of a specified length that are equivalent with
//a compared string, resulting in all null operations.
//@param String x: An ASCII string
//@param int L: An integer that is 1  L    n
//@param int[] a: An array of optimal edits to x
//@returns String[] substr: An array of the substrings of length at least L in
//String x that aligns exactly, via a run of no-ops, to a substring in y
```

```
    private static String[] commonSubstrings(String x, int L, int[] a){
        int i = 0;
        int nops = 0;
        int optCtr= 0;
        int arrCtr = 0;

        String[] substr = new String[90];

        while(i < x.length()){
            if(a[i] - optCtr == 0){
                nops++;
            }
            else{
                i++;
                nops = 0;
            }
            if(nops == L){
                String cs = "";
                for(int j = 0; j < 9; j++){
                    cs += x.charAt(j);
                }
                substr[arrCtr] = cs;
                arrCtr++;
            }
        }
        return substr;
    }


    public static void main(String[] args){
        Scanner s = new Scanner(System.in);
        System.out.print( "Enter string 1: " );
        String x = s.nextLine();
        System.out.print( "Enter string 2: " );
        String y = s.nextLine();
        String x2 = '-' + x;
        String y2 = '-' + y;
        System.out.println("Aligning string 1 with string 2...");

        //running all functions using each others' outputs
        commonSubstrings(x, x.length() / 2, extractAlignment(alignStrings(x2, y2), x, y));
    }
}
```

b) The function alignStrings() creates a matrix whics costs $O(n^2)$, and then it iterates through each letter of the input string to do comparison operations, and fills in the matrix with values resulting in cost $O(n) + O(n^2)$. The extractAlignment() function compiles an array of length n, ultimately having a cost of $O(n)$. The commonSubstrings() function loops through the vector of operations once, which is of length n, resulting in another cost of $O(n)$. The final cost is $O(3n)+O(2n^2)$

c) I don't know.

d) I don't know.

2.(a)(i) MemPell(n) will run recursively until a base case of either 0 or 1 is reached. Once that happens, each index of array P from 0 to n will be filled in ascending order. The function traverses through its computation tree by running each recursive function call as soon as it is made. For example, in the line of code $P[n] = 2 * MemPell(n-1) + MemPell(n-2)$ the left-most function call of $2 * MemPell(n-1)$ will be made, and the function will continue recursion until a base case is reached. Once the base case is reached and $2 * MemPell(n-1)$ returns a value, the rest of the line of code, $+MemPell(n-2)$, will be run in order to provide a value for $P[n]$. Array P will be filled from index 0 to n each time the function is called.

(ii) The memoization in MemPell() allows it to only calculate each Pell number once. Its use of recursion to continuously divide each iteration's function calls resutls in n operations that have an asymptotic running time of $O(log(n))$, therefore the function has a total asymptotic running time of $O(nlog(n))$.

b) The time usage of DynPell() is $O(n)$ because it consists of one for loop that executes n operations. The space usage is also $O(n)$ because it stores all n elements in array P. This is slightly faster than the previous algorithm because it only does each operation once, instead of having multiple operations for the same values of n which happens often in the recursive version. The space usage here is the same as in MemPell().

c) The order of the variables in Hermione's code is erroneous, here is a correction:

```
FasterPell(n) :
   a = 1, b = 0
   for i = 2 to n
      c = 2*a + b
      b = a
      a = c
   end
   return a
```

The time complexity of this algorithm would still be $O(n)$, however its space usage is $O(1)$ because it does not store the values in an array - elements a, b, and c can be stored in registers and swapped out each iteration.

d)

	Pell()	MemPell()	DynPell()	FasterPell()
Time	$O(\phi^n)$	$O(nlogn)$	$O(n)$	$O(n)$
Space	$O(n)$	$O(n)$	$O(n)$	$O(1)$
Struct	Tree	Tree	Tree	Array

e) This is my implementation of FasterPell() in python:

```
n = input("Enter the Pell number to calculate: ")
import time
start_time = time.time()
a = 1
b = 0
for i in range (2, n):
    c  = 2 * a + b
    b = a
    a = c
print "#", n, "Pell number is:", a
r = (time.time() - start_time) * (10**9)
print "That calculation took about", int(round(r)), "nanoseconds to run."
```

Running program produces the following output:

```
Enter the Pell number to calculate: 217
# 217 Pell number is:
    1690452762517806008348348884429892215785396051012705640942443872561314055939117748 0
That calculation took about 256062 nanoseconds to run.
```

By entering my date of birth (Feb. '97) as the Pell number to calculate it took the program 256062 nanoseconds to run. We are to assume that it takes one nanosecond per operation. We also know that FasterPell has a running time of $O(n)$, and Rons classic recursive algorithm would take $O(\phi^n)$ time. Therefore, Ron's algorithm would take ϕ^{256062} nanoseconds to run, which equals $1.969387 * 10^{53497}$ years. The FasterPell algorithm is quite a bit faster than the classic recursive algorithm.