CSCI 3104 **Austin Griffith**
Problem Set 7

1. (45 pts) Recall that the *string alignment problem* takes as input two strings x and y, composed of symbols $x_i, y_j \in \Sigma$, for a fixed symbol set Σ, and returns a minimal-cost set of *edit* operations for transforming the string x into string y.

   Let x contain n_x symbols, let y contain n_y symbols, and let the set of edit operations be those defined in the lecture notes (substitution, insertion, deletion, and transposition).

   Let the cost of *indel* be 1, the cost of *swap* be 10 (plus the cost of the two *sub* ops), and the cost of *sub* be 10, except when $x_i = y_j$, which is a "no-op" and has cost 0.

   In this problem, we will implement and apply three functions.

   (i) `alignStrings(x,y)` takes as input two ASCII strings x and y, and runs a dynamic programming algorithm to return the cost matrix S, which contains the optimal costs for all the subproblems for aligning these two strings.

```
alignStrings(x,y) :                // x,y are ASCII strings
   S = table of length nx by ny    // for memoizing the subproblem costs
   initialize S                    // fill in the basecases
   for i = 1 to nx
        for j = 1 to ny
             S[i,j] = cost(i,j)    // optimal cost for x[0..i] and y[0..j]
   }}
   return S
```

   (ii) `extractAlignment(S,x,y)` takes as input an optimal cost matrix S, strings x, y, and returns a vector a that represents an optimal sequence of edit operations to convert x into y. This optimal sequence is recovered by finding a path on the implicit DAG of decisions made by `alignStrings` to obtain the value $S[n_x, n_y]$, starting from $S[0, 0]$.

```
extractAlignment(S,x,y) :  // S is an optimal cost matrix from alignStrings
   initialize a            // empty vector of edit operations
   [i,j] = [nx,ny]         // initialize the search for a path to S[0,0]
   while i > 0 or j > 0
        a[i]  = determineOptimalOp(S,i,j,x,y) // what was an optimal choice?
        [i,j] = updateIndices(S,i,j,a)        // move to next position
   }
   return a
```

   When storing the sequence of edit operations in a, use a special symbol to denote no-ops.

(iii) `commonSubstrings(x,L,a)` which takes as input the ASCII string x, an integer $1 \leq L \leq n_x$, and an optimal sequence a of edits to x, which would transform x into y. This function returns each of the substrings of length at least L in x that aligns exactly, via a run of no-ops, to a substring in y.

(a) From scratch, implement the functions `alignStrings`, `extractAlignment`, and `commonSubstrings`. You may not use any library functions that make their implementation trivial. Within your implementation of `extractAlignment`, ties must be broken uniformly at random.

Submit (i) a paragraph for each function that explains how you implemented it (describe how it works and how it uses its data structures), and (ii) your code implementation, with code comments.

Hint: test your code by reproducing the `APE` / `STEP` and the `EXPONENTIAL` / `POLYNOMIAL` examples in the lecture notes (to do this exactly, you'll need to use unit costs instead of the ones given above).

We define nx as the length of the first string, and ny as the length of the second string.

alignStrings creates a matrix size nxny, and fills this matrix using a nested loop, one running nx times, and the other running ny times. The alignStrings function I have implemented here only uses an array data structure and there are no recursive calls to it so there is no stack used in memory for this function. Since we loop through the whole thing regardless, we have O(nx ny) and Ω(nx ny), thus the asymptotic runtime is O(nx ny)

extractAlignment finds an optimal path in the matrix starting at the end, go- ing back to the beginning of the matrix. For each path, we will traverse some fraction of the matrix, but never all of it based on our way of traversal(up, left, diagonal once, or diagonal twice). Therefore our worst case would be insert- ing or deleting all the letters in one of the strings and vice versa. Therefore we have a runtime of O(max(nx, ny)). The data structures used in this function are the optimal cost matrix S, string variables, and the vector it returns as a that contains the optimal sequence of edit operations to convert x to y.

commonSubstrings only loops through the x string and the a list of optimal ops. Since the size of a len(x), it loops at most len(x) times. Therefore the runtime here is O(nx). The data structures this functions uses is similar to the rest. It takes in string variables, an integer L, and sequence a of optimal edits. The function also uses the list data structure and from my best understanding is still an array. The function is suppose to return each of the substrings of the given length L but I have had no luck getting it to return the common strings within the two documents.

(b) Using asymptotic analysis, determine the running time of the call
`commonSubstrings(x, L, extractAlignment( alignStrings(x,y), x,y ) )`
Justify your answer.

So for the big call to commonSubstrings(x,L,extractAlignment(alignStrings(x,y))), the asymptotic runtime is (nxny)+O(max(nx, ny))+O(nx) = O(nxny). I have determined this from the above observations of each function and what they are doing to calculate the optimal cost of aligning the two input strings.

(c) (15 pts extra credit) Describe an algorithm for counting the number of optimal alignments, given an optimal cost matrix S. Prove that your algorithm is correct, and give is asymptotic running time.

Hint: Convert this problem into a form that allows us to apply an algorithm we've already seen.

This counting of the number of optimal alignments given an optimal cost matrix is very similar to the traversal of a DAG graph. When starting in a directed graph you have a certain path or number of paths to traverse in the direction given. This is similar to that of the traversal of the matrix. When starting at the top left or bottom right and going through the operations of sub, swap, indel, it can be noted there is a direction or spefic way to get from the bottom to top or top to bottom when aligning the strings. I cannot produce an algorithm to do this count of optimal alignments but I know the traversal through the matrix and the traversal through a DAG graph are very similar so if you have an algorithm to count the number of optimal paths in a DAG graph you could implement it to count the optimal alignments for the matrix.

(d) String alignment algorithms can be used to detect changes between different versions of the same document (as in version control systems) or to detect verbatim copying between different documents (as in plagiarism detection systems).

The two `data_string` files for PS7 (see class Moodle) contain actual documents

recently released by two independent organizations. Use your functions from (1a) to align the text of these two documents. Present the results of your analysis, including a reporting of all the substrings in x of length $L = 10$ or more that could have been taken from y, and briefly comment on whether these documents could be reasonably considered original works, under CU's academic honesty policy.

The two documents by no means would pass CU's academic honesty policy because there is much overlap and similarity in the two documents.

2. (10 pts) Ginerva Weasley is playing with the network given below. Help her calculate the number of paths from node 1 to node 14.

   The total number of path form 1 to 14 is 18. Please see attached image for defining each number of paths from any given node j to 14.

3. (20 pts) Ron and Hermione are having a competition to see who can compute the nth Lucas number L_n more quickly, without resorting to magic. Recall that the nth Lucas number is defined as $L_n = L_{n-1} + L_{n-2}$ for $n > 1$ with base cases $L_0 = 2$ and $L_1 = 1$. Ron opens with the classic recursive algorithm:

   ```
   Luc(n) :
      if n == 0 { return 2 }
      else if n == 1 { return 1 }
      else { return Luc(n-1) + Luc(n-2) }
   ```

   which he claims takes $R(n) = R(n-1) + R(n-2) + c = O(\phi^n)$ time.

   (a) Hermione counters with a dynamic programming approach that "memoizes" (a.k.a. memorizes) the intermediate Lucas numbers by storing them in an array `L[n]`. She claims this allows an algorithm to compute larger Lucas numbers more quickly, and writes down the following algorithm.[1]

   ```
   MemLuc(n) {
      if n == 0 { return 2 } else if n == 1 { return 1 }
      else {
         if (L[n] == undefined) { L[n] = MemLuc(n-1) + MemLuc(n-2) }
         return L[n]
      }
   }
   ```

[1] Ron briefly whines about Hermione's `L[n]=undefined` trick ("an unallocated array!"), but she point out that `MemLuc(n)` can simply be wrapped within a second function that first allocates an array of size n, initializes each entry to `undefined`, and then calls `MemLuc(n)` as given.

i. Describe the behavior of `MemLuc(n)` in terms of a traversal of a computation tree. Describe how the array `L` is filled.
MemLuc(n) is recursively called twice in the function, MemLuc(n-1) and MemLuc(n-2). In terms of behavior of the Algorithm of traversal, the computation is inefficient because the call to MemLuc(n-1) will recursively call itself until it reaches the base case and will have much overlap with the next call to MemLuc(n-2). The array will fill as MemLuc makes it recursive calls.

ii. Determine the asymptotic running time of `MemLuc`. Prove your claim is correct by induction on the contents of the array.

The asymptotic running time of MemLuc is O(n) because the function must run n times from n to the base case.

(b) Ron then claims that he can beat Hermione's dynamic programming algorithm in both time and space with another dynamic programming algorithm, which eliminates the recursion completely and instead builds up directly to the final solution by filling the L array in order. Ron's new algorithm[2] is

```
DynLuc(n) :
   L[0] = 2,   L[1] = 1
   for i = 2 to n {  L[i] = L[i-1] + L[i-2]  }
   return L[n]
```

Determine the time and space usage of `DynLuc(n)`. Justify your answers and compare them to the answers in part (3a).

The time complexity of this function is O(n) because the loop runs from 2 to n. The space complexity of this function is dependent on n and the array L[i] get incremented every time the for loop runs.

(c) With a gleam in her eye, Hermione tells Ron that she can do everything he can do better: she can compute the nth Lucas number even faster because intermediate results do not need to be stored. Over Ron's pathetic cries, Hermione says

```
FasterLuc(n) :
   a = 2,  b = 1
   for i = 2 to n
      c = a + b
      a = b
      b = c
   end
```

[2]Ron is now using Hermione's undefined array trick; assume he also uses her solution of wrapping this function within another that correctly allocates the array.

```
return a
```

Ron giggles and says that Hermione has a bug in her algorithm. Determine the error, give its correction, and then determine the time and space usage of `FasterLuc(n)`. Justify your claims.

The error in the function is that a need to be set to 0 and b needs to be set to 2. The time complexity of this function is also O(n) because it the loop runs from 2 to n. The space complexity of this function is O(constant) because the function is not dependent on n it is dependent on the variables only.

(d) In a table, list each of the four algorithms as rows and for each give its asymptotic time and space requirements, along with the implied or explicit data structures that each requires. Briefly discuss how these different approaches compare, and where the improvements come from. (Hint: what data structure do all recursive algorithms implicitly use?)

See attached table image for this question.

(e) (5 pts extra credit) Implement `FasterLuc` and then compute L_n where n is the four-digit number representing your MMDD birthday, and report the first five digits of L_n. Now, assuming that it takes one nanosecond per operation, estimate the number of years required to compute L_n using Ron's classic recursive algorithm and compare that to the clock time required to compute L_n using `FasterLuc`.

See attached python file for implementation.