1. 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).

> `alignStrings(x,y)` is an algorithm that uses a matrix as a data structure that which is populated by the cost to reach that position in the matrix. alignStrings begins by creating an empty matrix that has a number of rows equal to the length of the first string and columns equal to the length of the second string. The matrix begins to fill by setting

the first row and first column as the base case by implementing them as a string created entirely from insert operations. Then the algorithm iterates for the lengths of the strings to compare the letters and add the cost, depending on the requested operation, to the matrix. Then the matrix uses a dynamic programming approach to fill in the rest of the entries based on the entries above, to the left of, and diagonal to that element. Using the above element represents a delete operation, going from the left represents an insert operation, and going diagonally is a substitution. A swap operation occurs from the second-most diagonal element from the desired position. While iterating through the strings, if two letters are the same then, then there is a "no-op" where there is no cost to change the strings, so it copies the cost of the upper-left diagonal element. This continues until both strings have been iterated through and the matrix has been populated.

```
def cost(matrix, i, j):
    cost1 = 9999999
    cost2 = 9999999
    cost3 = 9999999
    cost4 = 9999999
    if i >= 2 and j >= 2:
        cost1 = matrix[i-2][j-2] + swap  #Cost of swap
    if i-1 >= 0 and j-1 >= 0:
        cost2 = matrix[i-1][j-1] + sub  #Cost of Sub
    if i-1 >= 0:
        cost3 = matrix[i-1][j] + delete  #Cost of deleting from x
    if j-1 >= 0:
        cost4 = matrix[i][j-1] + insert  #Cost of inserting into x
    minimumCost = min(cost1, cost2, cost3, cost4)
    return minimumCost


def alignStrings(string1, string2):
    xn = len(string1)
    yn = len(string2)
    s = [[0 for y in range(yn)] for x in range(xn)] # create an empty matrix
                                                   #from lengths of strings
    for a in range(xn): # set first row and column as the base case.
        s[a][0] = a * insert
    for b in range(yn):
        s[0][b] = b * insert
    for i in range(1, xn): #iterate through ever letter of both strings
        for j in range(1, yn):
            if string1[i] == string2[j]: #same letter; same number as diagonal
                s[i][j] = s[i-1][j-1]
            else:   # not same letter so must perform operation
                s[i][j] = cost(s, i, j)
    return s
```

`extractAlignment` is a function that determines the optimal set of operations to perform by starting from the bottom right corner of the matrix and iterating back to the base cases. It does this by calling the `determineOptimalOps` function which takes a position in the matrix and finds the preceding position that, through most optimal operation, led to the current position. Using the same operations as above, the function creates a list of the optimal operations then chooses from those randomly to return to `extractAlignment`. The algorithm then calls `updateIndicites` to change the [i][j] coordinates based on the

operation has been passed, using the same logic for the direction in the `alignStrings` function. This process repeats until i = 0 or j = 0, meaning the algorithm has reached the end of one of the original strings. `extractAlignment` then returns a list of optimal operations to perform to string1.

```
def determineOptimalOps(matrix, x, y, i, j):
    current = matrix[i][j] # position of matrix at that location
    up = 99999
    left = 99999
    diag = 99999
    superdiag = 99999
    ops = []
    if i > 0:
        up = matrix[i-1][j] # delete operation
        if up + delete == current:
            ops.append("delete")
    if j > 0:
        left = matrix[i][j-1] # insert operation
        if left + insert == current:
            ops.append("insert")
    if i > 0 and j > 0:
        diag = matrix[i-1][j-1] # subsstitution
        if diag + sub == current:
            ops.append("sub")
    if i > 1 and j > 1:
        superdiag = matrix[i-2][j-2] # swap
        if superdiag + swap == current:
            ops.append("swap")
    if len(ops) == 0: # if list is empty, then no operations were performed
        return "no_op"
    optimal = random.choice(ops) #randomly chooses one of the optimal ops
    return optimal

def updateIndicies(operation, i, j):
    if operation == "no_op": #no op is always diagonal
        i-=1
        j-=1
    elif operation == "delete": #moves horizontally
        i-=1
    elif operation == "insert": #moves vertically
        j-=1
    elif operation == "sub": #moves diagonally
        i-=1
        j-=1
    elif operation == "swap": #moves diagonally two places
        i-=2
        j-=2
    return i,j


def extractAlignments(matrix, x, y):
    a=[] # empty list to store optimal operations
    xlength = len(x) -1
    ylength = len(y) -1
```

```
    i = xlength
    j = ylength
    while (i > 0 and j > 0):
        op = determineOptimalOps(matrix, x, y, i, j) # determine operation
        a.insert(0, op) # reads matrix from bottom
                        # must insert at beginning of list
        i,j = updateIndicies(op, i, j) #updates the i,j position
    while i != 0: # When at base case, moves to the matrix[0][0] position
        a.insert(0, "delete")
        i-=1
    while j != 0:
        a.insert(0, "insert")
        j-=1
    return a # returns list of optimal operations
```

The last algorithm `commonSubstrings` takes the list of optimal operations and string1 to print all of the occurrences where string1 and string2 match. The algorithm converts string1 into a list that can be iterated through, then begins iterating through a (the list of optimal operations). Because we only want to add strings that are of length L or more, the algorithm traverses through a and adds the letter at the corresponding position to a substring whenever a "no$_op$" $is seen. This process of adding letters to the substring continues for all sequential$ "no_ops" $and stops su$ "no_op" $operation. The algorithm the compares the substring to see if it is at least length L, and if so, is added to a list to$

```
def commonSubstrings(x, L, a):
    x = list(x)   #converts string1 into a list that can be traversed
    substr = ""
    substrList = []   #list of substrings
    for i in range(len(a)):
        # print ("i is", i)
        if (a[i] == "no_op"):  #add to substring
            substr = substr + x[i]
        else:
            if substr != "":   #reset substring, and add to appendix
                if len(substr) >= L:   #only add if string >= L
                    substrList.append(substr)
                substr = ""   #reset the substring to
            if (a[i] == "insert"):   #add extra letter if "insert"
                                     #because string is 1 letter longer
                x.insert(0, "_")
    if substr != "":
        if len(substr) >= L:   #if letter added on last iteration
            substrList.append(substr)
            substr = ""
    return substrList
    \end{verbatim}
```

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

The asymototic behavior of `commonSubstrings(x,L,extractAlignment(alignStrings (x,y), x,y))` is $\mathcal{O}(mn)$, where m is the length of string1 and n is the length of string2.

Note that mn is also the size of the matrix. To begin, we will start by analyzing `alignStrings`. This function creates an empty m*n matrix, assigns the base cases along the first column and first row (costing m and n respectively) then iterates through both strings in a nested for loop to populate that matrix. Both of these processes involve mn operations. Inside of the nested loop is a call to `cost()`, but that function has no loops or recursion, so it has a constant run time. Overall, `alignStrings` is $\mathcal{O}(mn)$. Moving to `extractAlignment`, the purpose of the function is to find the most optimal (shortest) path from (0,0) of the matrix to (m,n). To find the asymptotic behavior, we must look at the worst case behavior. As the run time of the function is determined by the number of elements it traverses, the worst case behavior would be moving entirely horizontally until i=0, then moving entirely vertically rather than ever moving along a diagonal. The horizontal length of the matrix is n and the vertical size is m, so the overall worst case behavior is $\mathcal{O}(mn)$. The two helper functions `determineOptimalOps` and `updateIndicies` both have constant run time so they do not impact the overall complexity of `extractAlignment`. Note that in this worst case scenario, `extractAlignment` would return a list of optimal operations a that has a length of mn. Moving to the final function, `commonSubstrings`, we can see that there is a single loop in the function that iterates through the entire length of a. We determined that a has a length of mn, and all processes within the loop are atomic (has constant run time), therefor `commonSubstrings` also runs for mn time. Overall, the run time of the algorithm is $T(n) = cmn$, which has an asymptotic behavior of $\mathcal{O}(mn)$.

(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.

The algorithm (we can call it `totalOptimalAlignments`) would start at the [m,n] position of S and act similarly to the `extractAlignments` function, moving through the matrix to find the most optimal path of operations. The difference between the two functions is that `extractAlignments` iterated through the path until it hit a base case, whereas `totalOptimalAlignments` would perform a Breadth First Search on the matrix. When faced with two eqaully optimal operations, `extractAlignments` would choose one randomly and continue iterating but `totalOptimalAlignments` would split at that junction, adding all of the possible optimal paths. There would be some counter to measure how many times a path reached the base case, and that would be what is returned from the function.

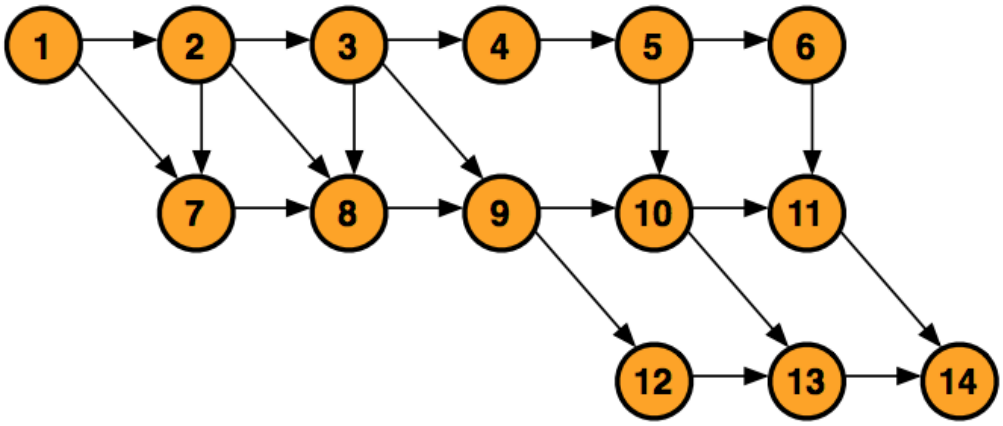
(d) String alignment algorithms can be used to detect changes between different versions of the same document (as in version control systems) or to detect verbatim copying between different documents (as in plagiarism detection systems). The two `data_string` files for PS7 (see class Moodle) contain 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.

Using the two data_string files with $L = 10$, the algorithm returns the shared substrings:

manufacturing ', ' gulf coast ', 'the company's ', ' a keynote speech today ', 'conference', ',"
said woods. "', 'mobil is strategically investing in new refining and chemical-manufacturing projects in the u', ' gulf coast region to expand its manufacturing and export capacity. the company's growing the gulf ', ' consists of 11 major chemical, refining, lubricant and liquefied natural gas projects at proposed new and existing facilities along the texas and louisiana coasts. investments began in 2013 and are expected to continue through at least 2022. ', 'facturing '

All of these substrings are strings in x that also appear in y. Some of them are explainable, such as the single words like 'conference' and 'manufacturing.' However, there are matching substrings that are far too long to be coincidence, such as ' consists of 11 major chemical, refining, lubricant and liquefied natural gas projects at proposed new and existing facilities along the texas and louisiana coasts. investments began in 2013 and are expected to continue through at least 2022. ' This string, and some others of similar length, show that some parts of x where copied from y and could reasonably be considered plagiarism.

2. Ginerva Weasley is playing with the network given below. Help her calculate the number of paths from node 1 to node 14. Hint: assume a "path" must have at least one edge in it to be well defined, and use dynamic programming to fill in a table that counts number of paths from each node j to 14, starting from 14 down to 1.



Node	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Preceding Nodes	–	1	2	3	4	5	1, 2	2, 3, 7	3, 8	5, 9	6, 10	9	10, 12	11, 13
Paths to Node	1	1	1	1	1	1	2	4	5	6	7	5	11	18

The graph is showing the dynamic programming approach where the number of paths to each node is equal to the sum of the number of paths to its preceding nodes. By doing this, we store the information of each node (sub-problem) and use that to solve for following nodes.

From the graph we can see that there are a total of 18 paths to the 14^{th} node.

3. 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) }
```

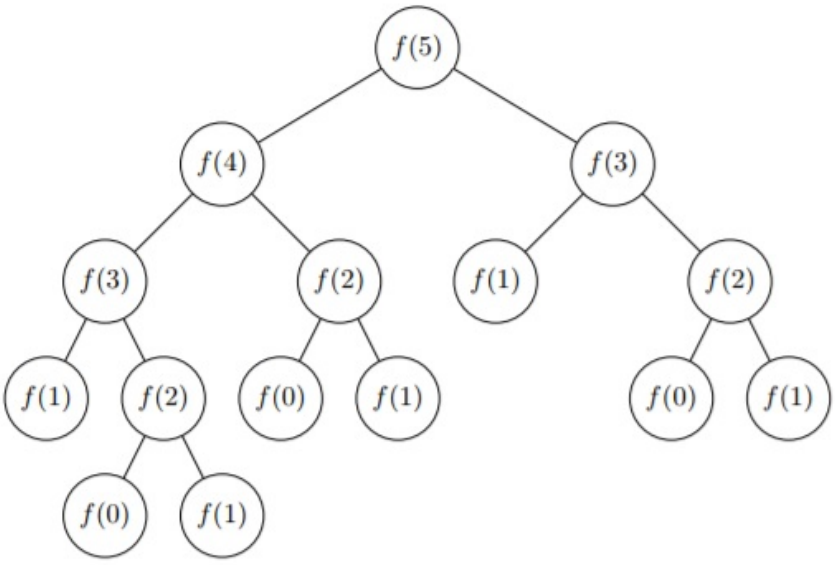
which he claims takes $R(n) = R(n-1) + R(n-2) + c = O(\phi^n)$ time.

(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.

```
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]
```

i. Describe the behavior of `MemLuc(n)` in terms of a traversal of a computation tree. Describe how the array L is filled.

In terms of a computation tree, a recursive call is the root of a computational tree and each child is the n-1 and n-2 Lucas numbers. Eventually, each child will lead to the leaves which are the bases cases, $L_0 = 2$ and $L_1 = 1$. This is demonstrated in the figure bellow.



The array L is filled in a similar way, where each item in array depends on the two items behind it. The difference is that the values are stored in the array so the algorithm does not have to traverse the entire array each time.

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

The asymototic running time of `MemLuc` is $\mathcal{O}(n)$. Assume that there are no values in array L. The L[n] position will then call the L[n-1] and L[n-2] positions. L[n-1] will be executed n-1 times, one call to fill each position of the array up to L[n-1]. L[n-2] will then take the value out of the array. The total run time would then be $(c(n-1)) + (c) = cn$, so the asymptotic run time is $\mathcal{O}(n)$.

Base Cases:

- n = 0. The algorithm returns 2. Intended behavior.
- n = 1. The algorithm returns 1. Intended behavior.
- n = 2. $n_2 = n_1 + n_0 = 1 + 2 = 3$. Returns 3.

Induction Hypothesis: Assume that for all natural numbers less than or equal to k, `MemLuc` returns the k^{th} Lucas number, where $L(k) = L(k-1) + L(k-2)$.
Inductive Step: Consider the number k+1: $L(k+1) = L(k) + L(k-1)$
Case 1: $L[k+1]$!= undefined. The algorithm returns L[k+1], as intended, which is the $k+1$ Lucas number.
Case 2: $L[k+1]$ = undefined. The The algorithm calculates L[k+1] by recursively calling `MemLuc(k)` and `MemLuc(k-1)`. These will continue to be called until L[k-q] = defined, where q is the number of times it was recursively called.

(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 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 DynLuc(n) algorithm functions by iterating from 2 to n and solving each lucas number along the way and storing that answer in an array. Because of this, the algorithm is $\mathcal{O}(n)$ in both space and time complexity because it takes n operations to reach the $n^{th} Lucas Number$.

(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
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 code is that the function returns a, which is the n-2 Lucas Number, where it should return c, the actual calculated Lucas Number for n. The time usage is $\mathcal{O}(n)$ because it uses a single loop to iterate from 2 to n. The space usage is $\mathcal{O}(1)$ because it does not store the answers to the previous Lucas numbers, so it only uses the 3 variables, which is constant space

complexity.

(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?)

Algorithm	Time Complexity	Space Complexity	Data Structures
Luc(n)	$\mathcal{O}(2^n)$	$\mathcal{O}(n \log n)$	Stack
MemLuc(n)	$\mathcal{O}(n)$	$\mathcal{O}(n^2 \log n)$	Array, Stack
DynLuc(n)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	Array
FasterLuc(n)	$\mathcal{O}(n)$	$\mathcal{O}(1)$	None

The table shows how each following algorithm improves on the previous, either in time or complexity. MemLuc improves on Luc by implementing an array to store the already calculated Lucas numbers so it does not have to calculated each one for each call. However, MemLuc uses an array and the stack which involves a lot of data, so the other algorithms improve on the space complexity rather than the time. DynLuc uses dynamic programming instead of recursion to avoid using the stack and FasterLuc uses variables instead of any data structure to run with constant space complexity.

(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`.

The first 5 digits of the 0617^{th} Lucas number is 88180. Assuming it takes n operations to find the n^{th} Lucas number with `FasterLuc`, then it takes 617 nanoseconds. With the original `Luc` function, it would take 2^{617} operations or $5.139 * 10^{185}$ nanoseconds to complete. This is the same as $1.725 * 10^{178}$ years! It takes $8.815 * 10^{182}$ times longer to calculate the 617^{th} Lucas number with `Luc(n)` than it does with `FasterLuc(n)`.