There are 75 regular and 20 extra credit points available on this problem set.

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.

   (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.

   Code for `alignStrings`, `extractAlignment`, and `commonSubstrings` is at the end of these solutions.

   The implementation of `alignStrings` first allocates the optimal cost matrix S with $(n_x+1)\times(x_y+1)$ zeros, and then fills in the base case costs for aligning either all of x with an empty y (first row), or all of y with an empty x (first column). These costs are all indel costs. It then enters two nested `for` loops, which fills in the table one row at a time. For each entry, it computes the cost of each of the four operations (except if we cannot do a swap yet) using the given cost array. It then records the smallest cost value in the element and proceeds.

   The implementation of `extractAlignment` starts at the bottom-right corner of S, and then computes the costs that each of the four operations would have proposed adding to their corresponding subproblem to get to the current entry of S. Each of these calculations has to check for boundary conditions (which would prevent this operation from being valid). The minimum costs are then extracted and ties are broken uniformly at random, the selected operation pre-pended to the list of optimal choices, and finally the indices are updated to reflect the location of the chosen subproblem we built off of. One detail: in order to denote a sub operation that was a "no-op," we multiply the op index by -1.

The implementation of `commonSubstrings` steps through the list of optimal operations until it finds a no-op. It marks that location, and then steps forward until it reaches a non-no-op. If the length of that run is at least L, then it writes out those characters, starting at the mark and ending just before the non-no-op. The main loop index is then updated to restart at the location of the non-no-op.

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

`alignStrings` takes $O(n_x n_y)$ time, because we perform $O(1)$ operations to fill in each entry of S and the size of S is $O(n_x n_y)$.

`extractAlignment` takes $O(n_x + n_y)$ time, because the worst-case alignment is to delete one string and insert the other, corresponding to a backwards trace through S in which we first traverse every row, followed by every column, or vice versa.

Finally, `commonSubstrings` takes $O(n_x + n_y)$ time, because that is the maximum number of operations in an optimal alignment.

Hence, the call's running time is dominated by `alignStrings`, and is $O(n_x n_y)$. Optionally, if we define $n = \max(n_x, n_y)$, then the running time is $O(n^2)$.

(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.*

Recall that the alignment algorithm fills in the optimal cost matrix S according to a directed acyclic graph (a DAG), in which a pair of elements $S[i, j]$ and $S[\ell, k]$ are connected by a directed edge if the optimal cost at $S[i, j]$ can be derived from the optimal cost at $S[\ell, k]$ plus one of the four edit operations. Hence, every optimal alignment represents a path from $S[0, 0]$ to $S[n_x, n_y]$ on this DAG.

In lecture, we sketched a dynamic programming algorithm for counting the number of paths on a DAG. This algorithm can solve our problem here. We could translate the matrix S and its implicit DAG into an actual graph $G = (V, E)$, but this requires a somewhat complicated indexing scheme, mapping the $n_x n_y$ elements in S onto a one-dimensional adjacency list for V. Working with S directly is slightly less complicated, and consumes the same amount of time, asymptotically.

In `extractAlignment`, we use a series of conditionals to compute the costs for

each of the four possible edit operations we considered before assigning a value to some $S[i, j]$, with the goal of determining which of them were optimal choices. For simplicity, we now abstract those conditionals into a single function `optChoice` that takes as input the indices i, j of our current location in S and a given operation name, and returns a boolean value depending on whether that operation corresponded to an optimal choice or not.

A dynamic programming algorithm that counts the number of optimal paths is the following recursive function, which uses a table X to memoize the answers, where `NumPaths(0,0)=1` is the base case:

```
NumPaths(i,j) {
   if i == j == 0 { return 1 }  // base case
   else {
      if (X[i,j] == undefined) {
         temp = 0
         if optChoice(i,j,xdel) { temp += NumPaths(i-1,j)   }
         if optChoice(i,j,ydel) { temp += NumPaths(i,j-1)   }
         if optChoice(i,j,sub)  { temp += NumPaths(i-1,j-1) }
         if optChoice(i,j,swap) { temp += NumPaths(i-2,j-2) }
         X[i,j] = temp
      }
      return X[i,j]
}}
```

Correctness follows by strong induction. If $i = j = 0$, there is only one optimal alignment (aligning two empty strings) and `NumPaths` correctly returns 1. Now, we assume that the algorithm works for all intermediate values $i = \ell > 0$ and $j = k > 0$, meaning that $X[\ell, k]$ gives the correct number of optimal alignments for subproblems $0..\ell$ and $0..k$. Now, consider $i = \ell+1$ or $j = k+1$. In either case, $X[i, j]$ is undefined, and we enter the inner conditional. We then check which of the four already-computed subproblems was an optimal choice for aligning i, j, and sum their corresponding values, which we calculate recursively. On a DAG, the number of paths between a pair of nodes $a, b \in V$ is defined as the sum of the number of paths from a to each $c \in V$ such that $(c \to b) \in E$. Hence, each of these recursive calls refers to an element of X that is already computed optimally, and thus $X[i, j]$ is computed optimally.

The running time depends on how many elements in X are filled in, and how many times each of those elements is referred to by subsequent calls. Every element is referred to by at most a constant number of calls, and at worst, we fill every element. Thus, the running time is given by the size of the table, $O(n_x n_y)$.

(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).*

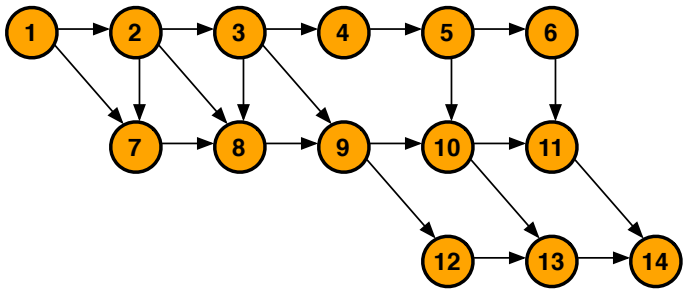
The two `data_string` *files for PS6 (see class Moodle) contain actual documents recently released by two independent organizations. Use your functions from* (**??**) *to align the text of these two documents. Present the results of your analysis, including a reporting of all the substrings in x of length $L = 10$ 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.*

These are fairly long strings to align (several thousand characters long, each), and so the alignment takes some time. One optimal alignment finds 11 overlapping strings of 10 characters or more:

	L	substring
1	14	`'manufacturing'`
2	12	`' gulf coast '`
3	17	`' the companys '`
4	24	`' a keynote speech today '`
5	10	`'conference'`
6	20	`', said woods.  '`
7	93	`'mobil is strategically investing in new refining and chemical-manufacturing projects in the u'`
8	100	`' gulf coast region to expand its manufacturing and export capacity.  the companys growing the gulf'`
9	242	`' 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.'`
10	11	`' to create '`
11	10	`'facturing '`

The three longest (7, 8, and 9 above) all come from the same paragraph in x and represent a near-verbatim copy from a paragraph in y. As such, this would qualify as gross plagiarism under the CU's academic honesty policy.

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.*



A simply dynamic programming approach works well, using a table to store or "memoize" the intermediate results. The number of paths from node i to node 14 equals the sum of the number of paths to node 14 at each of the nodes pointed to by i. Thus, we start calculating from node 14 and, in this way, trace back to node 1.

For example, the number of paths from node 9 to node 14 *equals* the number of paths from node 10 to node 14 *plus* the number of paths of node 12 to node 14. Applying this rule recursively, allows us to populate the table that allows us to build up the correct answer from intermediate results:

node i	14	13	12	11	10	9	8	7	6	5	4	3	2	1
num. paths from i to 14	0	1	1	1	2	3	3	3	1	3	3	9	15	18

Hence, the number of paths from node 1 to node 14 is 18.

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.*

```
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.*

Each time `MemLuc(k)` is called, two recursive function calls are placed on the stack, one for `MemLuc(k-1)` and one for `MemLuc(k-2)`. If we assume these are explored in-order, then the computation tree's structure is logically equivalent to a depth-first search tree and the `MemLuc(k-1)` branches are evaluated before the `MemLuc(k-2)` branches. The in-order traversal thus reaches a base case after repeated calls to `MemLuc(k-1)` until $k < 2$.

When this happens, the stack contains $\Theta(n)$ frames. As frames are popped off the stack, we begin storing values in the $L[n]$ array. After three calls to the base cases (to store $L[2]$ and $L[3]$), all subsequent branches find that their subproblems have already been computed and thus no new recursive calls are made. Structurally, this means that an internal node in the computation or recursion tree at depth i has a left-subtree of size i and a right-subtree of size 1. That is, once the base cases have been entered into the array, each subsequent entry $L[i]$ is filled simply by combining $L[i-1]$ and $L[i-2]$. As a result, the entries in L are filled in order, from $i = 1$ to n.

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

The running time is the number of steps the in-order traversal takes to walk the tree from top to bottom and back. Each internal node is visited twice,

once on the way down and once on the way back up, while leaf nodes are visited only once. There are $n-1$ internal nodes and n leaf nodes, so the running time should be $2(n-1)+n = \Theta(n)$. We now prove this by induction.

We argued above that the values in the $L[n]$ array are filled from left (small n) to right (large n) by an in-order traversal of the recursion tree. $L[3]$ is the first value stored, by the `MemLuc(3)` call, whose children are base cases for Lucas numbers, and thus functions as the base case for our induction. Without loss of generality, we assume that $L[0]$ and $L[1]$ have also been filled, because calls `MemLuc(n)` for $n < 2$ are base case calls, and function like accessing an element in the L array (constant time cost).

Assume the values $L[j]$ for $0 \leq j < k$ have been filled. When we encounter `MemLuc(k)` on the stack, by assumption $L[k]$ is undefined and thus we make two recursive calls: `MemLuc(k-1)` and `MemLuc(k-2)`. In each case, the inner conditional fails because $k-2 < k-1 < k$ and thus those entries in L are not undefined. When these return, we make a single addition operation and store the result in $L[k]$.

Thus, for each value stored in L, we make a single addition operation and by induction the number of additions is the size of L, which is $\Theta(n)$. □

(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* (**??**)*.*

Ron is wrong about the running time, which is still $\Theta(n)$. This fact can be seen immediately by the size of the `for` loop, as each pass through the loop takes constant time (1 addition, 1 assignment and 2 array access calls; all atomic operations).

Ron is also wrong about the space, which is still $O(n)$. This version takes less space than `MemLuc()`, but not asymptotically so. In the previous version, the size of the stack in the recursive version is $O(n)$ and the L array takes $\Theta(n)$ space. `DynLuc()` takes less space because it eliminates the space used by the stack, but because it still uses the array L, omitting the stack only reduces the memory usage by a constant factor.

(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 bug is the line `return a`, which should instead be `return b`. The error means that the algorithm returns L_{n-1} rather than L_n.

The running time is $\Theta(n)$, which again can be seen by the size of the `for` loop, as each pass through the loop takes constant time (1 addition and 2 assignments; all atomic operations). The space requirement is now only $\Theta(1)$ because we use only three atomic-sized scalar values to store the previous results.

(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.*

Looking back at the different versions, the following table compares the running

times, space requirements and data structures used:

algorithm	time	space	data structures
`Luc(n)`	$\Theta(\phi^n)$	$\Theta(n)$	stack
`MemLuc(n)`	$\Theta(n)$	$\Theta(n)$	stack, array
`DynLuc(n)`	$\Theta(n)$	$\Theta(n)$	array
`FasterLuc(n)`	$\Theta(n)$	$\Theta(1)$	scalars

The improvement from `Luc(n)` to `MemLuc(n)` lies in the running time; the improvement from `MemLuc(n)` to `DynLuc(n)` is only in simplifying the data structure used (and reducing the constant in the running time, since we don't have to deal with stack frames, etc.). The improvement from `DynLuc(n)` to `FasterLuc(n)` lies in the space requirement, as we dispense with the array and use only a few scalars (this will also improve the running time, but only by a constant factor).

Comparing `FasterLuc(n)` to `Luc(n)`, we see a huge improvement in both the running time and the space usage. This makes it realistic to compute essentially arbitrarily large values of `Luc(n)`, which was not possible with the original exponential-time algorithm.

(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`*.*

Let's use an (arbitrarily chosen) birthday of 1 July 2017, which in MMDD is 0701. `FasterLuc(701)` yields $3.164723815335666 \times 10^{146}$, so the first 5 digits are 31647. The classic recursive algorithm would take ϕ^{701} nanoseconds, where $\phi = (1 + \sqrt{5})/2 \approx 1.61803399\ldots$ is the golden ratio. There are 3.1536×10^{16} nanoseconds in a year, which seems like a lot, but it would still take `Luc(701)` an astounding 10^{130} years to complete, compared to the milliseconds it took for `FasterLuc(701)`. Hence, dynamic programming for the win.