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 13 (plus the cost of the two *sub* ops), and the cost of *sub* be 12, 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 list of edit operations
   [i,j] = [nx,ny]         // initialize the search for a path to S[0,0]
   while i > 0 or j > 0
        a.prepend(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).

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

(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.

(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 $L = 9$ 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.

2. (20 pts) Ron and Hermione are having a competition to see who can compute the nth Pell number P_n more quickly, without resorting to magic. Recall that the nth Pell number is defined as $P_n = 2\,P_{n-1} + P_{n-2}$ for $n > 1$ with base cases $P_0 = 0$ and $P_1 = 1$. Ron opens with the classic recursive algorithm:

```
Pell(n) :
   if n == 0 { return 0 }
   else if n == 1 { return 1 }
   else { return 2*Pell(n-1) + Pell(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 Pell numbers by storing them in an array `P[n]`. She claims this allows an algorithm to compute larger Pell numbers more quickly, and writes down the following algorithm.[1]

```
MemPell(n) {
   if n == 0 { return 0 } else if n == 1 { return 1 }
   else {
      if (P[n] == undefined) { P[n] = 2*MemPell(n-1) + MemPell(n-2) }
      return P[n]
   }
}
```

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

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

(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 P array in order. Ron's new algorithm[2] is

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

Determine the time and space usage of `DynPell(n)`. Justify your answers and compare them to the answers in part (2a).

[1] Ron briefly whines about Hermione's `P[n]=undefined` trick ("an unallocated array!"), but she point out that `MemPell(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 `MemPell(n)` as given.

[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.

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

```
FasterPell(n) :
   a = 0,  b = 1
   for i = 2 to n
      c = 2*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 `FasterPell(n)`. Justify your claims.

(d) In a table, list each of the four algorithms as columns 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?)

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