CSCI 3104 Spring 2018 Problem Set #7
Alison Ostlund (05/22)
Worked with: Kyle Ward, Jason Lubrano, Vamshi Arugonda, Hannah Haines, Denver Cherin, and Matt Donovan

---

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 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).

For **alignStrings** I created a two dimensional array, which holds my cost matrix. Then using for loops I used the base cases in the array. Using these, I made a function called cost(x,y,i,j,S) which takes in my cost matrix S, two strings, and the current position in matrix. This calculates the minimum cost for a certain index using the surrounding values in the cost matrix. For swap the algorithm checks if it is out of bounds by checking if i and j are both greater than 2. Then the algorithm calculates the cost for each edit op, sub, indel, and swap using the current index and the given costs of each op. From there the minimum cost of these is returned to the cost matrix current position.

extractAlignment takes in the two strings and the main array. The algorithm initialize the optimal array, which is what will be returned. The two lengths of the strings are put into the indices of i and j, which will then be used in a while loop

to loop through and use the determineOptimalOp(S,i,j,x,y) which determines all the optimal operations working up from the most optimal at [i,j]. Then this set of operations is reversed because it is working up, and needs to be reversed to match the order asked in the write up.

commonSubstrings takes in the optimal array a, string x, and L, which is equal to one or equal to the length of x. The the algorithm iterates through based on what the operation is. If it hits a NOOP, it will move to the next op, else it will find which edit op is used. If reach the end of our string length,then the algorithm will append the string to common substrings array.

(b) Using asymptotic analysis, determine the running time of the call **commonSubstrings**(x, L, extractAlignment (alignStrings(x,y), x,y)) Justify your answer.
extractAlignment = O(nx * ny) **extractAlignment** = O(nx+ny) **commonSubstrings** = O(nx+ny) The running time for the algorithm is $\mathcal{O}(n^2)$. This is because alignStrings is $O(nx * ny)$ extractAlignment is $O(nx + ny)$, and the runtime for commonSubstrings is $O(nx + ny)$. Using this, can calculate the total running time for commonSubstrings(x, L, extractAlignment(alignStrings(x,y), x,y)) is $O(nxny)$ because the $O(nx + ny)$ functions will become insignificant.

(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.

idk.

(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.

i dont know

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 $P_n = 2P_{n-1} + P_{n-2}$ 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(ϕn) time

(a) Hermione counters with a dynamic programming approach that memorizes (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.

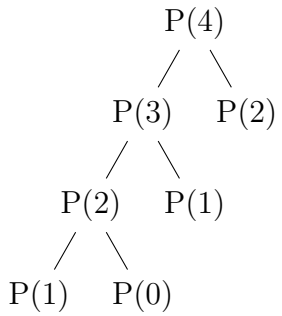
```
MemPell(n) {
  if n == 0 { return 0 } else if n == 1 { return 1 }
  else {
    if (P[n] == undefined) { P[n] = 2*MemPell(n-1) + MemPell(n-2) }
    return P[n]
  }
}
```

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

In terms of a traversal tree, MemPell(n) starts with the nth element in the array. This will become the parent node of the tree. From there the tree will traverse down looking at the n-1 and n-2 element. These elements are not yet solved in the algorithm so we will traverse down the tree until we reach a known element, aka the base case of p0 = 0 or p1 = 1. So when n-1 = 1 or 0 we will stop traversing down because the base case is known and start traversing up while solving the algorithm and filling the array. Having the left node be the n-1 branch and the right node being the n - 2 branch. Traversing up we can ignore any nodes and filling he array until we reach n the parent node, and therefore the array is completely filled.Getting through the array is $O(n)$.

Therefore by the way the tree is traversed, the array will be filled linearly starting with the base cases of p0 = 0 and p1 = 1, so will continue filling the

array at p2 = 2.
So for Example when n=4:



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

The asymptotic running time is O(n). This is because the tree will have to have at most have to traverse the tree fully, and that is at most n time. So the algorithm will have to compute the Pell numbers as many times in P(n), which is n times. And the space is also O(n) because this accounts for the call stack and assigning each element into the array, which is n times.
Base Case: we know p0 = 0, p1 = 1, therefore these will already be known in the array. So we can say that $T(n) \leq n$

Induction Step: When n != 1 and 0, so n > 1 we have the stored values of the base case. So n-1, and n-2 is known, n is not. When traversing up the tree, the elements in the array are filled in ascending order. So when tying to find n, the algorithm is dependent on the fact that n-1 and n-2 are known elements.While traversing up the tree we are storing the values calculated in the array, meaning when we are looking at the ith element we have the i-1 element and i-2 element stored for our use. Meaning as we traverse the algorithm we have a time complexity of O(n) because we will use i-1 and i-2 at most n times.

(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

```
DynPell(n) :
  P[0] = 0, L[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).

DynPell makes use of a for loop which will produce a complexity of n while the rest of the algorithm uses atomic operations with a constant time of O(1).Therefore n+ O(1) = O(n). Giving DynPell a time complexity of O(n). DynPell uses a space O(n) since the array P has at most n values that are populated.

Where as in 2a the algorithm there uses a time of O(n) because it goes through the array and/or recurses down the tree at most n times. Because recurrence is utilized in the algorithm, the tee has a depth of 2^n giving a space complexity of $O(2^n)$. Which is a lot more space usage and less optimal than DynPell's space complexity although they have the same time complexity.

(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.

Errors:

For loop should go from range 2 to n+1. When going from just n will not account for the actual value you are searching for dude to the recurrence in the equation. Using just n will return the value for n-1.

Return c instead of a. When returning a you are returning the n-1 value, rather than the n value. To fix this we return c since c is (P[n]) which is the value n we are calculating.

(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?)

Algorithms	Running time	Space	Data Structs
Pell	$O(\phi^n)$	$O(n)$	Stack
MemPell	$O(n)$	$O(n)$	Stack and Arrays
DynPell	$O(n)$	$O(n)$	Arrays
FasterPell	$O(n)$	$O(1)$	None

(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.
My birthday is 0522 so pass in value of 522

```
def FasterPell(n):
    a = 0
    b = 1
    for i in range(2, n+1):
        c = 2 * b + a
        a = b
        b = c

    return c

FasterPell(522) # 0522 doesn't work so used 522
```

First 5 numbers of return value: 22769.....
The classic recursive algorithm takes O(ϕ^n) time therefore will ϕ^n nanoseconds so in order to find the years, plug 522 for n and computer ϕ^{522} which gives a large number of 1.234662e+109 nanoseconds which is needed to be converted into years, which gives the value of 3.91508751902587501e+92 years. When using FasterPell, we use the time complexity of O(n) so we use n operations. Giving a MUCH faster time complexity of 522 nanoseconds.