CSCI 3104 Spring 2018 Problem Set 7
Mohamed Al-Rasbi (07/25)

Problem 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, youll need to use unit costs instead of the ones given above).*

`alignStrings(x, y)`:
First I find the length of string x(which is the number of columns in the cost matrix) and the length of string y(which is the number of rows in the cost matrix).
I create the cost matrix S, which is a two dimensional list in Python, and initialize the 0th column and the 0th row such that $S[0][j] = i$ and $S[i][0] = i$. And then we go through each element in the matrix and find the minimum cost by calling `cost(S, i, j, x, y)` function.
In the cost function, we find the minimum value of the four operations and append it to the matrix.

`extractAlignment(S, x, y)`:
The function starts from the last element in the bottom right of the cost matrix to find the optimal operations by using `determineOptimalOp(S, i, j, x, y)`.
In that function we find the cost of each operation, and find the minimum value. If we have multiple minimum values, one of them will be returned randomly.
Depending on that operation, the indices will be updated. The same thing will happen until we reach the index (0, 0) and then the list of operations will be returned.

`commonSubstrings(x,L,a)`:
We want to return each substring from x that aligns exactly to a substring in y. We do that by going through the operation in the list a.
I want to note that I implemented the code such that the first operation in a will be applied to the last letter in x, and the second will be applied to the last-1, and so on.
By doing that and adding some more if statement(as you can see in the code), we will get a list of substrings that are at least of the length L.

```
def cost(S, i, j, x, y):

    #cost of each operation
    sub = 12 + S[i-1][j-1]
    indelup = 1 + S[i-1][j]
    indelleft = 1 + S[i][j-1]
    minCost = float("inf")

    #check if we can use the swap operation
    if(i >=2 and j >=2):
        swap = 37 + S[i-2][j-2] #37=13+12+12
        minCost = min(sub, indelup, indelleft, swap)
    else:
        minCost = min(sub, indelup, indelleft)

    if(x[j-1] == y[i-1]):
        equal = S[i-1][j-1]
        minCost = min(minCost, equal)


    return minCost


def alignStrings(x, y):

    cols = len(x) + 1
    rows = len(y) + 1

    #fill matrix S with zeros
    S = [[0 for i in range(cols)] for j in range(rows)]
    #initilize S
    for j in range(cols):
     S[0][j] = j
    for i in range(rows):
     S[i][0] = i

    for j in range(1, cols):
     for i in range(1, rows):
     S[i][j] = cost(S, i, j, x, y)

    return S



def determineOptimalOp(S, i, j, x, y):
    #check if no-op
    if(x[j-1] == y[i-1]):
     return "no-op"

    #initilize values
    sub, indelup, indelleft, swap = float("inf"), float("inf"), float("inf"), float("inf")
    if(i >= 2 and j >= 2):
        sub = S[i-1][j-1]
        indelup = S[i-1][j]
        indelleft = S[i][j-1]
        swap = S[i-2][j-2]
    elif(i >= 1 and j >= 1):
        sub = S[i-1][j-1]
        indelup = S[i-1][j]
        indelleft = S[i][j-1]
    elif(i >= 1):
        indelup = S[i-1][j]
    elif(j >= 1):
        indelleft = S[i][j-1]


    #choose the optimal operation(s)
    opsVal = [sub, indelup, indelleft, swap]
    opsName = ["sub", "indelup", "indelleft", "swap"]
    chosenOps = []

    minVal = min(opsVal)
    for i in range(4):
        if(opsVal[i] == minVal):
            chosenOps.append(opsName[i])

    return random.choice(chosenOps)

def updateIndices(S, i, j, operation):
    #depending on the operation, indices are updated
    if(operation == "swap"):
        return (i - 2, j - 2)
 elif(operation == "indelup"):
        return (i - 1, j)
 elif(operation == "indelleft"):
        return (i, j - 1)
 else:
 #sub or no-op
 return (i - 1, j - 1)

def extractAlignment(S, x, y):
 a = []
 i = len(y)
 j = len(x)
 while i > 0 or j > 0:
 #find the optimal operation
 operation = determineOptimalOp(S, i, j, x, y)
 a.append(operation)
 print (i, j)
 (i, j) = updateIndices(S, i, j, operation)

 return a


def commonSubstrings(x, L, a):
 #the list of substrings of length at least L
 substrings = []
 sub = ""
 index = len(x) - 1
 for i in range(len(a)):
 if a[i] == "no-op":
 sub += x[index - i]
 else:
 if len(sub) >= L:
 substrings.append(sub[::-1]) #reverse the string
 sub = ""

 return substrings
```

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

`alignStrings` fills the cost matrix by performing $O(1)$ operations. The dimension of the cost matrix is $x.y$, so we can say it takes time $O(n_x n_y)$
`extractAlignment` run through the elements in the cost matrix to find the optimal operations. The worst case is to delete one string and insert the other, which is traversing every row and column. And that is $O(n_x + n_y)$
`commonSubstrings` is also $O(n_x + n_y)$, since that is the maximum number of operations in strings alignment.

We can see that the function `alignStrings` has the worst running time. Hence, the running time of this call is $O(n_x n_y) = O(n^2)$

(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.*

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

We can modify the function `extractAlignment` so we can find the number of optimal alignments. We use the function `determineOptimalOp` to return the optimal operation (randomly in case we find more the one operation that has the same cost).
So, this is me idea about an algorithm that can find the number of optimal alignments:
1. Define $i = len(y)$ and $j = len(x)$
2. The base case is $i == 0$ and $j == 0$ return 1. Because if we compare two empty strings, there is only one optimal alignment.
3. Modify the function `determineOptimalOp` so it returns a list of the optimal operations (the operations that have the same minimum value) instead of just returning one operation.
4. Depending on that operation, we update the value of i and j, and then call the function recursively on each operation.
5. We do that until we reach the base case.

`Prove the algorithm is correct: Using induction`
`[Base case]:` If i and j equal 0, then there is only one optimal alignment, which is comparing two empty strings. And the function correctly returns 1.
`[Induction step]:` We assume the algorithm will give us the number of optimal solutions when $i = a > 0$ and $j = b > 0$. Now, consider $i = a + 1$ and $j = b + 1$. The function `determineOptimalOp` will be called, then the values of i and j will be decreased by at least 1. Therefore, $i = (a + 1) - 1 = a$ and $j = (b + 1) - 1 = b$.
`[Conclusion]:` Proving that the algorithm works for $i = a + 1$ and $j = b + 1$, and the values of i and j are decreased in each recursive call, we will reach the base case eventually and find the number of optimal alignments. Hence, this algorithm is correct.

`Asymptotic running time`
The running time depends on the dimension of the cost matrix, which is $n_x.n_y$, so it is $O(n^2)$.

(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.*

This is what I got by aligning the two text files.

- `?investment`
- `rogram du`
- `lance of`
- `sis exactly the kind of investment, economic development and job creation that will help put americans back to work, the president said.  many of the products that will be manufactured here in the united states by american workers will`
- `rica.  the spirit of optimism sweeping the country is already boosting job growth, and it is only the`
- `g jobs in the united states gulf coast region.  president trump made a promise to bring back`
- `manufacturi`
- `n mobil for job-cre`
- `ervice the`
- `.  ng many more jobs in th`
- `.  will have a mult`
- `.  ing the gulf prog`
- `n new refin`

As you can see, my program might not be perfect, but it got fair results from aligning the two text files.
We can see that the two text files have many similarities, so these two documents will not be considered as original work under CU's academic honesty policy. I think this will be considered plagiarism.

Problem 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) }
```

(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.*

[i.]:
We can see from the pseudocode that the base case is when $n == 0$ or $n == 1$, otherwise if P[n] is not defined then two recursive calls will happen.
If we traverse through the recursive calls, we will notice that the first recursive call, 2*MemPell(n-1), will be called before MemPell(n-2). Therefore, what will happen is (assuming the first recursive call is the left child of the tree), the left subtree will be processed before the right subtree.
We should notice that when we reach the leaf node in the left subtree (which is the base case), we will start computing. So, the computation will start from $i = 0$ to $i = n$.
When we reach the root, the right subtree will be processed. However, all the values of P[n] from the right subtree are saved, so the right subtree is actually of the size 1.

[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.

The tree is filled by calling 2*P[i-1] + P[i-2]. So what happens, after the two base cases are entered in the array, when we want to call P[i], is that P[i-1] and P[i-2] are called. This is what is going to happen until i = n.

`[ii.]:`
The running time depends on the steps taken to traverse through the tree. Each node is traversed twice, except the leaf nodes. We should note that there are $n-1$ internal nodes and n leaf nodes (since each internal node has one leaf node, except the last internal node has two leaf nodes). Therefore, the running time is $2(n-1)+n = \Theta(n)$
`Proof:  By induction`
`Base case:` By doing in-order traversal, we know that the tree will start processing the most left child and start going up (from i = 0 to n). The first returned value will be 2, which is the sum of the two base cases $(2(1)+0=2)$. The two base cases are when $n=0$ and $n=1$, since they pass the first if statement, the returned value is 0 or 1.
`Induction step:` Assuming the values P[i] have been filled for $0 \leq i < j$. When P[j] is going to be filled to the array, we will assume it is not defined. So, MemPell(j-1) and MemPell(j-2) will be called recursively. Since $j-2 < j-1 < j$, then j-2 and j-1 are defined, their values will be returned, hence the value of $P[j]$ will be calculated.
`Conclusion:` For each value P[j], one more addition will be made. Therefore, by induction, we proved that the number of additions is the size of P, which is $\Theta(n)$.

(b) *Ron then claims that he can beat Hermiones 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, 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).

The running time is still the same, $\Theta(n)$. We can see that from the size of the for loop (from i = 2 to n). The operations inside the for loop are all atomic (1 addition, 1 multiplication, 1 assignment, 2 array access).
The space is also $\Theta(n)$. It is less than the space used in the algorithm in 2a, but it not asymptotically smaller. The space is less by a constant factor, since the the function DynPell eliminated the recursion (so eliminated the space used be the stack).

[2] Ron is now using Hermiones 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 Rons 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.

`Bug:` The function FasterPall returns a, but it should return b. Returning a means returning the nth-1 Pell number. However, b is the right value to be returned, since it is going to be equal to the nth Pell number.
`Time:` Still $\Theta(n)$. We can see that from the for loop (i = 2 to n).
`Space:` The space now is different. It is $\Theta(1)$. The variable c is declared and initialized in the scope of the for loop, so it is not stored. Hence, the number of operations is constant to find the nth value (since they are all atomic operations).

(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?)*

Algorithm	Time	Space	Data structures
Pell	$O(\phi^n)$	$\Theta(n)$	Stack
MemPell	$\Theta(n)$	$\Theta(n)$	Stack and Array
DynPell	$\Theta(n)$	$\Theta(n)$	Array
FasterPell	$\Theta(n)$	$\Theta(1)$	Scalars

The improvement from `Pell` to `MemPell` lies in the running time. That is because of using an array in `MemPell`.
The improvement from `MemPell` to `DynPell` lies in simplifying the data structures used (using arrays only) by not using recursion.
And the improvement from `DynPell` to `FasterPel` lies in the space. We see that `FasterPell` only uses scalars, so that helped reducing the space from $\Theta(n)$ to $\Theta(1)$.

If we compare `Pell` to `FasterPell`, we see a huge improvement in the running time and space used. In `FasterPell`, larger values of pell numbers can be computed, in contrast to `Pell`, where it was taking a long time.

(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 Rons classic recursive algorithm and compare that to the clock time required to compute P_n using FasterPell.*

`FasterPell:`
n = 725
FasterPell(725) = first five digits = 29416
Takes 725 nanoseconds

`Pell:`
$\phi = (1 + \sqrt{5})/2 = 1.6180339887$
$\phi 725 = 1.6180339887^{725} = 3.2812488725533576 * 10^{151}$ nanoseconds
There are about $3.154 * 10^{16}$ nanoseconds in one year.
Therefore, it will take $1.040345235432263 * 10^{135}$ years to compute the 725th pell number using the first alogrithm, while FasterPell takes only about 725 nanoseconds!

I worked with:
Yazeed Almuqwishi, Abdullah Bajkhaif, Mahmoud Almansori, Firas Almakhrouki, Yousif Barnawi, Omar Mohammed