Name: Jonathan Phouminh

ID: 106054641

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 9b (51 points) **Fall 2019, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Verbal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solution:

- The solutions **should be typed** and we cannot accept hand-written solutions. Here's a short intro to Latex.
- You should submit your work through **Gradescope** only.
- If you don't have an account on it, sign up for one using your CU email. You should have gotten an email to sign up. If your name based CU email doesn't work, try the identikey@colorado.edu version.
- Gradescope will only accept **.pdf** files (except for code files that should be submitted separately on Gradescope if a problem set has them) and **try to fit your work in the box provided**.
- You cannot submit a pdf which has less pages than what we provided you as Gradescope won't allow it.

Name: Jonathan Phouminh

ID: 106054641

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 9b (51 points) **Fall 2019, CU-Boulder**

Important: This assignment has 1 (Q2) coding question.

- You need to submit 1 python file.
- The .py file should run for you to get points and name the file as following -
  If Q2 asks for a python code, please submit it with the following naming convention - `Lastname-Firstname-PS9b-Q2.py`.
- You need to submit the code via Canvas but the table/plot/result should be on the main .pdf.

Name: Jonathan Phouminh

ID: 106054641

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**

Problem Set 9b (51 points) **Fall 2019, CU-Boulder**

1. (21 pts) The sequence P_n of Pell numbers is defined by the recurrence relation

$$P_n = 2P_{n-1} + P_{n-2} \qquad (1)$$

with seed values $P_0 = 1$ and $P_1 = 1$.

(a) (5 pts) Consider the recursive top-down implementation of the recurrence (1) for calculating the n-th Pell number P_n.

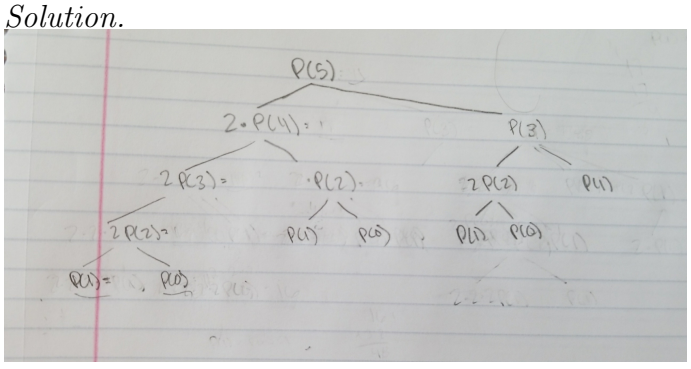
i. Write down an algorithm for the recursive top-down implementation in pseudocode.

Solution.

```
def pellNum(n):
if (n == 1 or n == 0):          # once base case has been hit return
    return 1
else:
    result = (2*pellNum(n-1) + pellNum(n-2))
return result
```

ii. Draw the tree of function calls to calculate P_5. You can call your function f in this diagram.

Solution.



iii. Write down the recurrence for the running time $T(n)$ of the algorithm.
Solution.
$T(n) = T(n-1) + T(n-2) + \Theta(1)$
$T(n) \in O(2^n)$

(b) (6 pts) Consider the dynamic programming approach "top-down implementation with memoization" that memoizes the intermediate Pell numbers by storing them in an array $P[n]$.

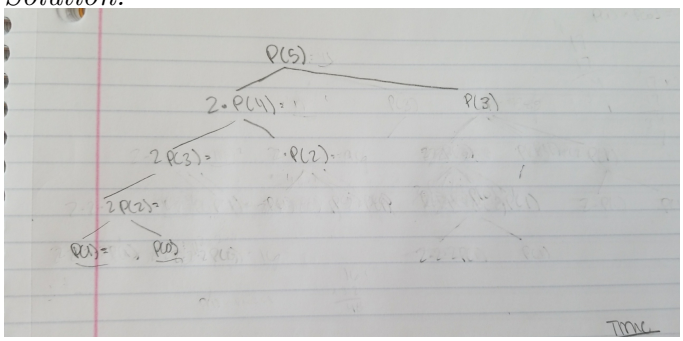
i. Write down an algorithm for the top-down implementation with memoization in pseudocode.

Solution.

```
def pellNum_memo(array,n):
     if n == 1 or n == 0:
        array[n] = 1
        return 1
     if array[n] is not None:
         return array[n]
    else:
     array[n] = (2*pellNum_memo(array,n-1) + pellNum_memo(array,n-2))
     return array[n]
```

ii. Draw the tree of function calls to calculate P_5. You can call your function f in this diagram.

Solution.



iii. In order to find the value of P_5, you would fill the array P in a certain order. Provide the order in which you will fill P showing the values.

Solution.

$p_1 : P(1) = 1$
$p_2 : P(0) = 1$
$p_3 : P(2) = 3$
$p_4 : P(3) = 7$
$p_5 : P(4) = 17$
$p_6 : P(5) = 41$

iv. Determine and justify briefly the asymptotic running time $T(n)$ of the algorithm.

Solution.

The runtime of of T(n) will be $O(n)$ as the algorithms bottleneck is calculating $2 * P_n - 2$ until its hits the base case and since we are memoizing the other nodes of the tree are being accessed in $\theta(1)$ time.

(c) (5 pts) Consider the dynamic programming approach "iterative bottom-up implementation" that builds up directly to the final solution by filling the P array in order.

i. Write down an algorithm for the iterative bottom-up implementation in pseudocode.

Solution.

```
def pennNum_bottomUp(n):
    memo = []
    if n == 1 or n == 2:  # base cases
        return 1
    if n = 3:
        return 3          # trivially true
    memo.append(1)      # this allows us to actually perform bottom up
    memo.append(1)
    for i in range(2,n+1):    # essentially here, we will just build up pell
        memo.append((2*memo[i-1] + memo[i-2]))   # keep populating the array
    return memo[n]
```

ii. In order to find the value of P_5, you would fill the array P in a certain order using this approach. Provide the order in which you will fill P showing the values.
Solution.

```
p_1 = 1
p_2 = 1
p_3 = 3
p_4 = 7
p_5 = 17
p_6 = 41
```

iii. Determine and justify briefly the time and space usage of the algorithm.

Solution. The runtime of our algorithm will also run in $O(n)$ time but it will also use $O(n)$ space usage as we are allocating an array to fill up for every $n-1$ value of n until n hits.

(d) (3 pts) If you only want to calculate P_n, you can have an iterative bottom-up implementation with $\Theta(1)$ space usage. Write down an iterative algorithm with $\Theta(1)$ space usage in pseudocode for calculating P_n. There is no requirement for the runtime complexity of your algorithm. Justify your algorithm does have $\Theta(1)$ space usage.
Solution.

```
def pennNum_bottomUp_Efficient(n):
    if n == 1 or n == 0:
        return 1
    if n = 3
        return 3
    maximum
    minOne = 1
    minTwo = 1
    for i in range(2,n+1):
        maximum = 2*minOne + minTwo
        minOne = maximum
    return max
```

This algorithm uses $\Theta(1)$ space usage because it only uses one extra variable of space as opposed to having an array of items with size n, so we are only using a constant amount of memory.

(e) (2 pts) In a table, list each of the four algorithms as rows and in separate columns, provide each algorithm's asymptotic time and space requirements. Briefly discuss how these different approaches compare, and where the improvements come from.

Solution.

```
--------------------------------------------------------------------------------
pell with no memoization
Runtime: O(2^n)
space complexity: O(n)
--------------------------------------------------------------------------------
pell with memoization
Runtime: O(n)
space complexity: O(n)
    improvement with this algorithm was in the time as
    we didn't have to keep recalculating already solved problems
--------------------------------------------------------------------------------
pell calculated via bottom up
Runtime: O(n)
space complexity: O(n)
    no improvement to memoization, just a different way of solving with
    DP
--------------------------------------------------------------------------------
pell calculated via bottom up with O(1) space complexity
Runtime: O(n)
space complexity: O(1)
    improvement from previous bottom up came from not filling a n-sized
    array, and instead using a constant amount of memory at each iteration.
```

Name: Jonathan Phouminh

ID: 106054641

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 9b (51 points) **Fall 2019, CU-Boulder**

2. (10 pts) Write a single python code for the following. There is a very busy student at CU who is taking CSCI 3104. They know that this course has a ton of homework and they don't want to attempt all of the homework. This student cherishes the downtime and has **decided not to do any two consecutive assignments**.
Assume that the student gets a list of assignments with the points associated at the beginning of the semester. Use dynamic programming to pick which assignments to complete to maximize the available points while not solving any two consecutive assignments.

   Input: [2,7,9,3,1]
   Output: 12
   Explanation: Maximum points available = 2 + 9 + 1 = 12.

   (a) (2 pts) Show an example with at least 4 assignments to show why the greedy strategy
   $max(sum(even\_indexed\_terms), sum(odd\_indexed\_term))$ does not work.
   Example - For the above list, $sum(even\_indexed\_terms) = 2 + 9 + 1$ and $sum(even\_indexed\_terms) = 7 + 3$. But, coincidentally their max gives out the optimal answer. You have to provide an example where this doesn't work.
   Solution.

```
let array = [10,1,1,10]
    we choose the greedy approach we will get the maximum value
    to be at value 11, which in reality the true maximum value
    is equal to 20.
```

   (b) (4 pts) Write the bottom-up DP table filling version that takes the list of assignments and outputs the maximum points that the student can attempt. (If it helps, you can code the recursive approach for practice but you don't need to submit that)

   (c) (4 pts) Write the DP version for part (b) which uses $O(1)$ space.
   Note that you don't have to submit anything for part (b) and (c) on the pdf but only the commented code in the python file.

Name: Jonathan Phouminh

ID: 106054641

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 9b (51 points) **Fall 2019, CU-Boulder**

3. (10 pts) Suppose we are trying to create an optimal health shake from a number of ingredients, which we label $\mathcal{I} = \{1, \ldots, n\}$. Each cup of an ingredient contributes p_i units of protein, as well as c_i calories. Our goal is to maximize the amount of protein, such that the shake uses no more than C calories. **Note that you can use more than one cup of each ingredient.**

   - Design an DP based algorithm which takes the arrays p and c and calories C as input and outputs the maximum protein you can put using no more than C calories.
   - In order to fill a particular table entry, you would need to access some sub-problems. Explain briefly which decisions each of those sub-problems represent.
   - Also, provide the runtime and space requirement of your algorithm.

   Solution.

```python
#implementing via bottom up approach, half pseudo code, have code
def maxProtein(proteinArray,caloriesArray,MAXCALORIE):
    if len(proteinArray) == 1:  # base case
        return MAXCALORIE//caloriesArray[0] * proteinArray[0]        # determines how many cups you can fit into caloric limit
                                                                     # then gets you the total amount of protein you just maximized
    if len(proteinArray) == 2:
        return max(MAXCALORIE//caloriesArray[0] * proteinArray[0],MAXCALORIE//caloriesArray[1] * proteinArray[1])

    # now we need to dynamicallys solve
    maximumProtein = []     # memo
    maximumProtein.append(MAXCALORIE//caloriesArray[0] * proteinArray[0])       # this sets the first subproblems correct answer
    maximumProtein.append(max(MAXCALORIE//caloriesArray[0] * proteinArray[0],MAXCALORIE//caloriesArray[1] * proteinArray[1]))    # this sets the second problems correct answer

    currentMax = maximumProtein[1]      # this variable holds the current maximum value at the iteration before considering the next item
    maxProtein_considering_currentItem = 0  # counter to keep track of maxprotein
    size = len(caloriesArray)
    for i in range(2,size):
        n = len(maximumProtein) -1 # gives us the most recent maxium value
        currentMax = maximumProtein[n]

        '''
        At this point in the code we would implement a method that calculates the most optimal protein value considering the two items
        and store it in a variable called, maxProtein_considering_currentItem

        we go about this by first taking the current item we have and subtracting the amount of calories it takes up from the MAXCALORIE amount.
        After we know what the remaining calorie count is we can go back into the memo of solutions and find the sub-problem that has already been solved
        that can fit into the current calorie count ( this value is also maximal to all other subsequent values in the memo of solutions).

        Once we have thsoe two values we sum the protein count of the item that is being considered in this iteration and the solution to the subproblem
        we have just found that fits into the remaining calorie count.
        '''
        currentMax = max(currentMax,maxProtein_considering_currentItem) # so from here the maximal value of the currentMax
        maximumProtein.append(currentMax)
    return maximumProtein[i]
```

In my solution, each subproblem represents the solution to the value of MAXCALORIE count after considering taking the current item. For example if the MAXCALORIE value is equal to 500 and the current calorie value for the current item is equal to 400, we would take the current item and access the subproblem whose MAXCALORIE value is equal to 100 and add the protein amount of both of the current item and the optimal solution to the subproblem that had MAXCALORIE of 100.

Name: Jonathan Phouminh
ID: 106054641

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**
Problem Set 9b (51 points) **Fall 2019, CU-Boulder**

The runtime of the algorithm is $O(n^2)$, and space complexity is $\Theta(n)$

Name: Jonathan Phouminh

ID: 106054641

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**

Problem Set 9b (51 points) **Fall 2019, CU-Boulder**

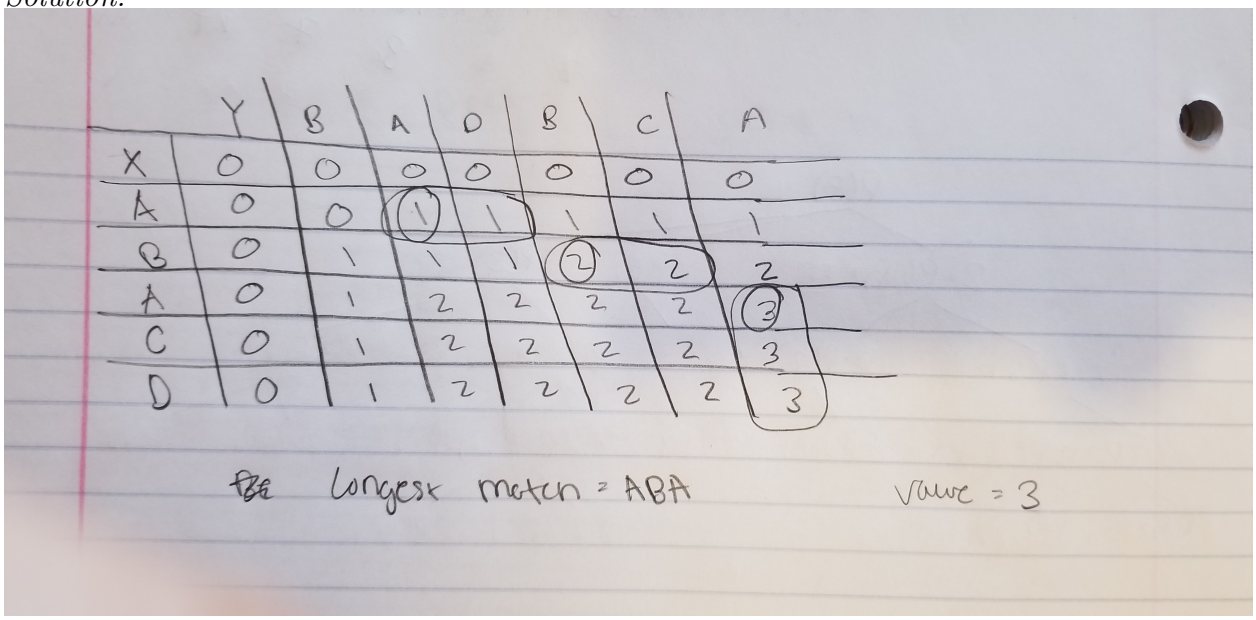
4. (10 pts) In recitation you learnt the longest common sub-sequence (LCS) problem, where you used a DP table to find the length of the LCS and to recover the LCS (there might be more than one LCS of equal length). For example - For two sequences $X = \{A, B, C, B, D, A, B\}$ and $Y = \{B, D, C, A, B, A\}$. Here's a complete solution. Grey cells represent one of the LCS (BCBA) and the red-bordered cells represent another (BCAB). Note that you have to provide only one optimal solution.

PS9/LCS.jpeg

(a) (6 pts) Draw the complete table for $X = \{A, B, A, C, D\}$ and $Y = \{B, A, D, B, C, A\}$.

- Fill in all the values and parent arrows.
- Backtrack and circle all the relevant cells to recover the actual LCS and not only the length. Do not forget to circle the appropriate characters too.
- Report the length of the LCS and the actual LCS.

Solution.



Name: Jonathan Phouminh

ID: 106054641

CSCI 3104, Algorithms **Profs. Hoenigman & Agrawal**

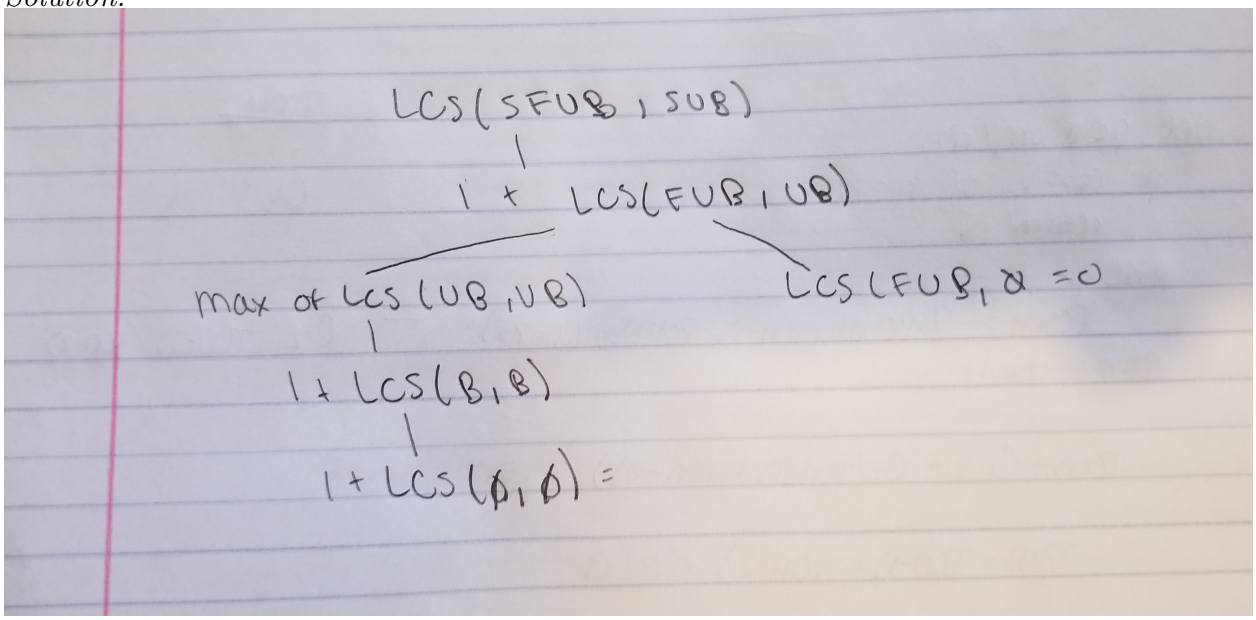
Problem Set 9b (51 points) **Fall 2019, CU-Boulder**

(b) (4 pts) If you draw the recursive tree for the recursive version of LCS, you will get something like this. Here we show all the recursive calls till the base case and

`PS9/LCS_recursive.jpeg`

annotate the children calls with a $'Max'$ or a $'+1'$ while indicating the base case calls. We also compute the values from bottom to top as we get them. Draw a tree like above for the LCS calls for string $'SFUB'$ and $'SUB'$ i.e. $LCS(SFUB, SUB)$.

Solution.



COLLABORATED WITH:
Zach Chommala
Bao Nguyen
Wayne Wood