Name: Michael Rogers
ID: 105667404

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: Michael Rogers

ID: 105667404

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: Michael Rogers
ID: 105667404

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.

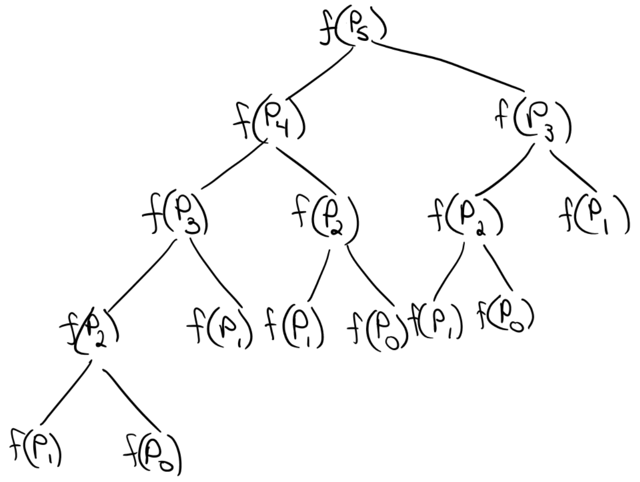
i. Write down an algorithm for the recursive top-down implementation in pseudocode.

Solution. Recursive Approach:

```
def pellNumsRecursion(n):
   if n < 2 and n >= 0:
      return 1
   else:
       return 2 * pellNums(n - 1) + pellNums( n - 2)
```

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

Solution. Tree:



iii. Write down the recurrence for the running time $T(n)$ of the algorithm.

Solution. T(n) = 2T(n - 1) + T(n - 2) + $\Theta(1) \in O(2^n)$

(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.

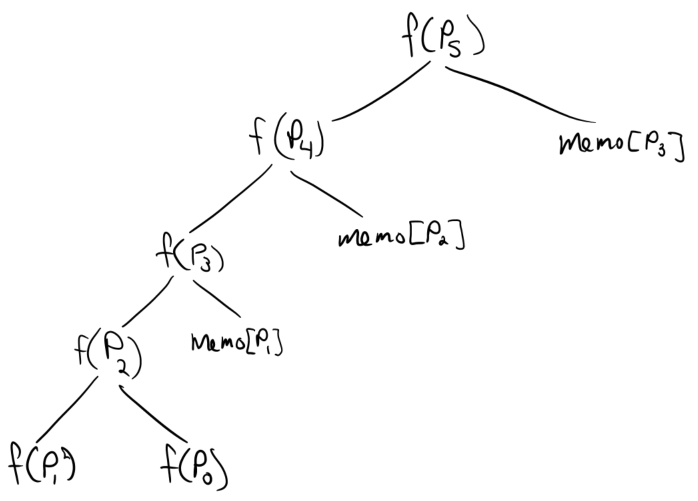
Solution. DP Approach:

```
def pellNumsDP(n):
   memo = []

   if n < 2 and n >= 0:
      return 1
   if solution of n in memo:
      return memo[n]
   return memo[n] = 2 * pellNumsDP(n - 2) + pellNumsDP(n - 1)
```

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. $P = [P_5, P_4, P_3, P_2, P_1, P_0]$

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

Solution. $T(n) \in O(n)$ This approach runs in O(n) time because it must go through all of the sub-problems and store them at least once in the worst case.

(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. Bottom-Up:

```
def pellNumsDPBU(n):
    P = []
    P[0] = 1
    P[1] = 1

    for i in range(2, n):
        P[i] = 2(P[i-1]) + P[i-2]
```

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. $P = [P_0, P_1, P_2, P_3, P_4, P_5]$

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

Solution. $T(n) \in O(n)$ This algorithm uses O(n) space because it has to iterate and store each element from 1, ..., n.

(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. $\Theta(1)$ Space:

```
def pellNums1(n):
    P = []
    P[0], P[1] = 1

    for i in range(2, n):
        P[i] = 2(P[i-1]) + P[i-2]
        if P[i - 3] != NULL:
            P[i-3].remove()
```

This algorithm uses $\Theta(1)$ space because it is deleting elements behind the elements reqired for the solution. At any time, there are no more than 3 elements in the list, the element to calculate, and the two behind it being used for calculation.

(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.

Recursion	$O(2^n)$	$O(2^n)$
Top-Down	O(n)	O(n)
Iterative	O(n)	O(n)
Iterative w/ Deletion	O(n)	O(1)

The improvements come from the fact that we are decreasing the number of elements either read or stored with each iteration. In the last method, we are deleting so we have a constant number of elements at any given time.

Name: Michael Rogers
ID: 105667404

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. Assignments = {20, 10, 1, 30, 1 }

   sumEven = 22
   sumOdd = 40
   optimal = 50

   (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: Michael Rogers
ID: 105667404

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. Algorithm:

```
def optimalShake(p, c, calories):
   // Matrix of size: Number of Itmes x Calorie Weights
   dp = [matrix where all the entries are zero]

   for i in number of items:
      for j in calories:
         // If the calories is zero, dp[i][j] = 0
         if i == 0 or j == 0:
            dp[i][j] = 0
         elif c[i-1] <= j:
            dp[i][j] = max(p[i-1] + dp[i-1][j-c[i-1]], dp[i-1][j])
         else:
            dp[i][j] = dp[i-1][j]
   return dp[Number of items][j]
```

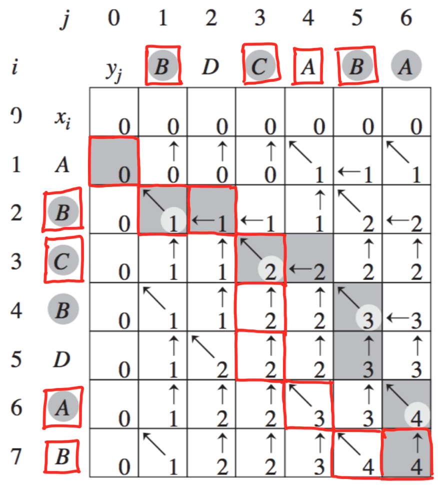
These subproblems represent the three different possibilities that are possible when filling in the table. This is essentailly the same problem as the knapsack problem with different words. It is first considering the first ingredient, then the first and second, and so on until it considers all ingredients to find the optimal solution. At each iteration, it is deciding if there is a better solution, or if the previous solution is still best. The first if statement is our base case. Running time: O(n^2), Space Complexity: O(Cn) Where C is the max calories.

Name: Michael Rogers
ID: 105667404

CSCI 3104, Algorithms
Problem Set 9b (51 points)

Profs. Hoenigman & Agrawal
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.



	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

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

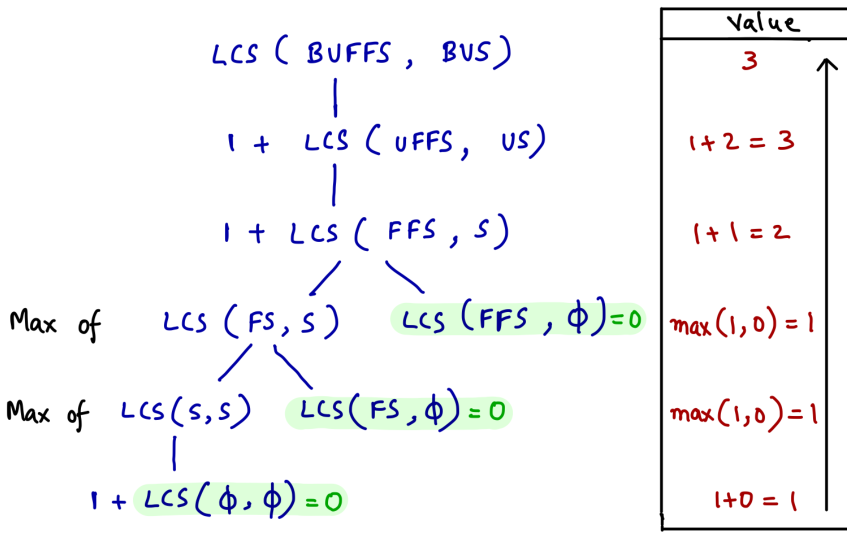
- Fill in all the values and parent arrows.
- Backtrack and circle all the relevant cells to recover the actual LCS and not only the length. Do not forget to circle the appropriate characters too.
- Report the length of the LCS and the actual LCS.

Solution. See next page

		0	1	2	3	4	5	6
		y_i	B(*S)	A(*S)	D(*S)	B	C	A
0	x_i	0	0	0	0	0	0	0
			↑					
1	A	0	0	1	1	1	1	1
			↑	←	←			
2	B(*S)	0	1	1	1	2	2	2
					↑	←	←	
3	A(*S)	0	1	2	2	2	2	3
							↑	←
4	C	0	1	2	2	2	3	3
								↑
5	D(*S)	0	1	2	3	3	3	3

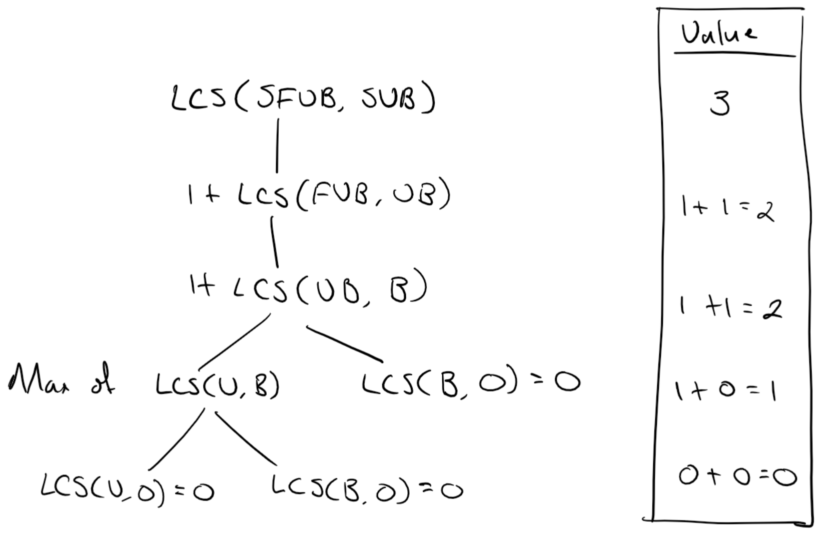
*S = Selected in LCS
LCS: {B,A,D}

(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



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.


LCS(SFUB, SUB)
1+ LCS(FUB, UB)
1+ LCS(UB, B)
Max of LCS(U, B)
LCS(B, 0) = 0
LCS(U, 0) = 0
LCS(B, 0) = 0
Value
3
1+1 = 2
1 +1 = 2
1+0 = 1
0+0 = 0