Name:

ID:

CSCI 3104, Algorithms
Homework 4 (100 pts)

Escobedo & Jahagirdar
Summer 2020, 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**, we cannot accept hand-written solutions. Here's a short intro to **Latex**.
- In this homework we denote the asymptomatic *Big-O* notation by $\mathcal{O}$ and *Small-O* notation is represented as o.
- We recommend using online Latex editor **Overleaf**. Download the **.tex** file from Canvas and upload it on overleaf to edit.
- 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 Canvas 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.

Piazza threads for hints and further discussion

Piazza Threads
Question 1a
Question 1b
Question 1c
Question 1d
Question 1e
Question 2
Question 3

Recommended reading:
Dynamic Programming: Chapter 15 complete

Name:
ID:

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Homework 4 (100 pts) **Summer 2020, CU-Boulder**

1. (65 pts) The sequence L_n of Lucas numbers is defined by the recurrence relation

$$L_n = L_{n-1} + L_{n-2} \qquad (1)$$

with seed values $L_0 = 2$ and $L_1 = 1$.

(a) (14 pts) Consider the recursive top-down implementation of the recurrence (1) for calculating the n-th Lucas number L_n.

i. (8 pts) Write down an algorithm for the recursive top-down implementation in pseudocode.

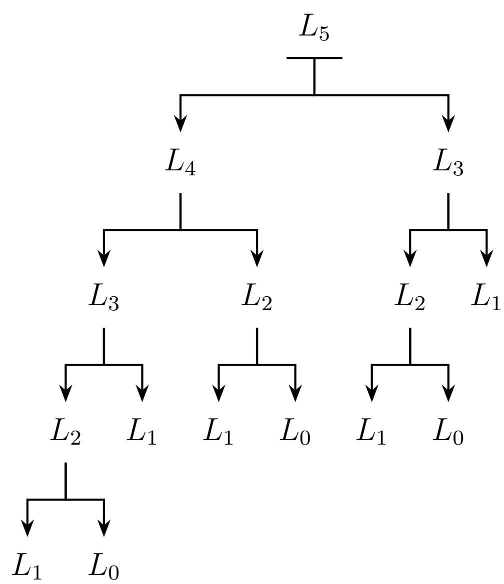
function nLucas(n)

if n==0 **then**
  return 2

  if n==1 **then**
    return 1

    return nLucas($n-1$) + nLucas($n-2$) =0

ii. (2 pts) Draw the tree of function calls to calculate L_5. You can call your function f in this diagram.



iii. (4 pts) Write down the recurrence relation along with the base case for the running time $T(n)$ of the algorithm.

Base Condition T(0) = 2, T(1)= 1
Recurrence Relation T(n)= T(n-2) + T(n-1)
where T(n) is the running time

iv. (18 pts) Consider the dynamic programming approach “top-down implementation with memoization” that memoizes the intermediate Lucas numbers by storing them in an array $L[n]$.

A. (10 pts) Write down an algorithm for the top-down implementation with memoization in pseudocode.

Require: L_0, L_1
Ensure: n^{th} Lucas Number

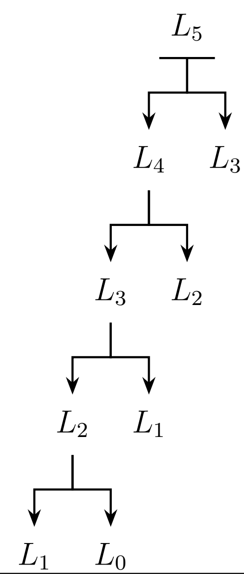
```
function LucasTopDown(n)
    L[n] = -1

    if n==0 then L[n] = 0

    else if n==1 then L[n] = 1

    else L[n]=LucasTopDown(n-1)+LucasTopDown(n-2)
        return L[n]
```

B. (2.5 pts) Draw the tree of function calls to calculate L_5. You can call your function f in this diagram.



C. (2 pts) In order to find the value of L_5, you would fill the array L in a certain order. Provide the order in which you will fill L showing the values.

- Lucas[5]=Lucas[4]+Lucas[3]
- Lucas[4]=Lucas[3]+Lucas[2]
- Lucas[3]=Lucas[2]+Lucas[1]
- Lucas[2]=Lucas[0]+ Lucas[1]
- Lucas[0]=2, Lucas[1]=1

D. (4 pts) Determine and justify briefly the asymptotic running time $T(n)$ of the algorithm.

Time Complexity is linear in time ie O(n). As the previous computed values is already stored in memoization table. Lookup time : O(1) and for n^{th} Lucas Number,it will have n-1 iterations.
Hence Time complexity :(n-1) $\times$ O(1) = O(n).

v. (16 pts) Consider the dynamic programming approach "iterative bottom-up implementation" that builds up directly to the final solution by filling the L array in order.

A. (10 pts) Write down an algorithm for the iterative bottom-up implementation in pseudocode.

Require: L_0, L_1
Ensure: n^{th} Lucas Number

```
function LucasBottomUp(n)
    L[0] ← 2
    L[1] ← 1

    for i ← 2 to N do
        L[i] = L[i − 1] + L[i − 2]
    end for
    return L[n]
```

B. (2 pts) In order to find the value of L_5, you would fill the array L in a certain order using this approach. Provide the order in which you will fill L showing the values.

- Lucas[0]=2, Lucas[1]=1
- Lucas[2]=Lucas[0]+ Lucas[1]
- Lucas[3]=Lucas[2]+Lucas[1]
- Lucas[4]=Lucas[3]+Lucas[2]
- Lucas[5]=Lucas[4]+Lucas[3]

C. (4 pts) Determine and justify briefly the time and space usage of the algorithm.

Time Complexity: O(n) as it will be having n iteration and each operation takes O(1).
Space Complexity:O(n) as additional(or auxillary) array of size n is required for storing memoized values.

vi. (7 pts) If you only want to calculate L_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 L_n. There is no requirement for the runtime complexity of your algorithm. Justify your algorithm does have $\Theta(1)$ space usage.

Require: L_0, L_1

Ensure: n^{th} Lucas Number

```
function LucasIterative(n)
    a ← 2
    b ← 1
    if n==0 then return a
        for i ← 2 to N do
            c = a + b
            a = b
            b = c
        end for
        return b
```

As no additional space in the form of array is required and each iteration takes O(1) space. Hence space complexity is O(1).

vii. (10 pts) In a table, list each of the four algorithms(as part of (a), (b), (c), (d)) 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.

Question	**Time Complexity**	**Space Complexity**
a	2^n	N
b	N	N
c	N	N
d	N	1

Using top-down and iterative bottom up uses the concept of memoization, in which past values are stored.This prevents repetition of subproblem. Hence reduces time complexity from exponential to linear in 1.b and 1.c For the storing the previous computed values,memoization table in the form of array is required.

(b) (10 pts) Consider the following DP table for the Knapsack problem for the list $A = [(12, 9), (3, 6), (9, 3), (15, 12), (18, 9)]$ of (weight, value) pairs. The weight threshold $W = 30$.

- Fill in the values of the table.
- Draw the backward path consisting of backward edges and do not draw (or erase them) the edges that are not part of the optimal backward paths.

(a) (6 pts) Fill the table with the above requirements (You can also re-create this table in excel/sheet or on a piece of paper and add picture of the same).

Weight	Value	items_considered	0	1	2	3	4	5	6	7	8	9	10
-	-	no items											
12	9	A[0..0]											
3	6	A[0..1]											
9	3	A[0..2]											
15	12	A[0..3]											
18	9	A[0..4]											

Wt	Cost	I	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	9	1	0	0	0	0	0	0	0	0	0	0	0	0	9	9	9
3	6	2	0	0	0	6	6	6	6	6	6	6	6	6	9	9	9
9	3	3	0	0	0	6	6	6	6	6	6	6	6	6	9	9	9
15	12	4	0	0	0	6	6	6	6	6	6	6	6	6	9	9	9
18	9	5	0	0	0	6	6	6	6	6	6	6	6	6	9	9	9

Wt	Cost	I	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	9	1	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
3	6	2	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
9	3	3	15	15	15	15	15	15	15	15	15	18	18	18	18	18	18
15	12	4	15	15	15	18	18	18	18	18	18	18	18	18	21	21	21
18	9	5	15	15	15	18	18	18	18	18	18	18	18	18	21	21	21

(c) (2 pts) Which cell has the optimal value and what is the optimal value for the given problem?

Cell containing weight 15 has optimal value, ie Item 3.
Optimal Value: 27 units

(d) (2 pts) List out the optimal subset and provide it's weight and value.

Optimal Subset : (Weight, Value)
(3,6),(15,12) ,(12,9).
Optimal Value:27 units

(e) (25 pts) Given an array of n size, the task is to find the longest subsequence such that the **absolute difference** between two adjacent values in the sequence is odd and less than or equal to 5. i.e the **absolute difference** between the adjacent elements is one of the values from the set $\{1, 3, 5\}$
For the definition of subsequence click here

Example 1:
Input: $\{10, 30, 5, 8, 27, 1, 4, 9, 14, 17\}$
output: 6
Explanation: Here the longest sequence satisfying the above condition will be $\{10, 5, 8, 9, 14, 17\}$ having a size of 6

Example 2:
Input: $\{10, 30, 6, 9, 27, 22, 20, 19\}$
output: 4
Explanation: There are several sequences of length 4 one such sequence is $\{30, 27, 22, 19\}$ having a size of 4

(a) (5 pts) State the base case and recursive relation that can be used to solve the above problem using dynamic programming.

$$\text{LCS(A[],i)} = \begin{cases} 1, & \text{if } i \leq 0 \\ max(1, max(1, 1 + LCS(A[], i))), & i > 0, \text{diff=\{1,3,5\}} \end{cases}$$

where A[.] is the array containing element and i is the index of array

(b) (10 pts) Write down well commented pseudo-code or paste real code to solve the above problem.

```
function LCSDiff(A[], idx)

    if idx==0 then return 1        ▷ Only one subsequence ends at first index, the number itself
        ans ← 1        ▷ As answer keeps on changing, so initialize to idx value

        for i ← idx − 1 to 0 do        ▷ Comparing the first element with remaining element and checking if difference is 1 or 3 or 5

            if abs(A[idx]-A[i])≤5 and (A[idx]-A[i])%2≠0  then ans ← max(1, max(ans, 1 + LCSDiff(A[], i)))
            end if
        end for
        return ans        ▷ Return the length of answer

```

(c) (5 pts) Discuss the space and runtime complexity of the code, providing necessary justification.

$O(N)$ for iterating and comparison of difference with adjacent element is less than 5 and is odd. $O(N)$ for returning the maxlength of such subsequence.Time complexity : $O(N) \times O(N) = O(n^2)$,
Space Complexity: O(N), as auxillary array is used for storing the result.

(d) (5 pts) Show how you can modify your pseudo-code or real code to return an optimal subsequence (if the problem has multiple optimal subsequences as part of it's solution, it is sufficient to return any one of those).

Instead of 2 for loops, we can use Hashmap for searching and storing the required answer in the hashmap. This will reduce time complexity to $O(N)$, as n times for searching and n times for storing required answer into hashmap. If there are multiple optimal substructure,then storing the answer with the index pair will give us the answer.

(f) ***Extra Credit (5% of total homework grade)*** *For this extra credit question, please refer the leetcode link provided below or click here. Multiple solutions exist to this question ranging from brute force to the most optimal one. Points will be provided based on Time and Space Complexities relative to that of the most optimal solution.*

Please provide your solution with proper comments which carries points as well.

https://leetcode.com/problems/regular-expression-matching/

```
Replace this text with your source code inside of the .tex document
```