Python for Engineers 0509-1820, Fall 2020-2021

Home Assignment 7

Memoization (and recursion)

General instructions

- Read the questions **carefully** and make sure your programs work according to the requirements.
- The homework needs to be done individually!
- Read the submission rules on the course web page. All the questions need to be submitted together in the file `ex7_012345678.py` attached to the homework, after changing the number 012345678 with your ID number (Teudat Zehut if you have one, otherwise it's typically a number beginning with 9).
- How to write the solution: in this homework, you need to complete the code in the attached outline file.
- Check your code: in order to ensure correctness of your programs and their robustness in the presence of faulty input, for each question run your program with a variety of different inputs, those that are given as examples in the question and additional ones of your choice (check that the output is correct and that the program does not crash).
- The questions are checked automatically. You thus need to write your solutions only in the specified spaces in the outline file.
- Unless stated otherwise, you can suppose that the input received by the functions is correct.
- You are not allowed to change the names of the functions and variables that already appear in the attached outline file.
- You are not allowed to erase the instructions that appear in the outline file (except the lines that contain the keyword pass).
- You are not allowed to use outside libraries (you may not use import)
- All the functions that you write need to be recursive (**non-recursive functions will not be accepted**). Note that functions need to return values, not print them.
- Submission is due by: see course web page.

Note: Python implementations (Thonny, IDLE, PyCharm,…) limit the stack depth and thus the amount of outstanding recursive calls. This is done to prevent infinite recursion. See example:

```
>>> def a(x):
        a(x)


>>> a(5)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    a(5)
  File "<pyshell#3>", line 2, in a
    a(x)
  File "<pyshell#3>", line 2, in a
    a(x)
  File "<pyshell#3>", line 2, in a
    a(x)
  [Previous line repeated 990 more times]
RecursionError: maximum recursion depth exceeded
```

As a side effect, you might encounter this limit if your program requires a very large number of recursive calls despite being correct. In this assignment, feel free to ignore it, since we shall test your code with inputs that only require a reasonable number of recursive calls.

FYI, it is possible to increase the limit from within the Python shell, by running:

```
>>> import sys
>>> sys.setrecursionlimit(10000)
```

Question 1

In class we saw the Fibonacci sequence, defined as $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_{n+1} + F_n$. The first few numbers in the sequence are: **0, 1**, 1, 3, 5, 8, 13, 21, 34, …

Similarly, we can define the Four-bonacci sequence as $Q_0 = 0$, $Q_1 = 1$, $Q_2 = 2$, $Q_3 = 3$ and $Q_{n+4} = Q_{n+3} + Q_{n+2} + Q_{n+1} + Q_n$. The first few numbers in it are: **0, 1, 2, 3**, 6, 12, 23, 44, 85, … since $6 = 0 + 1 + 2 + 3$, $12 = 1 + 2 + 3 + 6$, etc.

Part A

Implement a recursive function `four_bonacci_rec(n)`, which gets a nonnegative integer n and returns Q_n. Do not use memorization.

Part B

Implement a recursive and memoized function `four_bonacci_mem(n, mem=None)`, which gets a nonnegative integer n and returns Q_n. The runtime is improved by using memoization, thus avoiding multiple unneeded computations, hence the 2nd argument (which is None or a dictionary).

Q1 Examples:

```
four_bonacci_rec(0)
```

```
0
```

```
four_bonacci_mem(0)
```

```
0
```

```
four_bonacci_rec(1)
```

```
1
```

```
four_bonacci_mem(2)
```

```
2
```

```
four_bonacci_rec(3)
```

```
3
```

```
four_bonacci_mem(5)
```

```
12
```

```
four_bonacci_rec(6)
```

```
23
```

```
four_bonacci_mem(100) #Can be done in a reasonable time only with memoization
```

```
1413851827268925536570438396 0
```

Measuring performance (do not submit)

Compare the running time of your two implementations for n=28 using the following code:

```
from timeit import default_timer as timer
n = 28
start = timer(); four_bonacci_rec(n); end = timer()
print(f'Time without memoization for {n}: {end-start}')
start = timer(); four_bonacci_mem(n); end = timer()
print(f'Time with memoization for {n}: {end-start}')
```

Question 2

Recall question 4 from the previous home assignment:

> You need to climb a staircase containing *n* steps (a strictly positive number). At each moment during the climbing, you can choose whether to climb one step or two steps at a time. How many different ways are there to climb the whole *n* steps?

Implement a recursive, memoized function `climb_combinations_memo(n, memo=None)` that computes this number. Using memorization, we can improve the running time of the non-memoized implementation you wrote in HW6. Again, the 2nd argument is None or a dictionary.

Examples:

```
climb_combinations_memo(1)
```

```
1
```

```
climb_combinations_memo(2)
```

```
2
```

```
climb_combinations_memo(7)
```

```
21
```

```
climb_combinations_memo(42)
```

```
433494437
```

Question 3

The sequence of Catalan numbers often appears in relation to various combinatorial questions. They can be defined recursively using the formula $C_0 = 1$ and $C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i}$. The first few Catalan numbers are: **1**, 1, 2, 5, 14, 42, 132, 429.

For example, $C_7 = 429 = 1 \cdot 132 + 1 \cdot 42 + 2 \cdot 14 + 5 \cdot 5 + 14 \cdot 2 + 42 \cdot 1 + 132 \cdot 1$.

Implement a recursive and memoized function `catalan_rec(n, mem=None)`, which gets a nonnegative integer n and returns C_n. The 2nd argument is None or a dictionary, surprise-surprise.

Do not use other definitions or formulas for Catalan numbers.

For more information on Catalan numbers, please see wikipedia.

Examples:

```
catalan_rec(0)
```

```
1
```

```
catalan_rec(1)
```

```
1
```

```
catalan_rec(2)
```

```
2
```

```
catalan_rec(3)
```

```
5
```

```
catalan_rec(4)
```

```
14
```

```
catalan_rec(42)
```

```
3904442991190444395924 0
```

Question 4

The local grocer needs to return change in the amount of n and would like to know in how many different ways this can be done using a specified list `lst` of coin denominations.

For instance, for $n = 5$ and lst = [1, 2, 5, 6] this can be done in four ways:

- A single 5 coin: [5];
- Two 2 coins and one 1 coin: [2,2,1];
- One 2 coin and three 1 coins: [2,1,1,1];
- Five 1 coins: [1,1,1,1,1].

For $n = 4$ and the same list there are three ways: [2,2], [2,1,1], [1,1,1,1].

Assumptions:

- Coins are unmarked (e.g., [2,2,1], [2,1,2], [1,2,2] are all equivalent), so different orders should only be counted once.
- There is an infinite supply of coins of each denomination.
- The elements of `lst` are distinct positive integers.
- n is an integer.
- When `lst` is empty, the function should return 1 for $n = 0$ and 0 otherwise (why?)

Part A

Implement a recursive function `find_num_changes_rec(n, lst)`, which gets an integer n and a list `lst` of coin denomination, and returns the number of possible ways to return change.

Do not use memoization.

Hint: similar to the dynamic programming examples shown in class, consider at each point the last element of the list, and choose whether to use it (ponder the exact recursive step!)

For instance, when lst = [1,2,5] we can skip 5 (so we won't use it any further) or use it (so less change needs to be dealt with). Don't forget that each denomination can be used multiple times. Think of the base cases: $n = 0$ (return 1, i.e, one way to return no change), empty list, $n < 0$.

It may happen that the answer is zero (when the task is impossible). See examples.

Part B

Implement a recursive, memoized function `find_num_changes_mem(n, lst, mem=None)`, with the same arguments n and `lst` as before, and also the memoization dictionary (or None).

Hint: recall lists cannot be used in dictionary keys; use tuples where appropriate.

Q4 Examples:

```
find_num_changes_rec(5,[5,6,1,2])
```

```
4
```

```
find_num_changes_rec(-1,[1,2,5,6])
```

```
0
```

```
find_num_changes_rec(1,[2,5,6])
```

```
0
```

```
find_num_changes_rec(4,[1,2,5,6])
```

```
3
```

```
find_num_changes_mem(5,[1,2,5,6])
```

```
4
```

```
find_num_changes_mem(-1,[1,2,5,6])
```

```
0
```

```
find_num_changes_mem(5,[1,2,5,6])
```

```
4
```

```
find_num_changes_mem(1,[2,5,6])
```

```
0
```

```
find_num_changes_mem(4,[1,2,5,6])
```

```
3
```

```
find_num_changes_mem(1430,[1,2,5,6,13]) # Cannot be done in a reasonable time without memoization
```

```
231919276
```

Measuring performance (do not submit)

Compare the running time of your two implementations for n=143, `lst=[1,2,5,6,13]`:

```
from timeit import default_timer as timer
n = 143; lst = [1, 2, 5, 6, 13]
start = timer(); find_num_changes_rec(n, lst); end = timer()
print(f'Time without memoization for {n} and {lst}: {end-start}')
start = timer(); find_num_changes_mem(n, lst); end = timer()
print(f'Time with memoization for {n} and {lst}: {end-start}')
```