EECE 230X – Introduction to Computation and Programming
Programming Assignment 7

- This programming assignment consists of 4 problems.
- Prerequisites: Topic 9
- Related material: Recursion I: recursive functions, factorial, recursion stack, Fibonacci numbers, recursion tree, memoization, tower of Hanoi, recursive binary search

Problem 1. Recurrences

Consider the following recurrence:

$$f_n = \begin{cases} 1 & \text{if } n = 0, 1, 2 \\ f(n-1) + f(n-2) + f(n-3) + f(\lfloor n/3 \rfloor) & \text{if } n \geq 3, \end{cases}$$

where if x is a real number, $\lfloor x \rfloor$ means the floor of x, i.e., the largest integer less than or equal to x.

Thus $f_3 = 1 + 1 + 1 + 1 + 1 = 4$, $f_4 = 4 + 1 + 1 + 1 = 7$, $f_5 = 7 + 4 + 1 + 1 = 13$, etc.

a) **Slow recursive.** Implement a recursive function `f(n)`, which given `n`, returns `f(n)`. Also find the number of recursive calls for $n = 25$.

To find the number of recursive call made by `f()`, use a *global* variable `N` initialized to zero and increment `N` in `f()`.

> *Global variables.* Recall that a function has read-only access to variables defined in a higher scope. The *global* keyword allows the function to modify such variables. For instance, the following code will give the error "local variable 'N' referenced before assignment":
>
> ```
> def g():
>     N=N+1
>
> N = 0
> g()
> print("After calling g(), N=", N)
> ```
>
> The reason is that if a variable is updated within a function, it is interpreted as a local variable. To give `g()` write-access to `N` without making it local, use the *global* keyword:
>
> ```
> def g():
>     global N
>     N=N+1
>
> N = 0
> g()
> print("After calling g(), N=", N)
> g()
> print("After calling g(), N=", N)
> ```
>
> The output of this program is:
>
> ```
> After calling g(), N= 1
> After calling g(), N= 2
> ```

In general, the use of global variable is discouraged, but they are helpful in certain cases such as counting the number of times a function was called.

Test program:

```
print("f(n) for n = 0...9:")
for i in range(10):
    N=0
    print(f(i), end=" ")
N=0
print("\nf(25): ",f(25))
print("Number of recursive calls for 25:", N)
```

Output:

```
a)
f(n) for n = 0...9:
1 1 1 4 7 13 25 46 85 160
f(25):  2786521
Number of recursive calls for 25: 3715361
```

b) **Memoized.** The recursive function in Part (a) makes many unnecessary calls to solve repeated problems. Implement a memoized recursive function `fMemoized(n)`, which given `n`, returns f_n. Follow the approach of memoized Fibonacci. As in Part (a), use a global variable to count the number of recursive calls. Try your function on the test cases in Part (a). For $n = 25$, you should get 93 recursive calls instead of 3715361.

c) **Iterative.** Now, implement a non-recursive function `fIterative(n)`, which given `n`, returns f_n. Use a list in `fIterative`. Try it on the above test cases. In this part, the global variable `N` is irrelevant.

Problem 2. Recursive power

In this problem, the use the exponentiation operator `**` is forbidden. The objective of this problem is to implement an efficient recursive function to compute the x^n, where x is a number and n is an integer. In Part (a), you will implement a slow recursive function which takes $\Theta(n)$ arithmetic operations (addition, subtraction, multiplication, and division). In Part (b), you will implement a faster recursive function which takes $\Theta(\log n)$ arithmetic operations. In Part (c), you will use the the idea in Part (b) to compute the n'th Fibonacci number in $\Theta(\log n)$ arithmetic operations.

a) **Slow recursive power.** Write a recursive function `recPowerSlow(x,n)`, which given a number `x` (int or float) and an integer `n`, returns x^n using the following recursive definition of power:

$$power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ \frac{1}{power(x,-n)} & \text{if } n < 0 \\ x \times power(x, n-1) & \text{if } n \geq 1 \end{cases}$$

Assume that $x \neq 0$ if $n < 0$.

Note that we have $n + 1$ recursive calls if $n \geq 0$ and $n + 2$ recursive calls if $n < 0$. Each recursive call takes $\Theta(1)$ arithmetic operations Thus, in total, the function takes $\Theta(n)$ arithmetic operations.

Test program:

```
print("recPowerSlow(0,0):",recPowerSlow(0,0))
print("recPowerSlow(0,3):",recPowerSlow(0,3))
print("recPowerSlow(3,0):",recPowerSlow(3,0))
print("recPowerSlow(3,1):",recPowerSlow(3,1))
print("recPowerSlow(-3,1):",recPowerSlow(-3,1))
print("recPowerSlow(2,4):",recPowerSlow(2,4))
print("recPowerSlow(2,-4):",recPowerSlow(2,-4))
x = 1.25
n = 82
print(x,"**",n,"          :",x**n)
print("recPower(",x,",",n,"):",recPowerSlow(x,n))
```

Output:

```
recPowerSlow(0,0): 1
recPowerSlow(0,3): 0
recPowerSlow(3,0): 1
recPowerSlow(3,1): 3
recPowerSlow(-3,1): -3
recPowerSlow(2,4): 16
recPowerSlow(2,-4): 0.0625
1.25 ** 82           : 88434366.00416711
recPower( 1.25 , 82 ): 88434366.00416712
```

b) **Fast recursive power.** Write a more efficient recursive function `recPowerSlow(x,n)`, which given a number `x` and an integer `n`, returns x^n. Aim for $\Theta(\log n)$ arithmetic operations. Try your function on the test cases in Part (a).

Below is the idea illustrated via an example:

The naive method to compute x^9 is:

$$x^9 = x^8 x$$
$$x^8 = x^7 x$$
$$x^7 = x^6 x$$
$$x^6 = x^5 x$$
$$x^5 = x^4 x$$
$$x^4 = x^3 x$$
$$x^3 = x^2 x$$
$$x^2 = xx$$

A much more efficient method is:

$$x^9 = x^8 x$$
$$x^8 = (x^4)^2$$
$$x^4 = (x^2)^2$$
$$x^2 = (x)^2$$

First assume $n \geq 1$. Write a recursive definition to calculate x^n inspired by the above example. Once you figure out the recurrence, the recursive function will immediately follow. Then take care of the cases $n < 0$ and $n = 0$. Submit the recurrence with your code as a comment.

c) **n'th Fibonacci number in $\Theta(\log n)$ arithmetic operations.** We verified in Topic 9 that Fibonacci numbers satisfy the following recurrence: For $n \geq 1$,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}.$$

Use the idea of your recursive function in Part (b) to compute the n'th Fibonacci number in $\Theta(\log n)$ arithmetic operations.

Complete the following code, in which 2×2 matrices are represented as lists of lists, i.e.,

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix}$$

is represented as the list of lists

$$A = [[a_{00}, a_{01}], [a_{10}, a_{11}]].$$

Thus $A[i][j] = a_{i,j}$, for $i = 0, 1$ and $j = 0, 1$

```
def multMatrices(A,B):
     """ A and B are lists of lists representing two 2 by 2 matrices"""
     C00 = A[0][0]*B[0][0]+A[0][1]*B[1][0]
     C01 = ...
     C10 = ...
     C11 = ...
     C = [[C00, C01], [C10, C11]]
     return C

def fibMatrixVersion(n): #Wrapper function
    """ Assumes  n is a non-negative integer """
    def recMatrixPowerFast(X,n):
    """ X is a list of lists representing a 2 by 2 matrix and n is non-negative integer"""
        .
        .
        .
    C = recMatrixPowerFast(....,n)
    return ....


for i in range(11):
    print("F_",i,":", fibMatrixVersion(i))
print("F_ 100 :", fibMatrixVersion(100))
```

Output:

```
F_ 0 : 0
F_ 1 : 1
F_ 2 : 1
F_ 3 : 2
F_ 4 : 3
F_ 5 : 5
F_ 6 : 8
F_ 7 : 13
F_ 8 : 21
F_ 9 : 34
F_ 10 : 55
F_ 100 : 354224848179261915075
```

Problem 3. Finding the mode of a unimodal list

A non-empty size-n list L is called *unimodal* if there exists an integer m, $0 \leq m \leq n-1$, such that $L[i-1] < L[i]$ for $i = 1, \ldots, m$ and $L[i-1] > L[i]$ for $i = m+1, \ldots, n-1$. We call the index m the *mode* of A.

Examples:

- The list `L = [1,2,4,7,11,10,8,4,-9]` is unimodal and its mode is 4 since:
  - $1 < 2 < 4 < 7 < 11$
  - The index of 11 in L is 4
  - $11 > 10 > 8 > 4 > -9$
- The list `L = [1,2,5,20]` is unimodal and its mode is 3 since it is sorted in increasing order and the index of last element is 3.
- The list `L =[50,2,1]` is unimodal and its mode is 0 since it is sorted in decreasing order and the index of first element is 0.
- The list `L = [1]` is unimodal and its mode is 0

Consider the following problem:

Find-Mode-Problem: Given a unimodal list, find its mode.

The naive approach to find the mode is to mimic idea of the sequential search: traverse the elements of L one by one and break the loop when you see the mode. The worst case time is thus $\Theta(n)$. We are interested in faster solutions.

Note that in this problem you are not supposed to check if the list is unimodal and your algorithm is not supposed to work properly if it is not unimodal.

a) **Fast Recursive Mode Finder.** Write an recursive function `recursiveModeFinder(L, low, high)` for the *Find-Mode-Problem*.

   The initial call of the function is `recursiveModeFinder(L, 0, len(L)-1)`.

   Mimic the idea of the recursive binary search algorithm: consider the middle of the list and recur on the upper sublist or the lower sublist after suitable considerations. Note that the boundaries of the upper an lower sublits are not necessarily the same as in binary search.

   Aim for $O(\log n)$ time.

   Test your function on the above examples.

b) **Fast iterative mode Finder.** Write an efficient iterative function `fastIterativeModeFinder(L)` for the *Find-Mode-Problem*.

   Mimic the idea of the iterative binary search.

   Aim also for $O(\log n)$ time.

   Test your function on the above examples.

Problem 4 ($\star$). Hanoi with 4 needles and two towers

Consider the following variation of the tower of Hanoi Problem. We have 4 needles instead of 3 and 2 towers of disks instead of one. The two towers are identical and they are originally on Needles 1 and 2. As in the original puzzle, each tower consists of n disks. See Figure 1 which corresponds to $n = 5$. We would like to move the $2n$ disks to Needle 4 as shown in Figure 2. The rules of the puzzle are as in the original puzzle:

(1) Only one disk can be moved at a time

(2) The removed disk must be placed on one of the needles

(3) A larger disk cannot be placed on top of a smaller disk.

It follows from (3) that the disks must be at the end in an interleaved configuration as shown in Figure 2.

Write a recursive function which given n prints the sequence of moves.

(*Hints:* (1) Think in terms of problems and subproblems; (2) Use the function `moveDisks` for Hanoi with 3 needles.)

The solution is compact and it consists of only few lines.

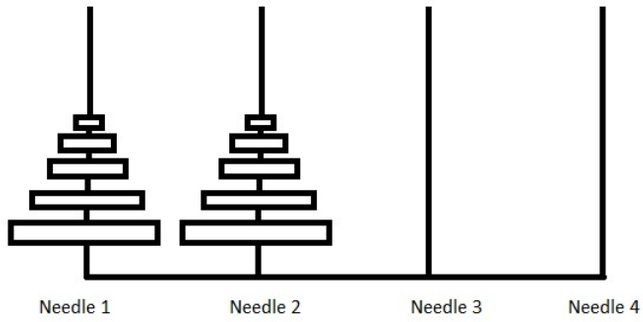

Figure 1

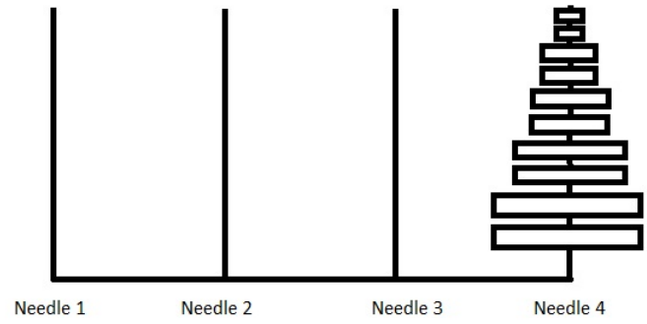

Figure 2

Below are valid moves for $n = 1, 2, 3$. Note the solution is not unique. That is, you may end up with other sequences.

```
Moves for n= 1 :
Move disk from  2  to  4
Move disk from  1  to  4
-----------------------------
Moves for n= 2 :
Move disk from  2  to  3
Move disk from  2  to  4
Move disk from  1  to  2
Move disk from  1  to  4
Move disk from  3  to  4
Move disk from  2  to  4
-----------------------------
Moves for n= 3 :
Move disk from  2  to  4
Move disk from  2  to  3
Move disk from  4  to  3
Move disk from  2  to  4
Move disk from  1  to  4
Move disk from  1  to  2
Move disk from  4  to  2
Move disk from  1  to  4
Move disk from  3  to  1
Move disk from  3  to  4
Move disk from  2  to  3
Move disk from  2  to  4
Move disk from  1  to  4
Move disk from  3  to  4
-----------------------------
```