LABORATORY 5 : Recursion

OBJECTIVES

- to understand basic concept of recursion
- to think recursively
- to write simple recursive programs

BACKGROUND

1. Recursion

   Recursion is simply a process of defining a problem in terms of (a simpler version of) the problem itself.

   The problem is breaking down to simpler forms until it is simple enough to be solved. We stop breaking the problem down when the solution to the problem is found.

   For example, the operation "eat a bowl of corn soup" can be defined as

   1. If the bowl is empty, stop.
   2. Eat one spoon of corn soup.
   3. "eat a bowl of corn soup"

   The job is to finish the entire bowl. When there is nothing more to eat, the job is done. While we are not done, we simplify our job by eating one spoon of soup → the bowl now has one spoon less → smaller amount of soup to finish → simpler job. Then, the operation is repeated on the remaining job (the remaining soup in the bowl).

   The case in which we end our recursion is called a **base case**. The case for which the problem is expressed in terms of itself is called a **general** (or **recursive**) **case**.

   The problem is simplified in such a way that it is moving closer to the base case. It is very important that the recursion progresses toward the base case. We will end up with an infinite recursion, otherwise.

   Recursive algorithm is essentially a loop like a `for` loop or a `while` loop.
   When do we prefer recursion to an iterative loop?

   The key to thinking recursively is to see the solution to the problem as a smaller version of the same problem.

   Let's consider a function to find the factorial of a given integer. For example, 6!.
   6! equals 6*5*4*3*2*1 . However, it is also correct to say 6! equals 6*5!.
   In seeing 6! as 6*5! , we now see the problem in a simpler version. We have also defined our problem in terms of itself as 6! is defined in terms of 5!. This is the essence of recursive problem solving. Now only the base case is left to be determined.

Ummm … let’s see. What is the simplest form of factorial? → 1!. 1! equals 1. Voilà ☺

Let's write the factorial function recursively using pseudocode.

```
Algorithm factorial(n)
Input : a positive integer, n
Output : factorial of n (n!)

    IF n = 1 THEN            // base case
        RETURN 1
    ELSE
        RETURN n * factorial(n - 1)
```

Could this be written iteratively? Try writing an iterative version of the factorial function.

Food for thought : what about computational complexity and execution efficiency of recursive algorithm ? Is it worse or better than its corresponding iterative version ? Why ?

2. Tracing a recursive function

When a function makes a call to another function, it is suspended until the called function finishes its job and returns. A call stack is used to keep states of function calls so that each function can resume its execution when the called function returns. Control is returned to the topmost non-terminating function in the call stack.

A call stack is LIFO (Last In First Out) data structure. The most recently added element is the first element removed.

In tracing a recursive function, there are “winding” and “unwinding” parts. The “winding” part is when the recursion makes calls towards the base case. The “unwinding” parts is when the recursion returns.

Let’s take a look at the following algorithm. Suppose `mystery(5)` is called.

```
Algorithm mystery(n)
Input : a positive integer, n
Output : ???

    IF n = 0 THEN                   // 1
        RETURN 0                    // 2
    IF num%2 = 0 THEN               // 3
        RETURN mystery(n - 1) + 1   // 4  n is even
    ELSE                            // 5
        RETURN mystery(n - 1) + 2   // 6  n is odd
```

`mystery(5)` is called. The “winding” part begins.
`mystery(5)` is added into the call stack.
`mystery(5)` stops at line 6 and calls `mystery(4)`.

`mystery(5), n = 5`	`line 6`

`call stack`

`mystery(4)` is added into the call stack.
`mystery(4)` stops at line 4 and calls `mystery(3)`.

`mystery(4), n = 4`	`line 4`
`mystery(5), n = 5`	`line 6`

`call stack`

`mystery(3)` is added into the call stack.
`mystery(3)` stops at line 6 and calls `mystery(2)`.

`mystery(3), n = 3`	`line 6`
`mystery(4), n = 4`	`line 4`
`mystery(5), n = 5`	`line 6`

`call stack`

`mystery(2)` is added into the call stack.
`mystery(2)` stops at line 4 and calls `mystery(1)`.

`mystery(2), n = 2`	`line 4`
`mystery(3), n = 3`	`line 6`
`mystery(4), n = 4`	`line 4`
`mystery(5), n = 5`	`line 6`

`call stack`

`mystery(1)` is added into the call stack.
`mystery(1)` stops at line 6 and calls `mystery(0)`.

`mystery(1), n = 1`	`line 6`	
`mystery(2), n = 2`	`line 4`	
`mystery(3), n = 3`	`line 6`	
`mystery(4), n = 4`	`line 4`	
`mystery(5), n = 5`	`line 6`	

`call stack`

`mystery(0)` is added into the call stack.
`mystery(0)` returns 0 and terminates.

~~`mystery(0), n = 0`~~	`line 2`	`returns 0`
`mystery(1), n = 1`	`line 6`	
`mystery(2), n = 2`	`line 4`	
`mystery(3), n = 3`	`line 6`	
`mystery(4), n = 4`	`line 4`	
`mystery(5), n = 5`	`line 6`	

`call stack`

The “unwinding” part begins.
Control returns to the topmost non-terminated function on the stack which is `mystery(1)` (line 6).
`mystery(1)` returns 2 which is 0 + 2 and terminates

~~`mystery(0), n = 0`~~	`line 2`	`returns 0`
~~`mystery(1), n = 1`~~	`line 6`	`returns 2`
`mystery(2), n = 2`	`line 4`	
`mystery(3), n = 3`	`line 6`	
`mystery(4), n = 4`	`line 4`	
`mystery(5), n = 5`	`line 6`	

`call stack`

Control returns to the topmost non-terminated function on the stack which is `mystery(2)` (line 4).
`mystery(2)` returns 3 which is 2 + 1 and terminates

~~`mystery(0), n = 0`~~	`line 2`	`returns 0`
~~`mystery(1), n = 1`~~	`line 6`	`returns 2`
~~`mystery(2), n = 2`~~	`line 4`	`returns 3`
`mystery(3), n = 3`	`line 6`	
`mystery(4), n = 4`	`line 4`	
`mystery(5), n = 5`	`line 6`	

`call stack`

Control returns to the topmost non-terminated function on the stack which is `mystery(3)` (line 6).
`mystery(3)`returns 5 which is 3 + 2 and terminates

~~mystery(0), n = 0~~	line 2	returns 0
~~mystery(1), n = 1~~	line 6	returns 2
~~mystery(2), n = 2~~	line 4	returns 3
~~mystery(3), n = 3~~	line 6	returns 5
mystery(4), n = 4	line 4	
mystery(5), n = 5	line 6	

call stack

Control returns to the topmost non-terminated function on the stack which is `mystery(4)` (line 4).
`mystery(4)`returns 6 which is 5 + 1 and terminates

~~mystery(0), n = 0~~	line 2	returns 0
~~mystery(1), n = 1~~	line 6	returns 2
~~mystery(2), n = 2~~	line 4	returns 3
~~mystery(3), n = 3~~	line 6	returns 5
~~mystery(4), n = 4~~	line 4	returns 6
mystery(5), n = 5	line 6	

call stack

Control returns to the topmost non-terminated function on the stack which is `mystery(5)` (line 6).
`mystery(5)`returns 8 which is 6 + 2 and terminates

~~mystery(0), n = 0~~	line 2	returns 0
~~mystery(1), n = 1~~	line 6	returns 2
~~mystery(2), n = 2~~	line 4	returns 3
~~mystery(3), n = 3~~	line 6	returns 5
~~mystery(4), n = 4~~	line 4	returns 6
~~mystery(5), n = 5~~	line 6	returns 8

call stack

Call stack is now empty. The function terminates. 8 is the answer to `mystery(5)`.

LABORATORY 5: Pre-lab

1. Read Section 4.1 of Goodrich's textbook.
2. For the following `mystery1` algorithm,

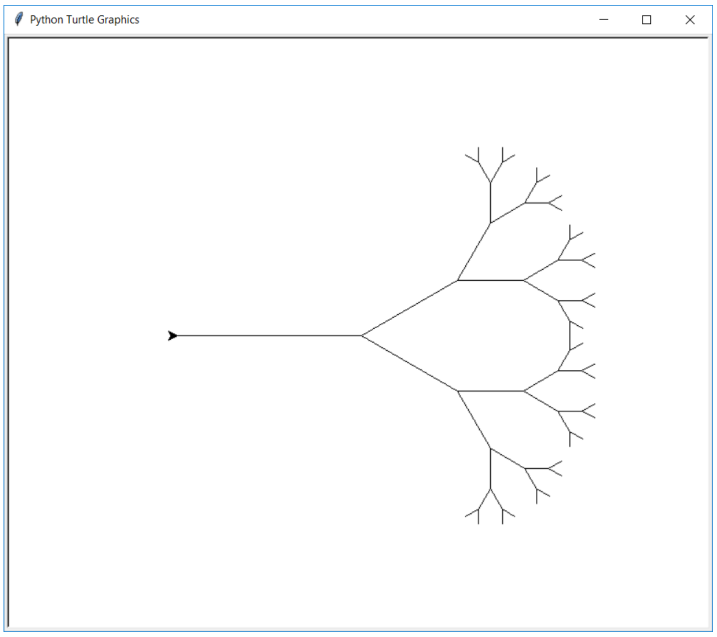
```
Algorithm mystery1(list)
Input : a list of integers, list
Output : ?

    IF length of list is 1 THEN
        RETURN first element in the list
    ELSE
        a ← first element in list
        b ← mystery1(rest of the list)
        IF a > b THEN
             RETURN a
        ELSE
             RETURN b
```

   2.1. What are base case and recursive case of `mystery1`
   2.2. Trace function `mystery1` for a list of 5 integers. Show the call stack.
   2.3. What does `mystery1` do ?
   2.4. Write an iterative version for `mystery1` (using pseudocode)

3. Write <u>recursive</u> algorithms (using pseudocode) to
   3.1. find out whether a given String is a palindrome.
   3.2. calculate `a*b` where `a` and `b` are positive integers. Note that you are not allowed to use the " `*` " operator.

LABORATORY 5: In-lab

1. For the following tree like figure, do the following



   1.1. Write a recursive algorithm (using pseudocode) to create this tree.

1.1.1. What is base case ?

1.1.2. What is recursive case ?

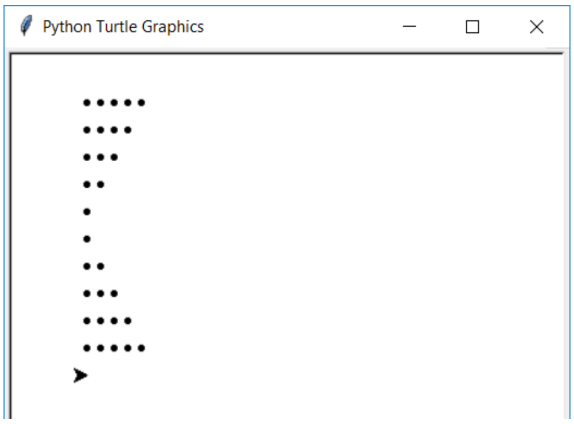
1.2. (optional) Write a program to draw this tree using `turtle`.

Hints :

- A branch is in Y shape–one long branch and two short branches. Each branch is a branch. Do you see the recursion coming ?
- What are values of angles between the three branches of the Y ?
- What is the relationship between length of big branch and length of small branch ?

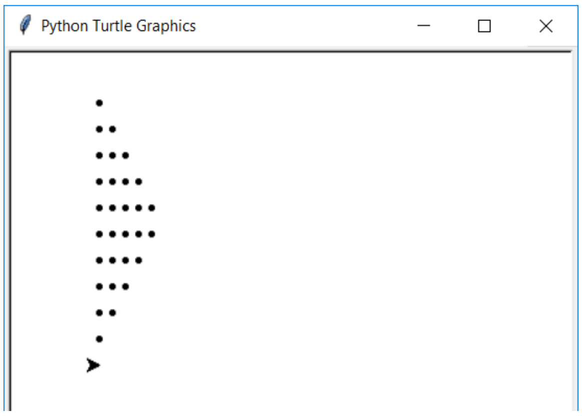
2. Write a recursive function `stars1(n)` to generate the following star pattern. Figure below is the output when `n` is 5. Print stars on Python console and use `turtle` to draw star pattern.

```
*****
****
***
**
*
*
**
***
****
*****
```



3. Write a recursive function `stars2(n)` to generate the following star pattern. Figure below is the output when `n` is 5. Print stars on Python console and use `turtle` to draw star pattern.

```
*
**
***
****
*****
*****
****
***
**
*
```



Check out available Turtle methods at https://docs.python.org/3/library/turtle.html.

LABORATORY 5: Post-lab

A sequence of Fibonacci numbers is as follows

1, 1, 2, 3, 5, 8, 13, 21, 34, …

fib(n) returns Fibonacci number at the nth position; e.g, fib(1) = 1 and fib(4) = 3.

1. write an iterative algorithm (using pseudocode) of fib(n)
2. write recursive definition of fib(n)
3. write a recursive algorithm (using pseudocode) of fib(n)
4. implement algorithms in 1 and 3 in Python
5. run your programs against a number of n's. record running time and draw graph (plot running time of both Fibonacci versions on the same graph)
6. explain what you found

Submission:

Due dates:

pre-lab, in-lab, post-lab : as stated in Canvas

You are to demonstrate your algorithm, test plan and program. Prepare to answer some questions individually.