CS1010S Tutorial 3

Wang Zexin

National University of Singapore
Year 2 QF

wang.zexin@u.nus.edu

February 5, 2017

Today's Agenda

Recap - Re-emphasize Recursion & Iteration

One way to convert between recursion and iteration!

- Recursion
    - base case (end condition) e.g. if n == 1: return something
    - recurrence relation e.g. $f(n) = g(f(n-1))$
    - starting status e.g. the first recursive function call
- Iteration
    - starting status (initialization) e.g n = 0
    - what you do in each loop e.g. the indented statements
    - end condition e.g. for i in range(1, **n+1**):
- How to convert between them? Notice their similarities

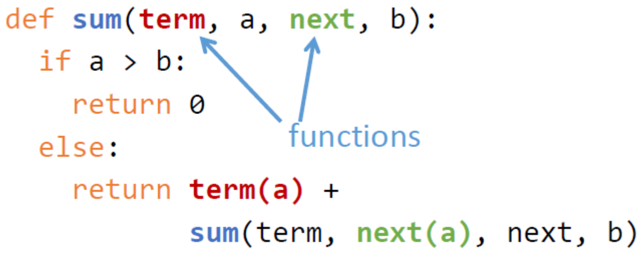
Recap - Higher order functions

What have you learnt?

What are higher order functions? Explain non-abstractly

- Taking functions as input
    - Generalize common patterns
    - Functions taken in describe the varying features
- Or their return values are functions (closure)
    - Generate functions of **similar logic** but different internal parameters
    - Often return the function(lambda) defined in function
- Specifically written higher-order functions in this course
    - sum - functions as input
    - product - functions as input
    - fold - functions as input
    - make_something - functions as output
    - accumulate - not introduced yet

Recap - Higher order function examples



```
def sum(term, a, next, b):
  if a > b:
    return 0
  else:
    return term(a) +
           sum(term, next(a), next, b)
```

Practice: find the sum of squares of integers?

Recap - Higher order function examples

```
def product(term, a, next, b):
  if a > b:
    return 1
  else:
    return term(a) * product(term, next(a), next, b)
```

Practice: How would you define factorial in terms of product?
Practice: How would you define exponential in terms of product?

Recap - Higher order function examples

```
def fold(op, f, n):
  if n==0:
    return f(0)
  else:
    return op(f(n), fold(op, f, n-1))
```

Fold is much more general than sum and product.
Practice: Can you mimic sum or product using fold?

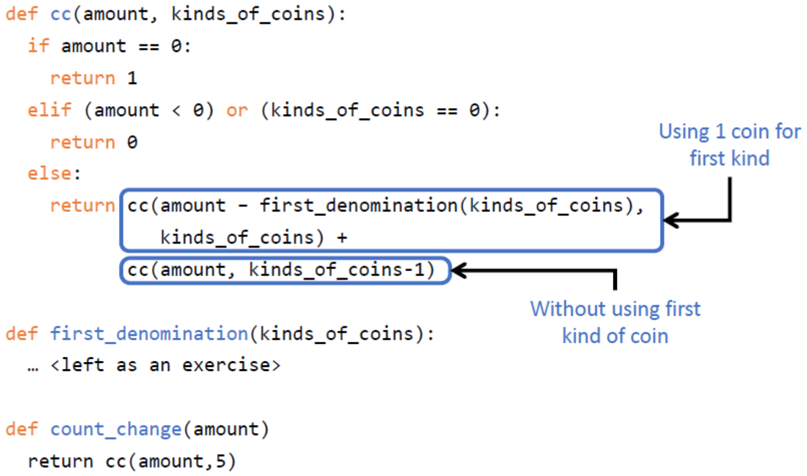
Recap - Order of Growth

- Method on lecture slides (may not work so well!)
  - Identify the basic computation steps
  - Try a few small values of n
  - Extrapolate for really large n
  - Look for **worst case** scenario
- Time complexity
  - Iteration : k nested loops means $O(n^k)$
  - Recursion : k functions call of $f(n-1)$ means $O(k^n)$
  - In the future, look out for string/list concaternation, ...
- Space complexity
  - Iteration : k nested loops means nothing, usually $O(1)$
  - Recursion : k functions call of $f(n-1)$ means $O(n)$ (memory release)
  - In the future, look out for storage space taken by lists, tuples, dictionaries, ...

Question 1 Discussion

Draw the tree illustrating the process generated by the $cc(amount, d)$ function given in the lecture, in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the *amount* to be changed increases?

Question 1 function cc code



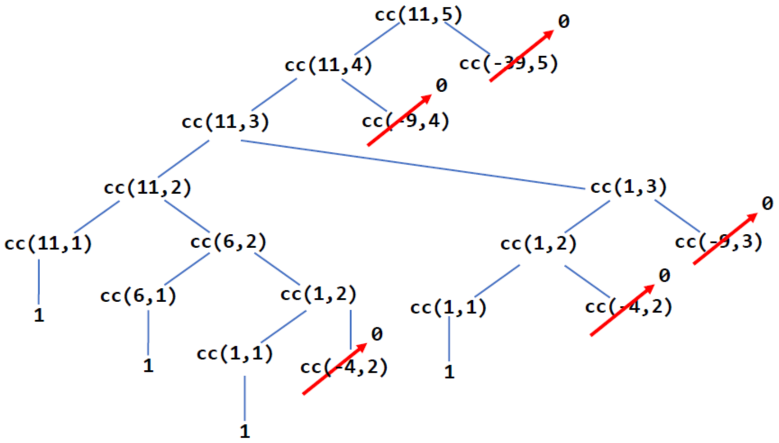
```
def cc(amount, kinds_of_coins):
  if amount == 0:
    return 1
  elif (amount < 0) or (kinds_of_coins == 0):
    return 0
  else:
    return cc(amount - first_denomination(kinds_of_coins),
              kinds_of_coins) +
           cc(amount, kinds_of_coins-1)

def first_denomination(kinds_of_coins):
  … <left as an exercise>

def count_change(amount)
  return cc(amount,5)
```

- Takeaway: you can also try this top-down approach

Question 1 Prof's solution



Order of growth of the space is $O(n)$ (assuming amount is n)
Order of growth of number of steps is $O(2^n)$

Question 2 Part (a) Discussion

A function f is defined by the rule that $f(n) = n$ if $n < 3$ and$f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n < 3$.

- Write a function that computes f by means of a recursive process.

Question 2 Part (b) Discussion

A function f is defined by the rule that $f(n) = n$ if $n < 3$ and$f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n < 3$.

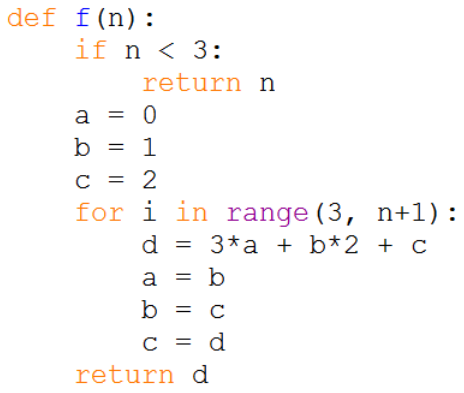
- Write a function that computes f by means of an iterative process.

Question 2 (a) Zexin's solution

```
def f(n):
    if n < 3:
        return n
    else:
        return f(n-1) + 2 * f(n-2) + 3 * f(n-3)
```

- Order of growth for time complexity is $O(2^n)$
- Order of growth for space complexity is $O(n)$
- Why is it so? Let's analyse it mathematically.

Question 2 (b) Zexin's solution



```
def f(n):
    if n < 3:
        return n
    a = 0
    b = 1
    c = 2
    for i in range(3, n+1):
        d = 3*a + b*2 + c
        a = b
        b = c
        c = d
    return d
```

- Order of growth for time complexity is $O(n)$
- Order of growth for space complexity is $O(1)$
- The storage space required remains constant as n increases.

Question 3 Discussion

Write a function *is_fib*(n) that returns True if n is a Fibonacci number, and False otherwise. What is the order of growth in terms of time and space for the function that you wrote? Explain.

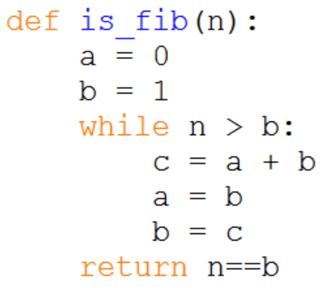
There are a few things which we can be sure:

- There is no closed form formula to determine if n is Fibonacci
- If n is not Fibonacci, it must lie between two Fibonacci numbers
- If n is Fibonacci, it must appear in the Fibonacci series (increasing)

Here we form our algorithm:

- By monotonicity, if there is Fibonacci number greater than n, n is not.
- If any number in the series appear to be the same as n, n is Fibonacci.

Question 3 Zexin’s solutions



```python
def is_fib(n):
    a = 0
    b = 1
    while n > b:
        c = a + b
        a = b
        b = c
    return n==b
```

- Order of growth for time complexity is $O(\log n)$
- Order of growth for space complexity is $O(1)$
- The storage space required remains constant as n increases.
- From the exponential growing rate of *fib*, we know that it will approximately take $\log n$ steps to reach n, so the time taken is logarithmically related to n.

Question 4 Discussion

Define a function *make_fare* that takes as arguments stage1, stage2, start_fare, increment, block1, block2 and returns a function that calculates the taxi_fare using those values.

Some obvious features of this function *make_fare*

- Its return value should be a function.
- Its input parameters determine how the function returned will be.
- Also, the internal logic of the return function should be the same.

Question 4 Other source to refer to

```
def taxi_fare(distance): # distance in metres
    if distance <= stage1:
        return start_fare
    elif distance <= stage2:
        return start_fare + (increment *
                             ceil((distance - stage1) / block1))
    else:
        return taxi_fare(stage2) + (increment *
                                    ceil((distance - stage2) / block2))
```

The function your *make_fare* returns should possess same internal logic.

Question 4 Zexin's solution

```
from math import ceil

def make_fare(stage1, stage2, start_fare, increment, block1, block2):
    def taxi_fare(distance): # distance in metres
        if distance <= stage1:
            return start_fare
        elif distance <= stage2:
            return start_fare + (increment *
                    ceil((distance - stage1) / block1))
        else:
            return taxi_fare(stage2) + (increment *
                    ceil((distance - stage2) / block2))
    return taxi_fare
```

How to figure out this solution? Let's do live programming.

Extra stuff: one past midterm question

If time permits, we will go through this. If you all have understood, we shall skip this question and go to the second one.

```
def boo(x):
    return lambda y: x(x(y))
print(boo(boo)(lambda x:x+1)(5))
```

- What is the output of the print statement?
- Don't be intimidated by the complexity of problem first.
- Let's tackle this problem part by part.

Extra stuff: one past midterm question

This question is of the same type as the previous Question 3.

```
def boo(x):
    return lambda y: x(x(y))
print(boo(boo)(lambda x:x+1)(5))
```

Let's tackle the *boo*(*boo*) part first.

- *boo*(*boo*) takes in boo as parameter.
- It will return lambda $y : x(x(y))$ with x replaced by boo.
- So *boo*(*boo*) is equivalent to lambda $y : boo(boo(y))$,
- which is just repeating boo twice for one single argument.

Extra stuff: one past midterm question

```
def boo(x):
    return lambda y: x(x(y))
print(boo(boo)(lambda x:x+1)(5))
```

Let's tackle the *boo*(*boo*)(lambda $x : x + 1$) part then.

- *boo*(*boo*) basically repeats boo twice for one single argument.
- Let's define a function to simplify lambda $x : x + 1$, addOne.
- Now we need to evaluate *boo*(*boo*(*addOne*)) layer by layer.
- *boo*(*addOne*) will return lambda y : *addOne*(*addOne*(y)),
- which is just lambda $y : y + 2$, and let's define it to be addTwo
- Similarly, *boo*(*addTwo*) will be addFour (lambda $y : y + 4$)

Extra stuff: one past midterm question

```
def boo(x):
    return lambda y: x(x(y))
print(boo(boo)(lambda x:x+1)(5))
```

Let's tackle the $boo(boo)(\text{lambda } x : x + 1)(5)$ part then.

- $boo(boo)(\text{lambda } x : x + 1)$ basically add four to the input.
- $addFour(5)$ will return 9.
- Is this answer correct? Let's test it out!

Extra stuff: one more past midterm question

```
x = 5
y = 10

def add_5(y):
    return lambda: y + 5

add_5(x)()
```

- This question is definitely easier than the previous one.
- Takeaway: they may test you on something never taught
- Fret not, most of you should be able to solve it.

Extra stuff: one more past midterm question

```
x = 5
y = 10

def add_5(y):
    return lambda: y + 5

add_5(x)()
```

- What does *lambda* : $y + 5$ mean?
- Let's guess!

Extra stuff: one more past midterm question

```
x = 5
y = 10

def add_5(y):
    return lambda: y + 5

add_5(x)()
```

- Now that we have reached concensus on the *lambda*
- Notice that this question tests on scoping as well.
- Look out for the difference between local y and global y

Extra stuff: one more past midterm question

```
x = 5
y = 10

def add_5(y):
    return lambda: y + 5

add_5(x)()
```

- *add*_5 takes x in as local y, and returns *lambda* : local $y + 5$
- Since x is always 5, it returns *lambda* : 10
- And so *add*_5$(x)()$ returns 10.

Feedback & more

- Slides + relevant material available at:

  `https://github.com/wangzexin/Teaching`

- After the tutorial, if you have further questions:

  `wang.zexin@u.nus.edu`

Thank You

wang.zexin@u.nus.edu