BY: Zhu_Ming

<CS1010S>

Tutorial 3

Recursion & Iteration
Order of Growth
Higher-Order Function

Lecture Recap

1. Recursion vs Iteration
2. Order of Growth
   - Space Complexity
   - Time Complexity
3. Higher-Order Function

1. Order of Growth

- Describe the limiting behavior of a function when the input value approach to a infinity
- Describe proportions of growth of time/space with respect to the growth of input value
- In another words, measure the efficiency of code

How to measure the efficiency??

- By space/memory and time required

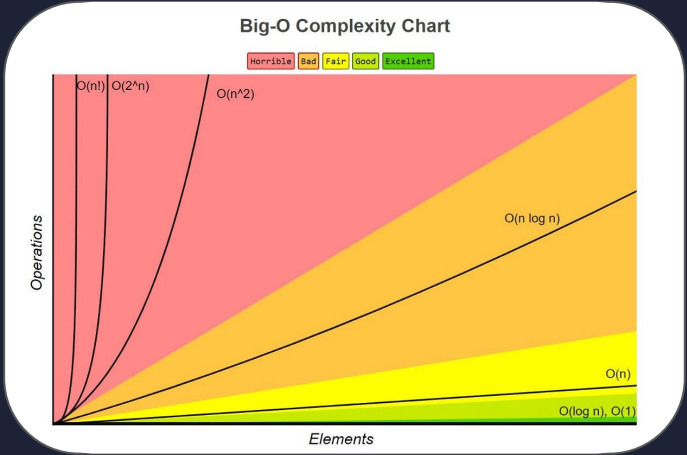
Why not numerical analysis??

Computer	code_iter(i)			code_rec(i)		
	i = 1	i = 10	i = 100	i = 1	i = 10	i = 100
A	1s	10s	100s	1s	1Day	1Year
B	10s	100s	1000s	10s	10Day	10Year

- Is the difference due to computer hardware???
- Is 1s fast or slow??
- How to normalize the data for analysis??
- Is the computer running in optimum condition??
- ...

Why Big-O Notation...

- Because Big-O has NO dependence on the hardware of computer
- Intrinsic properties of the code



Notation	Name
O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
O(nlog(n))	Linearithmic
O(n**k), where k ⩾ 1	Polynomial
O(k**n)	Exponential
O(n!)	Factorial
O(n**n)	Tetration

Time complexity

- Depends on the number of operation / evaluation / leaves

Space complexity

- Depends on the number of pending operations / variables / arrays / string

Important!!

Always evaluate the **worst case scenario**!!

Always take note what is the **n** in O(**n**) we referring to!!

Take the **most significant term**
O(n**2 + 2n) >>> O(n**2)

IMPORTANT SKILL: **CODE TRACING**
https://pythontutor.com/visualize.html#mode=edit

```python
def f(n):
    return 2 ** n
```

Time Complexity: O(1)
Space Complexity: O(1)

Computer can do arithmetic operation very fast

```python
def f(n):
    result = 0
    for i in range(100):
        result += 1
    return result
```

Time Complexity: O(1)
Space Complexity: O(1)

For loop is independent to the input!!

```python
def f(n):
    result = 0
    for i in range(n):
        result += 1
    return result
```

Time Complexity: O(n)
Space Complexity: O(1)

Lecture_Recap.tut03

```python
def f(n):
    if n == 0:
        return 1
    return 1 + f(n - 1)
```

Time Complexity: O(n)
Space Complexity: O(n)

```python
def f(n):
    result = 0
    for i in range(0, n - 1):
        result += 1
    return result
```

Time Complexity: O(n-1) >>> O(n)
Space Complexity: O(1)

Simplify by taking the most significant term

```python
def f(n):
    if n == 0:
        return 1
    else:
        return f(n - 1) + f(n - 1)
```

Time Complexity: O(2**n)
Space Complexity: O(n)

It is a binary tree!!

```python
def f(n):
    if n == 0:
        return 1
    else:
        temp = f(n - 1) O(n)
        return temp + temp O(1)
```

Time Complexity: O(n)
Space Complexity: O(n)

Notice that this is a linear tree!!

```python
def count_string(s):
    result = 0
    for i in range(len(s)):
        result += 1
    return result
```

Time Complexity: O(n), where n is len(s)
Space Complexity: O(1)

```
def concatenation(n):
    str_ = “”
    lst_ = []
    tup_ = ()
    for i in range(n): O(n)
        str_ += “1”    O(n)
        lst_ += [1]    O(n)
        tup_ += (1,)   O(n)
```

```
 Time Complexity: O(n * 3n) >>> O(3*n**2)
                  >>> O(n**2)
Space Complexity: O(n + n + n) >>> O(3n)
                  >>> O(n)

Concatenation is not efficient!!
```

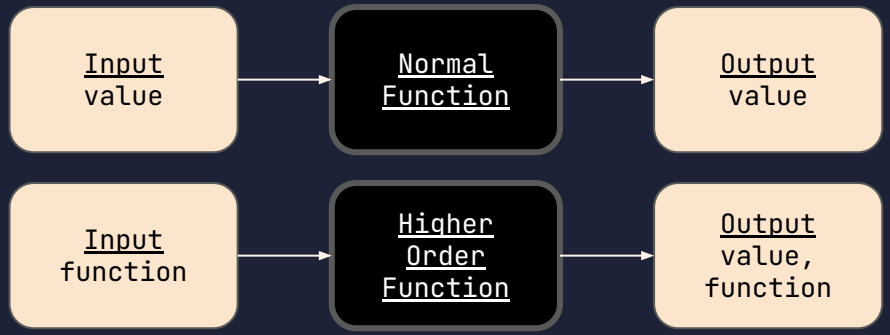
```
def concatenation(n):
    lst_ = []
    for i in range(n): O(n)
        lst_.append(1) O(1)
```

```
 Time Complexity: O(n * 1) >>> O(n)
Space Complexity: O(1 + 1 … n times) >>> O(n)

List.append is more time efficient than
concatenation +=!!
```

More information
1) Python Website
2) Other resources

2. Higher-Order Function



- Extract common pattern
- Make the code more generic

HOF return value

```
def general(func):
    def helper():
        return func(1)
    return helper

general(add_one) # 2
```

HOF return function

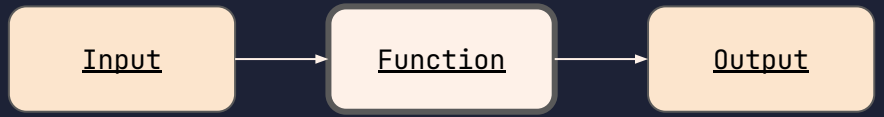
```
def general(func):
    def helper(x):
        return x
    return helper(func)

general(func) # return func
```

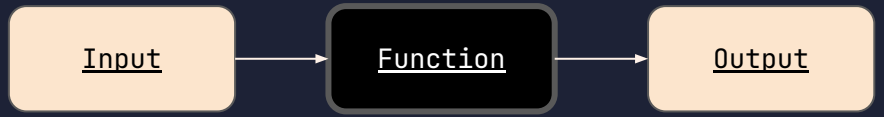
3. Functional Abstraction & Abstraction Barrier

- Client & Implementer relationship/contract

What implementer see:



What client see:



- Once Functional Abstraction is established, **EVERYONE** should follow the abstraction barrier

“Let the function do its job”

BY: Zhu_Ming

Any Questions?

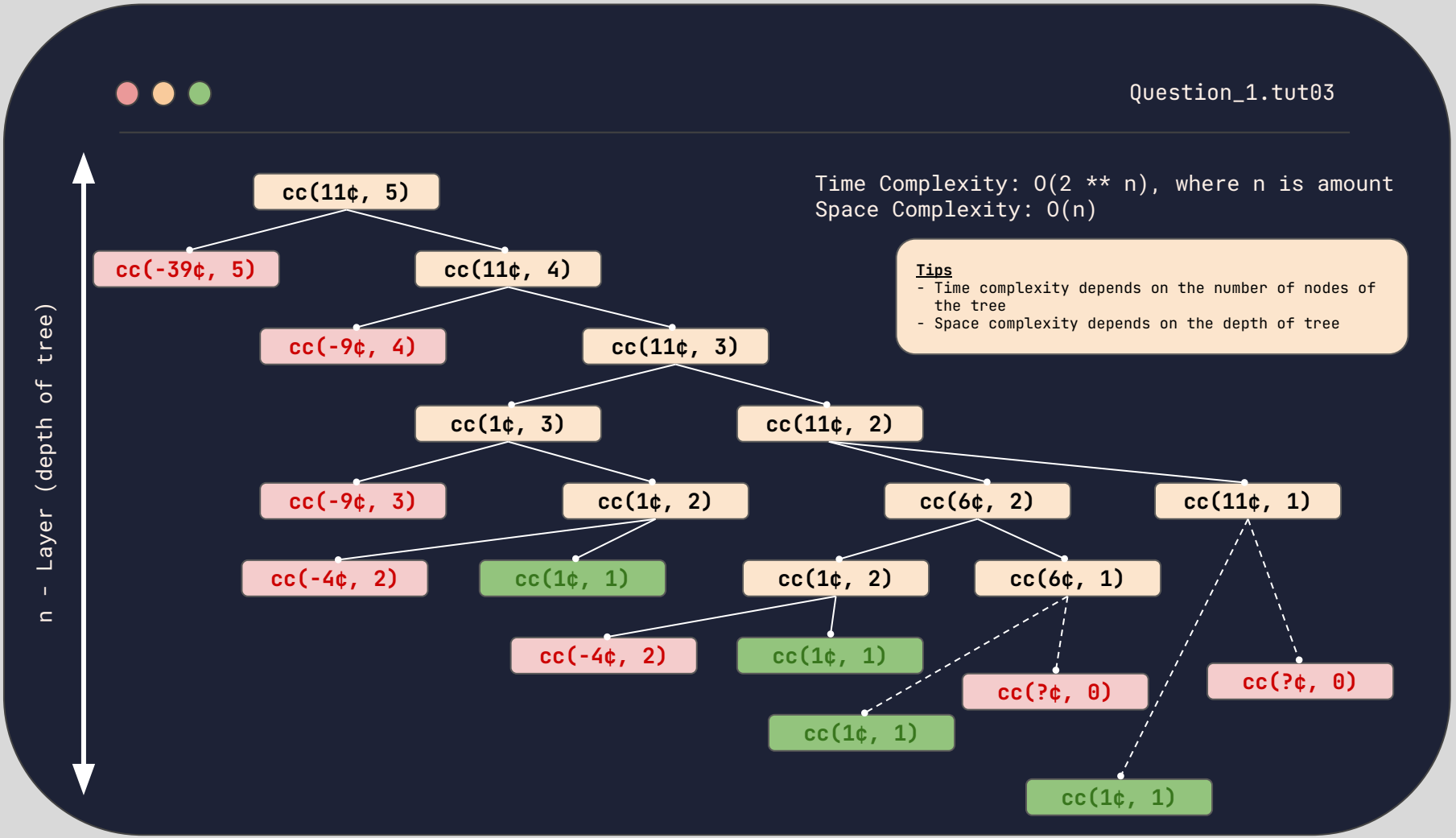
Question_1.tut03

Draw the tree illustrating the process generated by the **cc(amount,d)** function given in the lecture in making change for **11 cents.**

```
def cc(amount, d):
    if amount == 0:  # only one way to make 0¢
        return 1
    elif amount < 0 or d == 0:  # cant make change for negative ¢
        return 0                # cant make any change if no denoms
    else:
        return cc(amount - first_denomination(d), \
                  d) + \
                  cc(amount, d-1)

def first_denomination(d):
    # Functional Abstraction
    . . .
```

Question_1.tut03



Time Complexity: O(2 ** n), where n is amount
Space Complexity: O(n)

Tips
- Time complexity depends on the number of nodes of the tree
- Space complexity depends on the depth of tree

Question_2.tut03

$$f(n) = \begin{cases} n & n < 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & n \geq 3 \end{cases}$$

Write a function **f(n)** that computes f by a **recursive** process.

```
def f(n):
    if n < 3:  # only one way to make 0¢
        return n
    else:
        return f(n - 1) + 2 * f(n - 2) + 3 * f(n - 3)
```

```
Time Complexity: O(3 ** n)
Space Complexity: O(n)
```

$$f(n) = \begin{cases} n & n < 3 \\ f(n-1) + 2f(n-2) + 3f(n-3) & n \geq 3 \end{cases}$$

Write a function **f(n)** that computes f by an **iterative** process.

```
def f(n):
    if n < 0:  # only one way to make 0¢
        return n
    else:
        f0, f1, f2 = 2, 1, 0
        for i in range(n - 2):
        fn = f0 + (2 * f1) + (3 * f2)
        f0, f1, f2 = fn, f0, f1
    return fn
```

Time Complexity: O(n-2) >>> O(n)
Space Complexity: O(1)

Question_4.tut03

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.

```
def is_fib(n):
    if n < 0:
        return False #not fib
    elif n == 0 or n == 1:
        return True  #is fib
    a, b = 0, 1
    while b < n:  #generate fib
        a, b = b, a + b
        if b == n:
            return True
    return False
```

Time Complexity: O(log(n))
Space Complexity: O(1)

Question_5.tut03

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

```
def make_fare(stage1, stage2, start_fare, increment, block1, block2):
    def taxi_fare(distance):
        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

make_fare(1000, 10000, 3.0, 0.22, 400, 350)(3500)
>>> taxi_fare(3500)
>>> 4.54
```

Important!!
stage1, stage2, start_fare, increment, block1, block2 is **GLOBAL** with respect to **taxi_fare**, but **LOCAL** to **make_fare**

distance is **LOCAL** to taxi_fare, but **UNDEFINED** to **make_fare**

Extra_Question.tut02

EXTRA Practices

EXTRA_Question.tut03

(EXTRA)

Order of Growth

QUESTION 1

```
def isprime(n):
    if n <= 1:
        return False
    for i in range(2, int((n**0.5)+1)):
        return bool(n % i)
    return True
```

EXTRA_Question.tut03

```
 Time complexity: O(n**0.5)
Space complexity: O(1)
```

EXTRA_Question.tut03

(EXTRA)

Order of Growth

QUESTION 2

```
def weird(n):
    b = 0
    for i in range(n):
        for j in range(i, n):
            b += 1
```

EXTRA_Question.tut03

Time complexity: O(n ** 2)
Space complexity: O(1)

EXTRA_Question.tut03

(EXTRA)

Order of Growth

QUESTION 3

```
def weird(n):
    b = []
    for i in range(n):
        b.append(1)

    b = []
    for i in range(n): O(n)
        b += [1]       O(n)
```

EXTRA_Question.tut03

```
 Time complexity: O(n**2)
Space complexity: O(n)
```

EXTRA_Question.tut03

(EXTRA)

Order of Growth

QUESTION 4

```
def what_is_in(s, str_):
    return s in str_
```

EXTRA_Question.tut03

Time complexity: O(n)
Space complexity: O(1)

EXTRA_Question.tut02

(EXTRA)

Nested Lambda Function

Question 3

```
def f(x):
    return lambda y: (x, y(x))

def g(y):
    return lambda x: x(y)

print(g(2)(f)(lambda x: x + 1))
```

EXTRA_Question.tut02

OUTPUT:

(2, 3)

BY: Zhu_Ming

Any Questions?

Thank You!!

The End

See you next lesson