Master of Science in
Genomics and Bioinformatics

Lecture 7 (I) Recursion

GNBF5010

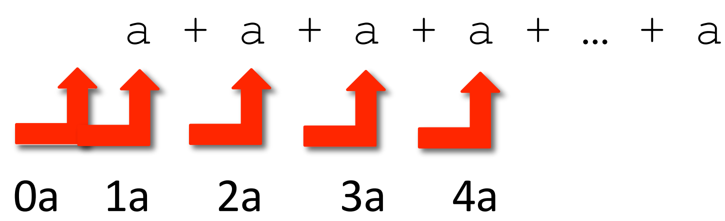
Instructor: Jessie Y. Huang

What is recursion?

- Algorithmically: a way to design solutions to problems by **divide-and-conquer** or **decrease-and-conquer**
  - reduce a problem to simpler versions of the same problem
- Semantically: a programming technique where a **function calls itself**
  - In programming, the goal is to NOT have infinite recursion
    - must have **1 or more base cases** that are easy to solve
    - must solve the same problem on **some other input** with the goal of simplifying the larger problem input

Multiplication – iterative solution

- "multiply `a` * `b`" is equivalent to "add `a` to itself `b` times"



- capture **state** by
  - an **iteration** number (`i`) starts at b

    `i` ← `i-1` and stop when 0
  - a current **value of computation** (`result`)

    `result` ← `result + a`

```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

Multiplication – recursive solution

- **recursive step**
  - think how to reduce problem to a **simpler/ smaller version** of same problem
- **base case**
  - keep reducing problem until reach a simple case that can be **solved directly**
  - when b = 1, a*b = a

$$a*b = \underbrace{a + a + a + a + \ldots + a}_{b\ \text{times}}$$

$$= a + \underbrace{a + a + a + \ldots + a}_{b\text{-}1\ \text{times}}$$

$$= a + \boxed{a * (b-1)}$$

recursive reduction

```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

base case

recursive step

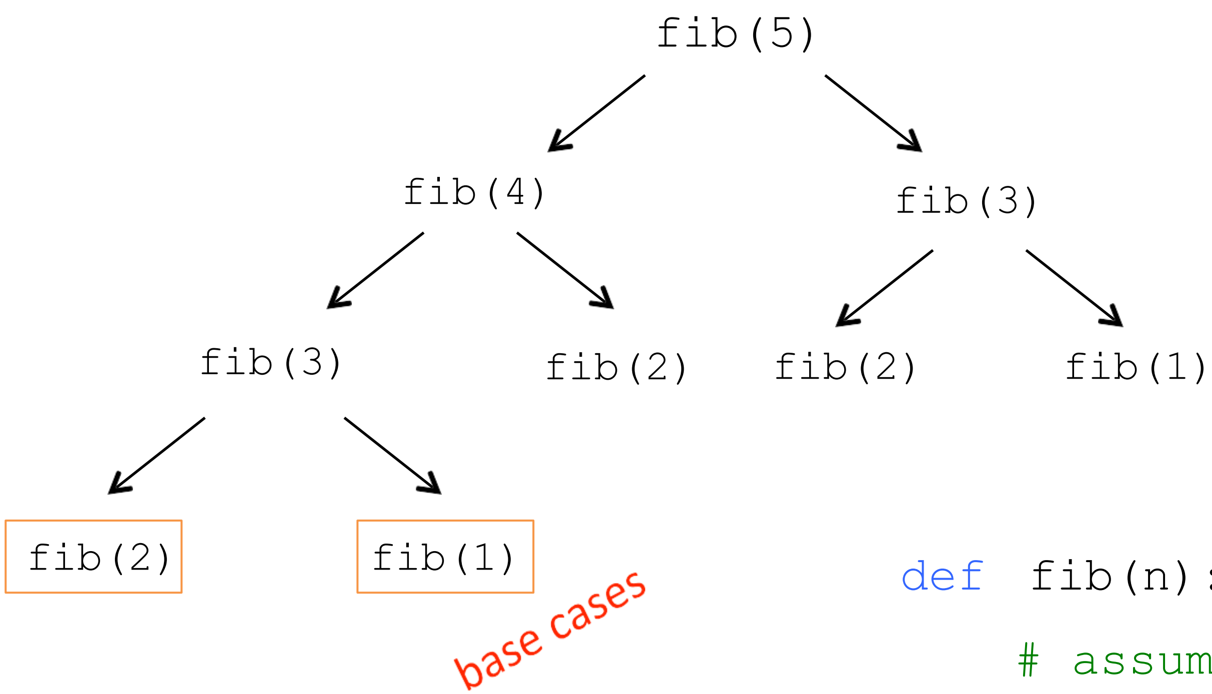
Inductive reasoning

- How do we know that our recursive code will work?
- `mult()` called with b > 1 makes a recursive call with a smaller version of b
- It must eventually reach the call with b=1

```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

Example 1: Fibonacci numbers

fib(n) = fib(n-1) + fib(n-2)




```
def fib(n):
    # assume n >= 1
    if n == 1 or n ==2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Example 2: Factorial

`n! = n*(n-1)*(n-2)*(n-3)* … * 1`

- for what `n` do we know the factorial?

  n = 1 →

```
if n == 1:
    return 1
```

base case

- how to reduce problem? Rewrite in terms of something simpler to reach base case

  n*(n-1)! →

```
else:
    return n*factorial(n-1)
```

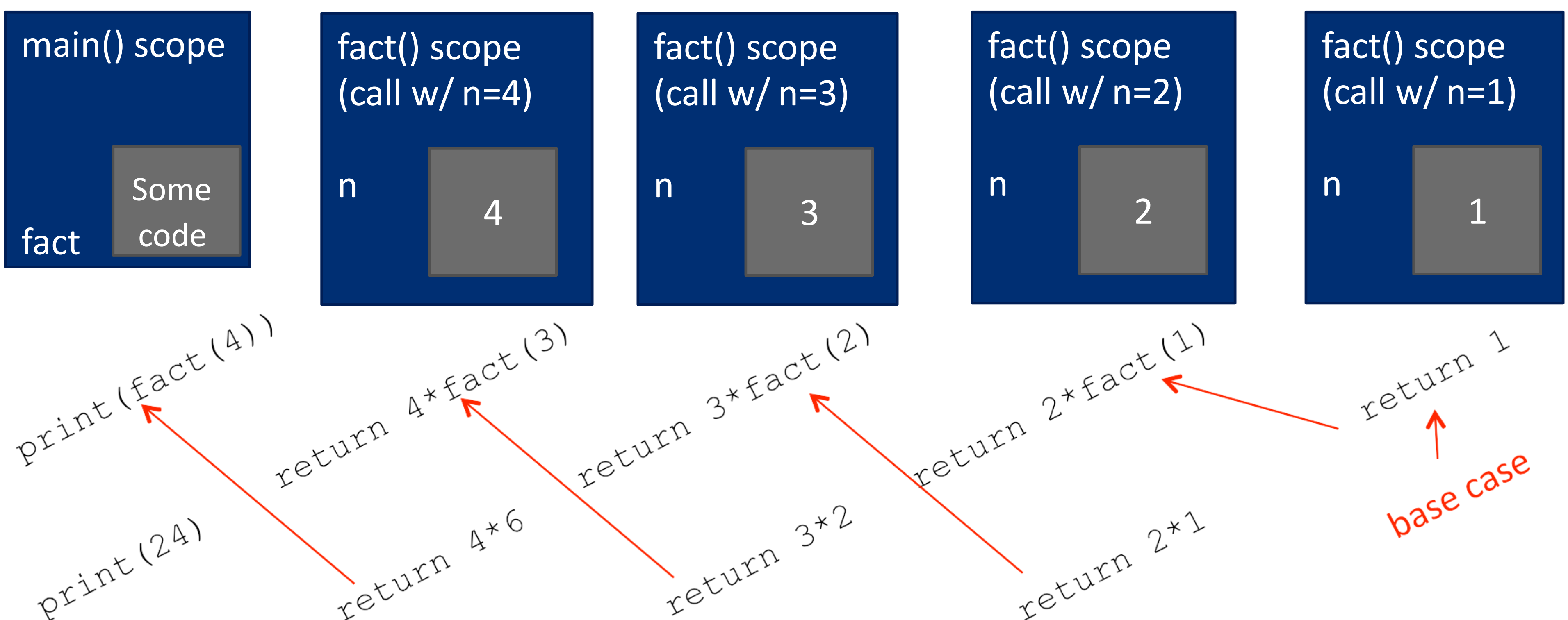
recursive step

Recursive function scope

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)

def main():
    print(fact(4))

main()
```



Iteration vs. Recursion

```
def factorial_iter(n):
    prod = 1
    for i in range(1,n+1):
        prod *= i
    return prod
```

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
```

- Every **recursion** can be implemented with **iteration**
- **Recursion** is usually **slower**
  - as function calls are stored in a stack to allow return to the caller
- Infinite **recursion** can lead to **system crash**
  - whereas infinite iteration consumes CPU cycles
- Reasons to use recursion
  - Some problems are more easily solved with recursion than with a loop.
  - For example, computing **factorials** and **Fibonacci numbers**, where the mathematical definition lends itself to the recursion, and searching binary trees.

Reading

- Chapter 12 of Starting Out with Python, 4th Edition