w08-Lec1

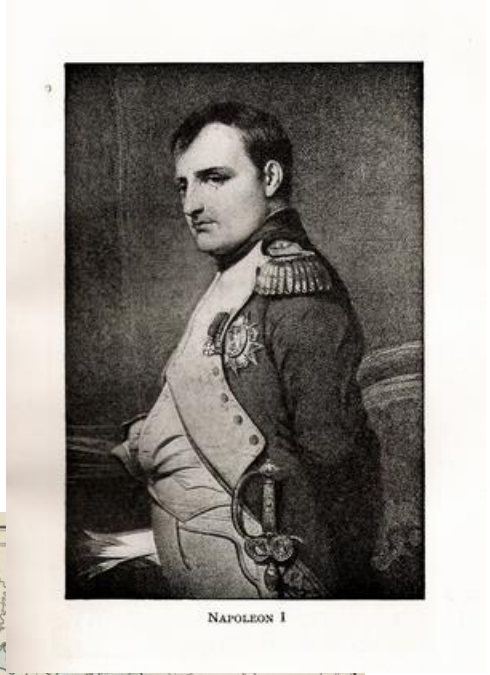
Recursion Part I

for 204111

Kittipitch Kuptavanich

Divide and Conquer

- หรือ **Divide and Rule**
  - ในทางประวัติศาสตร์และการปกครอง
    คือการสร้างอำนาจ
    หรือรักษาอำนาจไว้ โดยการ
    - แบ่งเป้าหมาย
      เป็นหน่วยเล็ก ๆ
      ที่มีกำลังน้อยกว่า (Divide)
    - แล้วเข้ายึดอำนาจ
      ทีละส่วน (Conquer)



NAPOLEON I

MAP TO ILLUSTRATE THE DOMINION OF NAPOLEON I.,
(IRRESPECTIVE OF OCCUPIED TERRITORIES) AT THE TIME OF HIS GREATEST POWER.

Recursion

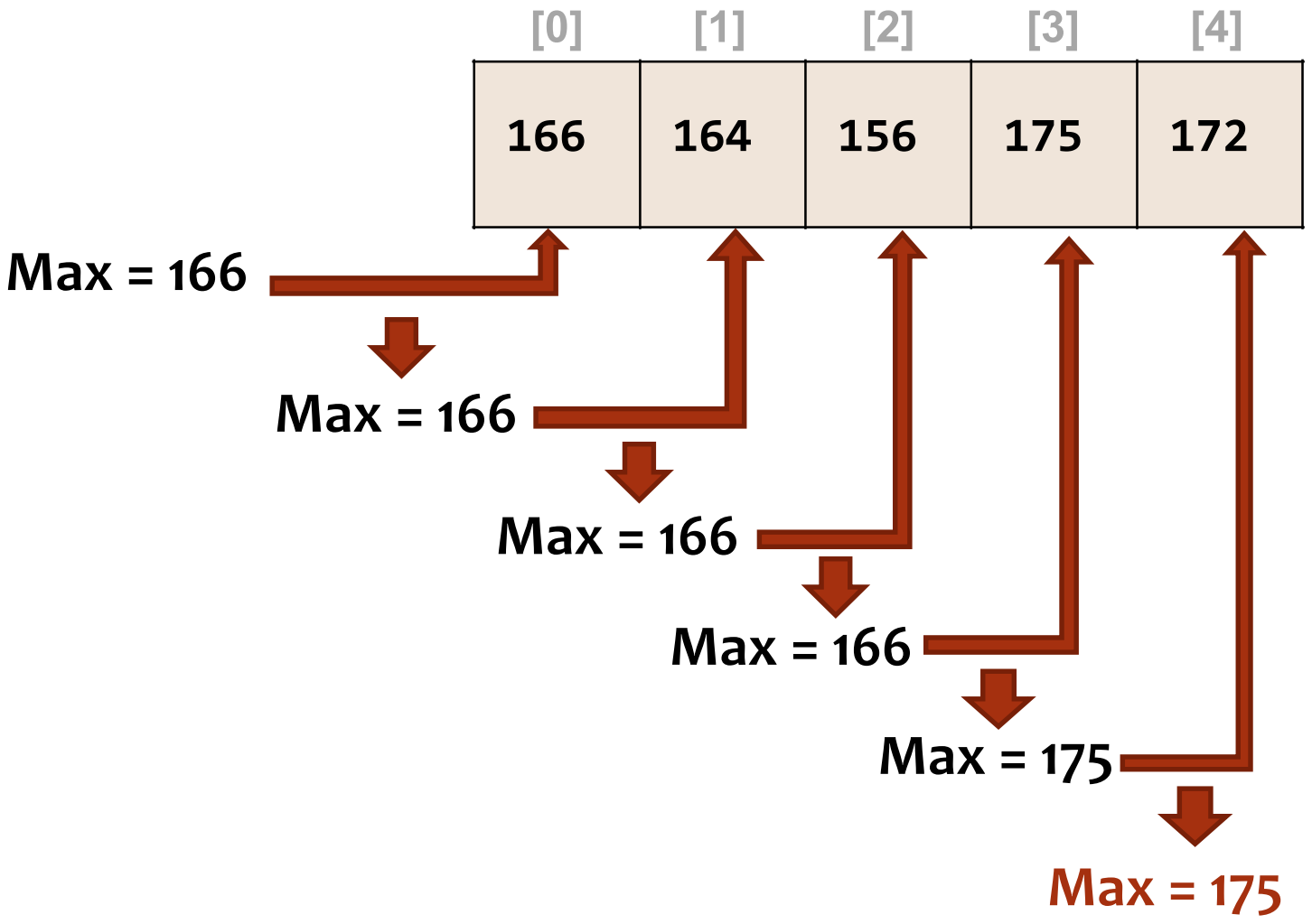
- **Recursion (หรือการเวียนเกิด) เป็นการใช้หลัก Divide and Conquer ในการแก้ปัญหา**
  - **Divide แบ่งปัญหาที่ต้องการแก้ เป็นปัญหาย่อย (Sub problem) - ควรแบ่งแล้วปัญหาเล็กลงหรือซับซ้อนน้อยลง**
  - **Conquer แก้ปัญหาย่อย – เรียกใช้ function ตัวเอง**
  - **Combine นำคำตอบของปัญหาย่อยมารวมกันเพื่อให้ได้คำตอบของปัญหาหลัก**

Example 1: find_max - Iteration

Load งานอยู่ที่คนคนเดียว

- **Iterative (loop) solution**
  - นำกระดาษมา เขียนค่า max = 0
  - ไล่ถามความสูง (height) ทีละคน
  - ถ้าเจอคนที่ความสูง (height) มากกว่าที่ max จดใน กระดาษ
    - Update ค่า max ในกระดาษเป็นค่า height

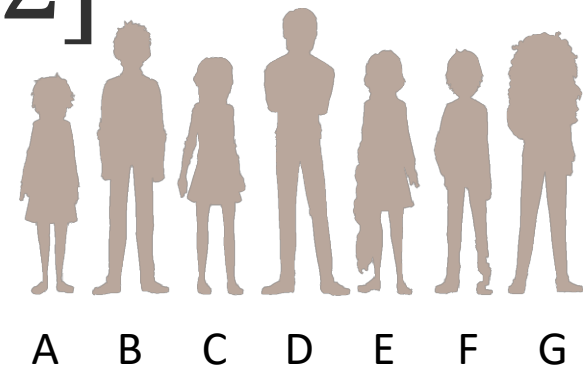
Example 1: find_max - Iteration [2]



Example 1: find_max - Iteration [3]

```
02 from functools import reduce
03
04 list_a = [166, 164, 156, 175, 172, 156, 182, 180, 171, 159]
05
06 # using reduce
07 # the built-in accumulator handles the repetitive task
08
09 max_ = reduce(lambda x, y: x if x > y else y, list_a)
10 print(max_)     # 182
11
```

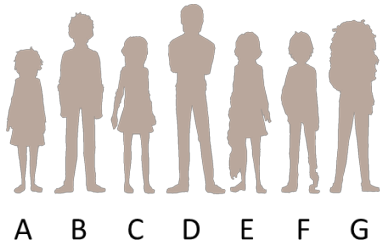
Example 1: find_max [2]



- **Recursive solution**
  - ต้องการหา **max of 7 people**
  - **A** บอกเพื่อนให้ หา `max_of_6` แล้ว **A** จะหา `max_of_7`

  - โดยเทียบความสูงของ **A** และ `max_of_6`

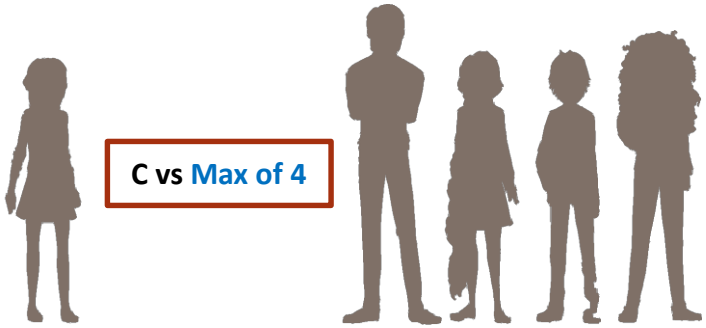
Example 1: find_max [3]

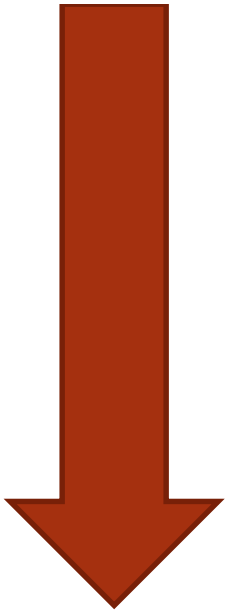


- ต้องการหา max of 6 people
  - B บอกเพื่อนให้ หา max_of_5 แล้ว B จะหา max_of_6

ปัญหาเล็กลง

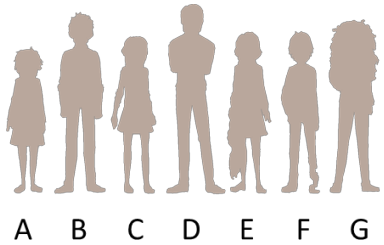
- ต้องการหา max of 5 people
  - C บอกเพื่อนให้ หา max_of_4 แล้ว C จะหา max_of_5





AND So on....

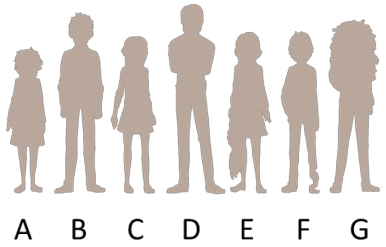
Example 1: find_max [4]



- ต้องการหา max of 1 people
  - G บอกว่า G สูงที่สุดถ้าอยู่คนเดียว max_of_1 = G
  - `return max_of_1` ให้ F

- F ได้ค่า `max_of_1` จาก G
  - F สูงน้อยกว่า `max_of_1`
  - ดังนั้น `max_of_2` = `max_of_1`
  - `return max_of_2` ให้ E

Example 1: find_max [5]



- E ได้ค่า `max_of_2` จาก F
  - E สูงน้อยกว่า `max_of_2`
  - ดังนั้น `max_of_3` = `max_of_2`
  - `return max_of_3` ให้ D

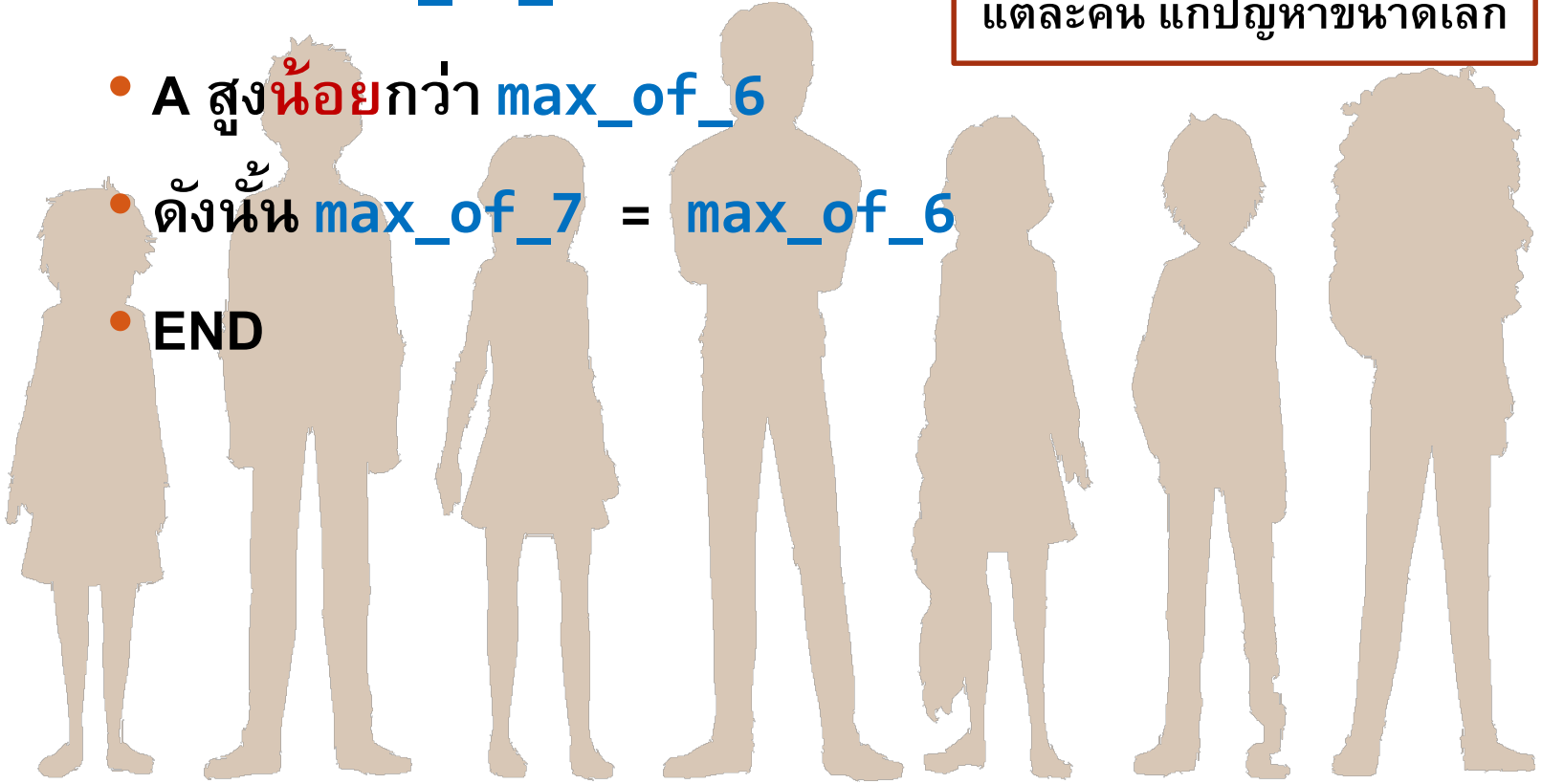
- D ได้ค่า `max_of_3` จาก E
  - D สูงมากกว่า `max_of_3`
  - ดังนั้น `max_of_4` = D
  - `return max_of_4` ให้ C

AND So on….

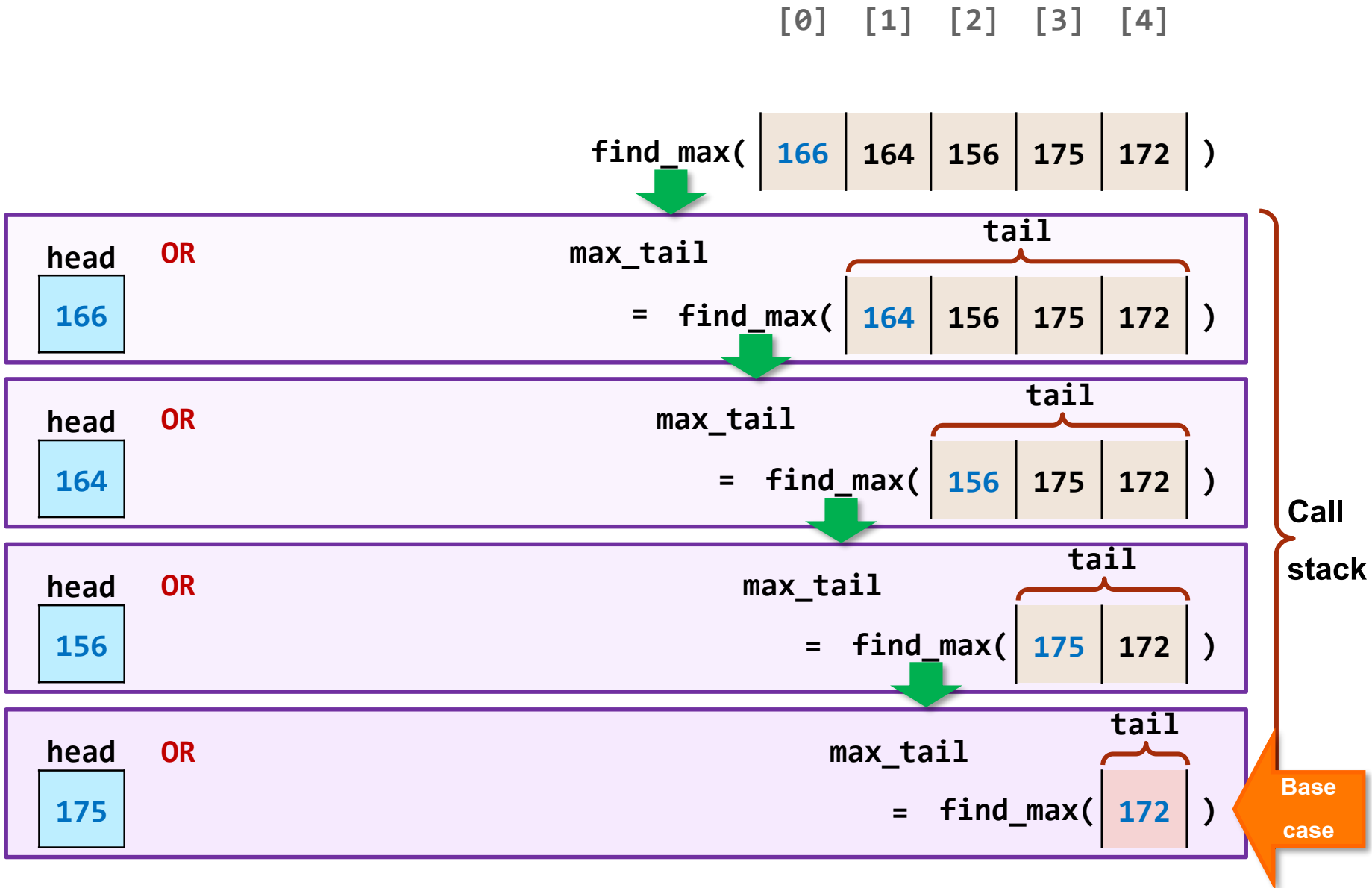
Example 1: find_max [6]

- A ได้ค่า `max_of_6` จาก B
  - A สูง**น้อย**กว่า `max_of_6`
  - ดังนั้น `max_of_7 = max_of_6`
  - END

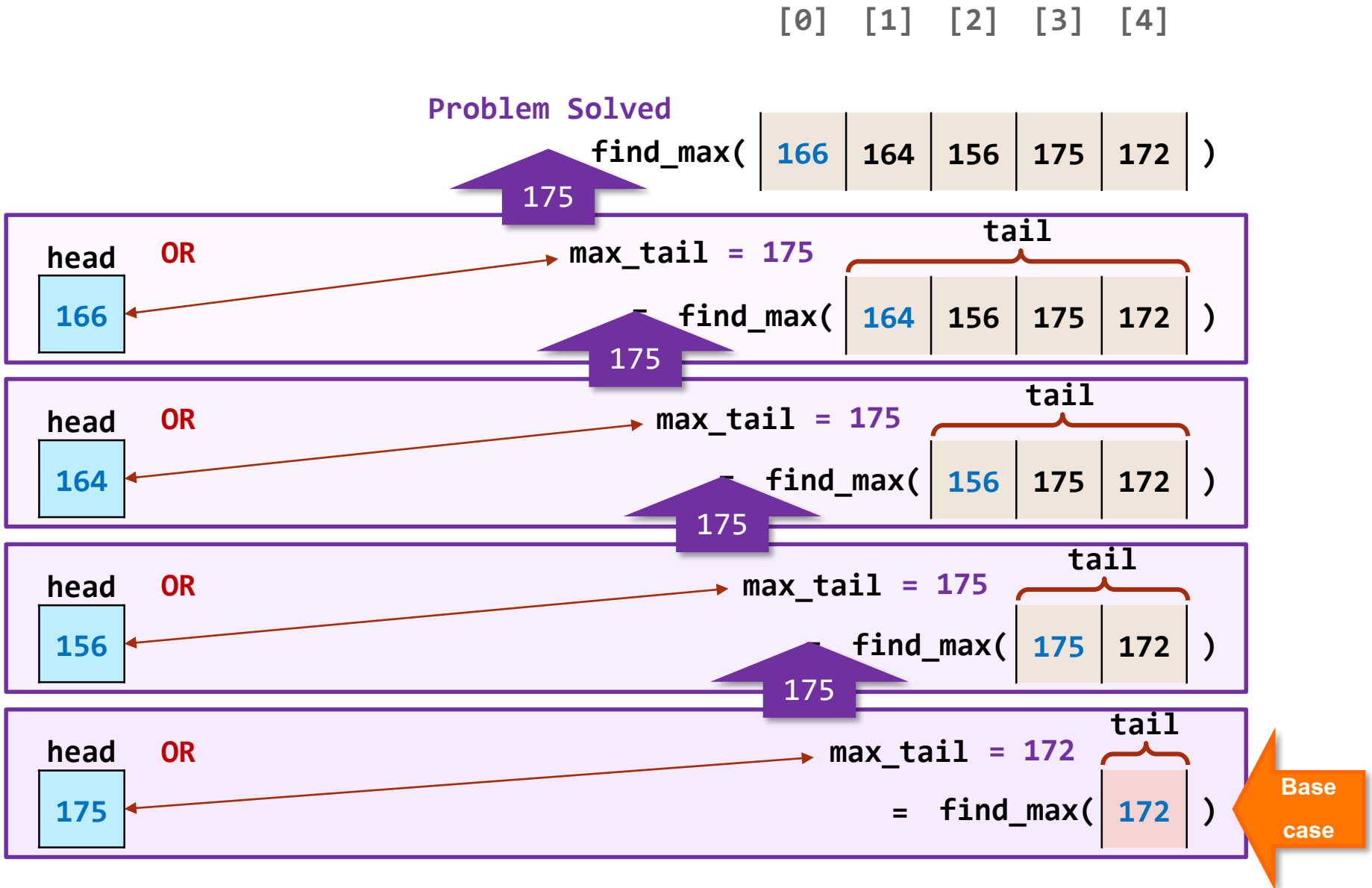
Load งานกระจาย
แต่ละคน แก้ปัญหาขนาดเล็ก



Example 1: find_max [7]



Example 1: find_max [8]



Example 1: find_max [9]

```
def find_max(list_a):
```

ต้องมี base case

```
  if len(list_a) == 1:
    return list_a[0]
```

```
  # แบ่งเป็นปัญหาที่เล็กลง (divide & conquer)
  head = head(list_a)
  tail = tail(list_a)
  max_tail = \
    find_max(tail)

  # นำคำตอบมารวมกัน (combine)
  if max_tail > head:
    return max_tail
  else:
    return head
```

```
  # แบ่งเป็นปัญหาที่เล็กลง (divide & conquer)
  head = list_a[0]
  tail = list_a[1:]
  max_tail = \
    find_max(tail)

  # นำคำตอบมารวมกัน (combine)
  if max_tail > head:
    return max_tail
  else:
    return head
```

Recursive Algorithm

Definition

- **An algorithm is called recursive if it solves a problem by *reducing it to an instance of the same problem with smaller input.* (Divide and conquer)**

General Structure

```
output recurse(arguments) {
  // PART 1 base case = terminate
  // ถ้าปัญหาเล็กพอที่จะ solve ได้ - ไม่จำเป็นต้องแบ่งอีกต่อไป)
  if smallEnough(arguments)
    return answer

  // PART 2 divide and conquer (แบ่งปัญหาและเรียกใช้ function ตัวเอง)
  myWorkLoad = someFunction(arguments)
  answerFromSubproblem = recurse(smallerArguments)

  // PART 3 combine (นำคำตอบมารวมกัน)
  answer = combine(myWorkLoad,answerFromSubproblem);

  return answer
}
```

ต้องแบ่งแล้วปัญหาเล็กลง หรือซับซ้อนน้อยลง และวิ่งเข้าสู่ base case

Adapted From:
http://ocw.mit.edu/courses/civil-and-environmental-engineering/1-00-introduction-to-computers-and-engineering-problem-solving-spring-2012

Example 2: Factorial

พิจารณา 5! และ 4!

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$
$$4! = \quad 4 \times 3 \times 2 \times 1$$

Define n! แบบ recursive

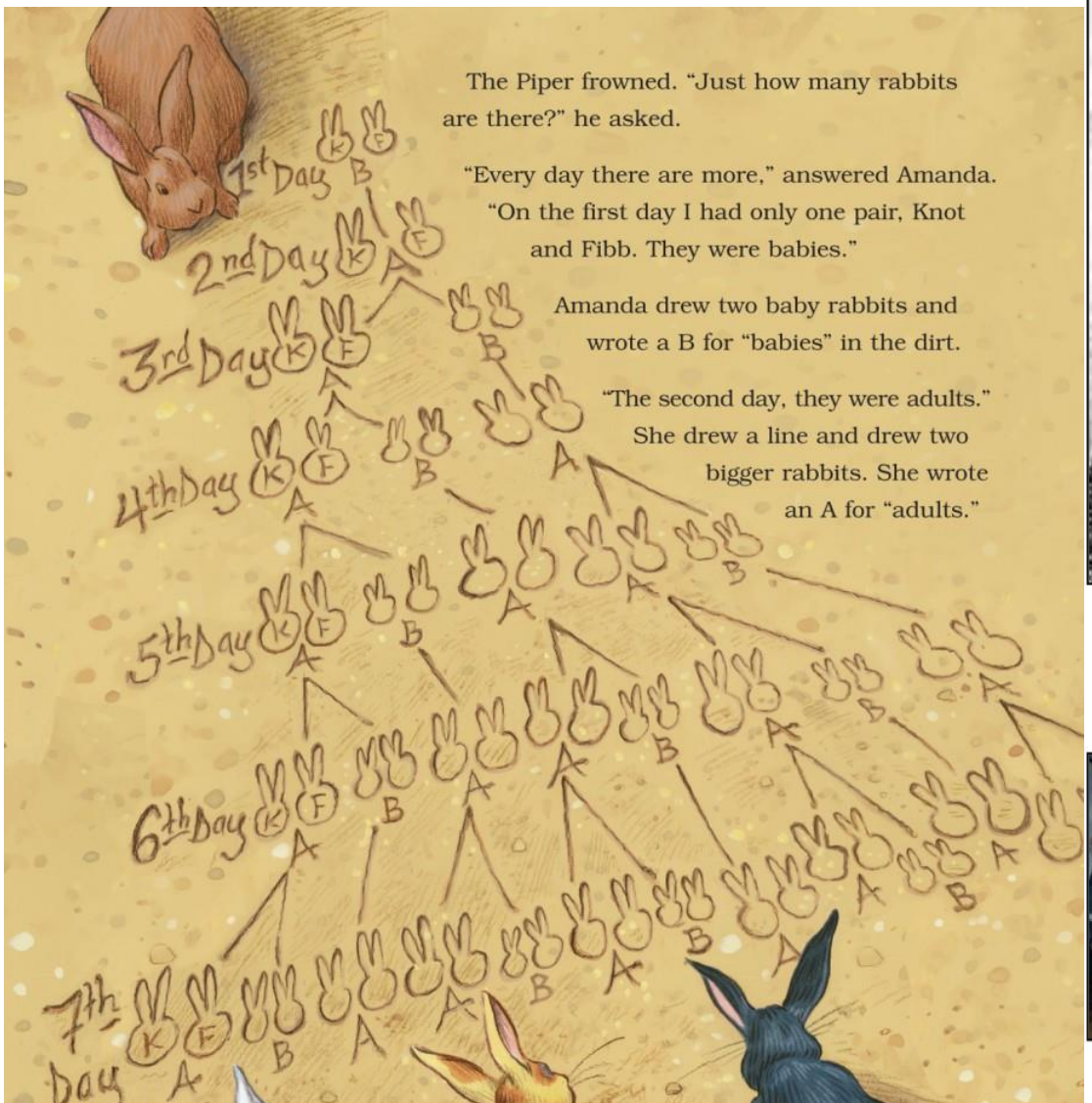
$$5! = 5 \times 4!$$

Base case? หยุดที่ 1

$$n! = \begin{cases} 1, & n = 0 \\ n \times (n-1)!, & n > 0 \end{cases}$$

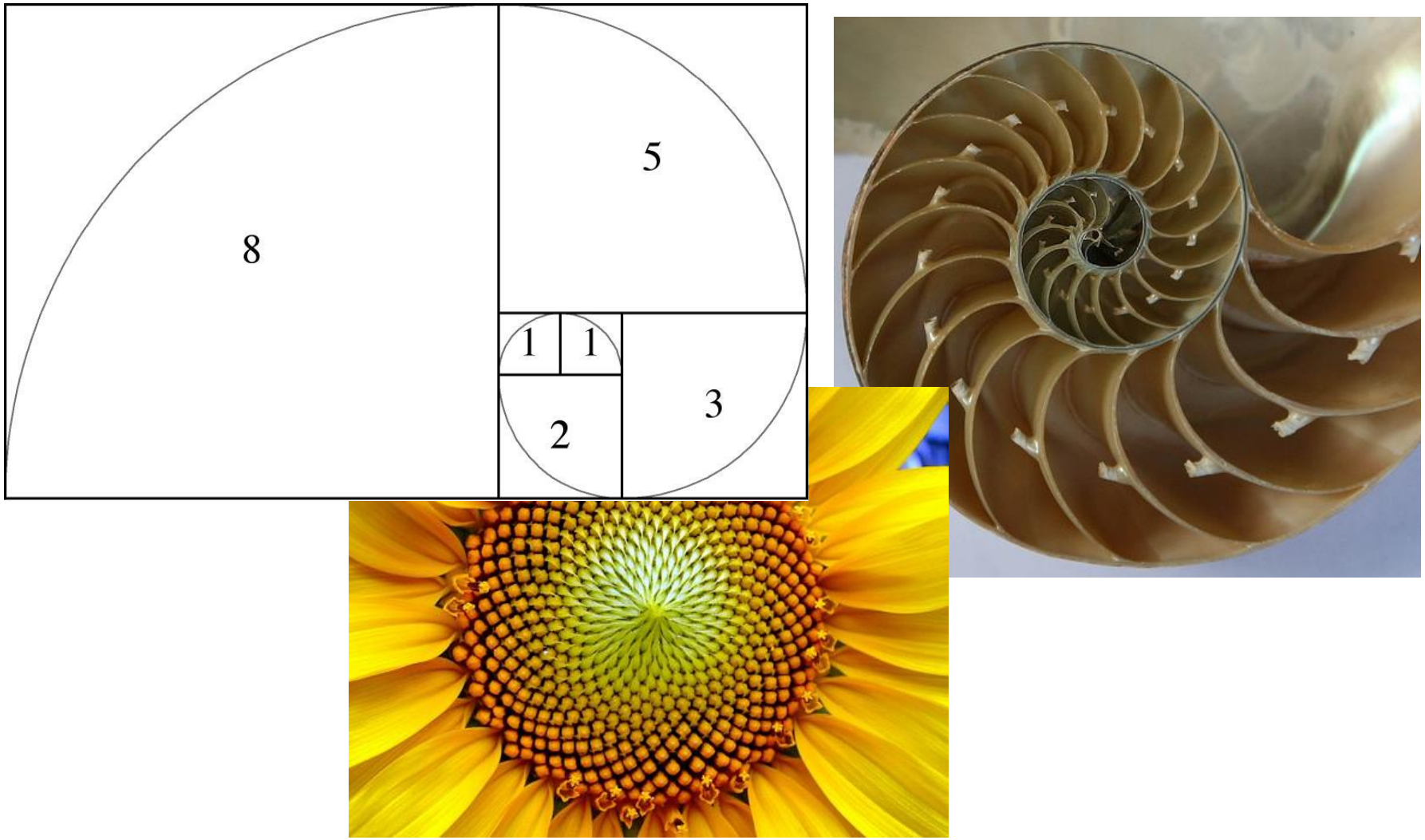
```
def factorial(x):
    # base case


    # divide & conquer


    # combine
```

Fibonacci… and his rabbits



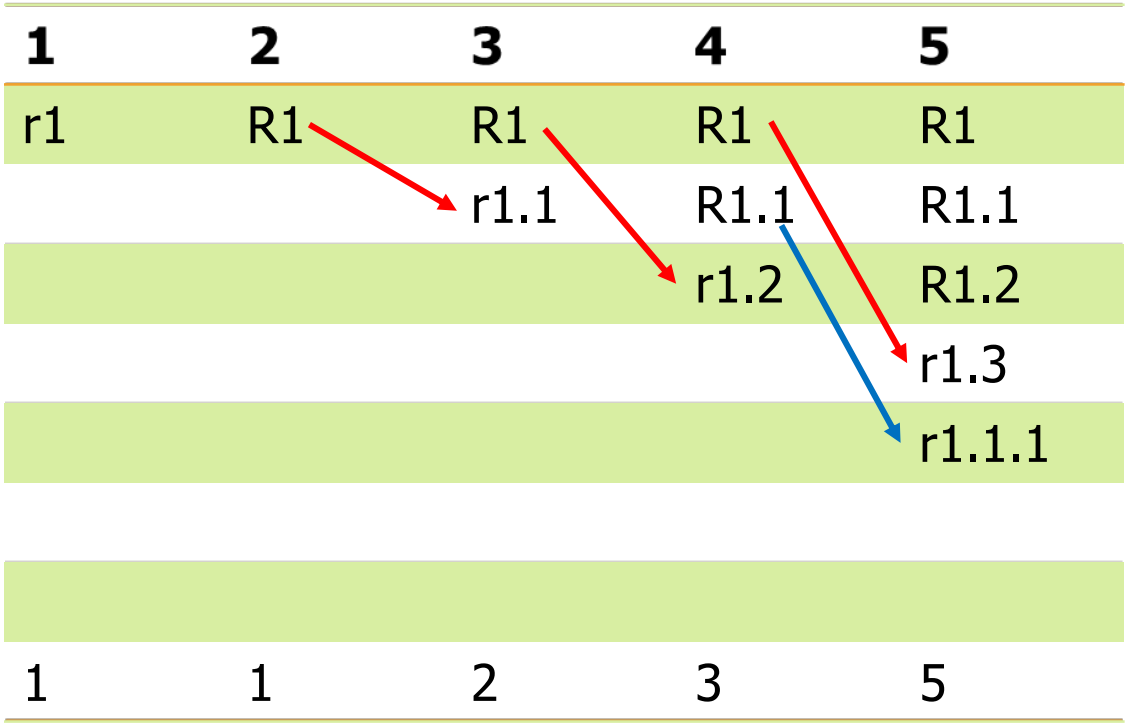
OK, OK… Let's talk rabbits…

Example 3: Fibonacci Sequence

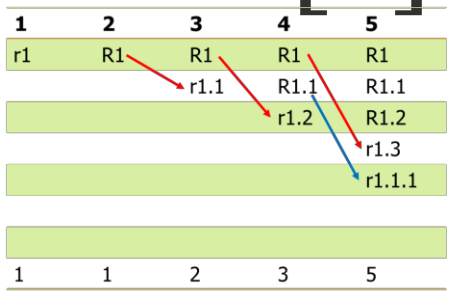


Example 3: Fibonacci Sequence [2]

- ลูกกระต่าย 1 ตัว
  - ใช้เวลา 1 เดือนจะโตเต็มวัย
- กระต่ายโตเต็มวัย 1 ตัว
  - ใช้เวลา 1 เดือนคลอดลูก 1 ตัว



Example 3: Fibonacci Sequence [3]



- **จำนวนกระต่ายในเดือนนี้**
  = จำนวนกระต่ายเดือนที่แล้ว + จำนวนกระต่ายเกิดใหม่เดือนนี้
  = จำนวนกระต่าย*เต็มวัย*เดือนที่แล้ว
  = จำนวนกระต่ายสองเดือนที่แล้ว

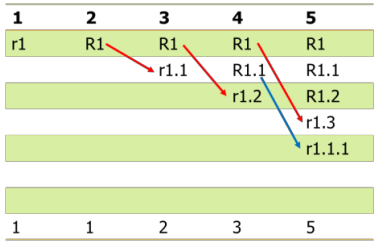
- **จำนวนกระต่ายในเดือนนี้**
  = จำนวนกระต่ายเดือนที่แล้ว + จำนวนกระต่ายสองเดือนที่แล้ว

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

- **1 1 2 3 5 8 13 21 .....**

Example 3: Fibonacci Sequence [4]

- **Divide:**
  - **Number of Rabbit of month (**
    - **Fib(n) = R + r**
- **Conquer**
  - `R = Fib(n - 1)`
  - `r = Fib(n - 2)`
- **Combine**
  - **total = R + r**
- **Base Case**
  - **Fib(1) = 1**
  - **Fib(2) = 1**



Fib (n) = Fib (n – 1) + Fib (n – 2)

```
03 def fib(x):
04
05     if x == 1:
06         return 1
07
08     if x == 2:
09         return 1
10
11
12     R = fib(x - 1)
13     r = fib(x - 2)
14
15
16     total = R + r
17     return total
```

The Three Laws of Recursion

1. A recursive algorithm must have a base case (or base cases).
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

Credit: http://interactivepython.org/courselib/static/pythonds/Recursion/recursionsimple.html

Mathematical Induction

- **Recursive programming is directly related to *mathematical induction***
- **The *base case* is to prove the statement true for some specific value or values of N.**
- **The *induction step* -- assume that a statement is true for all positive integers less than N, then prove it is true for N.**

Example 4: Prime Factor

ให้เขียนฟังก์ชัน `prime_factor(`x`)` **เพื่อแสดงค่าตัวประกอบเฉพาะของ integer** x $(x > 1)$ **โดยใช้ Recursion**

Input	**Output**
360	2 2 2 3 3 5
17	17

Factorization – Recap

- ในการหาตัวประกอบของ integer n เราสามารถใช้วิธีการลองหาร n ด้วยจำนวนเฉพาะ

$$k = 2, 3, 5, 7, 11, 13, \ldots .$$

- ถ้า n หารด้วย k ลงตัว $\rightarrow$ k เป็น factor ของ n
  - ทำการหารอีกครั้งด้วย k
- ถ้า n หารด้วย k ไม่ลงตัว
  - ลองจำนวนเฉพาะตัวถัดไป

Factorization – Recap [2]

- ตัวอย่าง 2394

List of primes: 2, 3, 5, 7, 11, 13, ...
more at: http://primes.utm.edu/lists/small/1000.txt

1. 2394/2 = 1197
2. **Can't divide** by 2 **again so try** 3
3. 1197/3 = 399
4. 399/3 = 133
5. **Can't divide** by 3 **again so try** 5
6. **Can't divide** by 5 so try 7
7. 133/7 = 19 (19 **is prime so we are done**)

$$2394 = 2 \times 3 \times 3 \times 7 \times 19$$

Factorization – Recap [3]

Notes:

- จำนวนเฉพาะมีมากมายไม่จำกัด
- เป็นไปไม่ได้ ที่จะมี **list** ของจำนวนเฉพาะทั้งหมด
- หรือไม่สามารถหา **list** ของจำนวนเฉพาะได้
- **Solution:**
  - ใช้เลขคี่ตัวถัดไป จากตัวหารปัจจุบัน
  - ทำไมถึงไม่ใช้เลขคู่?
- ทฤษฎี : ตัวประกอบเฉพาะ*ตัวแรก*ของจำนวนเต็มใด ๆ จะต้องมีค่าน้อยกว่าหรือเท่ากับรากที่สองของจำนวนเต็มนั้น ๆ
  - **Why?**
  - ดังนั้นหากตัวหาร k มากกว่า $\sqrt{n}$ แล้วยังไม่สามารถหา k ที่ $k \mid n$ แสดงว่า n เป็นจำนวนเฉพาะ (ควรหยุดหาตัวประกอบต่อ)

Base Case

Recursion Helper Functions

- ในบางกรณี เราจำเป็นต้อง ส่งต่อ **parameter** บางตัวเพื่ออำนวยความสะดวกในการทำ **recursion** ที่ **user** ไม่จำเป็นต้องทราบ หรือ **input** เข้ามา เช่นกรณี `prime_factor()`

```
USER: prime_factor(num)
HELPER: prime_factor_helper(num, divisor)
OR
USER: prime_factor(num, divisor=2)
```

- หรือ กรณี **array/list/string** หากต้องการ **recursive call** ณ ช่วง **index** ที่ย่อยลงไป เนื่องจากเป็นผลของการแบ่งปัญหาเป็น **Subproblem**

Example 4: Prime Factor [2]

```
02 def prime_factor(x):
03     prime_factor_helper(x, 2)
04
05
06 def prime_factor_helper(x, div):
07     # base case
08     if div > x ** 0.5:
09         print(_____x_____)
10         return x
11     # d & c
12     if (_____x mod 2 == 0_____):
13         print(div, end=" ")
14         prime_factor_helper(_____x/div, div_____)
15     else:
16         prime_factor_helper(_____x_____, div+1_____)
```

Reference

- **http://www.kosbie.net/cmu/fall-12/15-112/handouts/notes-recursion/notes-recursion.html**