

# Recursions

Assembled for 204112 by  
Kittipitch Kuptavanich  
Ratsameetip Wita

## Recursion

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

## Divide and Conquer

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



## Example 1: findMax

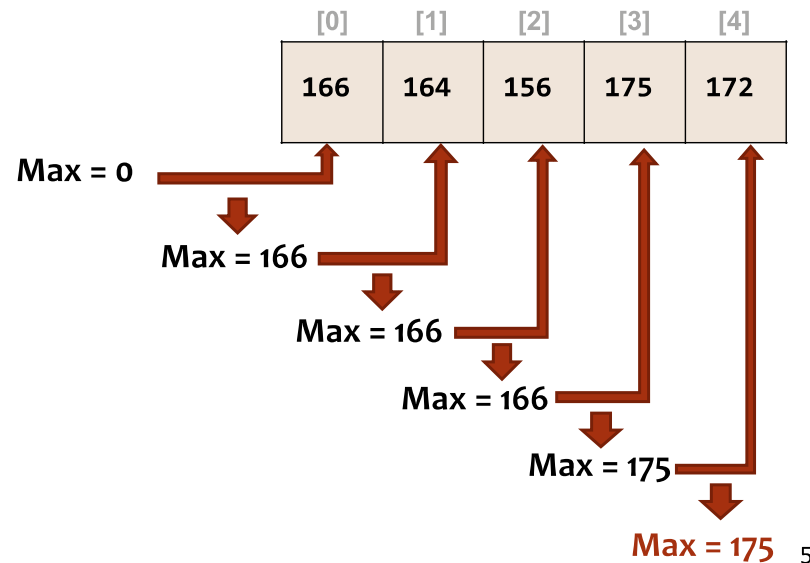
- **Iterative** (loop) solution

Load งานอยู่ที่คนคนเดียว

- นำกระดาดมา เขียนค่า  $\text{max} = 0$
- ไล่ถามความสูง (height) ที่ละคน
- ถ้าเจอคนที่ความสูง (height) มากกว่าที่  $\text{max}$  จดในกระดาด
- Update ค่า  $\text{max}$  ในกระดาดเป็นค่า height



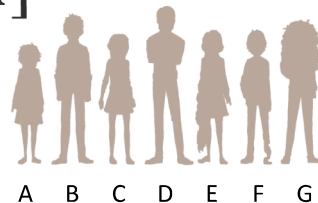
## Example 1: findMax [2]



## Example 1: findMax [4]

### • Recursive solution

- ต้องการหา max of 7 people
- A บอกเพื่อนให้ หา maxOf\_6 แล้ว A จะหา maxOf\_7



- โดยเทียบความสูงของ A และ maxOf\_6

## Example 1: findMax [3]

arrayA =  
 [166, 164, 156, 175, 172, 156, 182, 180, 171, 159]

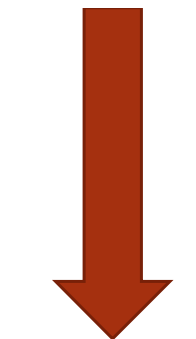
```
function findMaxIt(arrayA)
  maxH = 0
  for num in arrayA
    if num > maxH
      maxH = num
    endif
  endfor
  return maxH
```

## Example 1: findMax [5]

- ต้องการหา max of 6 people
  - B บอกเพื่อนให้ หา maxOf\_5 แล้ว B จะหา maxOf\_6
- ต้องการหา max of 5 people
  - C บอกเพื่อนให้ หา maxOf\_4 แล้ว C จะหา maxOf\_5



ปัญหาเล็กลง



AND So on....

## Example 1: findMax [6]



- ต้องการหา max of 1 people
- G บอกว่า G สูงที่สุดถ้าอยู่คนเดียว maxOf\_1 = G
- Return maxOf\_1 ให้ F



- F ได้ค่า maxOf\_1 จาก G
- F สูงน้อยกว่า maxOf\_1
- ดังนั้น maxOf\_2 = maxOf\_1
- Return maxOf\_2 ให้ E

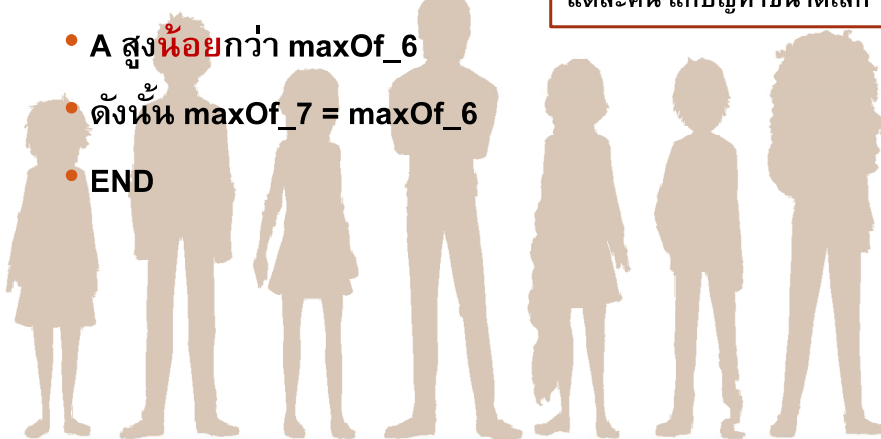


9

## Example 1: findMax [8]

- A ได้ค่า maxOf\_6 จาก B
- A สูงน้อยกว่า maxOf\_6
- ดังนั้น maxOf\_7 = maxOf\_6
- END

Load งานกระจาย  
แต่ละคน แก้ปัญหาลดเล็กลง



11

## Example 1: findMax [7]



- E ได้ค่า maxOf\_2 จาก F
- E สูงน้อยกว่า maxOf\_2
- ดังนั้น maxOf\_3 = maxOf\_2
- Return maxOf\_3 ให้ D



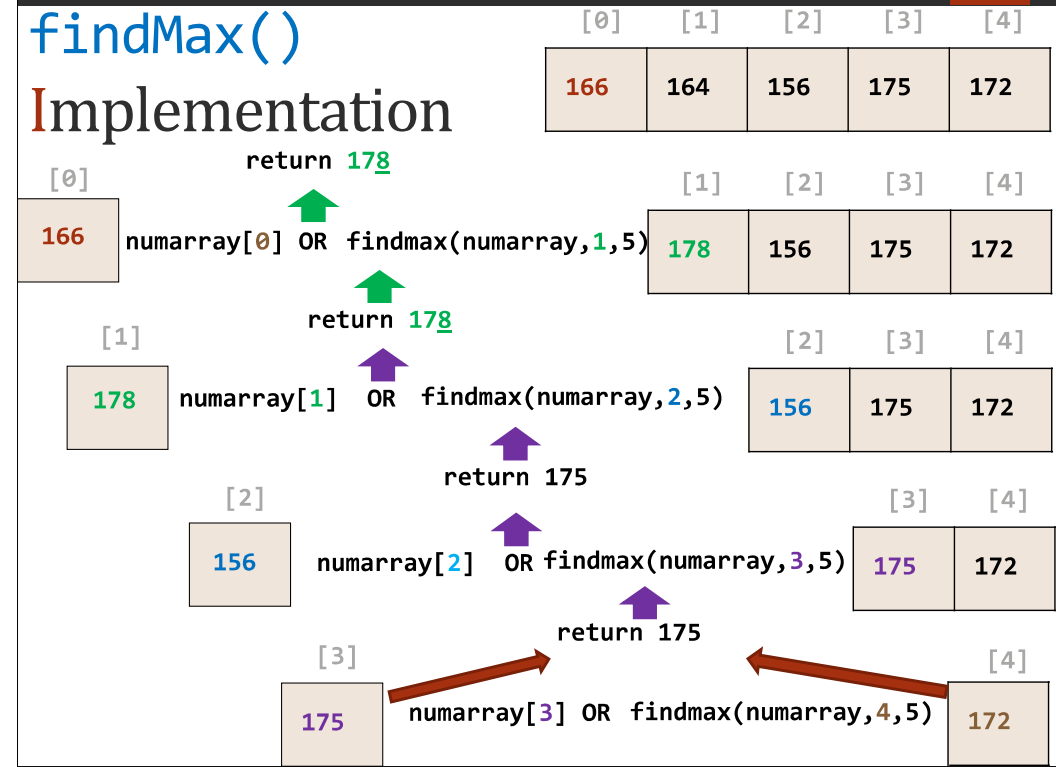
- D ได้ค่า maxOf\_3 จาก E
- D สูงมากกว่า maxOf\_3
- ดังนั้น maxOf\_4 = D
- Return maxOf\_4 ให้ C




AND So on....

## findMax()

### Implementation



## findMax() Implementation [2]

<pre>int findMax(numArray,pos,lenA) {     // แบ่งเป็นปัญหาที่เล็กลง (divide &amp; conquer)     firstBlock = numArray[pos]     maxRest =         findMax(numArray,pos+1,lenA)      // นำคำตอบมารวมกัน (combine)     if (firstBlock &gt; maxRest)         return firstBlock;     else         return maxRest; }</pre>	<div style="text-align: center;">  </div> <pre>int findMax(numArray,pos,lenA) {     if (pos == lenA - 1)         return numArray[pos];      // แบ่งเป็นปัญหาที่เล็กลง (divide &amp; conquer)     firstBlock = numArray[pos]     maxRest =         findMax(numArray,pos+1,lenA)      // นำคำตอบมารวมกัน (combine)     if (firstBlock &gt; maxRest)         return firstBlock;     else         return maxRest; }</pre>
---	--

## Example 2: Factorial

พิจารณา 5! และ 4!

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

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

Define n! แบบ recursive

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

Base case? หยุดที่ 1

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

```
int factorial(int n) {
    //base case

    //divide & conquer

    //combine
}
```

## General Structure

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

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

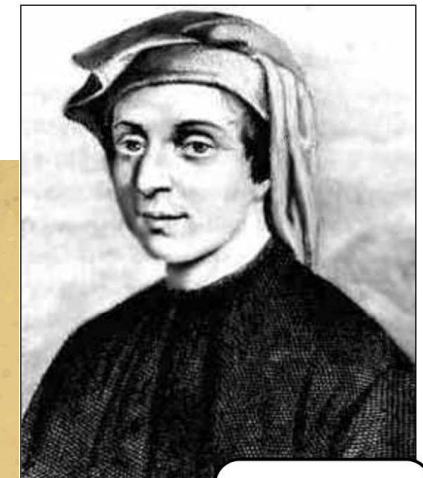
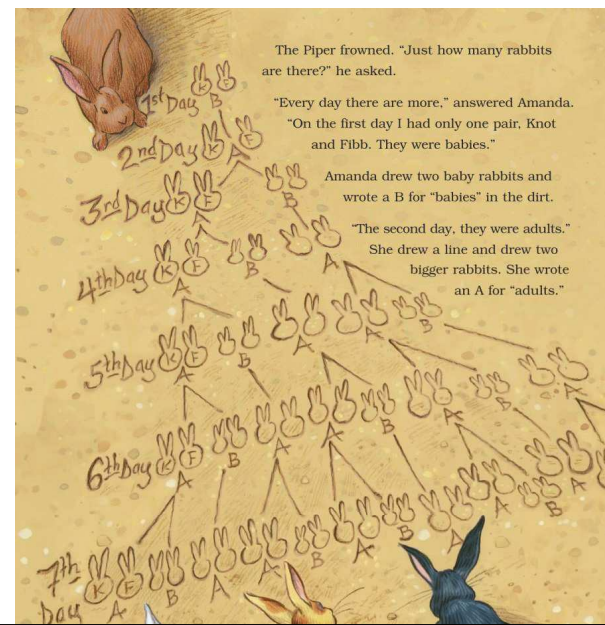
    // combine (นำคำตอบมารวมกัน)
    answer = combine(myWorkLoad,answerFromSubproblem);

    return answer
}
```

ต้องแบ่งแล้วปัญหาเล็กลง  
หรือซับซ้อนน้อยลง และ  
วิ่งเข้าสู่ base case

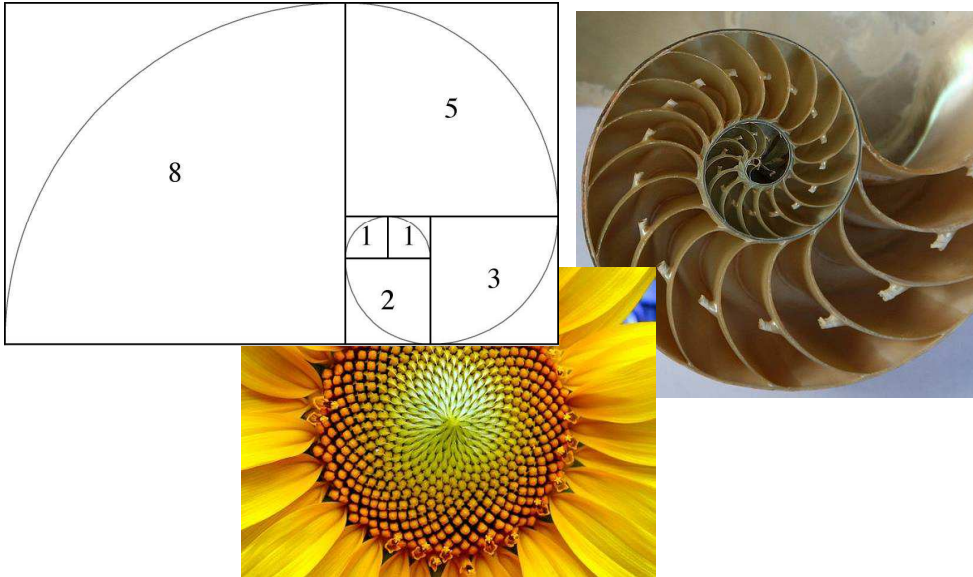
Adapted From:  
<http://ocw.mit.edu/courses/civil-and-environmental-engineering/1-00-introduction-to-computers-and-engineering-problem-solving-spring-2012>

## Fibonacci... and his rabbits



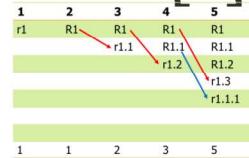
OK, OK...  
Let's talk  
rabbits...

## Example 3: Fibonacci Sequence



17

## Example 3: Fibonacci Sequence [3]



- จำนวนกระต่ายในเดือนนี้  
= จำนวนกระต่ายเดือนที่แล้ว + จำนวนกระต่ายเกิดใหม่เดือนนี้  
= จำนวนกระต่ายโตเต็มวัยเดือนที่แล้ว  
= จำนวนกระต่ายสองเดือนที่แล้ว
- จำนวนกระต่ายในเดือนนี้  
= จำนวนกระต่ายเดือนที่แล้ว + จำนวนกระต่ายสองเดือนที่แล้ว

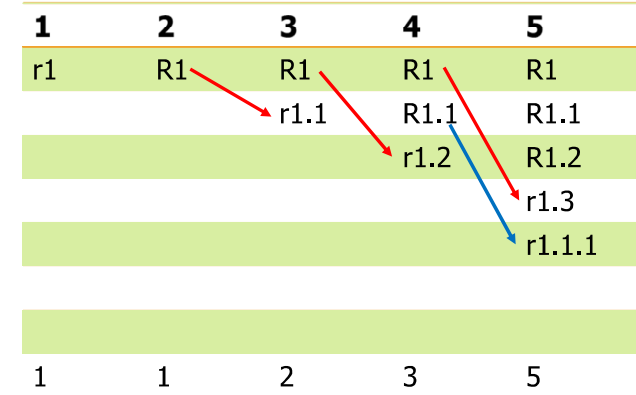
$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

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

19

## Example 3: Fibonacci Sequence [2]

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



18

## Example 3: Fibonacci Sequence [4]

- Divide:
  - Number of Rabbit of month (n)
    - $\text{Fib}(n) = R + r$
- Conquer
  - $R = \text{Fib}(n-1)$
  - $r = \text{Fib}(n-2)$
- Combine
  - $\text{total} = R + r$
- Base Case
  - $\text{Fib}(1) = 1$
  - $\text{Fib}(2) = 1$

```
function fib(x) {
    if (x == 1)
        return 1;

    if (x == 2)
        return 1;

    int R = fib(x - 1);
    int r = fib(x - 2);

    int total = R + r;
    return total;
}
```

20



# The Three Laws of Recursion

1. A recursive algorithm must have a **base case**.
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>

## Example 4: Prime Factor [2]

```
function primeFactor(x,num)
```

```
    //base case
```

```
    if (num == 1)
        return
```

```
    //d & c
```

```
    if (num % x == 0) then
        print(x)
        primeFactor(x,num/x)
```

```
    else
```

```
        primeFactor(x+1,num)
```

```
    endif
```

Print แล้ว ค่อย call

## Example 4: Prime Factor

ให้เขียนโปรแกรมเพื่อแสดงค่าตัวประกอบเฉพาะของ integer  $x$  โดยใช้ Recursion ทั้งนี้ให้อ่าน input จาก command redirection (บรรทัดแรกคือจำนวน test case) input1.txt

```
3
360
17
1
```

output1.txt

```
2 2 2 3 3 5
17
1
```

## Tail vs Head Recursions

- เราเรียก recursion ที่มี recursive call อยู่ส่วนหลังของ function ว่า tail recursion
  - Tail recursion มีลักษณะคล้าย loop
    - ทำงานส่วนของตัวเองก่อน แล้วส่งให้เพื่อนทำ
      - myworkLoad ก่อน แล้วค่อย call recurse(smallerArguments)
    - ไม่จำเป็นต้องรอผล return จากเพื่อนค่อยตัดสินใจ
      - ไม่ต้องรอ answerFromSubproblem เพื่อมา combine

## Tail vs Head Recursions [2]

- ถ้า recursion อยู่ส่วนต้นของ function เราเรียก recursion แบบนี้ว่า head recursion
- แบ่งงานให้เพื่อนก่อน แล้วต้องรอผลเพื่อมา combine
  - ต้องนำผล จาก recurse(smallerArguments) มา combine ถึง return ได้

25

## Recursion Helper Functions

- ในบางกรณี เราจำเป็นต้อง ส่งต่อ parameter บางตัวเพื่ออำนวยความสะดวกในการทำ recursion ที่ user ไม่จำเป็นต้องทราบ หรือ input เข้ามา
- เช่น กรณี array หากต้องการ recursive call ณ ช่วง index ที่ย่อยลงไป เนื่องจากเป็นผลของการแบ่งปัญหาเป็น subproblem

USER: processArray(int a[], int lengthA)

HELPER: processArray(int a[], int start, int end, int lengthA)

27

## Tail vs Head Recursions [3]

Tail Recursion	Head Recursion
<pre>void tail(n): {     if (n == 1)         return;     else         print(n);      tail(n-1) }</pre>	<pre>void head(n): {     if (n == 1)         return;     else         head(n-1);      print(n) }</pre>

ผลลัพธ์ของการ traverse array โดยใช้วิธี head vs tail?

26

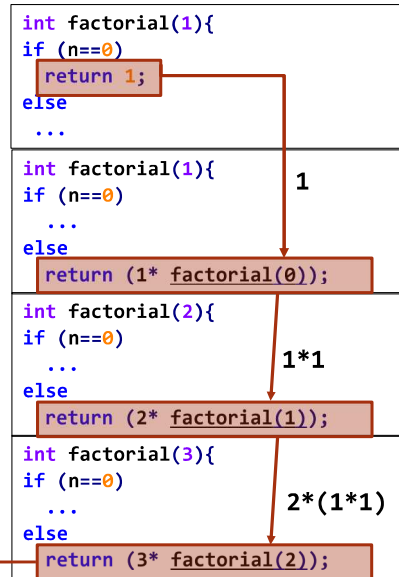
## Mathematical Induction

- 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.

28

# Recursive Memory Stack

```
int factorial(int n){
if (n==0)
    return 1;
else
    return (n* factorial(n-1));
}
```



factorial(3) = 3\*(2\*(1\*1))

29

# Fibonacci Revisited

```
function fib(n)
    if (n < 2)
        # Base case: fib(0) and fib(1) are both 1
        return 1
    else
        # Recursive case: fib(n) = fib(n-1) + fib(n-2)
        return fib(n-1) + fib(n-2)
    endif
endfor
```

```
for n ← 1 to 15
    print(fib(n))
endfor
```

1 1 2 3 5 8 13 21 34 55 89 144 233 377 610

30

# Fibonacci Revisited [2]

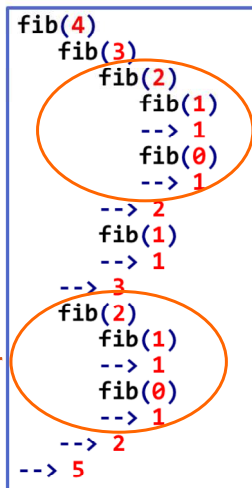
```
int fib(int n, int depth)
{
    int i, result;
    for (i = 0; i < depth; i++)
        printf(" ");
    printf("fib(%d)\n", n);

    if (n < 2)
        result = 1;
    else
        result = fib(n - 1, depth + 1) +
                  fib(n - 2, depth + 1);

    for (i = 0; i < depth; i++)
        printf(" ");
    printf("--> %d\n", result);
    return result;
}
```

ปัญหาบางลักษณะ  
ไม่เหมาะ  
กับวิธีแก้ปัญห  
แบบ recursion  
(สังเกตการคำนวณ  
ซ้ำ)

```
int main()
{
    int depth = 0;
    fib(4, depth);
    return 0;
}
```



31

# Iteration vs Recursion Example

Iterative

```
function factorial(n)
    factorial = 1
    for i ← 2 to n+1
        factorial *= i
    endfor

    return factorial

print(factorial(5))
```

Recursive

```
function factorial(n)
    if (n < 2)
        return 1
    else
        return n*factorial(n-1)
    endif

    print factorial(5)
```

32



## Iteration vs Recursion Example [2]

Iterative

```
function gcd(x,y)
  while (y > 0)
    r = x % y
    x = y
    y = r
  endwhile

  return x

print(gcd(1024, 360))
```

Recursive

```
function gcd(x,y)
  if (y == 0)
    return x
  else
    return gcd(y,x % y)
  endif

  print(gcd(1024, 360))
```

33

## Iteration vs Recursion Summary [2]

- **Recursion**
  - แบ่งงานเป็นส่วนย่อย ๆ เพื่อแก้ปัญหา
  - ใกล้เคียงกับการนิยามทางคณิตศาสตร์
  - วิธีเขียนสั้นและสวยงามกว่า ("more elegant")
- **Iteration**
  - เป็นการทำงานเดิมซ้ำ ๆ
  - สเต็ปการทำงานเหมือนการเขียนโปรแกรม
  - (อาจจะ) เข้าใจได้ง่ายกว่า

35

## Iteration vs Recursion Summary

- **Recursion** สามารถใช้กับปัญหาใด ๆ ที่สามารถเขียนเป็นรูปแบบการแก้ปัญหาในหน่วยย่อยลงได้
- การแก้ปัญหาโดยใช้ **Iteration** สามารถเขียนให้อยู่ในรูป **Recursion** ได้
- ในการเลือกใช้งานระหว่าง **Iterative** หรือ **Recursive** นั้นไม่มีหลักตายตัว
- เลือกตามความเหมาะสมของปัญหา
- **Recursive** ไม่จำเป็นต้องเป็นทางเลือกที่ดีกว่าเสมอ

34

## Iteration vs Recursion Summary [3]

	Recursion	Iteration
Elegance	++	--
Performance	--	++
Debugability	--	++

- **Conclusion (for now):**  
Use iteration when practicable. Use recursion when required (for "naturally recursive problems").

36

## When to use Recursion

- **หลีกเลี่ยง**การใช้ recursive function เมื่อมีการใช้ local arrays ขนาดใหญ่ (Recursive ใช้ Memory เยอะ)
- เลือกใช้ recursion เมื่อทำให้ลดเวลาการเขียนโปรแกรม หรือ ทำให้โปรแกรมลดความซับซ้อนลงมาก ๆ
- Recursion เหมาะสมกับงานประเภทที่ใช้ Divide-and-conquer algorithm ในการแก้ปัญหา เช่น merge sort และ binary search

## Reference

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