

CS1010

<http://www.comp.nus.edu.sg/~cs1010/>

Programming Methodology

UNIT 17

Recursion



NUS
National University
of Singapore

School of
Computing

Unit 17: Recursion

Objectives:

- Understand the nature of recursion
- Learn to write recursive functions
- Comparing recursive codes with iterative codes

Reference:

- Chapter 9: Recursion

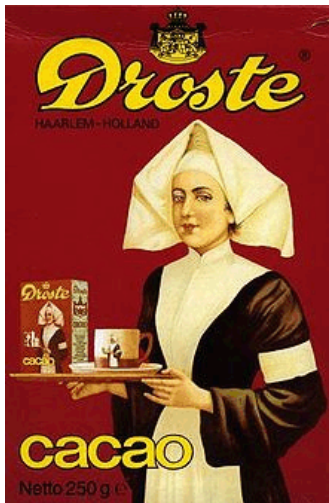
Unit 17: Recursion

1. Introduction
2. Two Simple Classic Examples
 - 2.1 Demo #1: Factorial
 - 2.2 Demo #2: Fibonacci
3. Gist of Recursion
4. Exercises
5. Thinking Recursively
 - 5.1 Think: Sum of Squares
 - 5.2 Demo #3: Counting Occurrences
6. Auxiliary Function
7. Exercises
8. Types of Recursion
9. Tracing Recursive Codes
10. Recursion versus Iteration
11. Towers of Hanoi (in separate file)

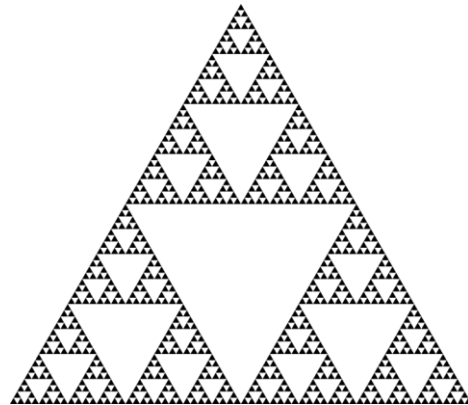
1. Introduction (1/3)

RECURSION A central idea in CS

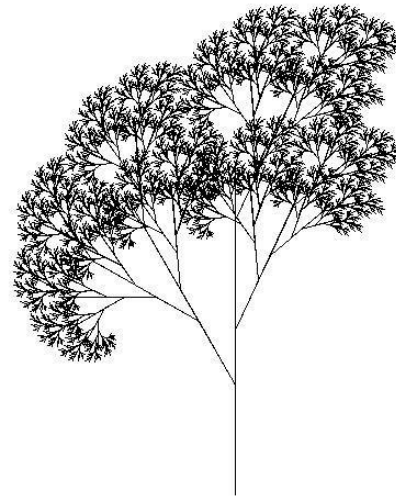
Some examples of recursion (inside and outside CS):



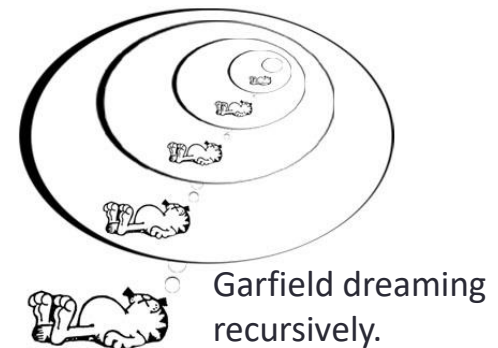
Droste effect



Sierpinski triangle



Recursive tree



1. Introduction (2/3)

RECURSION

A central idea in CS.

Definitions based on recursion:

Recursive definitions:

1. A person is a **descendant** of another if
 - the former is the latter's child, or
 - the former is one of the **descendants** of the latter's child.
2. A **list of numbers** is
 - a number, or
 - a number followed by a **list of numbers**.

Dictionary entry:

Recursion: See recursion.

Recursive acronyms:

1. GNU = GNU's Not Unix
2. PHP = PHP: Hypertext Preprocessor

**To understand
recursion, you must
first understand
recursion.**

1. Introduction (3/3)

- There is NO new syntax needed for recursion.
- **Recursion** is a form of (algorithm) design; it is a problem-solving technique for divide-and-conquer paradigm
 - Very important paradigm – many CS problems solved using it
- Recursion is:

**A method where
the solution to a problem
depends on
solutions to smaller instances
of the SAME problem.**

2. Two Simple Classic Examples

- From these two examples, you will see how a **recursive algorithm** works

Winding phase

Invoking/calling 'itself' to solve smaller or simpler instance(s) of a problem ...

... and then building up the answer(s) of the simpler instance(s).

Unwinding phase

2.1 Demo #1: Factorial (1/3)

$$n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$$

Iterative code (version 1):

```
// Pre-cond: n >= 0
int factorial_iter1(int n) {
    int ans = 1, i;
    for (i=2; i<=n; i++) {
        ans *= i;
    }
    return ans;
}
```

Iterative code (version 2):

```
// Pre-cond: n >= 0
int factorial_iter2(int n) {
    int ans = 1;
    while (n > 1) {
        ans *= n;
        n--;
    }
    return ans;
}
```

Unit17_Factorial.c

2.1 Demo #1: Factorial (2/3)

$$n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$$

Doing it the recursive way?

Recurrence relation:

$$n! = n \times (n-1)!$$

$$0! = 1$$

```
// Pre-cond: n >= 0
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n-1);
}
```

No loop!
But calling itself
(recursively) brings
out repetition.

Note: All the three versions work only for $n < 13$, due to the range of values permissible for type int. This is the limitation of the data type, not a limitation of the problem-solving model.

2.1 Demo #1: Factorial (3/3)

- Trace factorial(3). For simplicity, we write $f(3)$.

Winding:

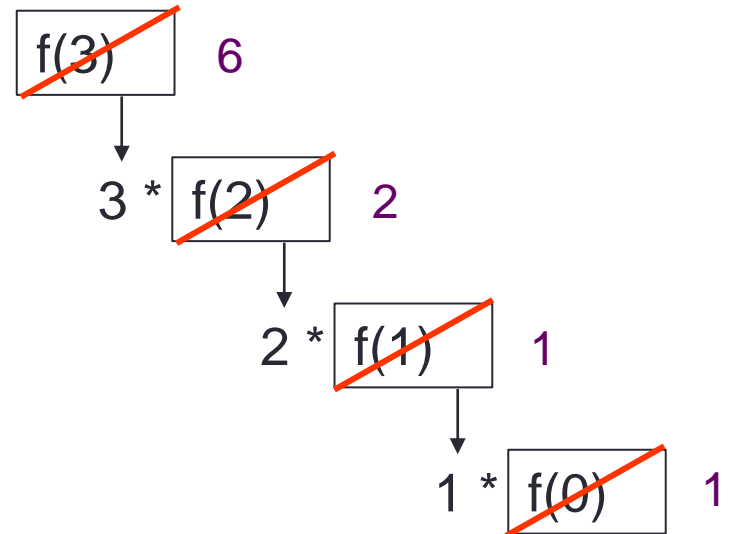
$f(3)$: Since $3 \neq 0$, call $3 * f(2)$
 $f(2)$: Since $2 \neq 0$, call $2 * f(1)$
 $f(1)$: Since $1 \neq 0$, call $1 * f(0)$
 $f(0)$: Since $0 == 0$, ...

Unwinding:

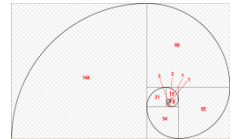
$f(0)$: Return 1
 $f(1)$: Return $1 * f(0) = 1 * 1 = 1$
 $f(2)$: Return $2 * f(1) = 2 * 1 = 2$
 $f(3)$: Return $3 * f(2) = 3 * 2 = 6$

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

Trace tree:



2.2 Demo #2: Fibonacci (1/4)



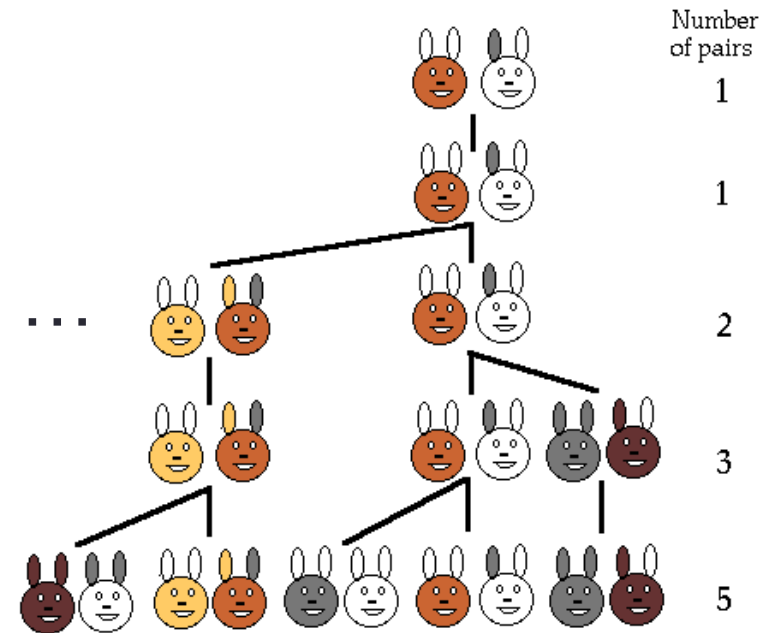
- The **Fibonacci series** models the rabbit population each time they mate:

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

- The modern version is:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

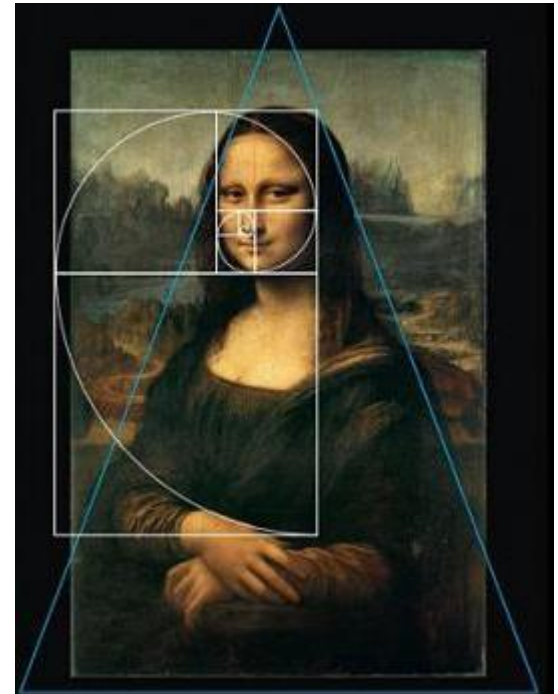
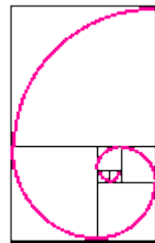
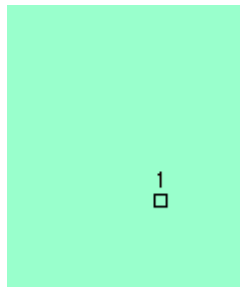
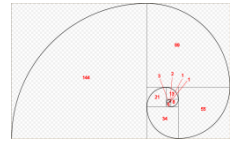
- Fibonacci numbers are found in nature (sea-shells, sunflowers, etc)



- <http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html>

2.2 Demo #2: Fibonacci (2/4)

- Fibonacci numbers are found in nature (sea-shells, sunflowers, etc)
- <http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibnat.html>



2.2 Demo #2: Fibonacci (3/4)

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Unit17_Fibonacci.c

Iterative code:

```
// Pre-cond: n >= 0
int fib_iter(int n) {
    int prev1 = 1,
        prev2 = 0, sum;

    if (n < 2)
        return n;
    for (; n>1; n--) {
        sum = prev1 + prev2;
        prev2 = prev1;
        prev1 = sum;
    }
    return sum;
}
```

Recursive code:

```
// Pre-cond: n >= 0
int fib(int n) {
    if (n < 2)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

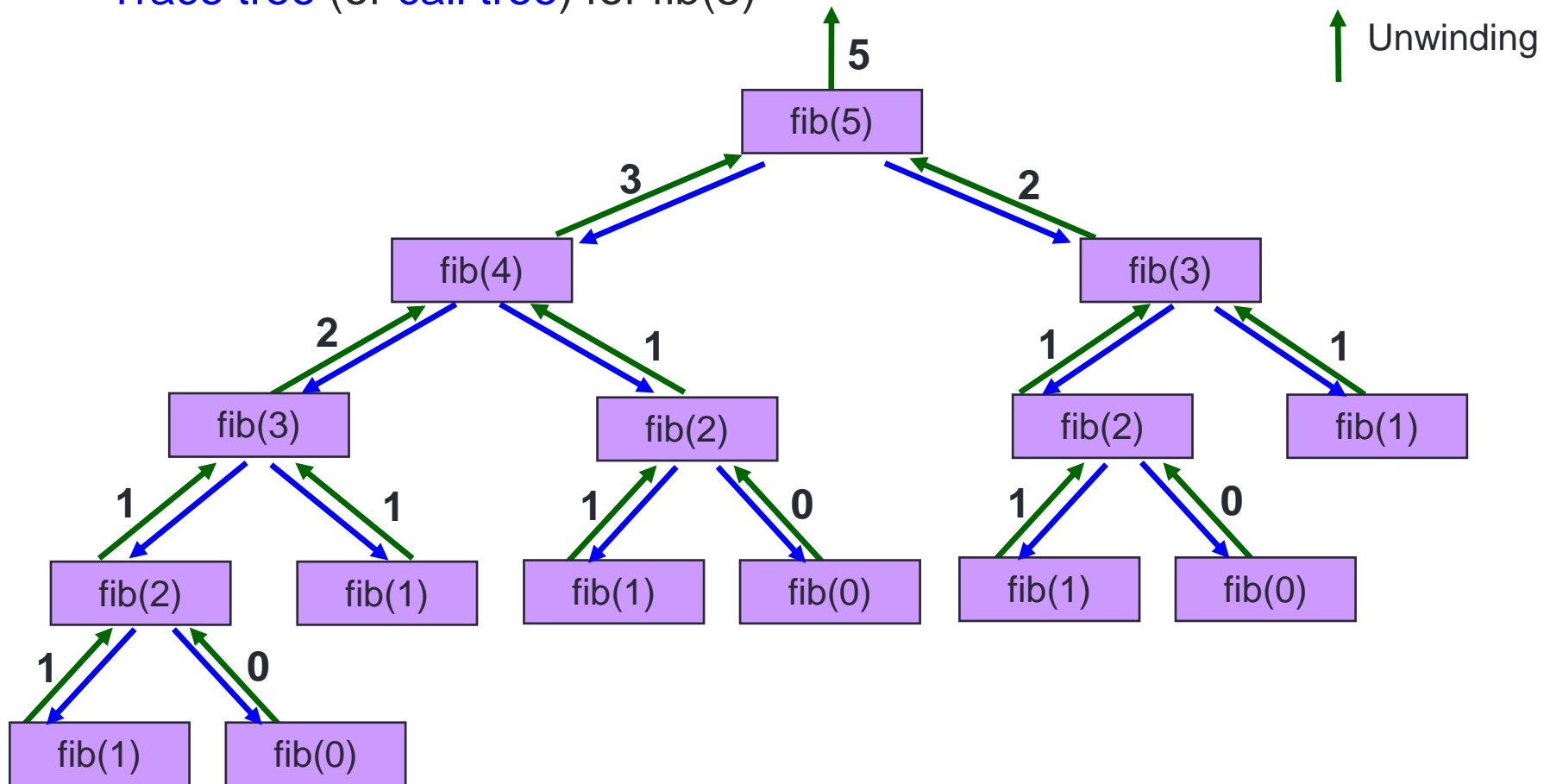
Recurrence relation:

$$\begin{aligned} f_n &= f_{n-1} + f_{n-2} \quad n \geq 2 \\ f_0 &= 0 \\ f_1 &= 1 \end{aligned}$$

```
int fib(int n) {  
    if (n < 2)  
        return n;  
    else  
        return fib(n-1) + fib(n-2);  
}
```

2.2 Fibonacci (4/4)

- fib(n) makes 2 recursive calls: fib(n-1) and fib(n-2)
- Trace tree (or call tree) for fib(5)



3. Gist of Recursion (1/6)

Iteration vs Recursion: How to compute factorial(3)?

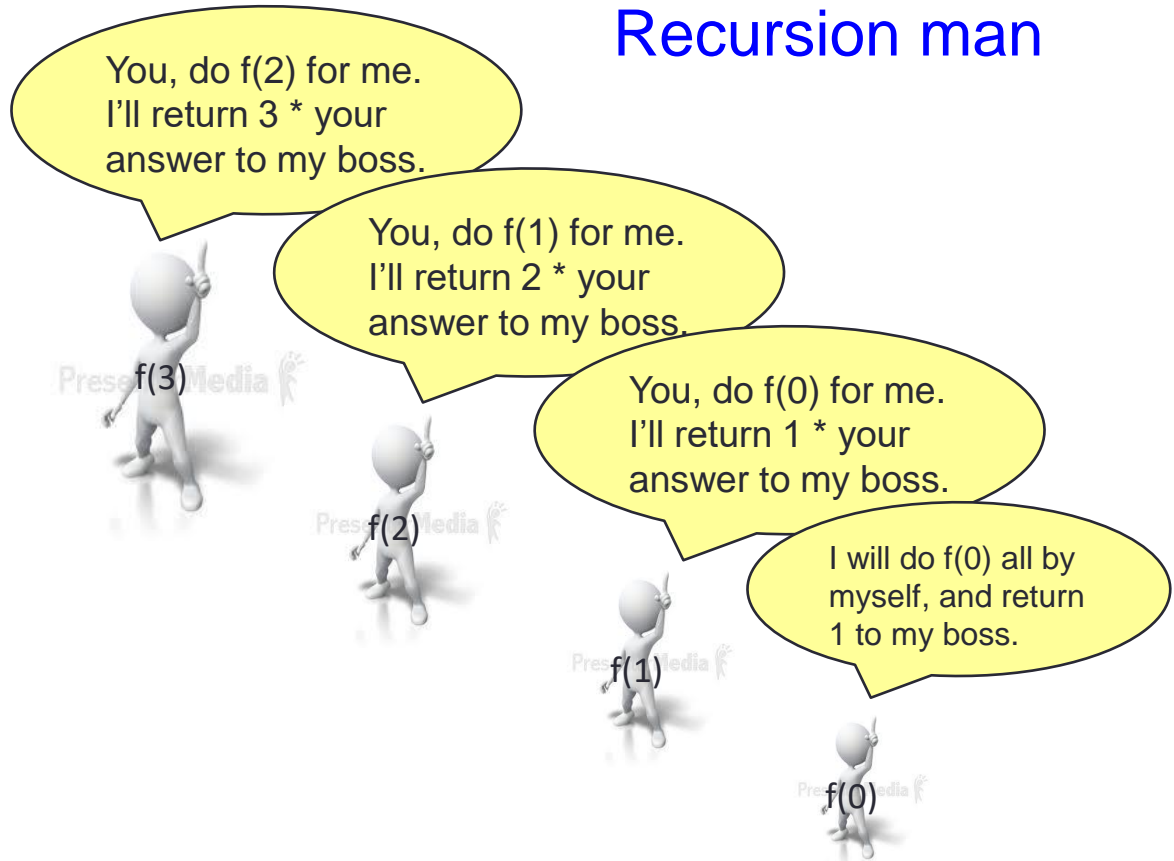


Iteration man

I do $f(3)$ all by myself...return 6 to my boss.



Recursion man



3. Gist of Recursion (2/6)

- Problems that lend themselves to a recursive solution have the following characteristics:
 - One or more **simple cases** (also called **base cases** or **anchor cases**) of the problem have a straightforward, non-recursive solution
 - The other cases can be redefined in terms of problems that are smaller, i.e. closer to the simple cases
 - By applying this redefinition process every time the recursive function is called, eventually the problem is reduced entirely to simple cases, which are relatively easy to solve
 - The solutions of the smaller problems are combined to obtain the solution of the original problem

3. Gist of Recursion (3/6)

- To write a recursive function:
 - Identify the **base case(s)** of the relation
 - Identify the **recurrence relation**

```
// Pre-cond: n >= 0
int factorial(int n) {
    if (n == 0)
        return 1;

    else
        return n * factorial(n-1);
}
```

```
// Pre-cond: n >= 0
int fib(int n) {
    if (n < 2)
        return n;

    else
        return fib(n-1) + fib(n-2);
}
```

3. Gist of Recursion (4/6)

- Always check for base case(s) first
 - What if you omit base case(s)?
- Do not write redundant base cases

```
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else if (n == 1)  
        return 1;  
    else if (n == 2)  
        return 2;  
    else if (n == 3)  
        return 6;  
    else  
        return n * factorial(n-1);  
}
```

redundant

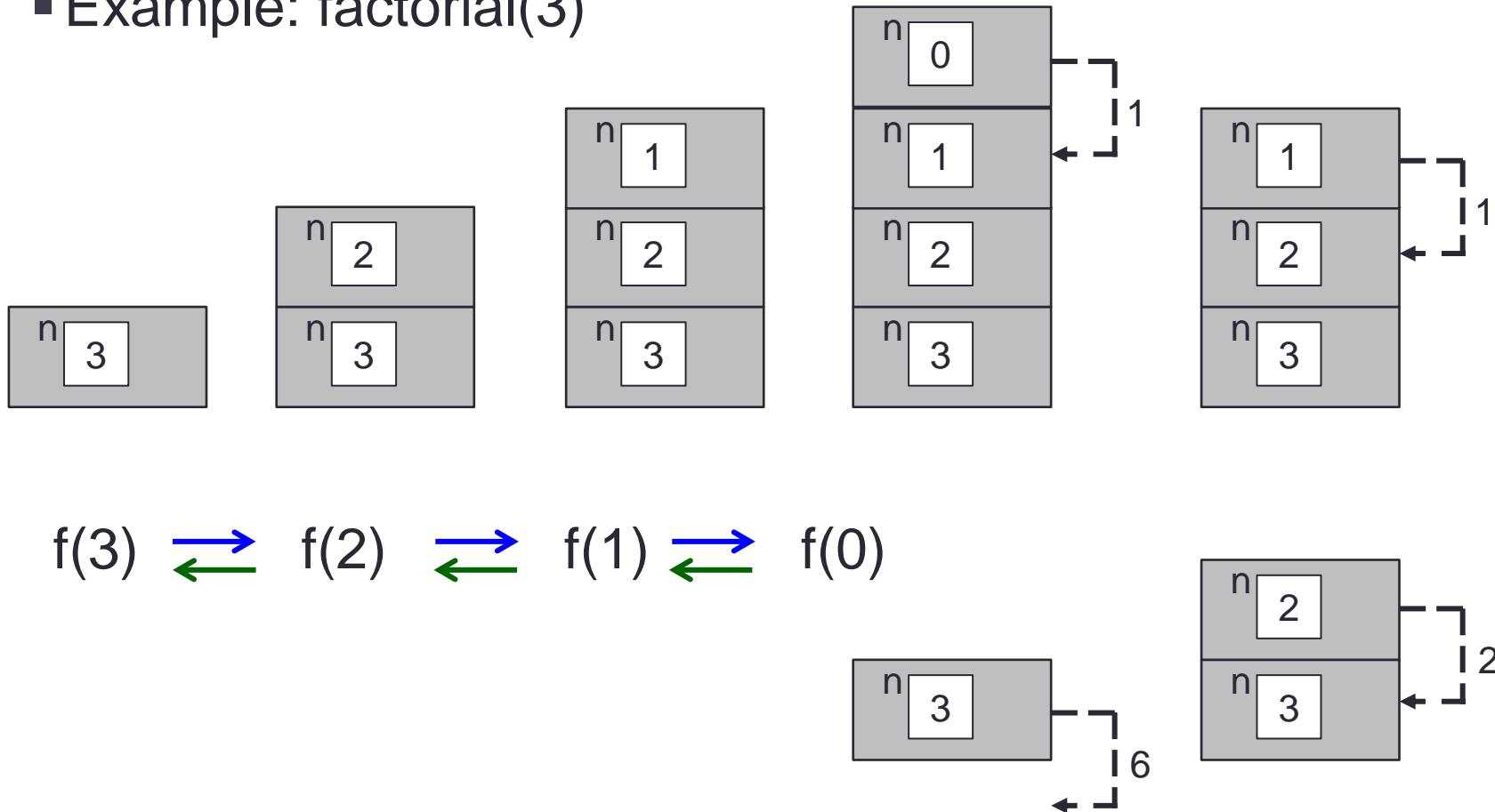
3. Gist of Recursion (5/6)

- When a function is called, an **activation record** (or frame) is created by the system.
- Each activation record stores the local parameters and variables of the function and its return address.
- Such records reside in the memory called **stack**.
 - Stack is also known as **LIFO** (last-in-first-out) structure
- A recursive function can potentially create many activation records
 - **Winding**: each recursive call creates a separate record
 - **Unwinding**: each return to the caller erases its associated record

3. Gist of Recursion (6/6)

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

■ Example: factorial(3)



4. Exercise #1: Greatest Common Divisor

- The recurrence relation for Greatest Common Divisor (GCD) of two non-negative integers a and b , not both zero, is given below:

$$\text{GCD}(a, b) = \begin{cases} a, & b = 0 \\ \text{GCD}(b, a \% b), & \text{otherwise} \end{cases}$$

- Write a function `int gcd(int a, int b)` to compute the GCD of a and b . Skeleton program `Unit17_GCD.c` is given.

4. Exercise #2: Tracing

- Given the following 2 recursive functions, trace `mystery1(3902)` and `mystery2(3902)` using the trace tree method.

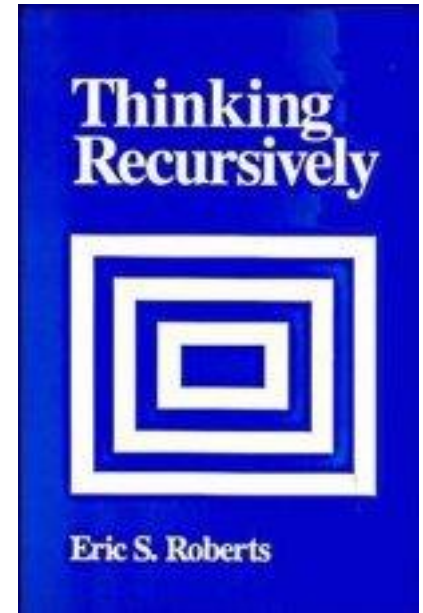
```
void mystery1(int n) {  
    if (n>0) {  
        printf("%d", n%10);  
        mystery1(n/10);  
    }  
}
```

```
void mystery2(int n) {  
    if (n>0) {  
        mystery2(n/10);  
        printf("%d", n%10);  
    }  
}
```

The order of statements does matter!

5. Thinking Recursively

- It is apparent that to do recursion you need to think “recursively”:
 - Breaking a problem into simpler problems that have identical form
- Is there only one way of breaking a problem into simpler problems?



5.1 Think: Sum of Squares (1/5)

- Given 2 positive integers x and y , where $x \leq y$, compute

$$\text{sumSq}(x,y) = x^2 + (x+1)^2 + \dots + (y-1)^2 + y^2$$

- For example

$$\text{sumSq}(5,10) = 5^2 + 6^2 + 7^2 + 8^2 + 9^2 + 10^2 = 355$$

- How do you break this problem into smaller problems?
- How many ways can it be done?
- We are going to show 3 versions
- See [Unit17_SumSquares.c](#)



5.1 Think: Sum of Squares (2/5)

- Version 1: 'going up'

```
int sumSq1(int x, int y) {  
    if (x == y) return x * x;  
    else return x * x + sumSq1(x+1, y);  
}
```

- Version 2: 'going down'

```
int sumSq2(int x, int y) {  
    if (x == y) return y * y;  
    else return y * y + sumSq2(x, y-1);  
}
```

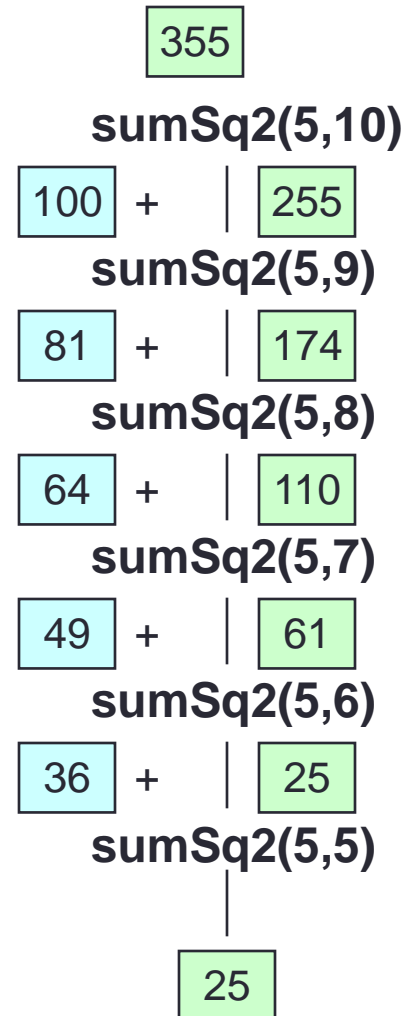
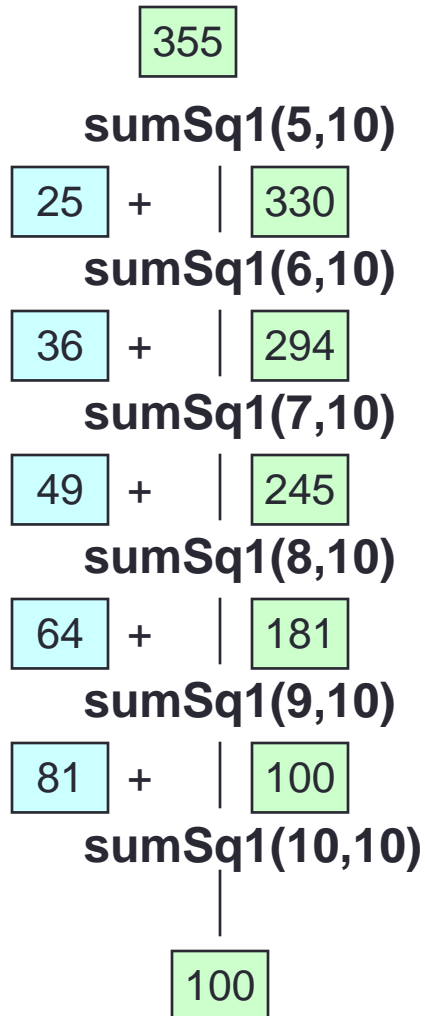
5.1 Think: Sum of Squares (3/5)

- Version 3: 'combining two half-solutions'

```
int sumSq3(int x, int y) {  
    int mid; // middle value  
  
    if (x == y)  
        return x * x;  
    else {  
        mid = (x + y) / 2;  
        return sumSq3(x, mid) + sumSq3(mid+1, y);  
    }  
}
```

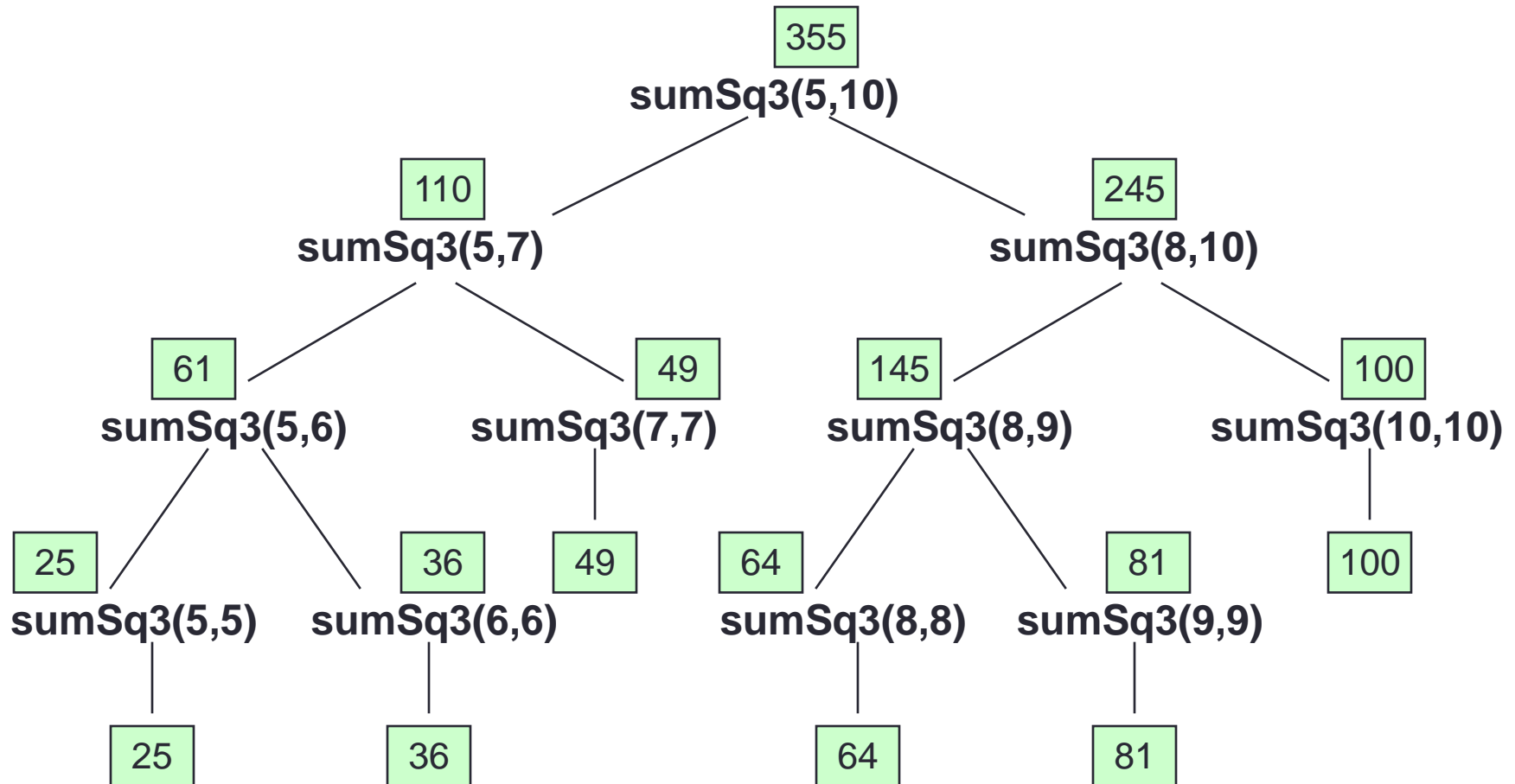
5.1 Think: Sum of Squares (4/5)

- Trace trees



5.1 Think: Sum of Squares (5/5)

- Trace tree



5.2 Demo #3: Counting Occurrences (1/4)

- Given an array

```
int list[ ] = { 9, -2, 1, 7, 3, 9, -5, 7, 2, 1, 7, -2, 0, 8, -3 }
```

- We want

```
countValue(7, list, 15)
```

to return 3 (the number of times 7 appears in the 15 elements of list).

5.2 Demo #3: Counting Occurrences (2/4)

Iterative code:

Unit17_CountValue.c

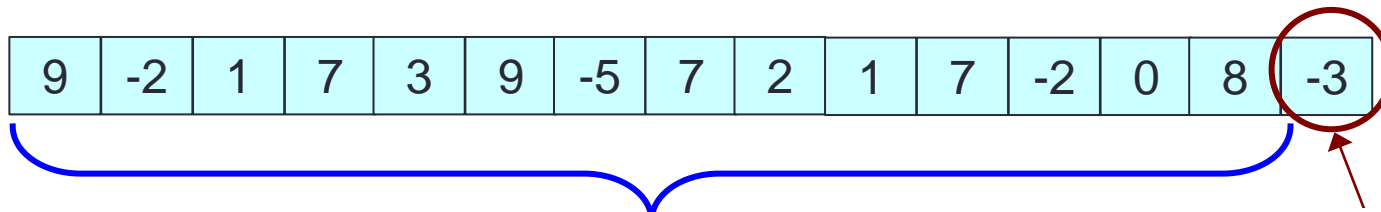
```
int countValue_iter(int value, int arr[], int size)
{
    int count = 0, i;

    for (i=0; i<size; i++)
        if (value == arr[i])
            count++;

    return count;
}
```

5.2 Demo #3: Counting Occurrences (3/4)

- To get `countValue(7, list, 15)` to return 3.
- Recursive thinking goes...



*... and get someone to
count the 7 in this smaller
problem, ...*

*If I handle the last
element myself, ...*

*... then, depending on whether the last element
is 7 or not, my answer is either his answer or
his answer plus 1!*

5.2 Demo #3: Counting Occurrences (4/4)

Recursive code:

Unit17_CountValue.c

```
int countValue(int value, int arr[], int size) {  
    if (size == 0)  
        return 0;  
    else  
        return (value == arr[size-1]) +  
               countValue(value, arr, size-1);  
}
```

Note: The second return statement is equivalent to the following (why?):

```
if (value == arr[size-1])  
    return 1 + countValue(value, arr, size-1);  
else  
    return countValue(value, arr, size-1);
```

6. Auxiliary Function (1/3)

- Sometimes, **auxiliary functions** are needed to implement recursion. Eg: Refer to Demo #3 Counting Occurrences.
- If the function handles the first element instead of the last, it could be re-written as follows:

```
int countValue(int value, int arr[],
               int start, int size) {
    if (start == size)
        return 0;
    else
        return (value == arr[start]) +
               countValue(value, arr, start+1, size);
}
```

6. Auxiliary Function (2/3)

- However, doing so means that the calling function has to change the call from:

```
countValue(value, list, ARRAY_SIZE)
```

to:

```
countValue(value, list, 0, ARRAY_SIZE)
```

- The additional parameter **0** seems like a redundant data from the caller's point of view.

6. Auxiliary Function (3/3)

- Solution: Keep the calling part as:

```
countValue(value, list, ARRAY_SIZE)
```

- Rename the original `countValue()` function to `countValue_recur()`. The recursive call inside should also be similarly renamed.
- Add a new function `countValue()` to act as a **driver function**, as follows:

```
int countValue(int value, int arr[], int size) {  
    return countValue_recur(value, arr, 0, size);  
}
```

- See program [Unit17_CountValue_Auxiliary.c](#)

7. Exercise #3: Sum Digits

- Write a recursive function `int sum_digits(int n)` that sums up the digits in n , assuming that n is a non-negative integer.
- Skeleton program `Unit17_SumDigits.c` is given.
- This exercise is mounted on CodeCrunch.
- Sample runs:

```
Enter a non-negative integer: 6543
Sum of its digits = 18
```

```
Enter a non-negative integer: 3708329
Sum of its digits = 32
```

7. Exercise #4: Sum Array

- Write a program [Unit17_SumArray.c](#) to read data into an integer array with at most 10 elements, and sum up all values in the array, using a recursive function.
- This exercise is mounted on CodeCrunch.
- Sample runs:

```
Enter number of elements: 6
Enter 6 values: 4 3 -2 0 1 3
Array read: 4 3 -2 0 1 3
Sum = 9
```

```
Enter number of elements: 8
Enter 8 values: 11 25 56 8 12 7 31 16
Array read: 11 25 56 8 12 7 31 16
Sum = 166
```

8. Types of Recursion

- Besides direct recursion (function A calls itself), there could be mutual or indirect recursion (we do not cover these in CS1010)
 - Examples: Function A calls function B, which calls function A; or function X calls function Y, which calls function Z, which calls function X.
- Note that it is not typical to write a recursive `main()` function.
- One type of recursion is known as **tail recursion**.
 - Not covered in CS1010

9. Tracing Recursive Codes

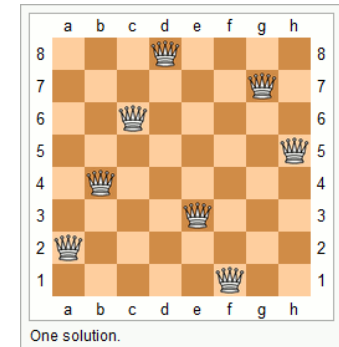
- Beginners usually rely on tracing to understand the sequence of recursive calls and the passing back of results.
- However, tracing a recursive code is tedious, especially for non-tail-recursive codes. The trace tree could be huge (example: fibonacci).
- If you find that tracing is needed to aid your understanding, start tracing with **small** problem sizes, then gradually see the relationship between the successive calls.
- Students should grow out of tracing habit and understand recursion by examining the relationship between the problem and its immediate subproblem(s).

10. Recursion versus Iteration (1/2)

- Iteration can be more efficient
 - Replaces function calls with looping
 - Less memory is used (no activation record for each call)
- Some good compilers are able to transform a tail-recursion code into an iterative code.
- General guideline: If a problem can be done easily with iteration, then do it with iteration.
 - For example, Fibonacci can be coded with iteration or recursion, but the recursive version is very inefficient (large call tree due to duplicate computations), so use iteration instead.

10. Recursion versus Iteration (2/2)

- Many problems are more naturally solved with recursion, which can provide elegant solution.
 - Towers of Hanoi
 - Mergesort (to be covered in CS1020)
 - The N Queens problem
- Conclusion: choice depends on problem and the solution context. In general, use recursion if ...
 - A recursive solution is natural and easy to understand.
 - A recursive solution does not result in excessive duplicate computation.
 - The equivalent iterative solution is too complex.



11. Towers Of Hanoi

- In a separate Powerpoint file.

Summary

- In this unit, you have learned about
 - Recursion as a design strategy
 - The components of a recursive code
 - Differences between Recursion and Iteration

End of File