CS1010E Lecture #12

Simple Recursion

The mirrors

NUS National University of Singapore
Department of Computer Science
School of Computing

Structure Review #1

PREVIOUS LECTURE

- Structure is a user-defined data type.

declaration

```
int a, b = 5;
```

```
typedef struct {
  char name[20];
  char gender;
  int age;
} person_t;

person_t lecturer,
         prof = {"Harry", 'M', 20};
```

assignment

```
a = b;
```

```
lecturer = prof;
```

Structure Review #2

PREVIOUS LECTURE

- Structure is a user-defined data type.

```
void print_int(int a);
```

function parameter

```
void print_struct(person_t prof);
```

```
void print_int(int *p);
```

pointer to parameter

```
void print_struct(person_t *ptr);
```

```
int return_int();
```

return value

```
person_t return_struct();
```

Structure Review #3

PREVIOUS LECTURE

- Dot operator (.): variable.member
  - Variable must be a structure variable
- Arrow operator (->): pointer->member
  - pointer must be a pointer for structure variable

```c
typedef struct {
  char name[20];
  char gender;
  int age;
} person_t;
```

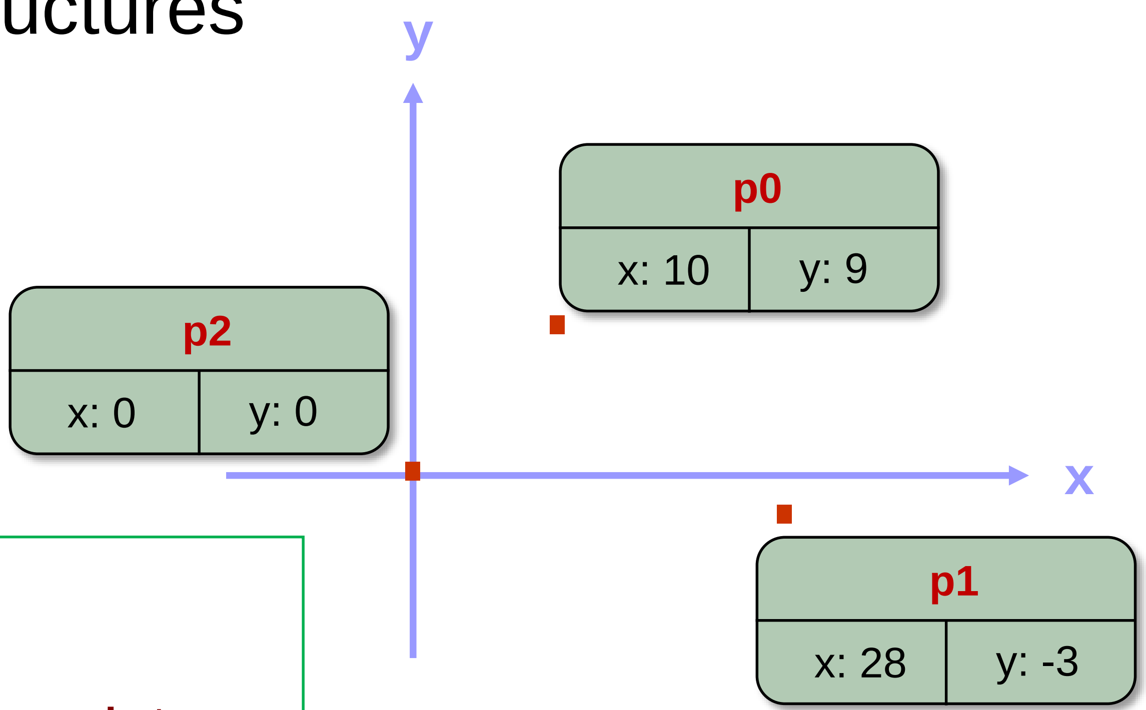
```c
person_t prof = {"Gary", 'M', 50};
person_t *ptr;

printf("name: %s\n", prof.name);

ptr = &prof;
strcpy(ptr->name, "Harry");
ptr->gender = 'M';
(*ptr).age = 20;

printf("name: %s\n", prof.name);
```

Structure Review #4

PREVIOUS LECTURE

- An array of structures

```
typedef struct {
  int x, y;
} point_t;
```



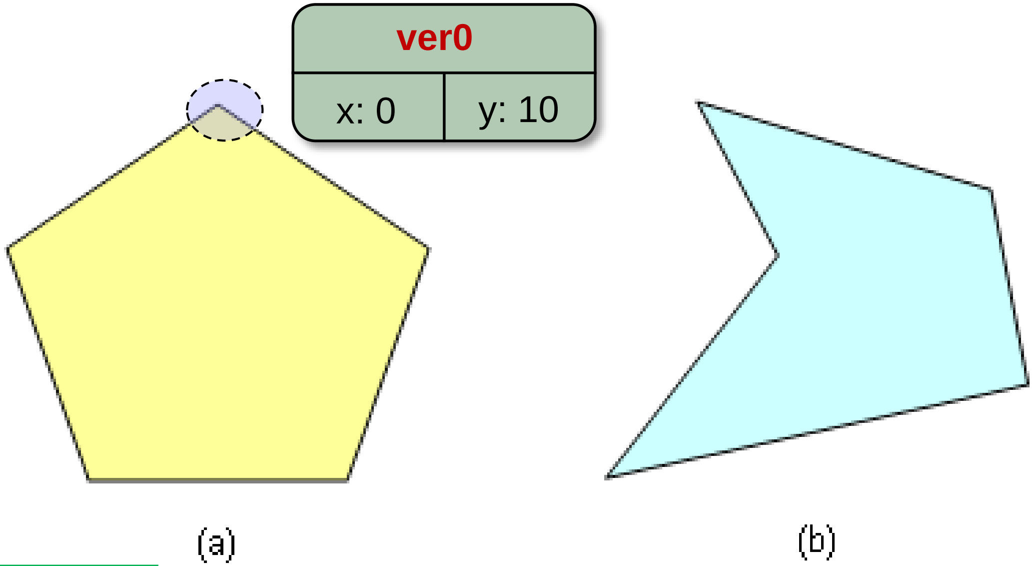

```
int main(void) {

  point_t p[3];
  // initialize three points
  p[0].x = 10; p[0].y = 9;
  p[1].x = 28; p[1].y = -3;
  p[2].x = p[2].y = 0;
}
```

Structure Review #5

PREVIOUS LECTURE

- Nested structures

```
typedef struct {
  int x, y;
} point_t;

typedef struct {
  int num_vertex;
  point_t ver[10];
} polygon_t;
```




```
int main(void) {
  polygon_t poly;
  poly.num_vertex = 5;
  // initialize ver0 of poly
  poly.ver[0].x = 0;
  poly.ver[0].y = 10;
}
```

Quiz

(CS1010 AY2010/11 Semester 1 Exam, Q2(a))

- What is the output of the following program?

```
typedef struct {
  int i, a[4];
} mystruct_t;
```

```
int main(void) {
  mystruct_t s, t;
  s.i = 5;
  s.a[3] = 10;
  t = s;
  printf("%d %d\n", t.i, t.a[3]);
  return 0;
}
```

```
5 10
```

Quiz

(CS1010 AY2013/14 Semester 1 Exam, Q2.2)

- What is the output of the following program?

```
typedef struct {
  char code[10];
  int num_stu;
} module_t;
```

```
void f(module_t m) {
  --m.num_stu;
}
```

```
int main(void) {
  module_t list[] = { {"CS1010", 300}, {"CS1231", 100} };

  f(list[1]);
  printf("%s %d\n", list[1].code, list[1].num_stu);

  return 0;
}
```

```
CS1231 100
```

Quiz

(CS1010E AY2010/11 Semester 2 Midterm Test, Q19)

- **What is printed out by the following C program?**

```
int f(int x) {
  return g(x+1) + g(x+2);
}

int g(int x) {
  return x+3;
}

int main(void) {
  printf("%d\n", f(1)+g(1));
  return 0;
}
```

15

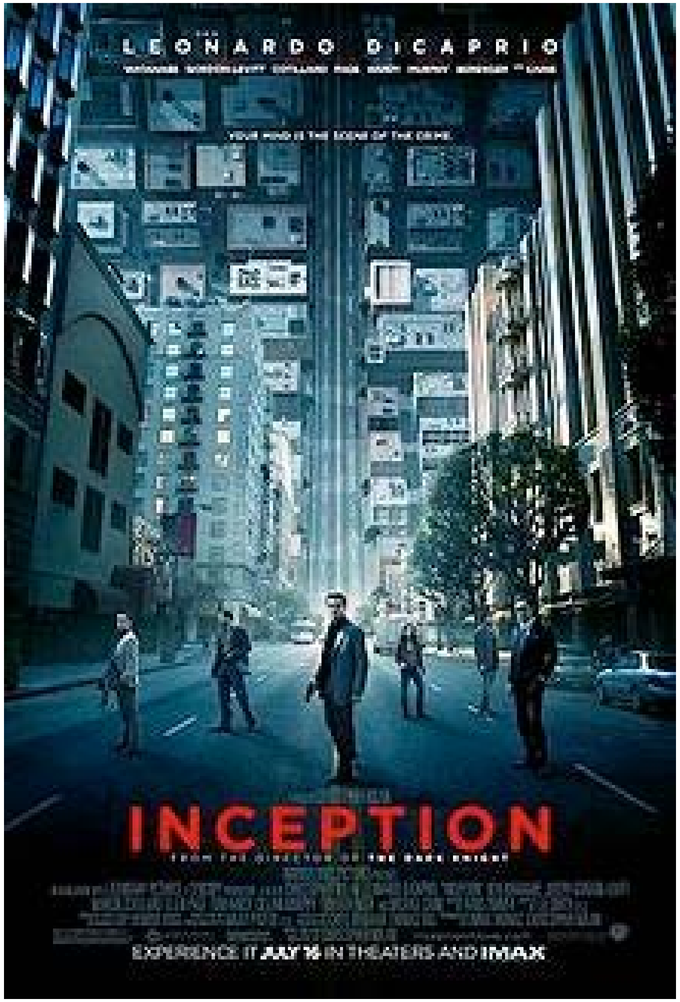
Learning Objectives

- At the end of this lecture, you should understand:
  - The nature of recursion.
  - How to write recursive functions.

Pictorial Example

- This's a picture in a picture.

Filmic Example




```
void dream(int level) {
   printf("entered dream %d\n", level);
   if (level == 3) {
      wakeup(level);
   } else {
      dream(level+1);
      wakeup(level);
   }
}

void wakeup(int level) {
   printf("waking up at %d\n", level);
}
```

```
entered dream level 1
entered dream level 2
entered dream level 3
waking up at level 3
waking up at level 2
waking up at level 1
```

Full story: http://repeatgeek.com/technical/how-to-view-inception-through-code/

Textual Examples

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.

To understand recursion, you must first understand recursion.

Write Recursive Program (1/2)

- 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.
  - A very important paradigm – many CS problems solved using it.
- Recursion is:

A method where
the solution to a problem
depends on
solutions to <u>smaller instances</u>
of the <u>SAME</u> problem.

Write Recursive Program (2/2)

- General idea of recursive algorithms:

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

Demo #1 : Factorial (1/3)

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

Iterative code (Ver. 1)

```
// Pre-condition: n >= 0
int factorial_v1(int n) {
  int i, ans = 1;
  for (i = 1; i <= n; i++) {
    ans *= i;
  }
  return ans;
}
```

Iterative code (Ver. 2)

```
// Pre-condition: n >= 0
int factorial_v2(int n) {
  int ans = 1;
  while (n >= 1) {
    ans *= n;
    n--;
  }
  return ans;
}
```

Demo #1 : Factorial (2/3)

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

recurrence relation:

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

$0! = 1$

Doing it in a recursive way?

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

No loop structure at all! But calling method itself (recursively) brings out repetition.

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

Demo #1 : Factorial (3/3)

- Trace factorial(3)

  ** for simplicity, we write f(3)*

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

Winding:

f(3): Since 3 ≠ 0, call 3 * f(2)

f(2): Since 2 ≠ 0, call 2 * f(1)

f(1): Since 1 ≠ 0, call 1 * f(0)

f(0): Since 0 == 0, …

Unwinding:

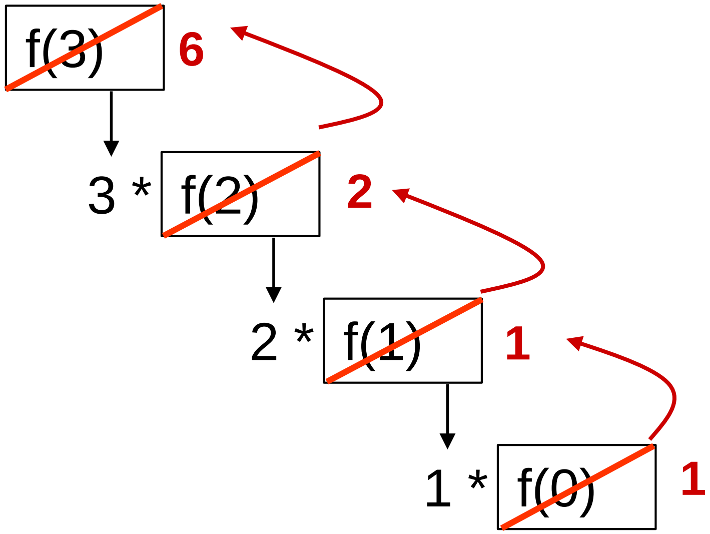
f(0): Return **1**

f(1): Return 1 * f(0) = 1 * 1 = **1**

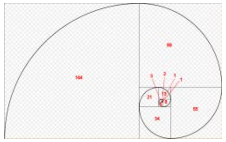
f(2): Return 2 * f(1) = 2 * 1 = **2**

f(3): Return 3 * f(2) = 3 * 2 = **6**

Trace tree:



Demo #2 : Fibonacci (1/3)

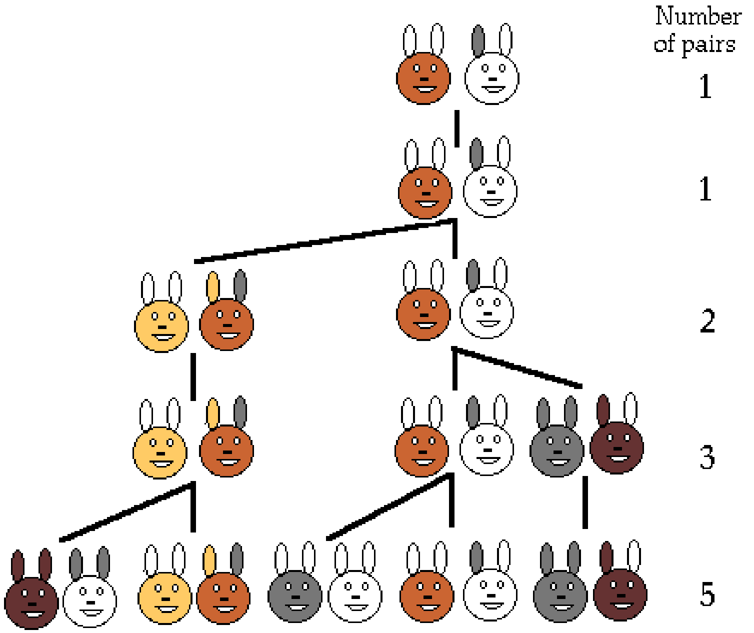


- The Fibonacci series models the rabbit population each time they mate:

  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

Demo #2 : Fibonacci (2/3)

0, 1, 1, 2, 3, 5, 8, 13, 21, …

recurrence relation:

$f_n = f_{n-1} + f_{n-2} \quad n \geq 2$

$f_0 = 0$

$f_1 = 1$

Iterative version:

```
// Pre-condition: n >= 0
int fib_iter(int n) {
  int prev1 = 1, prev2 = 0,
      current, i;

  if (n < 2) {
    return n;
  }

  for (int i = 2; i <= n; i++) {
    current = prev1 + prev2;
    prev2 = prev1;
    prev1 = current;
  }
  return current;
}
```

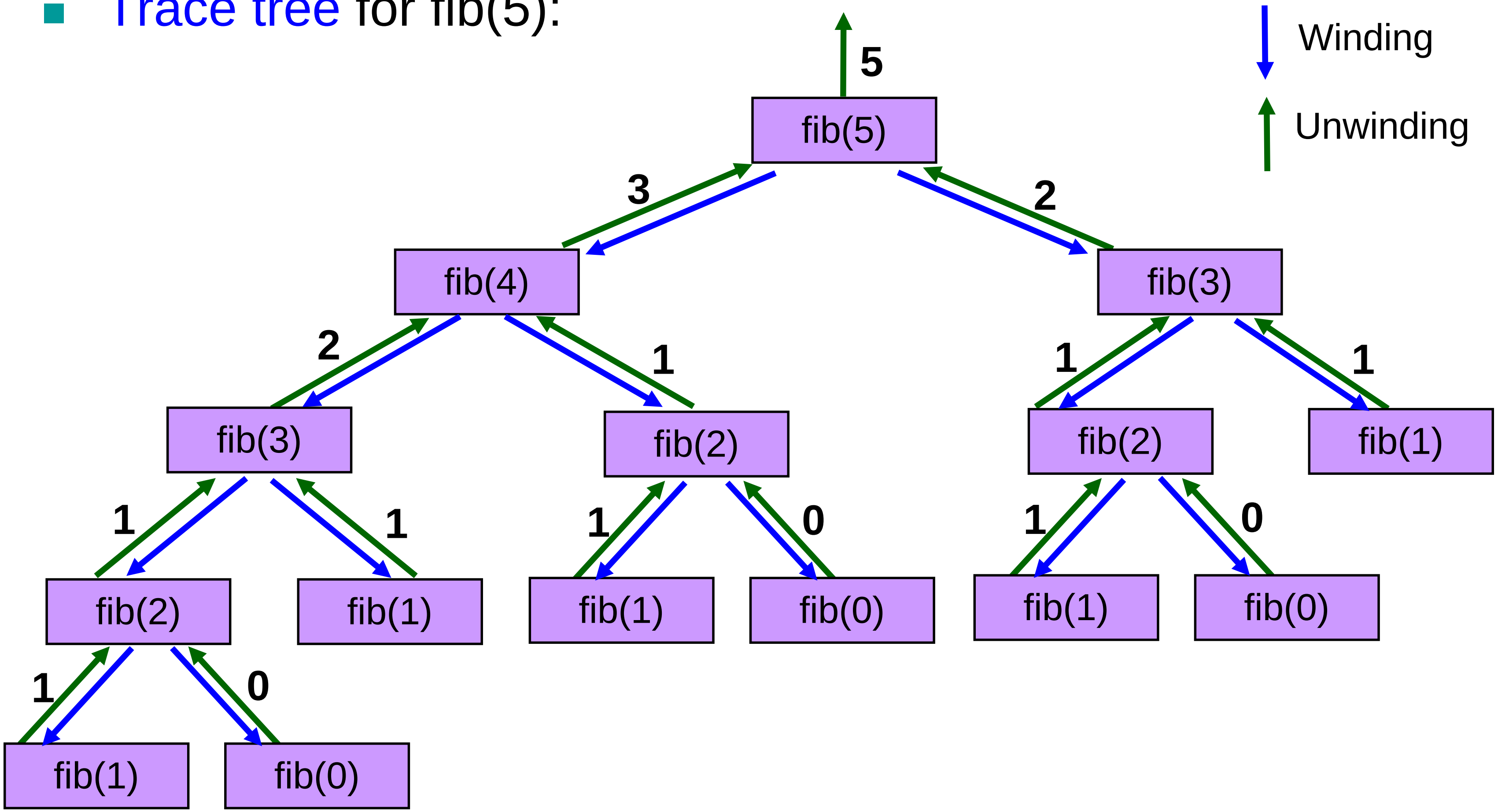
Recursive version:

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

Demo #2 : Fibonacci (3/3)

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

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



Exercises #1 : Tracing

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

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

2093

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

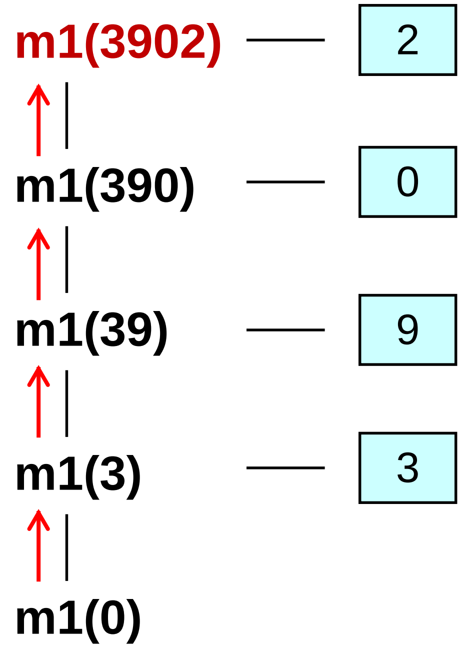
3902

The order of statements does matter!

Exercises #1 : Tracing

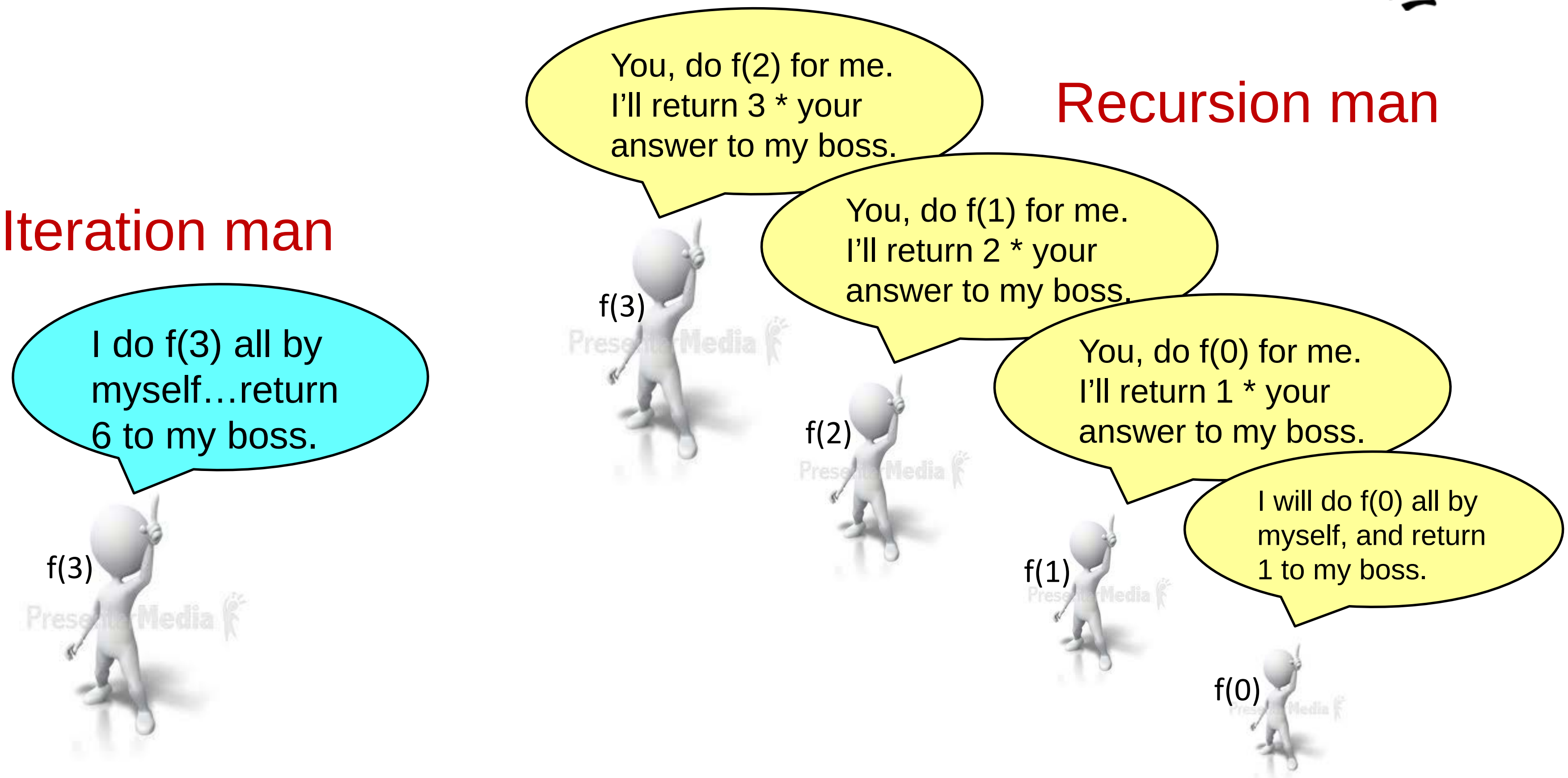
- Trace tree of mystery1(3902)

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



Gist of Recursion (1/4)

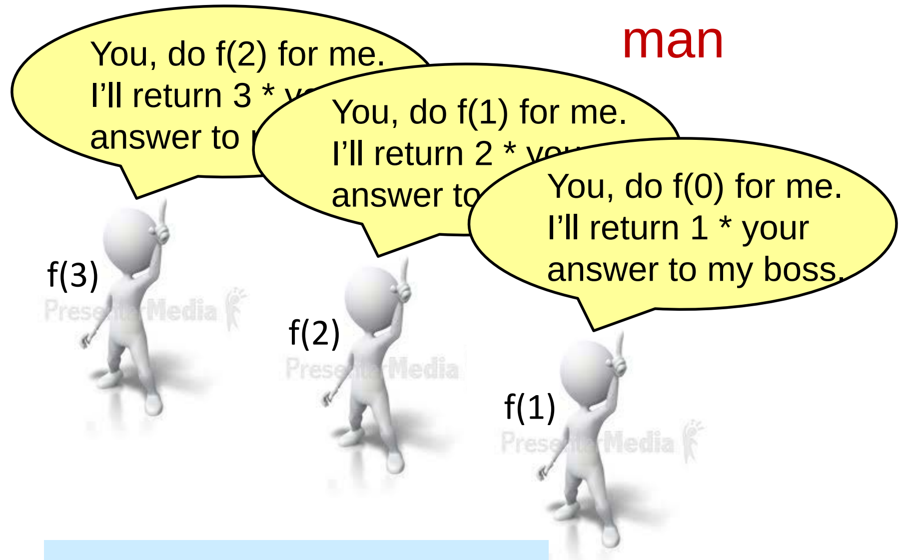
Iteration vs. Recursion: How to compute factorial(3)?



Gist of Recursion (2/4)

Iteration man

Recursion man



Iterative version:

```
// Pre-condition: n >= 0
int factorial_v1(int n) {
  int i, ans = 1;
  for (i = 1; i <= n; i++) {
    ans *= i;
  }
  return ans;
}
```

Recursive version:

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

Gist of Recursion (3/4)

- Problems that lead 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 method is called, eventually the problem is reduced entirely to simple cases, which are easy to solve.
  - The solutions of the smaller problems are then combined to obtain the solution of the original problem.

Gist of Recursion (4/4)

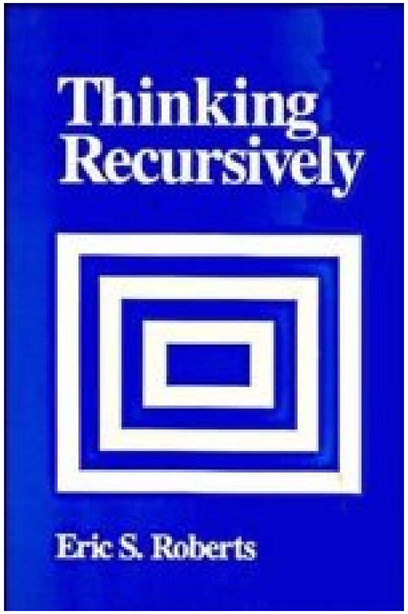
- To write a recursive function:
  - Identify the base case(s) of the relation
  - Identify the recurrence relation

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

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

Thinking Recursively

- It is apparent that to do recursion you need to think "recursively":
  - Breaking a problem into simpler problems that have identical form.



Demo #3 : Choosing Pokémon (1/3)

- How many ways can we choose k items out of n items?

Example: Choose 2 Pokémon out of 4

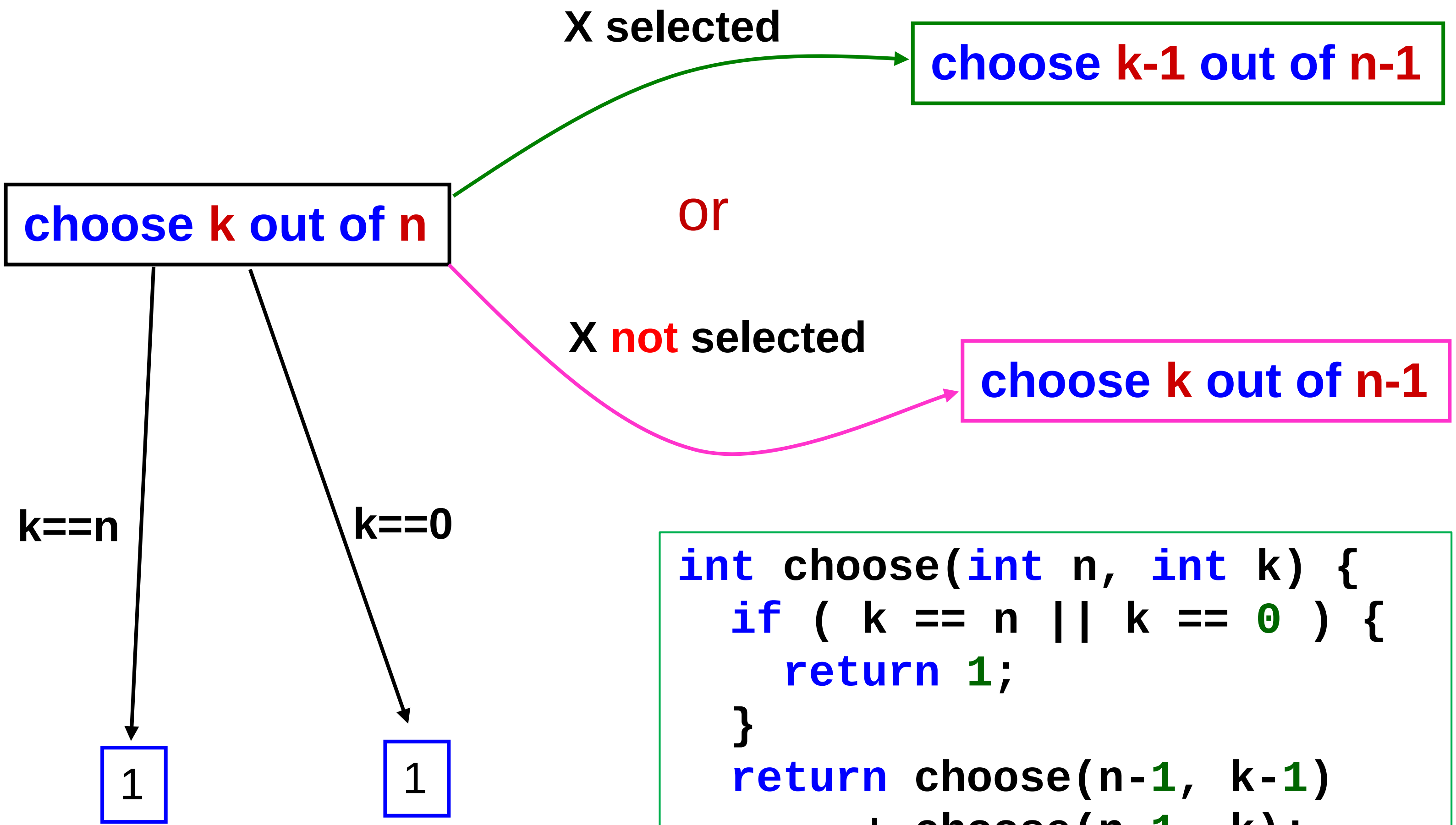
Choose 1 more

Either we choose this Pokémon,
then choose 1 out of remaining 3

Choose 2 more

Or we don't choose this Pokémon,
Then choose 2 out of remaining 3

Demo #3 : Choosing Pokémon (2/3)

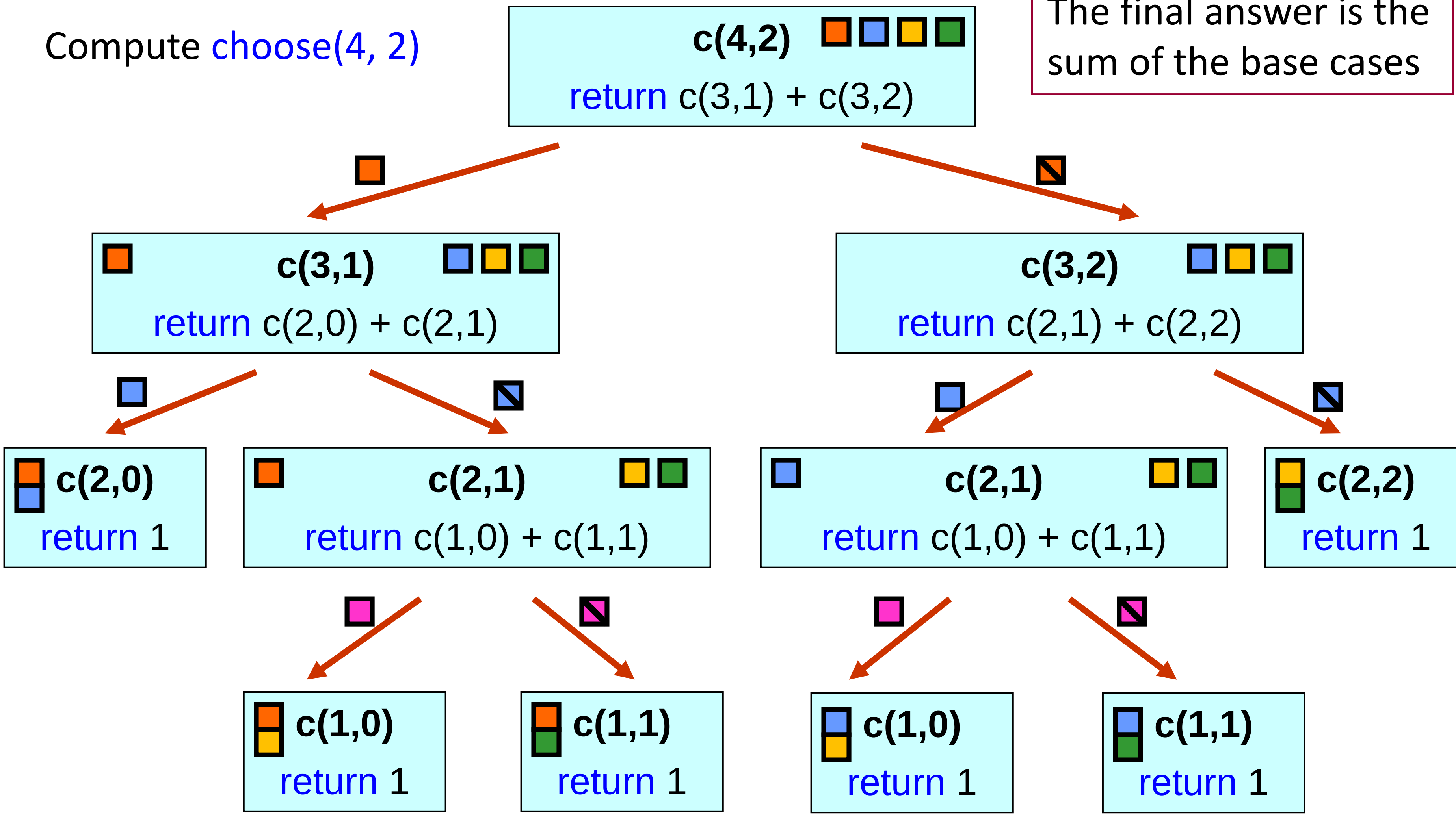



```
int choose(int n, int k) {
  if ( k == n || k == 0 ) {
    return 1;
  }
  return choose(n-1, k-1)
         + choose(n-1, k);
}
```

Demo #3 : Choosing Pokémon (3/3)

Compute choose(4, 2)

The final answer is the sum of the base cases



Demo #4 : Sum Array (1/3)

- Given an array

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

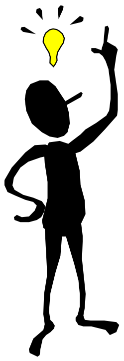
- We want the function call

```
sum_array(arr, 15)
```

to return the sum of elements in this array.

Demo #4 : Sum Array (2/3)

- To get sum_array(arr, 15) to return sum of the array, recursive thinking goes…



9	-2	1	7	3	9	-5	7	2	1	7	-2	0	8	-3

… and get someone to compute the sum for this smaller problem, …

If I handle the last element myself, …

… then my answer is just his answer plus the value of last element!

Demo #4 : Sum Array (3/3)

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]
9	-2	1	7	3	9	-5	7	2	1	7	-2	0	8	-3

Recursive version:

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

sum_array(arr, 15) → -3 + sum_array(arr, 14)

8 + sum_array(arr, 13)

0 + sum_array(arr, 12)

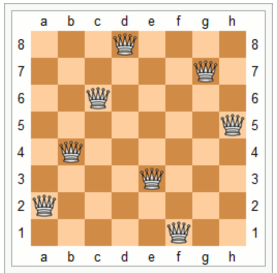
...

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 <u>tedious</u> and the trace tree could be huge (example: Fibonacci series).
- 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 sub-problem(s).

Recursion versus Iteration

- Iteration can be more efficient.
  - Replaces method calls with looping
- Many problems are more naturally solved with recursion, which can provide elegant solutions.
  - Merge Sort (covered in CS1020E)
  - The N Queens problem



One solution.

- Conclusion: choice depends on the 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.

Today's Summary

Simple Recursion

- Recursion as a design methodology
- The components of a recursive code
- Difference between Recursion and Iteration