Introduction to Computer Science
Lecture 9

Nazareno Aguirre

(based on material by Guillaume Hoffmann)

Today

- Functions
- The return statement
- Function prototypes
- Call-by-value

The big picture

- Problem decomposition: taking a problem and breaking it into small, manageable pieces is critical to writing large programs.
- Imperative programming languages typically provide *subroutines* (in C, **functions**) to decompose programs into smaller functional components.
- A program will now consist of one or more functions, one of them being `main()`.
- Program execution begins with `main()`
- main() can *call* other functions, including library functions such as printf(), rand().
- Functions use program variables, whose access is determined by **scope rules**.

Function Definition

- General form:

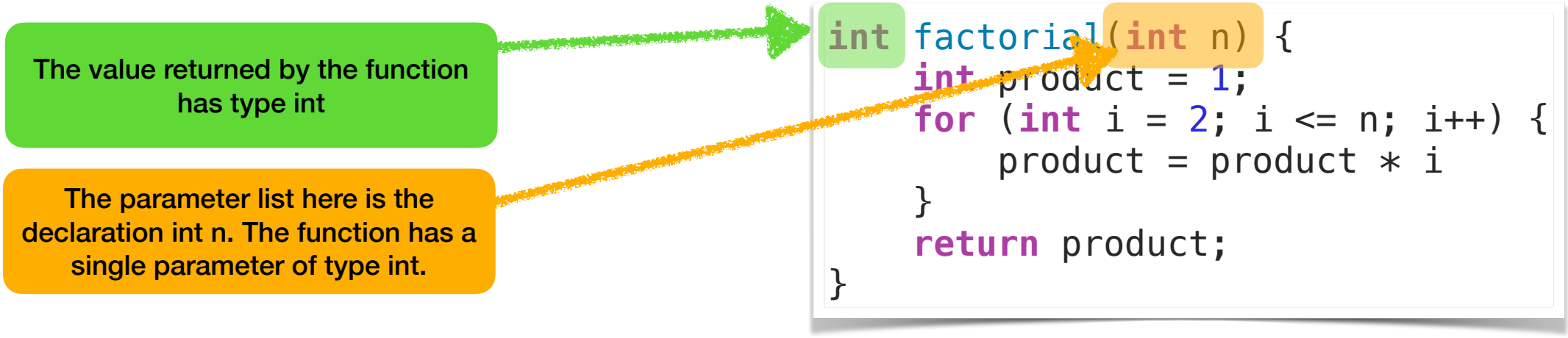
```
type function_name( parameter list ) { declarations statements }
```

- Everything before the first brace is the *header* of the function.
- Everything between the braces is the *body* of the function definition.
- The parameter list is a comma-separated list of declarations.

```
int factorial(int n) {
    int product = 1;
    for (int i = 2; i <= n; i++) {
        product = product * i
    }
    return product;
}
```

Detailed description of the example

- A definition alone does not execute anything. The function needs to be **called** for something to happen.
- Evaluating expression `factorial(7)` causes a call.
- The effect is to execute the code in the function definition, with `n` having the value 7.



Function definition/call example

It does not return any value

It does not have any parameters

```
void wrt_address(void) {
    printf("%s\n%s\n%s\n%s\n%s\n\n",
           "********************",
           "** SANTA CLAUS    **",
           "** NORTH POLE     **",
           "** EARTH          **",
           "********************"),
}
```

- Evaluating expression `wrt_address()` causes function to be called.

```
int main() {
    for (int i = 0; i < 3; i++) {
        wrt_address();
    }
    return 0;
}
```

Function Parameters and Local Variables

- In the definition, the name of the function is followed by a parenthesized list of **parameter declarations**.
- Parameters act as placeholders for values that are passed when the function is called.
- Sometimes, to emphasize their role as placeholders, these parameters are called the **formal parameters** of the function.
- The **function body** is a block and it may contain declarations of **local variables**.

```
int twice(int x) {
    return (2 * x);
}
```

```
int add(int a, int b, int c) {
    int sum = a + b + c;
    return sum;
}
```

One job, one function

- Designing programs as collections of functions is essential for dealing with complexity.
- If programs are adequately separated into functions, it can become easier to reason about single functions and their behavior
- Both the writing and debugging are made easier.
- It is also easier to maintain or modify programs modularized into functions.
- We can change just the set of functions that need to be rewritten and expect the rest of the code to work correctly.
- Functions should be clear, readable and self documenting.
  - It is important that each function has a single responsibility.
  - It is important to choose adequate names for functions, that reflect their behavior

The return statement

- The `return` statement may or may not include an expression.
- The expression being returned can be enclosed in parentheses, but this is not required.
- When a `return` statement is encountered, execution of the function is terminated and control is passed back to the calling environment.
- If the `return` statement contains an expression, then the value of the expression is passed back to the calling environment as well.

```
return;
return ++a;
return (a * b);
```

Return Statements and Returned Values

- There can be zero or more return statements in a function.
  - If there is zero, control comes back to calling environment at the end of the function body.
- Even if a function returns a value, a program does not need to use it:

```
getchar();      // get a char and do nothing with it
c = getchar();  // get a char and assign it to c
```

Function prototypes

- Like variables, C requires functions to be declared before they are used.
- The syntax to declare a function is called the **function prototype**.
- A prototype tells the number and type of arguments that are passed to the function and the type of the value that is to be returned by the function.
- Example: `char toUpper(char);`
- This tells that `toUpper` is a function that takes a single argument of type `char` and returns a `char`.

Example of top-down design: creating a table of powers

```c
#define N 7

long power(int, int);
void prn_heading(void);
void prn_tbl_of_powers(int);

int main(void) {
    prn_headin();
    prn_tbl_of_powers(N);
    return 0;
}
```

```c
void prn_heading(void) {
    printf("\n:::::     A TABLE OF POWERS     :::::\n\n");
}

void prn_tbl_of_powers(int n) {
    int i, j;
    for (i = 1; i <= n; ++i) {
        for (j = 1; j <= n; ++j) {
            if (j == 1) {
                printf("%ld", power(i, j));
            }
            else {
                printf("%9ld", power(i, j));
            }
        }
        putchar('\n');
    }
}
```

```c
long power(int m, int n) {
    int i;
    long product = 1;
    for (i = 1; i <= n; ++i) {
        product = product * m;
    }
    return product;
}
```

Here is the output of the program:

```
:::::  A TABLE OF POWERS  :::::

1         1         1         1         1         1         1
2         4         8        16        32        64       128
3         9        27        81       243       729      2187
.....
```

Alternative Style

- Because function definitions also serve as function prototypes, an **alternative style** is to remove the prototypes and to put the definitions before the calls.
- This makes `main()` go last.

```
long power(int m, int n)
{

}

void prn_tbl_of_powers(int n)
{
    .....
    printf("%ld", power(i, j));
    .....
}

int main(void)
{
    prn_heading();
    prn_tbl_of_powers(N);
    return 0;
}
```

Call-by-value

- Functions are invoked by writing their name and an appropriate list of arguments within parentheses.
  - These arguments match in number and type (or compatible type) the parameters in the parameter list in the function definition.
  - All arguments are passed “by value”. This means that each argument is evaluated, and its value is used locally in the execution of the function.
- In particular, **a variable itself is actually not passed to a function, what is passed is its value**.
  - So, a function does not modify a variable, if this is passed as argument to the function from the calling environment.

Example

```c
int compute_sum(int n);

int main(void) {
    int n = 3, sum;

    printf("%d\n", n);
    sum = compute_sum(n);
    printf("%d\n", n);
    printf("%d\n", sum);
    return 0;
}
```

```c
int compute_sum(int n) {
    int sum = 0;
    while (n > 0) {
        sum = sum + n;
        --n;
    }
    return sum;
}
```

Function Call Summary

- Each expression in the parameter list is evaluated.
- Each value is assigned to its corresponding formal parameter at the beginning of the body of the function.
- The body of the function is executed.
- If a return statement is reached, it is executed and the control is passed back to the calling environment.
- If the return statement includes an expression, it is evaluated and that value is passed back to the calling environment too.
- If no return statement is reached, control is passed back to the calling environment when the end of the body of the function is reached.

Summary

- Functions help structuring C programs. They help breaking down a problem into smaller subproblems, each solved by a corresponding function.
- A return statement ends the execution of a function and passes the control back to the calling environment. If the return statement contains an expression as well, then the value of that expression is passed back as well.
- A function prototype tells the compiler the type and number of its parameters and the type of its returned value. If there are no parameters, the word void is used; if the function returns no value, void is also used as return type.
- Arguments to functions are passed by value in C. They must be type compatible with the corresponding types specified in the function prototype or definition.

Recursion

A recursive definition is the definition of a concept in terms of itself

Recursion in algorithms:

- Natural approach to solve many computational problems
- A *recursive algorithm* uses itself to solve one or more smaller identical problems

Recursion in programming languages:

- Recursive functions implement recursive algorithms
- A *recursive function* includes a call to itself

The structure of a recursive definition

A recursive definition must involve:

- **Base cases**, simple cases in the definition that do not define the concept in terms of itself

- **Recursive cases**, cases whose definition is given in terms of simpler instances of the same concept

Every recursive case must eventually reach a base case.

Components of a Recursive Algorithm

1. What is a smaller ***identical*** problem(s)?
   ? Decomposition
2. How are the answers to smaller problems combined to form the answer to the larger problem?
   ? Composition
3. Which is the smallest problem that can be solved easily (without further decomposition)?
   ? Base/stopping case

An example: Factorial numbers

A classical example of a recursive definition in Mathematics is the definition of a factorial numbers:

- **Base case**: Factorial of one is one:

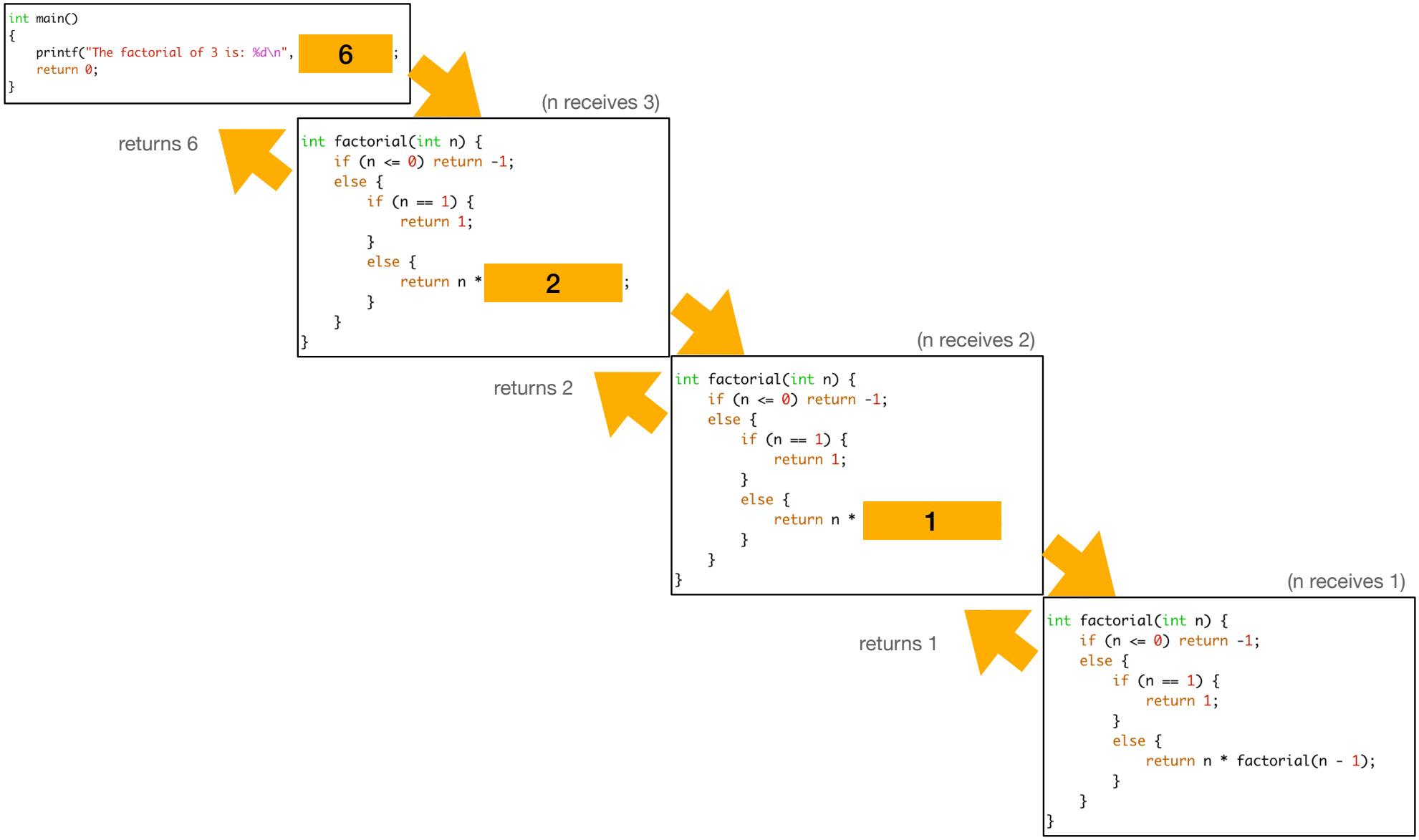
  1! = 1

- **Recursive case**: Factorial of a number n greater than one is n times the factorial of (n-1):

  n! = n x (n - 1)!, provided n > 1

Factorial function

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


int main()
{
printf("The factorial of 3 is: %d\n",
6
;
return 0;
}
(n receives 3)
returns 6
int factorial(int n) {
if (n <= 0) return -1;
else {
if (n == 1) {
return 1;
}
else {
return n *
2
;
}
}
}
(n receives 2)
returns 2
int factorial(int n) {
if (n <= 0) return -1;
else {
if (n == 1) {
return 1;
}
else {
return n *
1
}
}
}
(n receives 1)
returns 1
int factorial(int n) {
if (n <= 0) return -1;
else {
if (n == 1) {
return 1;
}
else {
return n * factorial(n - 1);
}
}
}

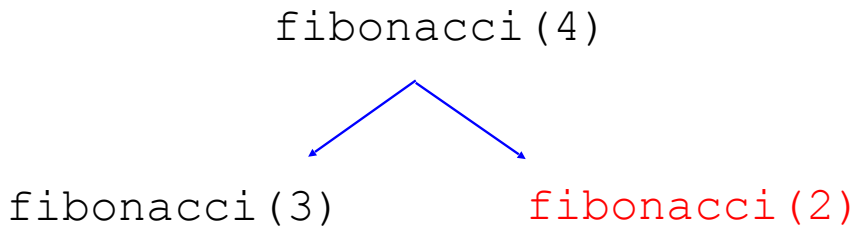
Fibonacci Numbers

- The numbers in the Fibonacci sequence can also be recursive defined:
- The n-th Fibonacci number is:
  - n, if n <= 1
  - The sum of the two previous Fibonacci numbers, if n > 1:
    - fib(n) = fib(n-1) + fib(n-2)

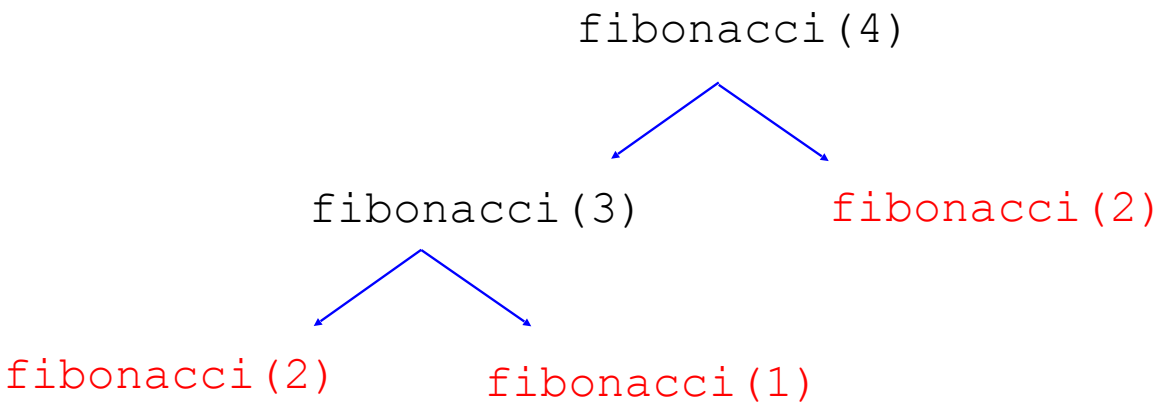
Fibonacci function

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

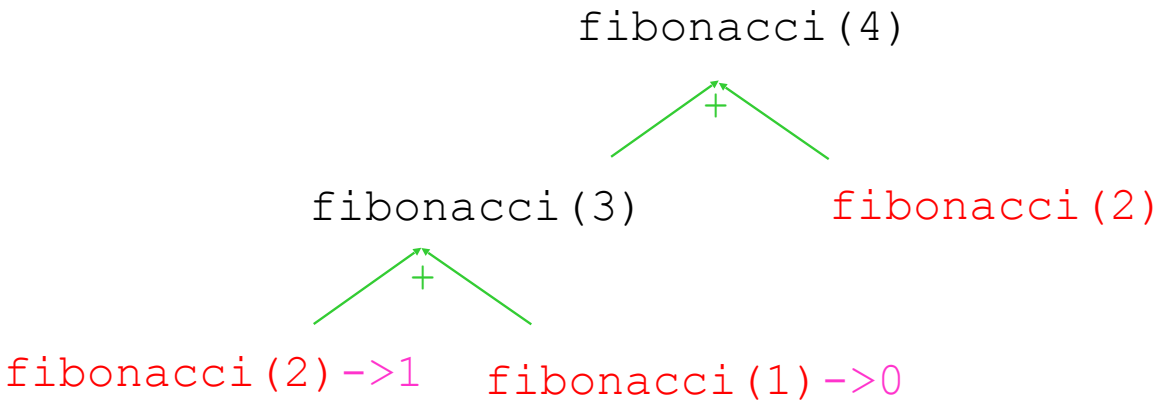
Execution Trace (decomposition)



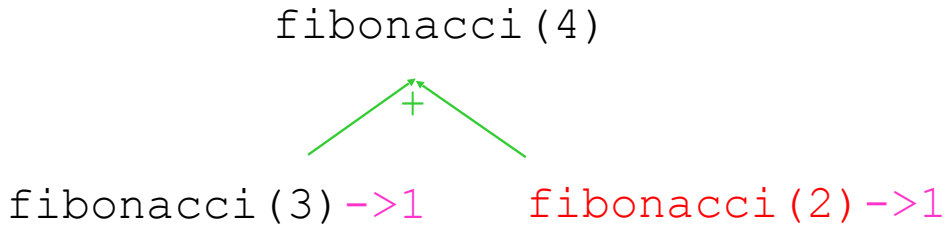
Execution Trace (decomposition)



Execution Trace (composition)



Execution Trace (composition)



Execution Trace (composition)

```
fibonacci(4)->2
```

Crucial aspects of a recursive function

- Case-based definitions
  - Using if-else statement (or some other branching statement)
- Some branches perform recursive calls (recursive cases):
  – "smaller" arguments or solve "smaller" versions of the same task (*decomposition*)
  – Combine the results (*composition*) [if necessary]
- Other branches: no recursive calls
  – stopping cases or base cases

Template

```
… rec_func(…)
{
    if ( … )// base case
    {
    }
    else // decomposition & composition
    {
    }
    return … ; // if not void method
}
```

Is this correct?

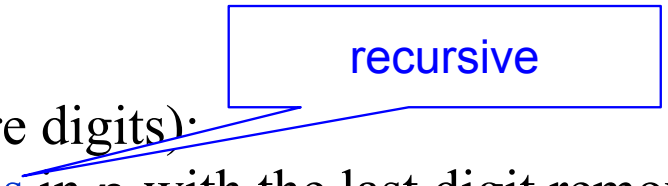
```
public static int factorial(int n)
{
  return factorial(n - 1) * n;
}
```

Infinite recursion

- Infinite Recursion
  - Incorrectly defined recursive solution
    - No decomposition (recursive calls are not on smaller problem instances)
  - Base cases may exist, and not be reachable
    - (Insufficient base cases, incorrectly defined decomposition)
  - No base case
- *Stack:* keeps track of function calls
  - Method begins: add function local data onto the stack
  - Method ends: remove function local data from the stack
- Recursion never stops; stack eventually runs out of space
  - **Stack overflow error**

Number of Zeros in a positive number

- Example: 2030 has 2 zeros
- If `n` is smaller than 10, it has no digits
- If n is grater than 10 (i.e., it has two or more digits):
  - the number of zeros is the number of zeros in `n` with the last digit removed
  - plus an additional 1 if the last digit is zero
- Examples:
  - number of zeros in `20030` is number of zeros in `2003` plus 1
  - number of zeros in `20031` is number of zeros in `2003` plus 0



zero_count function

```
int zero_count(int n) {
    if (n < 10) {
        return 0;
    }
    else {
        int prefix_count = zero_count(n / 10);
        if (n % 10 == 0) {
            return prefix_count + 1;
        }
        else {
            return prefix_count;
        }
    }
}
```

Summary

- Recursive function: a function that calls itself
- Very powerful algorithm design technique
- Recursive algorithm design:
  - Decomposition (smaller identical problems)
  - Composition (combine results)
  - Base case(s) (smallest problem, no recursive calls)
- Implementation
  - Conditional (e.g. if-then-else) statements to separate different cases
  - Avoid infinite recursion
    - Make sure recursive calls are on smaller problem instances (decomposition)
    - Base cases must exist and be reachable from all (valid) function calls