Recursion

CIS 15 : Spring 2007

Functionalia

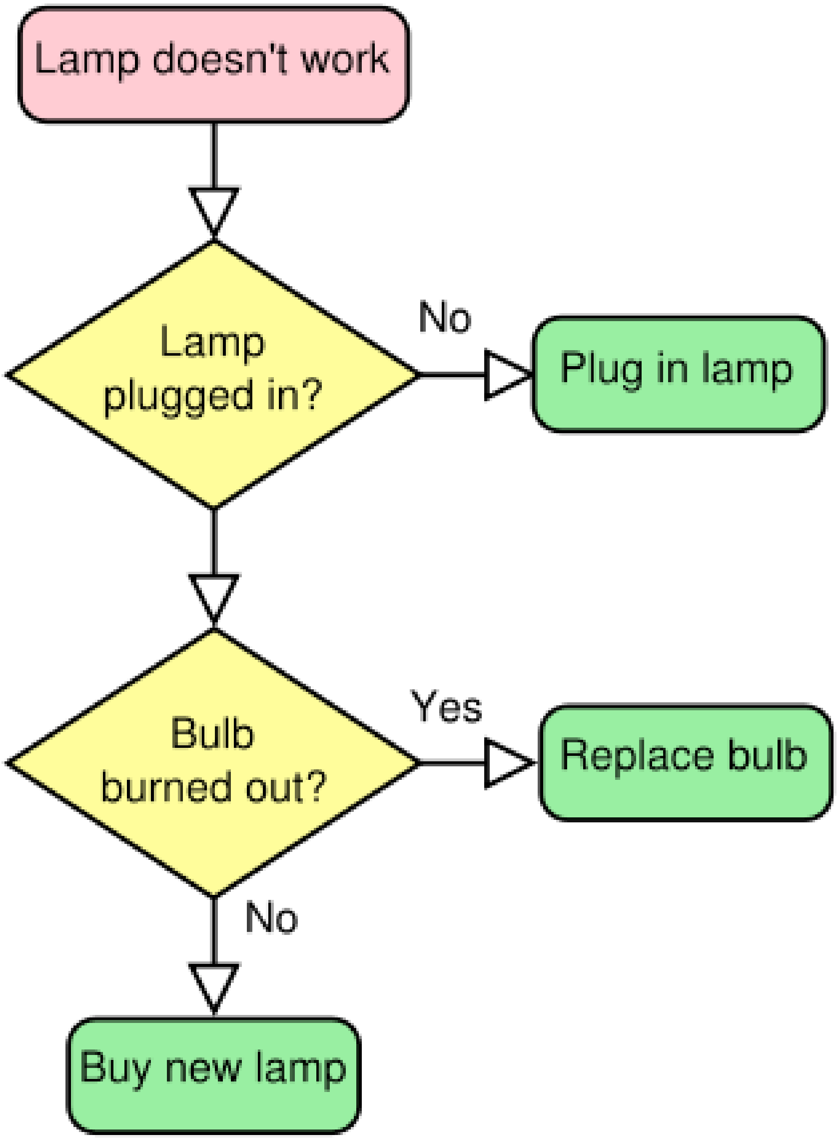
HW 1 is **DUE FRIDAY 23rd, 11:59 PM**

- Do the BASIC Program First!
- Connect Four

Today:

- How to Submit HW
- Some UNIX Stuff
- Recursion

FLOW CHART EXAMPLE



Submitting Homework

Submit Homework by running a script on your homework file:

```
$ ~chipp/Public/bin/hw1-submit hw1.cpp
```

(**hw1.cpp** is the name of your HomeWork 1 file)

Don't forget the tilde (~) at the beginning of the command!

Alternatively, try this:

```
$ /users1/chipp/Public/bin/hw1-submit hw1.cpp
```

And if all else fails, e-mail your **hw1.cpp** file to me at:

chipp@sci.brooklyn.cuny.edu

Finally, please check that your program runs on the UNIX machines!

UNIX Un-Frustations

Fix your Backspace / Delete Key

```
$ pioc^?^?
$ setty erase delete
$ setty erase backspace
```

Run *tcsh* when you login

```
$ tcsh
> exit
$ exit
[logged out]
```

* From Cha. 27,28 and Appendix A: *Just Enough UNIX* *

UNIX Un-Frustations

Fix your Terminal Setting (For those logging in from home)

For *ksh (*the default shell when you log in)

```
$ TERM=vt100
```

```
$ export TERM
```

For *tcsh* (if you specify it when you log in)

```
$ setenv TERM vt100
```

Advanced Users (try running *screen*)

```
$ screen
```

More Info on Screen

- http://www.bangmoney.org/presentations/screen.html

Recursion

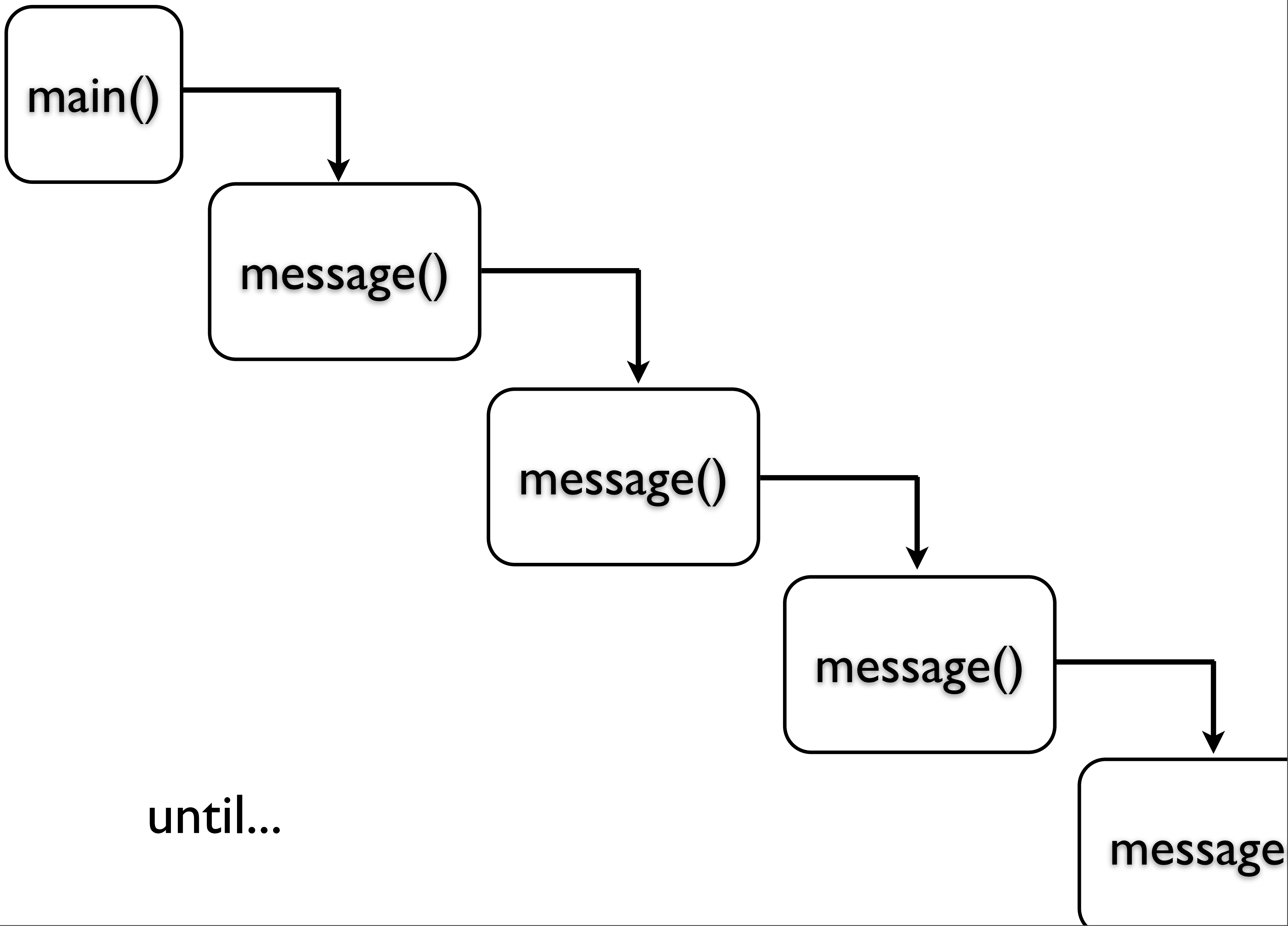
```
#include <iostream>
using namespace std;
void message()
{
   cout << "I've been feeling somewhat repetitive lately.\n";
   message();
}

void main()
{
   message();
}
```

Try it out

```
$ cd recursion
$ ./recur
```

Recursion

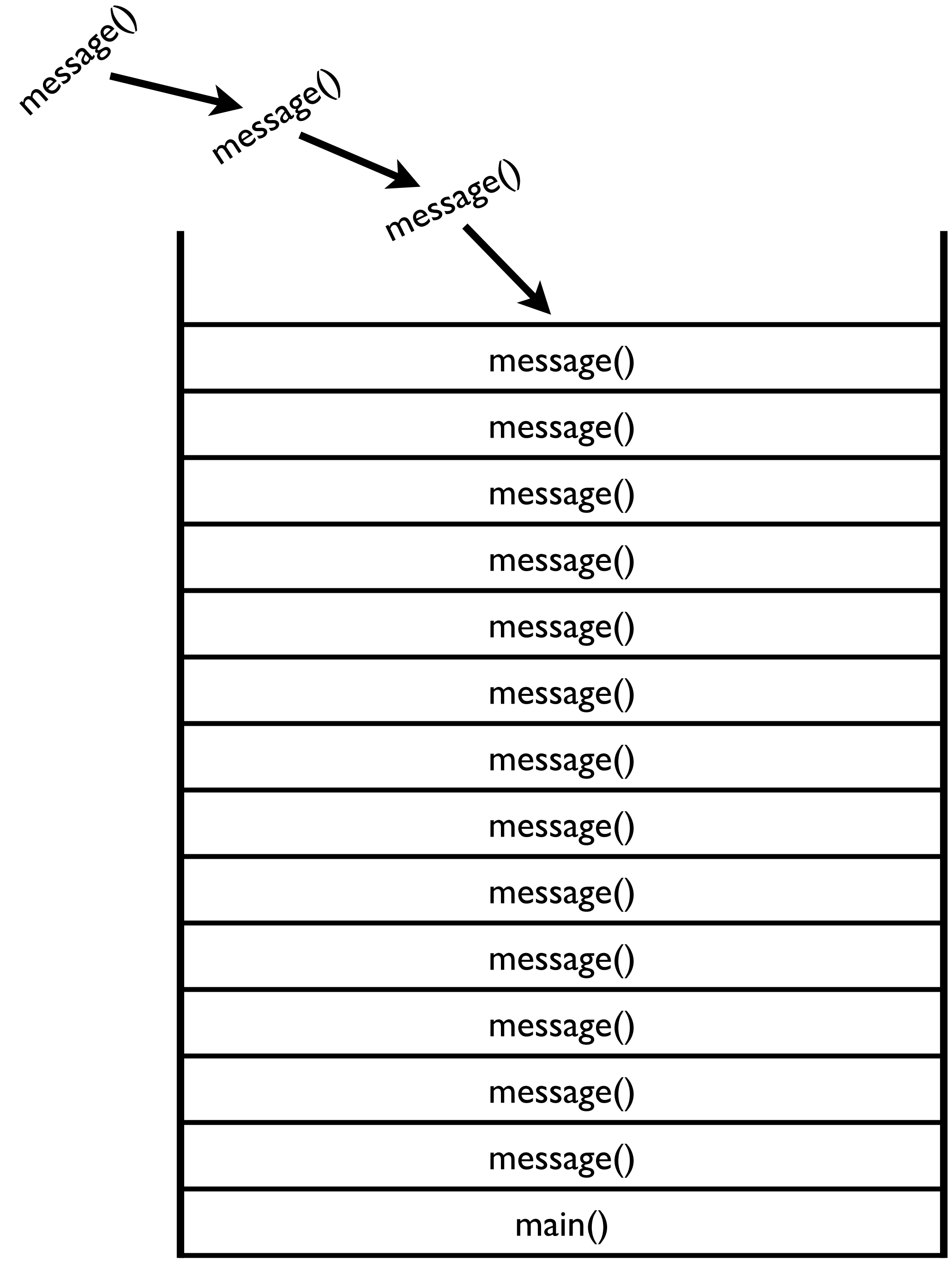


until...

Call Stack Over Flow

Call Stack - a special **stack (LIFO)** is used to keep track of information about the *currently active functions* in a computer program (technically a single task in a program).

<ins>Stored Information:</ins> return address, function parameters, return values, local variables.



Recursion (with a Limit)

```
#include <iostream>
using namespace std;

void message(int times)
{
   if(times > 0)
   {
      cout << "I've been feeling somewhat repetitive lately.\n";
      message(times - 1);
   }
}

void main()
{
   message(5);
}
```

Try it out

```
$ recur_times
I've been feeling somewhat repetitive lately.
I've been feeling somewhat repetitive lately.
I've been feeling somewhat repetitive lately.
I've been feeling somewhat repetitive lately.
I've been feeling somewhat repetitive lately.
```

How many times does it run?

```
$ dbx recur_times
(dbx) trace in message
(dbx) run
```

Recursion (with a Limit and Home Grown Trace Options)

```
#include <iostream>
using namespace std;

void message(int times)
{
   cout << “message() is called with times = “ << times << endl;
   if(times > 0)
   {
      cout << “I’ve been feeling somewhat repetitive lately.\n”;
      message(times - 1);
   }
   cout << “message() returns with times = “ << times << endl;
}

void main()
{
   message(5);
}
```

What is the Output?

What is the Output?

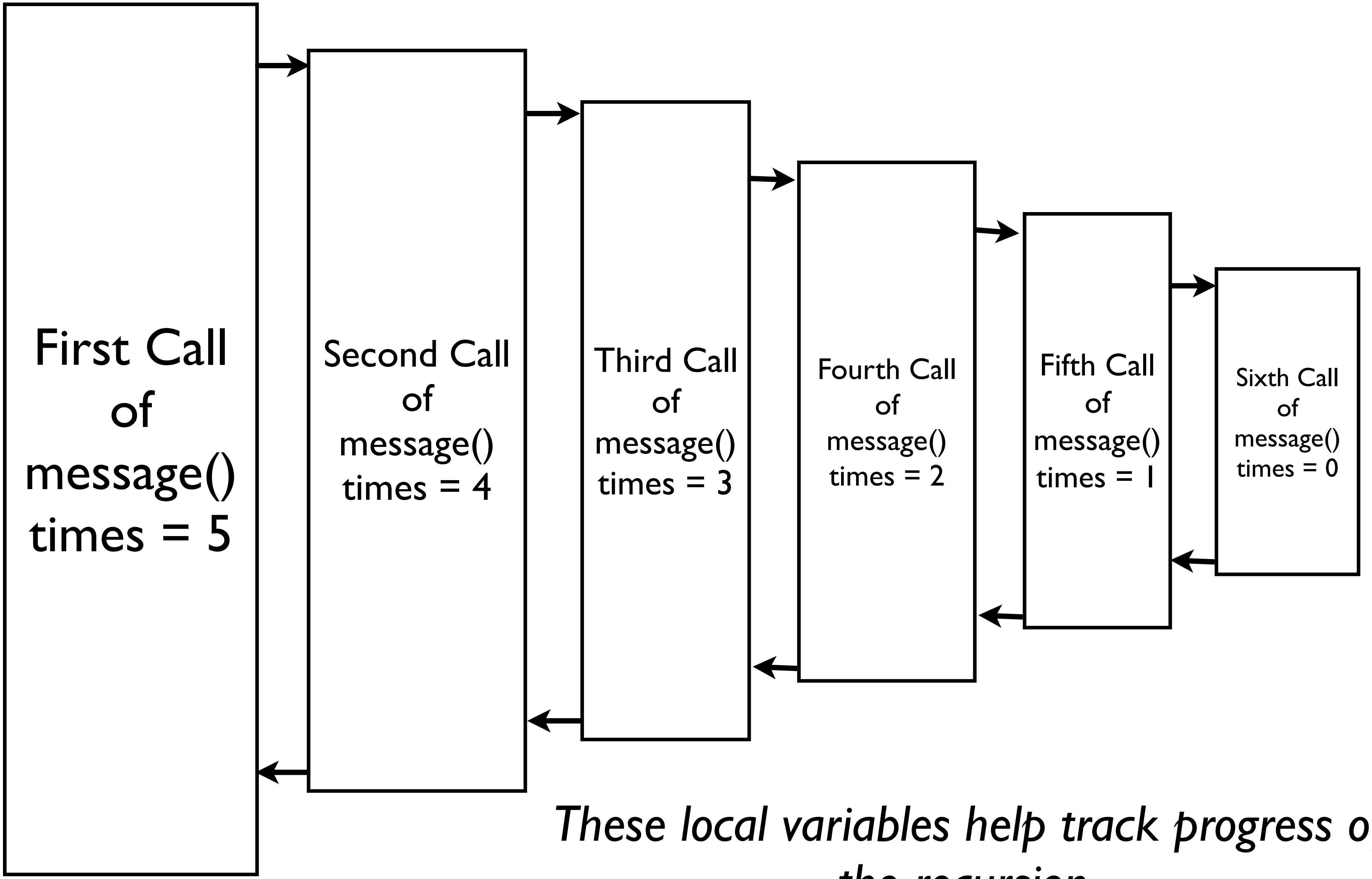
```
$ ./recur_times_trace
message() is called with times = 5
I've been feeling somewhat repetitive lately.
message() is called with times = 4
I've been feeling somewhat repetitive lately.
message() is called with times = 3
I've been feeling somewhat repetitive lately.
message() is called with times = 2
I've been feeling somewhat repetitive lately.
message() is called with times = 1
I've been feeling somewhat repetitive lately.
message() is called with times = 0
message() returns with times = 0
message() returns with times = 1
message() returns with times = 2
message() returns with times = 3
message() returns with times = 4
message() returns with times = 5
```

What is the Output?

```
$ ./recur_times_trace
message() is called with times = 5
I've been feeling somewhat repetitive lately.
message() is called with times = 4
I've been feeling somewhat repetitive lately.
message() is called with times = 3
I've been feeling somewhat repetitive lately.
message() is called with times = 2
I've been feeling somewhat repetitive lately.
message() is called with times = 1
I've been feeling somewhat repetitive lately.
message() is called with times = 0
message() returns with times = 0
message() returns with times = 1
message() returns with times = 2
message() returns with times = 3
message() returns with times = 4
message() returns with times = 5
```

How many variables named times exist?

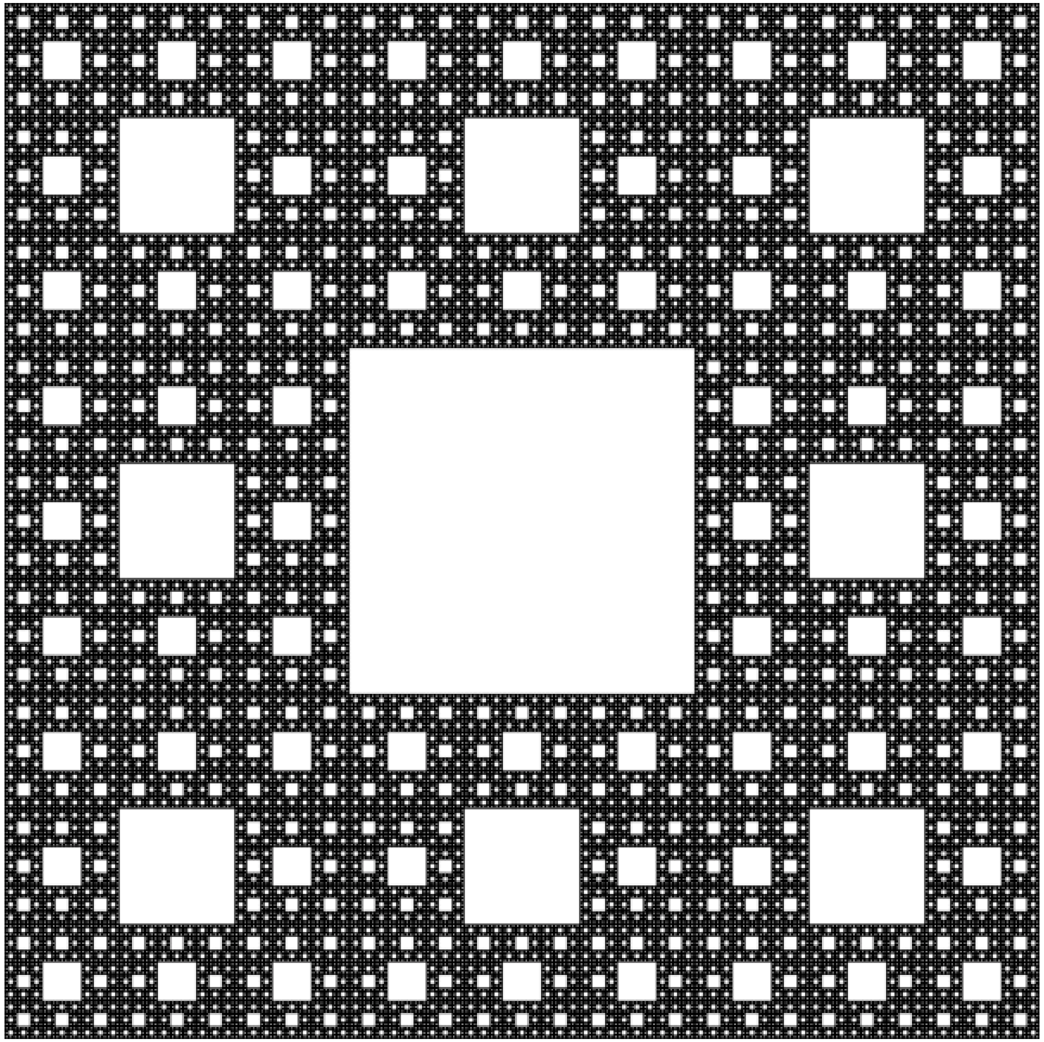
There are 6 variables named times (all local variables)



These local variables help track progress of the recursion.

Solving Problems

Recursion can solve a problem if it can be broken down into successive smaller problems that are identical to the overall problem.



Sierpinski carpet (a plane fractal) can be drawn and defined recursively.

Solving Problems with Recursion

1. Any problem that can be solved **recursively** can be solved iteratively with a loop.

2. Recursive functions incur a lot of *overhead* - the time, and the space necessary to store the return address, local variables, and parameters on the Call Stack. (Thus, they are not always desirable solutions). There are ways to optimize recursion (i.e. *Tail Recursion*).

3. Recursion is fundamental to many ***functional*** programming languages: *Lisp, Scheme*... (C and what you use of C++ now is *Procedural)*

Breaking a Problem Down into a Recursive Function

Recursive Functions are broken down to two parts:

1. *Base Case*: one case where the problem can be solved *without* a recursive call.

2. *Recursive Case:* reduce the problem down into smaller similar problems.

Those of you studying Discrete Mathematics :

Recursion Problem Solving is analogous to an Inductive Proof and it's two parts: 1. the *Base Step* and 2. the *Inductive Step*

Example: Factorials

A **factorial** of an non-negative number ***n*** is defined with these rules:

- If $n = 0$ then $n! = 1$
- If $n > 0$ then $n! = 1 \times 2 \times 3 \times .... \times n - 1 \times n$

n	*n!*
0	1
1	1
2	1 X 2 = 2
3	1 X 2 X 3 = 6
4	1 X 2 X 3 X 4 = 24
5	1 X 2 X 3 X 4 X 5 = 120
6	1 X 2 X 3 X 4 X 5 X 6 = 720
7	1 X 2 X 3 X 4 X 5 X 6 X 7 = 5,040

Example: Factorials

A **factorial** of an non-negative number ***n*** is defined with these rules:

- If $n = 0$ then $n! = 1$
- If $n > 0$ then $n! = 1 \times 2 \times 3 \times .... \times n - 1 \times n$

Define factorial $n!$ as a function *factorial(n)*

- If $n = 0$ then $factorial(n) = 1$
- If $n > 0$ then $factorial(n) = 1 \times 2 \times 3 \times .... \times n -1 \times n$

Example: Factorials

A **factorial** of an non-negative number ***n*** is defined with these rules:

- If $n = 0$ then $n! = 1$
- If $n > 0$ then $n! = 1 \times 2 \times 3 \times \ldots \times n - 1 \times n$

Define factorial *n!* as a function *factorial(n)*

- If $n = 0$ then $factorial(n) = 1$
- If $n > 0$ then $factorial(n) = \boxed{1 \times 2 \times 3 \times \ldots \times n - 1} \times n$

What is another way of writing this?

Example: Factorials

A **factorial** of an non-negative number ***n*** is defined with these rules:

- If $n = 0$ then $n! = 1$
- If $n > 0$ then $n! = 1 \times 2 \times 3 \times .... \times n - 1 \times n$

Define factorial $n!$ as a function *factorial(n)*

- If $n = 0$ then $factorial(n) = 1$
- If $n > 0$ then $factorial(n) = \boxed{1 \times 2 \times 3 \times .... \times n - 1} \times n$

factorial(n - 1) !!!!

Example: Factorials

A **factorial** of an non-negative number ***n*** is defined with these rules:

- If $n = 0$ then $n! = 1$
- If $n > 0$ then $n! = 1 \times 2 \times 3 \times .... \times n - 1 \times n$

Define factorial $n!$ as a function *factorial(n)*

- If $n = 0$ then $factorial(n) = 1$
- If $n > 0$ then $factorial(n) = \boxed{1 \times 2 \times 3 \times .... \times n - 1} \times n$

First rule solves the problem without recursion:

- If $n = 0$ then $factorial(n) = 1$

Second rule can be broken

- If $n > 0$ then $factorial(n) = factorial(n - 1) \times n$

Example: Factorials

A **factorial** of an non-negative number ***n*** is defined with these rules:

- If $n = 0$ then $n! = 1$
- If $n > 0$ then $n! = 1 \times 2 \times 3 \times .... \times n - 1 \times n$

Define factorial $n!$ as a function *factorial(n)*

- If $n = 0$ then $factorial(n) = 1$
- If $n > 0$ then $factorial(n) = 1 \times 2 \times 3 \times .... \times n - 1 \times n$

First rule solves the problem without recursion:

- If $n = 0$ then $factorial(n) = 1$ BASE CASE

Second rule can be broken

- If $n > 0$ then $factorial(n) = factorial(n - 1) \times n$ RECURSIVE CASE

Translated into C++ Code

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

```
$ factorial
```

Another Example: Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...

How is this series generated?

Another Example: Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...

2 + 3 = 5

A number in the Fibonacci Sequence is defined as the sum of the previous two numbers.

Another Example: Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...

Mathematically Defined:

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

n	0	1	2	3	4	5	6	7	
F(n)	0	1	1	2	3	5	8	13	

Another Example: Fibonacci Sequence

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

n	0	1	2	3	4	5	6	7	
F(n)	0	1	1	2	3	5	8	13	

F(4) = ?

Another Example: Fibonacci Sequence

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

n	0	1	2	3	4	5	6	7	
F(n)	0	1	1	2	3	5	8	13	

F(4) = F(3) + F(2)

Another Example: Fibonacci Sequence

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

n	0	1	2	3	4	5	6	7	
F(n)	0	1	1	2	3	5	8	13	

F(4) = F(3) + F(2)
F(3) = F(2) + F(1)
F(2) = F(1) + F(0)

Another Example: Fibonacci Sequence

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

n	0	1	2	3	4	5	6	7	
F(n)	0	1	1	2	3	5	8	13	

F(4) = F(3) + F(2)
F(3) = F(2) + F(1)
F(2) = F(1) + F(0)
F(1) = 1
F(0) = 0

Another Example: Fibonacci Sequence

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

n	0	1	2	3	4	5	6	7	
F(n)	0	1	1	2	3	5	8	13	

F(4) = F(3) + F(2)
F(3) = F(2) + F(1)
F(2) = F(1) + F(0) = 1 + 0 = 1
F(1) = 1
F(0) = 0

Another Example: Fibonacci Sequence

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

n	0	1	2	3	4	5	6	7	
F(n)	0	1	1	2	3	5	8	13	

F(4) = F(3) + F(2)
F(3) = F(2) + F(1) = 1 + 1 = 2
F(2) = F(1) + F(0) = 1 + 0 = 1
F(1) = 1
F(0) = 0

Another Example: Fibonacci Sequence

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

n	0	1	2	3	4	5	6	7	
F(n)	0	1	1	2	3	5	8	13	

F(4) = F(3) + F(2) = 2 + 1 = 3
F(3) = F(2) + F(1) = 1 + 1 = 2
F(2) = F(1) + F(0) = 1 + 0 = 1
F(1) = 1
F(0) = 0

Write a recursive Fibonacci Function!

```
int fib(int n)
{



}
```

Write a recursive Fibonacci Function!

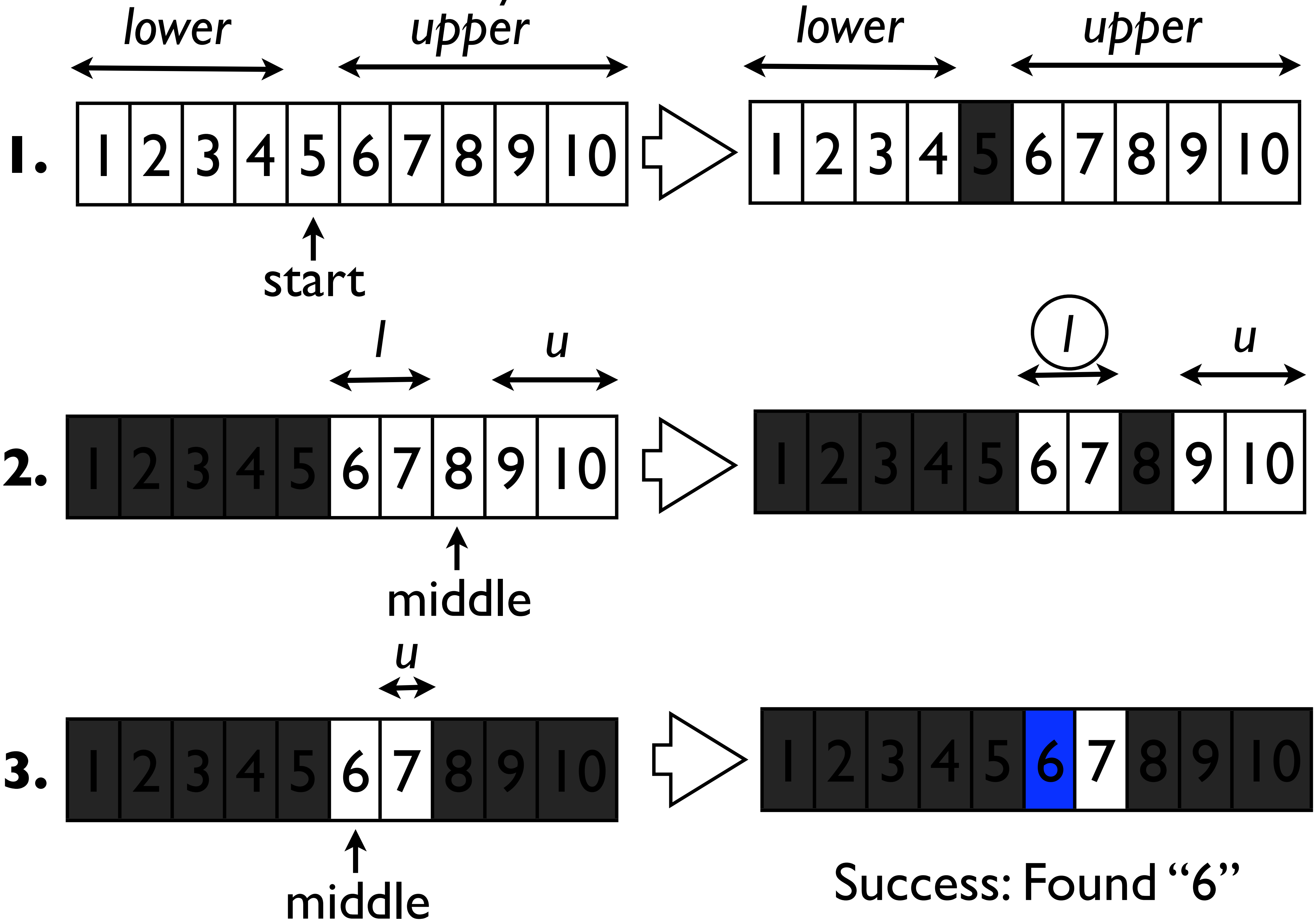
```
int fib(int n)
{
  if(n <= 0)
    return 0;             // BASE CASE
  else if(n == 1)
    return 1;             // BASE CASE
  else
    return fib(n-1) + fib(n-2);  // RECURSIVE CASE
}
```

Convert the Binary Search into a Recursive Solution

```
int binarySearch(int array[], int numelems, int value)
{
   int first = 0,                  // First array element
       last = numelems - 1,        // Last array element
       middle,                     // Mid point of search
       position = -1;              // Position of search value
   bool found = false;             // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;      // Calculate mid point
      if (array[middle] == value)       // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)   // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;            // If value is in upper half
   }
   return position;
}
```

Binary Search for number ‘6’



Summary

Recursive Functions

Elegant implementations - does not always run elegantly.

Solutions comprise of

- BASE Case(s)
- RECURSIVE Case(s)

Factorials, Fibonacci Sequence, and Binary Search...

READINGS: 19.1 - 19.4, 19.6, and 19.8 - 19.10