

CS221: Algorithms and Data Structures

Recursion and Iteration

Alan J. Hu

(Borrowing many slides from Steve Wolfman)

Learning Goals

By the end of this unit, you will be able to...

- Describe the relationship between recursion/iteration and induction (e.g., take a recursive code fragment and express it mathematically in order to prove its correctness inductively)
- Evaluate the effect of recursion on space complexity
- Describe how tail recursive algorithms can require less space
- Recognize algorithms as recursive or iterative
- Convert between recursive and iterative solutions
- Draw a recursion tree, and relate the depth to the number of recursive calls, and the size of the runtime stack
- Identify or produce an example of infinite recursion

Thinking Recursively

DO NOT START WITH CODE. Instead, write the *story* of the problem, in natural language.

Define the problem: What should be done given a particular input?

Identify and solve the (usually simple) base case(s).

Start solving a more complex version.

As soon as you break the problem down in terms of **any simpler version**, call the function recursively and **assume it works**. Do **not** think about how!

Thinking Recursively

This is the secret to thinking recursively!

Your solution will work as long as:

- (1) you've broken down the problem right
- (2) each recursive call really is simpler/smaller, and
- (3) you make sure all calls will eventually hit base case(s).

As soon as you break the problem down in terms of **any simpler version**, call the function recursively and **assume it works**. Do **not** think about how!

How a Computer Does Recursion

- This is NOT a good way to “understand recursion”!!!

How a Computer Does Recursion

- This is NOT a good way to “understand recursion”!!!
- But understanding how a computer actually does recursion IS important to understand the time and space complexity of recursive programs, and how to make them run better.

Function/Method Calls

- A function or method call is an interruption or aside in the execution flow of a program:

...

```
int a, b, c, d;
```

```
a = 3;
```

```
b = 6;
```

```
c = foo(a,b);
```

```
d = 9;
```

...

```
int foo(int x, int y) {  
    while (x>0) {  
        y++;  
        x >>= 1;  
    }  
    return y  
}
```

Function Calls in Daily Life

- How do you handle interruptions in daily life?
 - You're at home, working on CPSC221 project.
 - You stop to look up something in the book.
 - Your roommate/spouse/partner/parent/etc. asks for your help moving some stuff.
 - Your buddy calls.
 - The doorbell rings.

Function Calls in Daily Life

- How do you handle interruptions in daily life?
 - You're at home, working on CPSC221 project.
 - You stop to look up something in the book.
 - Your roommate/spouse/partner/parent/etc. asks for your help moving some stuff.
 - Your buddy calls.
 - The doorbell rings.
- You stop what you're doing, you memorize where you were in your task, you handle the interruption, and then you go back to what you were doing.

Function Calls in Daily Life

- How do you handle interruptions in daily life?
 - You're at home, working on CPSC221 project.
 - You stop to look up something in the book.
 - Your roommate/spouse/partner/parent/etc. asks for you help moving some stuff.
 - Your buddy calls.
 - The doorbell rings.
- You stop what you're doing, you memorize where you were in your task, you handle the interruption, and then you go back to what you were doing.

LIFO!

That's a stack!

Activation Records in Daily Life

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I am reading about the delete function in Koffman p. 26

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I have moved 20lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I am listening to my buddy tell some inane story about last night.

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

My buddy is just about to get to the point where he pukes...

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I am signing for a FedEx package.

My buddy is just about to get to the point where he pukes...

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

My buddy is just about to get to the point where he pukes...

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

My buddy has finally finished his story...

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I have moved 40lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I have moved 60lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I have moved 80lbs of steer manure to the garden.

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I am reading about the delete function in Koffman p. 27

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I am reading about the delete function in Koffman p. 28

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I am working on line X of my stack.cpp file...

Activation Records in Daily Life

I have finished my stack.cpp file! 😊

Activation Records in Daily Life

Activation Records on a Computer

- A computer handles function/method calls in exactly the same way! (Also, “interrupts”)

Activation Records on a Computer




```
...  
int a, b, c, d;  
a = 3;  
b = 6;  
c = foo(a,b);  
d = 9;  
...
```

```
int foo(int x, int y) {  
    while (x>0) {  
        y++;  
        x >>= 1;  
    }  
    return y  
}
```

Activation Records on a Computer

...

 int a, b, c, d;

a = 3;

b = 6;

c = foo(a,b);

d = 9;

...

```
int foo(int x, int y) {
```

```
    while (x>0) {
```

```
        y++;
```

```
        x >>= 1;
```

```
    }
```

```
    return y
```

```
}
```

a=?, b=?, c=?, d=?

Activation Records on a Computer

```
...  
int a, b, c, d;  
→ a = 3;  
  b = 6;  
  c = foo(a,b);  
  d = 9;  
...
```

```
int foo(int x, int y) {  
    while (x>0) {  
        y++;  
        x >>= 1;  
    }  
    return y  
}
```

a=3, b=?, c=?, d=?

Activation Records on a Computer

```
...  
int a, b, c, d;  
a = 3;  
b = 6;  
c = foo(a,b);  
d = 9;  
...
```



```
int foo(int x, int y) {  
    while (x>0) {  
        y++;  
        x >>= 1;  
    }  
    return y  
}
```

a=3, b=6, c=?, d=?

Activation Records on a Computer

...

int a, b, c, d;

a = 3;

b = 6;

c = foo(a,b);

d = 9;

...



```
int foo(int x, int y) {
```

```
    while (x>0) {
```

```
        y++;
```

```
        x >>= 1;
```

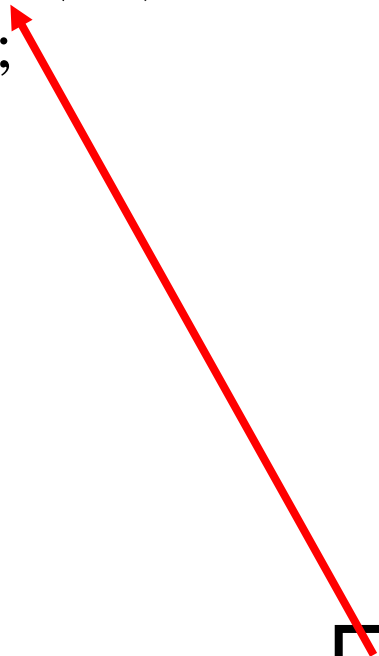
```
    }
```

```
    return y
```

```
}
```

x=3,y=6

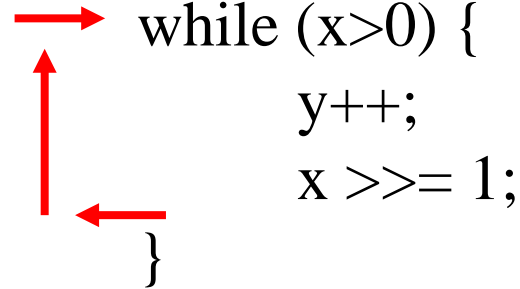
a=3, b=6, c=?, d=?



Activation Records on a Computer

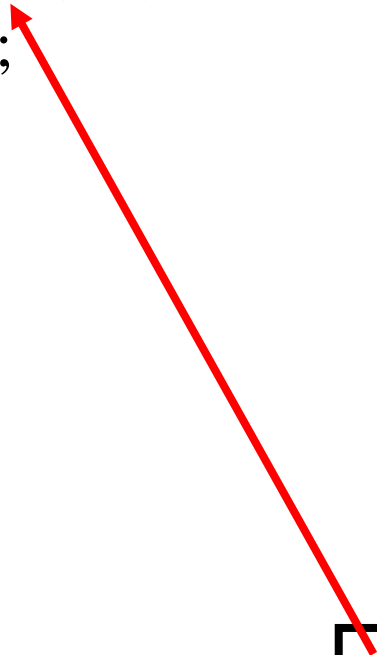
```
...  
int a, b, c, d;  
a = 3;  
b = 6;  
c = foo(a,b);  
d = 9;  
...
```

```
int foo(int x, int y) {  
  while (x>0) {  
    y++;  
    x >>= 1;  
  }  
  return y  
}
```



x=1,y=7

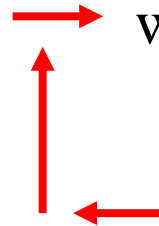
a=3, b=6, c=?, d=?



Activation Records on a Computer

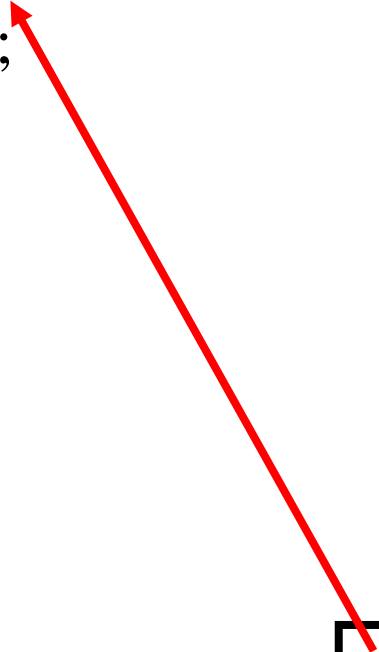
```
...  
int a, b, c, d;  
a = 3;  
b = 6;  
c = foo(a,b);  
d = 9;  
...
```

```
int foo(int x, int y) {  
  while (x>0) {  
    y++;  
    x >>= 1;  
  }  
  return y  
}
```



x=0,y=8

a=3, b=6, c=?, d=?



Activation Records on a Computer

```
...  
int a, b, c, d;  
a = 3;  
b = 6;  
c = foo(a,b);  
d = 9;  
...
```

```
int foo(int x, int y) {  
    while (x>0) {  
        y++;  
        x >>= 1;  
    }  
    return y  
}
```



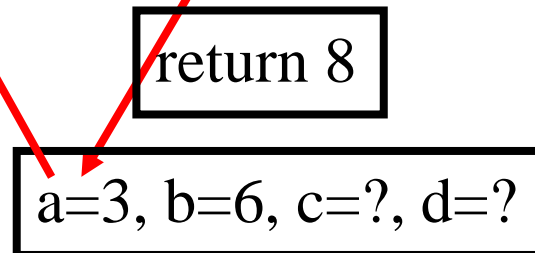
x=0,y=8

a=3, b=6, c=?, d=?

Activation Records on a Computer

```
...  
int a, b, c, d;  
a = 3;  
b = 6;  
c = foo(a,b);  
d = 9;  
...
```

```
int foo(int x, int y) {  
    while (x>0) {  
        y++;  
        x >>= 1;  
    }  
    return y  
}
```



Activation Records on a Computer

```
...  
int a, b, c, d;  
a = 3;  
b = 6;  
→ c = foo(a,b);  
d = 9;  
...
```

```
int foo(int x, int y) {  
    while (x>0) {  
        y++;  
        x >>= 1;  
    }  
    return y  
}
```

a=3, b=6, c=8, d=?

Activation Records on a Computer

...

int a, b, c, d;

a = 3;

b = 6;

c = foo(a,b);

→ d = 9;

...

```
int foo(int x, int y) {
```

```
    while (x>0) {
```

```
        y++;
```

```
        x >>= 1;
```

```
    }
```

```
    return y
```

```
}
```

a=3, b=6, c=8, d=9

Recursion is handled the same way!

$$fib_n = \begin{cases} \overset{\text{red arrow}}{1} & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$


n=4

Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & if n = 1 \\ \rightarrow 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

n=4

Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$


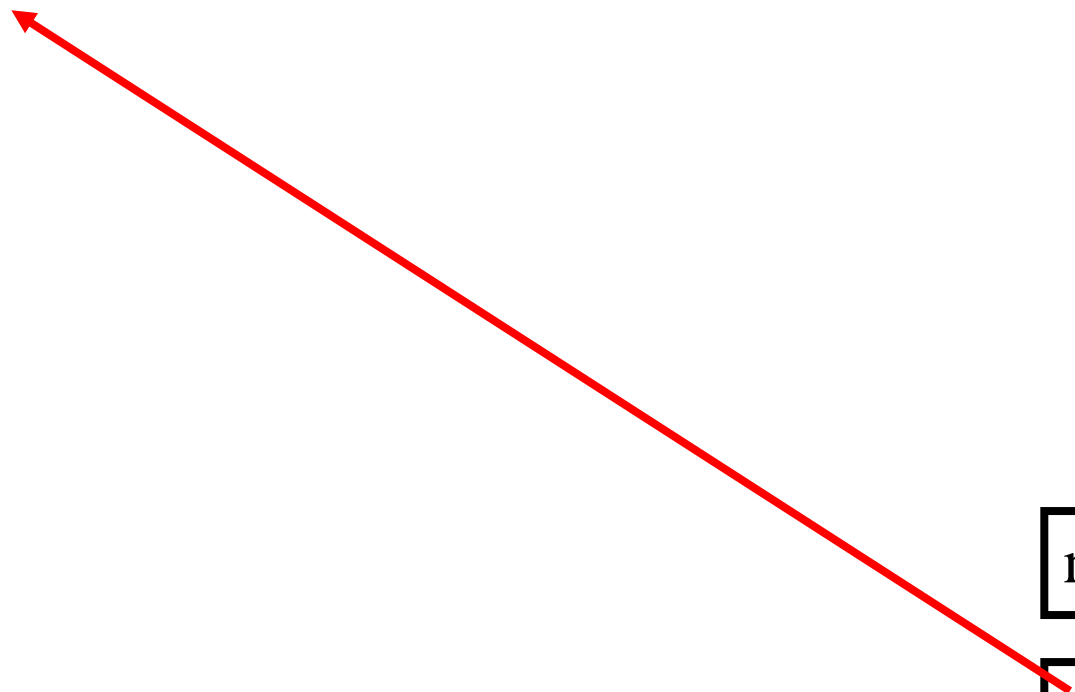
n=4

Recursion is handled the same way!

$$fib_n = \begin{cases} \overset{\rightarrow}{1} & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$

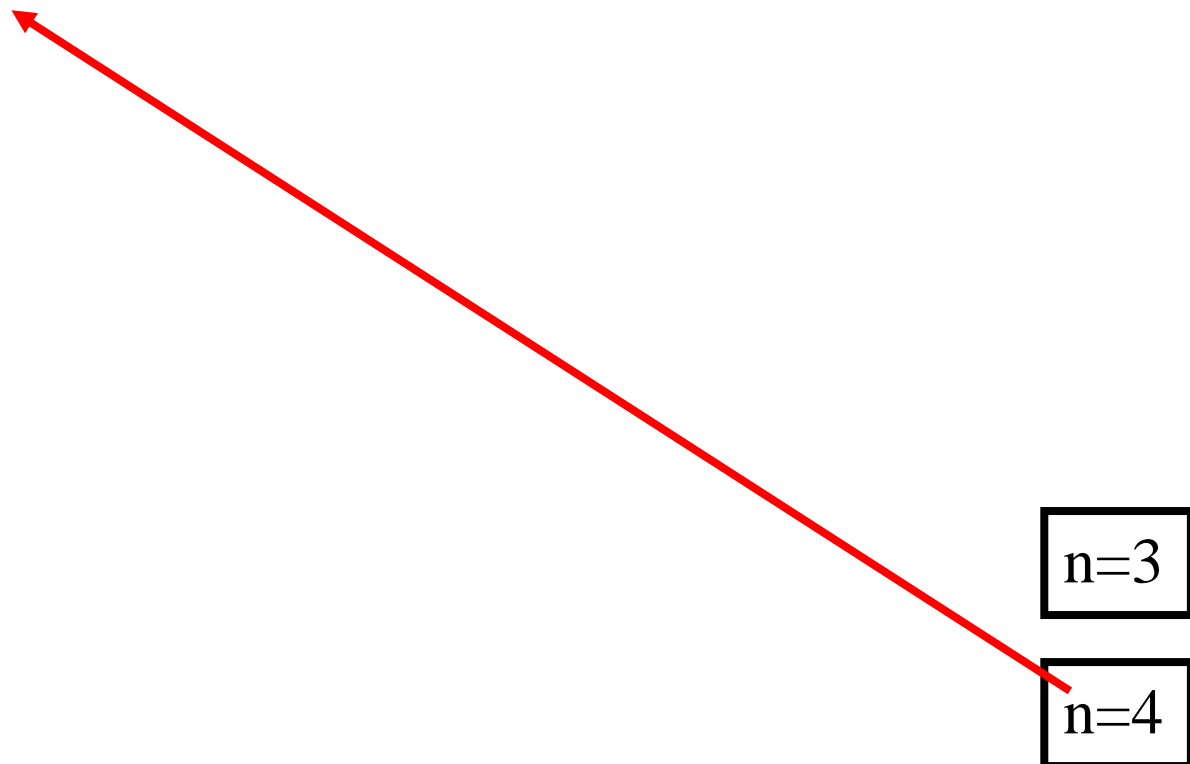
n=3

n=4



Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ \rightarrow 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$

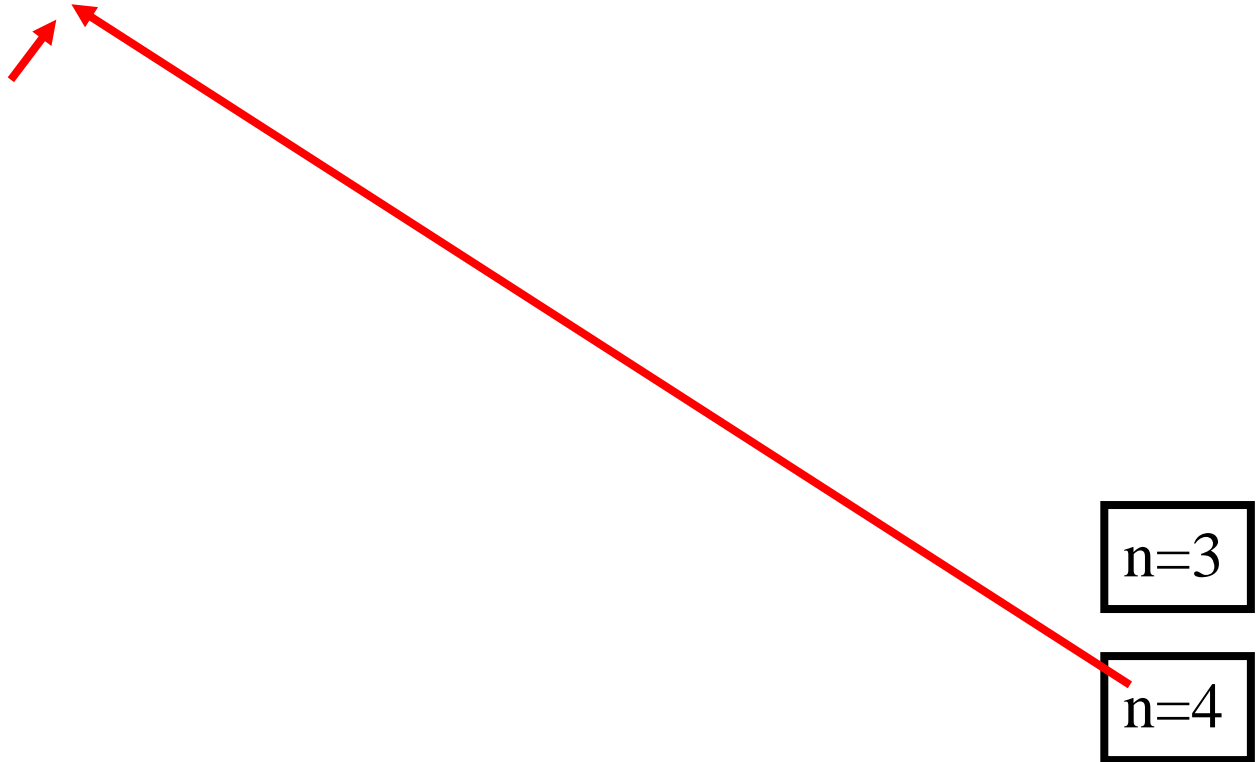


n=3

n=4

Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$



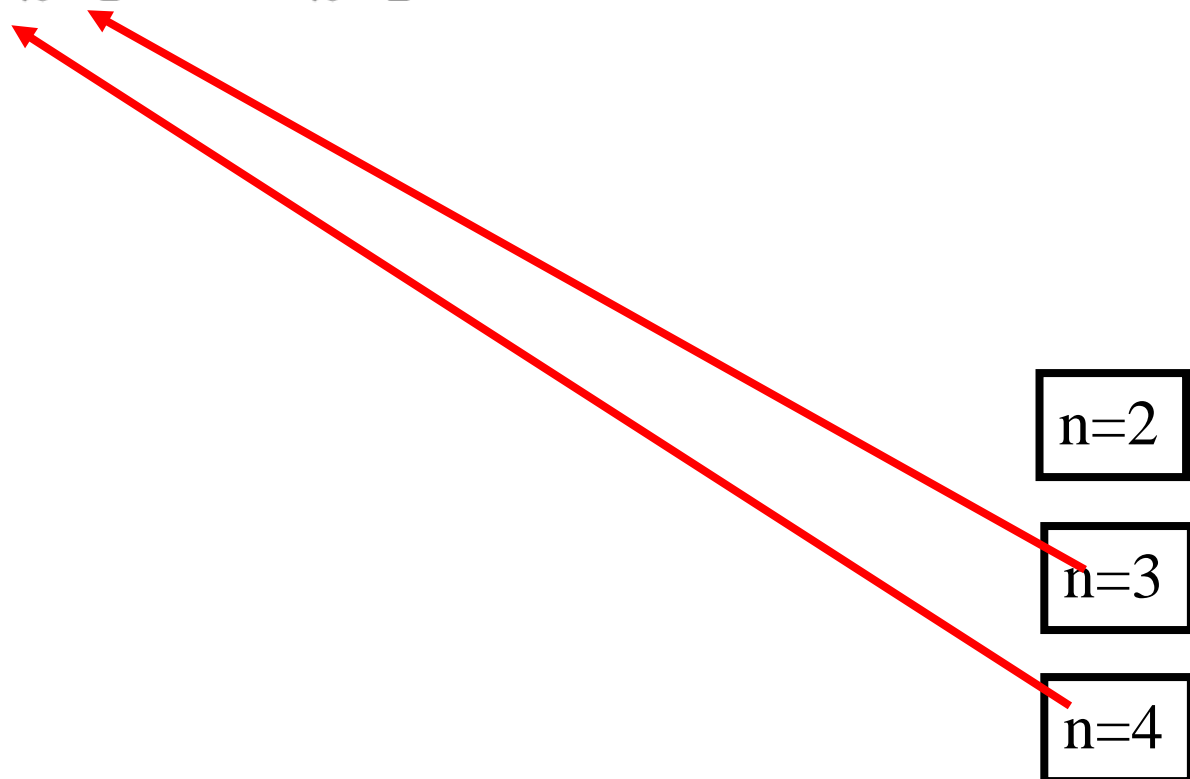
Recursion is handled the same way!

$$fib_n = \begin{cases} \rightarrow 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$

n=2

n=3

n=4



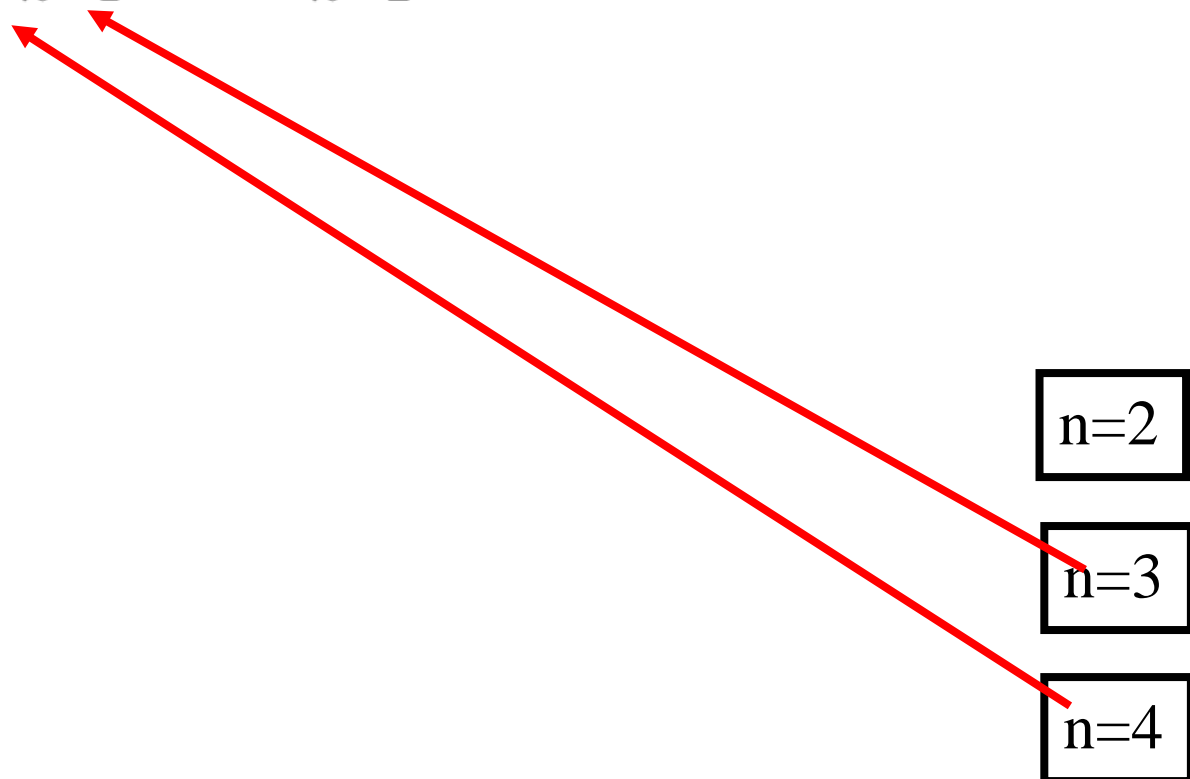
Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ \rightarrow 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$

n=2

n=3

n=4



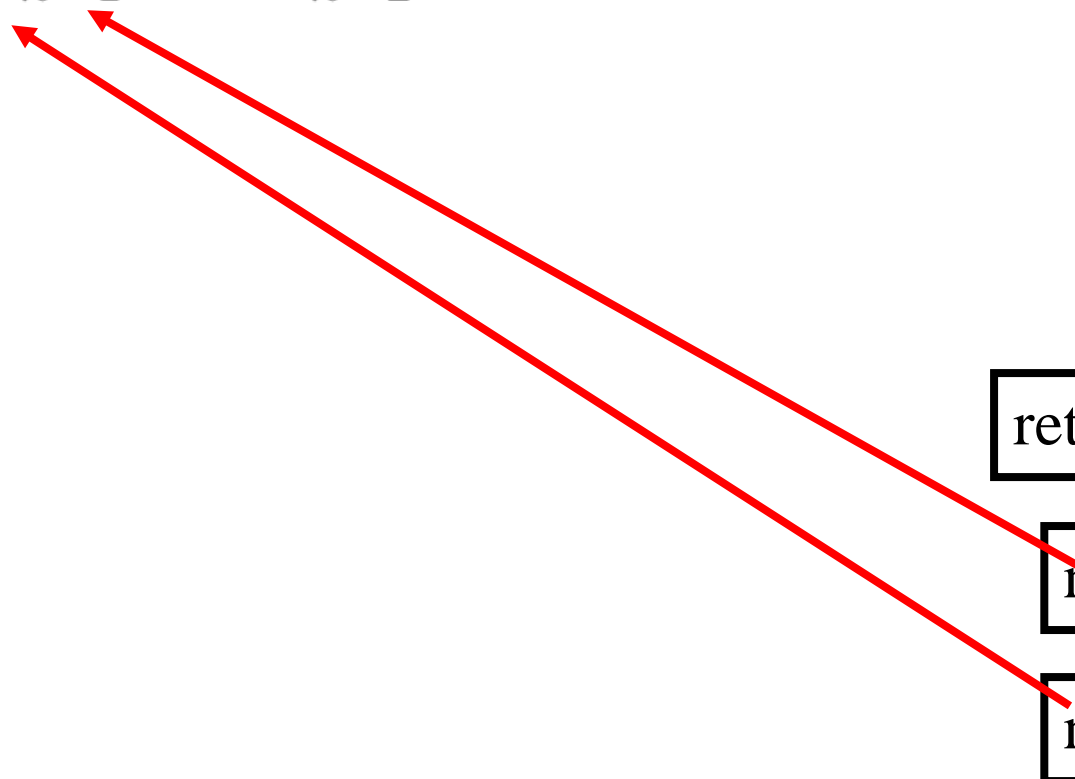
Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ \rightarrow 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$

return 1

n=3

n=4



Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$

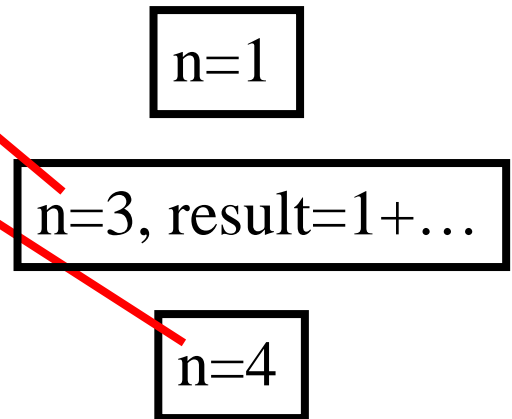


n=3, result=1+...

n=4

Recursion is handled the same way!

$$fib_n = \begin{cases} \rightarrow 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$



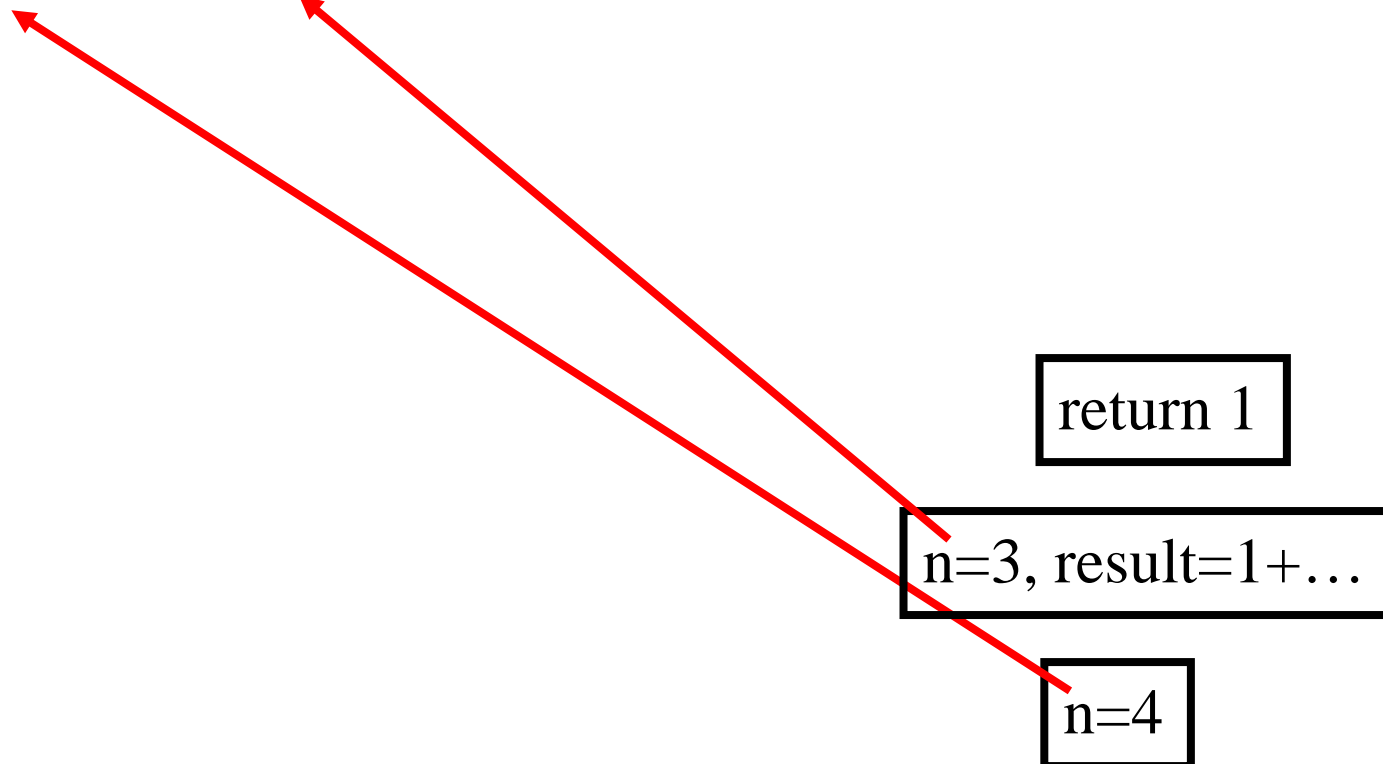
Recursion is handled the same way!

$$fib_n = \begin{cases} \rightarrow 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$

return 1

n=3, result=1+...

n=4



Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$



n=3, result=1+1

n=4

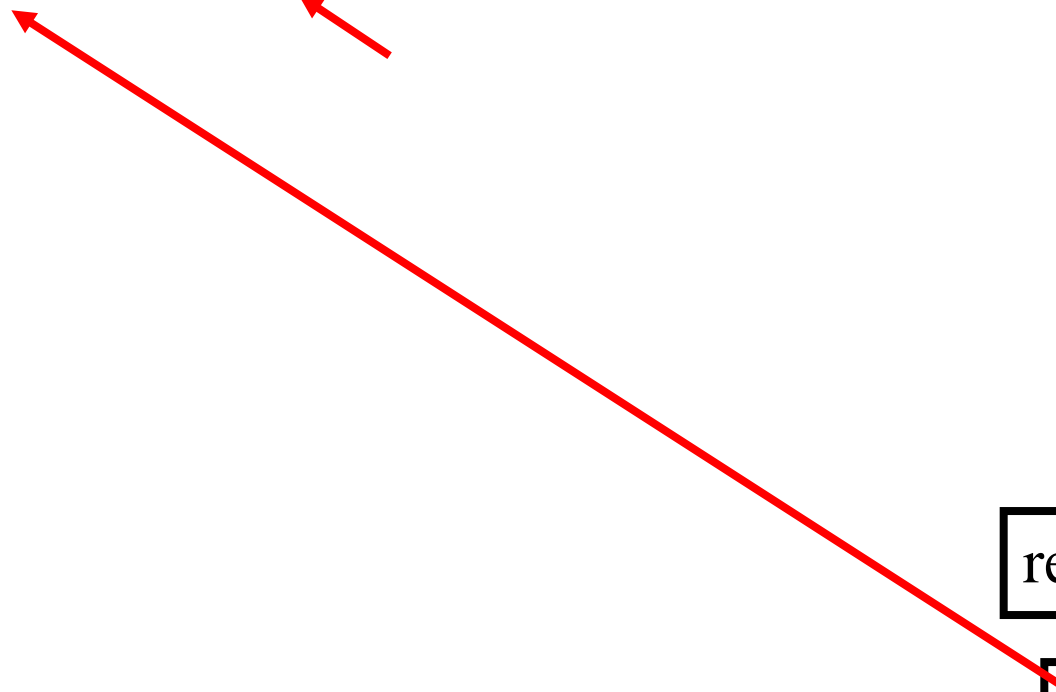
Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$




return 2

n=4




Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$


n=4, result=2+...

Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$


n=4, result=2+...

Recursion is handled the same way!

$$fib_n = \begin{cases} \rightarrow 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$

n=2

n=4, result=2+...


Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$

n=2

n=4, result=2+...

Recursion is handled the same way!


$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$


return 1

n=4, result=2+...




Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$


n=4, result=2+1

Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$


return 3

Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & if\ n = 1 \\ 1 & if\ n = 2 \\ fib_{n-1} + fib_{n-2} & if\ n \geq 3 \end{cases}$$

As I said before, do NOT try to think about recursion this way!

Recursion is handled the same way!

$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

As I said before, do NOT try to think about recursion this way!

However, by seeing what the computer does, we can see what takes time and space:

Each call takes time. We will try to avoid wasted calls.
The max depth of the call stack is the max space,
because each activation record takes $O(1)$ space.

Aside: Activation Records and Computer Security

- Have you heard about “buffer overrun” attacks?
- Suppose, when talking to your buddy, he manages to make you forget what you were in the middle of doing before his call?
- Suppose a function messes up the return address in the call stack?


Aside: Computer Security

$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

n=2

n=4, result=2+...

Aside: Computer Security

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$



Evil attacker code:
install backdoor
install rootkit
install Sony DRM software
...

n=2

n=4, result=2+...



Aside: Computer Security

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$


Evil attacker code:
install backdoor
install rootkit
install Sony DRM software
...

n=2

n=4, result=2+...



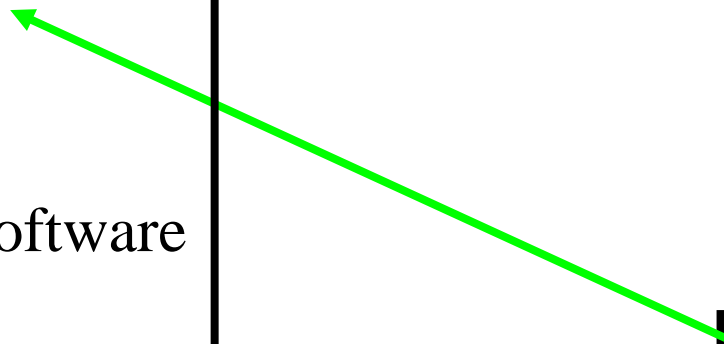
Aside: Computer Security

$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

Evil attacker code:
install backdoor
install rootkit
install Sony DRM software
...


return 1

n=4, result=2+...



Aside: Computer Security

$$fib_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ fib_{n-1} + fib_{n-2} & \text{if } n \geq 3 \end{cases}$$



Evil attacker code:
install backdoor
install rootkit
install Sony DRM software
...

n=4, result=2+...

Limits of the Call Stack

```
int fib(int n) {  
    if (n == 1)        return 1;  
    else if (n == 2)  return 1;  
    else                return fib(n-1) + fib(n-2);  
}  
cout << fib(0) << endl;
```

What will happen?

- a. Returns 1 immediately.
- b. Runs forever (infinite recursion)
- c. Stops running when n “wraps around” to positive values.
- d. Bombs when the computer runs out of stack space.
- e. None of these.

Function Calls in Daily Life

- How do you handle interruptions in daily life?
 - You're at home, working on CPSC221 project.
 - You stop to look up something in the book.
 - Your roommate/spouse/partner/parent/etc. asks for your help moving some stuff.
 - Your buddy calls.
 - The doorbell rings.

Tail Calls in Daily Life

- How do you handle interruptions in daily life?
 - You're at home, working on CPSC221 project.
 - You stop to look up something in the book.
 - Your roommate/spouse/partner/parent/etc. asks for your help moving some stuff.
 - Your buddy calls.
 - The doorbell rings.
- If new task happens just as you finish previous task, there's no need for new activation record.
- These are called **tail** calls.

Why Tail Calls Matter

- Since a tail call doesn't need to generate a new activation record on the stack, a good compiler won't make the computer do that.
- Therefore, a tail call doesn't increase depth of call stack.
- Therefore, the program uses less space if you can set it up to use a tail call.

Managing the Call Stack: Tail Recursion

```
void endlesslyGreet()  
{  
    cout << "Hello, world!" << endl;  
    endlesslyGreet();  
}
```

This is clearly infinite recursion. The call stack will get as deep as it can get and then bomb, right?

But... why? What *work* is the call stack doing?

There's *nothing* to remember on the stack!

Try compiling it with at least `-O2` optimization and running.
It won't give a stack overflow!

Tail Recursion

A function is “tail recursive” if for every recursive call in the function, that call is the absolute last thing the function needs to do before returning.

In that case, why bother pushing a new stack frame? There’s nothing to remember. Just re-use the old frame.

That’s what most compilers will do.

Tail Recursion

A function is “tail recursive” if for every recursive call in the function, that call is the absolute last thing the function needs to do before returning.

In that case, why bother pushing a new stack frame? There’s nothing to remember. Just re-use the old frame.

That’s what most compilers will do.

Note: KW textbook is WRONG on definition of tail recursion! They say it’s based on the last line, and the example they give is NOT tail recursive!

Tail Recursive?

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

Tail recursive?

- a. Yes.
- b. No.
- c. Not enough information.

Tail Recursive?

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

Tail recursive?

- a. Yes.
- b. No.
- c. Not enough information.

Tail Recursive?

```
int fact(int n) { return fact_acc(n, 1); }

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

Tail recursive?

- a. Yes.
- b. No.
- c. Not enough information.

Mythbusters: Recursion vs. Iteration

Which one can *do* more? Recursion or iteration?

MythBusters: Simulating a Loop with Recursion

```
int i = 0
while (i < n)
    doFoo(i)
    i++
```

```
recDoFoo(0, n)
```

Where `recDoFoo` is:

```
void recDoFoo(int i, int n)
{
    if (i < n) {
        doFoo(i)
        recDoFoo(i + 1, n)
    }
}
```

Anything we can do with iteration, we can do with recursion. 81

Mythbusters: Recursion vs. Iteration

Which one can *do* more? Recursion or iteration?

So, since iteration is just a special case of recursion (when it's tail recursive), recursion can do more.

But...

Mythbusters: Recursion vs. Iteration

Which one can *do* more? Recursion or iteration?

So, since iteration is just a special case of recursion (when it's tail recursive), recursion can do more.

But... If you have a stack (or can implement one somehow), **iteration with a stack** can do anything recursion can!

Mythbusters: Recursion vs. Iteration

Which one can *do* more? Recursion or iteration?

So, since iteration is just a special case of recursion (when it's tail recursive), recursion can do more.

But... If you have a stack (or can implement one somehow), **iteration with a stack** can do anything recursion can!

- (Aside: If you are developing a new computational paradigm, e.g., with DNA, being able to simulate a stack is a key building block.)

Mythbusters: Recursion vs. Iteration

Which one can *do* more? Recursion or iteration?

So, since iteration is just a special case of recursion (when it's tail recursive), recursion can do more.

But... If you have a stack (or can implement one somehow), **iteration with a stack** can do anything recursion can!

- This can be a little tricky.
- Better to let the computer do it for you!

Simulating Recursion with a Stack

- What does a recursive call do?
 - **Saves** current values of local variables and where execution is in the code.
 - Assigns parameters their passed in value.
 - Starts executing at start of function again.
- What does a return do?
 - Goes back to **most recent** call.
 - Restores **most recent** values of variables.
 - Gives return value back to caller.
- We can do on a stack what the computer does for us on the system stack...

Simulating Recursion with a Stack

- Cut the function at each call or return, into little pieces of code. Give each piece a name.
- Create a variable `pc`, which will hold the name of the piece of code to run.
- Put all the pieces in a big loop. At the top of the loop, choose which piece to run based on `pc`.
- At each recursive call, push local variables, push name of code to run after return, push arguments, set `pc` to `Start`.
- At `Start`, pop function arguments.
- At other labels, pop return value, pop local variables.
- At return, pop “return address” into `pc`, push return value.

This is not something we expect you to do in full generality in CPSC 221.

Simulating Recursion with a Stack

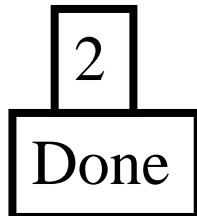
```
int factorial (int n) {
    if (n == 0) return 1;
    else
        return n *
            factorial(n - 1);
}
```

```
push(Done); push(n); pc=Start;
while (1) {
    if (pc==Done) break;
    if (pc==Start) {
        n=pop();
        if (n == 0) {
            pc=pop(); push 1; continue;
        } else {
            push(n); //save old n
            push(Middle);push(n-1);pc=Start;
            continue;
        }
    } else { //pc==Middle
        result=pop(); oldn=pop();
        result=oldn*result;
        pc=pop(); push(result);
    }
} // result is on top of stack
```

Anything we can do with recursion,
we can do with iteration w/ a stack.

Simulating Recursion with a Stack

```
→ int factorial (int n) {  
    if (n == 0) return 1;  
    else  
        return n *  
            factorial(n - 1);  
}  
  
→ push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```



Simulating Recursion with a Stack

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

n=2

Done

```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
→    if (n == 0) {  
        pc=pop(); push 1; continue;  
    } else {  
        push(n); //save old n  
        push(Middle);push(n-1);pc=Start;  
        continue;  
    }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

n=2

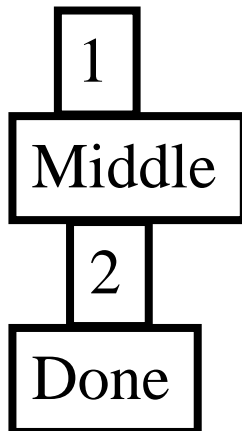
Done

```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

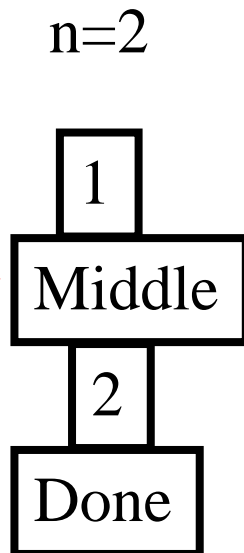
n=2



```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

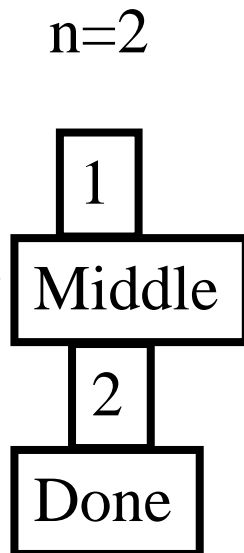


```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

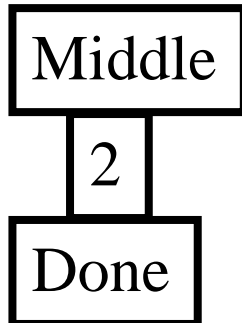
```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```



Simulating Recursion with a Stack

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

n=1

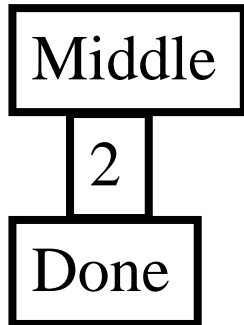


```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

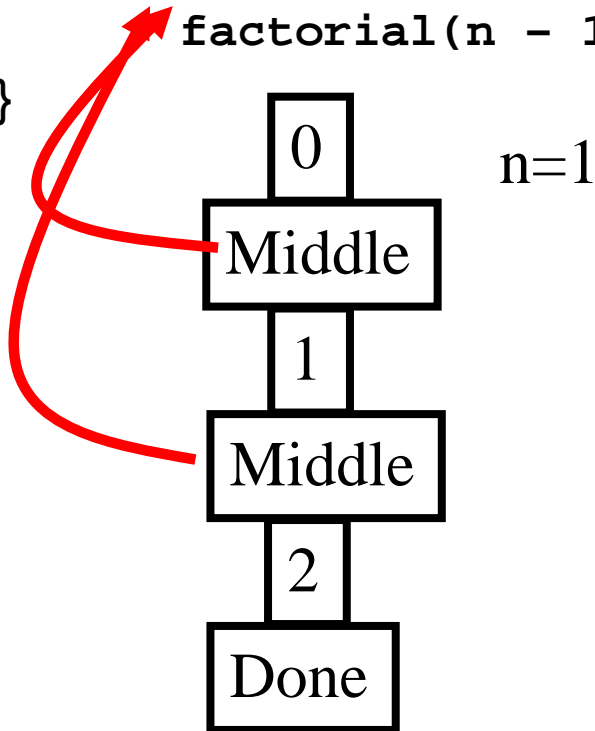
n=1



```
push(Done); push(n); pc=Start;  
while (1) {  
  if (pc==Done) break;  
  if (pc==Start) {  
    n=pop();  
    if (n == 0) {  
      pc=pop(); push 1; continue;  
    } else {  
      push(n); //save old n  
      push(Middle);push(n-1);pc=Start;  
      continue;  
    }  
  } else { //pc==Middle  
    result=pop(); oldn=pop();  
    result=oldn*result;  
    pc=pop(); push(result);  
  }  
} // result is on top of stack
```


Simulating Recursion with a Stack

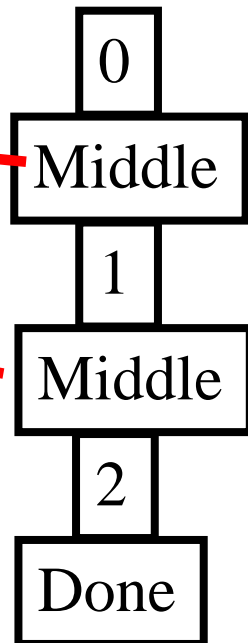
```
int factorial (int n) {  
    if (n == 0) return 1;  
    else  
        return n *  
        factorial(n - 1);  
}
```



```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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



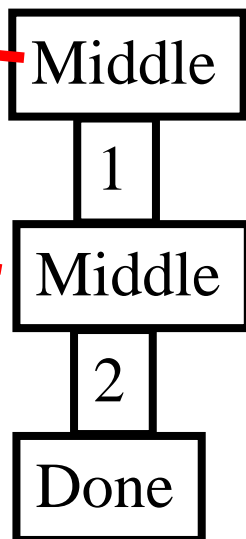
n=1

```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

n=0

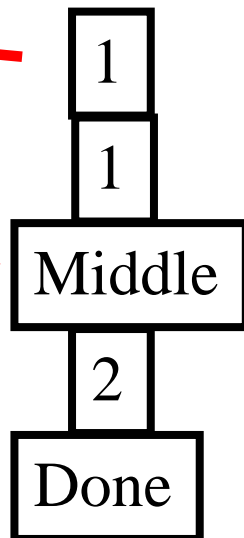


```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

n=0, pc=Middle

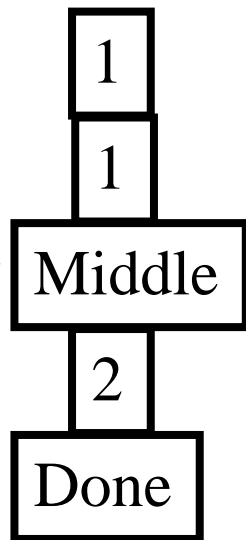


```
push(Done); push(n); pc=Start;  
while (1) {  
  if (pc==Done) break;  
  if (pc==Start) {  
    n=pop();  
    if (n == 0) {  
      pc=pop(); push 1; continue;  
    } else {  
      push(n); //save old n  
      push(Middle);push(n-1);pc=Start;  
      continue;  
    }  
  } else { //pc==Middle  
    result=pop(); oldn=pop();  
    result=oldn*result;  
    pc=pop(); push(result);  
  }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

n=0, pc=Middle

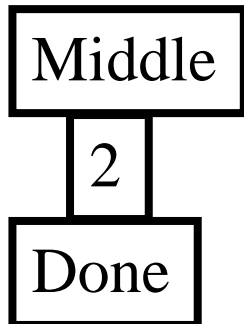


```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

result=1, oldn=1,
n=0, pc=Middle

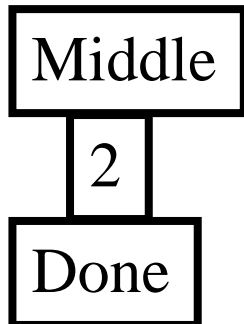


```
push(Done); push(n); pc=Start;  
while (1) {  
  if (pc==Done) break;  
  if (pc==Start) {  
    n=pop();  
    if (n == 0) {  
      pc=pop(); push 1; continue;  
    } else {  
      push(n); //save old n  
      push(Middle);push(n-1);pc=Start;  
      continue;  
    }  
  } else { //pc==Middle  
    result=pop(); oldn=pop();  
    result=oldn*result;  
    pc=pop(); push(result);  
  }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

result=1, oldn=1,
n=0, pc=Middle

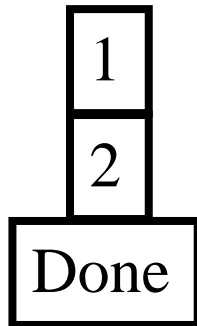


```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

result=1, oldn=1,
n=0, pc=Middle

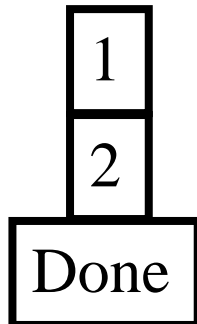


```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```


Simulating Recursion with a Stack

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

result=1, oldn=1,
n=0, pc=Middle



```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        → result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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

result=1, oldn=2,
n=0, pc=Middle

Done

```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        → result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack

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


result=2, oldn=2,
n=0, pc=Middle

Done

```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        → result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Simulating Recursion with a Stack


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



result=2, oldn=2,
n=0, pc=Done

2

```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```



Simulating Recursion with a Stack

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

result=2, oldn=2,
n=0, pc=Done

```
    push(Done); push(n); pc=Start;  
    while (1) {  
        → if (pc==Done) break;  
        if (pc==Start) {  
            n=pop();  
            if (n == 0) {  
                pc=pop(); push 1; continue;  
            } else {  
                push(n); //save old n  
                push(Middle);push(n-1);pc=Start;  
                continue;  
            }  
        } else { //pc==Middle  
            result=pop(); oldn=pop();  
            result=oldn*result;  
            pc=pop(); push(result);  
        }  
    } // result is on top of stack
```

Simulating Recursion with a Stack

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

result=2, oldn=2,
n=0, pc=Done

This is not something we expect
you to do in full generality in
CPSC 221.

2

```
push(Done); push(n); pc=Start;  
while (1) {  
    if (pc==Done) break;  
    if (pc==Start) {  
        n=pop();  
        if (n == 0) {  
            pc=pop(); push 1; continue;  
        } else {  
            push(n); //save old n  
            push(Middle);push(n-1);pc=Start;  
            continue;  
        }  
    } else { //pc==Middle  
        result=pop(); oldn=pop();  
        result=oldn*result;  
        pc=pop(); push(result);  
    }  
} // result is on top of stack
```

Steve's Fib Example

Computer handles recursion on the stack.

Sometimes you can see a clever shortcut to do it a bit more efficiently by only storing what's really needed on the stack:

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

We will prove that
Steve's program
works next time.

OK, this is cheating a bit (in a good way).
To get down and dirty, see continuations in CPSC 311.

Simulating Recursion with a Stack

- What does a recursive call do?
 - **Saves** current values of local variables and where execution is in the code.
 - Assigns parameters their passed in value.
 - Starts executing at start of function again.
- What does a return do?
 - Goes back to **most recent** call.
 - Restores **most recent** values of variables.
 - Gives return value back to caller.
- We can do on a stack what the computer does for us on the system stack...

Simulating Tail Recursion w/o Stack

- What does a recursive call do?
 - Saves current values of local variables and where execution is in the code.
 - Assigns parameters their passed in value.
 - Starts executing at start of function again.
- What does a return do?
 - Goes back to **most recent** call.
 - Restores **most recent** values of variables.
 - Gives return value back to caller.
- Why use a stack if you don't have to do any saving or restoring???

Tail Recursion into Iteration

```
int fact(int n) {  
    return fact_acc(n, 1);  
}
```

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

Tail Recursion into Iteration – Step 1

```
int fact(int n) {  
    return fact_acc(n, 1);  
}
```

```
int fact_acc (int n, int acc) {  
    if (n == 0) return acc;  
    else {  
        //return fact_acc(n - 1, acc * n);  
        acc = acc * n;  
        n = n-1;  
    }  
}
```

Assign parameters
their passed-in values

Tail Recursion into Iteration – Step 1

```
int fact(int n) {  
    return fact_acc(n, 1);  
}  
  
int fact_acc (int n, int acc) {  
  
    if (n == 0) return acc;  
    else {  
        //return fact_acc(n - 1, acc * n);  
        acc = acc * n;  
        n = n-1;  
    }  
  
}
```

Assign parameters
their passed-in values

Tail Recursion into Iteration – Step 2

```
int fact(int n) {  
    return fact_acc(n, 1);  
}
```

```
int fact_acc (int n, int acc) {  
    while (1) {  
        if (n == 0) return acc;  
        else {  
            //return fact_acc(n - 1, acc * n);  
            acc = acc * n;  
            n = n-1;  
        }  
    }  
}
```

Start executing at
beginning of function.

Tail Recursion into Iteration – Step 3

```
int fact(int n) {  
    return fact_acc(n, 1);  
}
```

```
int fact_acc (int n, int acc) {  
    while (n != 0) {  
        //if (n == 0) return acc;  
        //else {  
            //return fact_acc(n - 1, acc * n);  
            acc = acc * n;  
            n = n-1;  
        //}  
    }  
    return acc;  
}
```

Clean up your code
to look nicer.

Tail Recursion into Iteration – Step 3

```
int fact(int n) {  
    return fact_acc(n, 1);  
}
```

```
int fact_acc (int n, int acc) {  
    while (n != 0) {  
        acc = acc * n;  
        n = n-1;  
    }  
    return acc;  
}
```

Clean up your code
to look nicer.

Tail Recursion into Iteration

```
int fact(int n) {  
    return fact_acc(n, 1);  
}
```

```
int fact_acc (int n, int acc) {  
    while (n != 0) {  
        acc = acc * n;  
        n = n-1;  
    }  
    return acc;  
}
```

For 221, you should be able to look at a simple tail-recursive function and convert it to be iterative.

Today's Learning Goals

- See the similarity between a recursive function and a proof by induction.
- Prove recursive functions correct using induction.
- Prove loops correct using loop invariants.
- Appreciate how a proof can help you understand complicated code.

- (If we have time, use memoization to make recursive functions run faster.)

Induction and Recursion, Twins Separated at Birth?

Base case

Prove for some small
value(s).

Inductive Step

Otherwise, break a larger
case down into smaller
ones that we assume work
(the Induction Hypothesis).

Base case

Calculate for some small
value(s).

Recursion

Otherwise, break the problem
down in terms of itself
(smaller versions) and
then call this function to
solve the smaller versions,
assuming it will work.

Old Slide: Thinking Recursively

This is the secret to thinking recursively!

Your solution will work as long as:

- (1) you've broken down the problem right
- (2) each recursive call really is simpler/smaller, and
- (3) you make sure all calls will eventually hit base case(s).

As soon as you break the problem down in terms of **any simpler version**, call the function recursively and **assume it works**. Do **not** think about how!

Thinking Inductively

This is also the secret to doing a proof by induction!

Your solution will work as long as:

- (1) you've broken down the problem right
- (2) inductive assumption on cases that really are simpler/smaller,
- (3) you make sure you've covered all base case(s).

As soon as you break the problem down in terms of **any simpler version**, use the inductive hypothesis and **assume it works**. Do **not** think about how!

Induction and Recursion

- They even have the same pitfalls!
- When is it hard to do a proof by induction?
- When is it hard to solve a problem with recursion?

Induction and Recursion

- They even have the same pitfalls!
- When is it hard to do a proof by induction?
 - When you can't figure out how to break the problem down
 - When you miss a base case
- When is it hard to solve a problem with recursion?
 - When you can't figure out how to break the problem down
 - When you miss a base case

Proving a Recursive Function Correct with Induction is **EASY**

Just follow your code's lead and use induction.

Your base case(s)? **Your code's base case(s).**

How do you break down the inductive step? **However your code breaks the problem down into smaller cases.**

What do you assume? **That the recursive calls just work (for smaller input sizes as parameters, which better be how your recursive code works!).**

Proving a Recursive Function Correct with Induction is **EASY**

```
// Precondition: n >= 0.  
// Postcondition: returns n!  
int factorial(int n)  
{  
    if (n == 0)  
        return 1;  
  
    else  
        return n*factorial(n-1);  
}
```

Prove: `factorial(n) = n!`

Base case: $n = 0$.

Our code returns 1 when $n = 0$, and $0! = 1$ by definition.
✓

Inductive step: For any $k > 0$, our code returns $k * \text{factorial}(k-1)$. By IH, $\text{factorial}(k-1) = (k-1)!$ and $k! = k * (k-1)!$ by definition. **QED**

Perfect Card Shuffling

Problem: You have an array of n playing cards. You want to shuffle them so that every order is equally likely. You may use a function **randrange(n)**, which selects a number $[0, n)$ uniformly at random.

Proving A Recursive Algorithm Works

Problem: Prove that our algorithm for card shuffling gives an equal chance of returning every possible shuffle (assuming **randrange(n)** works as advertised).

Recurrence Relations...

Already Covered

See METYCSSA #5-7.

Additional Problem: Prove binary search takes $O(\lg n)$ time.

```
// Search array[left..right] for target.
// Return its index or the index where it should go.
int bSearch(int array[], int target, int left, int right)
{
    if (right < left) return left;
    int mid = (left + right) / 2;
    if (target <= array[mid])
        return bSearch(array, target, left, mid-1);
    else
        return bSearch(array, target, mid+1, right);
}
```

Binary Search Problem (Worked)

Note: Let n be # of elements considered in the array ($\text{right} - \text{left} + 1$).

```
int bSearch(int array[], int target, int left, int right)
{
    if (right < left) return left;  O(1), base case
    int mid = (left + right) / 2;  O(1)
    if (target <= array[mid]) O(1)
        return bSearch(array, target, left, mid-1);  ~T(n/2)
    else
        return bSearch(array, target, mid+1, right);  ~T(n/2)
}
```

Binary Search Problem (Worked)

For $n=0$: $T(0) = 1$

For $n>0$: $T(n) = T(\lfloor n/2 \rfloor) + 1$

To guess at the answer, we simplify:

Change $\lfloor n/2 \rfloor$ to $n/2$.

Change base case to $T(1)$

(We'll never reach 0 by dividing by 2!)

For $n=1$: $T(1) = 1$

For $n>1$: $T(n) = T(n/2) + 1$ Sub in $T(n/2) = T(n/4)+1$

$T(n) = (T(n/4) + 1) + 1$

$T(n) = T(n/4) + 2$ Sub in $T(n/4) = T(n/8)+1$

$T(n) = T(n/8) + 3$ Sub in $T(n/8) = T(n/16)+1$

$T(n) = T(n/16) + 4$

$T(n) = T(n/(2^i)) + i$

Binary Search Problem (Worked)

To guess at the answer, we simplify:

$$\text{For } n=1: T(1) = 1$$

$$\text{For } n>1: T(n) = T(n/2) + 1$$

$$\text{For } n>1: T(n) = T(n/(2^i)) + i$$

To reach the base case, let $n/2^i = 1$

$$n = 2^i \text{ means } i = \lg n$$

Why did that work out so well?

$$T(n) = T(n/2^{\lg n}) + \lg n = T(1) + \lg n = \lg n + 1$$

$$T(n) \in O(\lg n)$$

Binary Search Problem (Worked)

To **prove** the answer, we use induction:

For $n=0$: $T(0) = 1$

For $n>0$: $T(n) = T(\lfloor n/2 \rfloor) + 1$

$T(1) = T(0) + 1 = 2$

$T(2) = T(3) = T(1) + 1 = 3.$

Prove $T(n) \in O(\lg n)$

Let $c = 3, n_0 = 2.$

Base cases: $T(2) = 3 = 3 \lg 2 \checkmark$

Base cases: $T(3) = 3 \leq 3 \lg 3 \checkmark$

Binary Search Problem (Worked)

To **prove** the answer, we use induction:

For $n=0$: $T(0) = 1$

For $n>0$: $T(n) = T(\lfloor n/2 \rfloor) + 1$

$T(1) = T(0) + 1 = 2$

$T(2) = T(3) = T(1) + 1 = 3$.

Prove $T(n) \in O(\lg n)$

Let $c = 3, n_0 = 2$.

Base cases: $T(2) = 3 = 3 \lg 2 \checkmark$

Base cases: $T(3) = 3 \leq 3 \lg 3 \checkmark$

Alan's Aside: Note that Steve used 2 and 3 as base cases. Why? Because proof doesn't work at $T(1)$.

Binary Search Problem (Worked)

$$T(0) = 1, T(1) = 2, T(2) = 3, T(3) = 3$$

$$\text{For } n > 3: T(n) = T(\lfloor n/2 \rfloor) + 1$$

$$c = 3, n_0 = 2$$

Base cases: prev slides ✓

Induction hyp: for all $2 \leq k < n$, $T(k) \leq 3 \lg k$

Inductive step, $n > 3$, in two cases (**odd** & even)

$$n \text{ is odd: } T(n) = T((n-1)/2) + 1$$

$$\leq 3 \lg((n-1)/2) + 1$$

$$= 3 \lg(n-1) - 3 \lg 2 + 1$$

$$= 3 \lg(n-1) - 3 + 1$$

$$\leq 3 \lg n \quad \checkmark$$

$n \geq 5$, so
 $(n-1)/2 \geq 2$,
so IH applies

Binary Search Problem (Worked)

$$T(0) = 1, T(1) = 2, T(2) = 3, T(3) = 3$$

$$\text{For } n > 3: T(n) = T(\lfloor n/2 \rfloor) + 1$$

$$c = 3, n_0 = 2$$

Base cases: prev slides ✓

Induction hyp: for all $2 \leq k < n$, $T(k) \leq 3 \lg k$

Inductive step, $n > 3$, in two cases (odd & **even**)

$$n \text{ is even: } T(n) = T(n/2) + 1$$

$$\leq 3 \lg(n/2) + 1$$

$$= 3 \lg n - 3 \lg 2 + 1$$

$$= 3 \lg n - 3 + 1$$

$$\leq 3 \lg n \quad \checkmark$$

$n \geq 4$, so
 $n/2 \geq 2$,
so IH applies

QED!

Proof of Iterative Programs?

- We've seen that iteration is just a special case of recursion.
- Therefore, we should be able to prove that loops work, using the same general technique.
- Because loops are a special case (and are easier to analyze, so the theory was developed earlier), there is different terminology, but it's still induction.

Loop Invariants

We do this by stating and proving “invariants”, properties that are always true (don’t vary) at particular points in the program.

One way of thinking of a loop is that at the start of each iteration, the invariant holds, but then the loop *breaks* it as it computes, and then spends the rest of the iteration fixing it up.

Compare to the simplest induction you learned, where you assume the case for n and prove for $n+1$. Now, we assume a statement is true before each loop iteration, and prove it is still true after the loop iteration.

Caution!

- The description of loop invariants in the Epp textbook is slightly wrong and needlessly confusing:
 - The invariant doesn't need to be a predicate whose domain is only an integer. Any predicate will work.
 - It confuses the variables in the predicate with the number of times the loop executes.
 - It mixes up (1) proving that a predicate is a loop invariant with (2) using the loop invariant to show that a program works.
 - Termination should be handled separately from the reasoning about loop invariants.
- If you want a written reference, the Wikipedia page for “Loop Invariant” is correct.

Invariants in Daily Life

Suppose you have a bunch of house guests who are all well-behaved, so they always put things that they use back the way they found them.

- When you leave the house, you have put everything just the way you like (toilet seat position, books on the table, milk in the fridge, etc.)
- Where are they after your guests leave?
- Does it matter how many guests were there, or how often they used your stuff?

More Interesting Examples

- When the police search for a fugitive, they:
 1. establish a perimeter that contains the suspect,
 2. maintain the invariant “The suspect is within the search perimeter.” as they gradually shrink the perimeter.
- The same approach is used for fighting wildfires:
 1. establish a perimeter that contains all burning areas,
 2. maintain the invariant “All burning areas are within the perimeter.” as they gradually shrink the perimeter.

The approach works regardless of how long it takes.

More Interesting Examples

- When the police search for a fugitive, they:
 1. establish a perimeter that contains the suspect,
 2. maintain the invariant “The suspect is within the search perimeter.” as they gradually shrink the perimeter.
- The same approach is used for fighting wildfires:
 1. establish a perimeter that contains all burning areas,
 2. maintain the invariant “All burning areas are within the perimeter.” as they gradually shrink the perimeter.

The approach works regardless of how long it takes.

Do you see the induction happening I these examples?

Loop Invariants: The Easy Way

- Convert for-loops to while-loops.
 - Easiest to reason about while-loops.

```
int i=1; // etc. initialization stuff
while (condition) {
    loop body;
}
```

Loop Invariants: The Easy Way

- Write your loop invariant to be true at the exact same time as you check the loop condition.
 - In a while-loop, this is at the top/bottom of the loop body.
 - No need to worry about the `i++` in a for-loop

```
int i=1; // etc. initialization stuff
while (condition) {
    loop body;
}
```

Loop Invariants: The Easy Way

- Base Case: prove that your loop invariant holds when you first arrive at the loop.

```
int i=1; // etc. initialization stuff
while (condition) {
    loop body;
}
```

Loop Invariants: The Easy Way

- Induction:
 - Assume the loop invariant holds at top of loop.
 - You also get to assume the loop condition is true. (Why?)
 - Prove that loop invariant holds at bottom of loop.

```
int i=1; // etc. initialization stuff
while (condition) {
    loop body;
}
```

Loop Invariants: The Easy Way

- Finishing the proof:
 - Upon exiting the loop, you can still assume loop invariant.
 - You also get to assume the loop condition is false.
 - Use those fact to prove whatever you need next.

```
int i=1; // etc. initialization stuff
while (condition) {
    loop body;
}
```

Loop Invariants: The Easy Way

- Termination:
 - You may need to make a completely separate argument that the loop will eventually terminate.
 - Usually, this is by showing that some progress is always made each time you go through the loop.

```
int i=1; // etc. initialization stuff
while (condition) {
    loop body;
}
```

Insertion Sort

```
for (int i = 1; i < length; i++)
{
    int val = array[i];
    int newIndex = bSearch(array, val, 0, i);
    for (int j = i; j > newIndex; j--)
        array[j] = array[j-1];
    array[newIndex] = val;
}
```

Rewrite as while loop!

Insertion Sort

```
int i = 1;
while (i < length)
{
    int val = array[i];
    int newIndex = bSearch(array, val, 0, i);
    for (int j = i; j > newIndex; j--)
        array[j] = array[j-1];
    array[newIndex] = val;
    i++;
}
```

Now, we need to come up with a good invariant.

Insertion Sort

```
int i = 1;
while (i < length)
    // Invariant: here (and at loop bottom), the elements in
    // array[0..i-1] are in sorted order.
    {
        // since i will go up by 1, put the last element in order!
        int val = array[i];
        int newIndex = bSearch(array, val, 0, i);
        for (int j = i; j > newIndex; j--)
            array[j] = array[j-1];
        array[newIndex] = val;
        i++;
    }
```

So, what's the base case?

Insertion Sort

```
int i = 1;
while (i < length)
    // Invariant: here (and at loop bottom), the elements in
    // array[0..i-1] are in sorted order.
    {
        // since i will go up by 1, put the last element in order!
        int val = array[i];
        int newIndex = bSearch(array, val, 0, i);
        for (int j = i; j > newIndex; j--)
            array[j] = array[j-1];
        array[newIndex] = val;
        i++;
    }
```

Base Case: When the code first reaches the loop invariant $i=1$, so $\text{array}[0..0]$ is trivially sorted.

Insertion Sort

```
int i = 1;
while (i < length)
    // Invariant: here (and at loop bottom), the elements in
    // array[0..i-1] are in sorted order.
    {
        // since i will go up by 1, put the last element in order!
        int val = array[i];
        int newIndex = bSearch(array, val, 0, i);
        for (int j = i; j > newIndex; j--)
            array[j] = array[j-1];
        array[newIndex] = val;
        i++;
    }
```

Proof of inductive case is just like before.

Insertion Sort

```
int i = 1;
while (i < length)
    // Invariant: here (and at loop bottom), the elements in
    // array[0..i-1] are in sorted order.
    {
        // since i will go up by 1, put the last element in order!
        int val = array[i];
        int newIndex = bSearch(array, val, 0, i);
        for (int j = i; j > newIndex; j--)
            array[j] = array[j-1];
        array[newIndex] = val;
        i++;
    }
```

Inductive Hypothesis: We assume $\text{array}[0..i-1]$ is sorted at top of loop, and $i < \text{length}$.

Insertion Sort

```
int i = 1;
while (i < length)
    // Invariant: here (and at loop bottom), the elements in
    // array[0..i-1] are in sorted order.
    {
        // since i will go up by 1, put the last element in order!
        int val = array[i];
        int newIndex = bSearch(array, val, 0, i);
        for (int j = i; j > newIndex; j--)
            array[j] = array[j-1];
        array[newIndex] = val;
        i++;
    }
```

Inductive Step: bSearch finds correct index to put array[i], so array[0..i] is sorted, then i++ happens...¹⁵⁷.

Insertion Sort

```
int i = 1;
while (i < length)
    // Invariant: here (and at loop bottom), the elements in
    // array[0..i-1] are in sorted order.
    {
        // since i will go up by 1, put the last element in order!
        int val = array[i];
        int newIndex = bSearch(array, val, 0, i);
        for (int j = i; j > newIndex; j--)
            array[j] = array[j-1];
        array[newIndex] = val;
        i++;
    }
```

Inductive Step: ... so loop invariant holds again at the bottom of the loop. QED

Insertion Sort

```
int i = 1;
while (i < length)
    // Invariant: here (and at loop bottom), the elements in
    // array[0..i-1] are in sorted order.
{
    // since i will go up by 1, put the last element in order!
    int val = array[i];
    int newIndex = bSearch(array, val, 0, i);
    for (int j = i; j > newIndex; j--)
        array[j] = array[j-1];
    array[newIndex] = val;
    i++;
}
```

When loop exits, $i == \text{length}$. Invariant says $\text{array}[0..i-1]$ is sorted, so $\text{array}[0..\text{length}-1]$ is sorted.

Loop Invariants: The Easy Way

BTW, this “Easy Way” is at least as formal, precise, and correct as any method where you see lots of math flying around (like in the Epp textbook).

It’s also the basis for tools like Microsoft’s Static Driver Verifier.

It’s also how Bob Floyd and Tony Hoare originally formalized this.

Aside: Formality vs. Sloppiness

- In real life, people are often a bit sloppy, just to make things easier. That's OK if you know what you're doing. When in doubt, fall back on the formal approach!
 - If you're very comfortable with for loops, you don't have to rewrite as a while loop.
 - Getting all the details can be tricky, but the core idea of your loop invariant is a GREAT comment to put in your code.

Aside: Formality vs. Looking Formal

- If you ever have to deal with a professor who thinks that a loop invariant needs to have an induction variable (which is incorrect, but not everyone knows this), just follow these steps:
 - Say “Let k (or i or some other convenient mathy variable name) represent the number of times the loop body executes. The proof is by mathematical induction on k .” at the beginning of your proof.
 - At the beginning of your base case, say “In the base case, $k=0$. When the execution first reaches the top of the loop body...” and then fill in the same base case you would have said doing things the easy way.
 - For your induction step, say “We assume the loop invariant holds after k executions of the loop body” at the beginning of the induction step. Prove that it holds at the end of the loop body, and then say, “So we see that the loop invariant still holds after $k+1$ executions of the loop body. This concludes the proof by mathematical induction.”

Steve's Practice: Prove the Inner Loop Correct

```
for (int i = 1; i < length; i++)
{
    // i went up by 1.  The last element may be out of order!
    int val = array[i];
    int newIndex = bSearch(array, val, 0, i);
    // What's the invariant?  Something like
    // "array[0..j-1] + array[j+1..i] = the old array[0..i-1]"
    for (int j = i; j > newIndex; j--)
        array[j] = array[j-1];
    array[newIndex] = val;
}
```

Prove by induction that the inner loop operates correctly.
(This may feel unrealistically easy!)

Finish the proof! (As we did for the outer loop, talk about what the invariant means when the loop ends.)

Steve's Practice (Solution): Prove the Inner Loop Correct

```
// What's the invariant?  Something like
// "array[0..j-1] + array[j+1..i] = the old array[0..i-1]"
for (int j = i; j > newIndex; j--)
    array[j] = array[j-1];
```

Base Case: At the start of the first iteration, $j=i$, so $\text{array}[0..j-1]$ is exactly array the old $\text{array}[0..i-1]$.

Inductive Step: Assume the invariant holds at the top of the loop. The invariant doesn't care about $\text{array}[j]$, so we can overwrite it with $\text{array}[j-1]$. But after $j--$, the invariant holds once again for the new j .

When the loop terminates, $j==\text{newIndex}$. Therefore, $\text{array}[0..\text{newIndex}-1] + \text{array}[\text{newIndex}+1..i]$ equals the old $\text{array}[0..i-1]$.

Steve's Fib Example

Computer handles recursion on the stack.

Sometimes you can see a clever shortcut to do it a bit more efficiently by only storing what's really needed on the stack:

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

We will prove that
Steve's program
works next time.

OK, this is cheating a bit (in a good way).
To get down and dirty, see continuations in CPSC 311.

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Where does the loop invariant go?

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

What should the invariant be?

Where does the loop invariant go?

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

What should the invariant be?

This is the step that requires insight...

Hmm... I'm replacing n by $n-1$ and $n-2$, or I'm increasing $result$ when $n \leq 2$ (when $fib(n)=1$).

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

What should the
invariant be?

This is the step that requires insight...

So, it's sort of like stuff on the stack, plus result doesn't change...

Aha! Sum of fib(i) for all i on stack, plus result equals fib(n)

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

OK, so now, what's the base case?

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

OK, so now, what's the base case?

Initially, n is only item on stack, and result=0. $\text{fib}(n)+0=\text{fib}(n)$.

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

Note that for a loop invariant proof,
the base case is NOT something like $n=0$.
The (implicit) induction variable is the
number of times through the loop!

OK, so now, what's the base case?

Initially, n is only item on stack, and result=0. $\text{fib}(n)+0=\text{fib}(n)$.

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

And the inductive case?

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

And the inductive case? Assume inductive hypothesis.

We pop a number n off the stack. If $n \leq 2$, then $\text{fib}(n)=1$, so by increasing result by 1, we maintain inductive hypothesis... 174

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

And the inductive case? Assume inductive hypothesis.
If $n > 2$, we push $n-1$ and $n-2$. But since $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
(by definition), the sum of fib(i) for all i on the stack is
unchanged.

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

```
  n = pop
  if n < 2
  el
  retu
```

At this point, the “loop invariant” proof itself is done!!! (You have proven by induction that the loop invariant always holds.)

The next step is to use the loop invariant to prove that the program works.

And the inductive case? Assume inductive hypothesis. If $n > 2$, we push $n-1$ and $n-2$. But since $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ (by definition), the sum of fib(i) for all i on the stack is unchanged.

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

What can we conclude at loop
exit?

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

Loop invariant holds, and
(not (not isEmpty))

We still have loop invariant (since it's invariant), and we have the exit condition: isEmpty.

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

Loop invariant holds, and
(not (not isEmpty))

Since stack is empty, sum of fib of stuff on stack is 0. So, $0 + \text{result} = \text{fib}(n)$. Therefore, $\text{result} = \text{fib}(n)$. QED

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Sum of fib(i) for all i on stack,
plus result equals fib(n)

Loop invariant holds, and
(not (not isEmpty))

The loop invariant helps us understand if/why the code works!
(BTW, loop invariants are great things to put in comments.)

Steve's Fib Example

```
int fib(int n)
  result = 0
  push(n)
  while not isEmpty
    n = pop
    if (n <= 2) result++;
    else push(n - 1); push(n - 2)
  return result
```

Termination for this example takes some work, too.

The key is that if you think of what's on the stack as a string of numbers, the stack contents always get earlier in “alphabetical order”. E.g., [5] > [4,3] > [4,2,1] > [4,2] > [4] > [3,2] > [3] > ... This way of ordering is called “lexicographical order”.

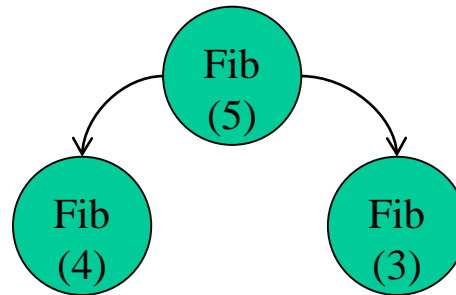
Topic Change: Memoization

- This is an easy-to-program trick to make certain kinds of recursive functions run a lot faster...

Accidentally Making Lots of Recursive Calls; Recall...

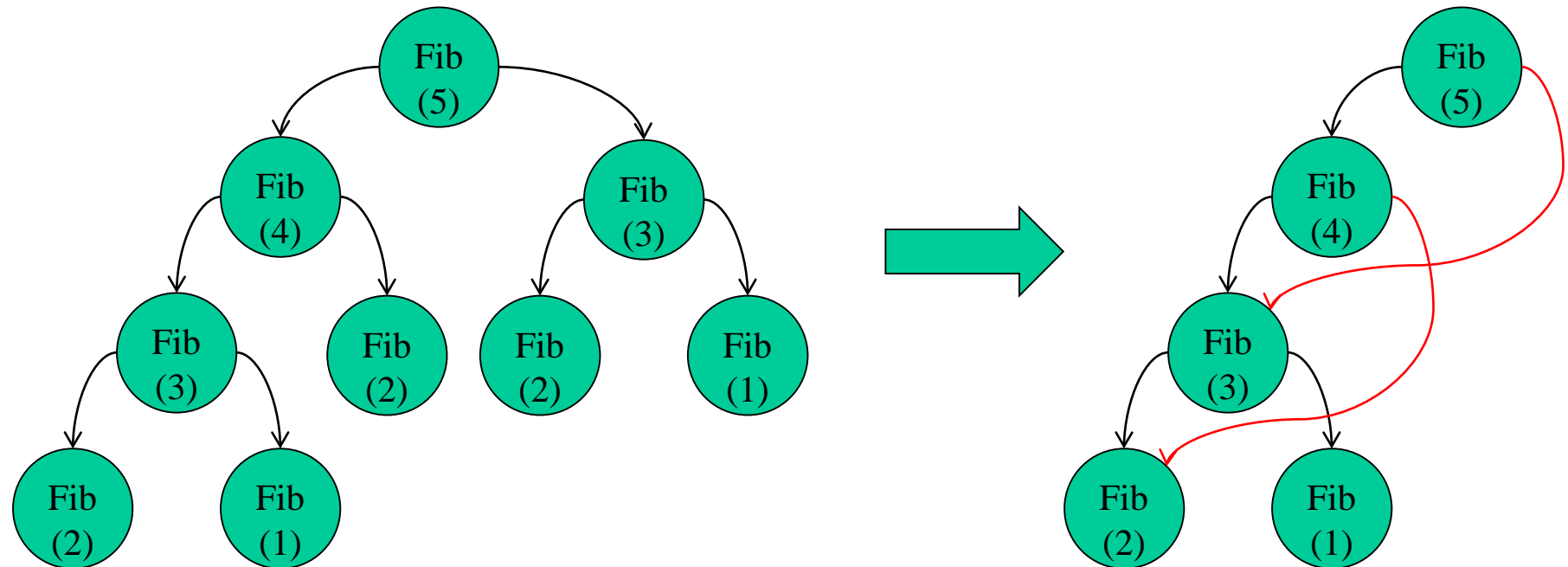
```
int Fib(n)
  if (n == 1 or n == 2) return 1
  else return Fib(n - 1) + Fib(n - 2)
```

Finish the recursion tree for Fib(5)...



Avoiding Duplicate Calls

We're making an exponential number of calls! This is bad.
Plus, many calls are duplicates... That means wasted work!

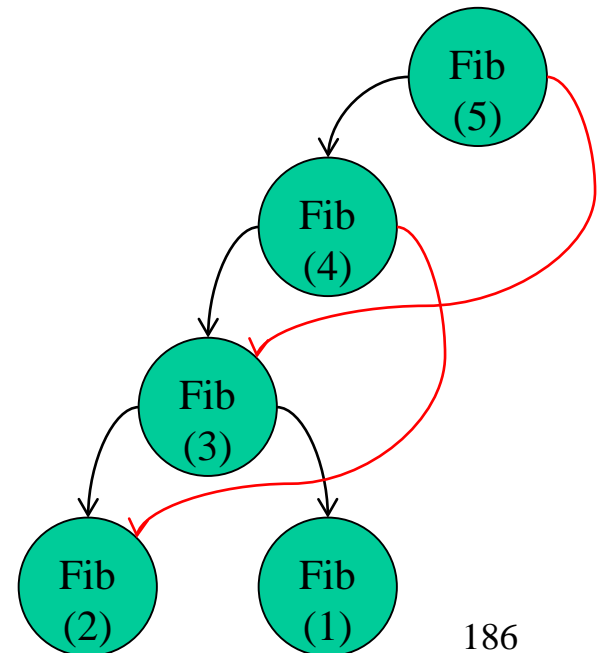


Memoization

- Keep a table of all calls you've computed already.
 - Initially, this is empty.
 - This trick only works if the number of possible calls is much smaller than the total number of **times** you make recursive calls.
- At start of function, check if you've solved this case before. If so, return old solution.
- After computing a solution, store it in table before returning. (Leave a “memo” to yourself.)

Fixing Fib with Recursion and “Memoizing”

```
int[] fib_solns = new int[large_enough]; // init to 0
fib_solns[1] = 1;
fib_solns[2] = 1;
int fib_memo(int n)
{
    // If we don't know the answer...
    if (fib_solns[n] == 0)
        fib_solns[n] = fib_memo(n-1) +
                       fib_memo(n-2);
    return fib_solns[n];
}
```



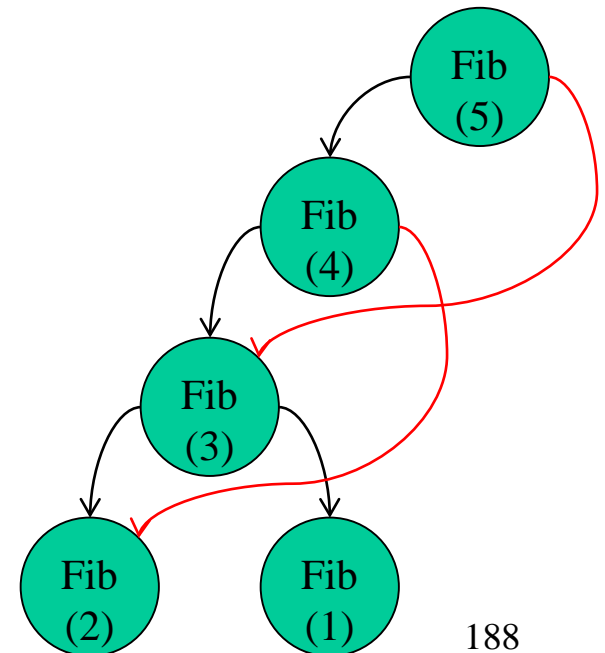
Aside: “Dynamic Programming”

- It turns out that you can often build up the table of solutions iteratively, from the base cases up, instead of using recursion.
- For historical reasons, this is called “dynamic programming”. You’ll see this a lot in CPSC 320.
- The advantage of dynamic programming is that once you see how the table is built up, you can often use much less space, keeping only the parts that matter.
- The advantage of memoization, though, is that it’s very easy to program.

Fixing Fib with “Dynamic Programming”

```
int[] fib_solns = new int[large_enough]; // init to 0
fib_solns[1] = 1;
fib_solns[2] = 1;
```

```
int fib(int n) {
    for (int i=3; i<=n; i++) {
        fib_solns[i] = fib_solns[i-1] +
                       fib_solns[i-2];
    }
    return fib_solns[n];
}
```



Fixing Fib with “Dynamic Programming” – Optimizing Space

```
int[] fib_solns = new int[2]; // init to 0
fib_solns[0] = 1;
fib_solns[1] = 1;
```

```
int fib(int n) {
    for (int i=3; i<=n; i++) {
        old_fib = fib_solns[0];
        fib_solns[0] = fib_solns[1];
        fib_solns[1] = fib_solns[0] +
            old_fib;
    }
    return fib_solns[1];
}
```

