CPSC 221
Basic Algorithms and Data Structures

Recursion and Iteration

Textbook References:
Koffman: Chapter 7
EPP 3rd edition: 5.1, 5.2 7.1, 7.2, 4.2, 4.3, 4.5
EPP 4th edition: 6.1, 6.2 7.1, 7.2, 5.2, 5.3, 5.5

Hassan Khosravi
January – April 2015

(Borrowing many slides from Alan Hu and Steve Wolfman)

Learning goals (Induction and Recursion)

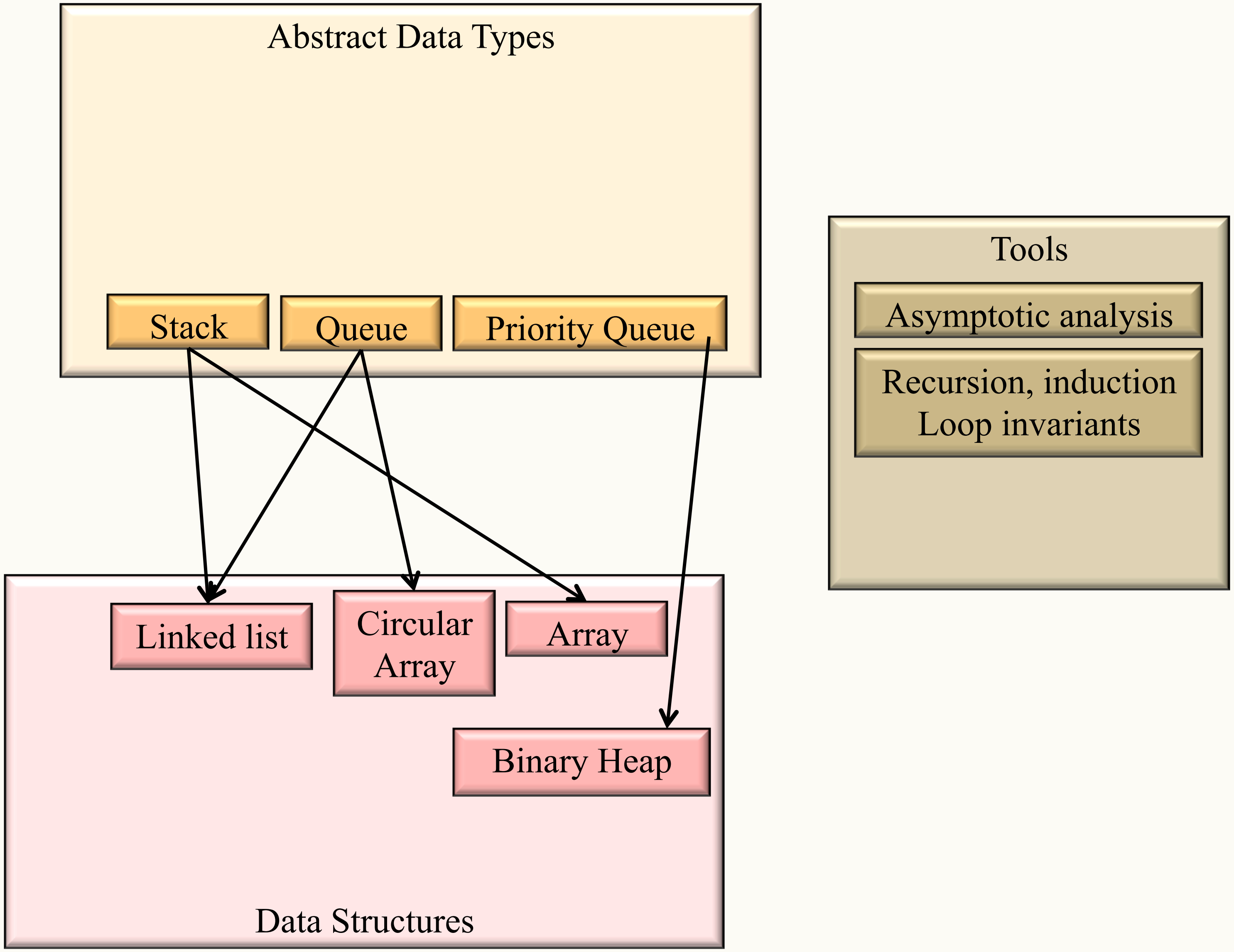
- Describe the relationship between recursion and induction (e.g., take a recursive code fragment and express it mathematically in order to prove its correctness inductively).
- Evaluate the effect of recursion on space complexity (e.g., explain why a recursively defined method takes more space then an equivalent iteratively defined method.).
- Describe how tail recursive algorithms can require less space complexity than non-tail recursive algorithms.
- Recognize algorithms as being iterative or recursive.
- Convert recursive solutions to iterative solutions and vice versa.
- Draw a recursion tree and relate the depth to a) the number of recursive calls and b) the size of the runtime stack. Identify and/or produce an example of infinite recursion

Learning goals (Loop Invariants)

- Take a loop code fragment and express it mathematically in order to prove its correctness inductively (specifically describing that the induction is on the iteration variable).

- In simpler cases, determine the loop invariant.

CPSC 221 Journey



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!

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 you help moving some stuff.
  - Your buddy calls.
  - The doorbell rings.

LIFO!
That's a stack!

- 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.

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=?

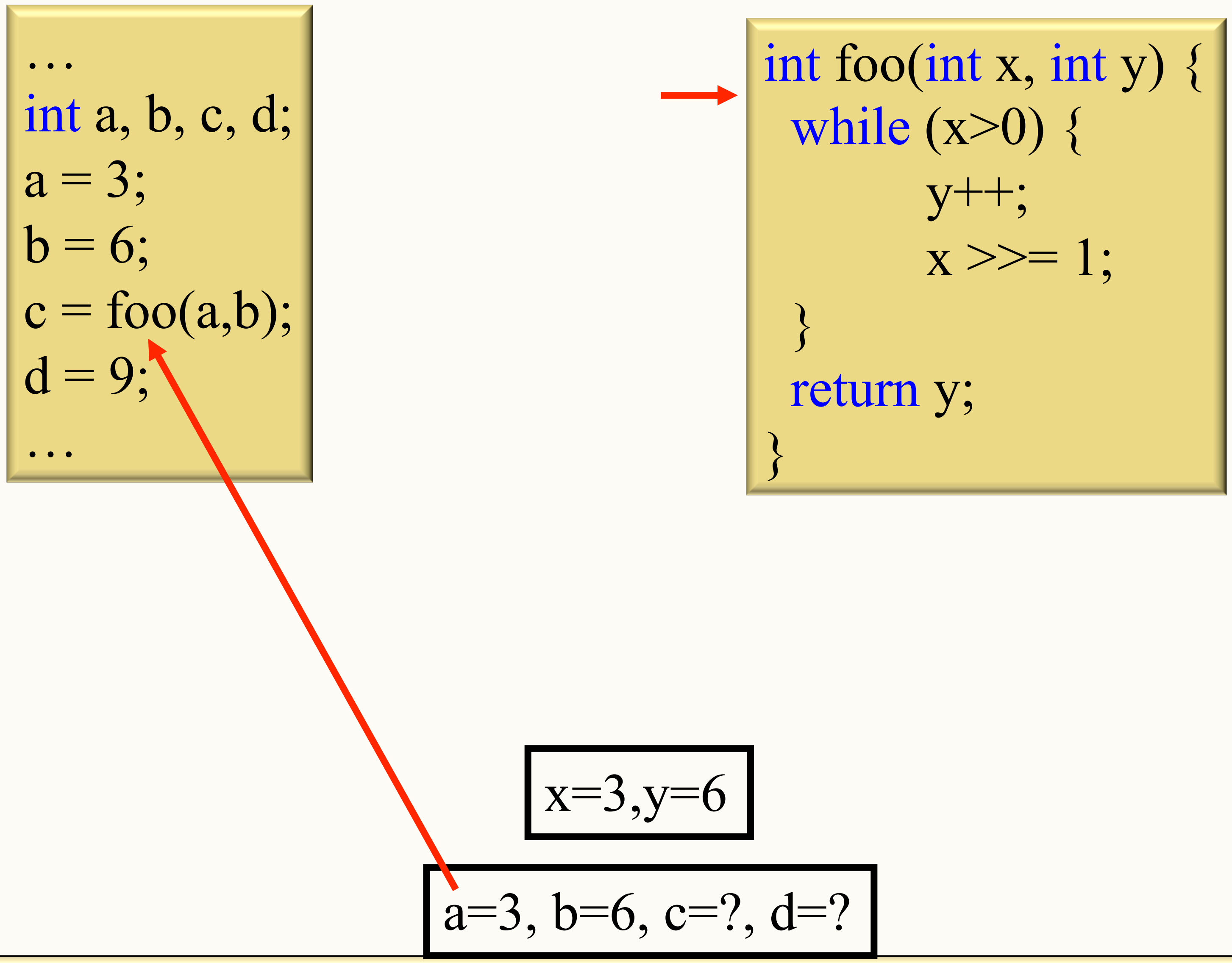
Activation Records on a Computer

```
…
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=?

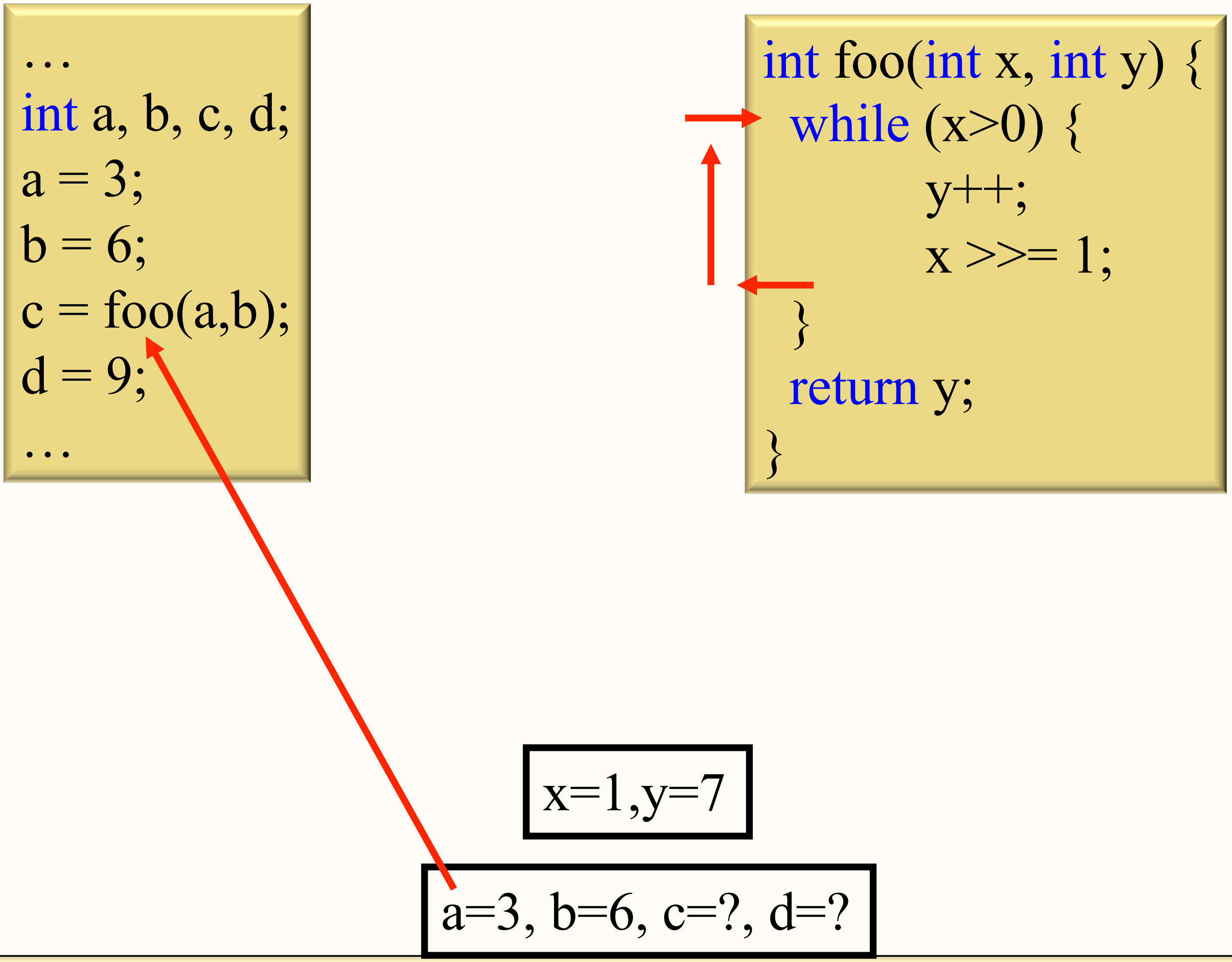
Activation Records on a Computer



```
…
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;
}
```

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;
}
```

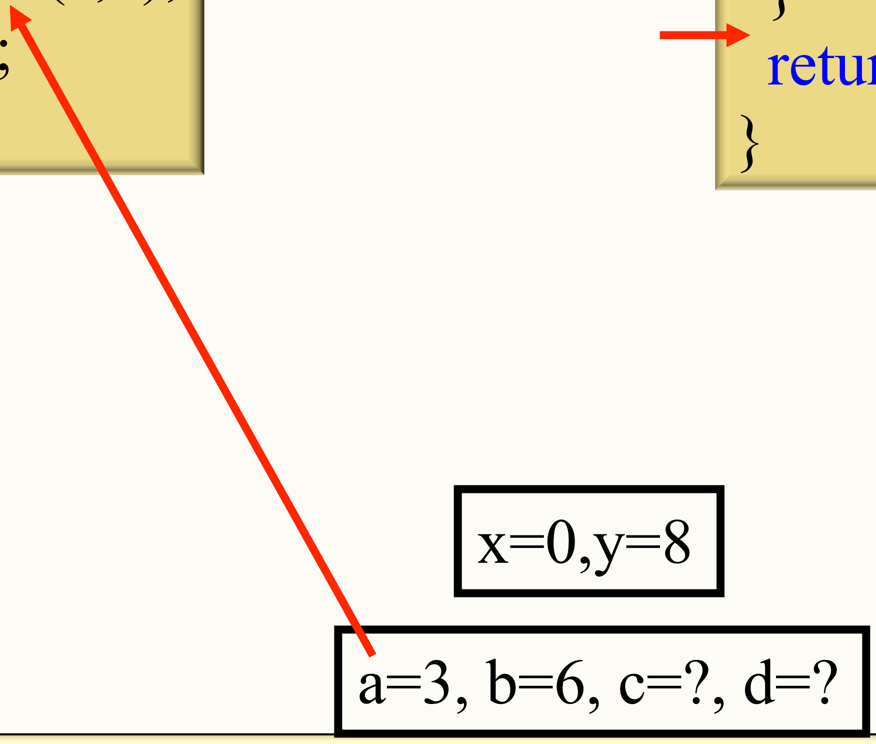
x=0,y=8

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;
…
```

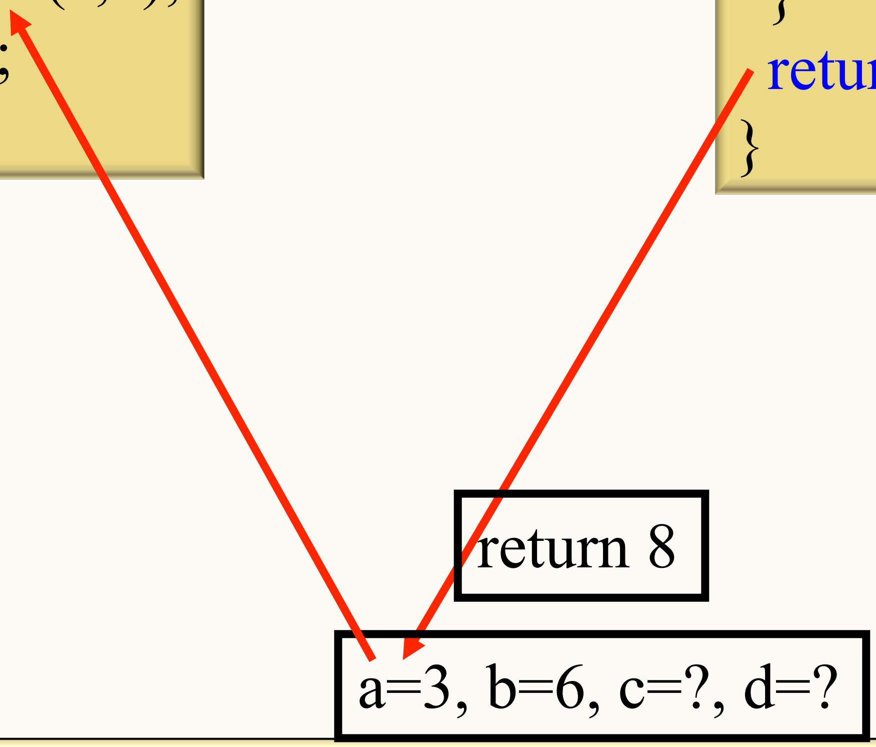
```
int foo(int x, int y) {
  while (x>0) {
        y++;
        x >>= 1;
  }
  return y;
}
```



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!

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

$\boxed{n=4}$

Recursion is handled the same way!

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

$\boxed{n=4}$

Recursion is handled the same way!

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

$\boxed{n=4}$

Recursion is handled the same way!

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

$\boxed{n=3}$

$\boxed{n=4}$

Recursion is handled the same way!

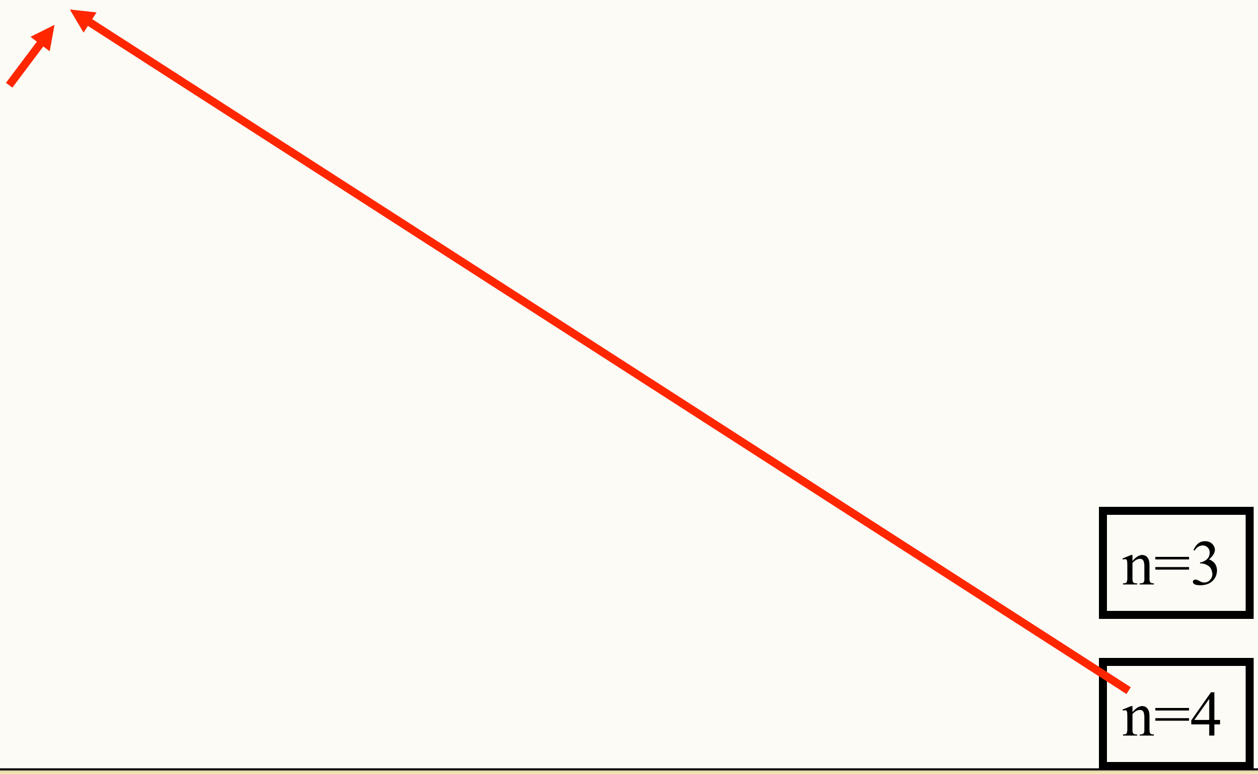
$$fib_n = \begin{cases} 1 & if n = 1 \\ \rightarrow 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

$\boxed{n=3}$

$\boxed{n=4}$

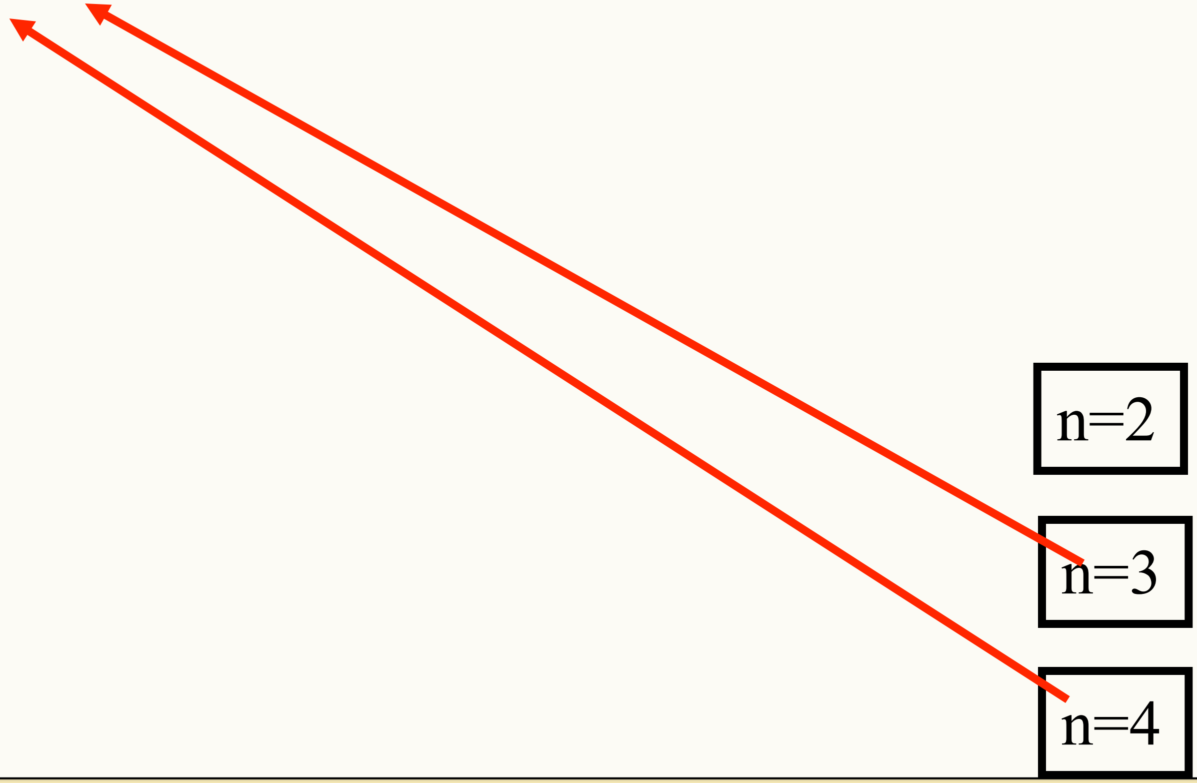
Recursion is handled the same way!

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



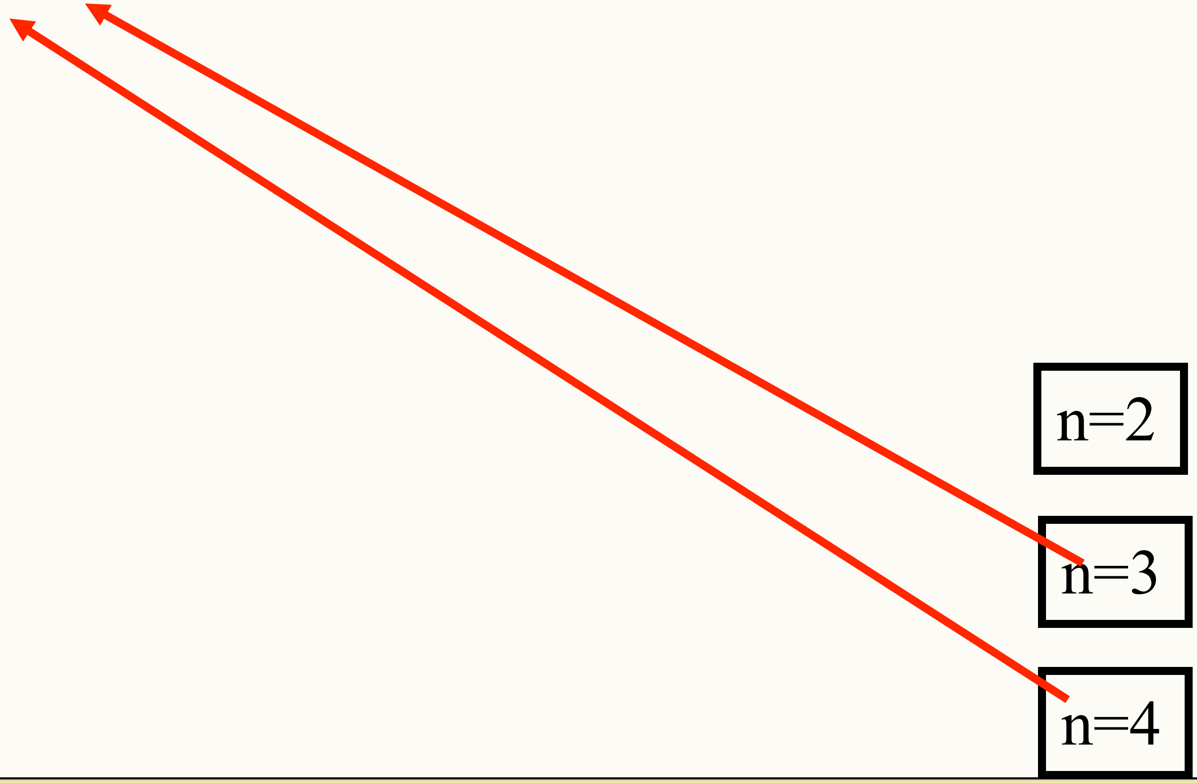
Recursion is handled the same way!

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



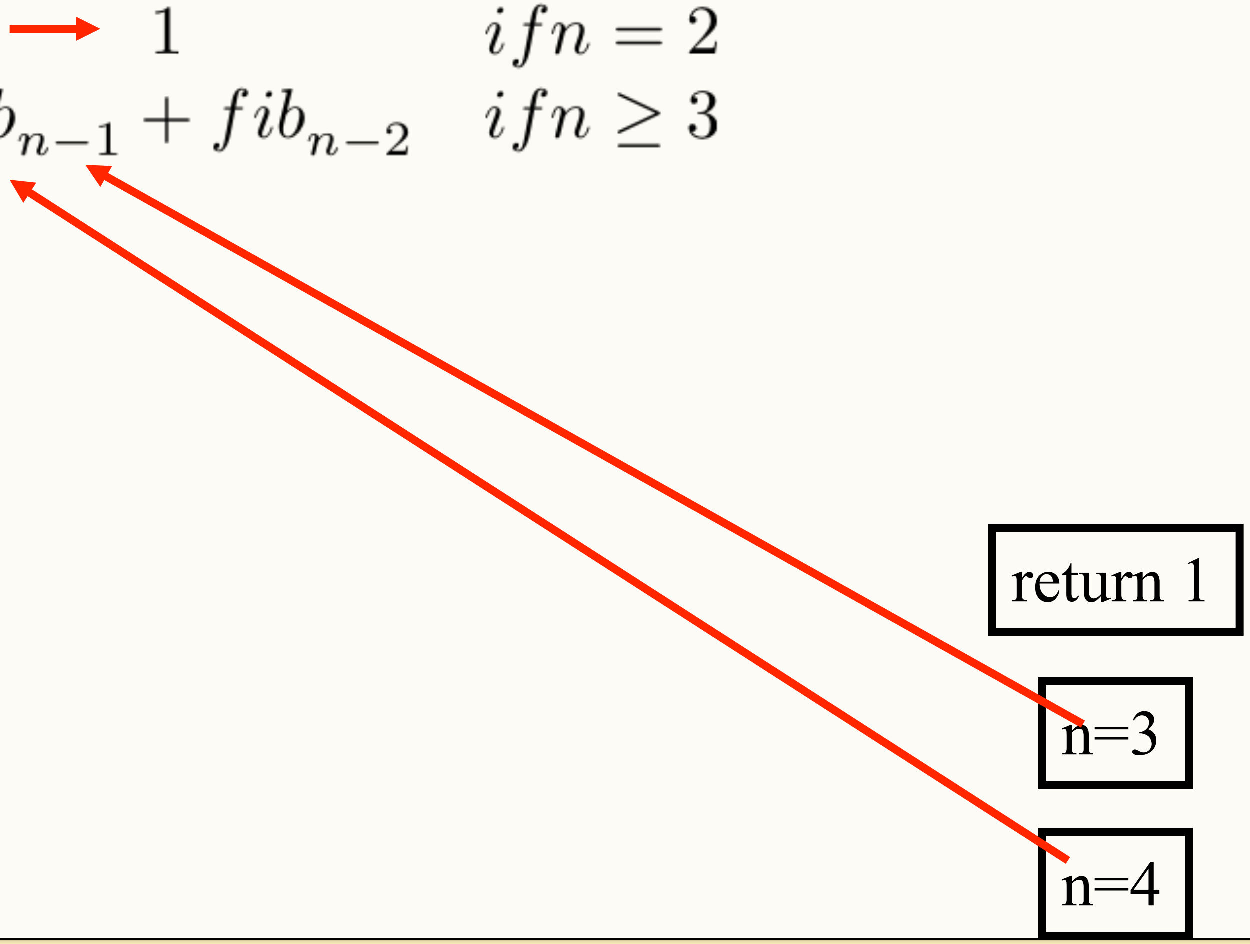
Recursion is handled the same way!

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



Recursion is handled the same way!

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



Recursion is handled the same way!

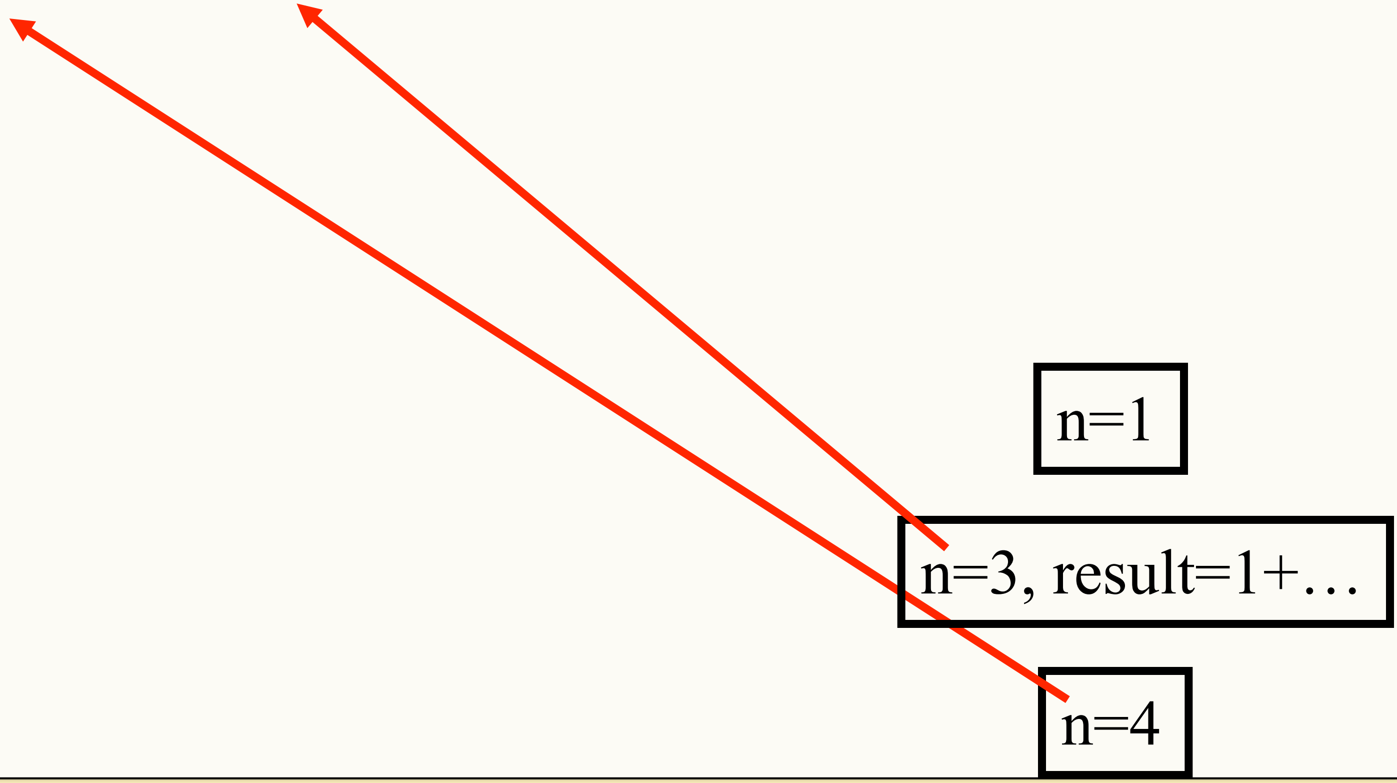
$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

$\boxed{n=3, \text{result}=1+\ldots}$

$\boxed{n=4}$

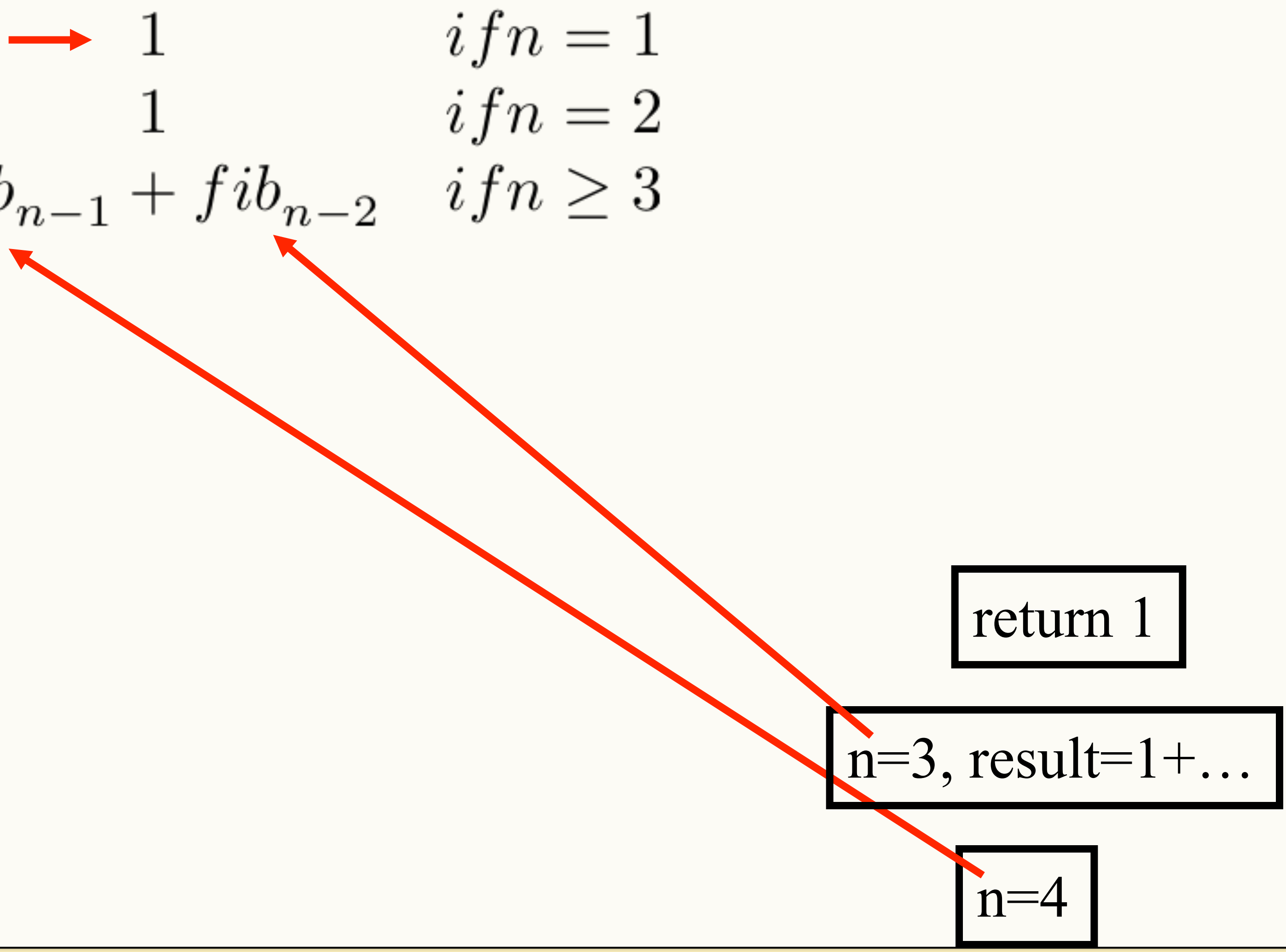
Recursion is handled the same way!

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



Recursion is handled the same way!

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



Recursion is handled the same way!

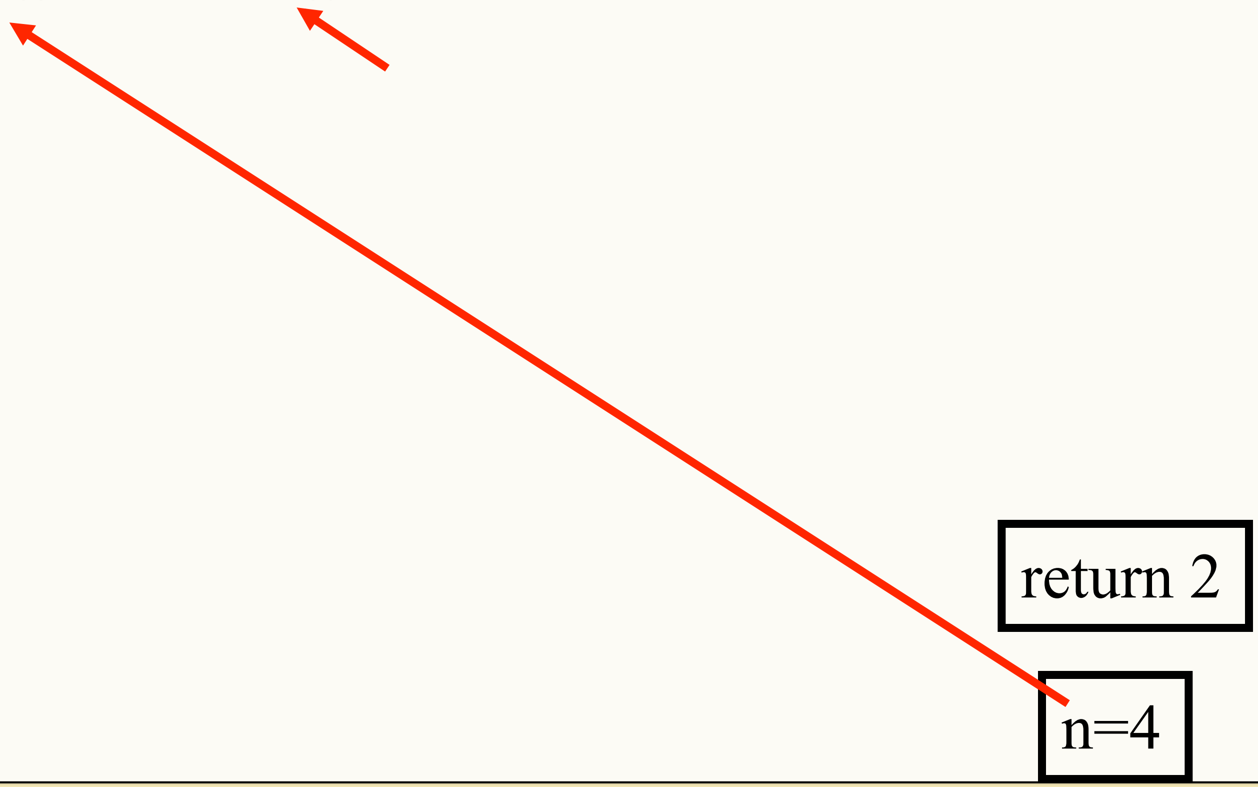
$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

$\boxed{\text{n=3, result=1+1}}$

$\boxed{\text{n=4}}$

Recursion is handled the same way!

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



Recursion is handled the same way!

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

$$\boxed{\text{n=4, result=2+…}}$$

Recursion is handled the same way!

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

$\boxed{\text{n=4, result=2+…}}$

Recursion is handled the same way!

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

$\boxed{n=2}$

$\boxed{n=4, result=2+\ldots}$

Recursion is handled the same way!

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

$\boxed{n=2}$

$\boxed{n=4, result=2+\ldots}$

Recursion is handled the same way!

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

$\boxed{\text{return 1}}$

$\boxed{\text{n=4, result=2+…}}$

Recursion is handled the same way!

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

$\nwarrow$

$\boxed{n=4, result=2+1}$

Recursion is handled the same way!

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

$\boxed{\text{return } 3}$

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!

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

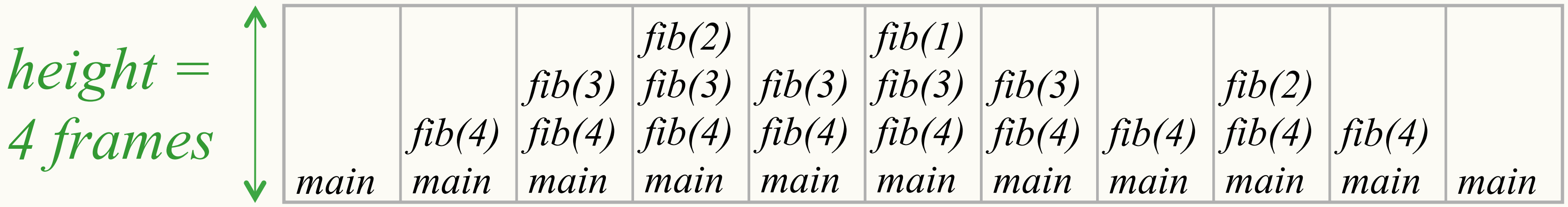
The call (or "run-time") stack

			fib(2)		*fib(1)*					
		fib(3)	*fib(3)*	*fib(3)*	*fib(3)*	*fib(3)*		*fib(2)*		
	fib(4)	*fib(4)*	*fib(4)*	*fib(4)*	*fib(4)*	*fib(4)*	*fib(4)*	*fib(4)*	*fib(4)*	
main	*main*	*main*	*main*	*main*	*main*	*main*	*main*	*main*	*main*	*main*

$$\longrightarrow$$

Efficiency and the Call Stack

- The *height* of the call stack tells us the maximum memory we use storing the stack.



- The number of calls that go through the call stack tells us something about time usage. (The # of calls multiplied by worst-case time per call bounds the asymptotic complexity.)

When calculating memory usage, we must consider stack space! But only the non-tail-calls count. see later slides on tail recursion and tail calls.

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

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

$\boxed{\text{n=2}}$

$\boxed{\text{n=4, result=2+…}}$

Aside: Computer Security

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

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

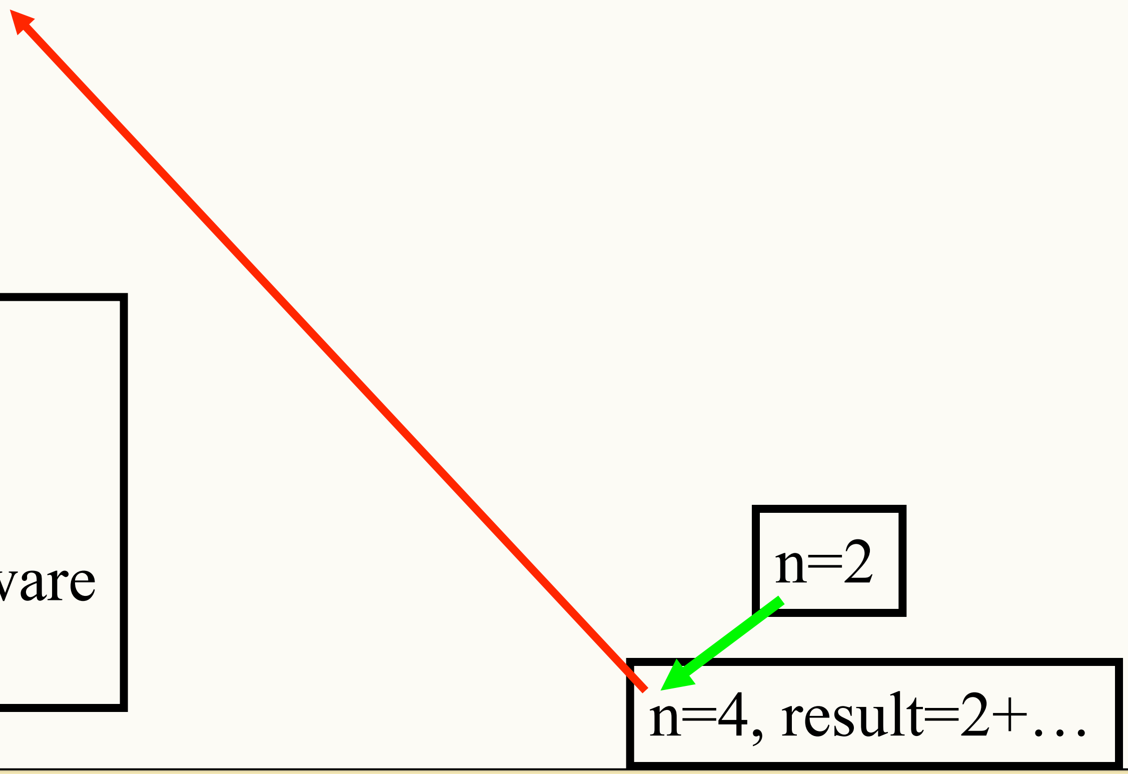
$\boxed{n=2}$

$\boxed{n=4,\ result=2+\ldots}$

Aside: Computer Security

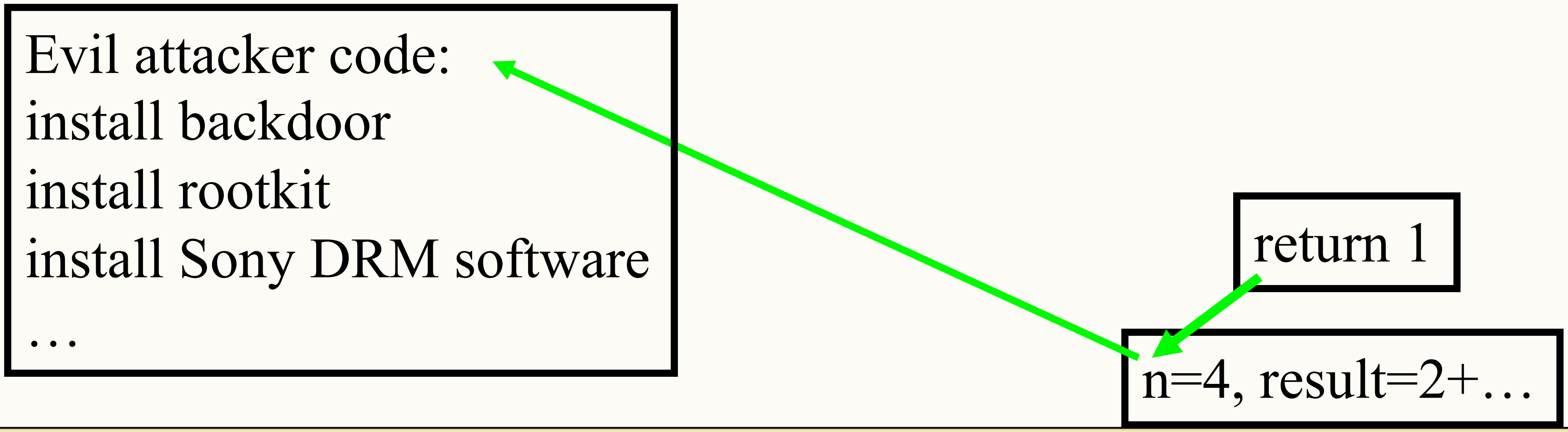
$$fib_n = \begin{cases} 1 & if n = 1 \\ \rightarrow 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

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



Aside: Computer Security

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



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

return 1

n=4, result=2+…

Aside: Computer Security

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

$\rightarrow$

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

n=4, result=2+…

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!

Random String Permutations

Problem: Permute a string so that every reordering of the string is equally likely. You may use a function **randrange(n)**, which selects a number **[0,n)** uniformly at random.

Random String Permutations Understanding the Problem

- A string is:
  - an empty string **or** a letter plus the rest of the string.
- We want every letter to have an equal chance to end up first. We want all permutations of the rest of the string to be equally likely to go after.
- And.. there's only one empty string.

Random String Permutations Algorithm

```
PERMUTE(s):
  if s is empty, just return s

  else:
    use randRange to choose a random first letter
    permute the rest of the string  (minus that random letter)
    return a string that starts with the random letter
       and continues with the permuted rest of the string
```

Converting Algorithm to Psudocode

```
PERMUTE(s):
  if s is empty, just return s
  else:
    choose random letter
    permute the rest
    return random letter + rest
```

Random String Permutations (Code)

```
string permute(string str) {
  // if s is empty
  if (str.size() == 0)
    return str; // just return s

  else {
    int index = choose_random_index(str);
    char c = str[index];

    string rest = str.erase(index, 1);
    string rest_permuted = permute(rest);

    return c + rest_permuted;
  }
}
```

```
g++ -std=c++11 permute.cc
```

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.

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?

A: Yes this will result in a stack overflow

B: No, this will magically not result in a stack overflow

C: It depends

D: None of the above

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, 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 (should be CPSC 110 review!)

- 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.

CPSC 221 Administrative Notes

- Programming project
  - Due Mon, March 2nd
  - Make sure that it is work in progress

– Assignment 1 will be returned today
  - We'll make a post on Piazza regarding who marked which question
  - If you have any concerns, talk to the responsible TA first
  - If you're still unhappy with your mark, you can make a formal appeal and I'll remark your entire assignment

Midterm Logistics

- This Wednesday, Feb 25, from 6pm to (approx) 8pm.
- Our section is in HENN 201+202
- Two-stage exam: individual and group!
  - Individual portion is 70 minutes. Normal midterm rules apply. No collaboration of any kind permitted!
  - Group portion is 40 minutes, starting after individual portion ends. Groups of 3-4 students do ONE exam together. (Do not tear it apart!)
  - Your mark is max of your individual score or a weighted average with your group exam score. (Group part can only help you.)
- Exam is open-book (3 books max), and open-notes (up to 3" binder of A4 or US-Letter size paper).

Three more things

- Midterm: Form study groups
  - Collaborate on Piazza
  - Answer to your peers post
- Compilers and optimization by default
- Hows PeerWise?

So, Where Were We?

- Recognize algorithms as being iterative or recursive.
- Draw a recursion tree and relate the depth to a) the number of recursive calls and b) the size of the runtime stack. Identify and/or produce an example of infinite recursion
- Evaluate the effect of recursion on space complexity (e.g., explain why a recursively defined method takes more space then an equivalent iteratively defined method.).
- Describe how tail recursive algorithms can require less space complexity than non-tail recursive algorithms.

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);
}
```

Is this function tail recursive?

a. Yes.

b. No.

c. Not enough information.

Tail Recursive?

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

Is this function 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);
}
```

Is this function 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);
}
```

Is this function tail recursive?

a. Yes.

b. No.

c. Not enough information.

Tail Recursive?

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

Is this function tail recursive?

a. Yes.

b. No.

c. Not enough information.

Tail Recursive?

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

Is this function tail recursive?

a. Yes.

b. No.

c. Not enough information.

Recursion vs. Iteration

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

A: Recursion

B: Iteration

C: As powerful as each other

D: I don’t know!

Simulating a Loop with Recursion

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

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

```
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.

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

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

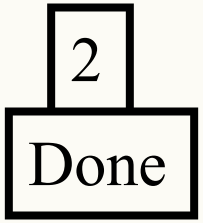
```
push(Done); push(n); pc=Start;
while (1) {
   if (pc==Done) break;
   if (pc==Start) {
     n=pop();
     if (n == 0) {
       pc=pop(); push(1); continue;
     } else {
       push(n); //save old n
       push(Middle);push(n-1);pc=Start;
       continue;
     }
   } else { //pc==Middle
     result=pop(); 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);
}
```

```
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
```

n=2

Done

Simulating Recursion with a Stack

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

n=2

Done

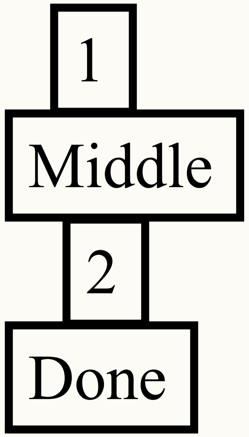
```
push(Done); push(n); pc=Start;
while (1) {
  if (pc==Done) break;
  if (pc==Start) {
    n=pop();
    if (n == 0) {
      pc=pop(); push(1); continue;
    } else {
      push(n); //save old n
      push(Middle);push(n-1);pc=Start;
      continue;
    }
  } else { //pc==Middle
    result=pop(); oldn=pop();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

```
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
```

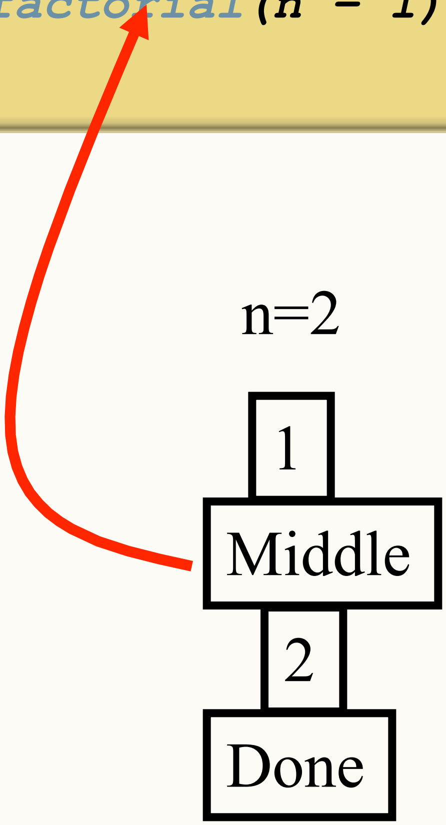
n=2



Simulating Recursion with a Stack

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

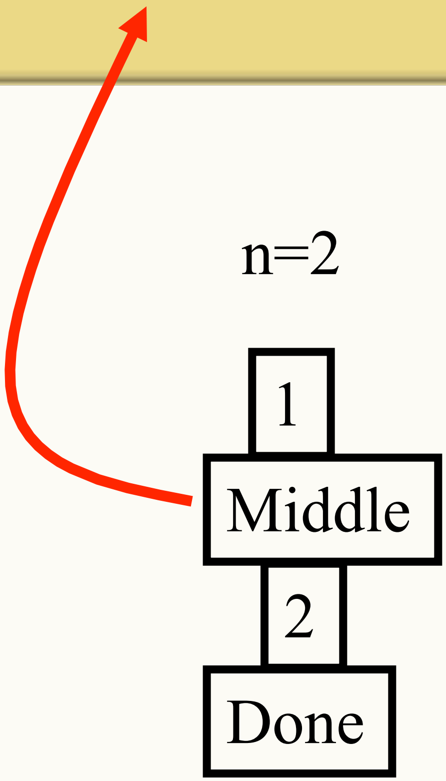
```
push(Done); push(n); pc=Start;
while (1) {
  if (pc==Done) break;
  if (pc==Start) {
    n=pop();
    if (n == 0) {
      pc=pop(); push(1); continue;
    } else {
      push(n); //save old n
      push(Middle);push(n-1);pc=Start;
      continue;
    }
  } else { //pc==Middle
    result=pop(); oldn=pop();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



Simulating Recursion with a Stack

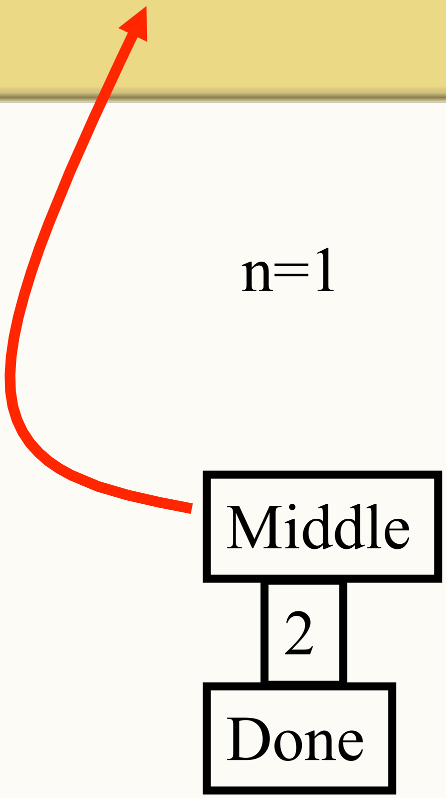
```
int factorial (int n) {
  if (n == 0) return 1;
  else
    return n *
   factorial(n - 1);
}
```

```
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);
}
```

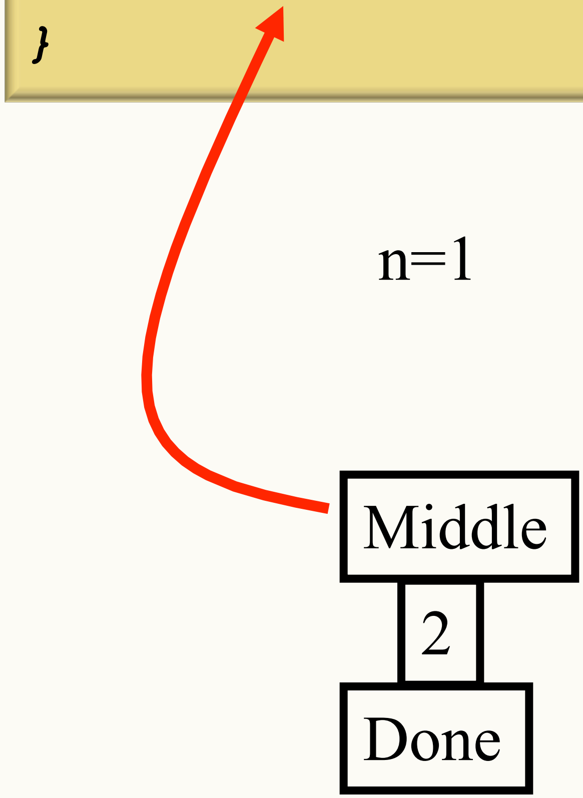


```
push(Done); push(n); pc=Start;
while (1) {
  if (pc==Done) break;
  if (pc==Start) {
    n=pop();
    if (n == 0) {
      pc=pop(); push(1); continue;
    } else {
      push(n); //save old n
      push(Middle);push(n-1);pc=Start;
      continue;
    }
  } else { //pc==Middle
    result=pop(); oldn=pop();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

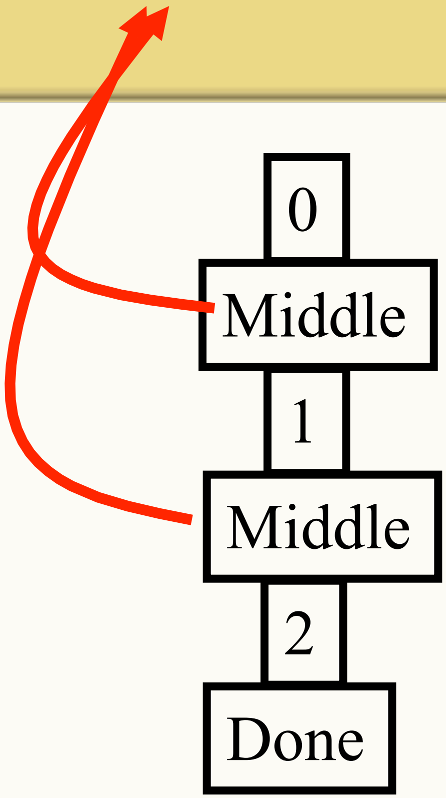
```
push(Done); push(n); pc=Start;
while (1) {
   if (pc==Done) break;
   if (pc==Start) {
     n=pop();
     if (n == 0) {
       pc=pop(); push(1); continue;
     } else {
       push(n); //save old n
       push(Middle);push(n-1);pc=Start;
       continue;
     }
   } else { //pc==Middle
     result=pop(); 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
```

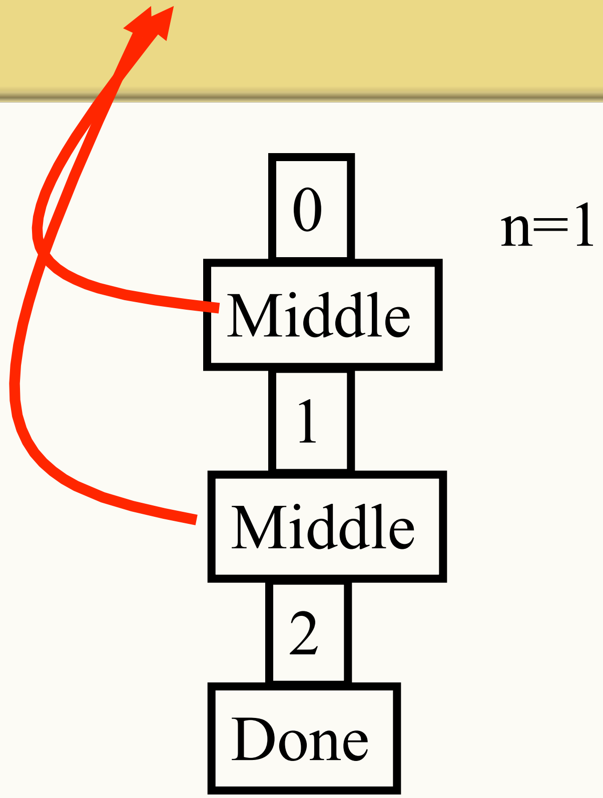


n=1

Simulating Recursion with a Stack

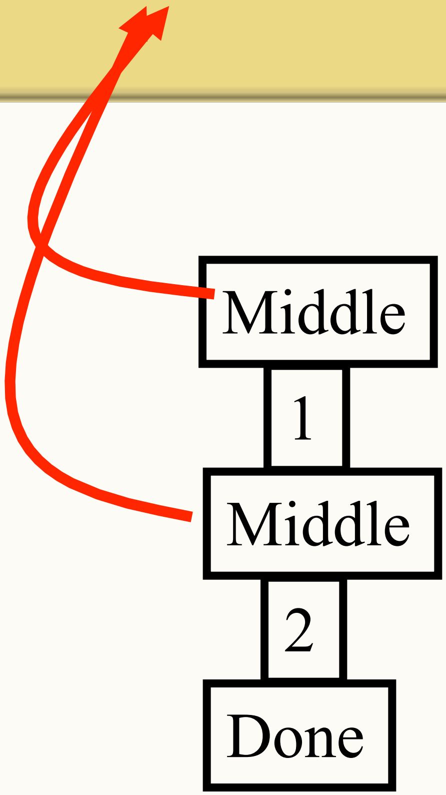
```
int factorial (int n) {
  if (n == 0) return 1;
  else
    return n *
   factorial(n - 1);
}
```

```
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);
}
```

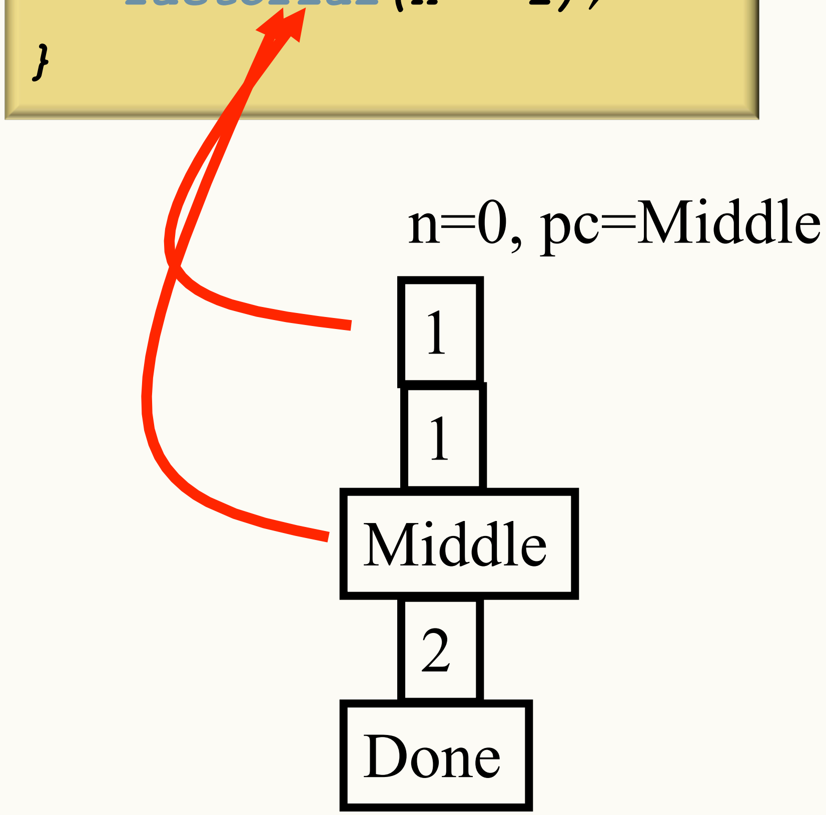


```
push(Done); push(n); pc=Start;
while (1) {
  if (pc==Done) break;
  if (pc==Start) {
    n=pop();
    if (n == 0) {
      pc=pop(); push(1); continue;
    } else {
      push(n); //save old n
      push(Middle);push(n-1);pc=Start;
      continue;
    }
  } else { //pc==Middle
    result=pop(); oldn=pop();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

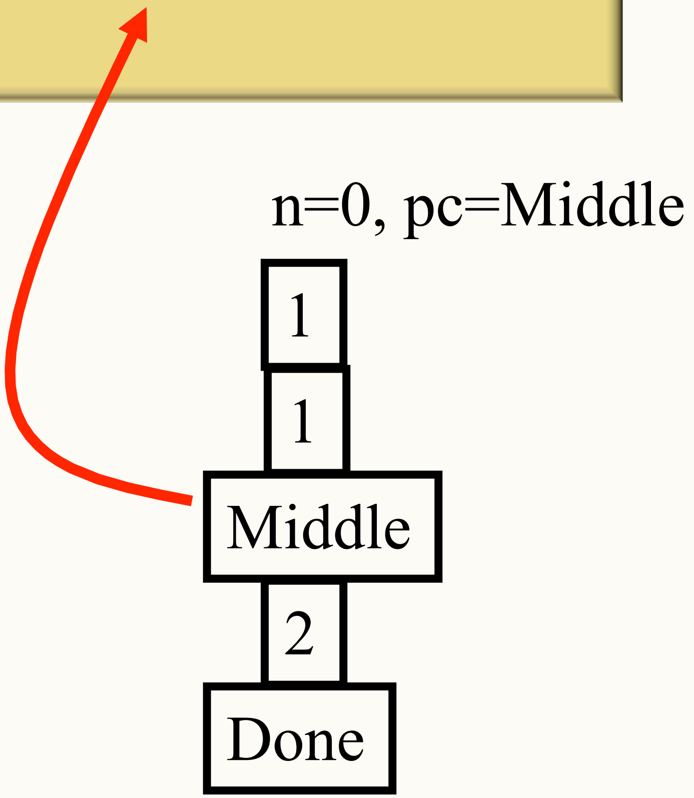
```
push(Done); push(n); pc=Start;
while (1) {
  if (pc==Done) break;
  if (pc==Start) {
    n=pop();
    if (n == 0) {
      pc=pop(); push(1); continue;
    } else {
      push(n); //save old n
      push(Middle);push(n-1);pc=Start;
      continue;
    }
  } else { //pc==Middle
    result=pop(); oldn=pop();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

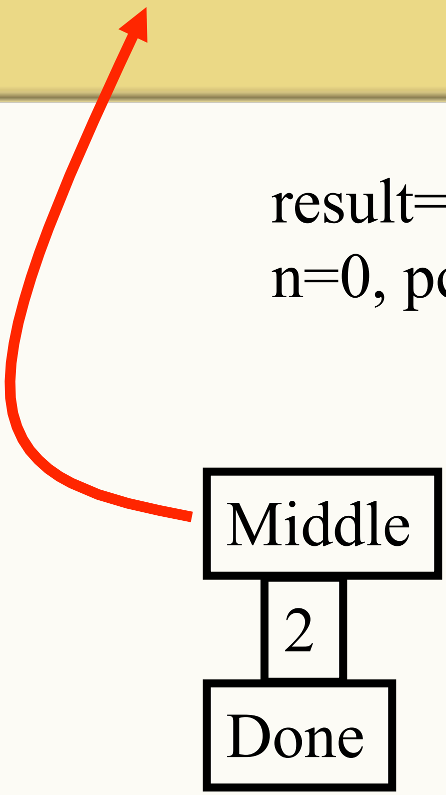


Simulating Recursion with a Stack

```
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
```

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

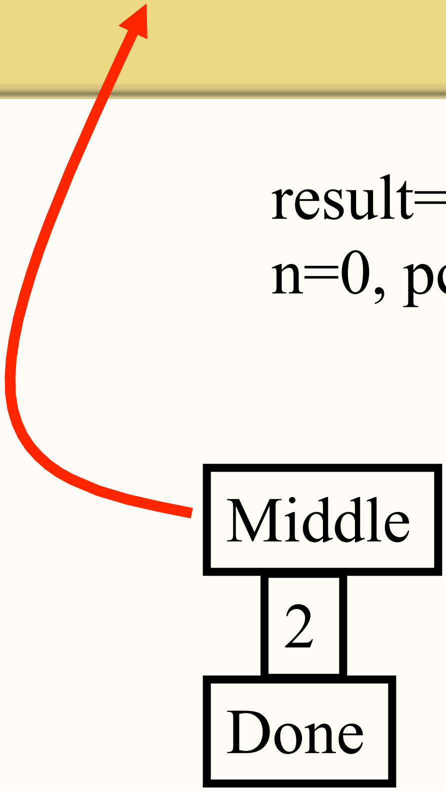


Simulating Recursion with a Stack

```
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
```

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

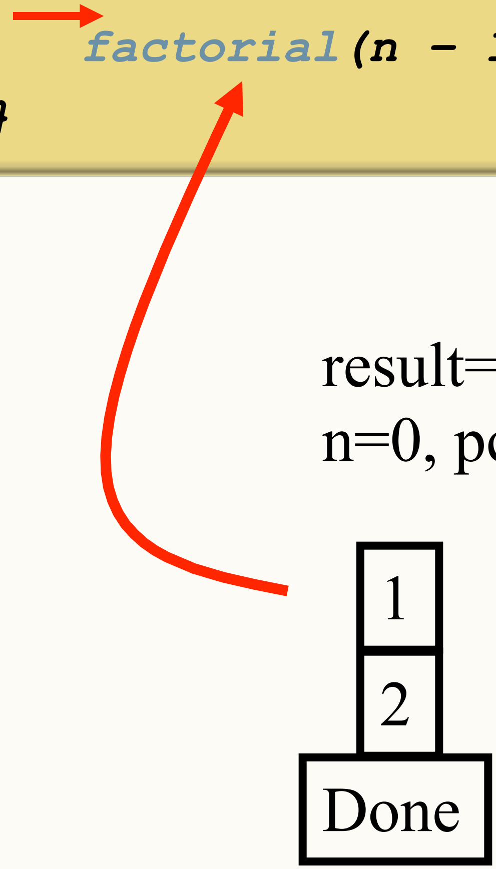


Simulating Recursion with a Stack

```
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
```

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

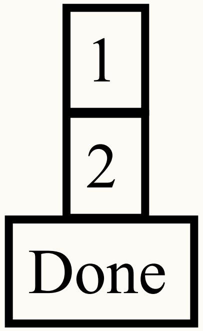


Simulating Recursion with a Stack

```
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
```

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

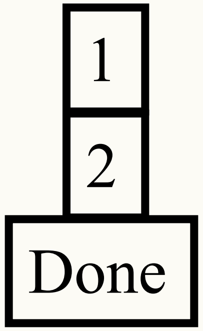


Simulating Recursion with a Stack

```
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
```

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



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);
}
```

```
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
```

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

$\boxed{\text{Done}}$

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
```

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

$\boxed{2}$

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
```

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

$\boxed{2}$

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
```

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

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

$\boxed{2}$

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.

Recursion vs. Iteration

Which one is more efficient or powerful?
Recursion or iteration?

It's probably easier to shoot yourself in the foot without noticing when you use recursion, and the call stack may carry around a bit more (a constant factor more) memory than you really need to store, but otherwise…

Neither is more efficient.

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

Clean up your code to look nicer.

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

Let’s get back to proofs

- 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.

Induction and Recursion, Twins Separated at Birth?

Induction	**Recursion**
Base case Prove for some small value(s).	Base case Calculate for some small value(s).
Inductive Step Otherwise, break a larger case down into smaller ones that we assume work (the Induction Hypothesis).	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.

Thinking Recursively (Old Slide)

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 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*factorial(k-1)`**. By IH, **`factorial(k-1)`** = (k-1)! and k! = k*(k-1)! by definition.

Random String Permutations Algorithm (old example)

```
PERMUTE(s):
  if s is empty, just return s

  else:
    use randRange to choose a random first letter
    permute the rest of the string  (minus that random letter)
    return a string that starts with the random letter
       and continues with the permuted rest of the string
```

Random String Permutations (Code)

```
string permute(string str) {
  // if s is empty
  if (str.size() == 0)
    return str; // just return s

  else {
    int index = choose_random_index(str);
    char c = str[index];

    string rest = str.erase(index, 1);
    string rest_permuted = permute(rest);

    return c + rest_permuted;
  }
}
```

Proving A Recursive Algorithm Works

- **Problem**: Prove that our algorithm for randomly permuting a string gives an equal chance of returning every permutation (assuming **`randrange(n)`** works as advertised).

Proving A Recursive Algorithm Works

- Base case: an empty string cannot be permuted; so returning the empty string is correct.
- Induction hypothesis: Assume that our call to permute(str) works (uniformly randomly permutes str).
- Inductive step: We choose the first letter uniformly at random from across the string. To get a random permutation, we need only randomly permute the remaining letters. When we call permute(substr), it does exactly that.

CPSC 221 Administrative Notes

- How was midterm?

- Connect
  - Lab #4
  - PeerWise #2,
  - Assignment #1 → see assignment follow-up on Piazza

- Programming Assignments
  - Milestone being marked
  - Final handin due on Monday

So, Where were we

- Describe the relationship between recursion 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 (e.g., explain why a recursively defined method takes more space then an equivalent iteratively defined method.).
- Describe how tail recursive algorithms can require less space complexity than non-tail recursive algorithms.
- Recognize algorithms as being iterative or recursive.
- Convert recursive solutions to iterative solutions and vice versa.
- Draw a recursion tree and relate the depth to a) the number of recursive calls and b) the size of the runtime stack. Identify and/or produce an example of infinite recursion

Today

- Take a loop code fragment and express it mathematically in order to prove its correctness inductively (specifically describing that the induction is on the iteration variable).

- In simpler cases, determine the loop invariant.

Recurrence Relations

- See examples from asymptotic analysis slides
- **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);
}
```

Recurrence Relations

```
// 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; O(1)

  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)
}
```

Note: Let n be # of elements considered in the array (right – left + 1).

```
For n=0:  T(0)  =  1
For n>0:  T(n)  =  T(⌊n/2⌋)  +  1
```

Binary Search Problem

```
For n=0: T(0)  = 1
For n>0: T(n)  = T(⌊n/2⌋)  + 1
```

To guess the answer we simplify

Change ⌊n/2⌋ to n/2, so 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
```

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

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

$= T(n/4) + 2$

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

$= T(n/16) + 4$

$= T(n/(2^i)) + i$ **let $n/2^i = 1$ so $n = 2^i \rightarrow i = \lg n$**

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

$= T(1) + \lg n = \lg n + 1$ $T(n) \in O(\lg n)$

Binary Search Problem

To **prove** that $T(n) \in O(\lg n)$, we use induction:

```
For n=0: T(0) = 1
For n>0: T(n) = T(⌊n/2⌋) + 1
```

Base cases

$T(0) = 1 \rightarrow$ but lg 0 is undefined

$T(1) = T(0) + 1 = 2 \rightarrow$ but lg 1 is 0

$T(2) = T(1) + 1 = 3 \rightarrow$ $T(2) \le c \lg 2 \rightarrow 3 \le c$

Let $c = 3$, $n_0 = 2$.

Base cases: $T(2) = 3 = 3 \lg 2$ ✓

Base cases: $T(3) = 3 \le 3 \lg 3$ ✓

Binary Search Problem

$T(0) = 1$, $T(1) = 2$, $T(2) = 3$, $T(3) = 3$
For $n>3$: $T(n) = T(\lfloor n/2 \rfloor) + 1$

- **Induction hyp: for all $2 \le k < n$,**
  - **$T(k) \le 3 \lg k$**
- **Inductive step, $n > 3$, in two cases**
  - **(odd & even)**

$$\begin{aligned} T(n) &= T((n-1)/2) + 1 \quad n \ge 5, \text{ so } (n-1)/2 \ge 2, \text{ IH applies} \\ &\le 3 \lg((n-1)/2) + 1 \\ &= 3 \lg(n-1) - 3 \lg 2 + 1 \\ &= 3 \lg(n-1) - 3 + 1 \\ &= 3 \lg(n-1) - 2 \\ &\le 3 \lg n \ \checkmark \end{aligned}$$

Binary Search Problem

T(0) = 1, T(1) = 2, T(2) = 3, T(3) = 3
For n>3: T(n) = T(⌊n/2⌋) + 1

- **Induction hyp:** for all $2 \le k < n$,
  - $T(k) \le 3 \lg k$
- **Inductive step**, $n > 3$, in two cases
  - (odd & even)

$$\begin{aligned} T(n) &= T(n/2) + 1 \quad \text{n ≥ 4, so n/2 ≥ 2, IH applies} \\ &\le 3 \lg(n/2) + 1 \\ &= 3 \lg n - 3 \lg 2 + 1 \\ &= 3 \lg n - 3 + 1 \\ &= 3 \lg n - 2 \\ &\le 3 \lg n \ ✓ \end{aligned}$$

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.

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.

Proving a Loop Invariant

Induction variable: number of times through the loop.

Base case: Prove the invariant is true before the first loop guard test.

Induction hypothesis: Assume the invariant holds just before some (unspecified) iteration's loop guard test.

Inductive step: Prove the invariant holds at the end of that iteration (just before the next loop guard test).

Termination: Make sure the loop will eventually end!

Loop Invariants: The Easy Way

- Convert for-loops to while-loops.
  - Easiest to reason about while-loops.
- Write your loop invariant to be true at the exact same time as you check the loop condition.
  - This is at the top/bottom of the loop body.
- Base Case: prove that your loop invariant holds when you first arrive at the loop.

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

Loop Invariants: The Easy Way

- Induction:
  - Assume the loop invariant holds at top of loop.
  - Prove that loop invariant holds at bottom of loop..
- 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.

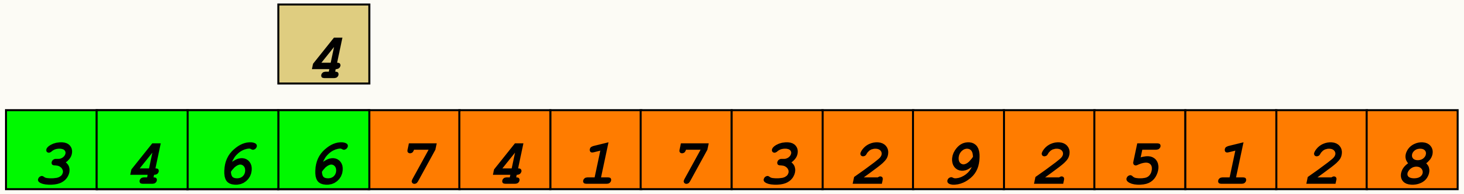
```
int i=1; // initialization stuff
while (condition) {
  loop body;
}
```

Insertion Sort

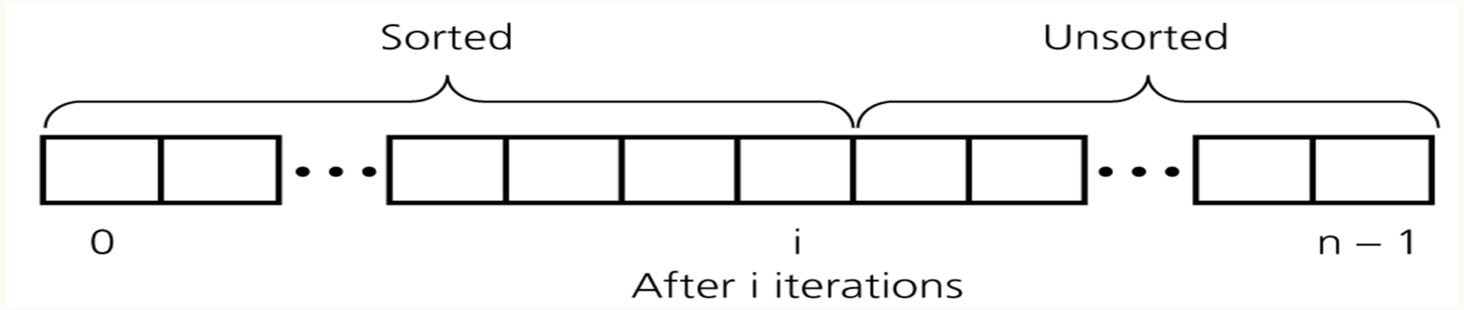
- Given a list, take the current element and insert it at the appropriate position of the list, adjusting the list every time you insert

Insertion Sort

- while some elements unsorted:
  - Using linear/binary search, find the location in the sorted portion where the 1st element of the unsorted portion should be inserted
  - Move all the elements after the insertion location up one position to make space for the new element



the fourth iteration of this loop is shown here



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
  - Invariant: the elements in array[0 . . i −1] consist of the elements originally in array[0 . . i −1] but in sorted order.

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++;
 }
```

- So, what's the base case?
  - Base Case: When the code first reaches the loop invariant i=1, so array[0..0] is trivially sorted.

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++;
 }
```

- Proof of inductive case is just like before.
  - Inductive Hypothesis: We assume array[0..i-1] is sorted at the top of the loop, and i<length.

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++;
 }
```

- Inductive Step: bSearch finds the correct index to put array[i], we shift elements array[newindex…i-1] so array[0..i] is sorted.
  - so loop invariant holds again at the bottom of the 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++;
 }
```

- Termination: The outer loop ends when i==length, which is eventually reached as i is increased in every iteration.
  - Invariant says array[0..i-1] is sorted, so array[0..length-1] is sorted.

Insertion Sort (in-class exercise)

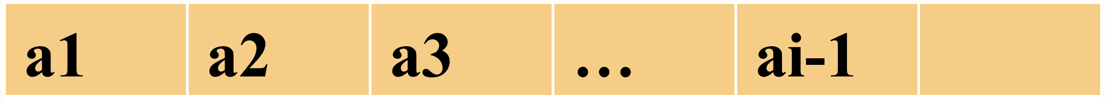
```
int i = 1;
 while (i < length){
   int val = array[i];
   int newIndex = bSearch(array, val, 0, i);
   for (int j = i; j > newIndex; j--)
     array[j] = array[j-1];

   array[newIndex] = val;
   i++;
 }
```

- Prove by induction that the inner loop operates correctly.
  - Talk about what the invariant means when the loop ends.

Insertion Sort (in-class exercise)

```
for (int j = i; j > newIndex; j--)
    array[j] = array[j-1];
```

- Invariant: The elements in array[0..j-1] + array[j+1..i] consist of the elements that were originally in array[0..i-1]
- Base Case: At the start of the first iteration, j==i, so array[0..j-1] is exactly the original array[0..i-1].



Insertion Sort (in-class exercise)

```
for (int j = i; j > newIndex; j--)
        array[j] = array[j-1];
```

- Inductive Hypothesis: we assume the invariant holds at the top of the loop after j iteration.

a1	**a2**	**…**	**aj-1**	**aj**	**aj+1**	**…**	**ai-1**	
a1	**a2**	**…**	**aj-1**		**aj**	**aj+1**	**…**	**ai-1**

- Inductive Step: The invariant doesn't care about array[j], so we can overwrite it with array[j-1]. So after j--, the invariant holds once again for the new j.

a1	**a2**	**…**	**aj-1**		**aj**	**aj+1**	**…**	**ai-1**
a1	**a2**	**…**		**aj-1**	**aj**	**aj+1**	**…**	**ai-1**

Insertion Sort (in-class exercise)

```
for (int j = i; j > newIndex; j--)
    array[j] = array[j-1];
```

- Termination: Since new index is an integer between 0 and j, loop terminates.
- When the loop terminates, j==newIndex. Therefore, array[0..newIndex-1] + array[newIndex+1..i] equals the old array[0..i-1].

Back to fibs

- 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
```

- Try a few numbers to convince yourself it works

Back to fibs

```
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, and what would it 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<=2 (when fib(n)=1).
  - So, it’s sort of like stuff on the stack, plus result doesn’t change…

Back to fibs

```
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
```

- Invariant: Sum of fib(i) for all i on stack, plus result equals fib(n)
- Base case: So now, what's the base case?

Back to fibs

```
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
```

- Invariant: Sum of fib(i) for all i on stack, plus result equals fib(n)
- Base case: Initially, n is only item on stack, and result=0. fib(n)+0=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!

Back to fibs

```
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
```

- Inductive step: Assume inductive hypothesis. We pop a number n off the stack.
  - If n <=2, then fib(n)=1, so by increasing result by 1, we maintain inductive hypothesis
  - If n>2, we push n-1 and n-2. But fib(n)=fib(n-1)+fib(n-2) (by definition), the sum of fib(i) for all i on the stack is unchanged.

Back to fibs

```
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: By reducing n each time, we're converging towards n=2. The loop will terminate when stack is empty.

- When stack is empty, sum of fib(i) for all i on stack is 0.
  - So, 0+result=fib(n). Therefore, result=fib(n).

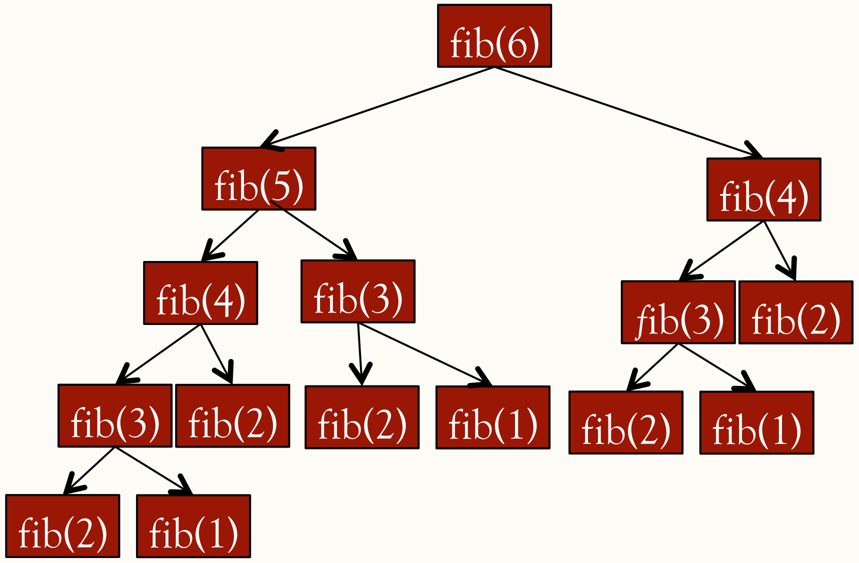
Back to fibs

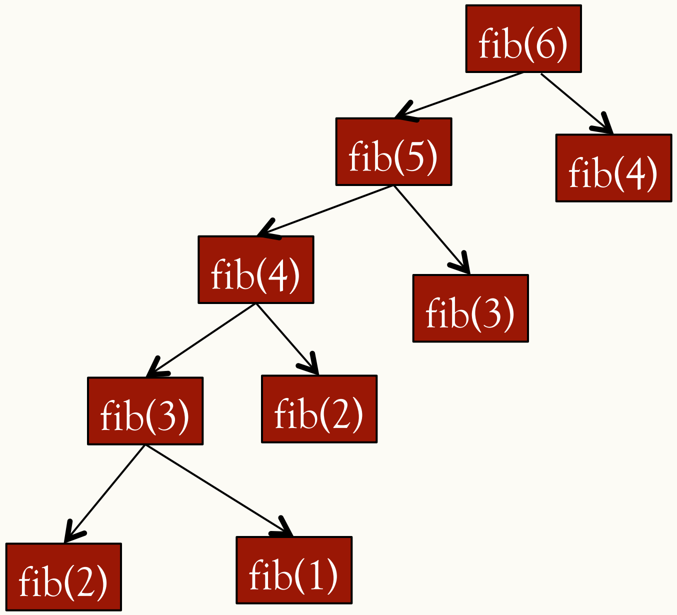
```
int fib(int n)
   result = 0
   push(n)
   while not isEmpty
     n = pop()
     if (n <= 2) result++;
     else push(n – 1); push(n – 2)
 return result
```

- Termination: 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] >…. By reducing n each time, we're converging towards n=2.
- When stack is empty, sum of fib(i) for all i on stack is 0.
  - So, 0+result=fib(n). Therefore, result=fib(n).

Avoiding Duplicate Calls Memoization

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





n	F_n
6	8
5	5
4	3
3	2
2	1
1	1

Memoization

```
int memoized_fib(int n) {
  if (ANSWERS[n] > 0) {
    // There's a value in ANSWERS[n].
    return ANSWERS[n];
  }
  else {
    int result;
    if (n == 1 || n == 2) {
      result = 1;
    }
    else {
      result = memoized_fib(n-1) + memoized_fib(n-2);
    }
    ANSWERS[n] = result;
    return result;
  }
}
```

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.
  - 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];
 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”

```
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];
 }
```

Learning goals revisited

- Describe the relationship between recursion 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 (e.g., explain why a recursively defined method takes more space then an equivalent iteratively defined method.).
- Describe how tail recursive algorithms can require less space complexity than non-tail recursive algorithms.
- Recognize algorithms as being iterative or recursive.
- Convert recursive solutions to iterative solutions and vice versa.
- Draw a recursion tree and relate the depth to a) the number of recursive calls and b) the size of the runtime stack. Identify and/or produce an example of infinite recursion

Learning goals revisited

- Take a loop code fragment and express it mathematically in order to prove its correctness inductively (specifically describing that the induction is on the iteration variable).

- In simpler cases, determine the loop invariant.