CPSC 259: Data Structures and Algorithms for Electrical Engineers

Recursion

Textbook References:

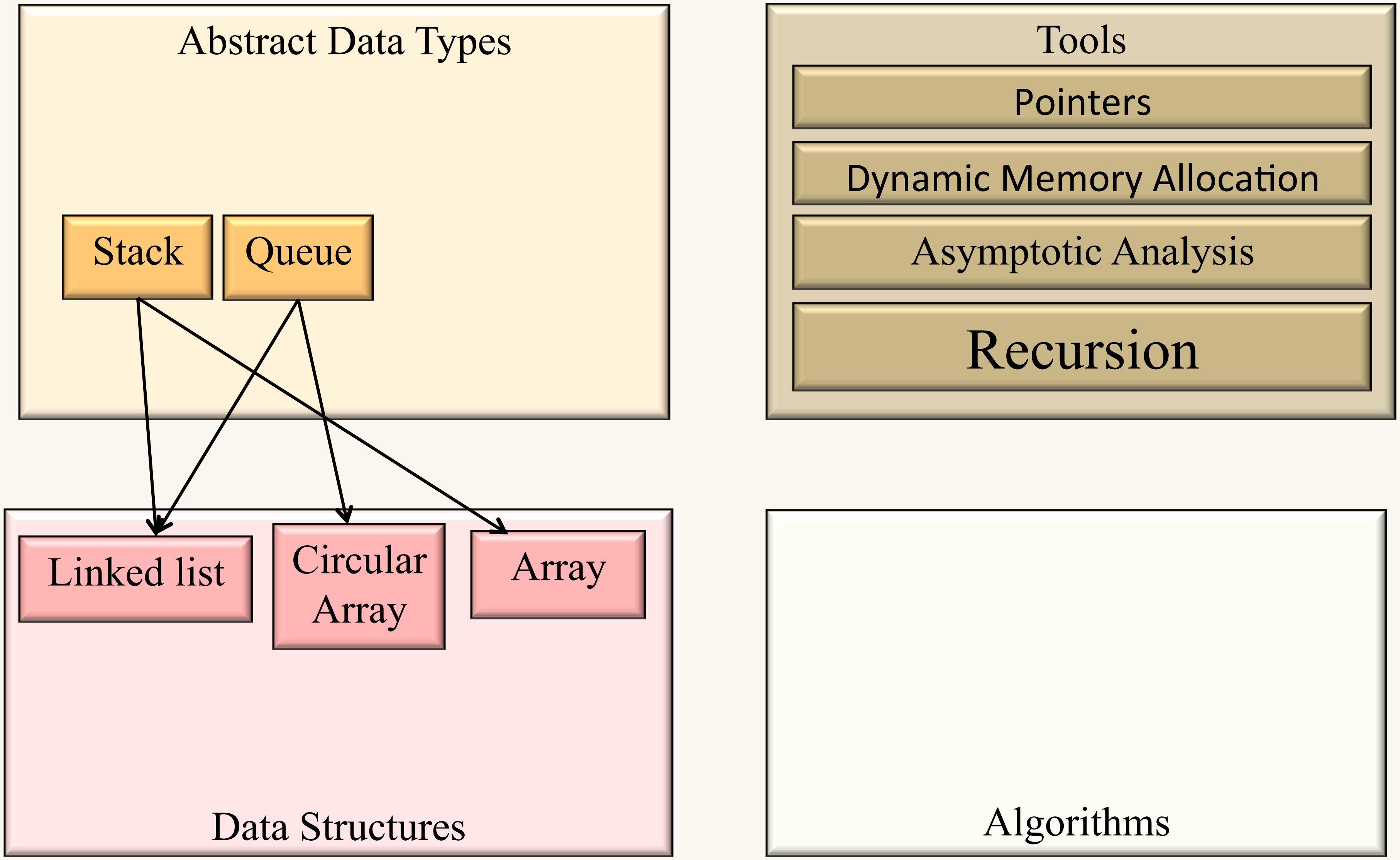
(a) Etter (Third Edition): Chapter 4, pages 193-197
(b) Thareja: Chapter 2, pages 83-93
(c) Thareja (second edition): 7.7.4

Hassan Khosravi
Borrowing some examples from Alan Hu and Steve Wolfman

Learning Goals

- design simple recursive functions.
- recognize algorithms as being iterative or recursive.
- describe how a computer runs recursive algorithms.
- demonstrate the ability to draw recursion trees.
- explain how stack overflow may arise as a result of recursion.
- explain why a recursively defined method may take more space than an equivalent iteratively defined method.

CPSC 259 Journey



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 CPSC259 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;
…
```

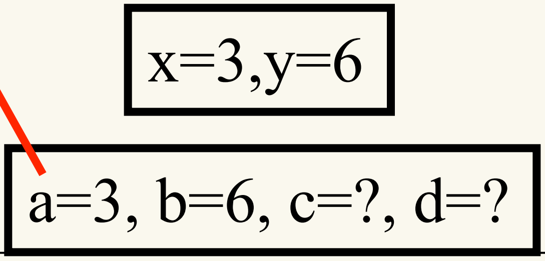
```
int foo(int x, int y) {
  while (x>0) {
      y++;
      x >>= 1;
  }
  return y;
}
```

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

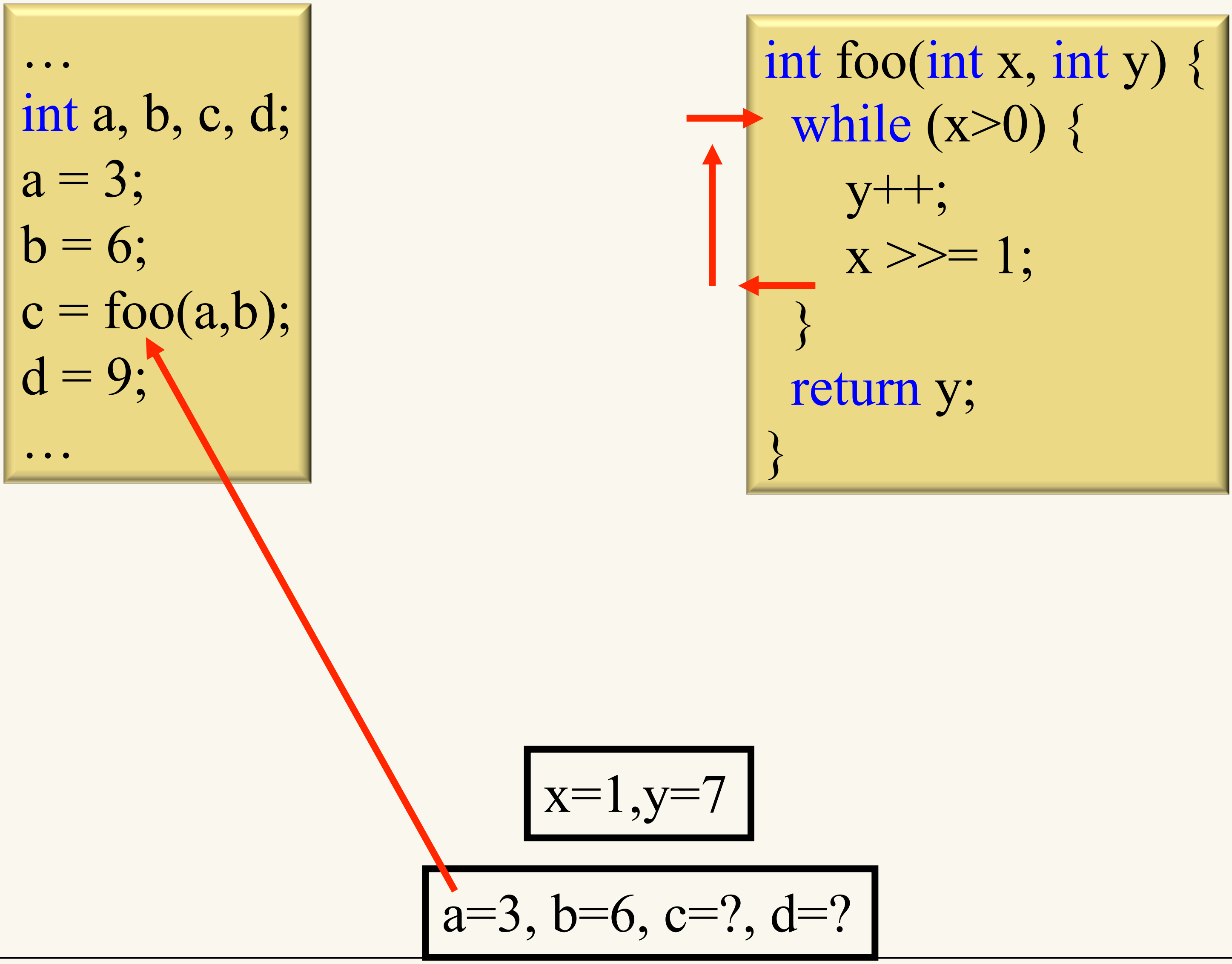
Activation Records on a Computer

```
…
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;
}
```



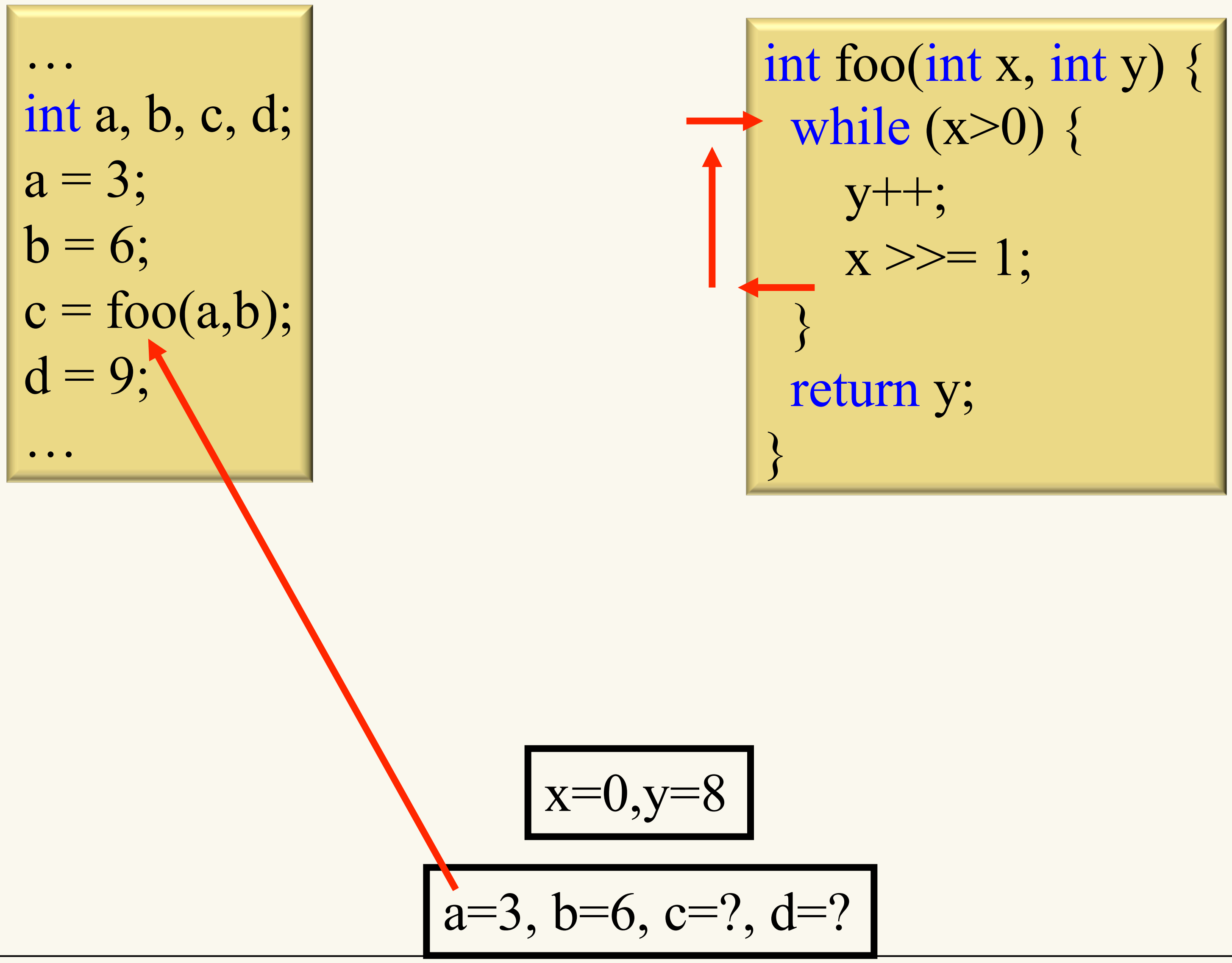
Activation Records on a Computer



```
…
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;
}
```

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

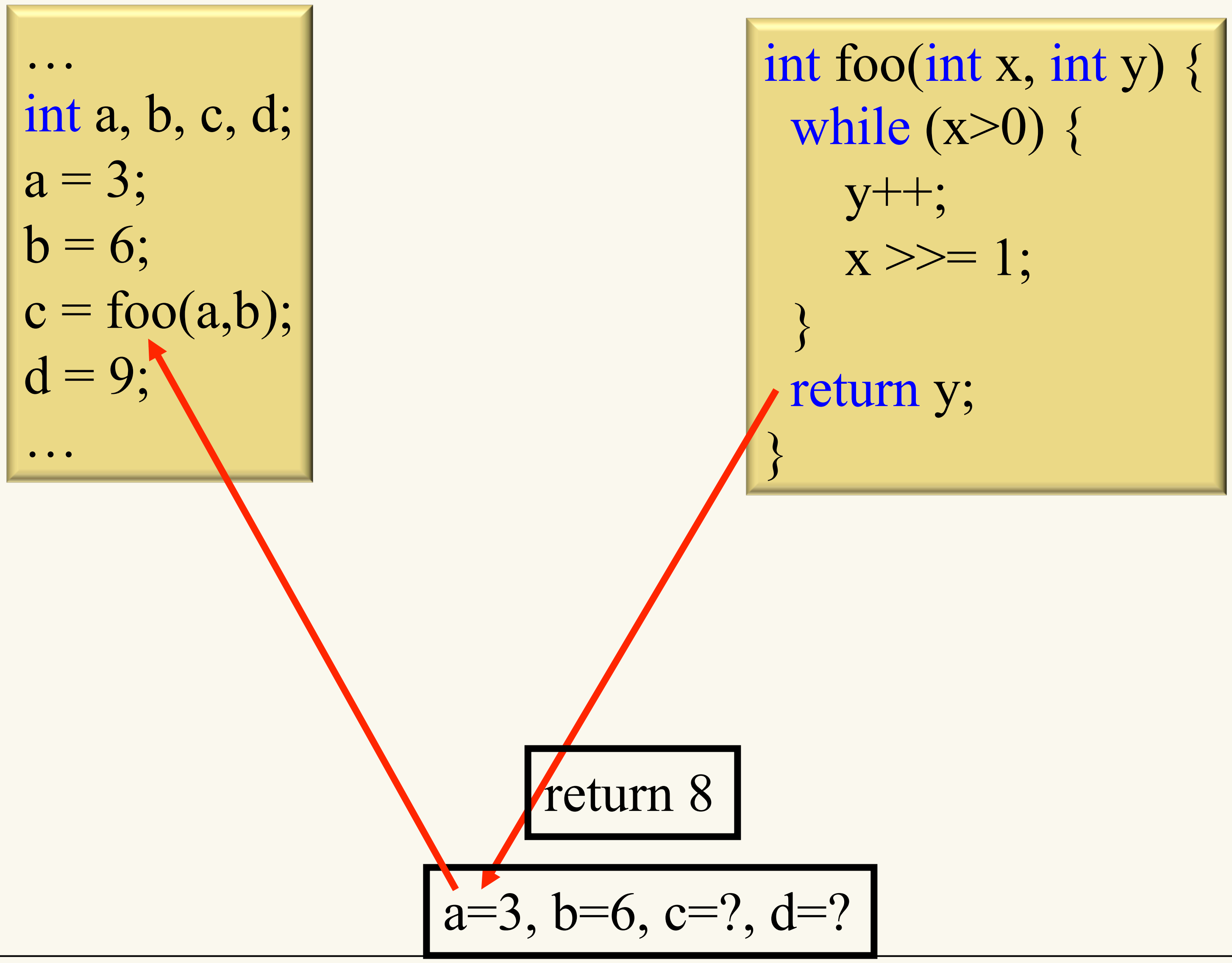
```
…
int a, b, c, d;
a = 3;
b = 6;
c = foo(a,b);
d = 9;
…
```

```
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!

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

$\boxed{n=4}$

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!

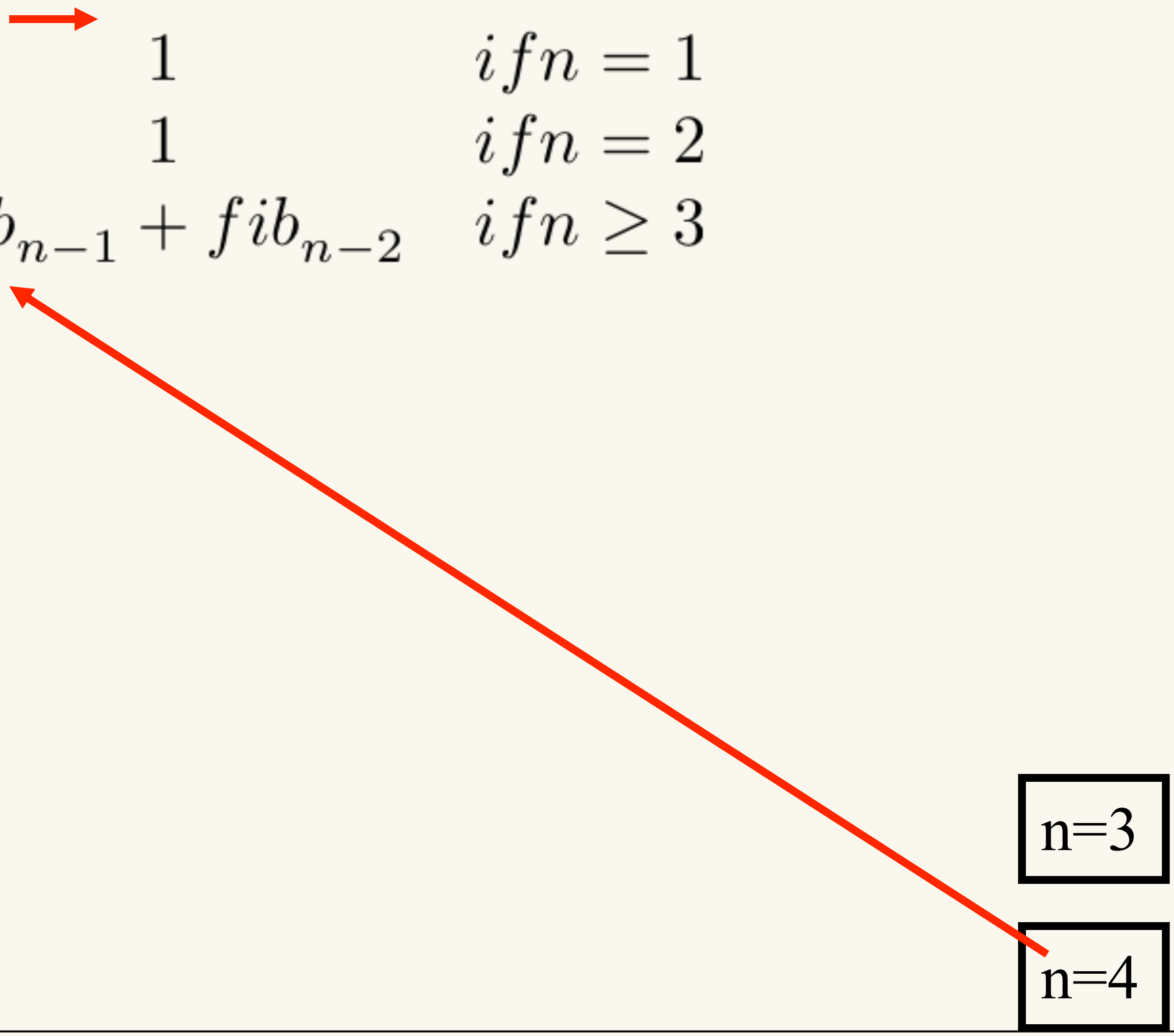
$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

$\nearrow$

$\boxed{n=4}$

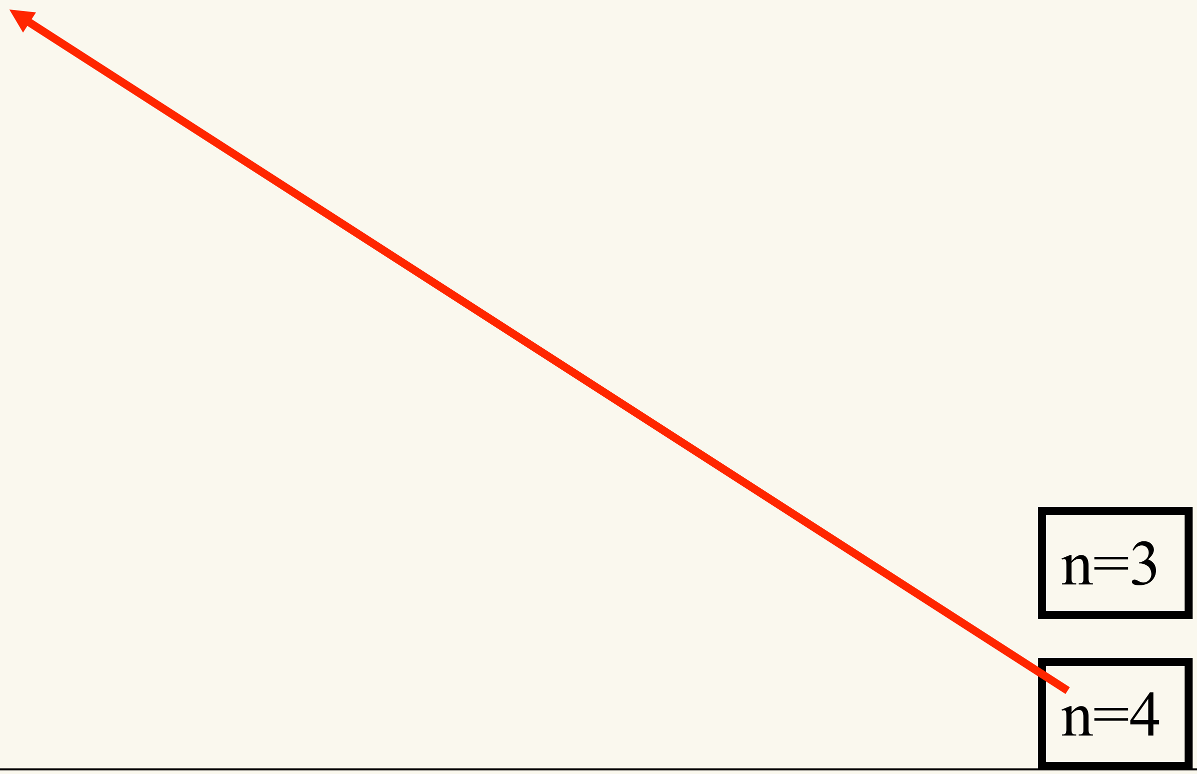
Recursion is handled the same way!

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



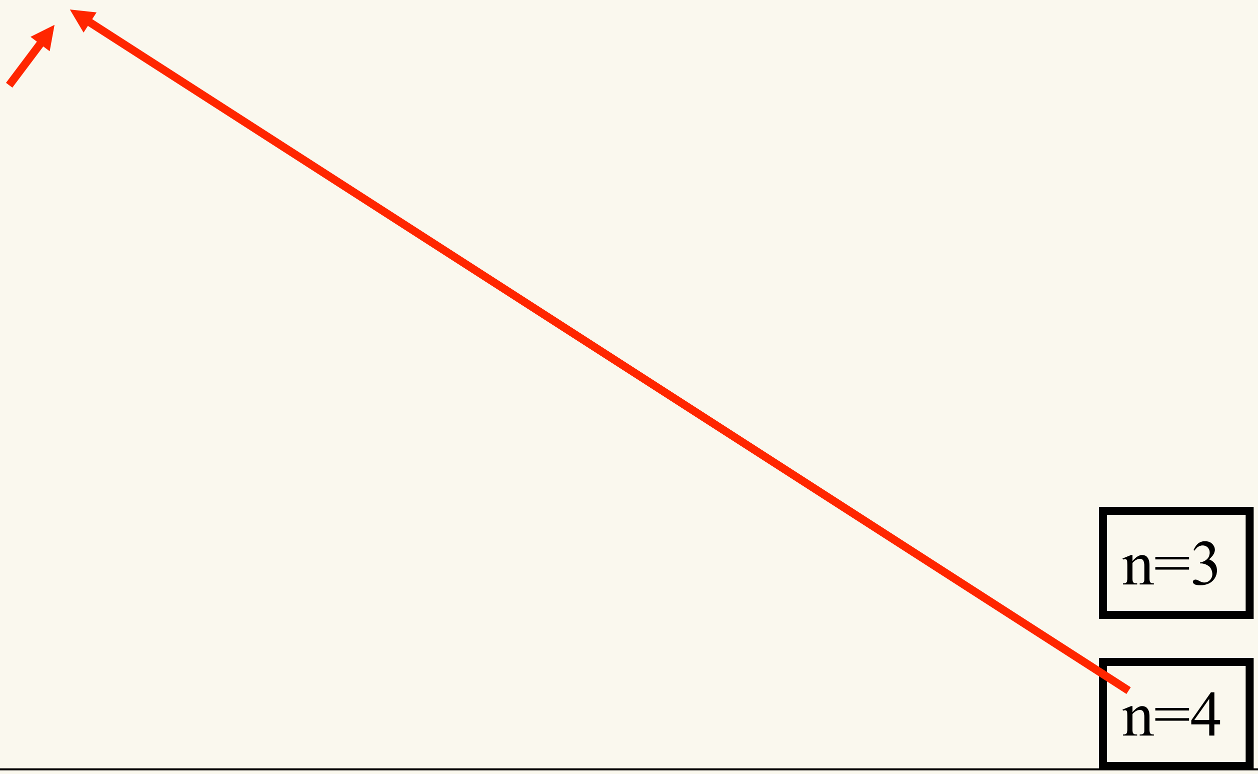
Recursion is handled the same way!

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



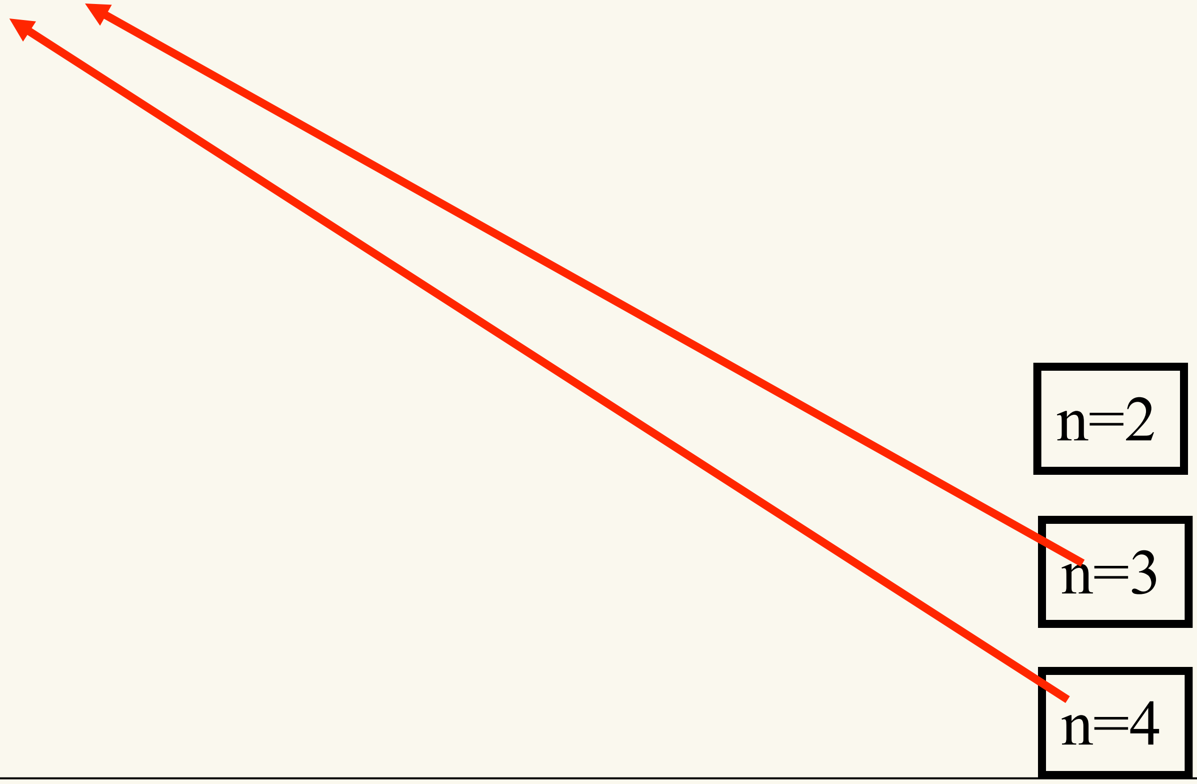
Recursion is handled the same way!

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



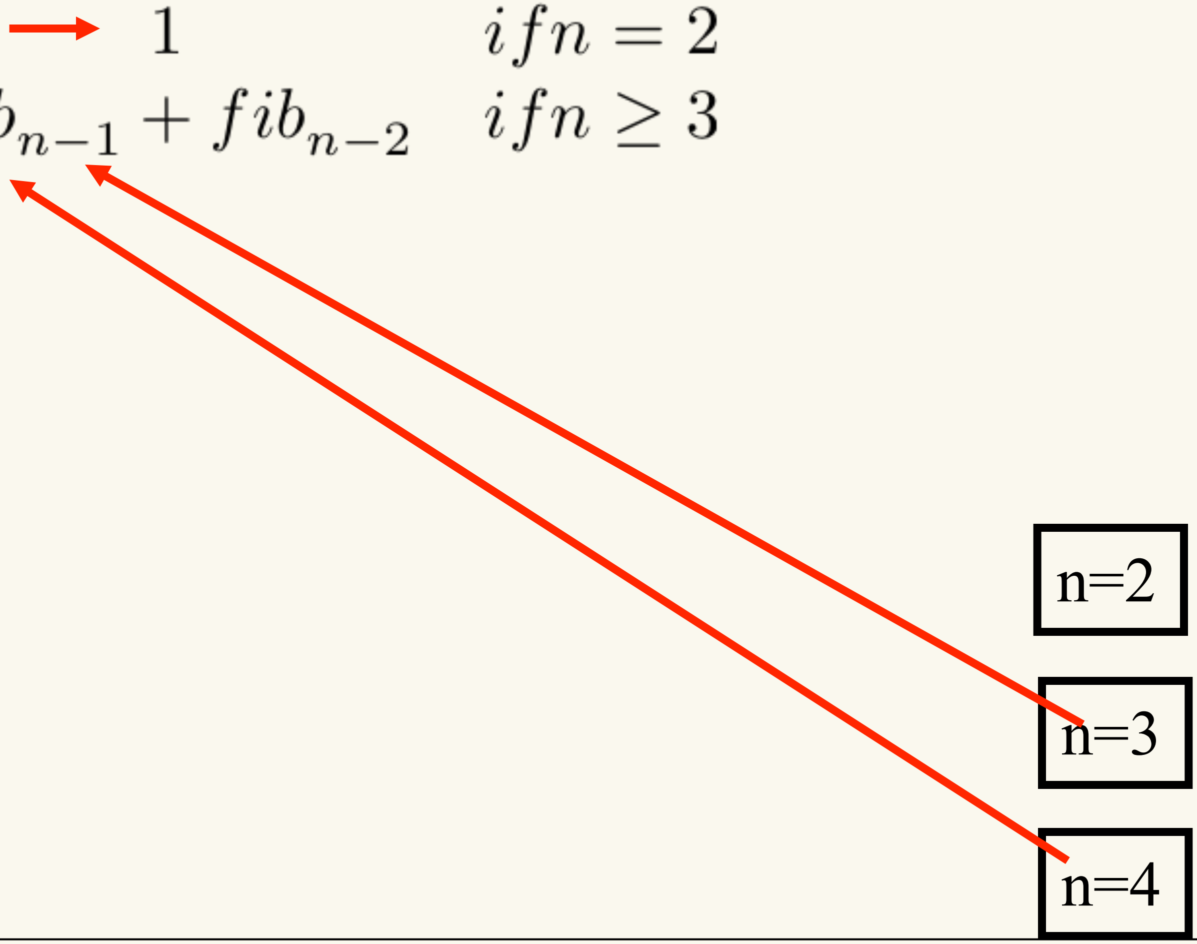
Recursion is handled the same way!

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



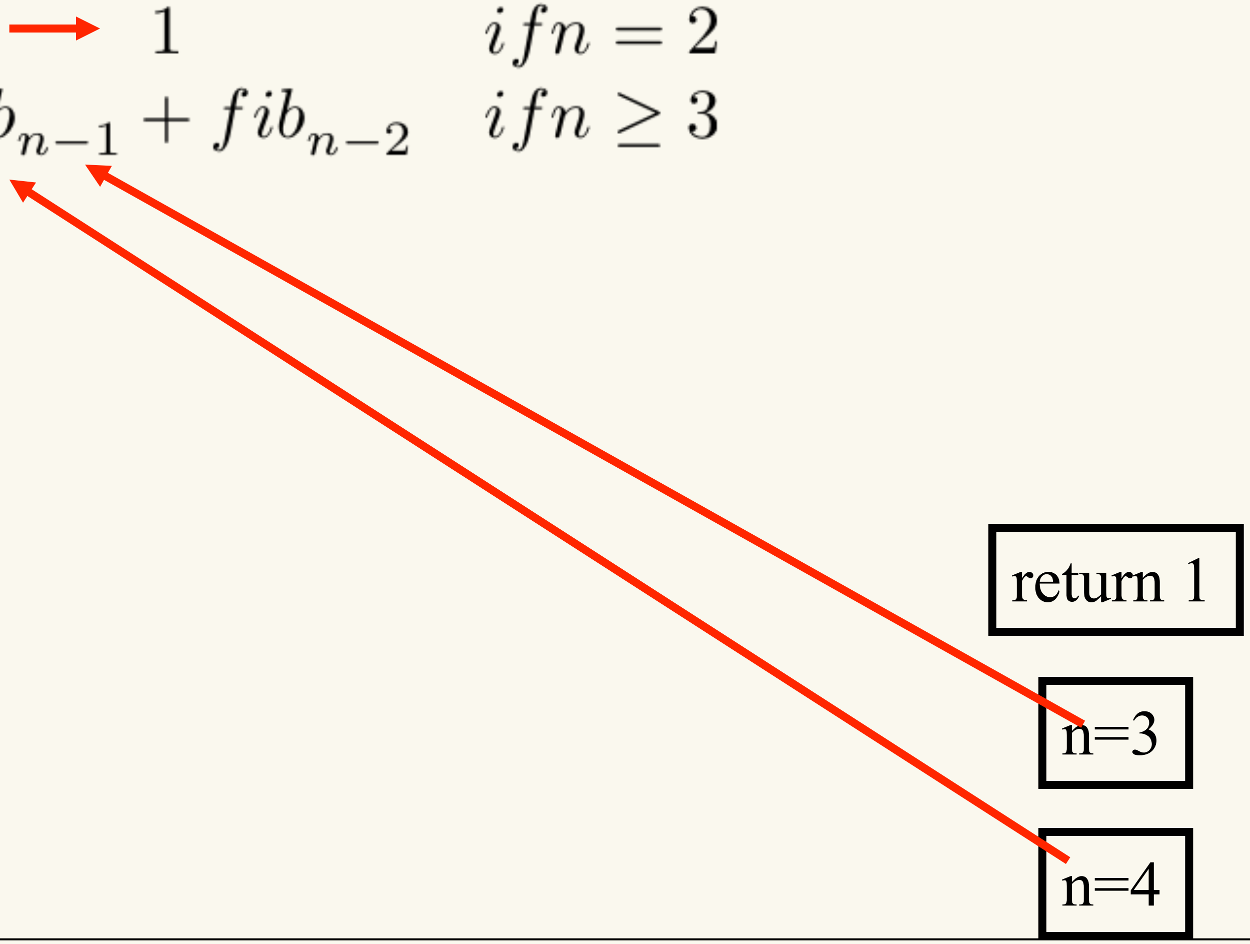
Recursion is handled the same way!

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



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!

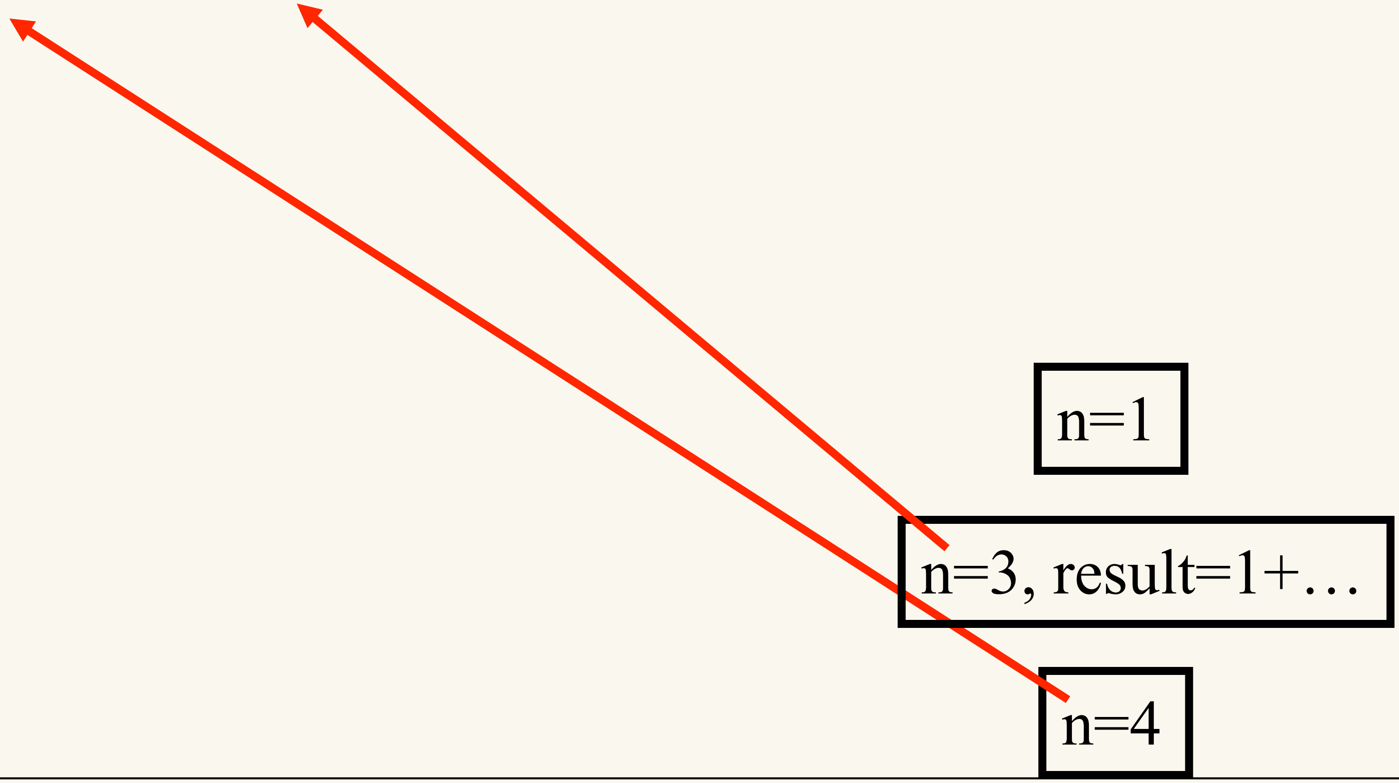
$$fib_n = \begin{cases} 1 & if n = 1 \\ 1 & if n = 2 \\ fib_{n-1} + fib_{n-2} & if n \geq 3 \end{cases}$$

$\boxed{\text{n=3, result=1+…}}$

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

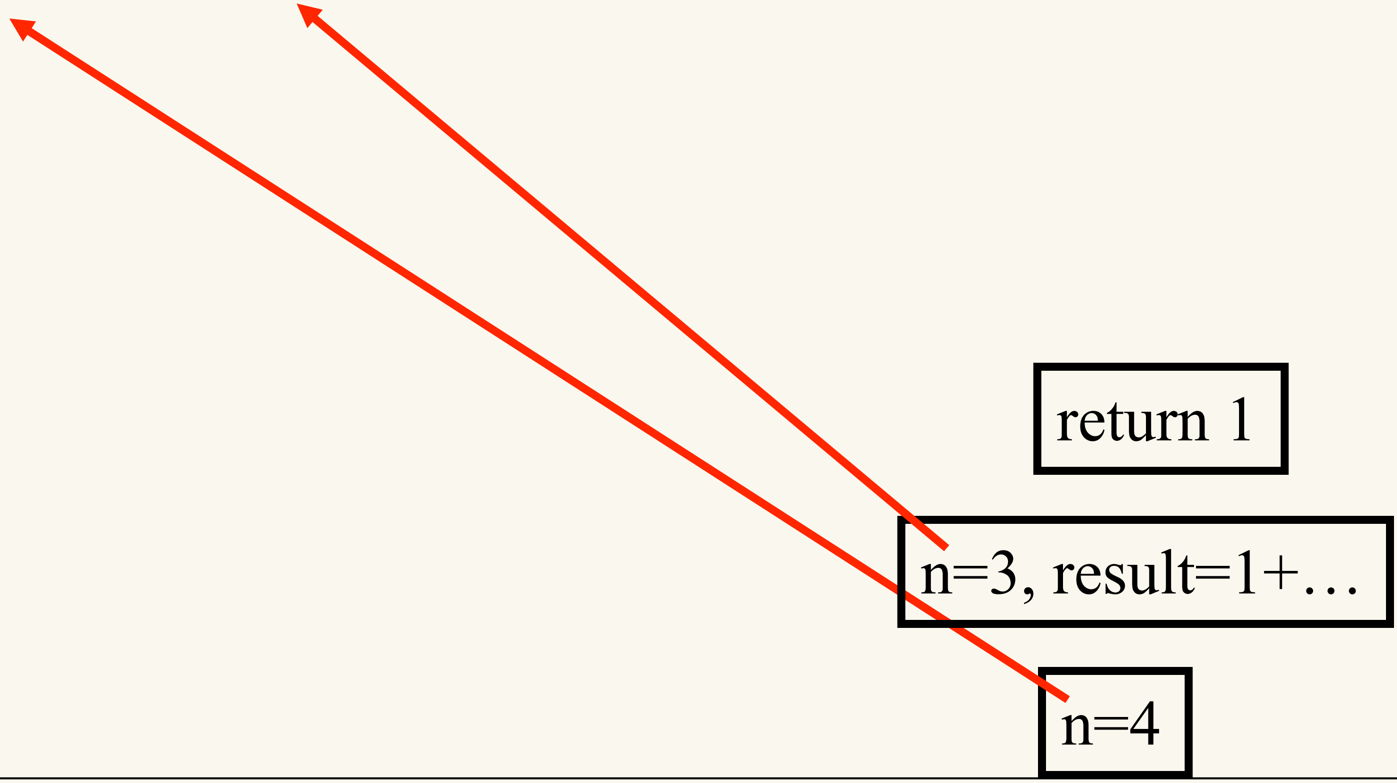
Recursion is handled the same way!

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



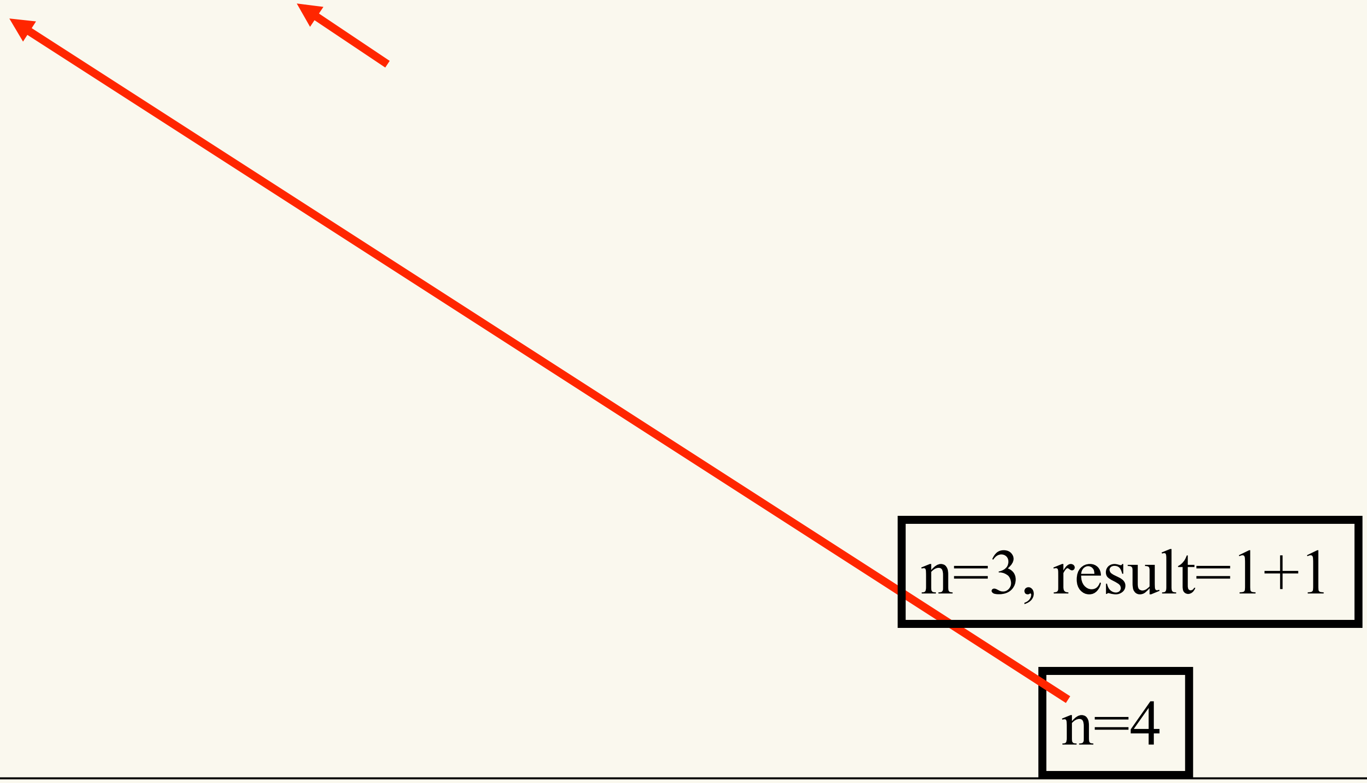
Recursion is handled the same way!

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



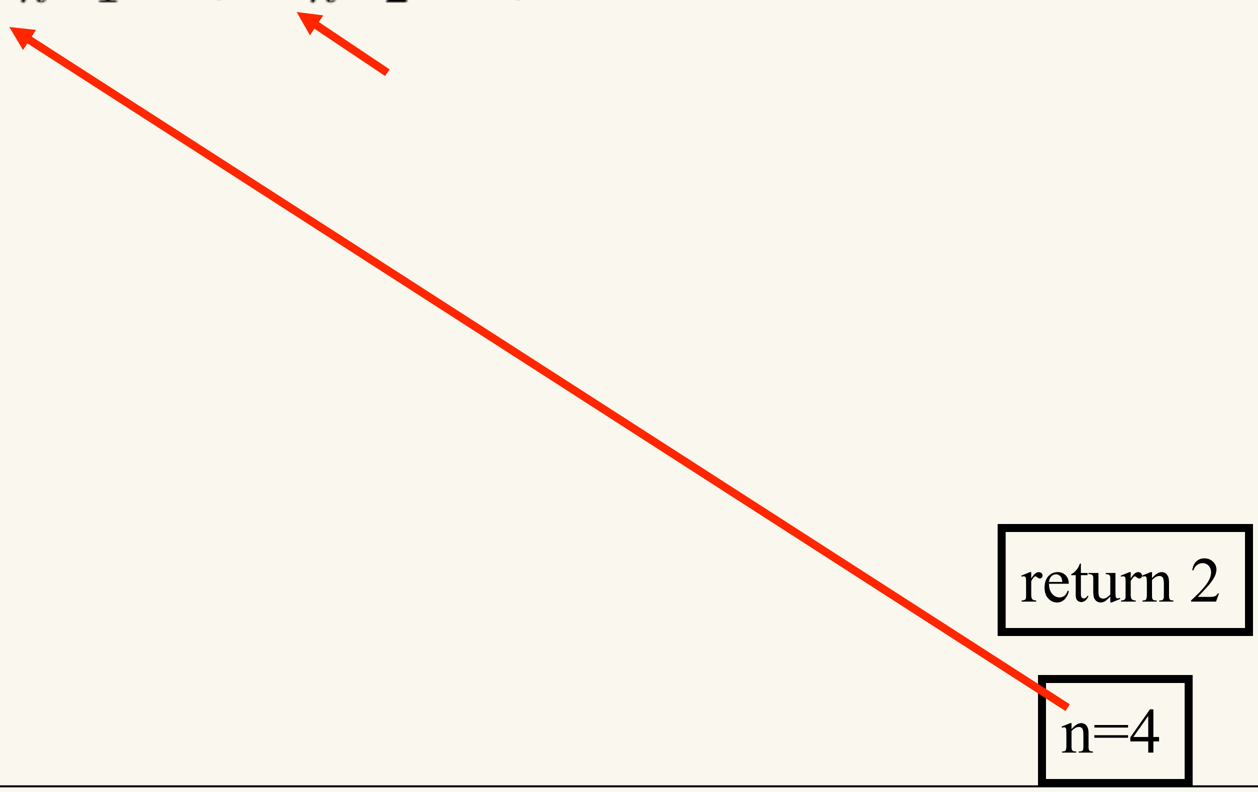
Recursion is handled the same way!

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



Recursion is handled the same way!

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



Recursion is handled the same way!

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

$\boxed{\text{n=4, result=2+…}}$

Recursion is handled the same way!

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

$\boxed{\text{n=4, result=2+…}}$

Recursion is handled the same way!

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

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

$\boxed{\text{n=4, result=2+…}}$

Recursion is handled the same way!

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

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

$\boxed{\text{n=4, result=2+…}}$

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!

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

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

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

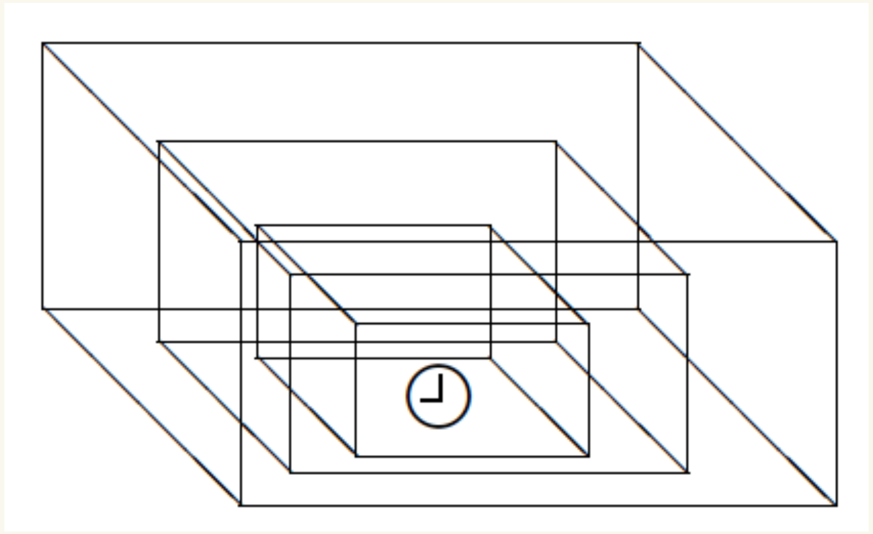
Recursion is handled the same way!

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

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

Thinking Recursively (Open a Present)

Problem: Your friends have given you a watch as a birthday present. To prolong the suspense when opening the present, they have wrapped it in several layers of gift-wrapping. Write an algorithm to open the present.



```
Open-Present(P)
  if  you can see the actual gift
     Say "Thank you"
  else
    Open the box
    Open-Present(contents of box)
```

Designing Recursive Functions (review)

- When designing recursive functions:
  - Don't start with code. Instead, write the *story* of the problem, in natural language.
  - 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!

When learning to drive a car there are two forms of knowledge:

(1) knowing how to operate a car.

(2) knowing how the car operates.

Naturally one can be a very good driver without having much knowledge of how a car itself operates.

Designing Recursive Functions

```
if  small enough to be solved directly:
        solve it.
else
      (1) recursively apply the algorithm to one or more smaller instances.
      (2) use the solution(s) from smaller instances to solve the problem.
```

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Factorial example

- Consider the following example
  - n! = n* (n-1) * (n-2)….*1
  - 5! = 5 * 4 * 3 * 2 * 1
  - 4! =   4 * 3 * 2 * 1
  - 5! = 5 * 4!
  - 0! = 1
- More generally

$$n! = \begin{cases} 1 & \text{for } n=0 \\ n*(n-1)! & \text{for } n>0 \end{cases}$$

Designing Recursive Functions

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

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Factorial example

factorial(0)

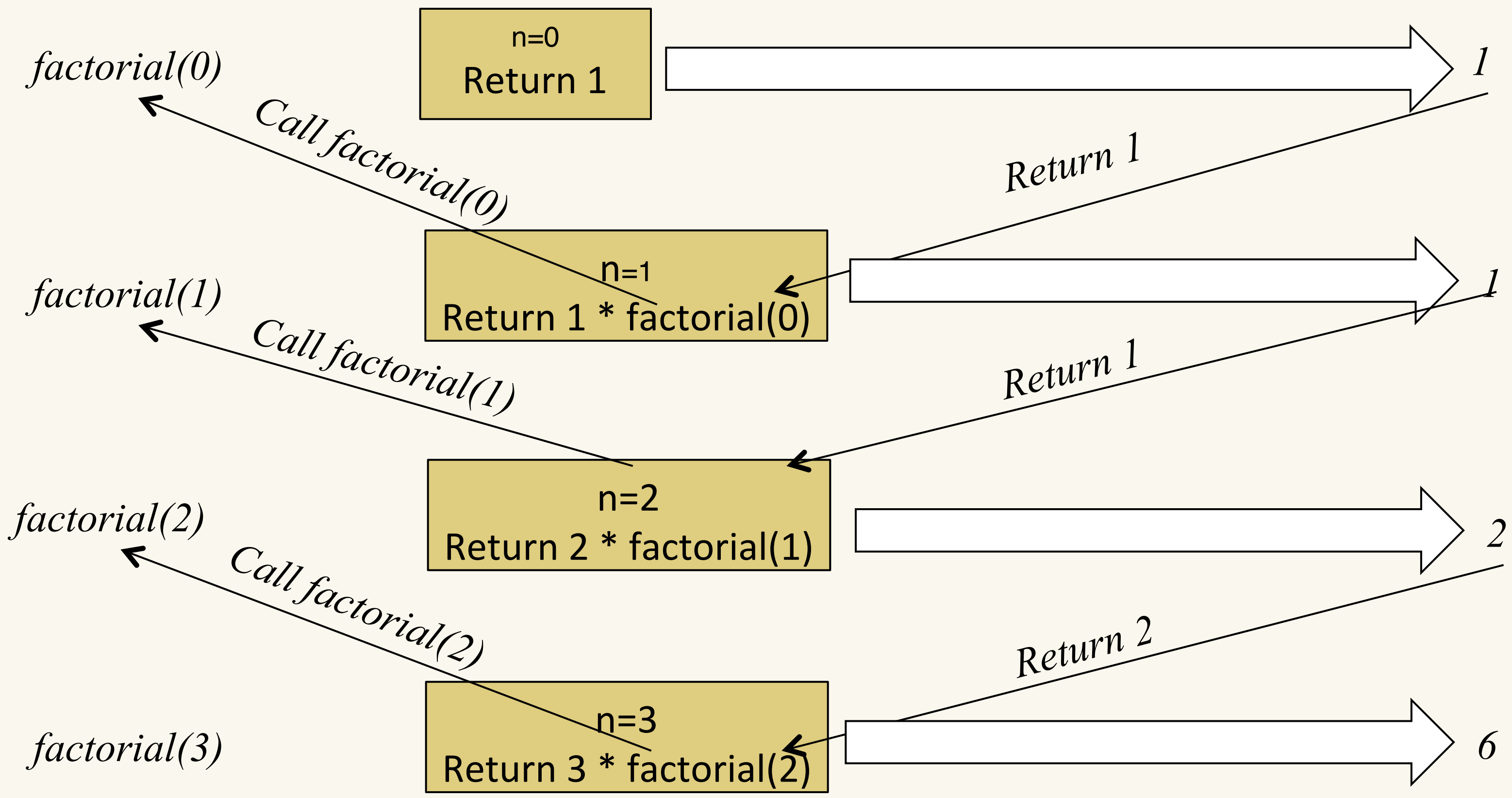
```
--> int factorial(int n)
    {
-->   if(n==0)
-->     return 1;
      else
        return (n * factorial(n - 1));
    }
```

factorial(1)

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

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

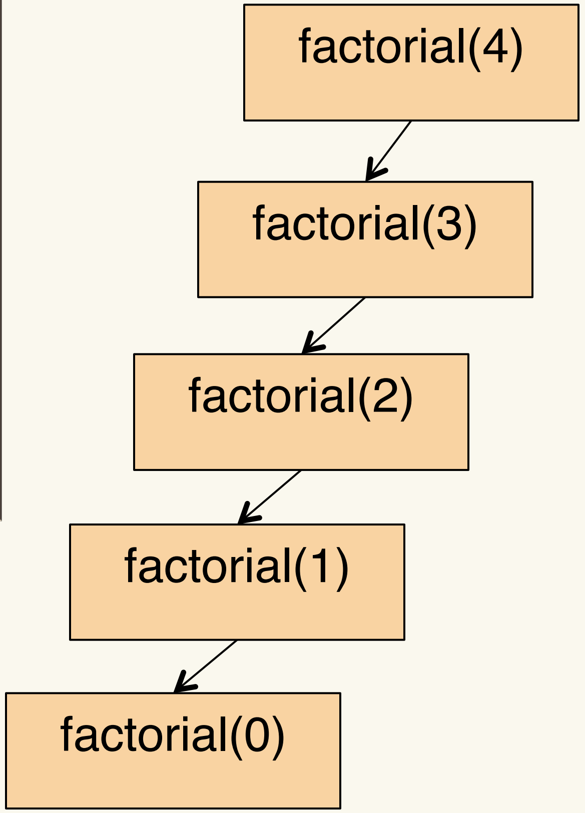
Factorial example- Work flow



Recursion Tree

```
int factorial(int n){

  if (n == 0)
    return 1;

  else
    return n * factorial(n-1);
}
```

factorial(4)



Palindrome Example

- Create a recursive function that determines whether or not a word is a palindrome. A palindrome is a word or sentence that reads the same forward as it does backward.

```
int is_palindrome(char * str, int length);
```

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Palindrome Example

```
int is_palindrome(char * str, int length){
    if (length <= 1)
        return 1;

   /* to be added */


}
```

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Palindrome Example

```
int is_palindrome(char * str, int length){
   if (length <= 1)
      return 1;

 /* is_palindrome(str + 1, length - 2) */



}
```

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Palindrome Example

```
int is_palindrome(char * str, int length){
   if (length <= 1)
      return 1;

   else
      return (str[0] == str[length - 1]) &&
            is_palindrome(str + 1, length - 2);
}
```

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Finding max example

- Using recursion, find the largest element in an array of integer values.

```
int maxRecurse(int nums[], int n);
```

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Finding max example

```
int maxRecurse(int nums[], int n){
    if (n== 1)
        return nums[0];

    /* to be added */
}
```

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Finding max example

```
int maxRecurse(int nums[], int n){
    if (n== 1)
        return nums[0];

    /* maxRecurse(nums, n-1) */
}
```

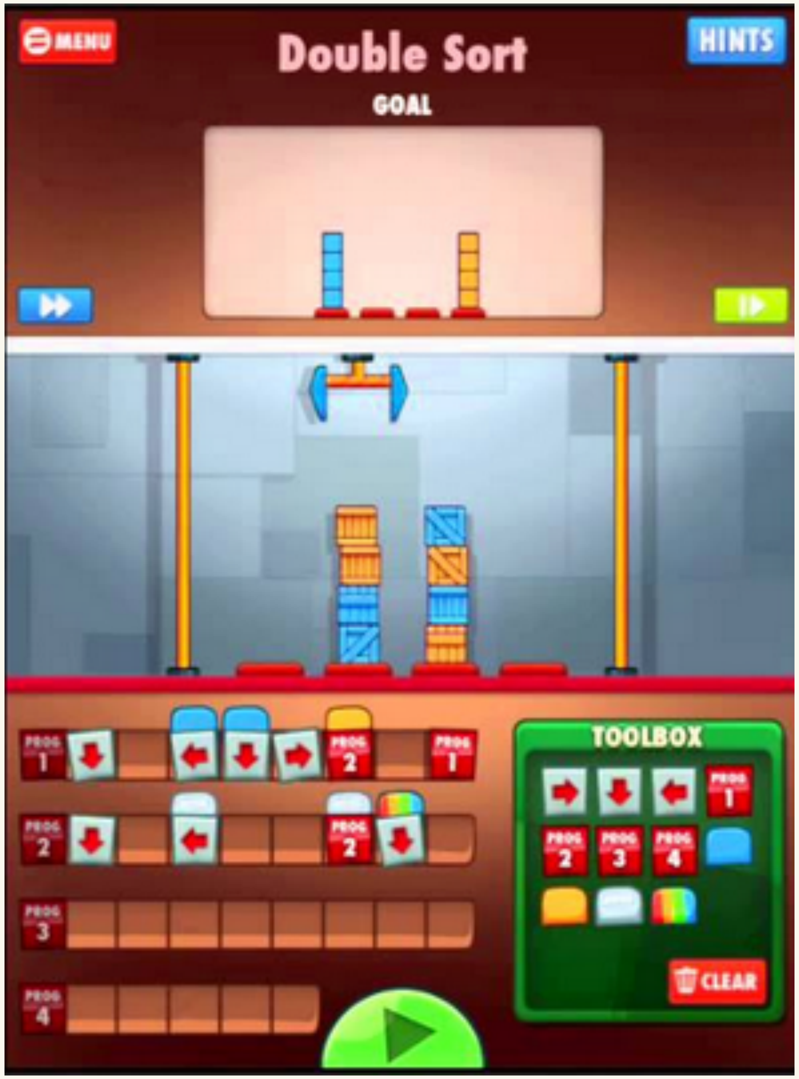
1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Finding max example

```
int maxRecurse(int nums[], int n){
    if (n== 1)
        return nums[0];

    return  max(maxRecurse(nums, n-1), nums[n-1]);
}
```

1. Recognize the base case(s) and provide solution(s).
2. Devise a strategy to split the problem into smaller versions of itself. Smaller versions **must** make progress towards the base case.
3. Without thinking about how the smaller version is computed, figure out how you can use it to solve the original problem we started off with.

Cargo-Bot
Cool Free Game that Uses Recursion on iPad



- Gameplay centers on a crane that moves and stacks a set of colored crates.
- Players write small visual programs to move the crates from an initial configuration to a goal configuration.
- The set of available instructions is quite small.
- recursion is the only mechanism for repetition.

Infinite Recursion

- We need to be very careful when writing recursive algorithms. What would happen in the following programs?

$n < 0$

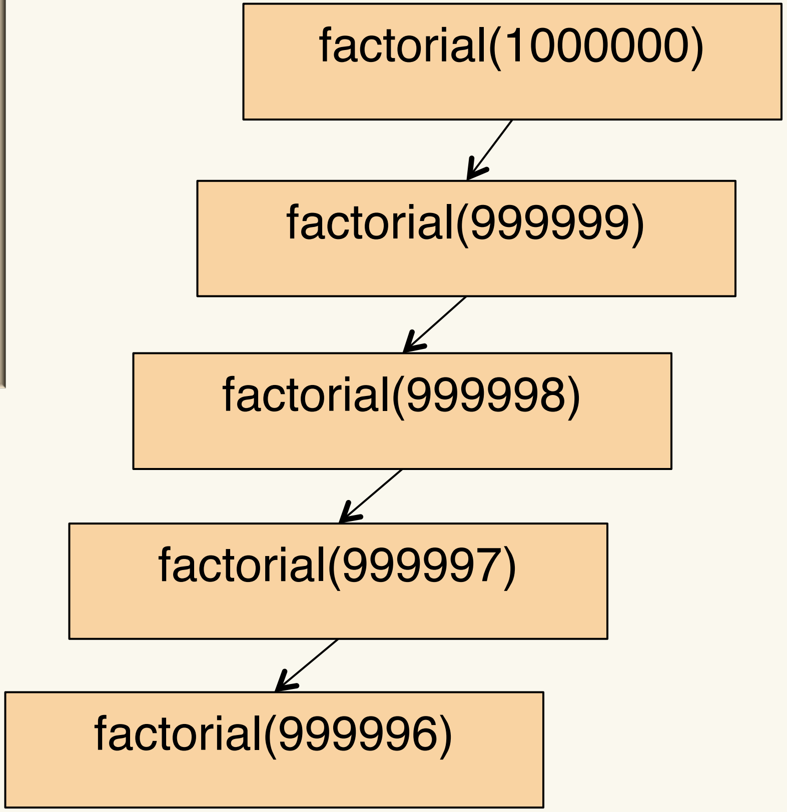
```
int factorial(int n){

  if (n == 0)
    return 1;

  else
    return n * factorial(n-1);
}
```

- Recursive calls to sub-problems must converge towards the base case(s).
  - Each call must bring the values in use closer to the halting conditions.

Stack Overflow

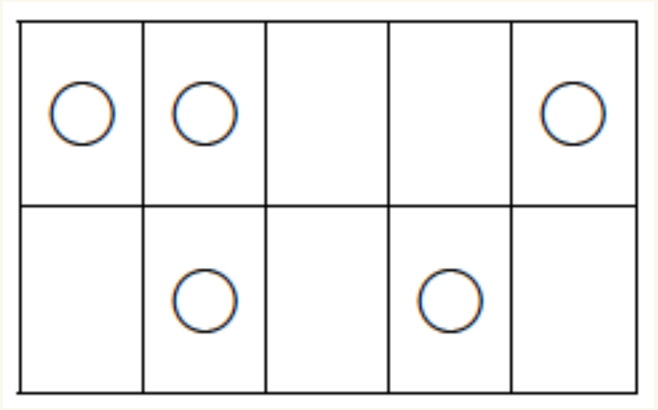
```
int factorial(int n){

  if (n == 0)
    return 1;

  else
    return n * factorial(n-1);
}
```

factorial(1000000)



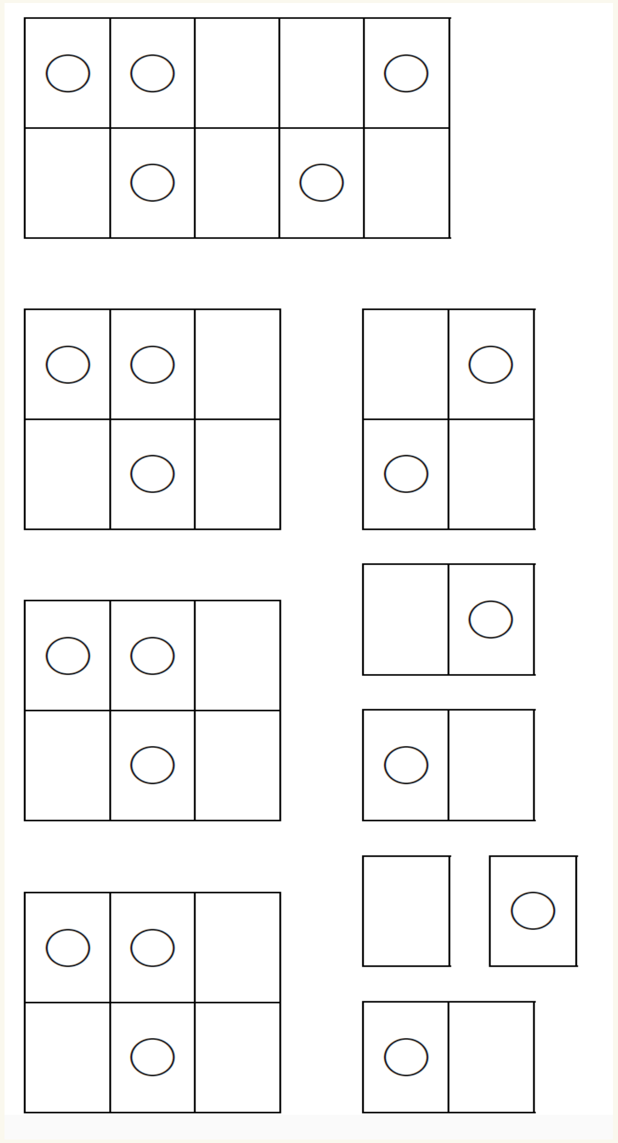
Thinking Recursively (Eat a Chocolate Bar)

Problem: You have a chocolate bar with nuts. Eat just the squares that have nuts in them. Write a recursive algorithm to solve the problem.



```
Eat Chocolate Bar(B)
   if  B is a single square then
         if  B has a nut then
               Eat it
   else
       Break the bar into two pieces
       Eat Chocolate Bar(Piece 1)
       Eat Chocolate Bar(Piece 2)
```

Thinking Recursively (Eat a Chocolate Bar)



```
Eat Chocolate Bar(B)
   if  B is a single square then
         if  B has a nut then
               Eat it
   else
       Break the bar into two pieces
       Eat Chocolate Bar(Piece 1)
       Eat Chocolate Bar(Piece 2)
```

The Fibonacci Numbers

- The Fibonacci numbers are the numbers in the sequence:
- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …
- The first two numbers are 1, and then each succeeding number can be generated by adding together the previous two numbers in the sequence. This leads to the following recursive definition:

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

n=4

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

n=3

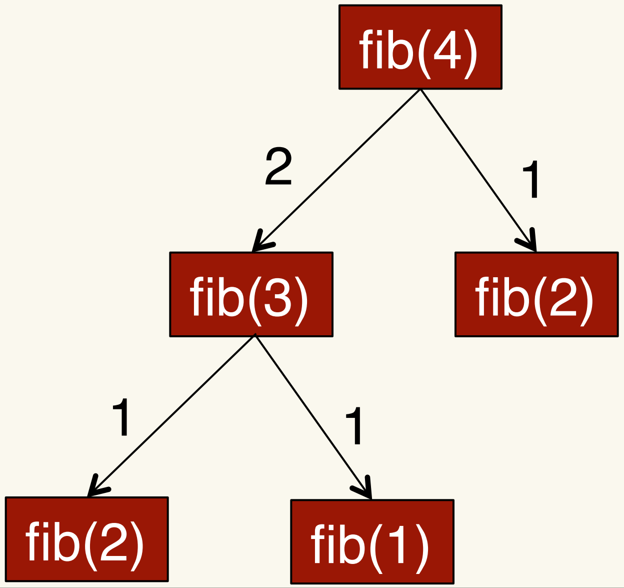
```
int fib(int n){
→if (n==1)
    return 1;
→else if (n==2)
    return 1;
→else
→   return fib(n-1)
    + fib(n-2);
}
```

n=2

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

n=1

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



n=2

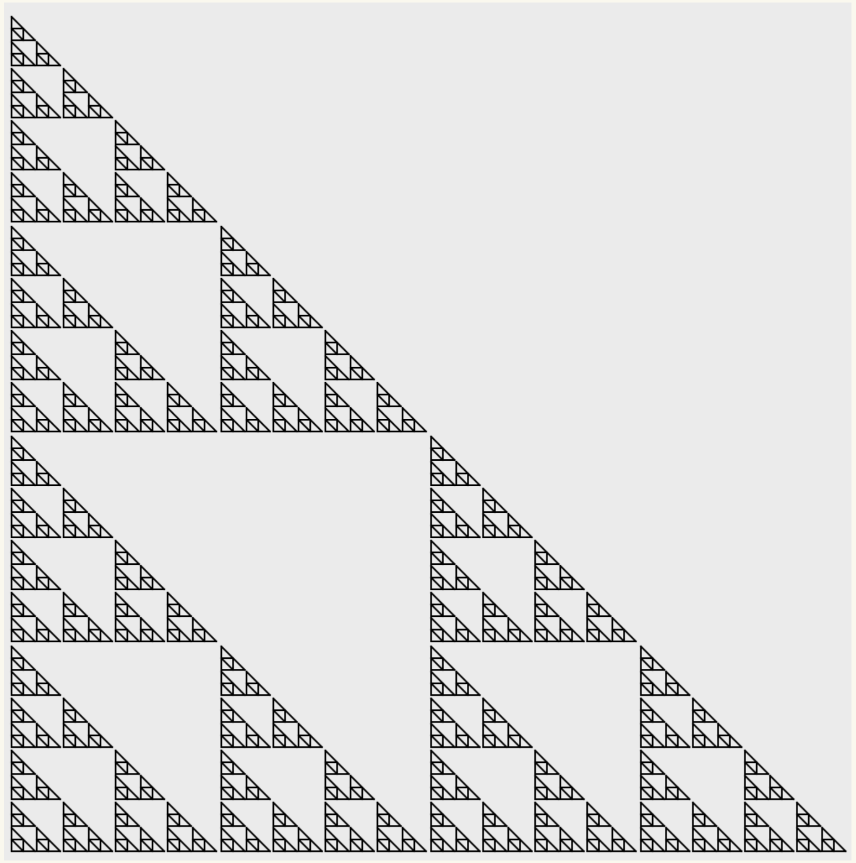
```
int fib(int n){
→if (n==1)
    return 1;
→else if (n==2)
→   return 1;
 else
    return fib(n-1)
    + fib(n-2);
}
```

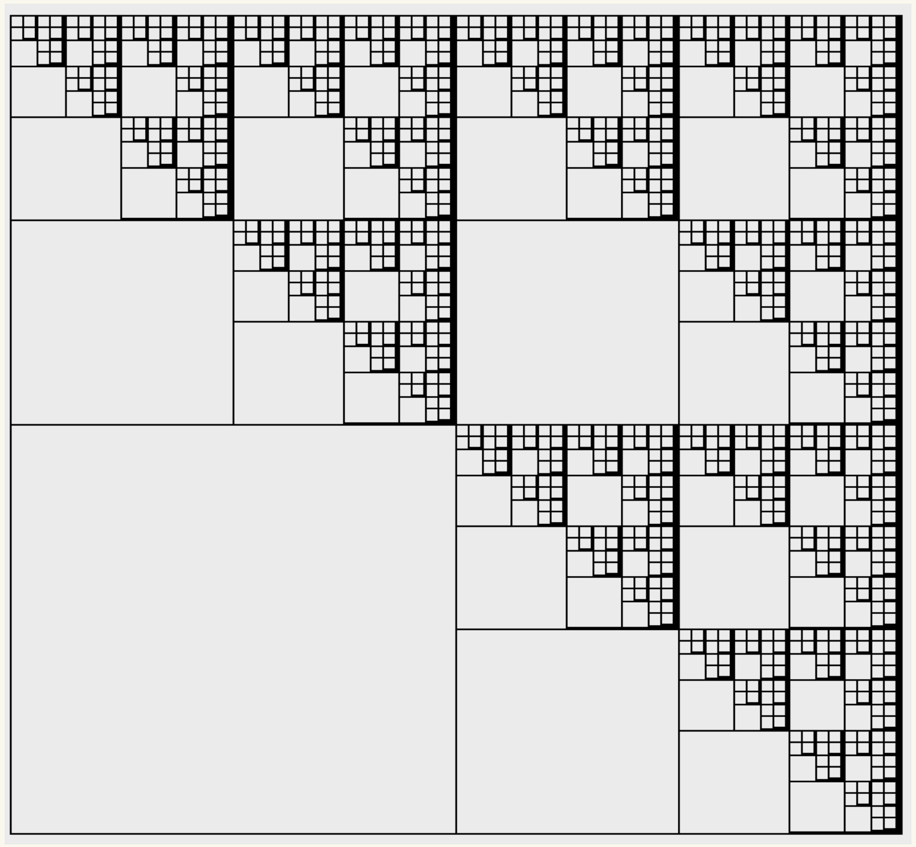
Fractals worksheet

`fancyTri (0, 0, 100);`

`fancySquare (0, 0, 100);`

Where the size of the biggest triangle or square is smaller than 10

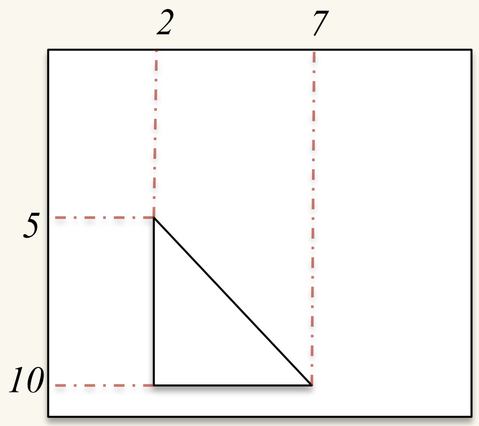




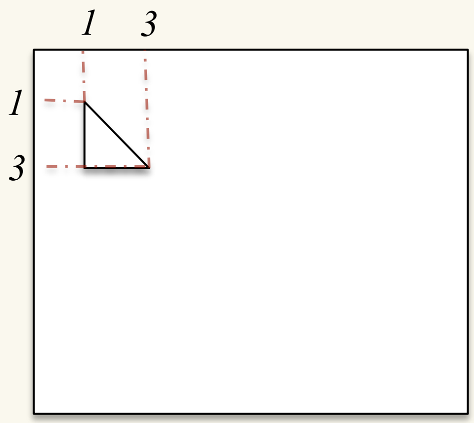
Triangle function

```
/*
 *    Purpose:  draws a right-angled, isosceles triangle on the
screen.
 *              The top left corner of the screen is mapped to (0,0).
 *    Param:    int x - x-coordinate of the upper vertex
 *    Param:    int y - y-coordinate of the upper vertex
 *    Param:    int size - length of the equal/ shorter sides
 */
void triangle(int x, int y, int size);
```

`triangle(2, 5, 5);`

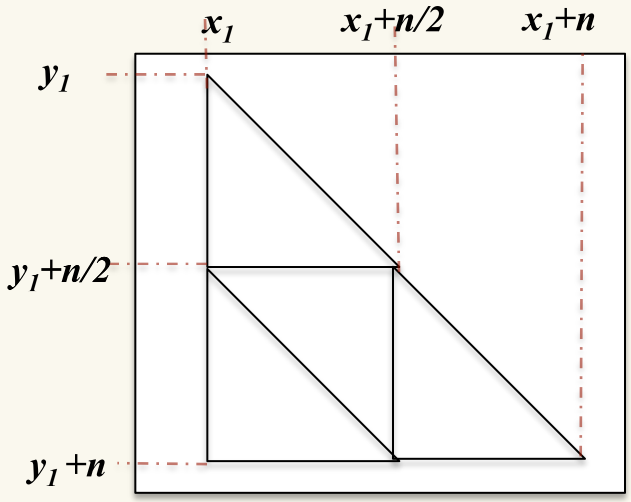


`triangle(1, 1, 2);`



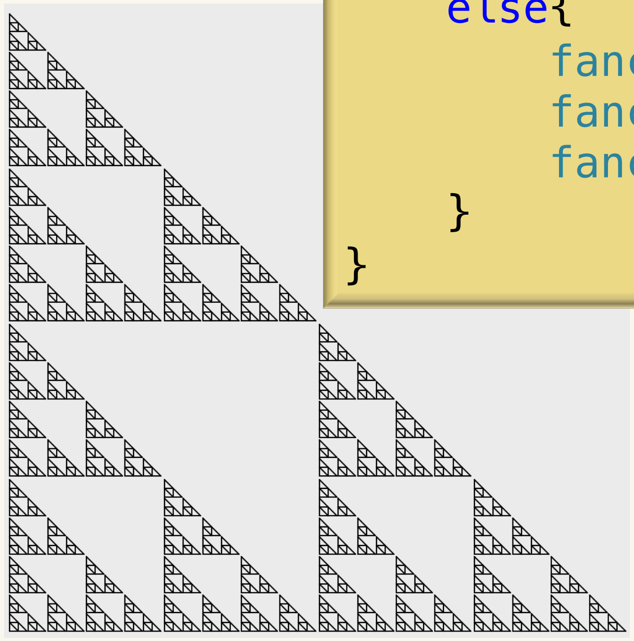
simpleTri(x_1, y_1, n)

```
/*
 *    Purpose: draws a simple picture using triangles as
 *             illustrated.
 *    Param:   int x – x-coordinate of the upper vertex
 *    Param:   int y – y-coordinate of the upper vertex
 *    Param:   int size – size of the bigger triangle
 */
void simpleTri(int x, int y, int size){
    triangle (x, y, size/2);
    triangle(x, y+size/2, size/2);
    triangle(x+size/2, y+size/2, size/2);
}
```



How is `fancyTri` constructed?

```
/*
 *    Purpose: draws a fancy picture using triangles
 *             as illustrated in the Fractals worksheet
 *    Param:   int x - x-coordinate of the upper vertex
 *    Param:   int y - y-coordinate of the upper vertex
 *    Param:   int size - size of the bigger triangle
 */
void fancyTri(int x, int y, int size){
    if(size<10)
        triangle (x, y, size);
    else{
        fancyTri(x, y, size/2);
        fancyTri(x, y+size/2, size/2);
        fancyTri(x+size/2, y+size/2, size/2);
    }
}
```



Clicker question

- How many nodes does the recursion tree of `fancyTri(0,0,20)` have?

A: 4

B: 5

C: 12

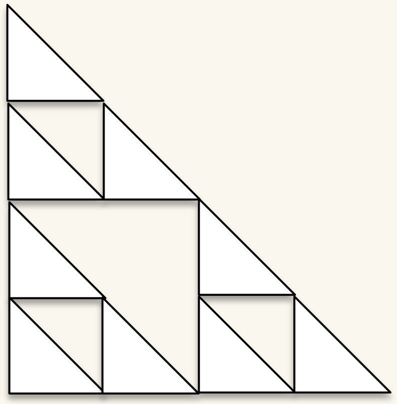
D: 13

E: 21

```
/*
 *    Purpose: draws a fancy picture using triangles
 *             as illustrated in the Fractals worksheet
 *    Param:   int x – x-coordinate of the upper vertex
 *    Param:   int y – y-coordinate of the upper vertex
 *    Param:   int size – size of the bigger triangle
 */
void fancyTri(int x, int y, int size){
    if(size<10)
        triangle (x, y, size);
    else{
        fancyTri(x, y, size/2);
        fancyTri(x, y+size/2, size/2);
        fancyTri(x+size/2, y+size/2, size/2);
    }
}
```

Clicker question (answer)

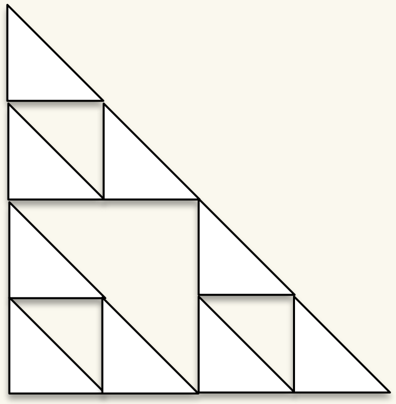
```
void fancyTri(int x, int y, int size){
    if(size<10)
        triangle (x, y, size);
    else{
        fancyTri(x, y, size/2);
        fancyTri(x, y+size/2, size/2);
        fancyTri(x+size/2, y+size/2, size/2);
    }
}
```

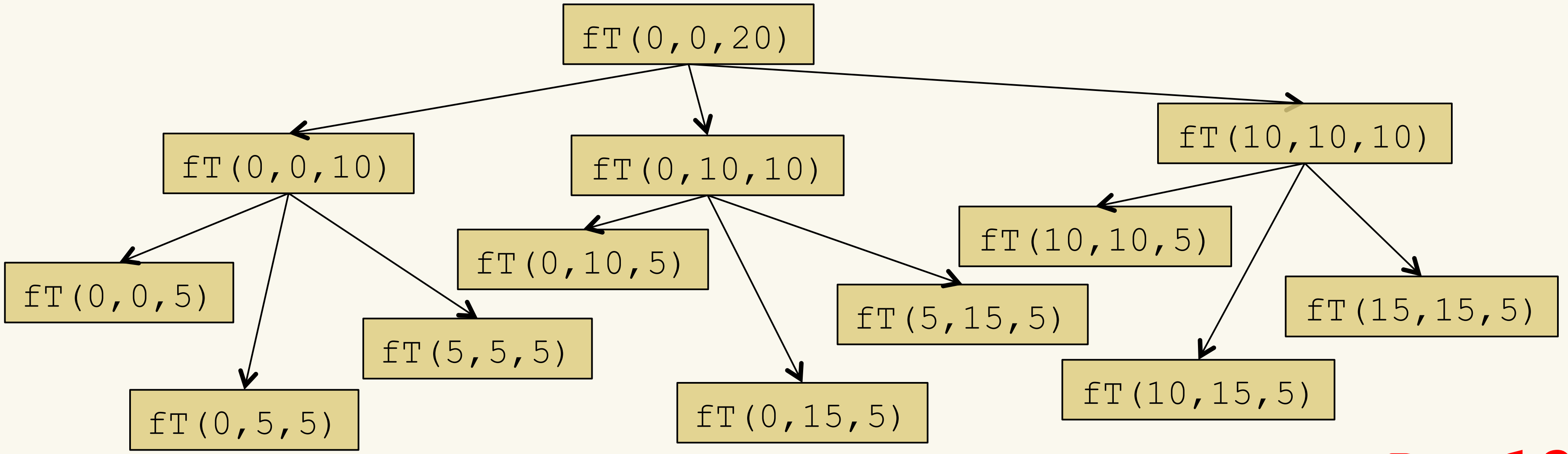


$$\begin{tikzcd}
 & & & & \text{fT(0,0,20)} \arrow[dlll] \arrow[d] \arrow[drrr] & & & & \\
 & \text{fT(0,0,10)} \arrow[dl] \arrow[d] \arrow[dr] & & & \text{fT(0,10,10)} \arrow[dl] \arrow[d] \arrow[dr] & & & \text{fT(10,10,10)} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{fT(0,0,5)} & \text{fT(0,5,5)} & \text{fT(5,5,5)} & \text{fT(0,10,5)} & \text{fT(0,15,5)} & \text{fT(5,15,5)} & \text{fT(10,10,5)} & \text{fT(10,15,5)} & \text{fT(15,15,5)}
\end{tikzcd}$$

Clicker question (answer)

```
void fancyTri(int x, int y, int size){
    if(size<10)
        triangle (x, y, size);
    else{
        fancyTri(x, y, size/2);
        fancyTri(x, y+size/2, size/2);
        fancyTri(x+size/2, y+size/2, size/2);
    }
}
```





D: 13

Can We Write Factorial Iteratively?

Recursive

```
int factorial(int n){

  if (n == 0)
    return 1;

  else
    return n *
    factorial(n-1);
}
```

Iterative

```
int factorial_iterative(int n){
  int factorial=1;

  for (int i = 1; i <= n; i++)
    factorial = factorial * i;

  return factorial;
}
```

Can We Write Fibonacci Iteratively?

Recursive

```
int fib(int n){
  if (n==1)
    return 1;

  else if(n==2)
    return 1;

  else
    return fib(n-1)
     + fib(n-2);
}
```

Iterative

```
int fib_iterative(int n) {
  int answer=1;
  int answerminus1 = 1;
  int answerminus2 = 1;
  for (int i = 3; i <= n; i++) {
    answer = answerminus1 +
             answerminus2;
    answerminus2 = answerminus1;
    answerminus1 = answer;
  }
  return answer;
}
```

Which One is Better?

Recursive

```
int factorial(int n){

  if (n == 0)
    return 1;

  else
    return n *
    factorial(n-1);
}
```

Iterative

```
int factorial_iterative(int n){
  int factorial=1;

  for (int i = 1; i <= n; i++)
    factorial = factorial * i;

  return factorial;
}
```

A: The recursive version

B: The iterative version

C: The two are as good as each other

D: None of the above

Appreciating Recursion

- 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
```

Appreciating Recursion Even More

- Let’s study a more complex example that would be challenging to solve without recursion.

- **Problem:** Design a method that prints out all permutations of a string that does not contain repeated characters.
  - A permutation is simply a rearrangement of the letters in the string.
  - For example, the string "abc" has six permutations.

abc	bac	cab
acb	bca	cba

Find the Base Case(s)

- What would be the simplest string(s)?
  - The empty string has a single permutation: itself

```
permutations(s):
   if s is empty:
         just return s
```

Simplify the Problem

- Suppose the input is "abc"
  - How can we generate all permutations that start with the letter 'a'?
  - Would assuming that I have all permutations of "bc" help?

$$\begin{tikzcd} \boxed{\begin{array}{c} \mathsf{bc} \\ \mathsf{cb} \end{array}} \arrow[r] & \boxed{\begin{array}{c} \mathsf{\mathbf{a}bc} \\ \mathsf{\mathbf{a}cb} \end{array}} \end{tikzcd}$$

  - How can we generate all permutations that start with the letter 'b'?
  - Would assuming that I have all permutations of "ac" help?

$$\begin{tikzcd} \boxed{\begin{array}{c} \mathsf{ac} \\ \mathsf{ca} \end{array}} \arrow[r] & \boxed{\begin{array}{c} \mathsf{\mathbf{b}ac} \\ \mathsf{\mathbf{b}ca} \end{array}} \end{tikzcd}$$

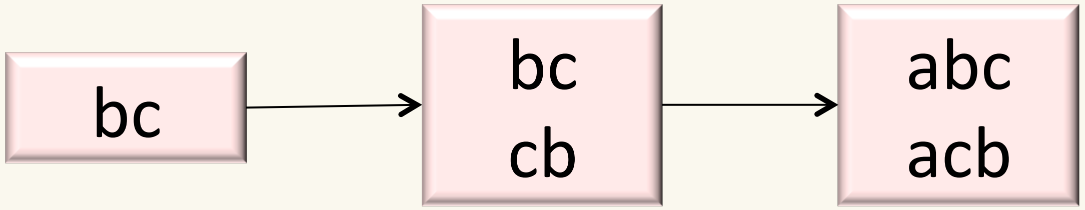
Write Algorithm (in English)

```
permutations(s):
 if s is empty:
       just return s
 else:
    – loop through all character positions
        • form a shorter word by removing the ith char
        • generate all permutations of the simpler word
        • add the removed character to the front of each
          permutation of the simpler word
    – return all permutations
```

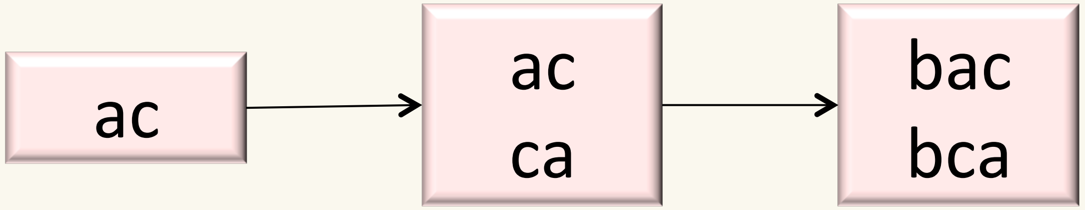
Test Your Algorithm

s = "abc"

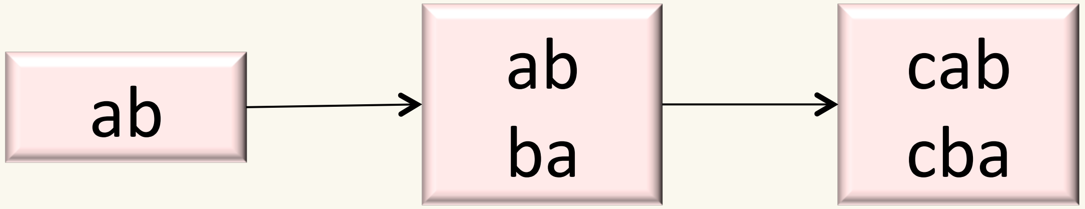
loop through all character positions: a

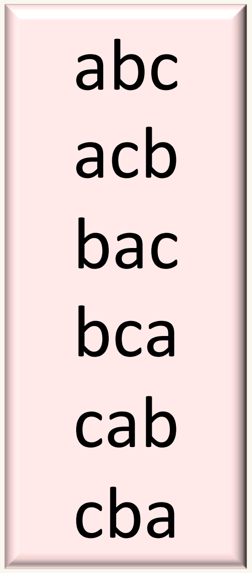


loop through all character positions: b



loop through all character positions: c

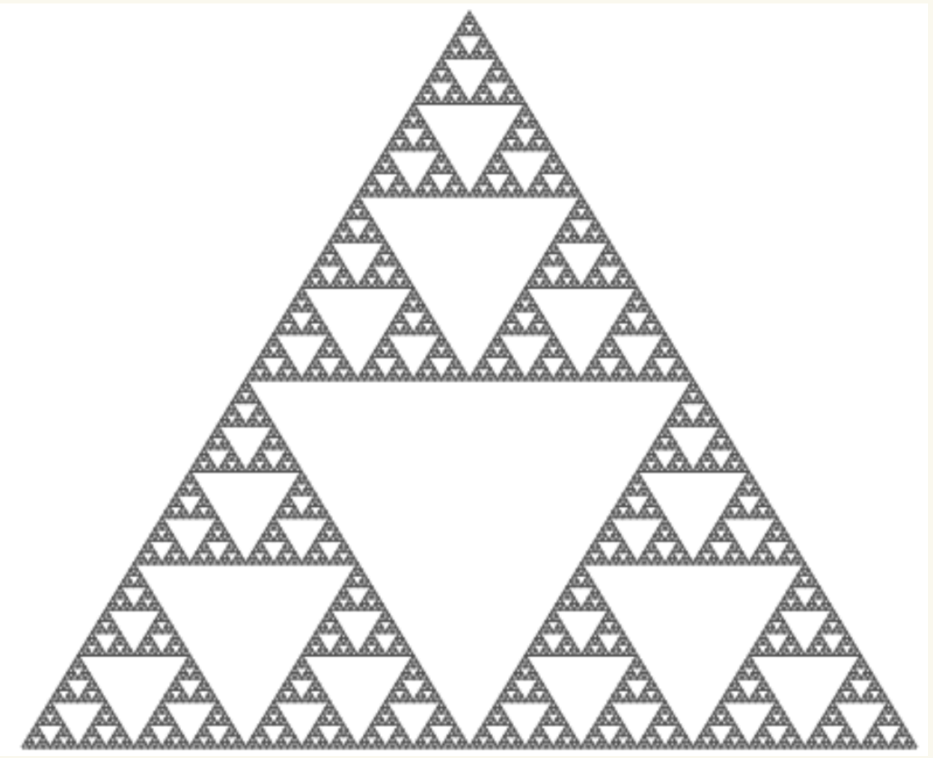


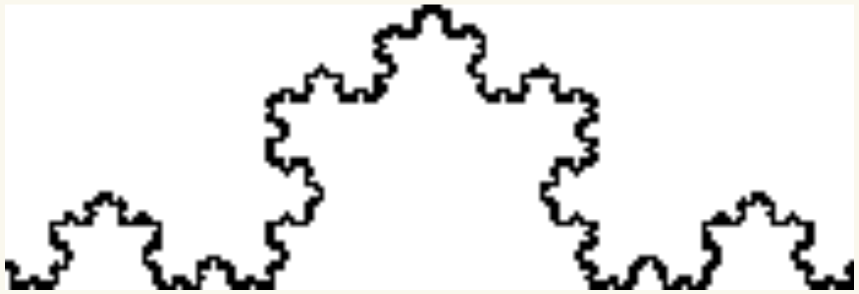


Classic Problems that can be Solved Recursively (Fractals)

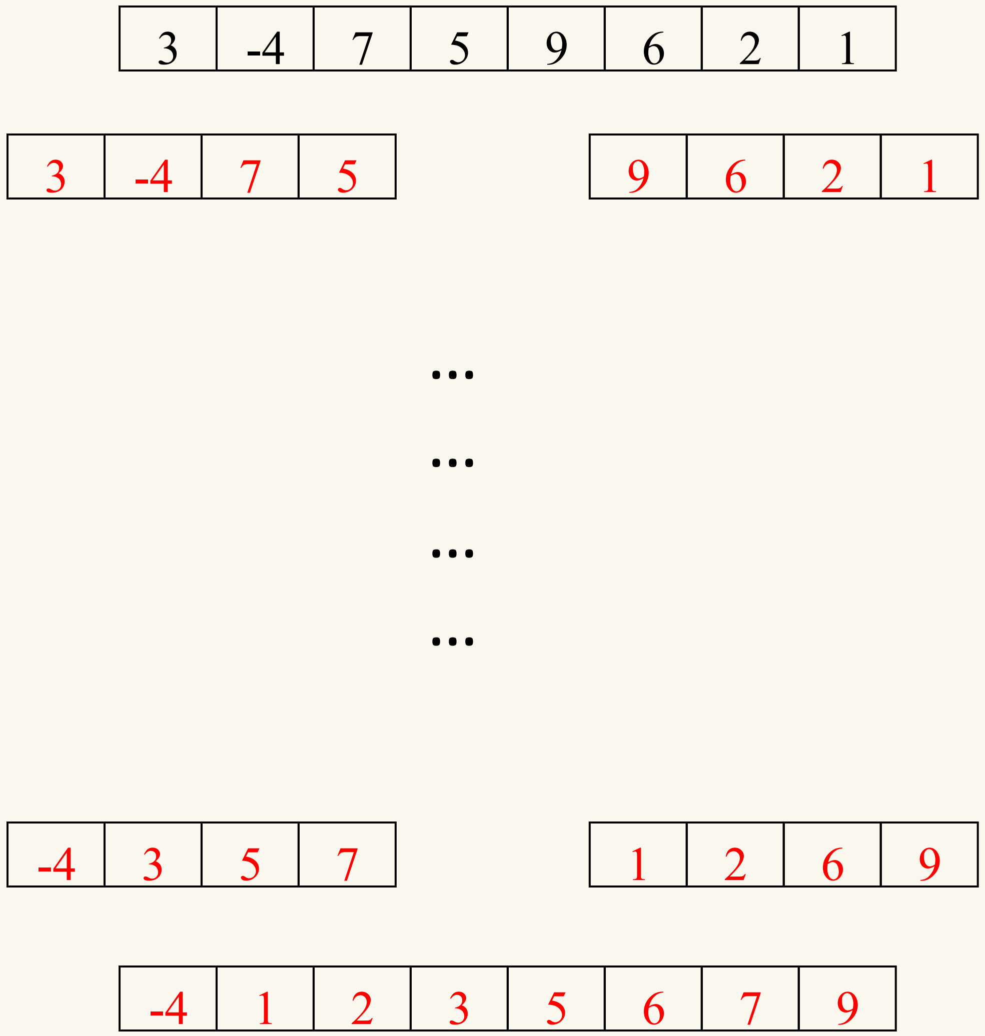
Sierpinski triangle

Koch Curve



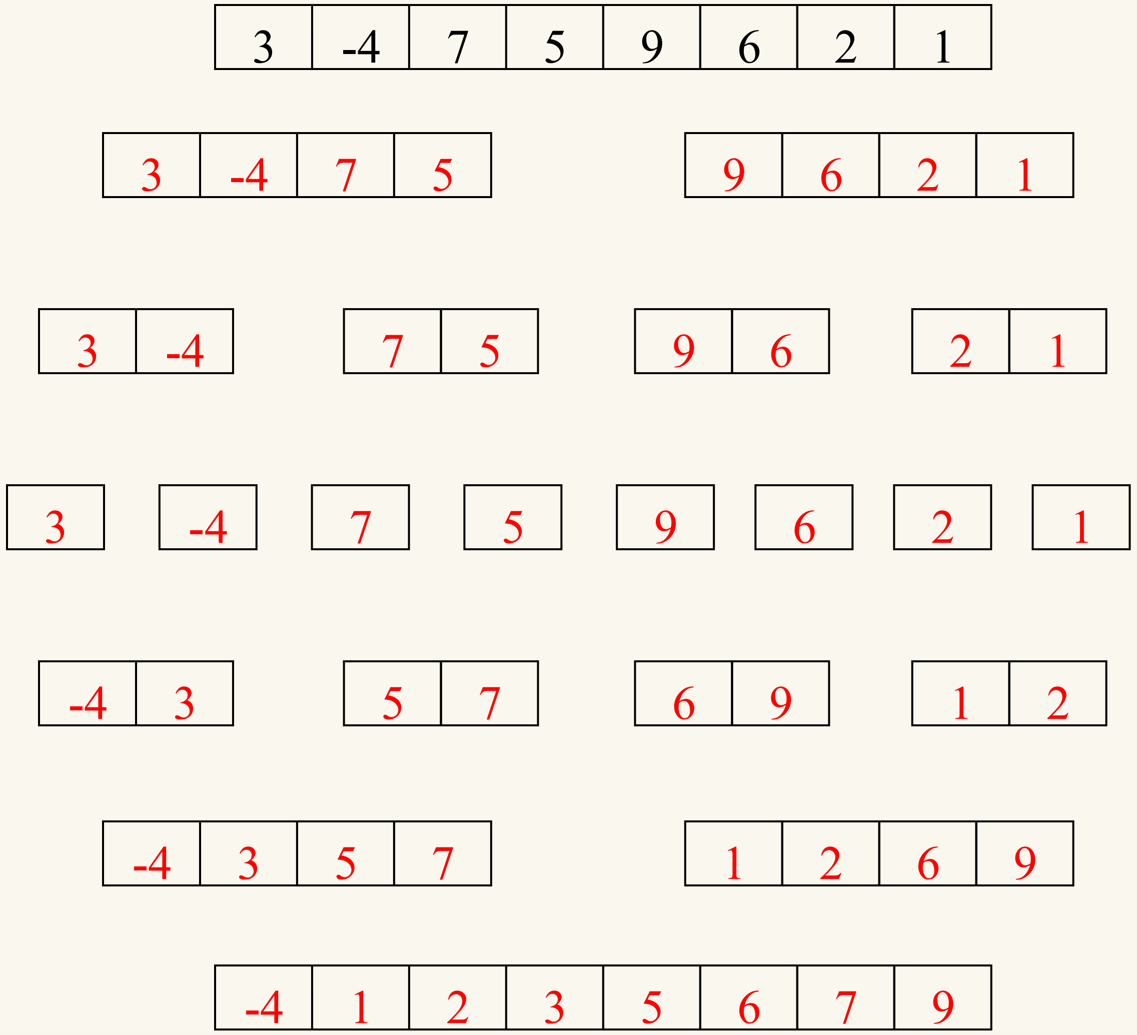


Classic Problems that can be Solved Recursively (Sorting)



```
mergesort(array)
 if size == 1
    return
else
    mergesort(leftHalf)
    mergesort(rightHalf)
    merge(leftHalf, rightHalf)
```

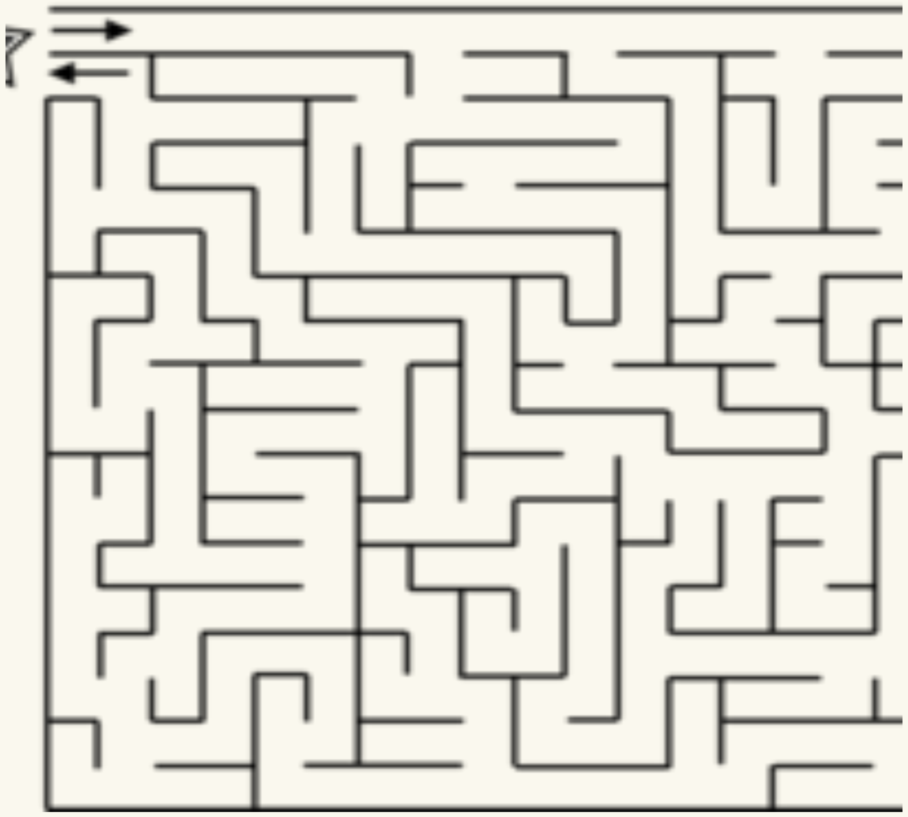
Classic Problems that can be Solved Recursively (Sorting)



```
mergesort(array)
 if size == 1
   return
else
   mergesort(leftHalf)
   mergesort(rightHalf)
   merge(leftHalf, rightHalf)
```

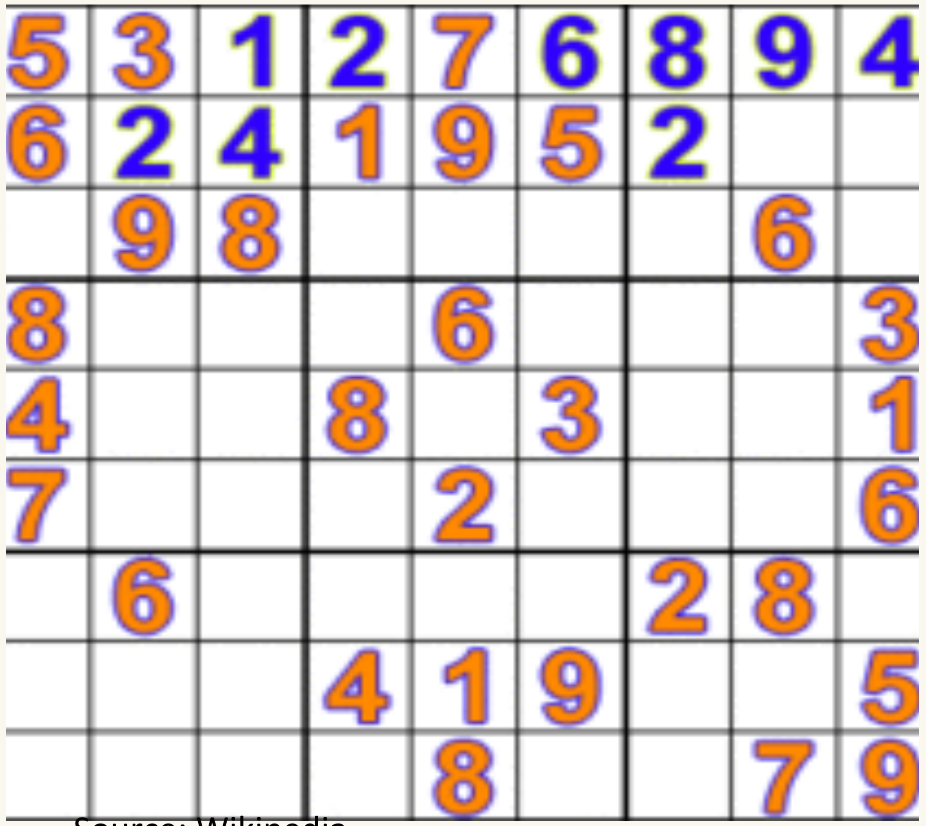
Classic Problems that can be Solved Recursively (Backtracking)

Maze



Source: Wikipedia
http://en.wikipedia.org/wiki/Maze

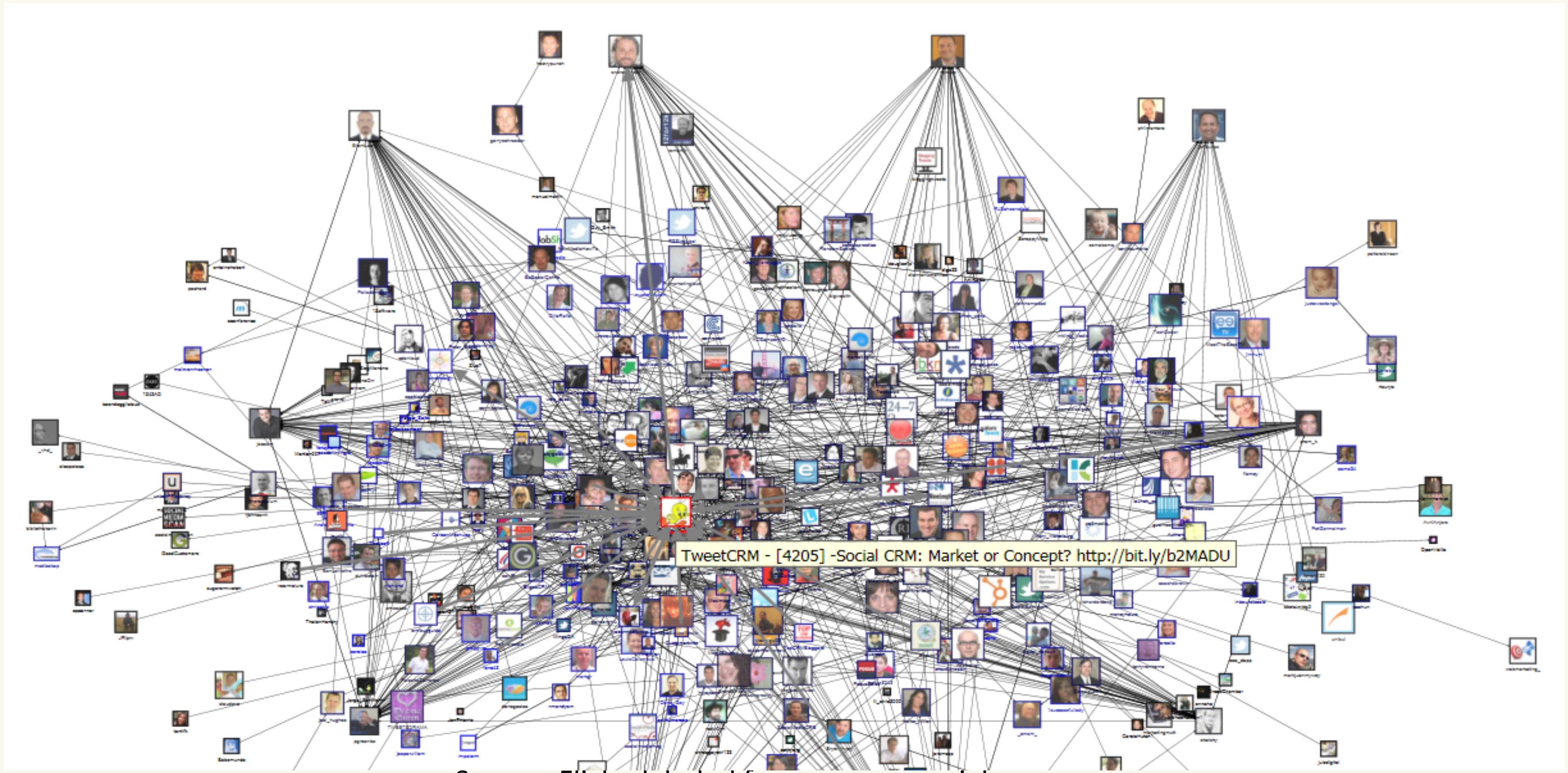
Sudoku



5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Source: Wikipedia
http://en.wikipedia.org/wiki/Backtracking

Classic Problems that can be Solved Recursively (operations on data structures)


TweetCRM - [4205] -Social CRM: Market or Concept? http://bit.ly/b2MADU

Source: Flickr, labeled for noncommercial reuse

Challenging Activity

- **Problem:** Try to write code that prints out all permutations of a string using iteration.

Which One is More Powerful?

- Iteration and recursion are equally powerful and can solve the same problems.
  - Simulating iteration with recursion is easy.

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

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

```
void recDoFoo(int i, int n){
  if (i >= n)
    return;

  else{
    doFoo(i);
    recDoFoo(i + 1, n);
  }
}
```

Which One is More Powerful?

- Iteration and recursion are equally powerful and can solve the same problems.
  - Simulating recursion with iteration is harder and beyond the scope of this course
  - But here is the general idea
    - Simulate the call stack by creating a stack yourself.
    - This can be tricky, so it is better to let the computer do this for you.

Iterative vs. Recursive Approaches

– Use iteration when
  - runtime is extremely important (even small differences)
  - your algorithm doesn't use a lot of extra space and you're happy with the runtime.

– Use recursion since
  - recursive programs can be relatively simpler to write, analyze, and understand than iterative versions.
  - recursive programs can benefit from the call stack in more complex problems that run slower and/or require more space.

Be Very Careful When You Use Recursion

- Poorly designed recursive procedures can use a lot of memory and be very slow.
  - It is is much easier to shoot yourself in the foot without noticing when you use recursion.

Can We Write Fibonacci Iteratively?

Recursive

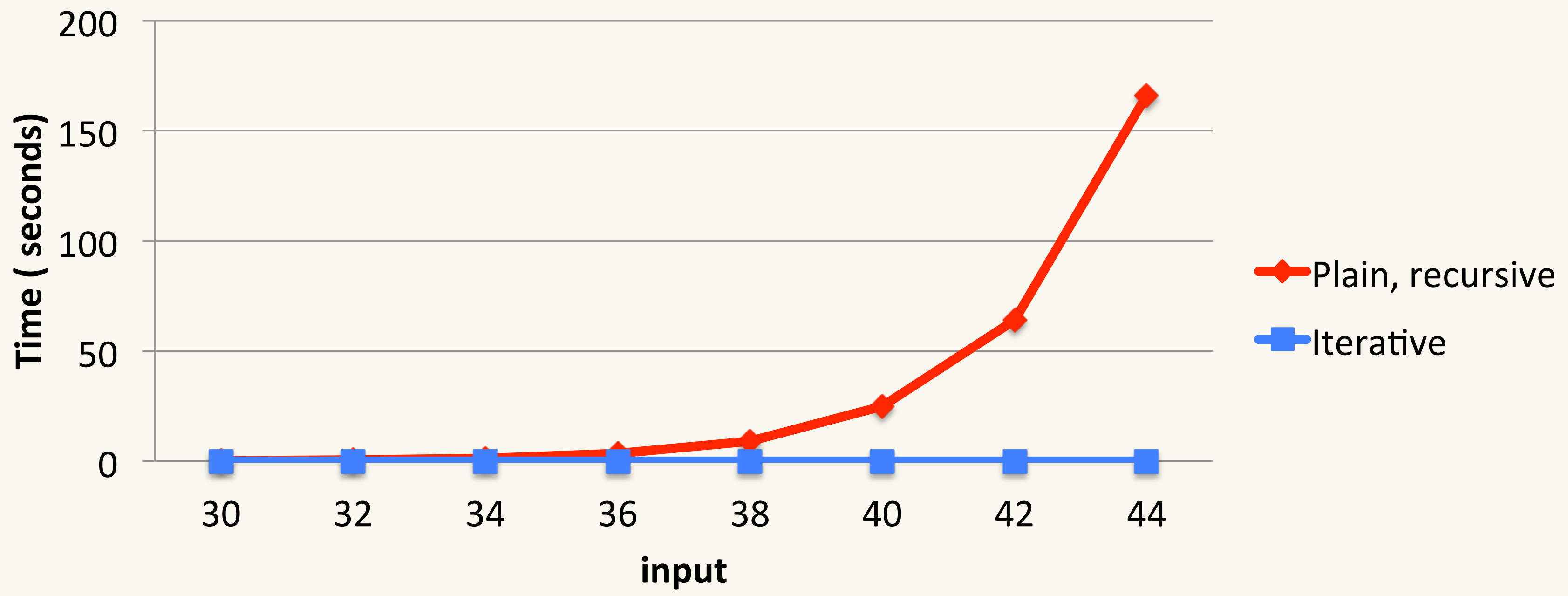
```
int fib(int n){
  if (n==1)
    return 1;
  else if(n==2)
    return 1;
  else
    return fib(n-1) +
    fib(n-2);
}
```

Iterative

```
int fib_iterative(int n) {
  int answer=1;
  int answerminus1 = 1;
  int answerminus2 = 1;
  for (int i = 3; i <= n; i++) {
    answer = answerminus1 +
             answerminus2;
    answerminus2 = answerminus1;
    answerminus1 = answer;
  }
  return answer;
}
```

Let’s run them

Runtime Comparison of Different Fibonacci Implementations (in seconds)



	30	32	34	36	38	40	42	44
Plain, recursive	0.19	0.5	1.37	3.6	9	25	64	166
Iterative	0.0004	0.00048	0.00051	0.00052	0.00055	0.00059	0.00062	0.00065

Recursion and efficiency

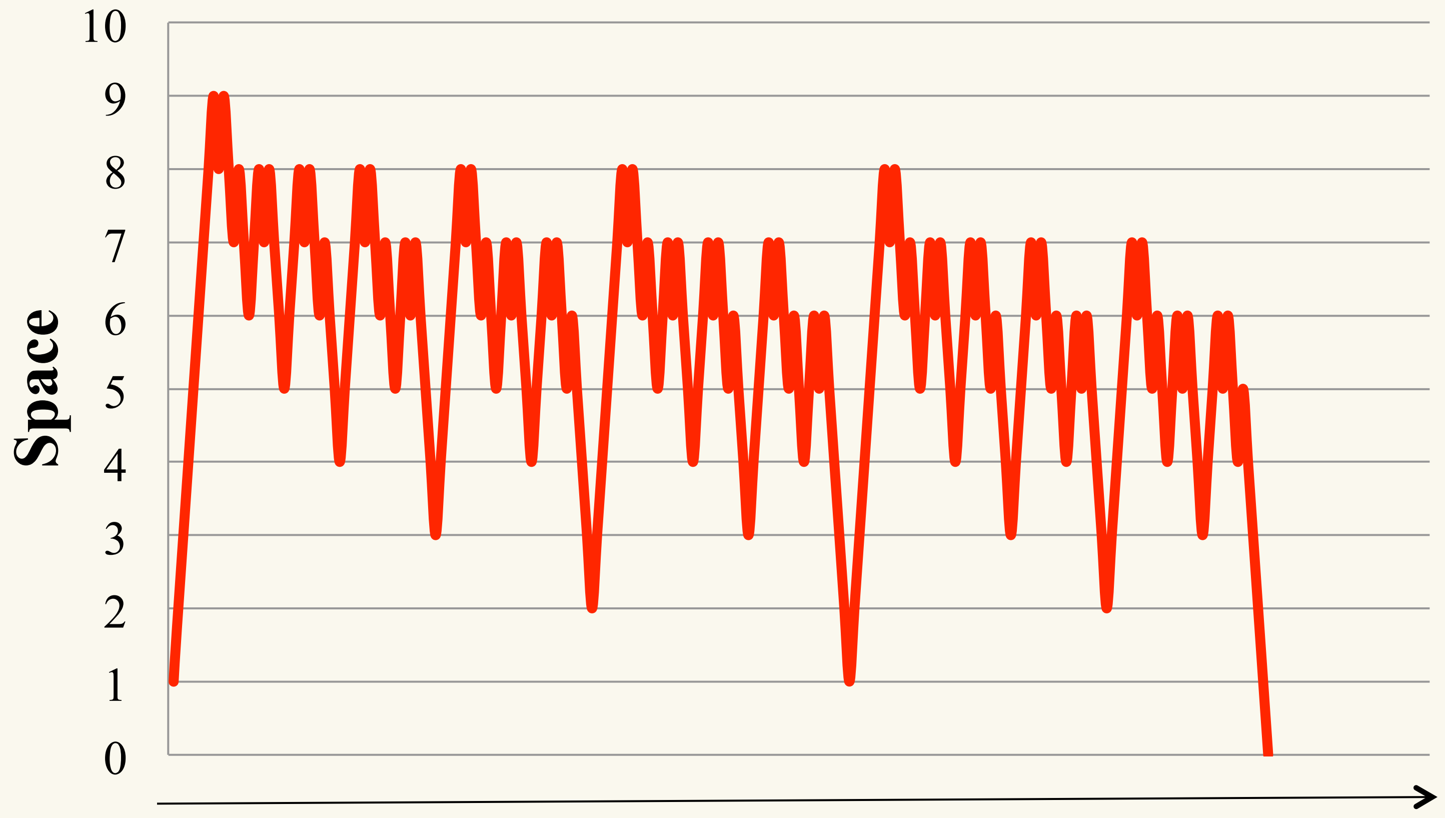
- Each function invocation generates an activation record that holds the values of arguments and local variables for that invocation. These activation records are stored in an area of memory called the run-time stack. The diagram below shows the state of the run-time stack when a call to `fib(4)` is made from main.

			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

- The height of `fib`'s recursion tree plus 1 is the maximum number of `fib` activation records on the run-time stack at any given time—in this case 3. Memory for the runtime stack is limited. If we attempt to generate more activation records than can be stored on the run-time stack, a **stack overflow** occurs and the program will crash.

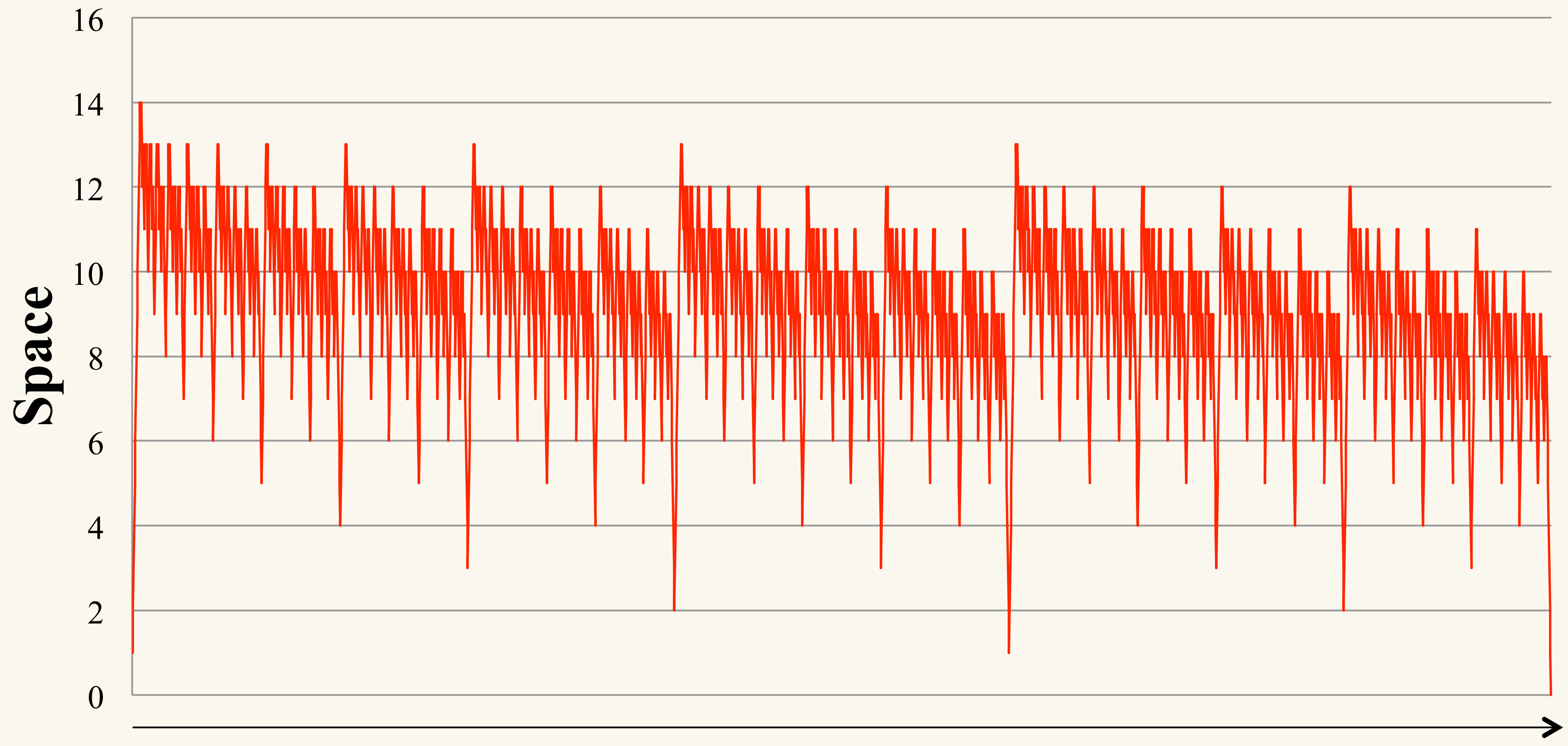
Plain Recursive Fibonacci

Stack space used in computing `fib(10)`



Plain Recursive Fibonacci

Stack space used in computing `fib(15)`



CPSC 259 Administrative Notes

- Starting Lab 4 – Week 1 on Monday
- Midterm Friday, Nov 13
- Unfortunately, we've had a few plagiarism cases that I'm following up on.
  - Course policy on plagiarism is stated on the course website

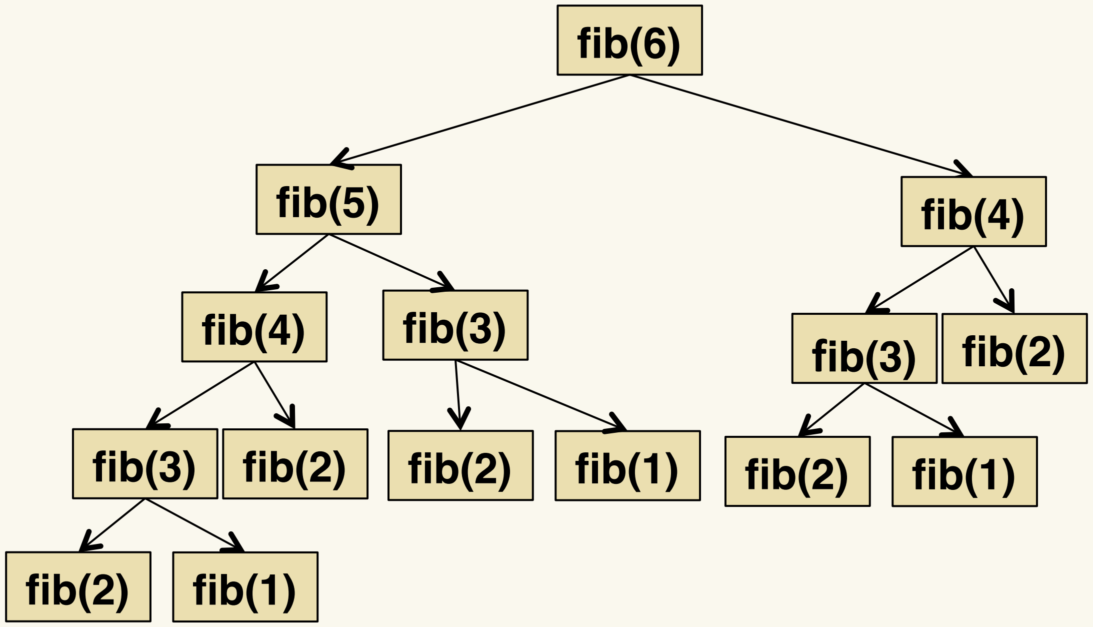
Thank you for your feedback

- Great to know that most people like how the course is progressing.
  - Average response to “Overall, the instructor was an effective teacher” was ~4.8

- Some common dislikes
  - 8 am lectures!
  - Textbook
    - Any recommendations?
    - Stanford's tutorial notes for C programming.
  - Quizzes
  - Midterm was a bit long

Redundant Computation

- Do we really need to compute `fib(3)` multiple times?



(1) Redundant computation
(2) Requires a lot of memory

Can we do better?

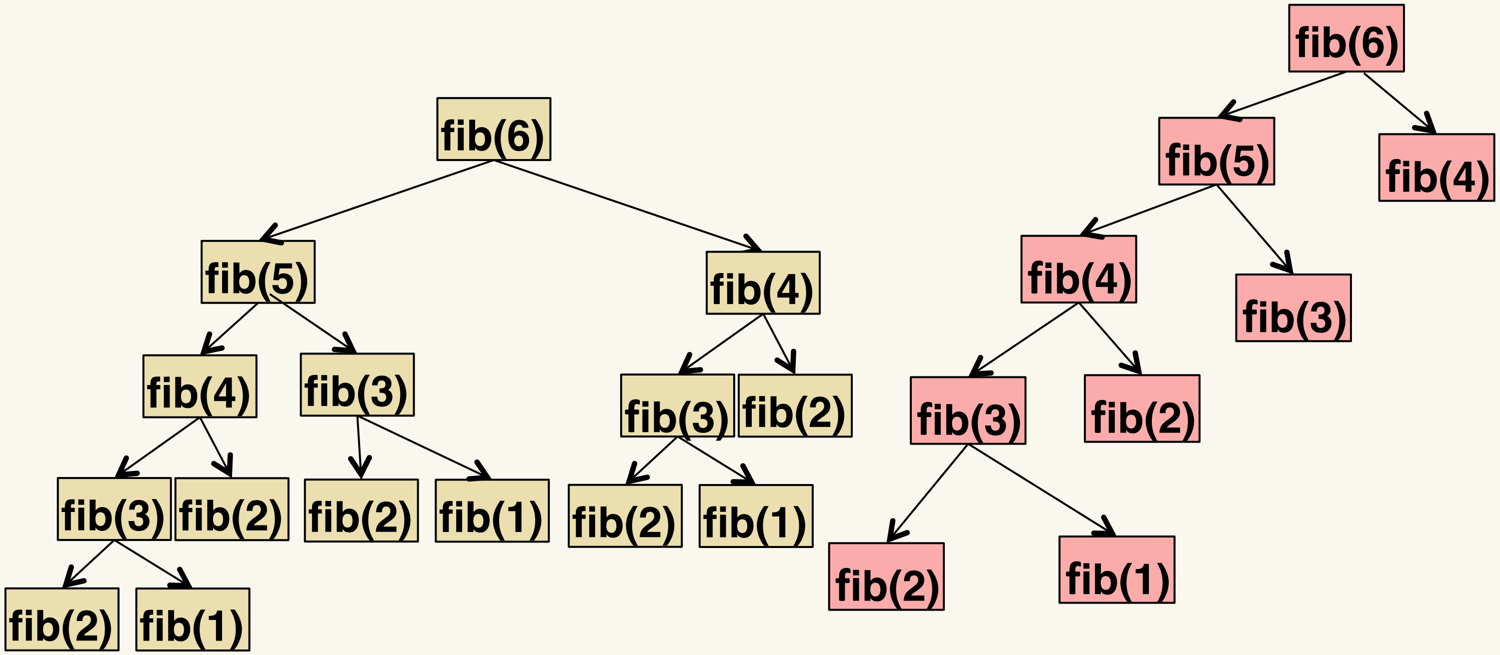
Be Very Careful When You Use Recursion

- Poorly designed recursive procedures can use a lot of memory and be very slow.
  - It is is much easier to shoot yourself in the foot without noticing when you use recursion.

- Well designed recursive procedures are almost (constant factor difference) as fast as iterative procedures
  - Memoization can make recursive functions run faster.

Recursion -- Memoization

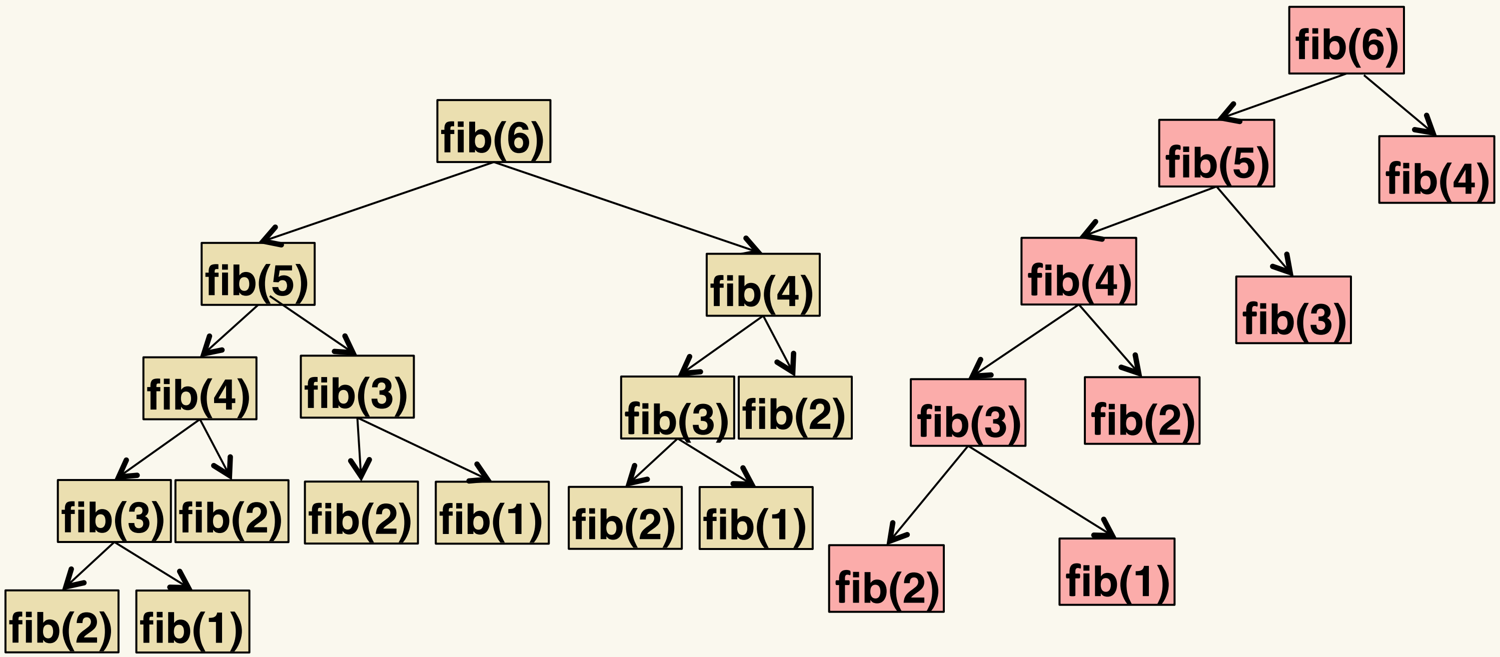
- Memoization
  - After computing a solution, store it in a table before returning. (Leave a "memo" to yourself.)
  - At start of the function, check if you've solved this case before. If so, return the already calculated solution.



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

Recursion -- Memoization

- Memoization
  - After computing a solution, store it in a table before returning. (Leave a "memo" to yourself.)
  - At start of the function, check if you've solved this case before. If so, return the already calculated solution.



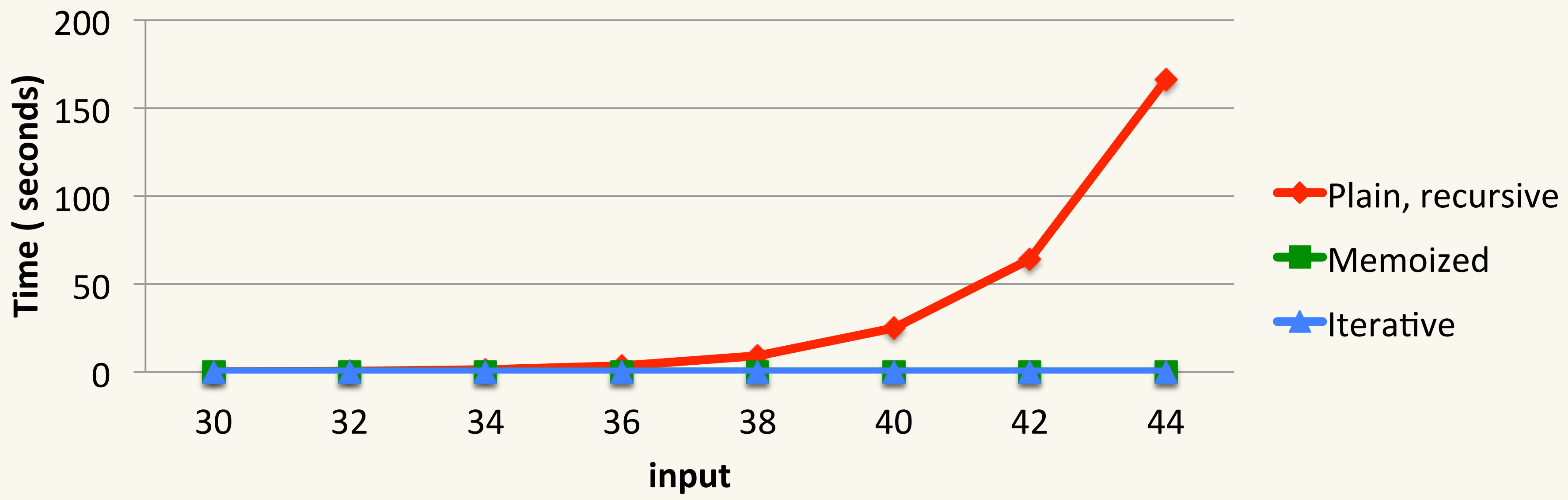
n	F_n
6	8
5	5
4	3
3	2
2	1
1	1

Memoizing Fib

```
int fibMem(int n)
{
    int* fibTable = (int*) calloc(sizeof(int) , MAX);
    return fibMemoHelper(n, fibTable);

}
```

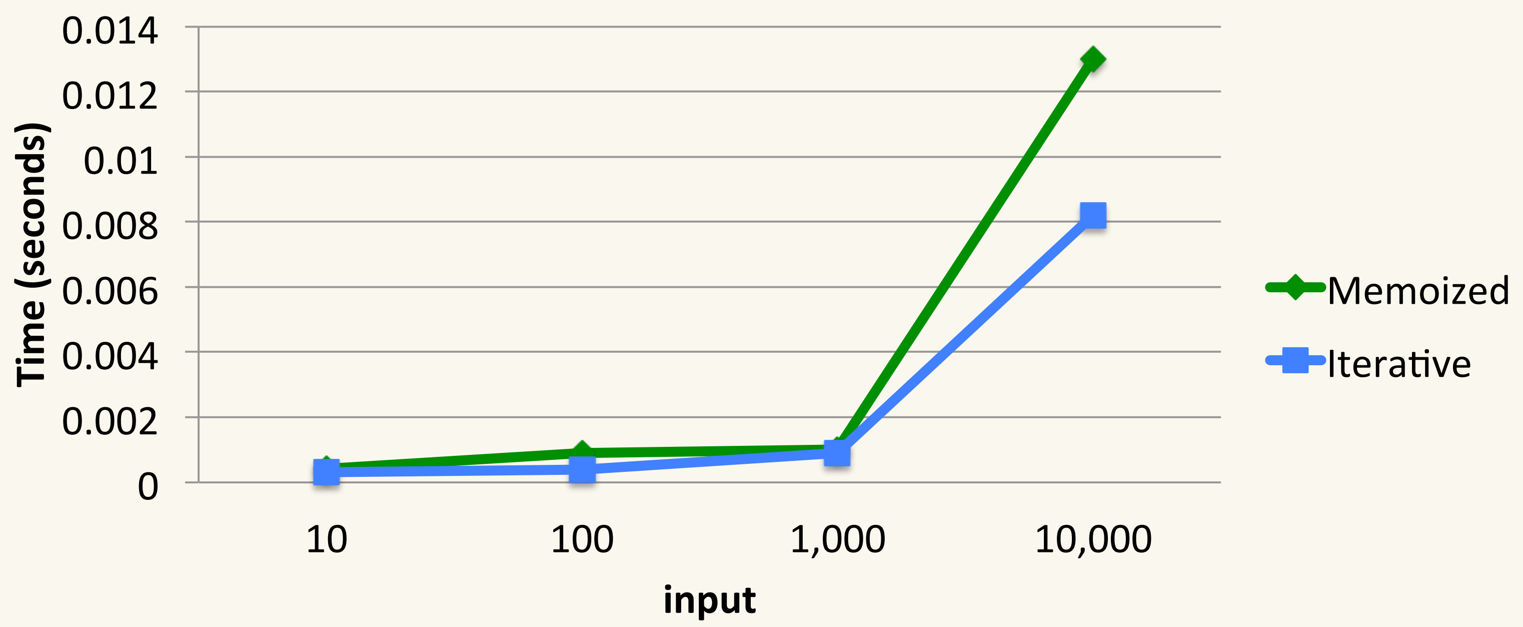
```
int fibMemoHelper(int n, int fibTable[])
{
    if (n==1)
        fibTable[1] = 1;
    else if (n==2)
        fibTable[2] = 1;
    else if (fibTable[n]==0)
     fibTable[n] = fibMemoHelper(n-1, fibTable)+
                   fibMemoHelper(n-2, fibTable);

    return fibTable[n];
}
```

Runtime Comparison of Different Fibonacci Implementations (in seconds)



	30	32	34	36	38	40	42	44
Plain, recursive	0.19	0.5	1.37	3.6	9	25	64	166
Memoized	0.0005	0.0005	0.00051	0.00053	0.00055	0.0006	0.0006	0.0006
Iterative	0.0004	0.00048	0.00051	0.00052	0.00055	0.00059	0.00059	0.00059

Runtime Comparison -- a Closer Look (1) (in seconds)



	10	100	1,000	10,000
Memoized	**0.00041**	**0.0009**	**0.001**	**0.013**
Iterative	**0.0003**	**0.0004**	**0.0009**	**0.0082**

Clicker Question

- `factorial_memoized(100)` is expected to run _____ than `factorial_plain_recursive(100)`?

- A: significantly faster

  B: slightly faster

  C: slightly slower

  D: significantly slower

  E: none of the above

Clicker Question (answer)

- `factorial_memoized(100)` is expected to run _____ than `factorial_plain_recursive(100)`?

- A: significantly faster

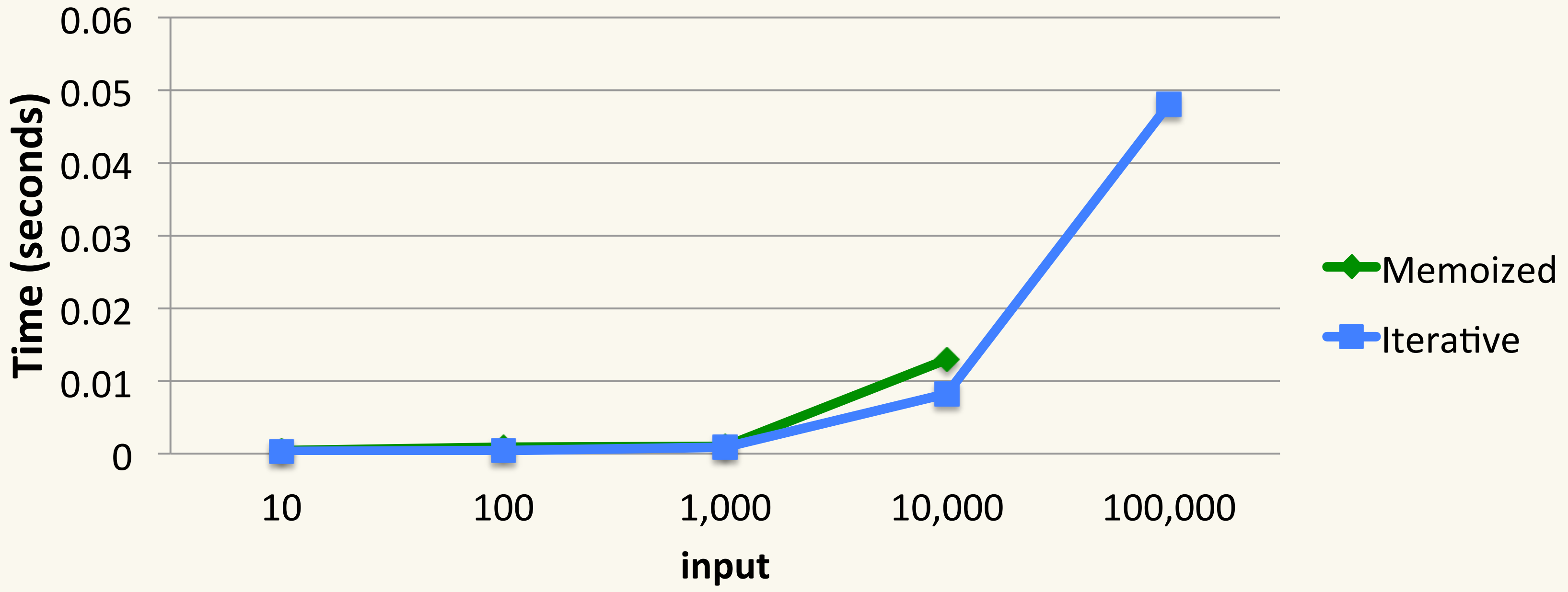
  B: slightly faster

  C: slightly slower

  D: significantly slower

  E: none of the above

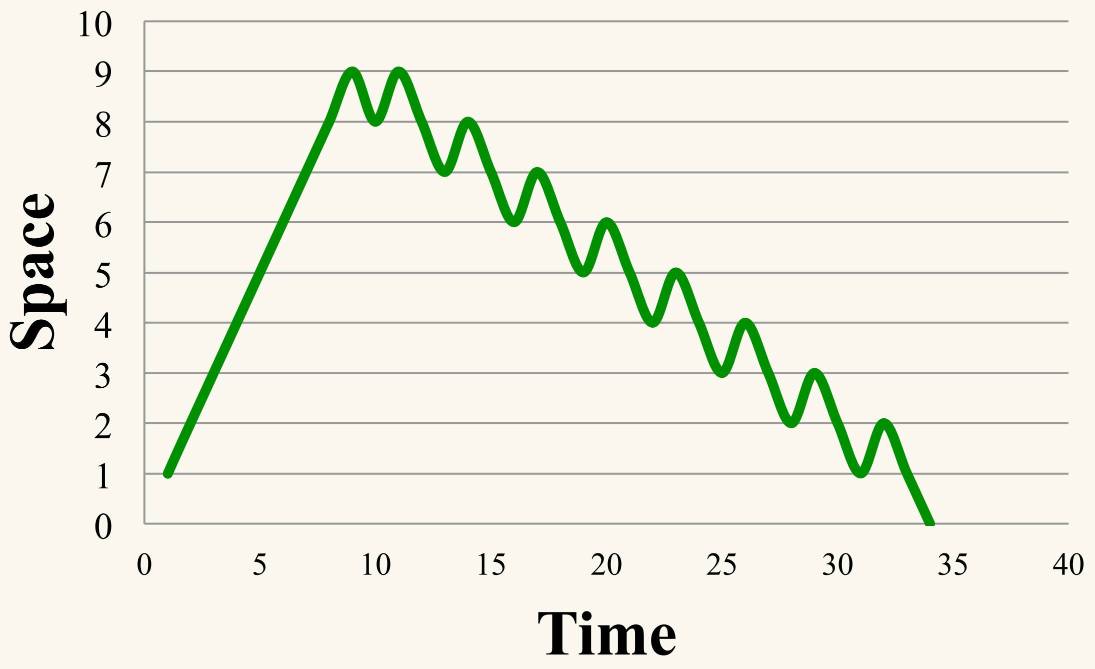
Runtime Comparison -- a Closer Look (2) (in seconds)



	10	100	1,000	10,000	100,000
Memoized	0.00041	0.0009	0.001	0.013	Crashed
Iterative	0.0003	0.0004	0.0009	0.0082	0.048

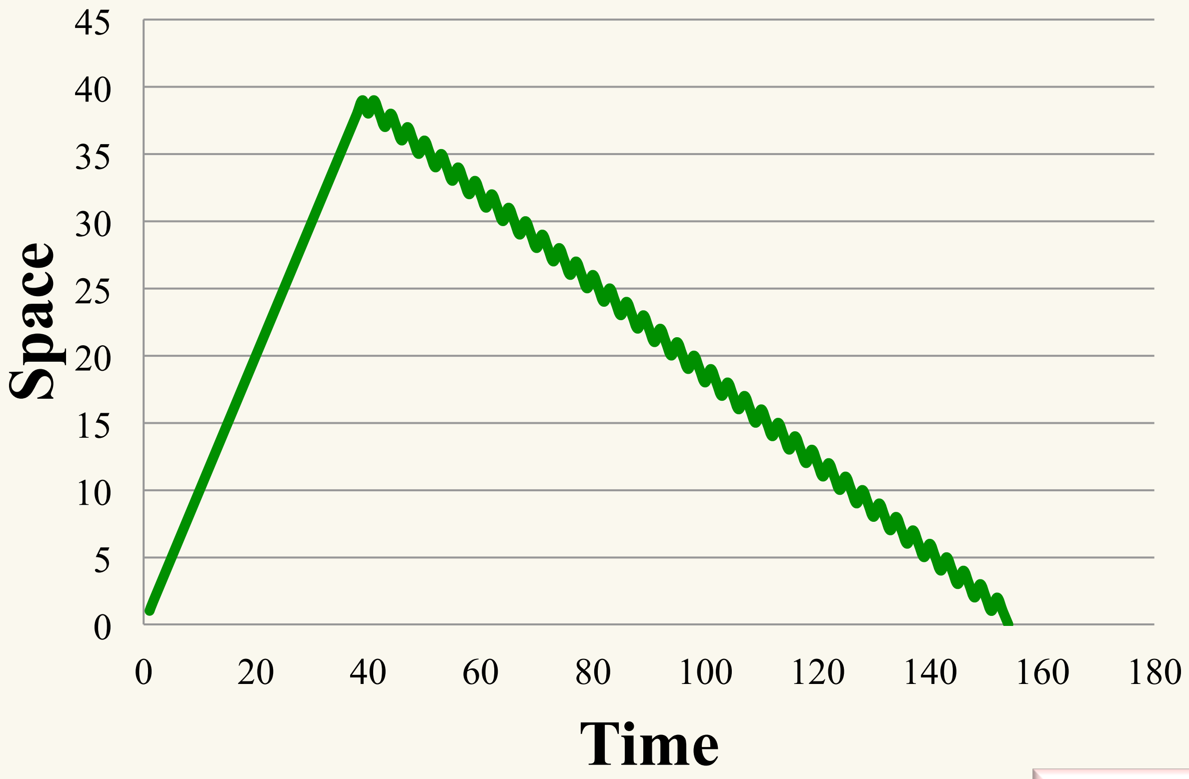
Memoized Fibonacci

Stack space used in computing `fib_memoized(10)`



Memoized Fibonacci

Stack space used in computing `fib_memoized(40)`



(1)~~Redundant computation~~
(2)Requires a lot of memory

Be Very Careful When You Use Recursion

- Poorly designed recursive procedures can use a lot of memory and be very slow.
  - It is is much easier to shoot yourself in the foot without noticing when you use recursion.

- Well designed recursive procedures can use almost (constant factor difference) the same amount of memory as iterative procedures:
  - Tail recursion can make recursive functions use less space.

Clicker Question

```
void endlesslyGreet(){
  printf("Hello World! \n");
  endlesslyGreet();
}
```

- What do you think would happen if you execute this function?

A: This will result in a stack overflow

B: This will magically not result in a stack overflow

C: It depends

D: None of the above

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.

Compare and Contrast

- How are these two functions
  - Similar?
  - Different?

```
int factorial(int n){

 if (n <=1)
    return 1;

  else
  return n * factorial(n-1);
}
```

```
void countinglyGreet(int n){

   if (n <=1)
     return;

   printf("Hello! \n");
   countinglyGreet(n-1);
}
```

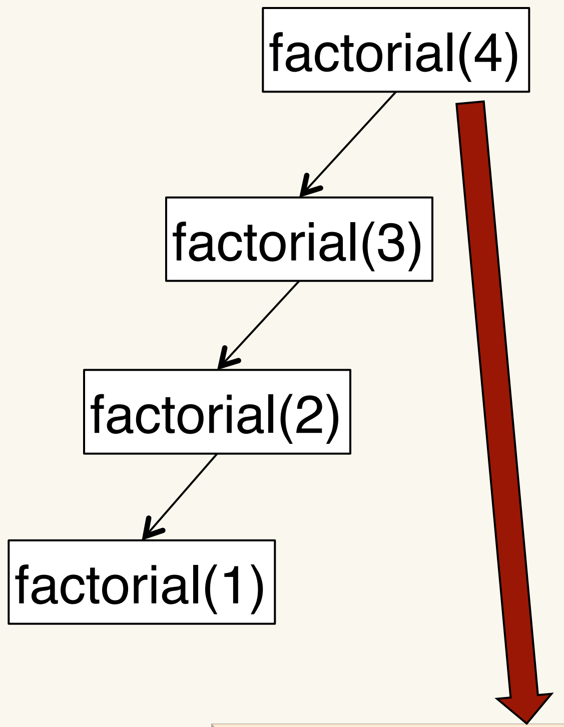
```
factorial(4);
```

```
countinglyGreet(4);
```

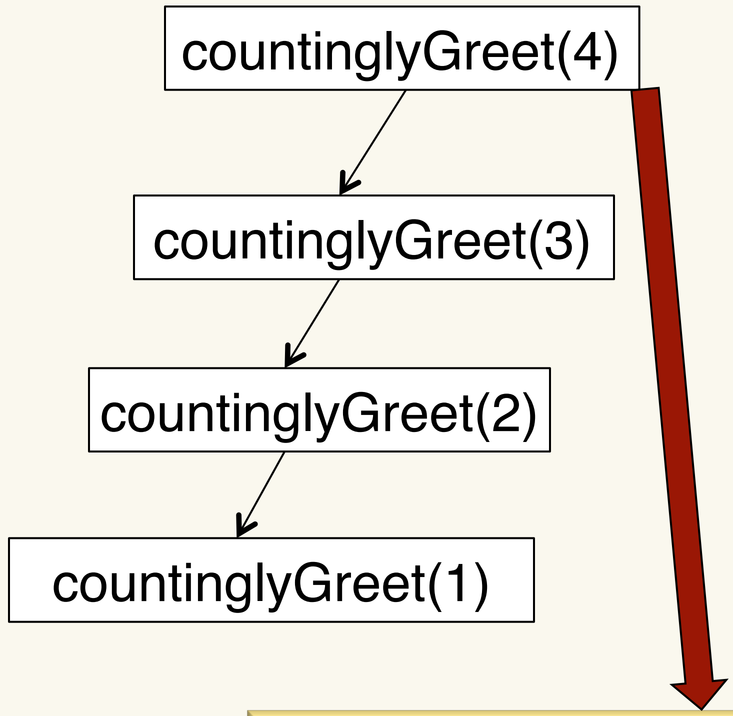
Compare and Contrast

`factorial(4);`

`countinglyGreet(4);`



Needs to wait for factorial(3)



Does not need to wait for countinglyGreet(3)

Managing the Call Stack: Tail Recursion

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

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

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.

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.

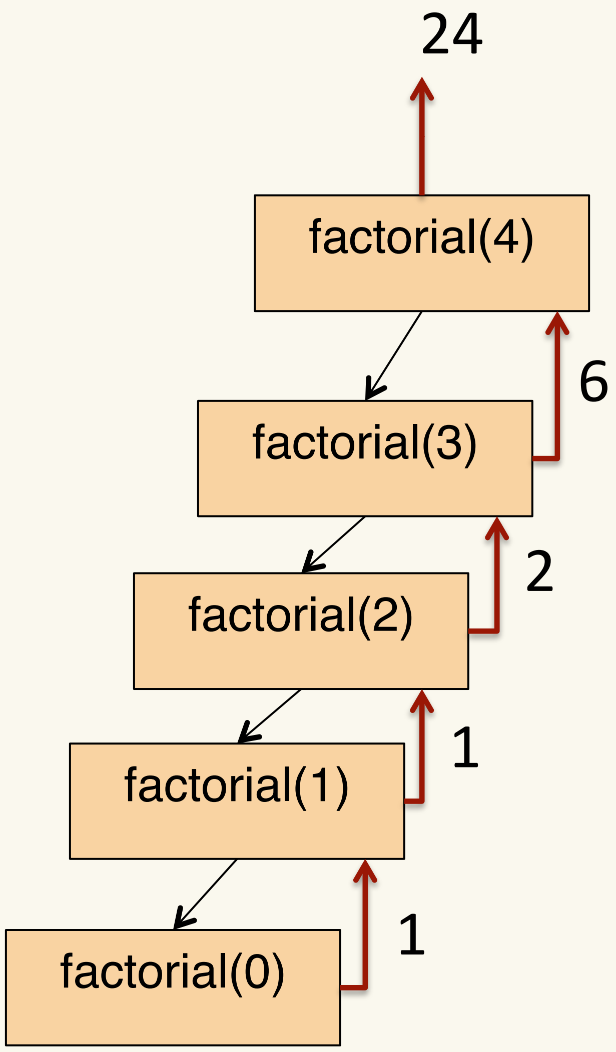
c. Not enough information.

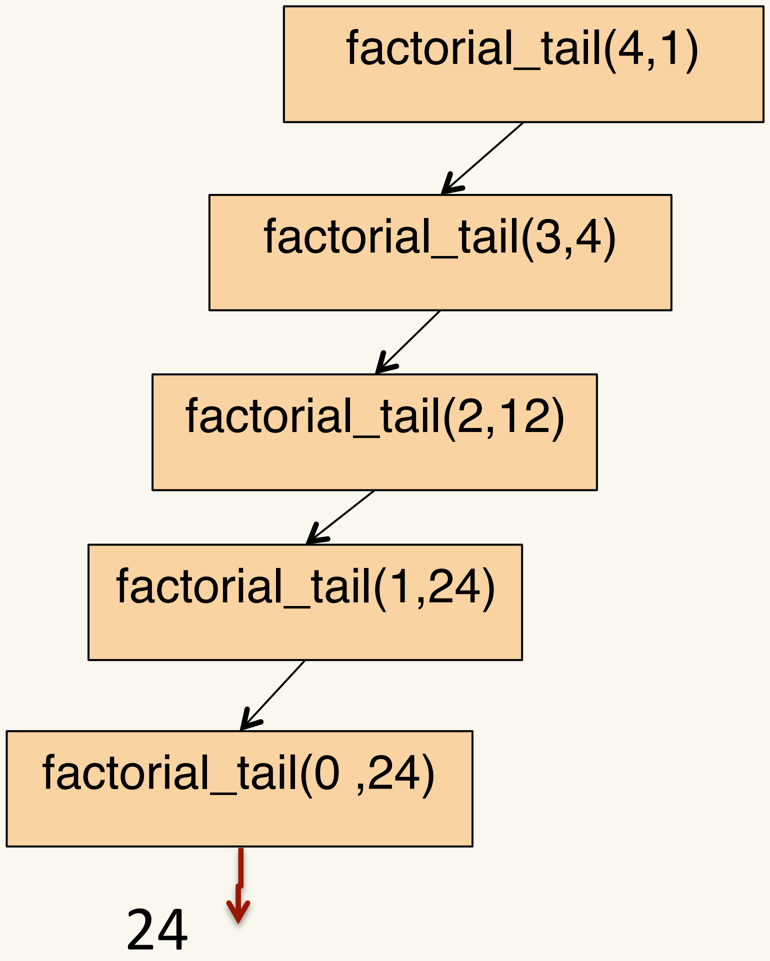
Tail Recursive Factorial

```
int factorial_tailrecursive(int n)
{
  int result = factorial_tailrecursive_helper(n, 1);
  return result;
}
```

```
int factorial_tailrecursive_helper(int n, int acc)
{
  if (n ==0)
    return acc;

  return factorial_tailrecursive_helper(n-1, acc*n);
}
```

Factorial


24
factorial(4)
6
factorial(3)
2
factorial(2)
1
factorial(1)
1
factorial(0)

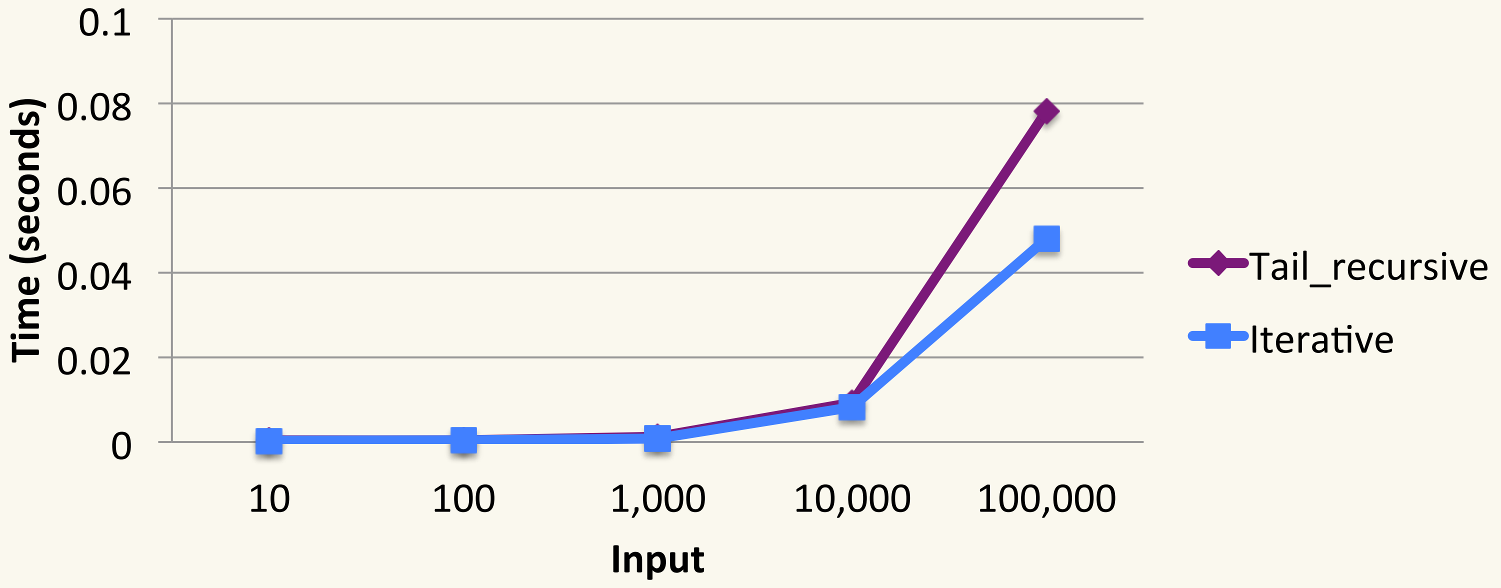

factorial_tail(4,1)
factorial_tail(3,4)
factorial_tail(2,12)
factorial_tail(1,24)
factorial_tail(0 ,24)
24

Tail Recursive Fibonacci

```
return fib_tailRecursive(n, 1, 1);
```

```
int fib_tailRecursive(int n, int next, int result) {
  if (n==1)
    return result;

  else{
    return fib_tailRecursive(n-1, result+next, next);

  }
```

Runtime Comparison (in seconds)
fib_tailRecursive vs. fib_iterative



	10	100	1,000	10,000	100,000
Tail_recursive	**0.00038**	**0.0005**	**0.0012**	**0.0093**	**0.078**
Iterative	**0.0003**	**0.0004**	**0.0009**	**0.0082**	**0.048**

Recursion vs. Iteration

- Iteration and recursion are equally powerful problem solving techniques.
  - recursive programs can be relatively simpler to write, analyze, and understand than iterative versions.
  - recursive programs can benefit from the call stack in more complex problems that run slower and/or require more space.

- Recursion may carry around a bit more (a constant factor more) memory than iteration but if well designed, neither is more efficient.

Learning Goals revisited

- design simple recursive functions.
- recognize algorithms as being iterative or recursive.
- describe how a computer runs recursive algorithms.
- demonstrate the ability to draw recursion trees.
- explain how stack overflow may arise as a result of recursion.
- explain why a recursively defined method may take more space than an equivalent iteratively defined method.