Algorithms

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today’s Lecture

- Recursion
- Topics:
  - Method call stack and activation records
  - Base case
  - Recursive case
  - Describe some recursive mathematical functions
  - Recursion and the method call stack
  - Print a list using recursion
  - Print a list in reverse using recursion
  - Insert into a sorted list using recursion
  - Describe when to use recursion

Activation Record (Stack Frame)

A record used at run time to store information about a function call, including the parameters, local variables, return address, and function return value (if a value-returning function)

Run-time Stack

A data structure that keeps track of activation records during the execution of a program

How Recursion Works

- Variables and parameters are not just stored anywhere on the stack.

- Variables and parameters from the ***same function*** are grouped together on the call stack.

```
void doSomething(int x) {
        String s;
        int z;
        // other code here…
}
```

Activation Record

Here is an activation record that would get created when doSomething is called.

Activation Record

DoSomething – int x; ← Parameter
string s; ← Local var.
int z; ← Local var.

Method Call Stack

Method Call Stack

- All variables declared in a function are stored in an ***activation record (or stack frame)***.

- The activation record for a function call stores all the variables and parameters declared in that function.

- **Activation Record Behavior**
  - **Method call → Push** new activation record on stack
  - **Method ends → Pop** top activation record off stack

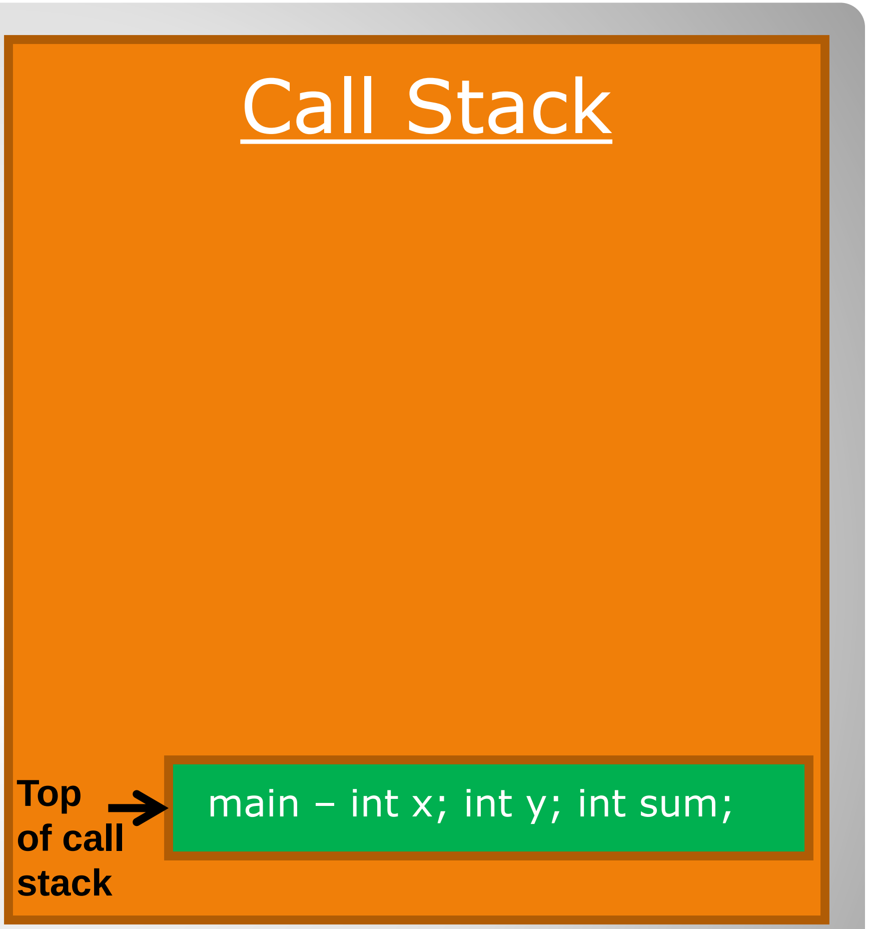
Method Call Stack Behavior

```
static int add(int num1, num2) {
   return num1 + num2;
}

static void show(int z) {
   System.out.println(z);
}

static void main(…) {
   int x, y, sum;
   x = 10;
   y = 20;
   sum = add(x, y);
   show(sum);
}
```

Execution is currently here (main just started)



CALL main – Push activation record on stack for main

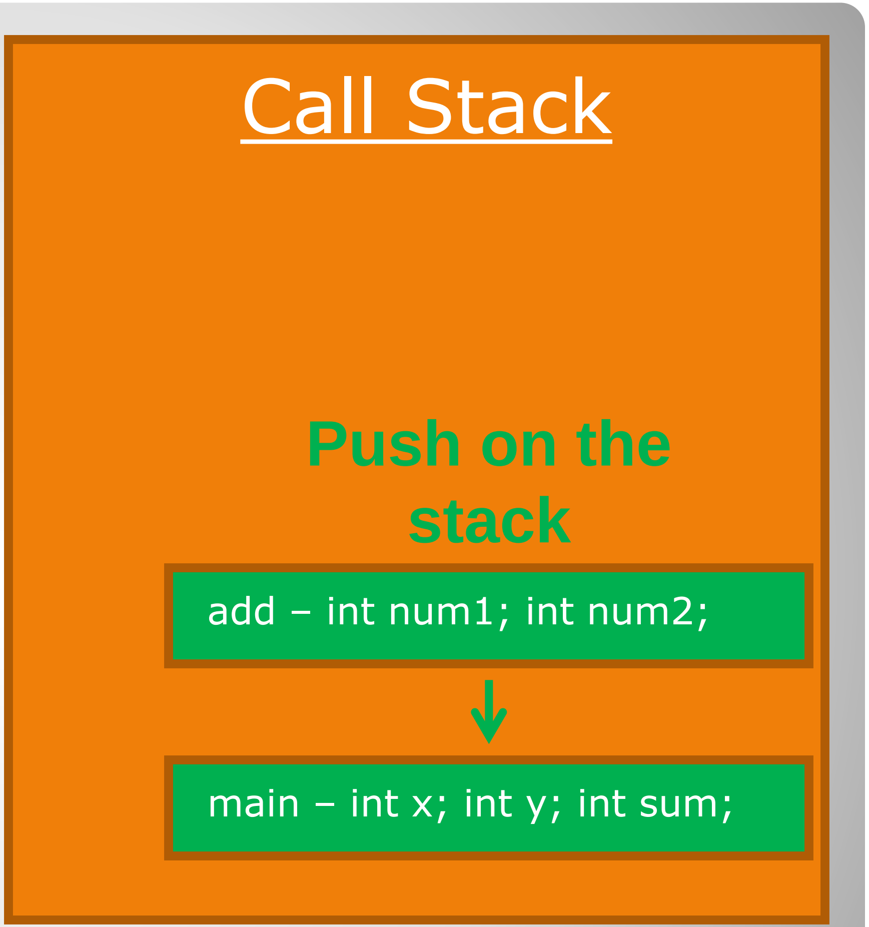
Method Call Stack Behavior

```
static int add(int num1, num2) {
   return num1 + num2;
}

static void show(int z) {
   System.out.println(z);
}

static void main(…) {
   int x, y, sum;
   x = 10;
   y = 20;
   sum = add(x, y);
   show(sum);
}
```

Execution here (Add just started)



Call Stack

Push on the stack

add – int num1; int num2;

main – int x; int y; int sum;

CALL ADD – Push activation record on stack for Add

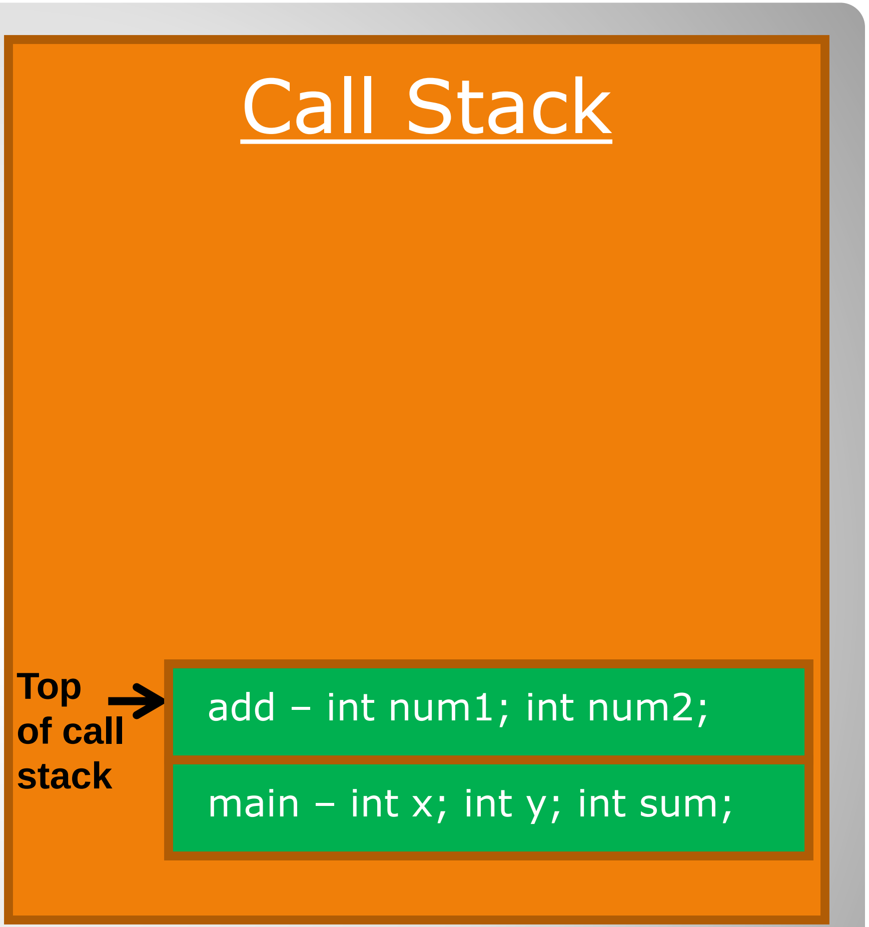
Method Call Stack Behavior

```
static int add(int num1, num2) {
   return num1 + num2;
}

static void show(int z) {
   System.out.println(z);
}

static void main(…) {
   int x, y, sum;
   x = 10;
   y = 20;
   sum = add(x, y);
   show(sum);
}
```

Execution here (Add just started)



Call Stack

Top of call stack

add – int num1; int num2;

main – int x; int y; int sum;

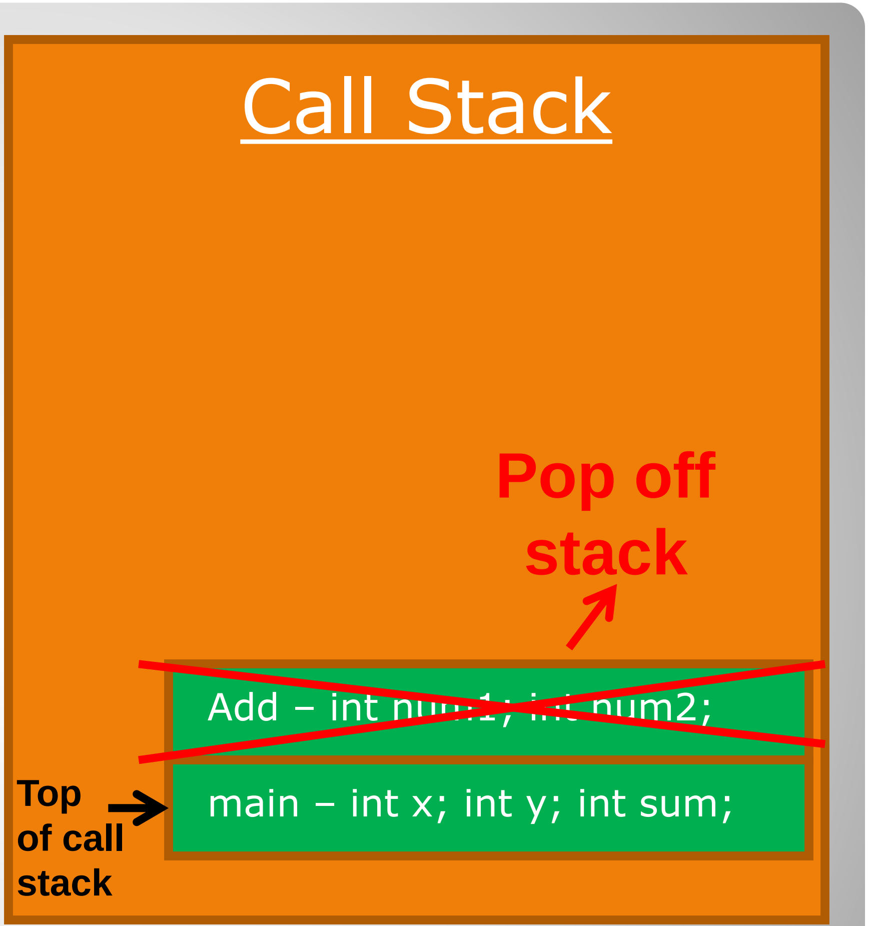
Add's activation record is now the top of the stack!!!

```
static int add(int num1, num2) {
   return num1 + num2;
}

static void show(int z) {
   System.out.println(z);
}

static void main(…) {
   int x, y, sum;
   x = 10;
   y = 20;
   sum = add(x, y);
   show(sum);
}
```

← **Execution here (Add just ended)**



Call Stack

Pop off stack

~~Add – int num1; int num2;~~

main – int x; int y; int sum;

Top of call stack

ADD ENDED - Add's activation record was popped off the stack!

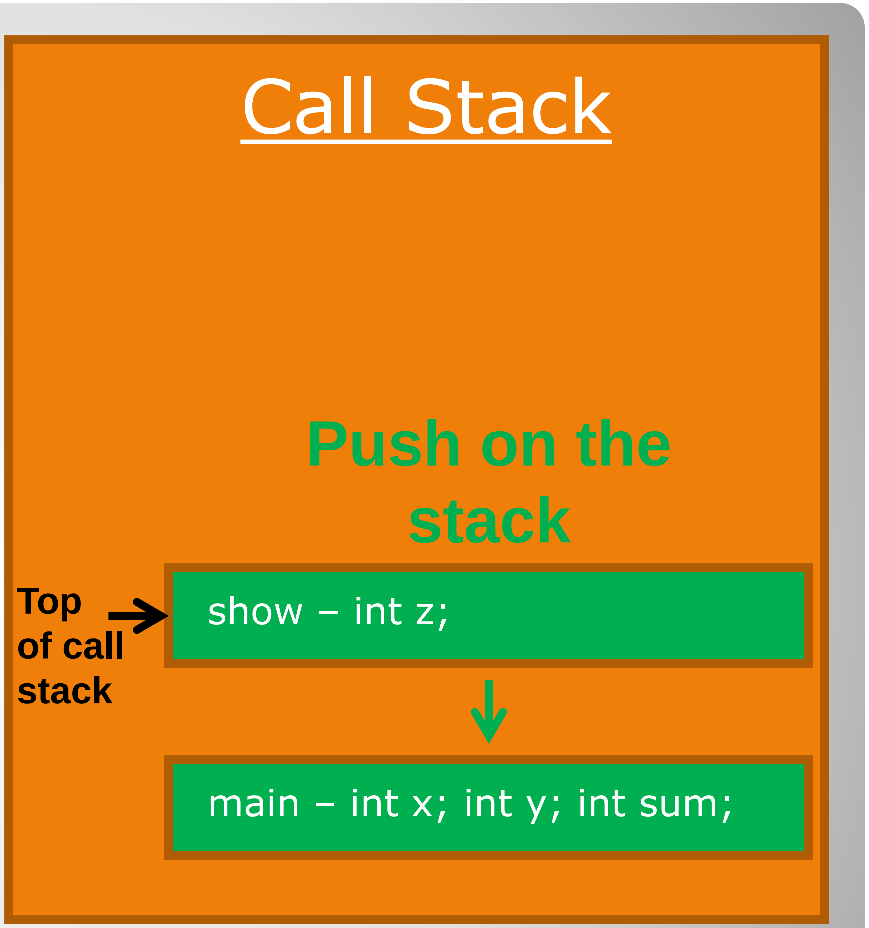
Method Call Stack Behavior

```
static int add(int num1, num2) {
   return num1 + num2;
}

static void show(int z) {
   System.out.println(z);
}

static void main(…) {
   int x, y, sum;
   x = 10;
   y = 20;
   sum = add(x, y);
   show(sum);
}
```

← Execution here (show just started)



Call Stack

Push on the stack

Top of call stack →

show – int z;

main – int x; int y; int sum;

CALL SHOW– Push activation record on stack for Show

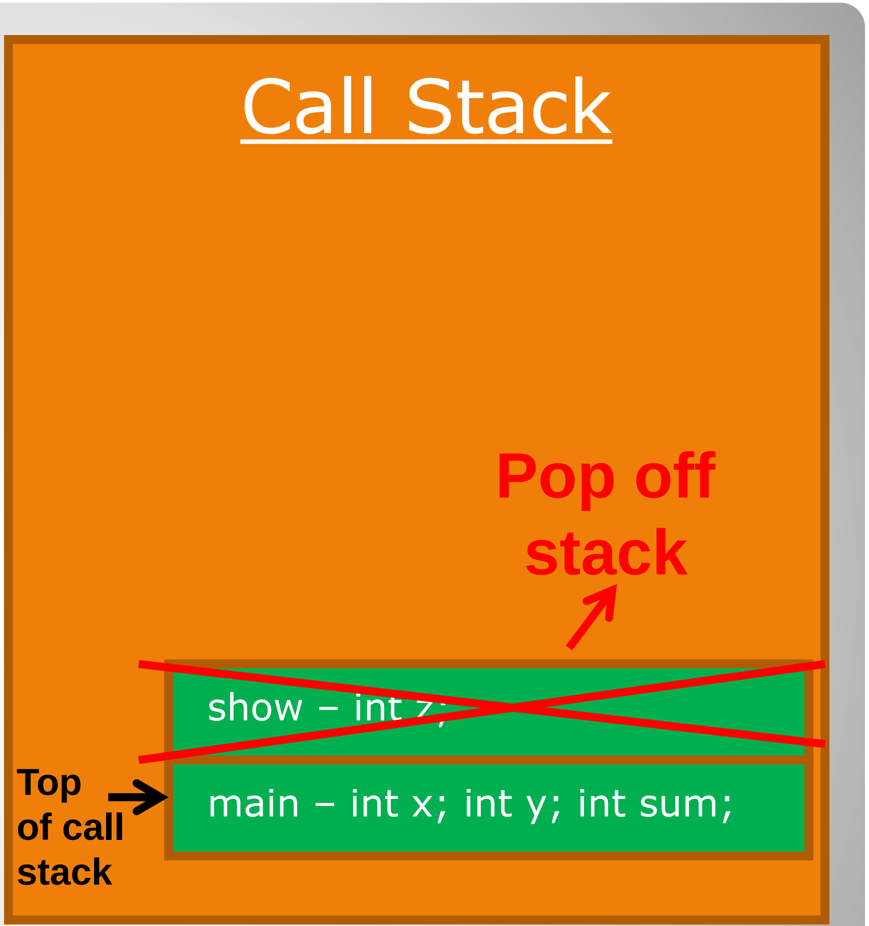
Method Call Stack Behavior

```
static int add(int num1, num2) {
   return num1 + num2;
}

static void show(int z) {
   System.out.println(z);
}

static void main(…) {
   int x, y, sum;
   x = 10;
   y = 20;
   sum = add(x, y);
   show(sum);
}
```

Execution here (show just ended)



Call Stack

Pop off stack

~~show – int z;~~

Top of call stack

main – int x; int y; int sum;

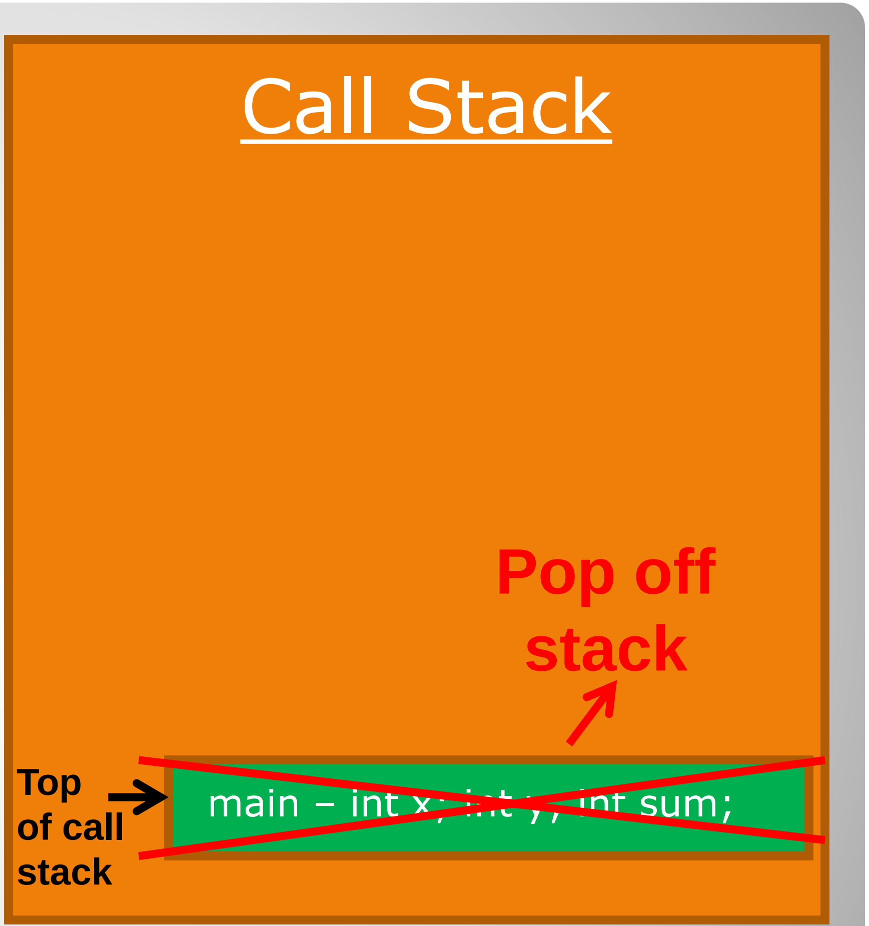
SHOW ENDED – Show's activation record was popped off the stack!

Method Call Stack Behavior

```
static int add(int num1, num2) {
   return num1 + num2;
}

static void show(int z) {
   System.out.println(z);
}

static void main(…) {
   int x, y, sum;
   x = 10;
   y = 20;
   sum = add(x, y);
   show(sum);
}
```



Call Stack

Pop off stack

Top of call stack

~~main – int x, int y, int sum;~~

MAIN ENDED – main's activation record was popped off the stack! Program done.

Method Call Stack Behavior

Video

- Recursion (Mario)

https://www.youtube.com/watch?v=fBJHeZgGQQ4

Recursion

- Do the following tasks, given a recursive routine
    - Determine whether the routine halts
    - Determine the base case(s)
    - Determine the general case(s)
    - Determine what the routine does
    - Determine whether the routine is correct and, if it is not, correct it

Recursion Goals

Recursion Goals

- Do the following tasks, given a simple recursive problem
  - Determine the base case(s)
  - Determine the general case(s)
  - Design and code the solution as a recursive void or value-returning function
- Decide whether a recursive solution is appropriate for a problem

How is recursion like a set of Russian dolls?

What Is Recursion?

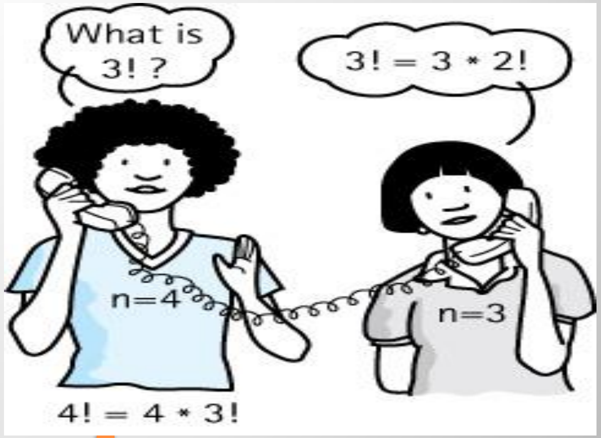
What Is Recursion?

- **Recursive call**
- A method call in which the method being called is the same as the one making the call
- **Direct recursion**
- Recursion in which a method directly calls itself
- **Indirect recursion**
- Recursion in which a chain of two or more method calls returns to the method that originated the chain

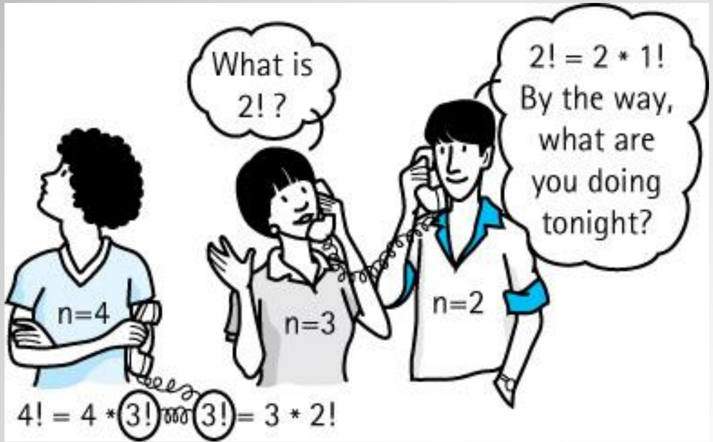
Recursive definition

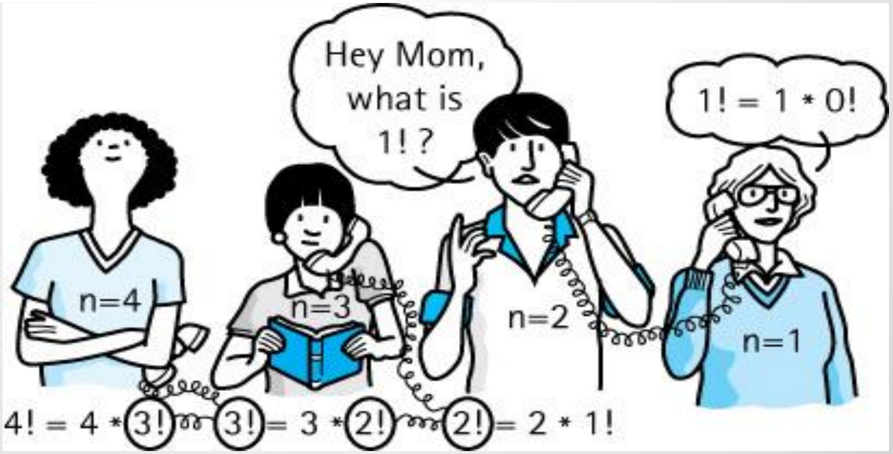
A definition in which something is defined in terms of a smaller version of itself

What is 3 factorial?



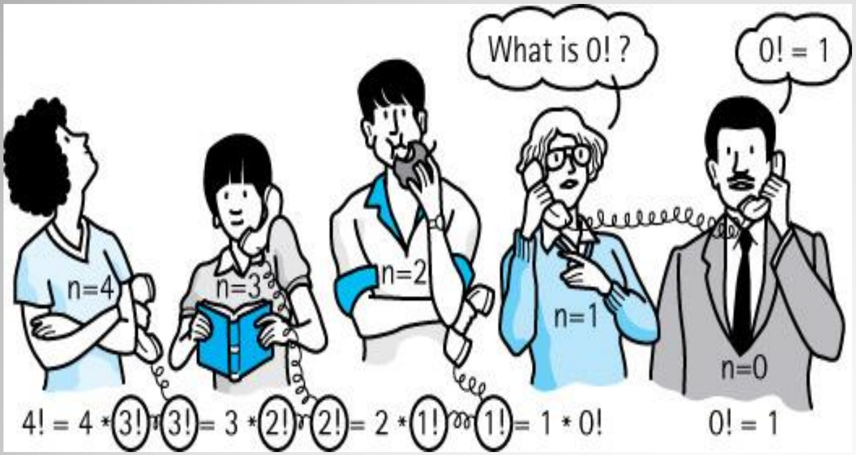
Example of Recursion

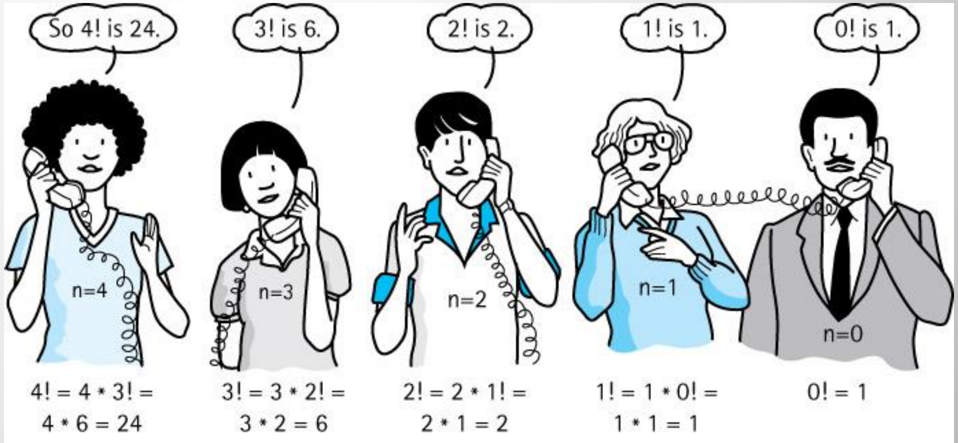




Example of Recursion

Examples of Recursion





Mathematical Description of Factorial

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

Writing Recursive Solutions - Factorial

Example of Recursion

- **Base case**
- The case for which the solution can be stated nonrecursively
- **General (recursive) case**
- The case for which the solution is expressed in terms of a smaller version of itself
- **Recursive algorithm**
- A solution that is expressed in terms of (a) a smaller instance(s) of itself and (b) a base case(s)

Writing Recursive Solutions

Algorithm for writing recursive solutions

- Determine the **size** of the problem
  - Size is the factor that is getting smaller
  - Size is usually a parameter to the problem
- Identify the **base case(s)**
  - The case(s) for which you know the answer
- Identify the **general case(s)**
  - The case(s) that can be expressed as a smaller version of the size

Let’s try it

Problem: Calculate X^n (X to the nth power)

Recursive formulation: $X*(X)*(X^n)*...*X$ (x n times)

What is the size of the problem?

Which case do you know the answer to?

Which case can you express as a smaller version of the size?

Writing Recursive Solutions - Power

Mathematical Description of Power

$$X^n = \begin{cases} 1, & \text{if } n = 0 \qquad \text{(Base)} \\ X * X^{n-1}, & \text{if } n > 0 \qquad \text{(Recursive)} \end{cases}$$

Writing Recursive Solutions - Power

Writing Recursive Solutions - Power

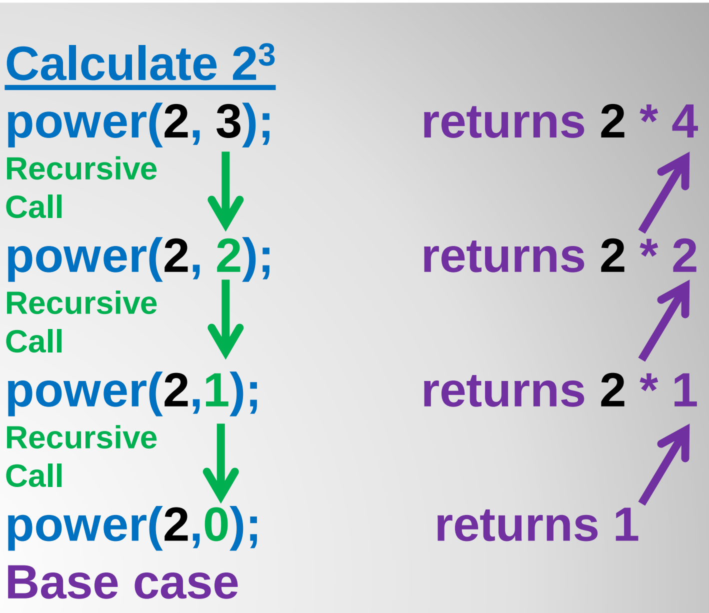
```
int power(int number, int exponent)
{
        // Is it the base case?
        if (exponent  ==  0)
        {
                // Base case
                return 1;
        }
        else
        {
                // Recursive case – Call on smaller case
                return  number  *  power(number, exponent - 1);
        }
}
```

Problem is a smaller version of itself.

Sample Execution - Power

```
int power(int number, int exponent)
{


        // Is it the base case?
        if (exponent  ==  0)
        {
                // Base case
                return 1;
        }
        else
        {
                // Recursive case – Call on smaller version of itself
                return  number  *  power(number, exponent - 1);
        }
}
```



Calculate 2^3

power(2, 3); returns 2 * 4

Recursive Call

power(2, 2); returns 2 * 2

Recursive Call

power(2,1); returns 2 * 1

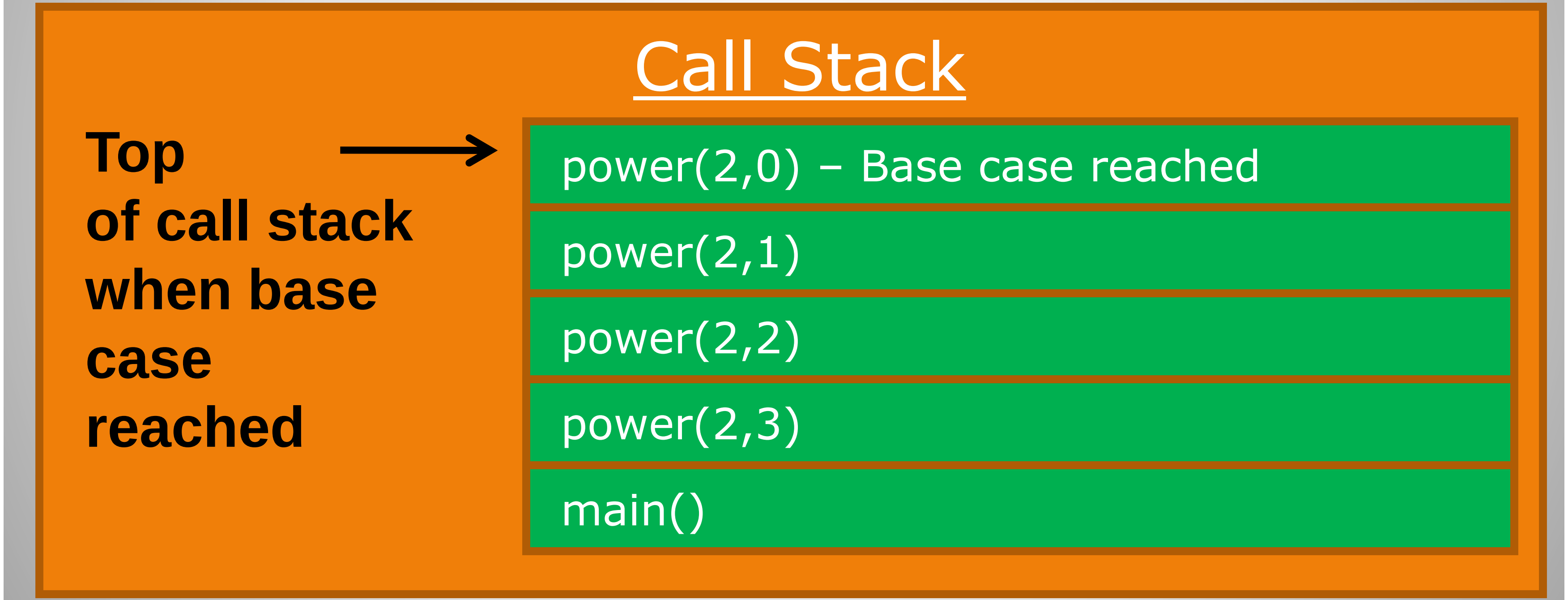
Recursive Call

power(2,0); returns 1

Base case

Sample Execution - Power

```
static void main(…) {
   int result = power(2,3);
   System.out.println(result);
}
```



Call Stack

Top of call stack when base case reached

power(2,0) – Base case reached

power(2,1)

power(2,2)

power(2,3)

main()

Writing Recursive Solutions - Power

- What would happen if we left out the base case?

```
int power(int number, int exponent)
{
            // Recursive case – Call on smaller case
            return  number  *  power(number, exponent - 1);
}
```

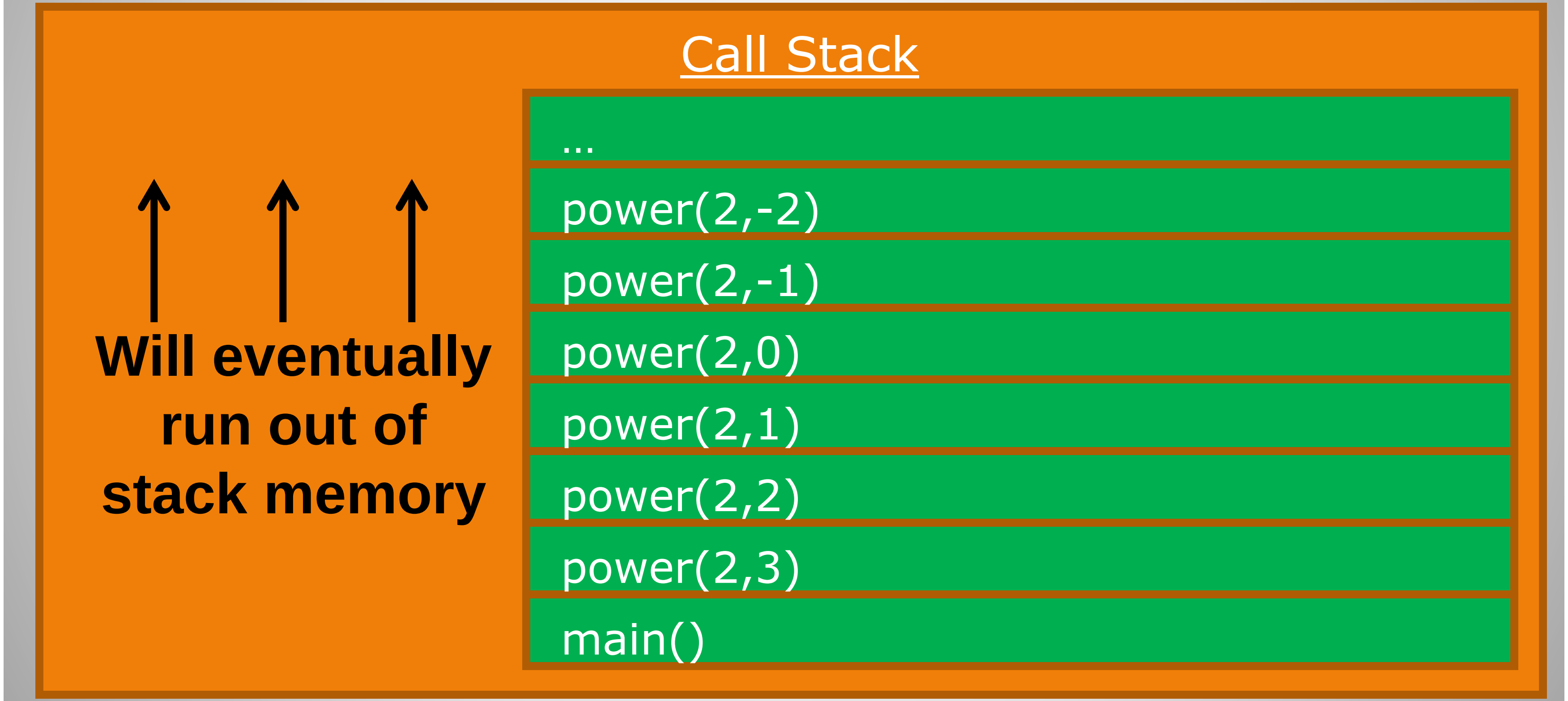
No base case in this method

Sample Execution – No base case

```
int power(int number, int exponent) {
   return  number  *  power(number, exponent - 1);
}
```

Stack Overflow!!!

METHOD CALLS NEVER STOP!!!



Call Stack

...

power(2,-2)

power(2,-1)

power(2,0)

power(2,1)

power(2,2)

power(2,3)

main()

Will eventually run out of stack memory

Pattern of solution

```
if  (some condition for which answer is known)
        solution statement

else
        function call on smaller version of itself
```

Base case

Recursive case

Writing Recursive Solutions

Shall we try it again?

Problem: Calculate Nth item in Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

What is the next number?

What is the size of the problem?

Which case do you know the answer to?

Which case can you express as a smaller version of the size?

Writing Recursive Solutions - Fibonacci

Mathematical Description of Fibonacci Sequence

$$F_n = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n >= 2 \end{cases}$$

Writing Recursive Solutions - Fibonacci

Writing Recursive Solutions - Fibonacci

```
int fibonacci(int n)
{
        if (n == 0 || n == 1)
                return n;
        else
                return fibonacci(n-2) + fibonacci(n-1);
}
```

That was easy, but it is not very efficient. Why?

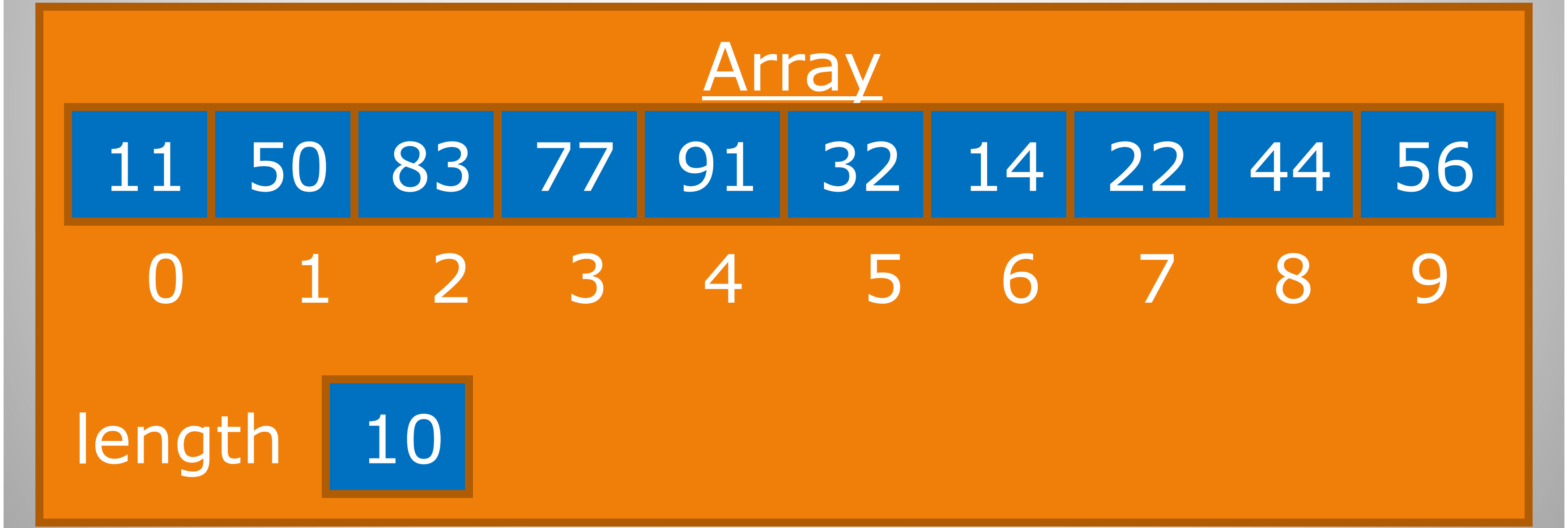
Writing Recursive Solutions

Shall we try it again?

Problem: Search a list of integers for a value and return true if it is in the list and false otherwise.

Recursive Search – Array

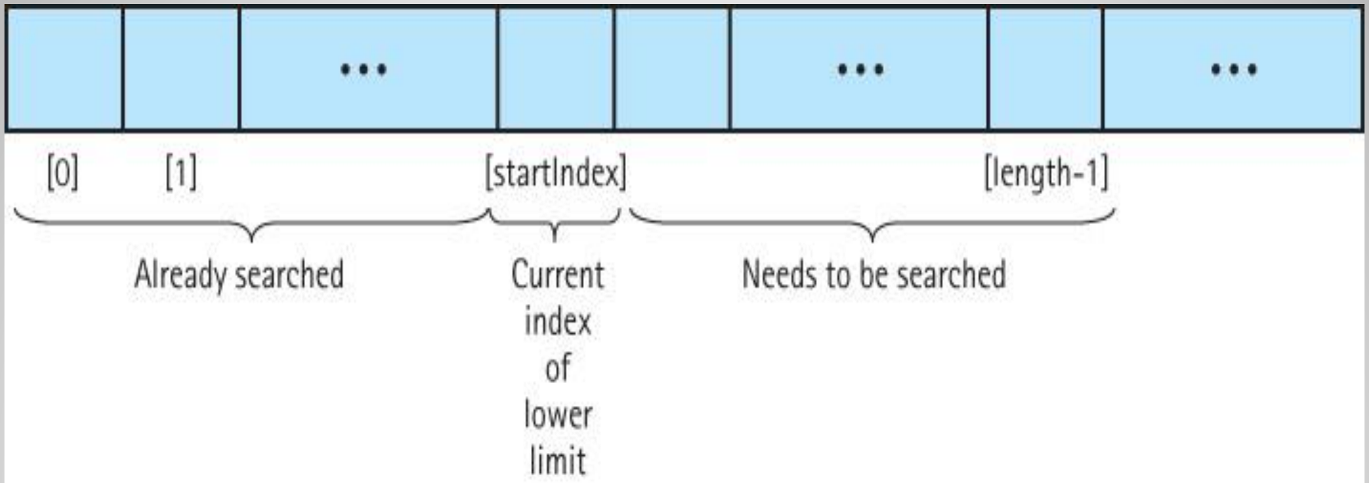
- Recursively search an array for an item.
- Assume the following list:



Array

11	50	83	77	91	32	14	22	44	56
0	1	2	3	4	5	6	7	8	9

length 10



```
boolean valueInArray(int value, int startIndex);
```

Which case do you know the answer to?

Which case can you express as a smaller version of the size?

Recursive Search – Array

Recursive Search – Array

```
int[] info = new int[10]; // Member variable
boolean valueInArray(int value, int startIndex) {
    if (startIndex == info.length)
        return false; // Reached end of list
    else if (info[startIndex] == value)
        return true;  // Found it
    else
        return valueInArray(value, startIndex + 1);
}
```

Base Case 1

Base Case 2

Recursive Case

Note

The array is a member variable and valueInArray has access to it.

Problem is a smaller version of itself. Call valueInArray but this time starting from the NEXT index in the list.

Recursive Search – Array

```
int[] info = new int[10]; // Member variable

boolean valueInArray(int value) {
    vallueInArray(value, 0); // Start recursion
}
boolean valueInArray(int value, int startIndex) {
    if (startIndex == info.length)
        return false; // Reached end of list
    else if (info[startIndex] == value)
        return true;  // Found it
    else return valueInArray(value, startIndex + 1);
}
```

Public function. User of class would actually call this one.

Private function. User does NOT call because it contains an implementation detail.

The implementation detail is that an array is used. The user would have to supply the starting index.

Why use recursion?

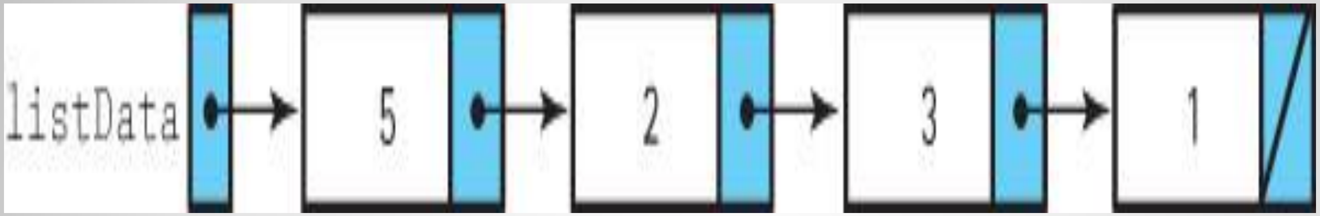
True, these examples could more easily be solved using iteration

However**,** a recursive solution is a natural solution in certain cases, especially when pointers are involved

Writing Recursive Solutions

Recursive Print – List (linked)

Printing a list in order recursively



Size?

Base case?

Recursive (general) case?

Recursive Print – List (linked)

```
void print(Node listPtr)
{
  if (listPtr != null)
  {
    System.out.prinln(listPtr.data);
    print(listPtr.next);
  }
}
```

Where is the base case?

Recursive Print – List (linked)

```
void print(Node listPtr)
{
   if (listPtr != null)
   {
      System.out.println(listPtr.data);
      print(listPtr.next);
   }
}
```

Where is the base case?

ANSWER: When listPtr is null

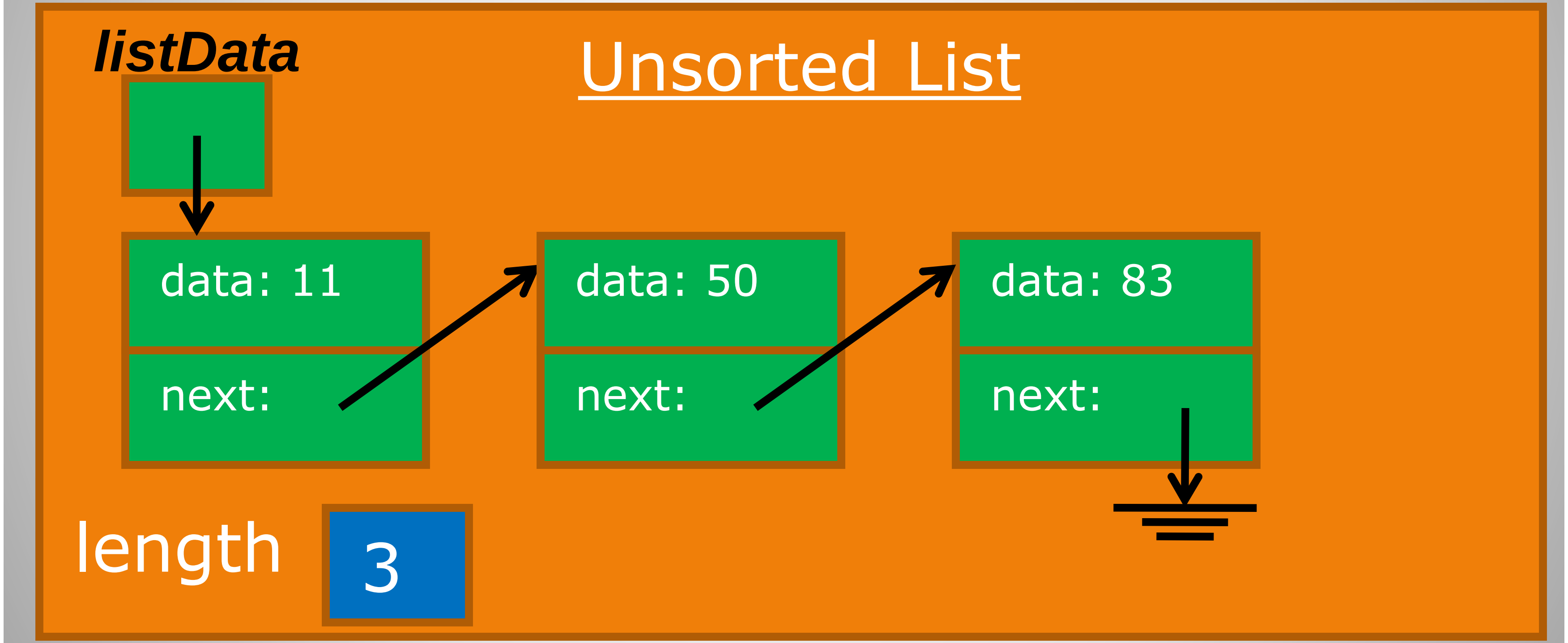
Recursive Print – List (linked)

```
// This version will call the recursive version
void print() {
   print(listData);
}

// Recursive version will call itself
void print(Node listPtr)
{
   if (listPtr != null)
   {     // Prints BEFORE recursive call
      System.out.println(listPtr.data);
      print(listPtr.next);
   }
}
```

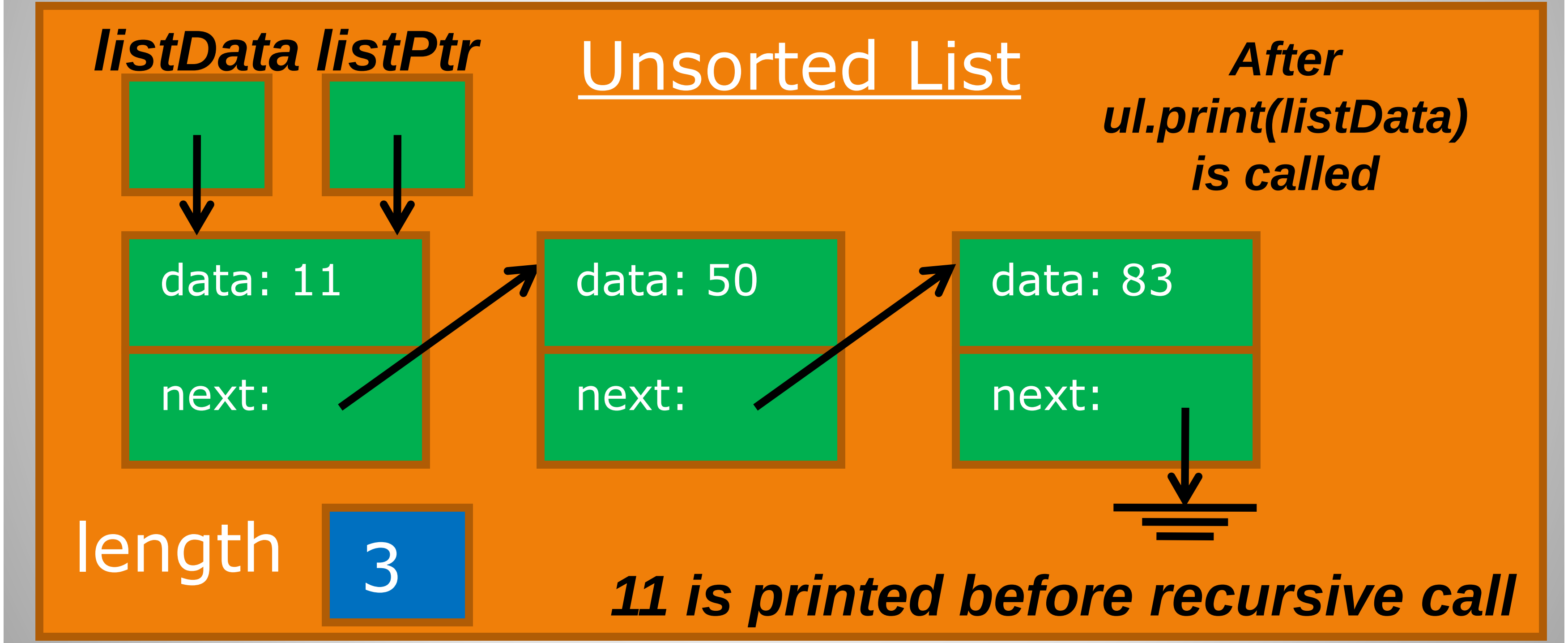
Recursive Print – List (linked)

```
UnsortedList ul;
ul.print();        // Call the helper to start
```



```
ul.print();      // Calls recursive version with start of list

ul.print(listData);  // First call to recursive version
```



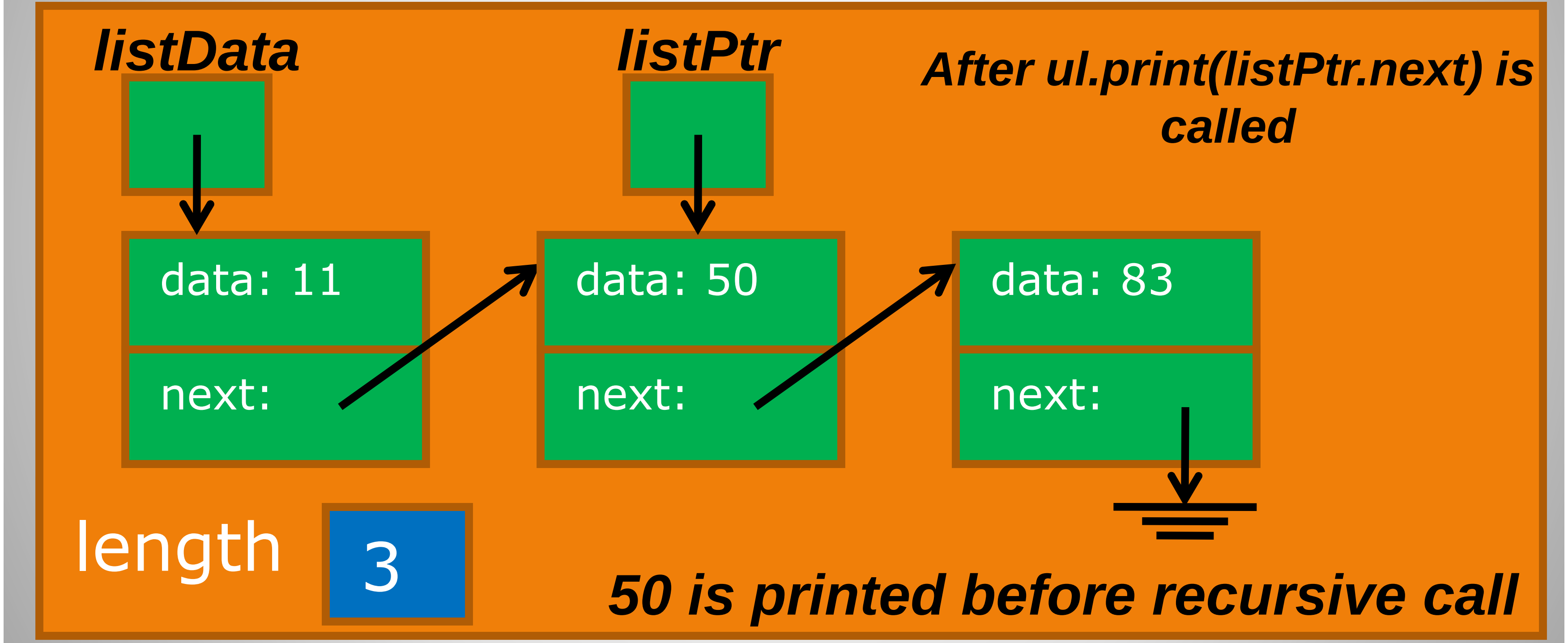
After ul.print(listData) is called

11 is printed before recursive call

Recursive Print – List (linked)

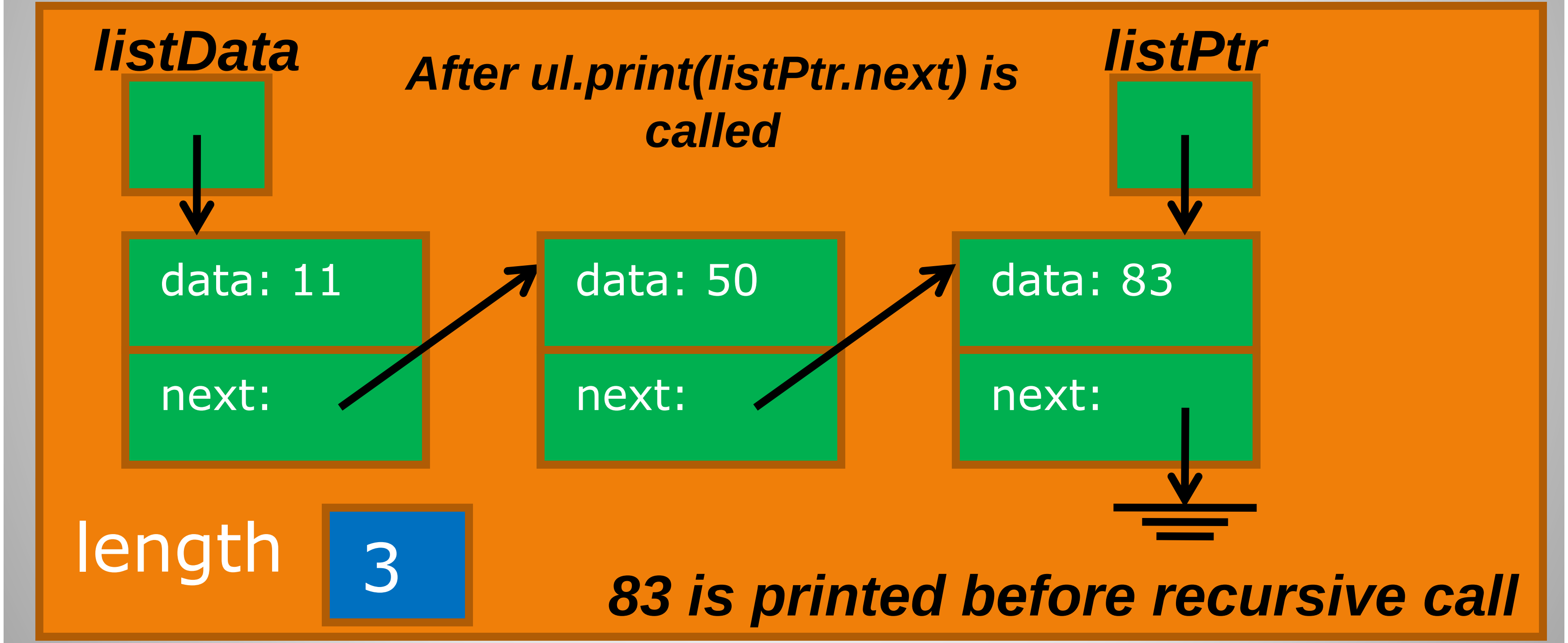
ul.print(listData);// Calls recursive version again

ul.print(listPtr.next);


listData
listPtr
After ul.print(listPtr.next) is called
data: 11
next:
data: 50
next:
data: 83
next:
length
3
50 is printed before recursive call

Recursive Print – List (linked)

```
ul.print(listPtr.next);
```



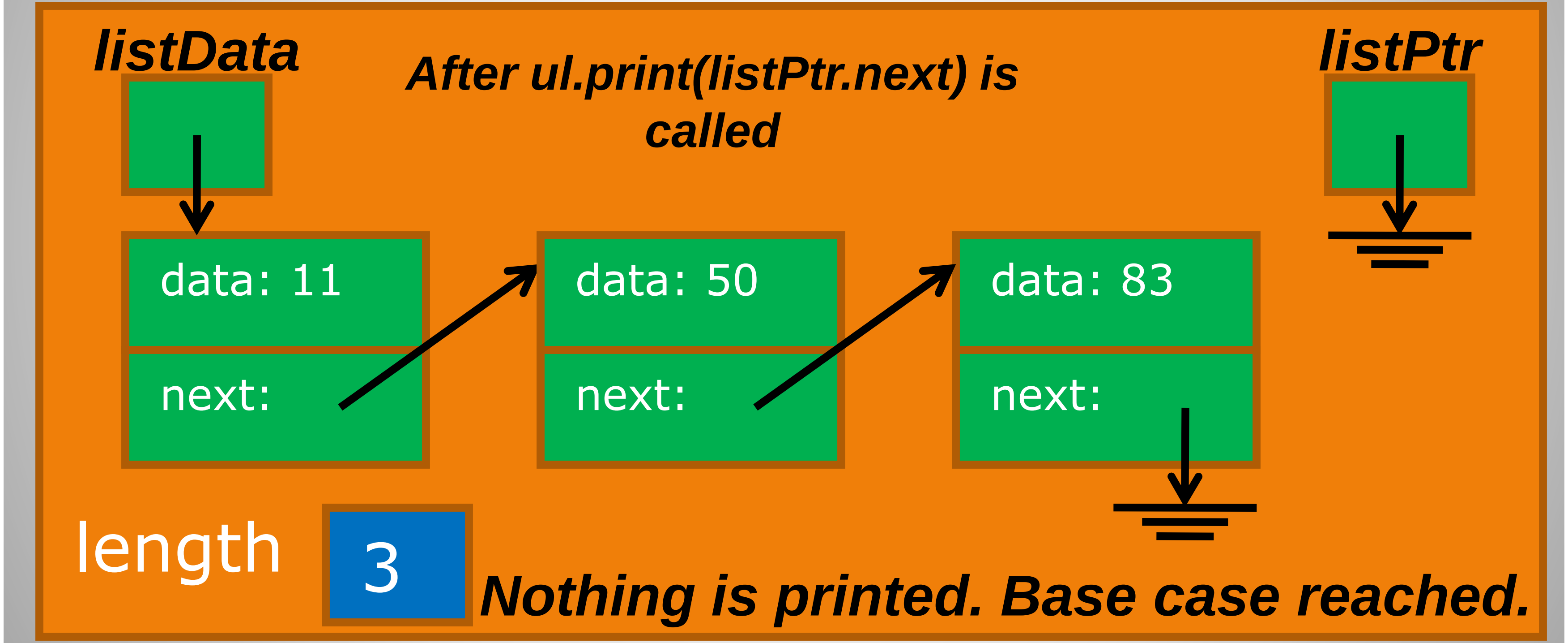
After ul.print(listPtr.next) is called

83 is printed before recursive call

Recursive Print – List (linked)

ul.print(listPtr.next);

All the recursive calls are done. Now "unwind" since base case has been reached (listPtr == null).



listData

After ul.print(listPtr.next) is called

listPtr

data: 11

next:

data: 50

next:

data: 83

next:

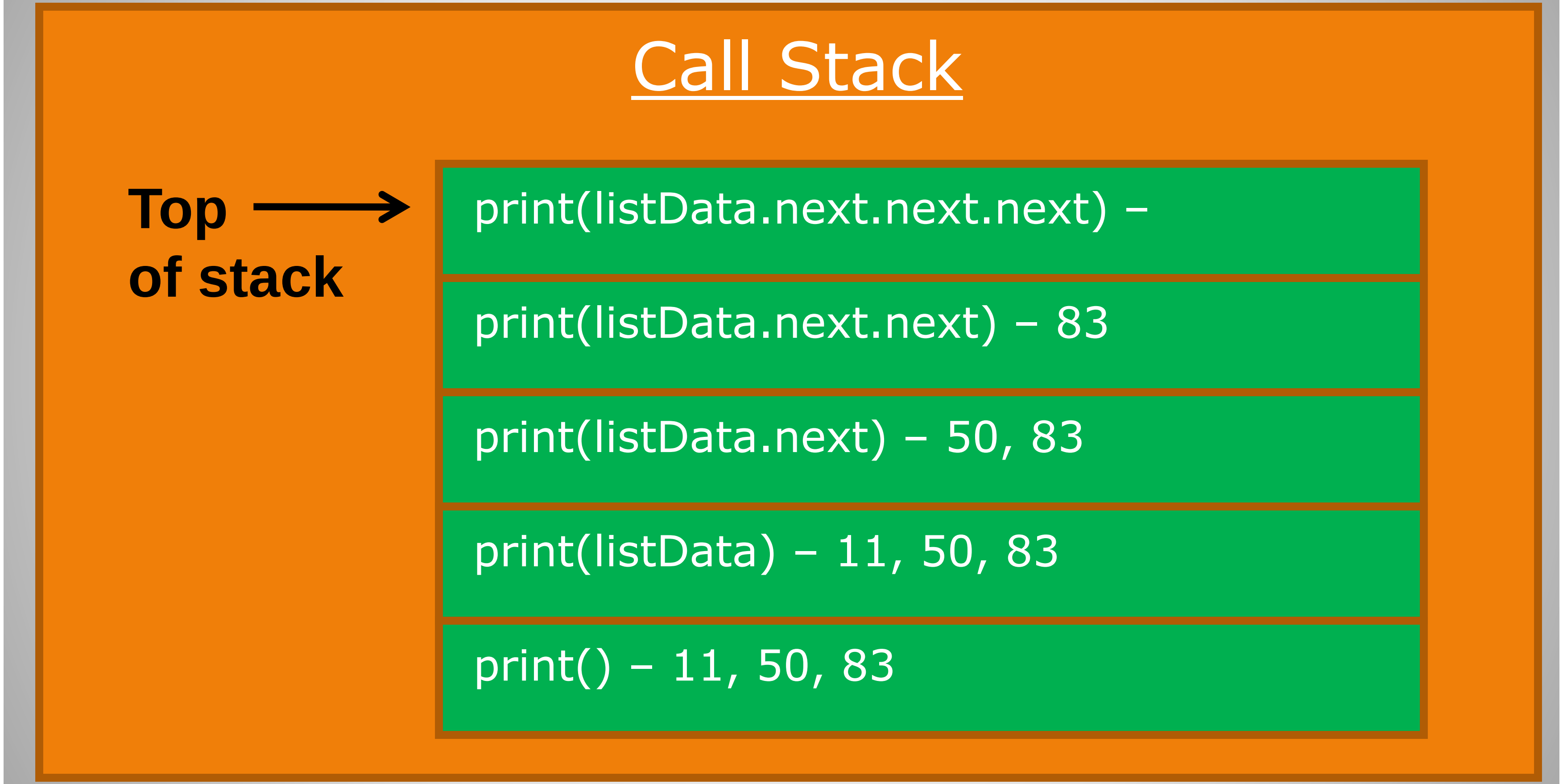
length

3

Nothing is printed. Base case reached.

Recursive Print – List (linked)

- Call stack when base case is reached



Call Stack

Top of stack

print(listData.next.next.next) –

print(listData.next.next) – 83

print(listData.next) – 50, 83

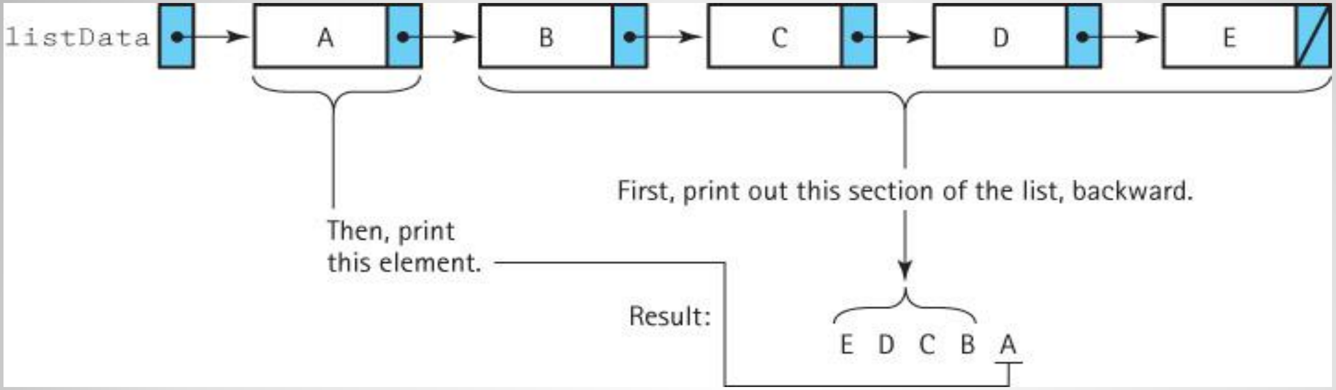
print(listData) – 11, 50, 83

print() – 11, 50, 83

Recursive Print – List (linked)

Recursive Print – List (linked)

Printing a list in **reverse** order



What must be changed from the in-order print to make the code print in reverse order?

Recursive Print – List (linked)

```
// This version will call the recursive version
void revPrint()
{
   revPrint(listData);
}
// Recursive version will call itself
void revPrint(Node listPtr)
{
   if (listPtr != null)
   {        // Prints AFTER recursive call
      print(listPtr.next);
      System.out.println(listPtr.data);
   }
}
```

Prints during the "unwind"

```
boolean mystery(int[] info, int item, int fromLoc, int toLoc)
{
   int  mid;
   if  ( fromLoc > toLoc )
     return  false;
   else
   {
     mid = ( fromLoc + toLoc ) / 2 ;
     if (info[mid] == item)
       return  true ;
     else
       if (item < info[mid])
         return mystery(info, item, fromLoc, mid-1);
       else
         return mystery(info, item, mid + 1, toLoc) ;
   }
}
```

What does this function return?

```
boolean binarySearch(int[] info, int item, int fromLoc, int toLoc)
{
  int  mid;
  if  ( fromLoc > toLoc )
    return  false;
  else
  {
    mid = ( fromLoc + toLoc ) / 2 ;
    if (info[mid] == item)
      return  true ;
    else
      if (item < info[mid])
        return binarySearch(info, item, fromLoc, mid-1);
      else
        return binarySearch(info, item, mid + 1, toLoc) ;
  }
}
```

What does this function return?

ANSWER: true if found false otherwise

Removing Recursion

Tail Recursion

The case in which a function contains only a single recursive invocation and it is the last statement to be executed in the function.

A tail recursive function can be replaced with iteration.

Stacking

Using a stack to keep track of each local environment, i.e., simulate the run-time stack .

When To Use Recursion

- Depth of recursive calls is relatively “shallow” compared to the size of the problem

- Recursive version does about the same amount of work as the nonrecursive version (same Big-O)

- The recursive version is shorter and simpler than the nonrecursive solution

SHALLOW DEPTH

EFFICIENCY

CLARITY

Recursion Real-time Speed

- **The recursive version is generally slower than an equivalent iterative version.**

- The reason the **recursive version** is slower is that it **generally requires more method calls**.

- Executing method calls is more time consuming than executing normal statements.

- **End of Slides**

End of Slides