Recursion

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

Recursion

- A Java method can call any other public Java method
  - `main()` is just a method itself, and we have called other methods from it
  - Thus, a method should be able to call itself
    - We call this a **Recursive Call**
  - At first thought this semms odd or even impossible
    - Why should we want to do this?
  - However, it will be very useful in a lot of different programming approaches

Recursion

- Before we look at the programming in detail, let's try to get the idea down, using math
- Some mathematical functions are in fact defined recursively
  - Example: Factorial

$$n! = n \times (n-1)!$$

  - Note that the function is fedined in terms of itself, but with an important change:
    - The **recursive call** is smaller in size $(n-1)$ than the original call n
    - This is vital to recursion being viable
  - Let's trace $4!$ on the board in this way to see what happens
    - But let's stop when we reach $-3!$

Recursion

- What we are missing in the previous slide is a condition that allows the recursion to stop
  - Every recursive algorithm must have some terminating condition, to keep it from recursing **forever**
  - We call this the **base case**
- What is the base case for factorial?

$$0! = 1$$

- This now allows us to complete our algorithm:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{Otherwise} \end{cases}$$

Recursion

- Three important rules for any recursive algorithm:
  1. There must be some **recursive case**, in which the algorithm **calls itself**
  2. There must be come **base case**, in which no recursive call is made
  3. The **recursive calls must leand** eventually **to the base case**
     - Usually by "reducing" teh problem size in some way
- Do not forget these rules!!!

More Recursion

- Let's look at another example:
  - Calculating an integer power of another integer

$$m^n = m \times m^{n-1} \quad \text{where } n > 0$$

  - Do not forget about the **base case**

$$m^n = 1 \quad \text{where } n = 0$$

  - The actions we take are slightly different from factorial, but the basic idea is similar
- Trace this on board
  - Note how **ifrst call made** is **last call to complete**
  - This is important in the implementation of recursion

Implementing Recursion

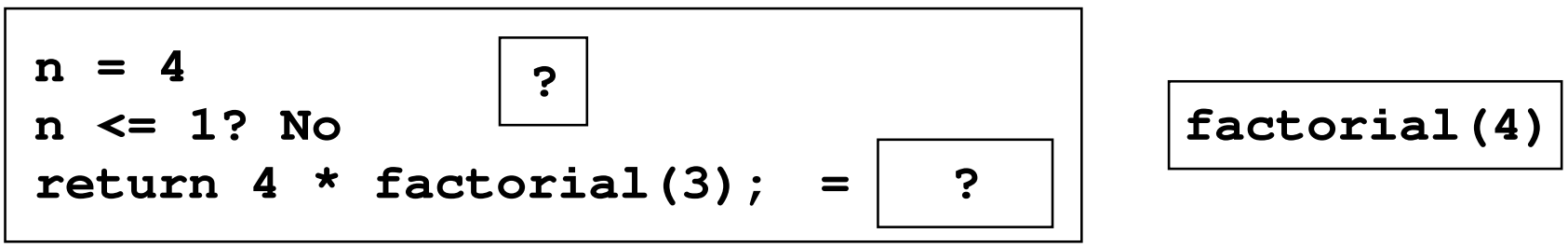
- So, how do we implement recursion?
  - Luckily, the computer code is very similar to the mathematical functions
  - COnsider factorial below:
    - Note that the recursive call is made within the return statement
    - This is fine – return is done **after** call completes

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

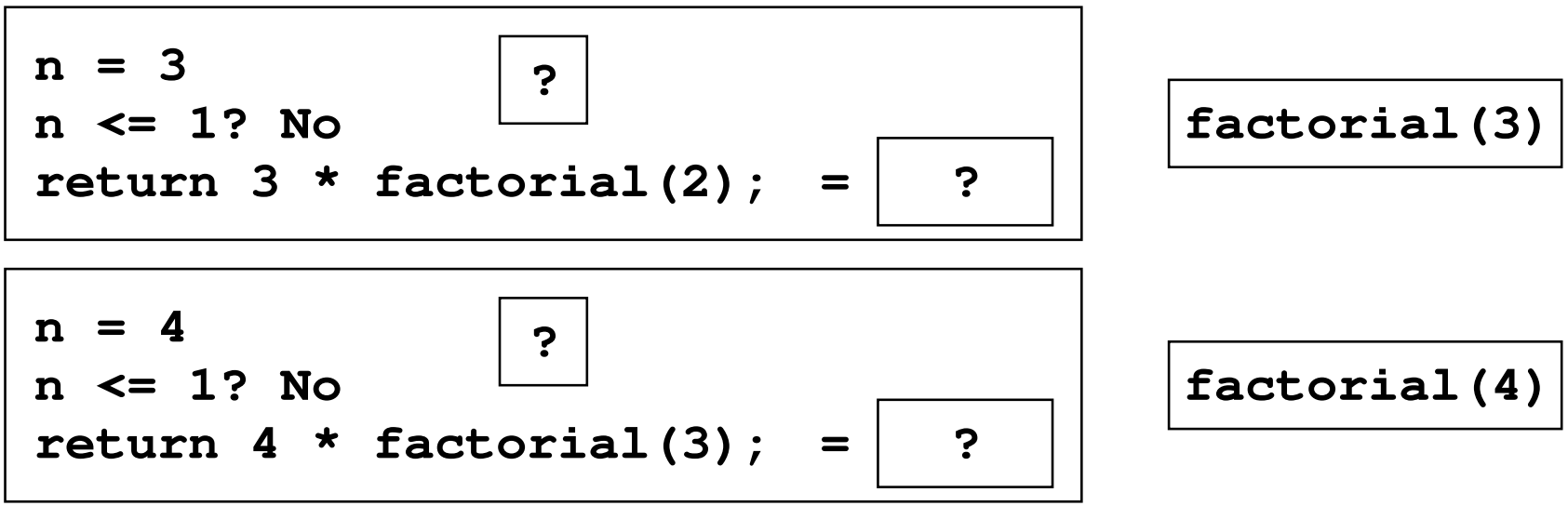
Implementing Recursion

- How does recursion actually work?
  - Each time a method is called, an **activation record (AR)** is allocated for it
    - This consists of memory for the parameters and local variables used in the method
  - Each new activation record is placed on the top of the **run-time stack**
  - When a method terminates, its activation record is removed from the top of the run-time stack
  - Thus, the first activation record placed onto the stack is the last one removed

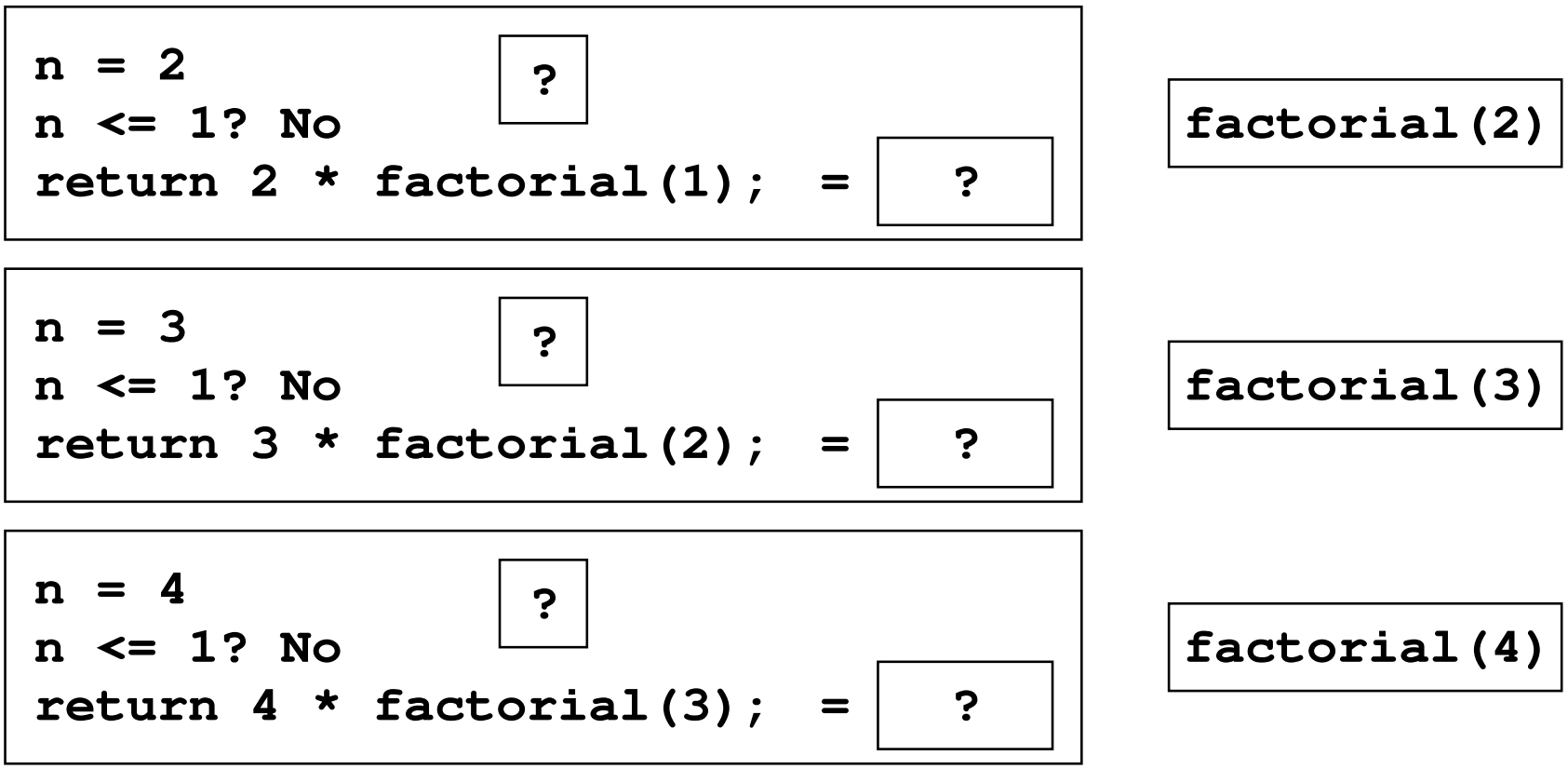
Activation Record and Run-Time Stack



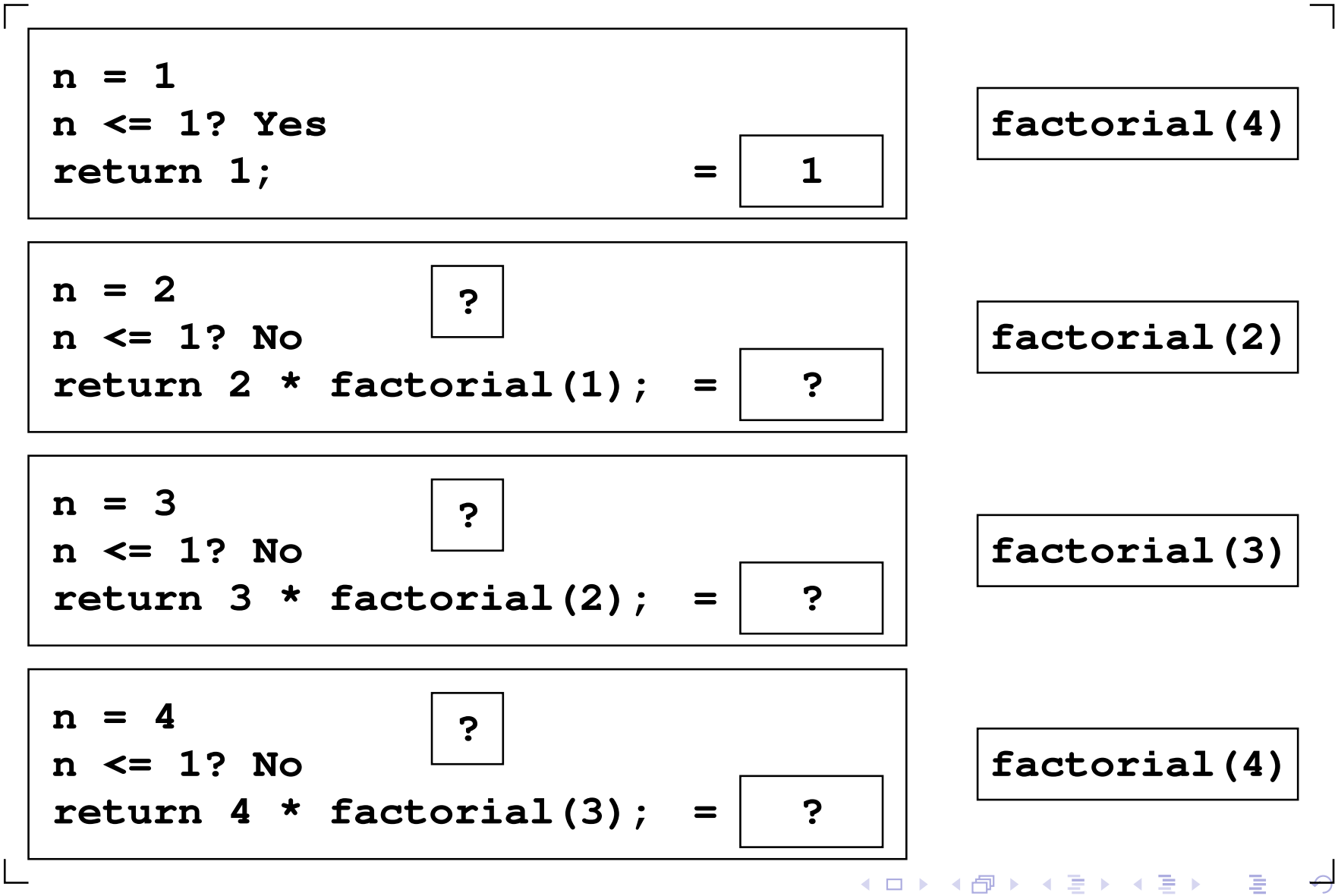
Activation Record and Run-Time Stack



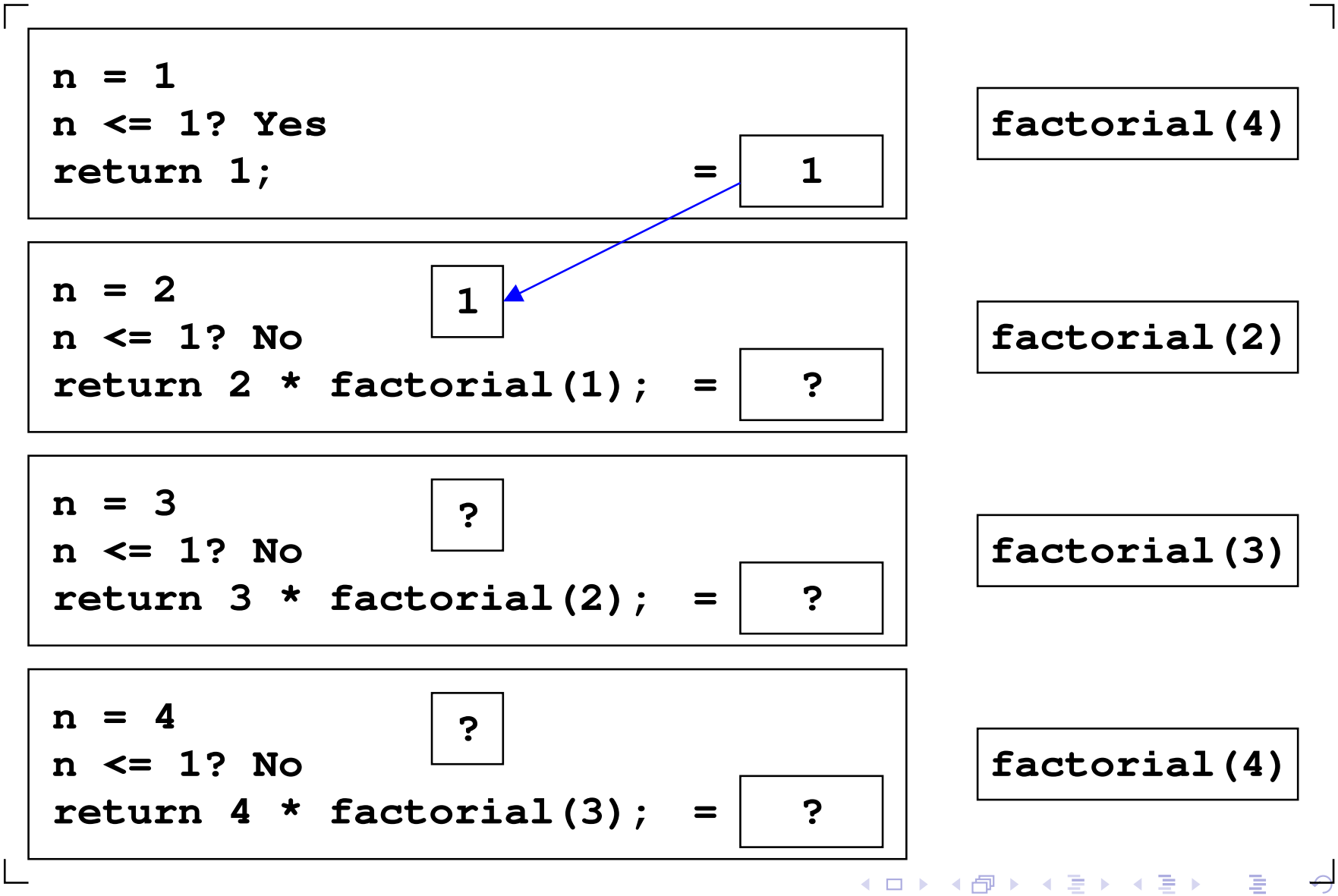
Activation Record and Run-Time Stack



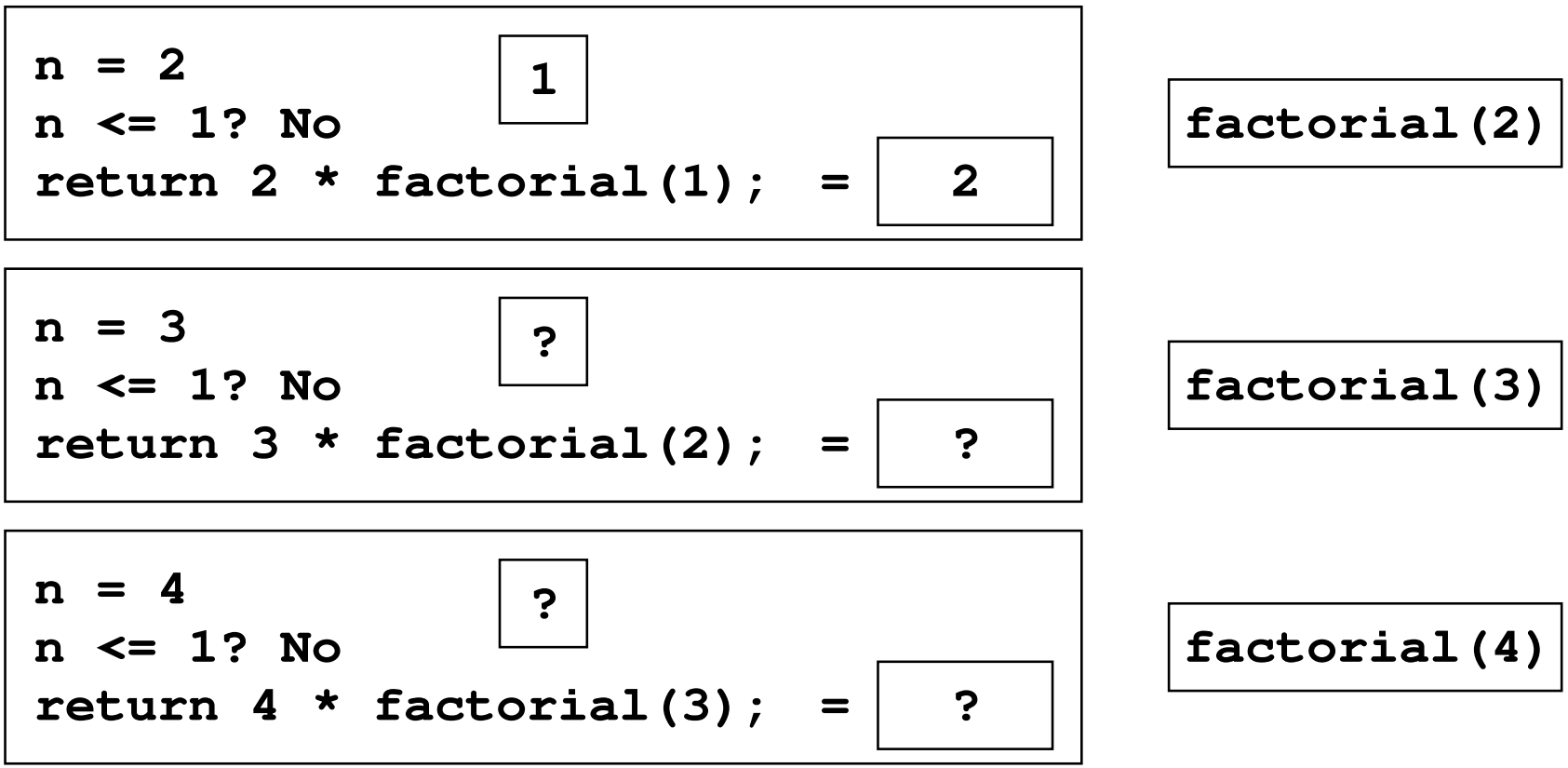
Activation Record and Run-Time Stack



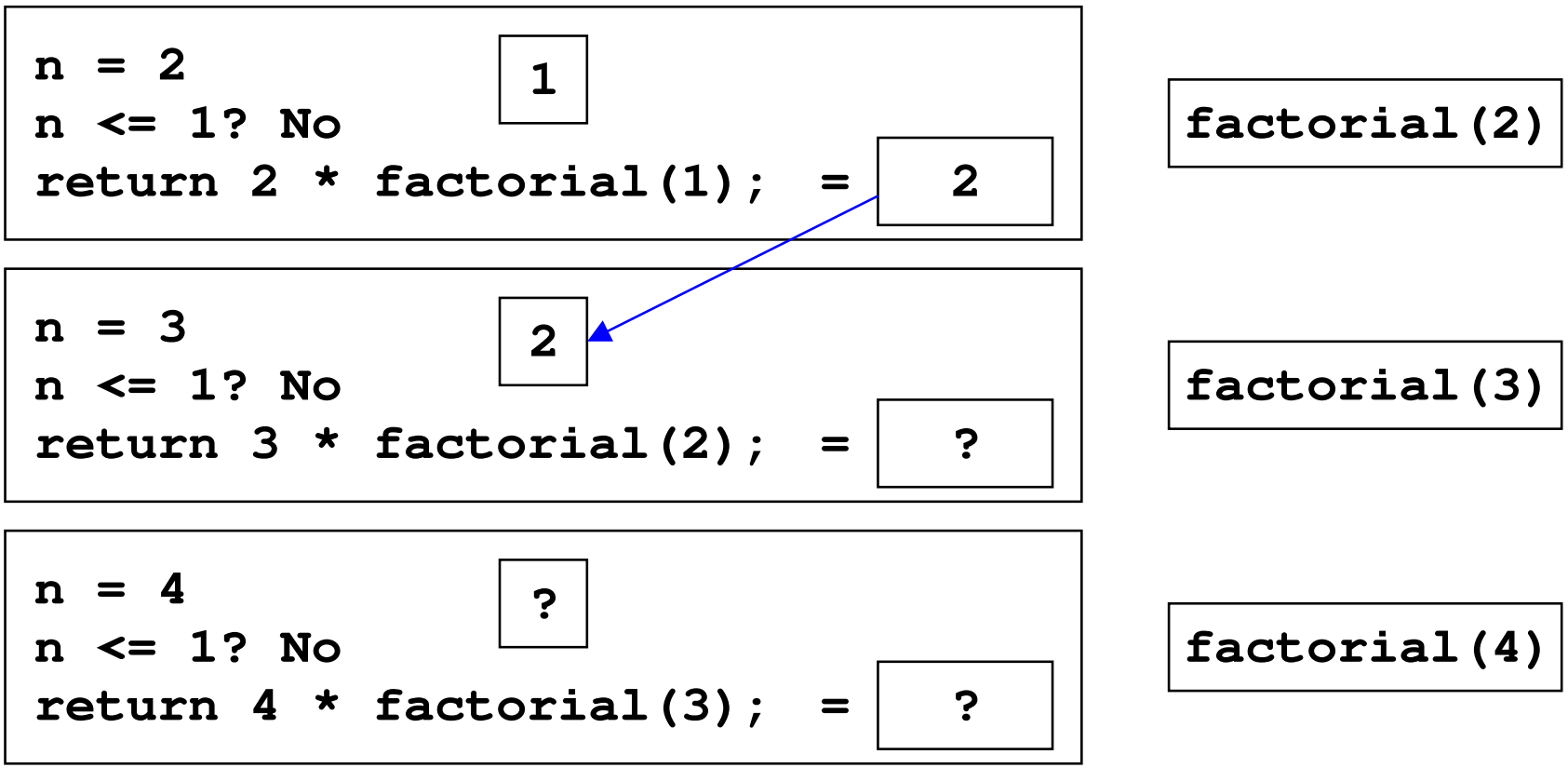
Activation Record and Run-Time Stack



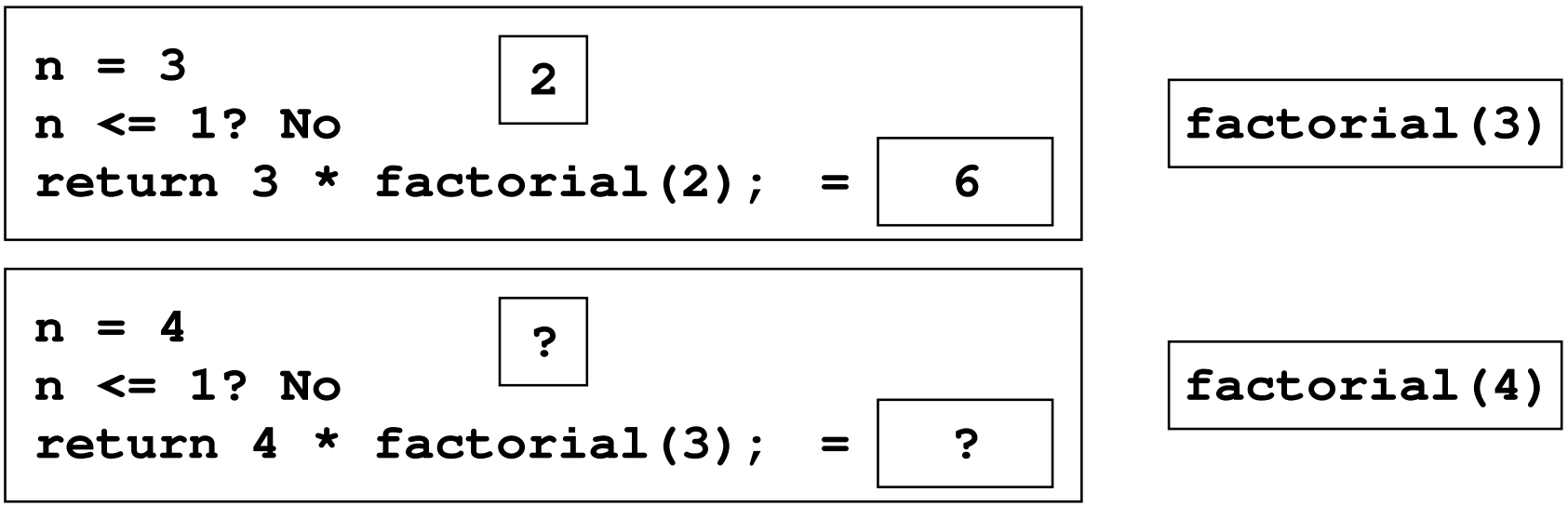
Activation Record and Run-Time Stack



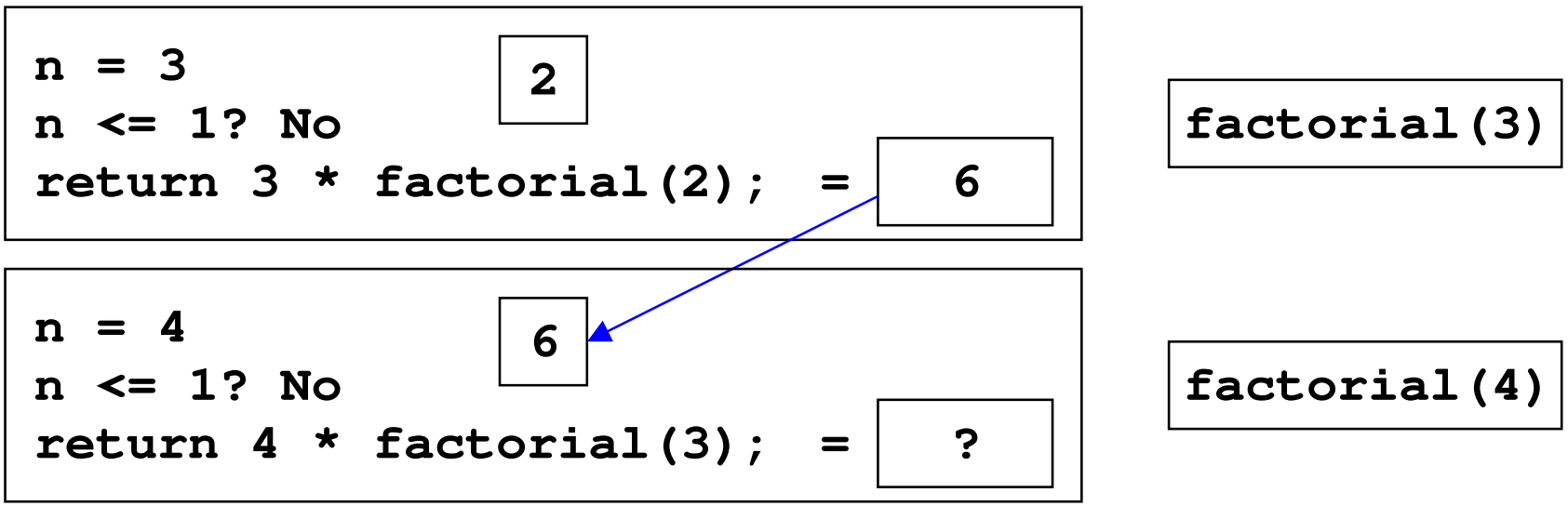
Activation Record and Run-Time Stack



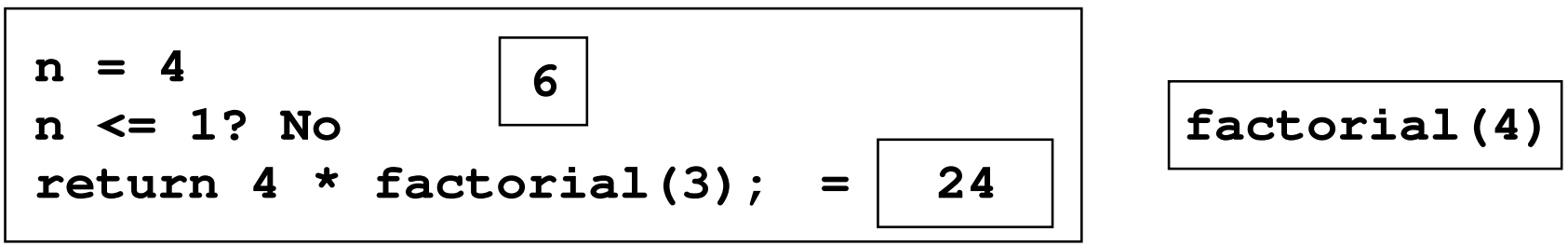
Activation Record and Run-Time Stack



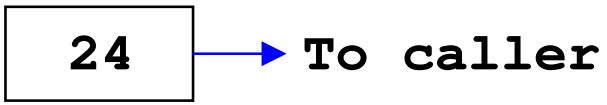
Activation Record and Run-Time Stack



Activation Record and Run-Time Stack



Activation Record and Run-Time Stack



Recursion vs Iteration

- Some recursive algorithms can also be easily implemented **with loop**
  - Both factorial and power can easily be done in this way
  - When possible, it is usually better to use iteration, since we do not have the overhead of the run-time stack (that we just saw on the previous slides)
- Other recursive algorithms are very difficult to do any other way (Tower of Hanoi in the textbook)
- You will see more about recursion in CS 0445

More Recursion

- Consider again **functional abstraction**
  - User of a method does not need to know how it is implemented
  - However, often recursive methods require more parameters that equivalent iterative methods
    - Extra parameters enable the testing for base cases
    - This can be problematic if the methods are part of an iterface, which specifies the method header
  - We can get around this by using an additional, non-recursive method
    - The additional method satisfies the required header
    - It then calls the recursive method, adding any extra nedded parameters

More Recursion

- For example, consider a method to reverse an array of `int`
  - The header might be something like

```
public static void reverse(int[] data)
```

  - However, to implement this recursively, we need extra parameters to keep track of the logical beginning and end of the array
  - These extra parameters can be added in a call to the recursive method

```
public static void reverse(int[] data)
{
    recursive_reverse(data, 0, data.length - 1);
}
```

- Let's look at the `recursive_reverse()`

recursive_reverse()

- To reverse an array of size 6, we need to perform the following:
  1. Swap data between index 0 and 5
  2. Swap data between index 1 and 4
  3. Swap data between index 2 and 3
- In other words, to reverse an array
  1. Swap data between the `firstIndex` and the `lastIndex`
  2. Reverse the same array but
     - change the `firstIndex` to `firstIndex + 1`
     - change the `lastIndex` to `lastIndex - 1`
- This is why the signature of the `recursive_reverse()` method is

```
public static void recursive_reverse(int[] data,
                                      int firstIndex,
                                      int lastIndex)
```