Recursion

Objectives

- Explain the underlying concepts of recursion
- Examine recursive methods and unravel their processing steps
- Explain when recursion should and should not be used
- Demonstrate the use of recursion to solve problems

Recursive Definitions

- ***Recursion***: defining something *in terms of itself*
- ***Recursive definition***
  - Uses the word or concept being defined ***in the definition itself***
  - Includes a ***base case*** that is defined directly, ***without*** self-reference

Recursive Definitions

- ***Example***: define a ***group of people***
  - ***Iterative definition:***
    a *group* is 2 people, or 3 people, or 4 people, or …
  - ***Recursive definition:***
    a *group* is: 2 people

    or, a *group* is: a *group* plus one more person
    - The concept of a group is used to define itself!
    - The ***base case*** is "a group is 2 people"

Exercise

- Give an iterative and a recursive definition of a ***sequence of characters*** e.g. CS 1027
  - ***Iterative definition****:* a *sequence of characters* is ?
  - ***Recursive definition****:* a sequence of characters is ?

Recursive Definitions

- ***Example***: consider the following ***list of numbers***:

  24, 88, 40, 37

  It can be defined recursively:

  list of numbers **:** is a **number**
  or a **number comma** ***list of numbers***

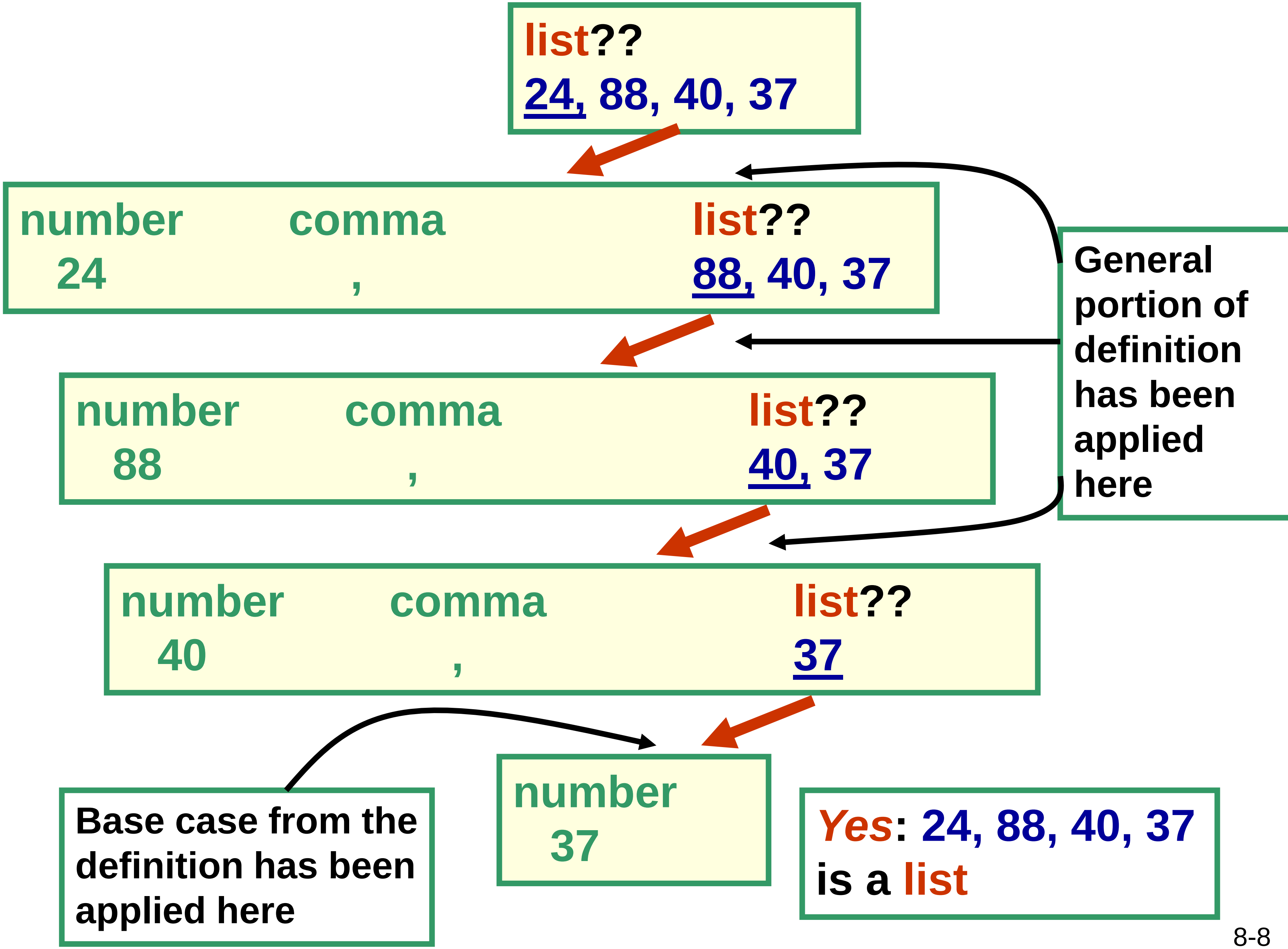
  i.e. It is defined to be a single number, or a number followed by a comma followed by a ***list of numbers***

Tracing a Recursive Definition

- To determine whether the sequence **24, 88, 40, 37** is a ***list of numbers***, apply the recursive portion of the definition:

  24 is a **number** and "**,**" is a **comma**,
  so **24,** **88, 40, 37** is a ***list of numbers***
  if and only if
  88, 40, 37 is a ***list of numbers***

- Apply the same part of the definition to the sequence **88, 40, 37**
- …
- Eventually, we'll need to apply the base case of the definition



list??
24, 88, 40, 37
number
24
comma
,
list??
88, 40, 37
General portion of definition has been applied here
number
88
comma
,
list??
40, 37
number
40
comma
,
list??
37
Base case from the definition has been applied here
number
37
Yes: 24, 88, 40, 37 is a list

Recursive Definitions

- A recursive definition consists of two parts:
  - The ***base case***: this defines the "***simplest***" case or starting point
  - The ***recursive part***: this is the "***general case***", that describes all the other cases in terms of "***smaller***" versions of itself
- Why is a base case needed?
  - A definition without a non-recursive part causes ***infinite recursion***

Discussion

- We can get information from our recursive definition by ***starting*** at the base case, for example:
  - 2 people form a group (base case)
  - So, 2 + 1 or 3 people form a group
  - So, 3 + 1 or 4 people form a group
  - etc.
- We can also get information by ***ending*** at the base case, for example:
  - Do 4 people form a group?

More Recursive Definitions

- Mathematical formulas are often expressed recursively
- ***Example***: the formula for ***factorial***

  for any positive integer ***n***, ***n!*** *(n factorial) is defined to be the product of all integers between* ***1*** *and* ***n*** *inclusive*
- Express this definition recursively

1! = 1 (the base case)

n! = n * (n-1)! for n>=2

- Now determine the value of **4!**

Discussion

- ***Recursion*** is an alternative to ***iteration***, and can be a very powerful problem-solving technique
- What is ***iteration***? repetition, as in a loop
- What is ***recursion***? defining something in terms of a ***smaller*** or ***simpler*** version of itself (why smaller/simpler?)

Recursive Programming

- ***Recursion*** is a programming technique in which a method can ***call itself*** to solve a problem
- A method in Java that invokes itself is called a ***recursive method***, and must contain code for
  - The ***base case***
  - The ***recursive part***

Example of Recursive Programming

- Consider the problem of computing the sum of all the numbers between **1** and **n** inclusive

  e.g. if **n** is **5**, the sum is

  1 + 2 + 3 + 4 + 5

- How can this problem be expressed recursively?

  Hint: the above sum is the same as

  5 + 4 + 3 + 2 + 1

Recursive Definition of Sum of 1 to n

$$\sum_{k=1}^{n} k = n + \sum_{k=1}^{n-1} k \qquad \text{for } n > 1$$

This reads as:

the sum of 1 to n = n + the sum of 1 to n-1

What is the base case?
the sum of 1 to 1 = 1

Trace Recursive Definition of Sum of 1 to n

$$\sum_{k=1}^{n} k = n + \sum_{k=1}^{n-1} k = n + (n-1) + \sum_{k=1}^{n-2} k$$

$$= n + (n-1) + (n-2) + \sum_{k=1}^{n-3} k$$

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

A Recursive Method for Sum

```
public static int sum (int n)
{
   int result;
   if (n == 1)
        result = 1;
   else
        result = n + sum (n-1);
   return result;
}
```

How Recursion Works

- What happens when ***any*** method is called?
  - A ***call frame*** *or* ***activation record*** is set up
  - That call frame is pushed onto the ***runtime stack*** *or* ***execution stack***
- What happens when a recursive method "***calls itself*** "? It's actually just like calling any other method!
  - An ***activation record*** is set up
  - That activation record ame is pushed onto the ***execution stack***

How Recursion Works

- Note: For a recursive method, how many copies of the code are there?
  - Just one! (like any other method)
- When does the recursive method stop calling itself?
  - When the base case is reached
- What happens then?
  - ***That invocation*** of the method completes, its activation record is popped off the execution stack, and control returns to ***the method that invoked it***

How Recursion Works

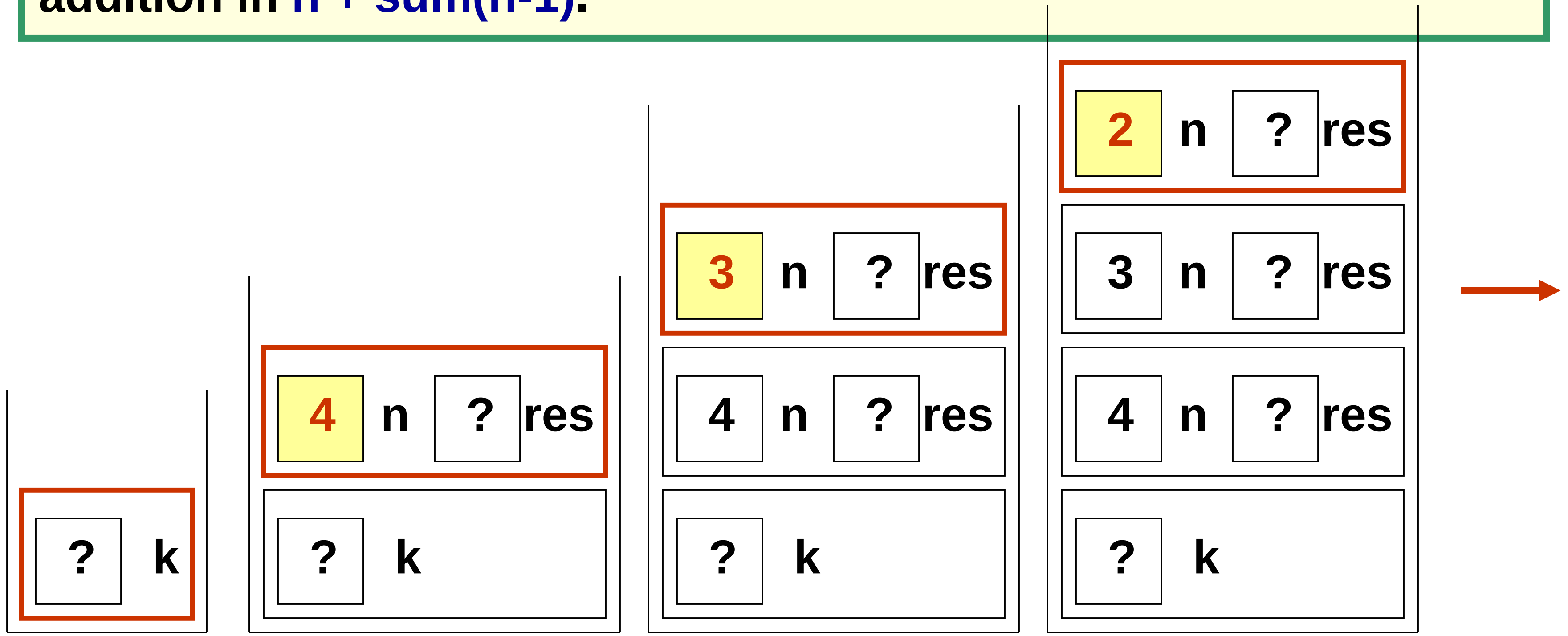
- But which method invoked it? the ***previous invocation*** of the recursive method
  - This method now completes, its activation record is popped off the execution stack, and control returns to ***the method that invoked it***
- And so on until we get back to the first invocation of the recursive method

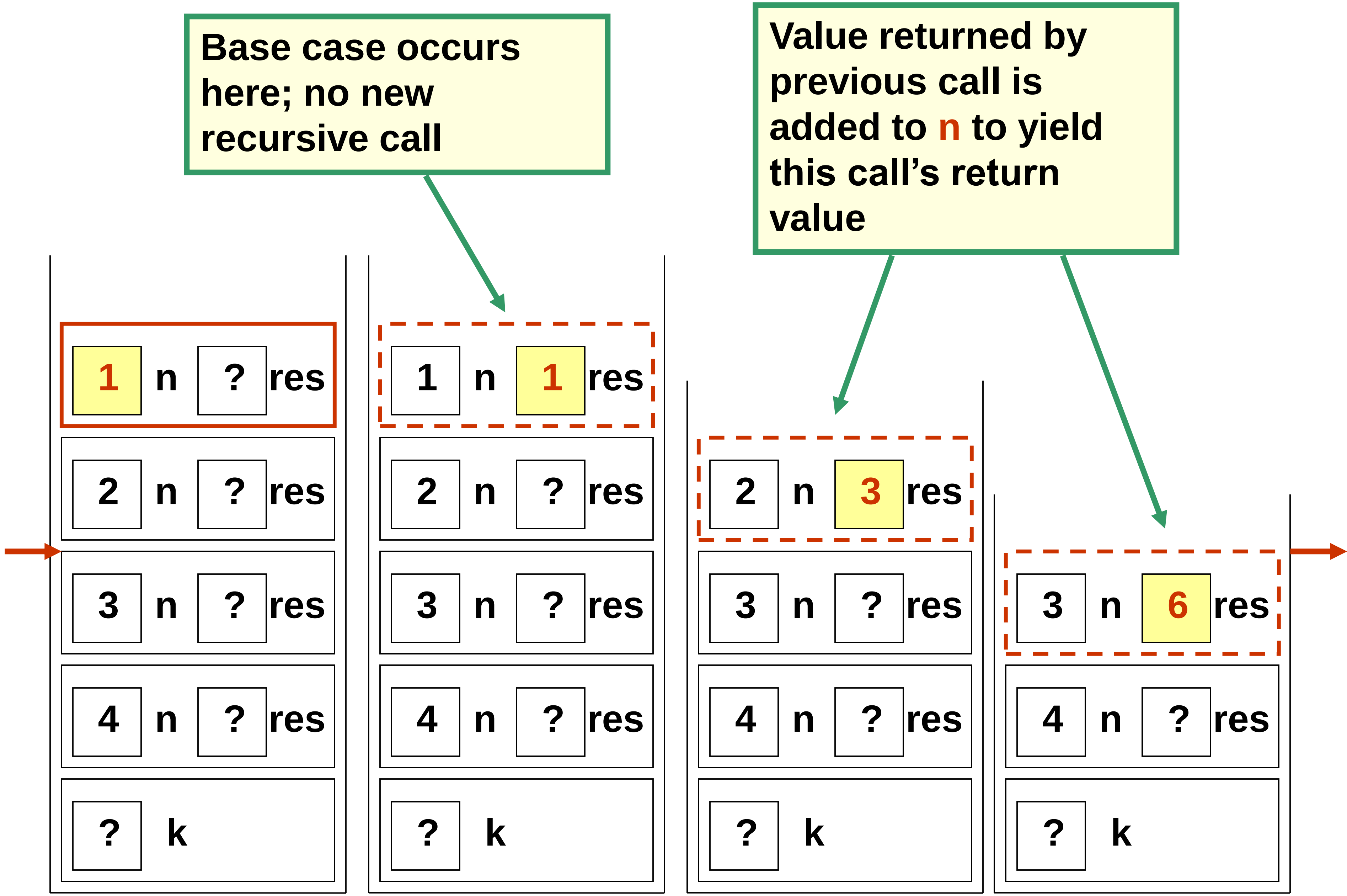
Tracing **int k = sum(4);**

Call is made from main( ).

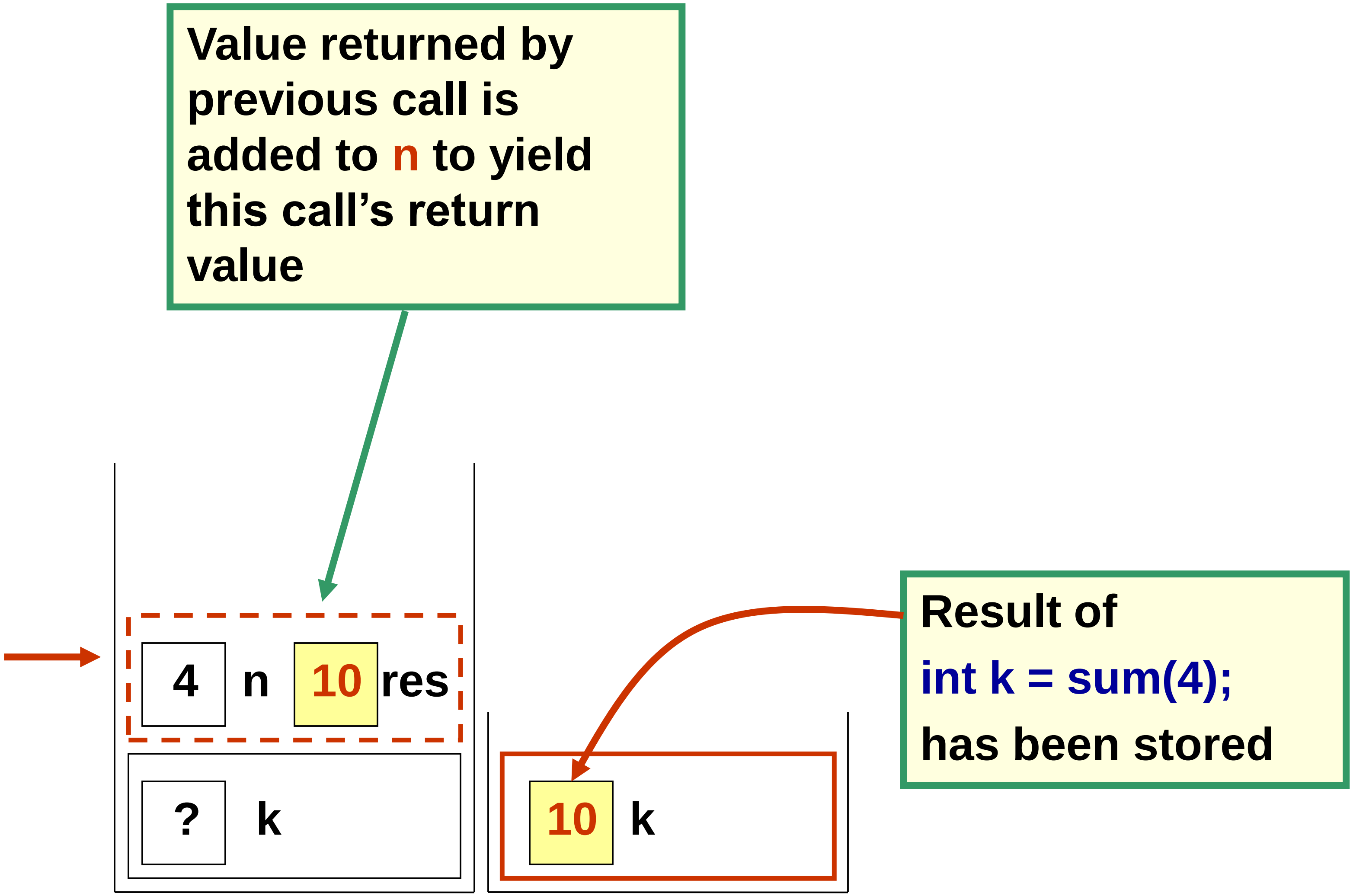
Bottom activation record on the execution stack is for the main program; all others are for calls to sum( ). The stack is redrawn at each call to sum( ), and just before each return.

Main program call returns to the OS; all others return to the addition in n + sum(n-1).





Base case occurs here; no new recursive call
Value returned by previous call is added to n to yield this call's return value
1 n ? res
2 n ? res
3 n ? res
4 n ? res
? k
1 n 1 res
2 n ? res
3 n ? res
4 n ? res
? k
2 n 3 res
3 n ? res
4 n ? res
? k
3 n 6 res
4 n ? res
? k


Value returned by previous call is added to n to yield this call's return value
4
n
10
res
?
k
10
k
Result of
int k = sum(4);
has been stored

Discussion: Recursion vs. Iteration

- Just because we ***can*** use recursion to solve a problem, doesn't mean we *should*!
- Would you use iteration or recursion to compute the sum of 1 to n? Why?

Exercise: Factorial Method

- Write an iterative method to compute the factorial of a positive integer.

- Write a recursive method to compute the factorial of a positive integer.

- Which do you think is faster, the recursive or the iterative version of the factorial method?

Example: Fibonacci Numbers

- ***Fibonacci numbers*** are those of the sequence

  1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

- Define them recursively:

  fib(1) = 1
  fib(2) = 1
  fib(n) = fib(n – 1) + fib(n – 2) **for n > 2**

- This sequence is also known as the solution to the ***Multiplying Rabbits Problem*** ☺

A Recursive Method for Fibonacci Numbers

```
// precondition (assumption) : n > = 1

public static int rfib (int n) {
   if ((n == 1) || (n == 2))
         return 1;
   else
         return rfib(n – 1) + rfib(n – 2);
}
```

An Iterative Method for Fibonacci Numbers

```
public static int ifib(int n) {
    if ((n == 1) || (n == 2))
        return 1;
    else {
        int prev = 1, current = 1, next;
        for (int i = 3; i <= n; i ++) {
            next = prev + current;
            prev = current;
            current = next;
        }
    return next;
    }
```

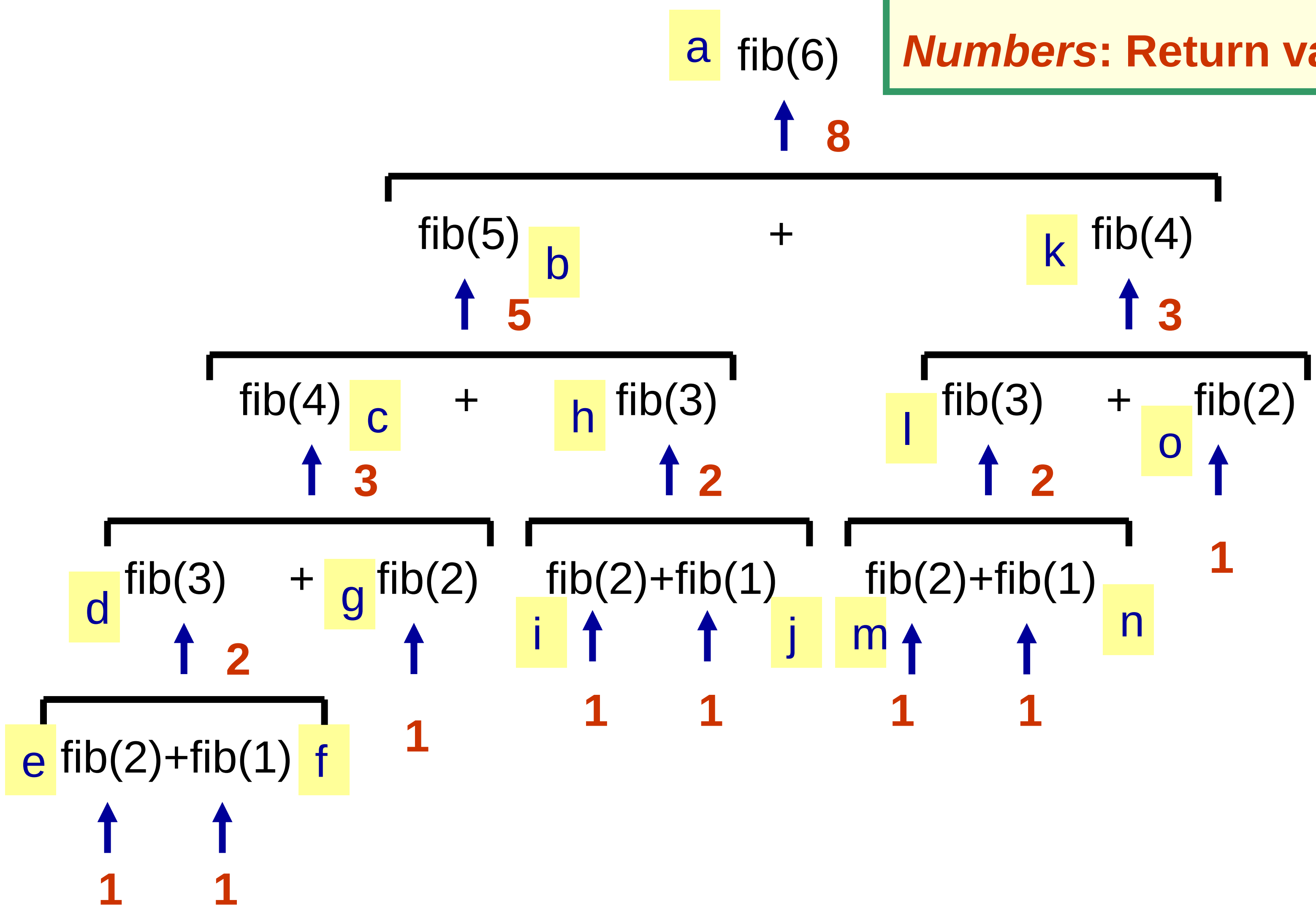
Discussion

- Which solution looks simpler, the recursive or the iterative?
- Which one is (***much***) faster? Why?
- ***Note***: recursive and iterative code for Fibonacci are both online - try running them both, and *time them*!

Evaluating fib(6)

***Letters*: Give <u>order</u> of calls**

***Numbers*: Return values**



Useful Recursive Solutions

- ***Quicksort for sorting a set of values***
- Backtracking problems in Artificial Intelligence
- Formal language definitions such as ***Backus-Naur Form (BNF)***

  <ident> ::= <letter> | <ident><letter> |
  <ident><digit>

  etc.
- Evaluating algebraic expressions in postfix form *(how did we do this earlier?)*
- etc.

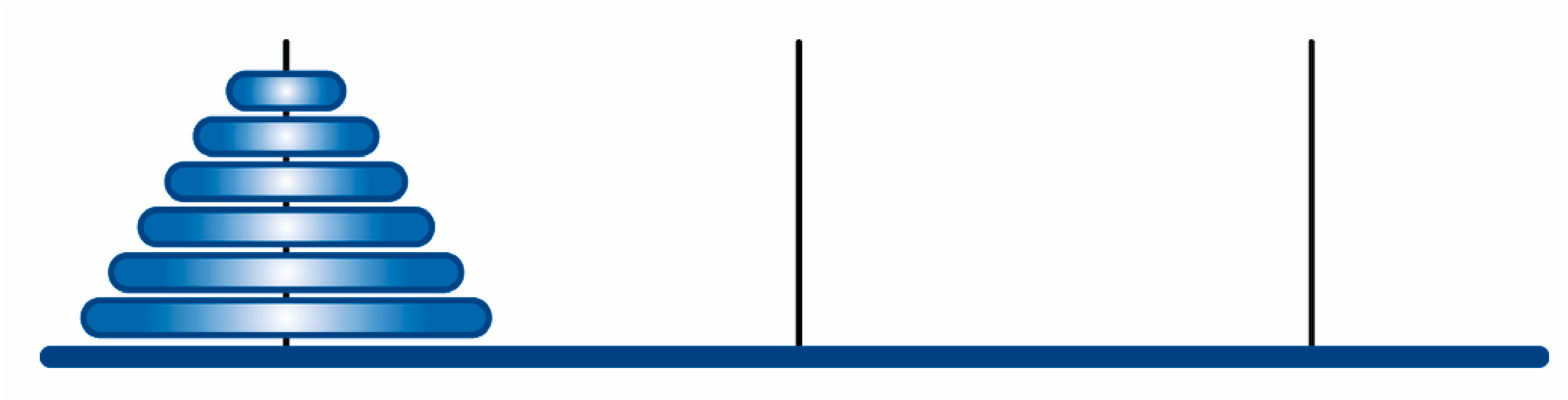
Recursive Solutions

- For some problems, recursive solutions are simpler and more ***elegant*** than iterative solutions
- ***Classic example***: ***Towers of Hanoi***
  - Puzzle invented in the 1880’s by a mathematician named Edouard Lucas
  - Based on a legend for which there are many versions, but they all involve monks or priests moving 64 gold disks from one place to another. When their task is completed, the world will end …

The Towers of Hanoi

- The ***Towers of Hanoi*** puzzle is made up of
  - Three vertical pegs
  - Several disks that slide onto the pegs
  - The disks are of varying size, initially placed on one peg with the largest disk on the bottom and increasingly smaller disks on top

The Towers of Hanoi Puzzle

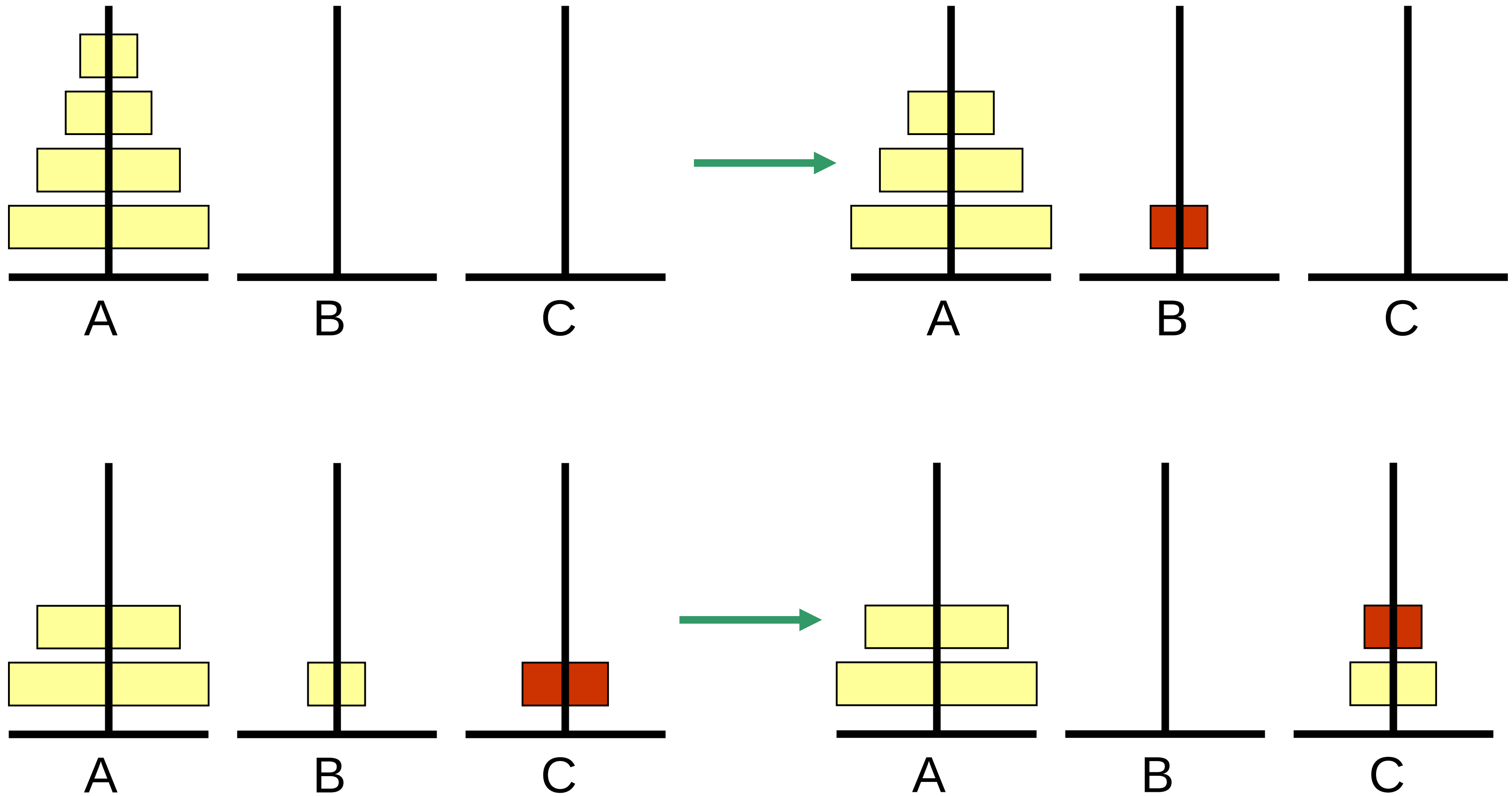


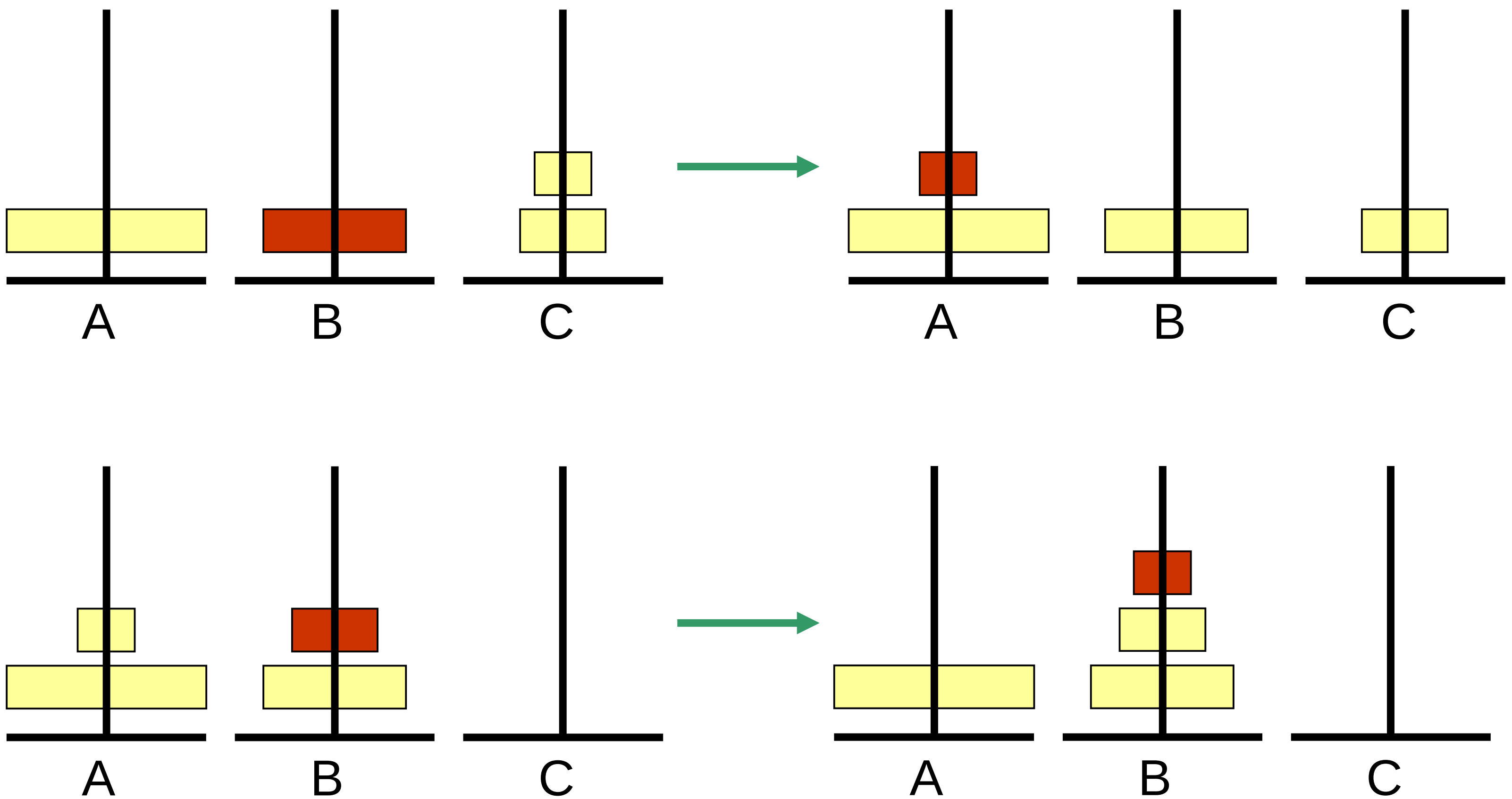
The Towers of Hanoi

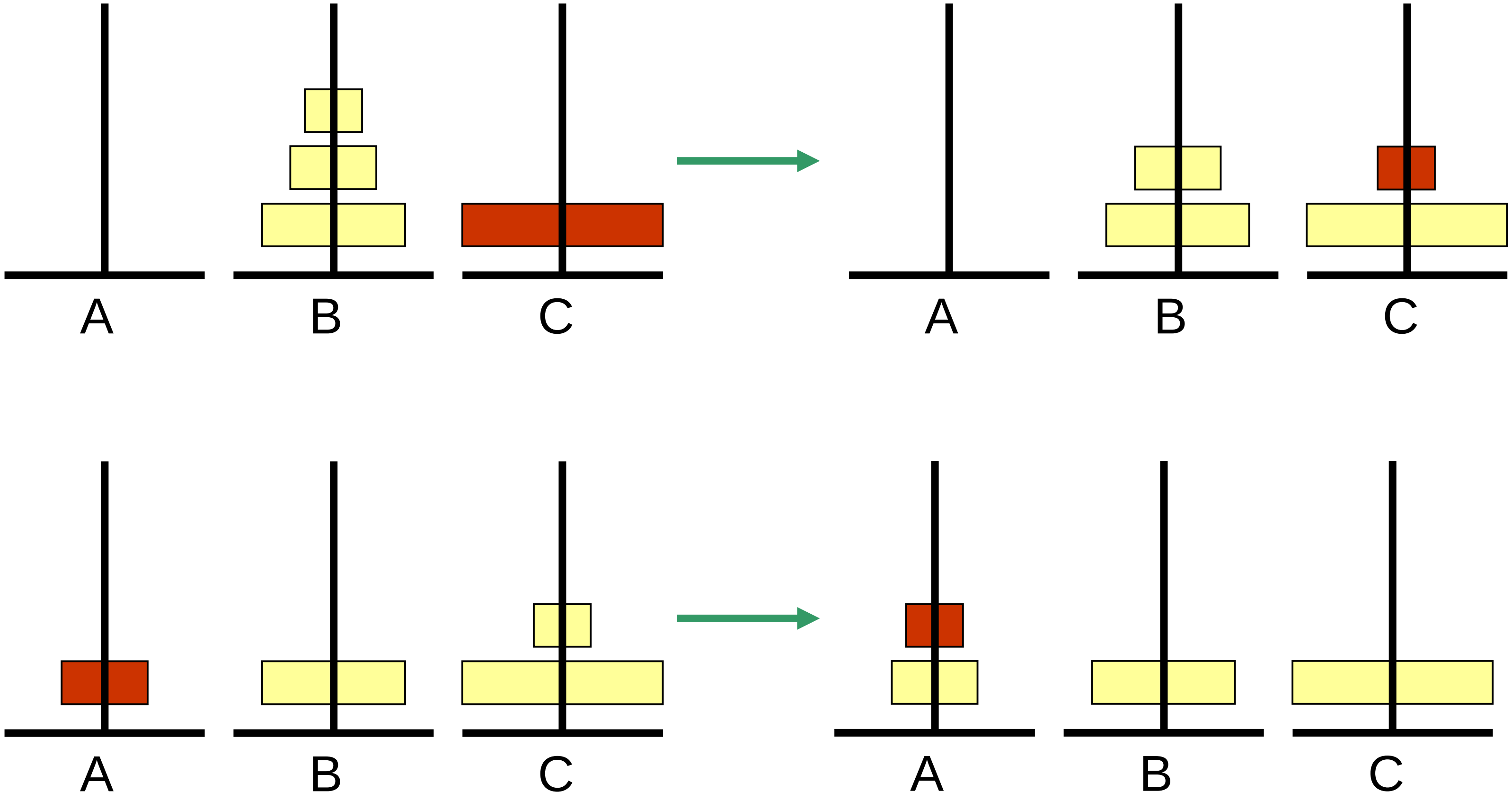
- ***Goal***: move all of the disks from one peg to another following these rules:
  - Only ***one*** disk can be moved at a time
  - A disk ***cannot*** be placed on top of a smaller disk
  - All disks must be on some peg (except for the one in transit)

Towers of Hanoi Solution: 4 disks

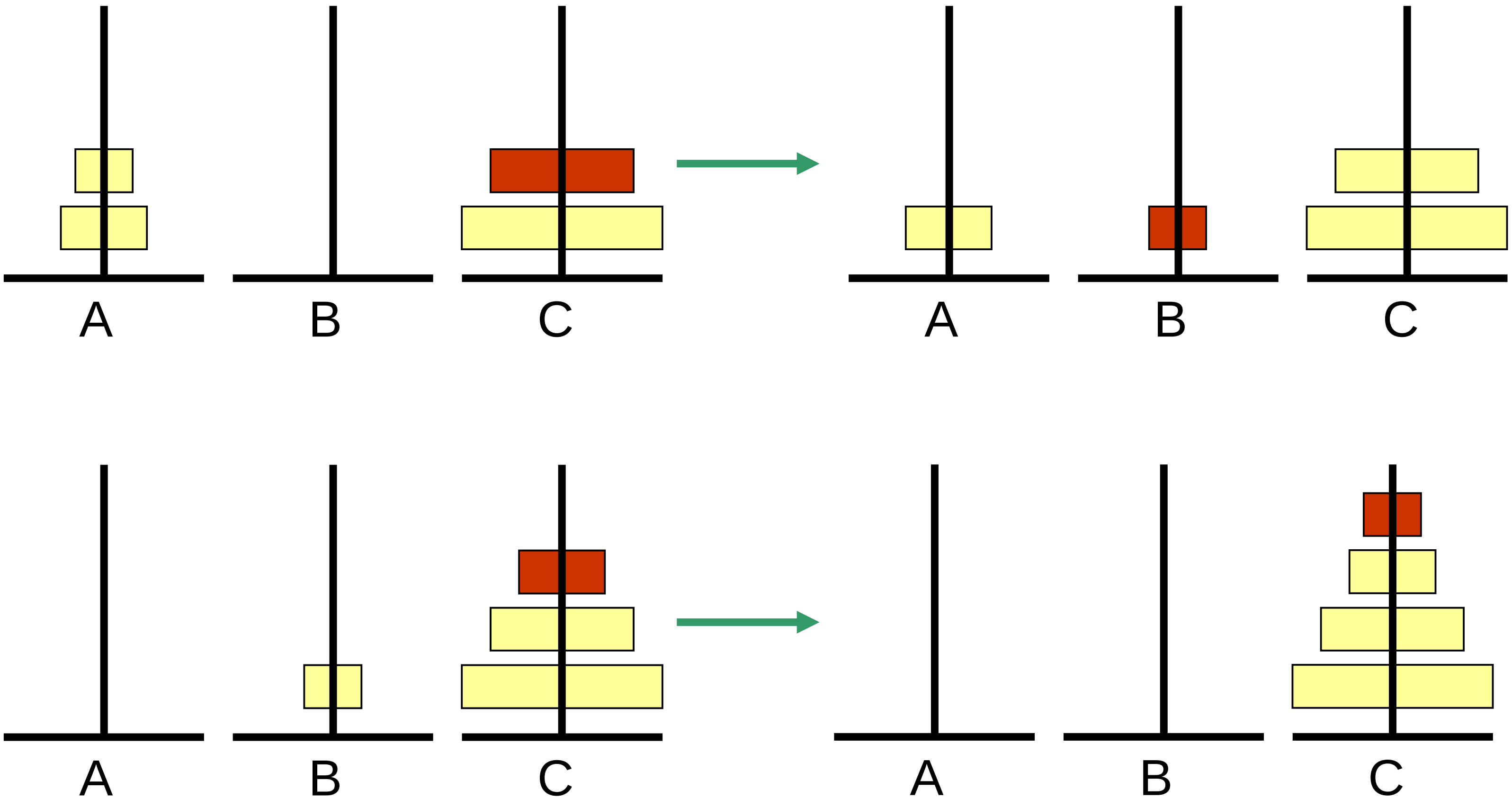
Goal: Move the disks from peg A to peg C




A
B
C
A
B
C
A
B
C
A
B
C



A
B
C
A
B
C
A
B
C
A
B
C


A
B
C
A
B
C
A
B
C
A
B
C

Towers of Hanoi Recursive Solution

- To move a stack of **n** disks from the original peg to the destination peg:
  - move the topmost **n-1** disks from the original peg to the extra peg
  - move the largest disk from the original peg to the destination peg
  - move the **n-1** disks from the extra peg to the destination peg
- The base case occurs when moving just the smallest disk (that is, when solving the **1-disk** problem)

Towers of Hanoi Recursive Solution

- Note that the number of moves increases ***exponentially*** as the number of disks increases!
  - So, how long will it take for the monks to move those 64 disks?
- The recursive solution is simple and elegant to express (and program); an iterative solution to this problem is much more complex
- See ***SolveTowers.java***, ***TowersOfHanoi.java***

Analyzing Recursive Algorithms

- ***Analyzing a loop***:
  determine the number of operations in each iteration of the loop and add it over the number of times the loop is executed

- ***Recursive analysis*** is similar:

  determine the number of operations in the method body and add it over the number of times the recursive call is *made*)

Analyzing Recursive Algorithms

- ***Example***: ***Towers of Hanoi***
  - Size of the problem? The number of disks n
  - Operations per call? A constant number
  - Except for the base case, *each* recursive call results in calling itself *twice* more
  - So, to solve a problem of **n** disks, we make **2^n-1** disk moves
  - Therefore the algorithm is **$O(2^n)$**, which is called ***exponential complexity***

Exercise

What is the time complexity of:

1. the recursive factorial method?
2. the iterative factorial method?
3. the recursive Fibonacci method?
4. the iterative Fibonacci method?