Recursion

Objectives

- Understand the underlying concepts of recursion
- Examine recursive methods and understand 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*

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

  A list of numbers can be defined recursively:

  list of numbers **:**

  - is a **number**
  - or a **number** **comma** ***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 will need to apply the base case of the definition

Is 24, 88, 40, 37 a list?

number **comma**
24 **,** **is <u>88,</u> 40, 37 a list?**

number **comma**
88 **,** **is <u>40,</u> 37 a list?**

number **comma**
40 **,** **is <u>37</u> a list?**

number
37

General portion of definition has been applied here

Base case from the definition has been applied here

***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***

More Recursive Definitions

- Mathematical formulas can often be expressed recursively
- ***Example***: the formula for ***factorial*** is:

  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

Discussion

- ***Recursion*** is an alternative to ***iteration***, and it is 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***, and
  - 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?

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 is equal to 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**

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

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

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

A Recursive Method for Sum

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

How Recursion Works

- What happens when a method is invoked?
  - An activation record, or call frame or frame is created
  - The activation record is pushed onto the runtime stack or execution stack
- Every time that the algorithm makes a recursive call a new activation record is created and pushed into the execution stack.

Activation Record

- An ***activation record*** contains:
  - Address to return to after method ends
  - Method’s formal parameter variables
  - Method’s local variables
  - Return value (if any)

Return address
Return value
Local variables
Formal Parameters

How Recursion Works

- When does the recursive method stop calling itself?
  - When the base case is reached
- What happens then?
  - That last 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 previous invocation of the method then 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

How Recursion Works

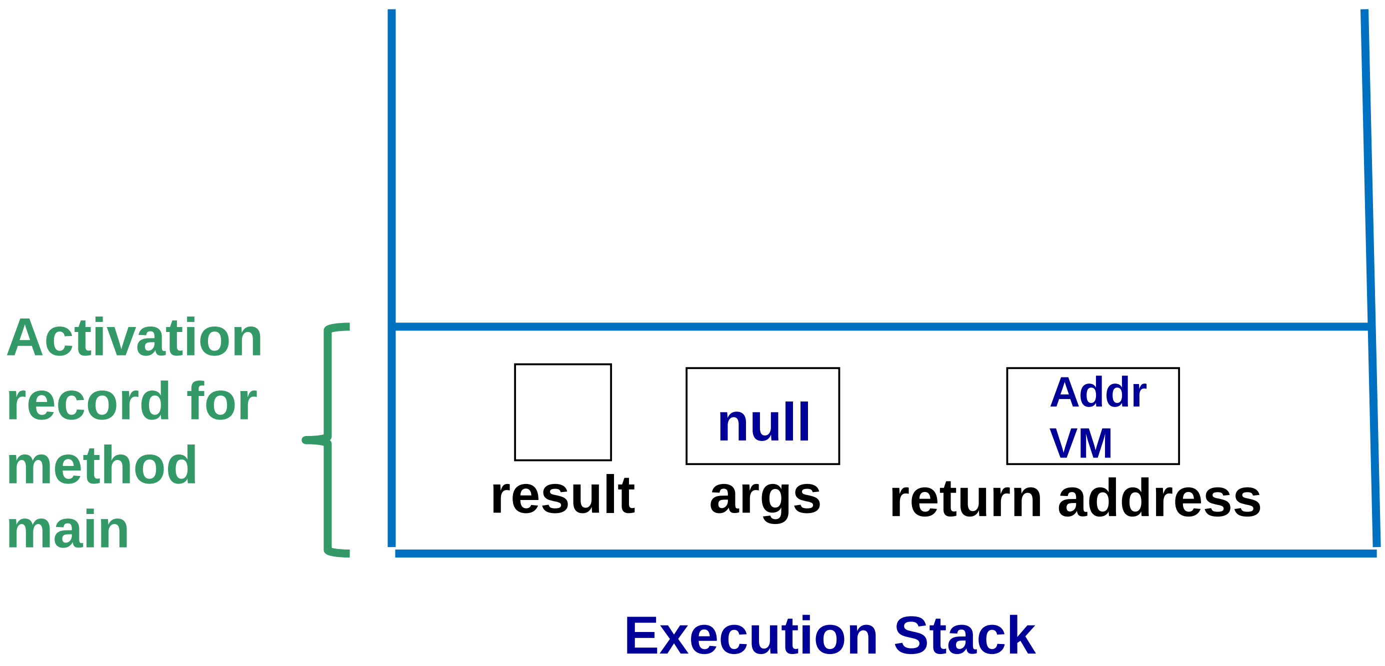
Consider the following program

```
public static void main (String[] args) {
    int result = sum(4); // Addr 1
}
```

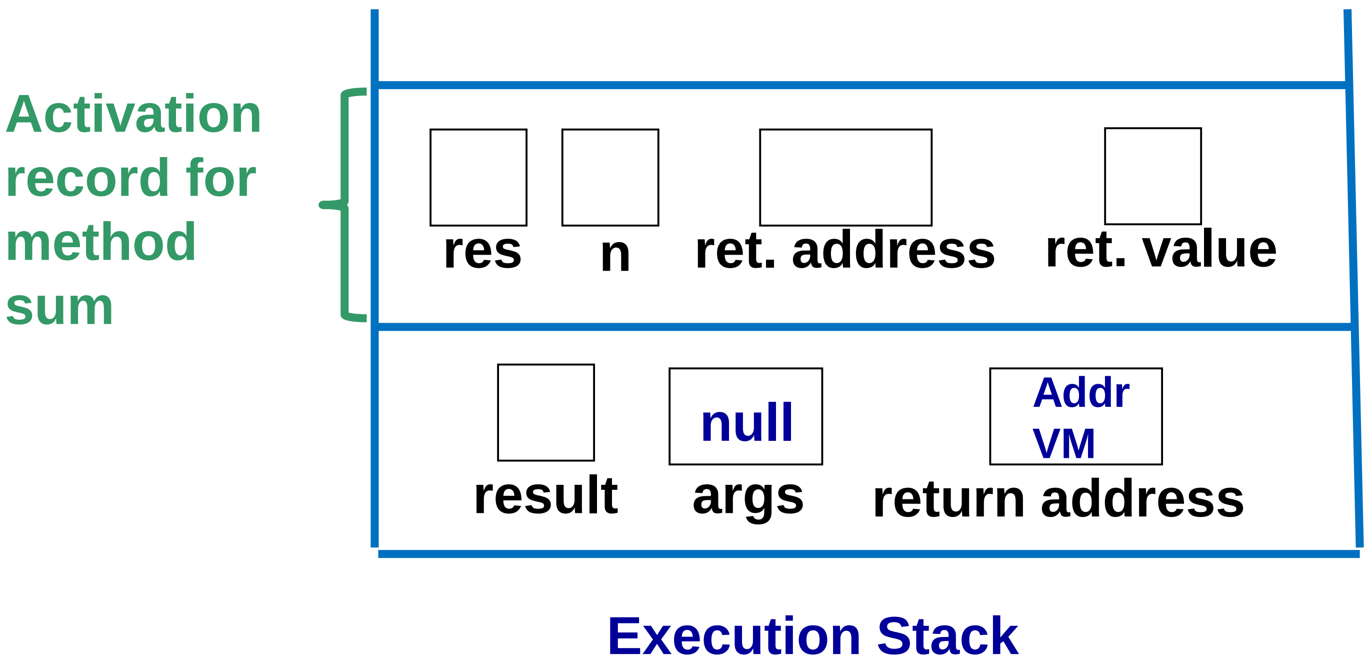
When the program is executed an activation record is created for method main. This activation record stores:

- The return address: in this case is the address of the part of the java virtual machine where the invocation to method main is made
- The variable result
- The parameter args

At this point the execution stack looks like the following figure. We assume that no parameter is passed to main, so args is null. Variable result has no value assigned to it yet, so we left its value blank. Addr VM denotes the address of the instruction of the virtual machine where method main was invoked.



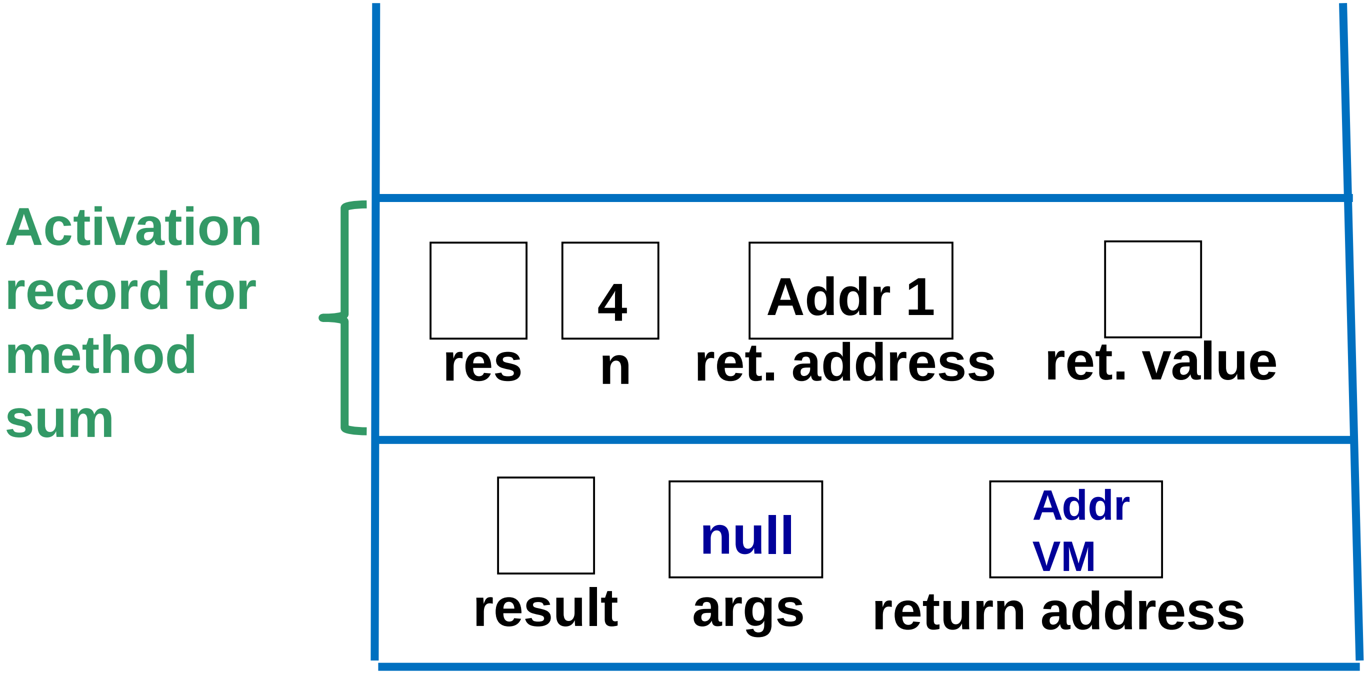
Once the activation record for method main has been created and the values of the parameters and return address have been stored in it, the execution of method main starts. The first and only statement of main invokes method sum. This causes the creation of another activation record, which is pushed into the execution stack:



Since method main invokes sum(4), the value of 4 is stored in n, the return address is the address of the statement

int result = sum(4); // Addr 1

where method sum is invoked. We will call this address, Addr 1. The value of variable res and the return value have not been computed yet:

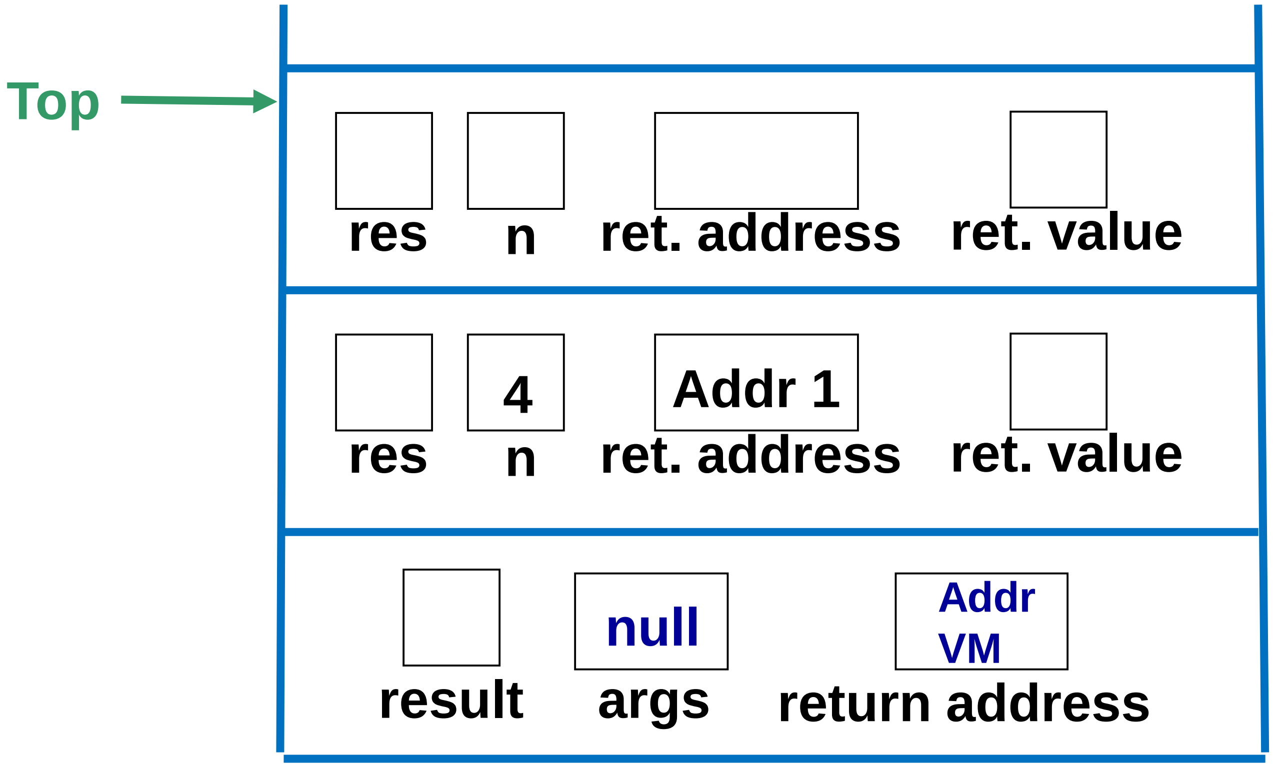


Execution Stack

Once the activation record has been created, the execution of method sum starts. Since n > 1, the statement

res = n + sum (n-1); // Addr 2

is executed. As this statement invokes method sum, a new activation record is created and pushed into the stack:

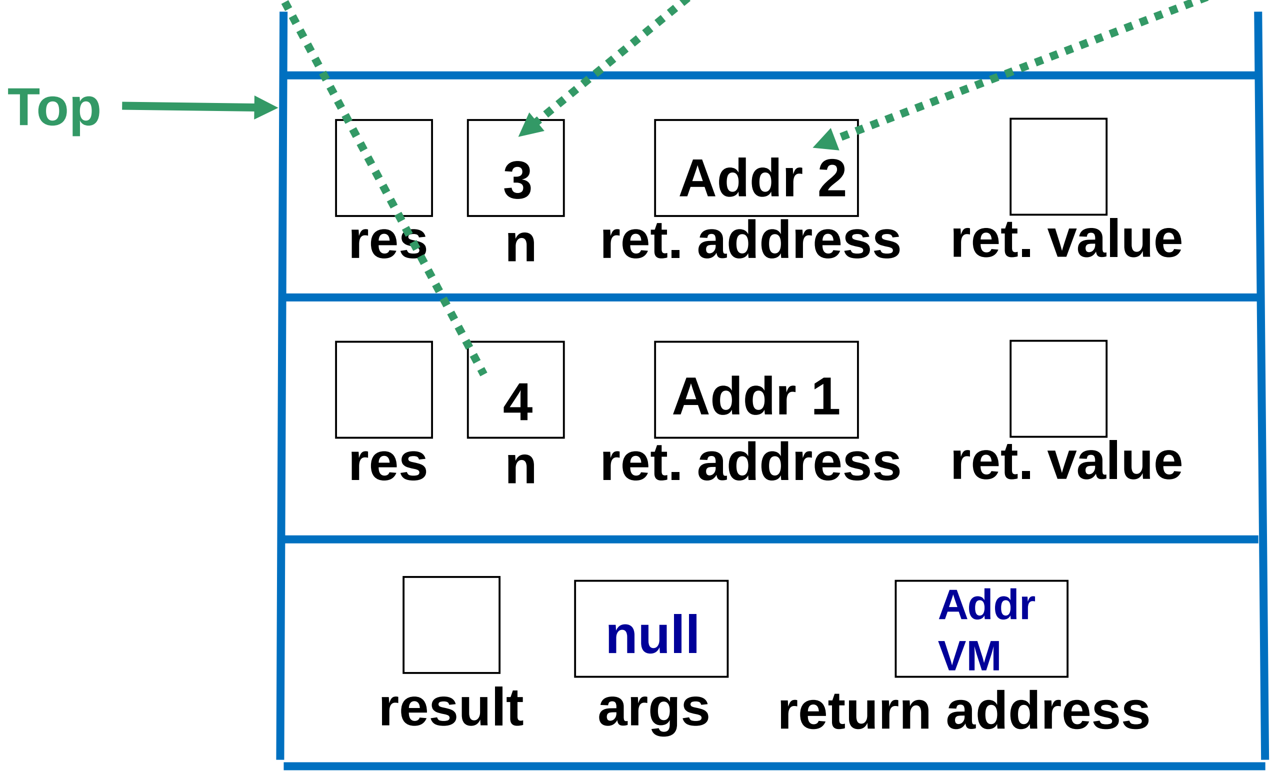


Execution Stack

Since n = 4, the value of the parameter of method sum in

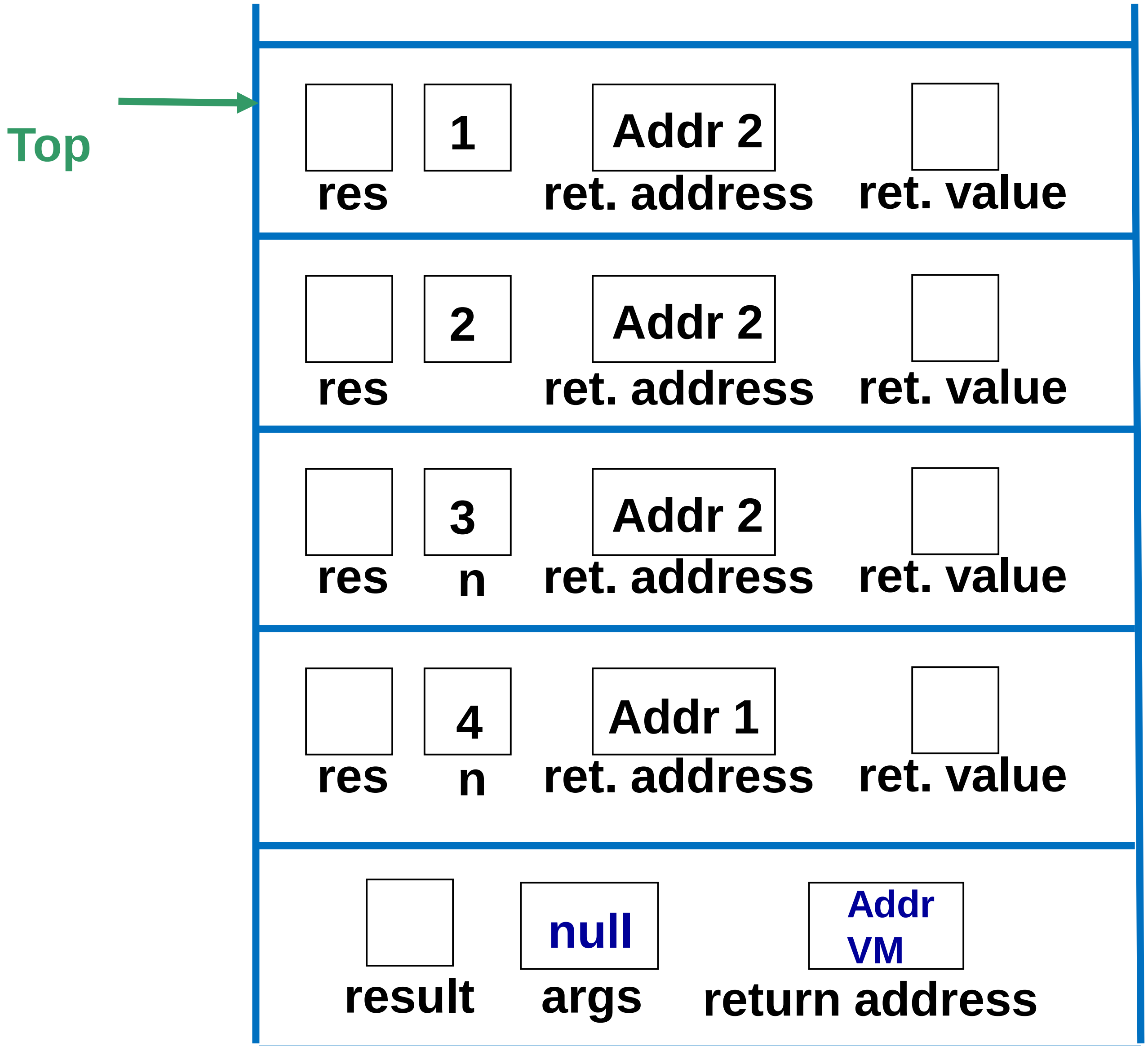
res = n + sum (n-1); // Addr 2

is equal to 3; thus we store the value 3 in n. The return address now is the address of the above statement, which we call Addr 2. This address is stored in the activation record:

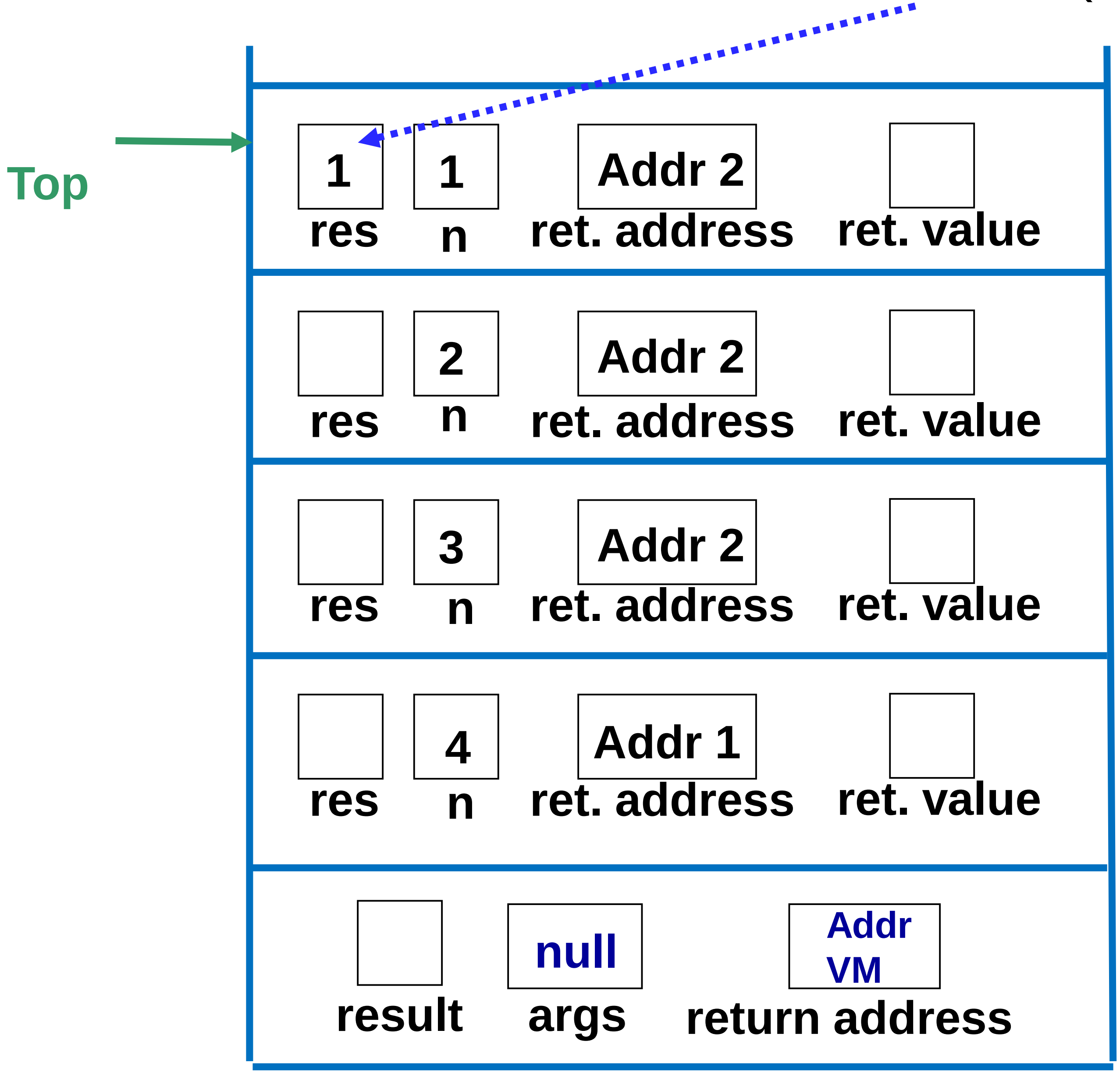


Execution Stack

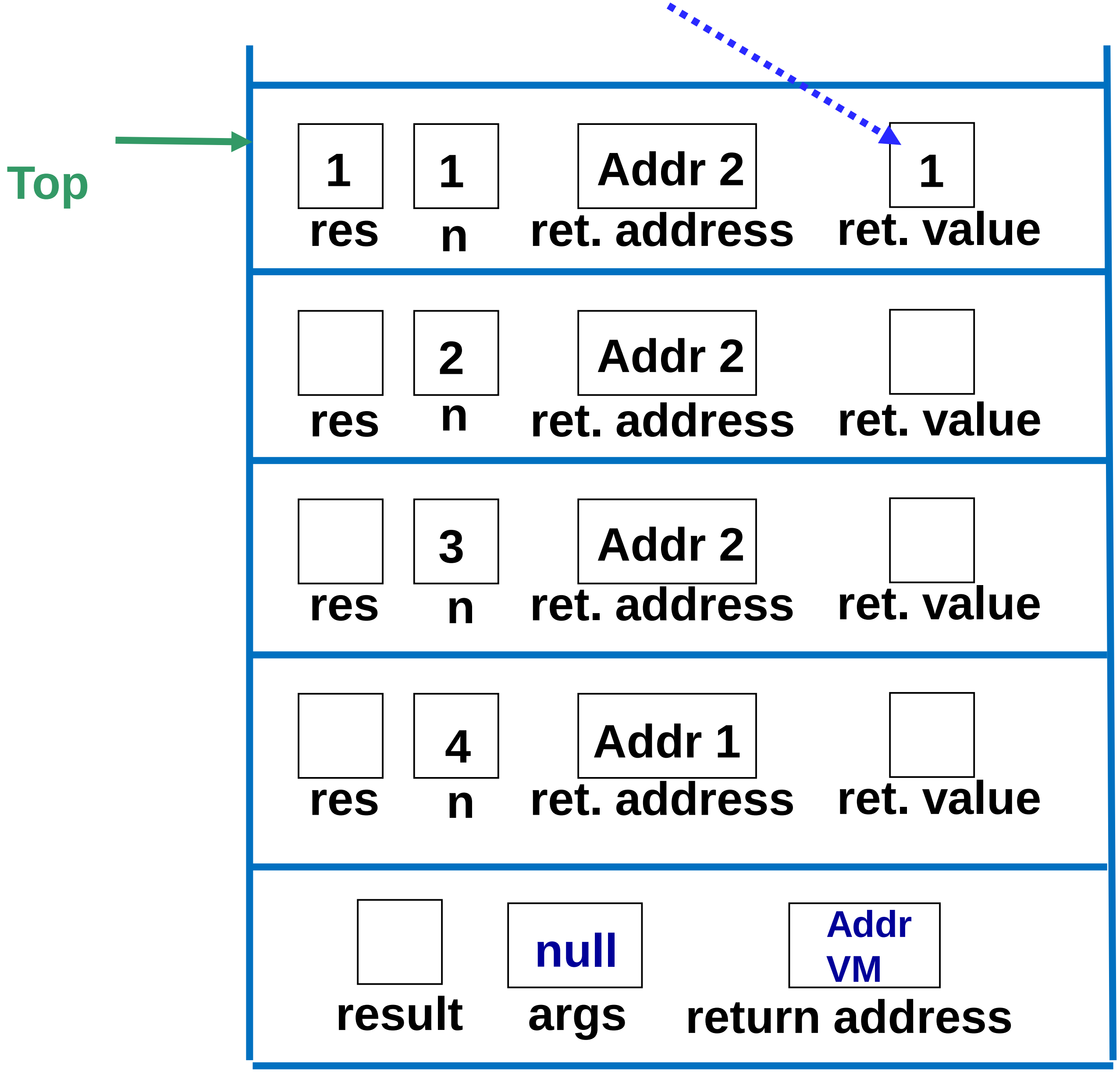
Then two more invocations to method sum with parameters 2 and 1 are made. After the last invocation the execution stack looks like this:



Since in the last invocation to method sum the value of n is 1 then method sum sets the value of res to 1 (base case):



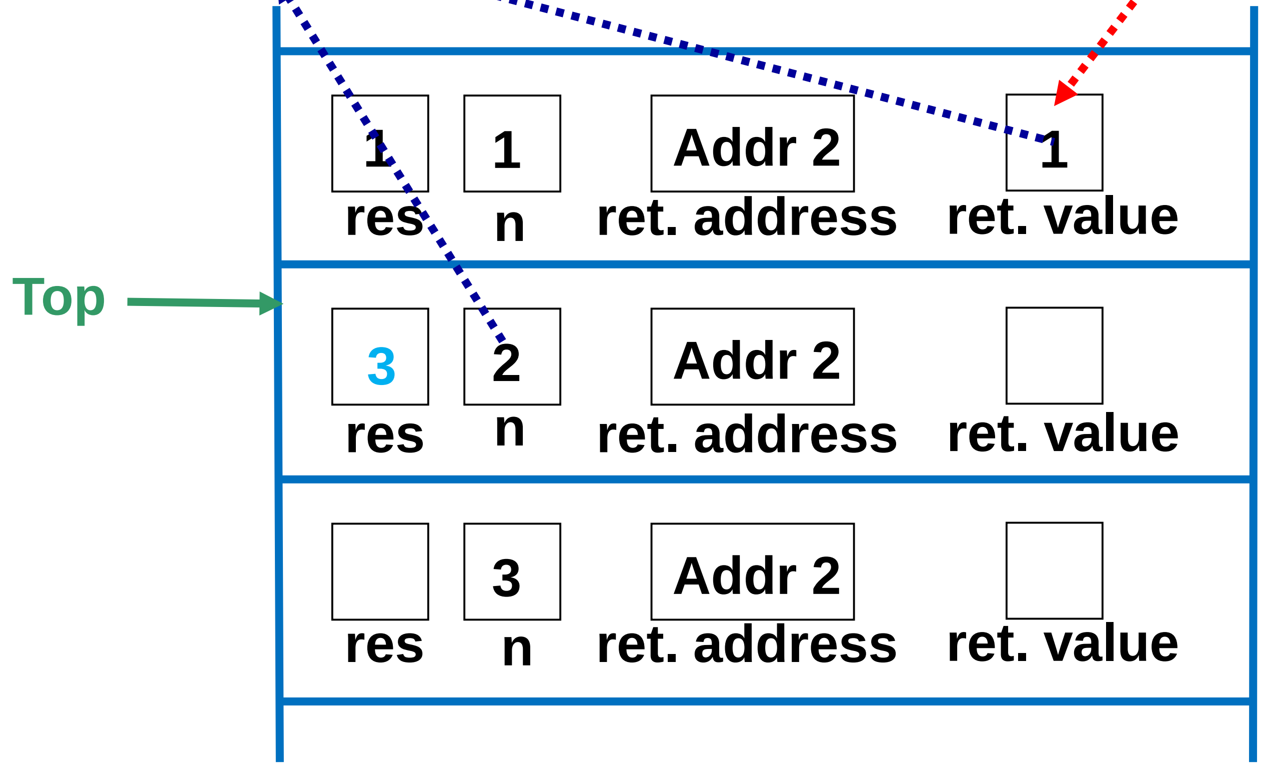
Then the method returns the value 1. The return value is stored in the activation record; the method ends …



and hence the activation record is popped off the execution stack. The return address Addr 2 is recovered and execution continues at the statement in that address:

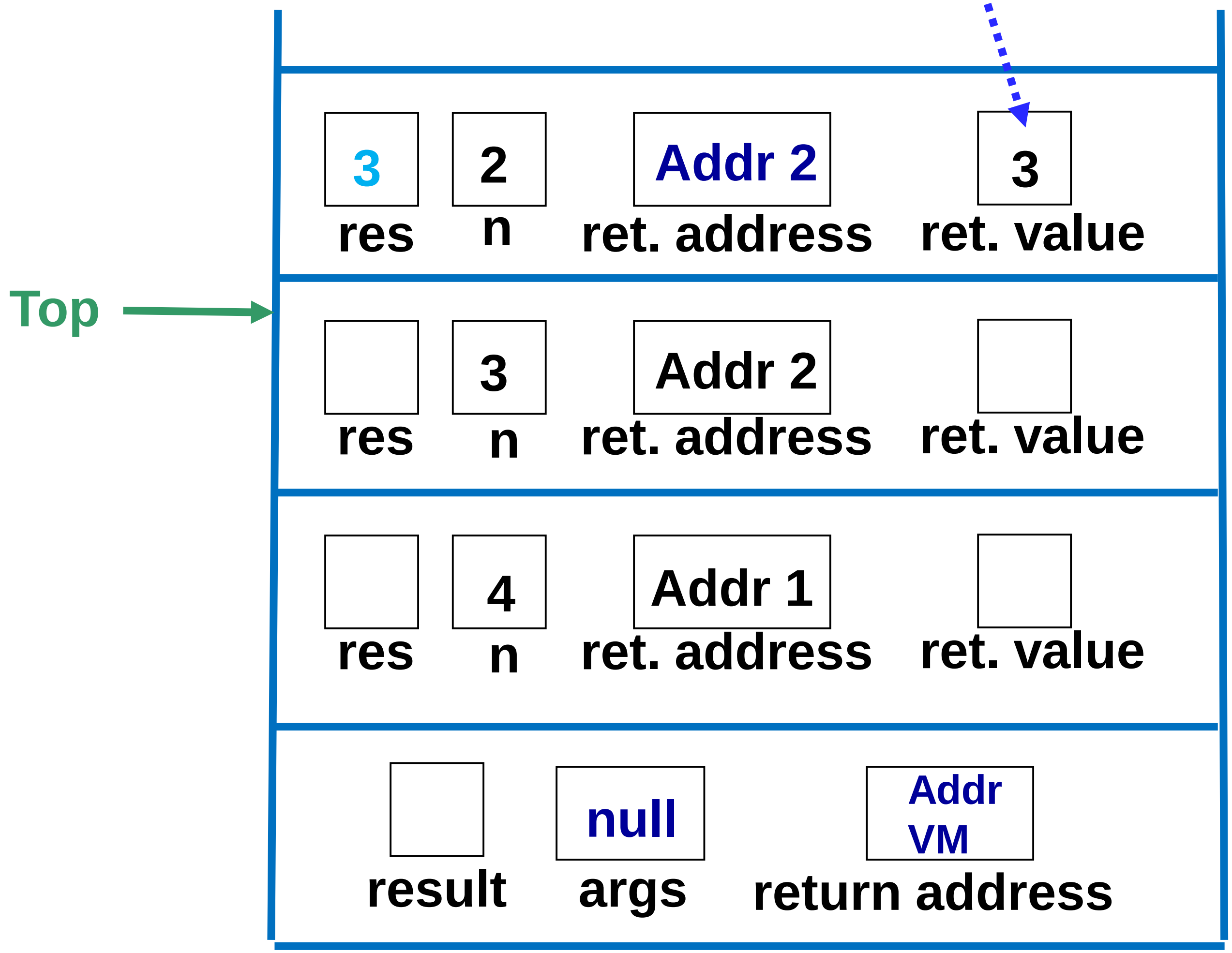
res = n + sum (n-1); // Addr 2

This call just finished and it returned the value 1, hence res takes value n + sum(n-1) = 2 + 1 = 3.



Then the method returns the value 3. The activation record is popped off the stack and execution continues at the statement at address Addr 2, i.e.

res = n + sum (n-1); // Addr 2

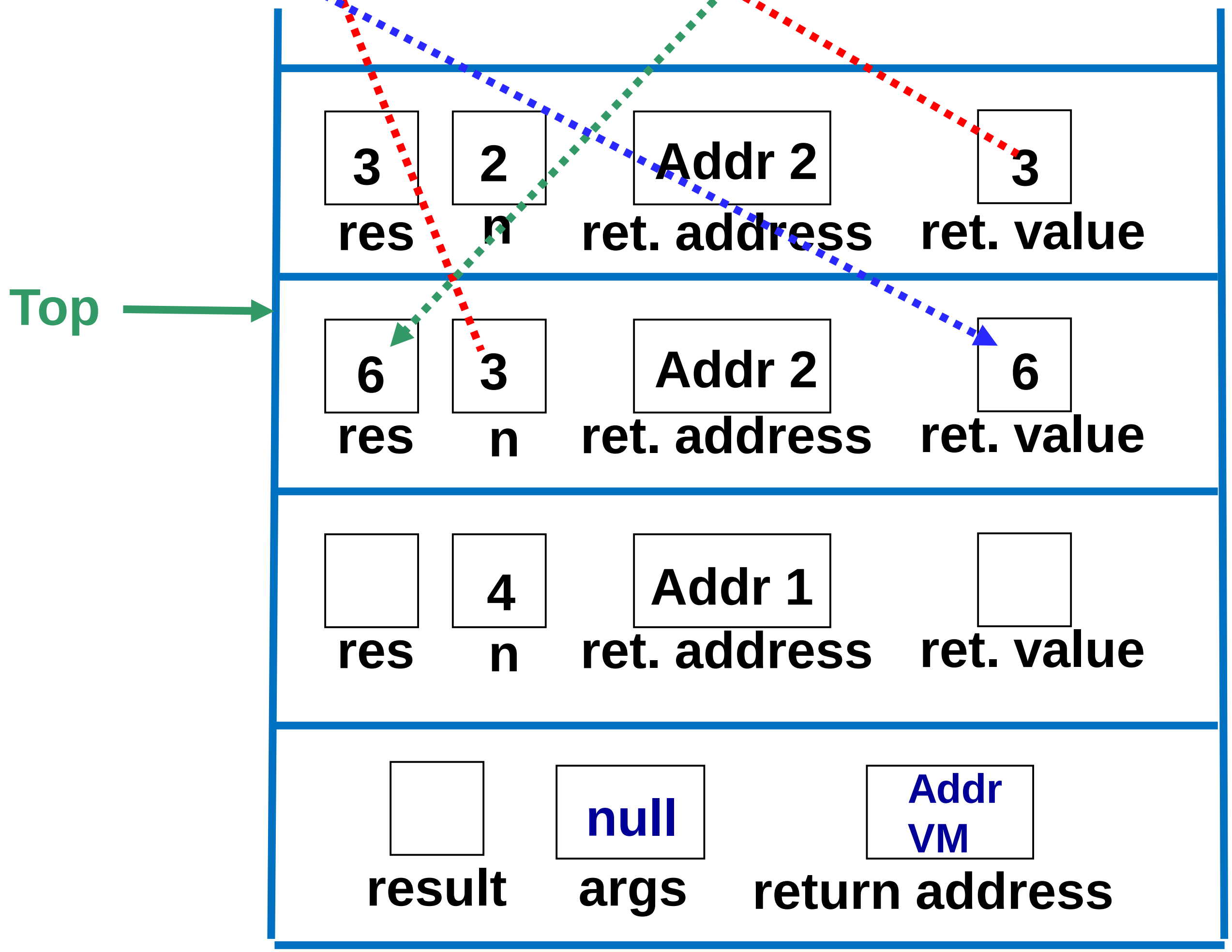


now res takes value

res = n + sum (n-1) = 3 + 3 = 6

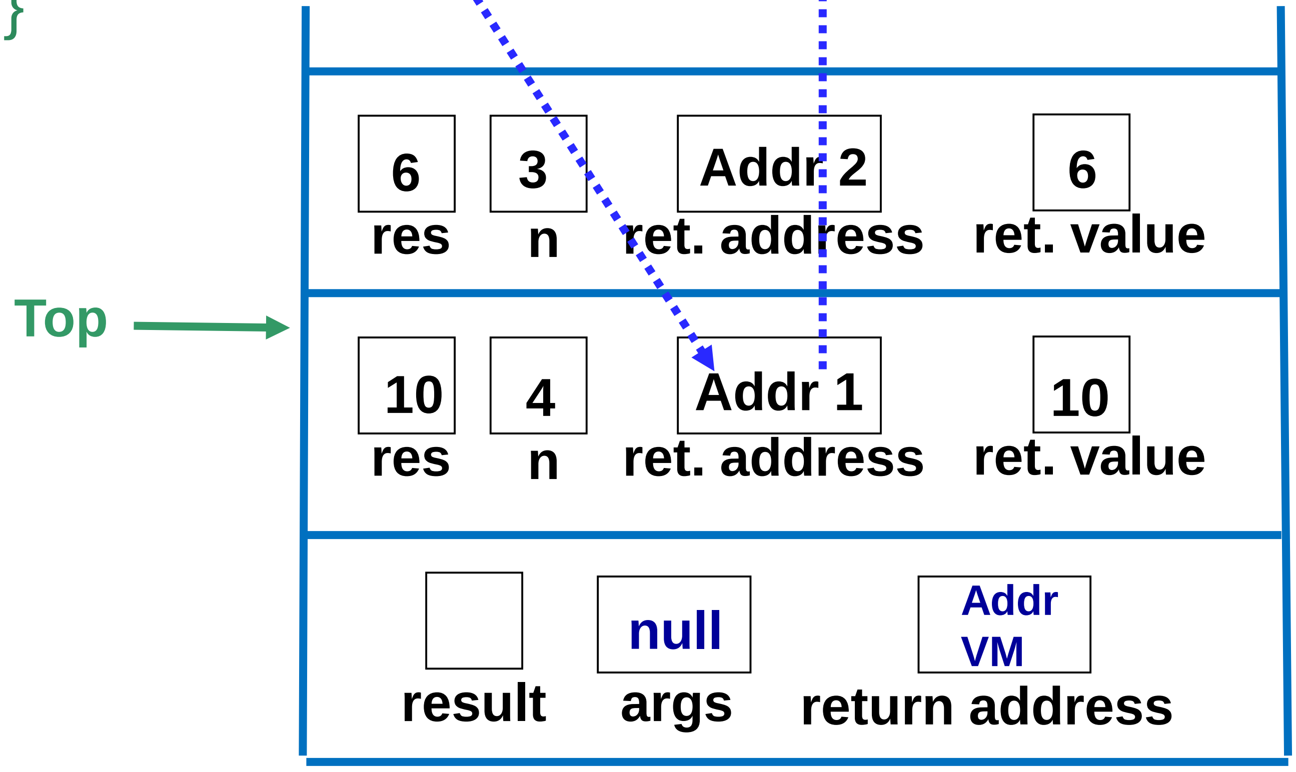
and the value

6 is returned



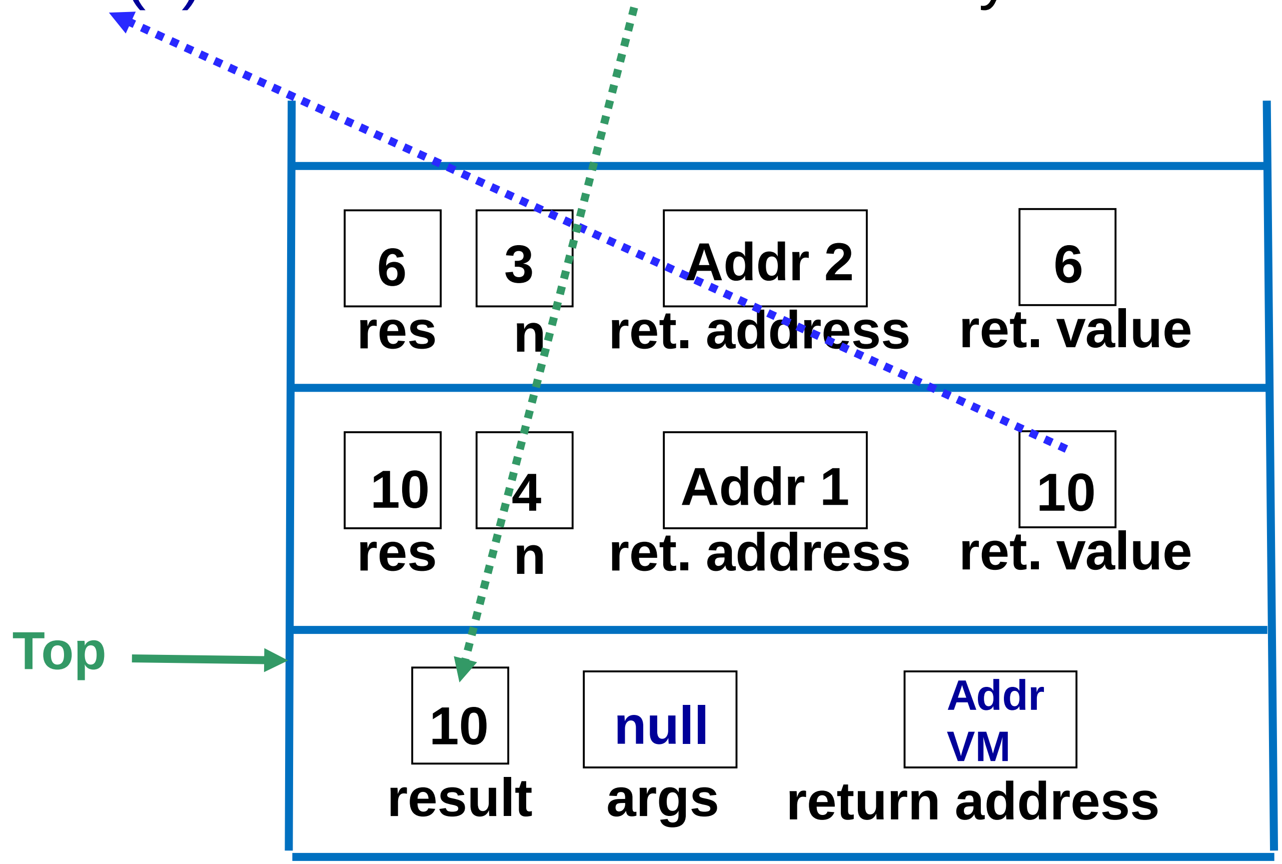
The activation record is popped off the stack and res takes value 6 + 4 = 10. This value is returned to statement in address Addr 1 and the activation record is popped off the stack:

```
public static void main (String[] args) {
    int result = sum(4); // Addr 1
}
```

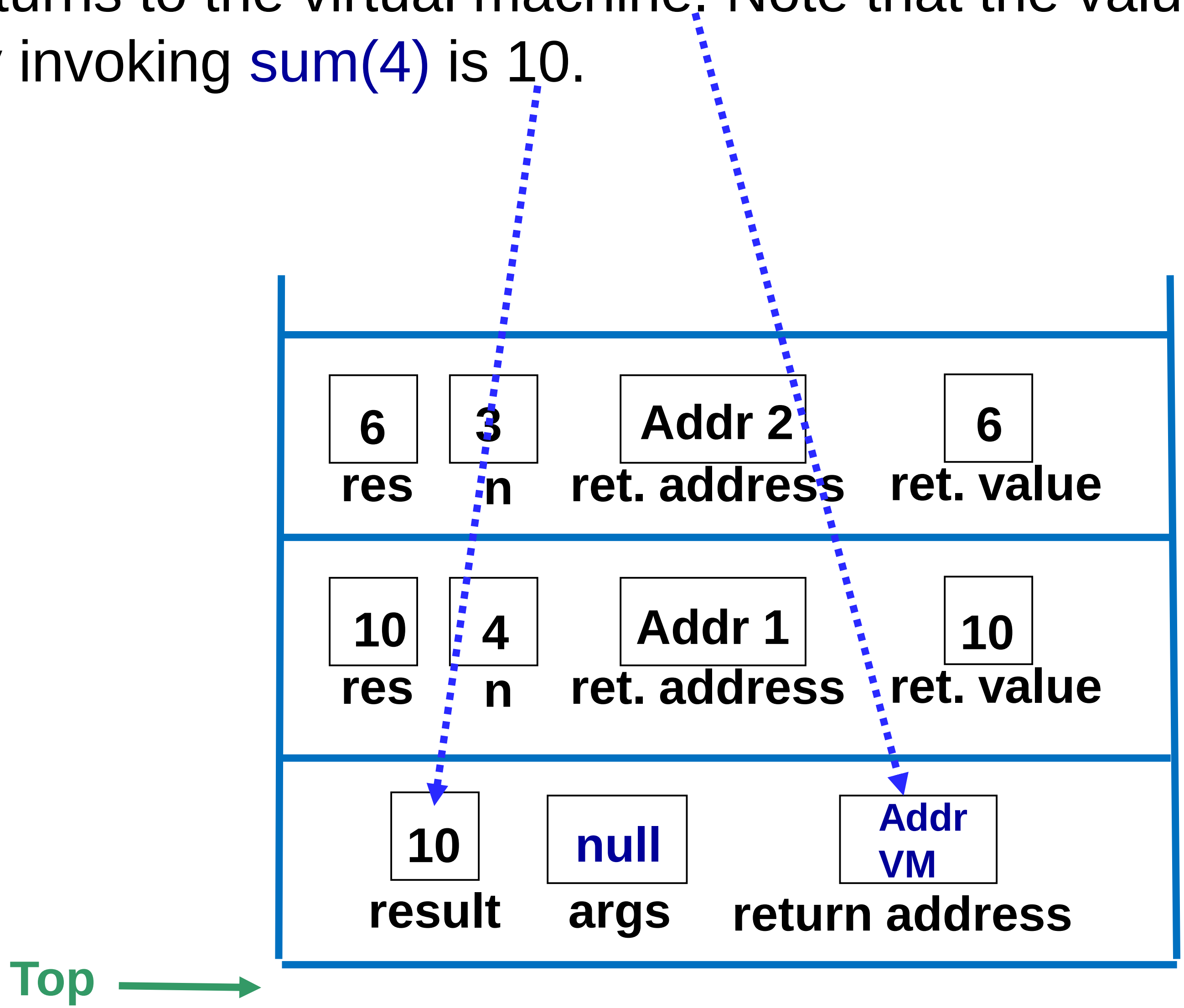


```
public static void main (String[] args) {
      int result = sum(4); // Addr 1
}
```

Note that we are back in method main. The value returned by sum(4) is stored in result and finally method main ends.



The last activation record is popped off the stack and control returns to the virtual machine. Note that the value returned by invoking sum(4) is 10.



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 this sequence

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

- We can define these numbers recursively:

  $$fib(1) = 1$$
  $$fib(2) = 1$$
  $$fib(n) = fib(n - 1) + fib(n - 2) \qquad \textbf{for } n > 2$$

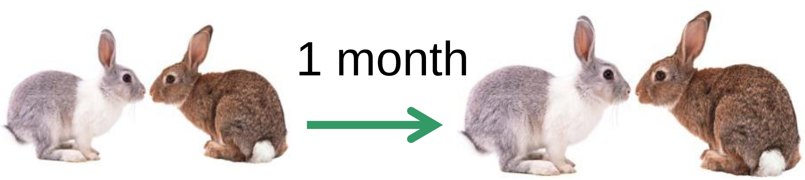
- This sequence is also known as the solution to the ***Multiplying Rabbits Problem*** ☺

Multiplying Rabbits Problem

We have a pair of rabbits…

Multiplying Rabbits Problem

Rabbits can mate after 1 month …

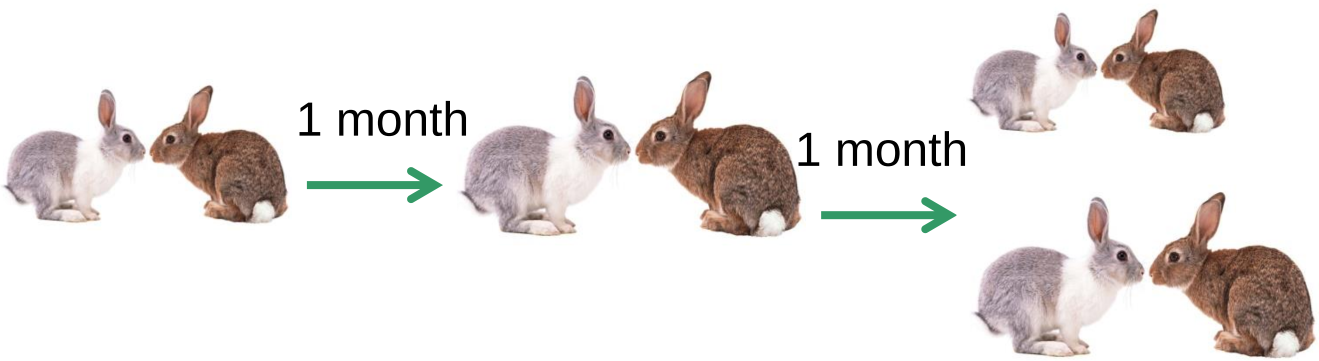


1 **1**

Number of rabbits

Multiplying Rabbits Problem

Rabbits can mate after 1 month and babies are born1 month after mating

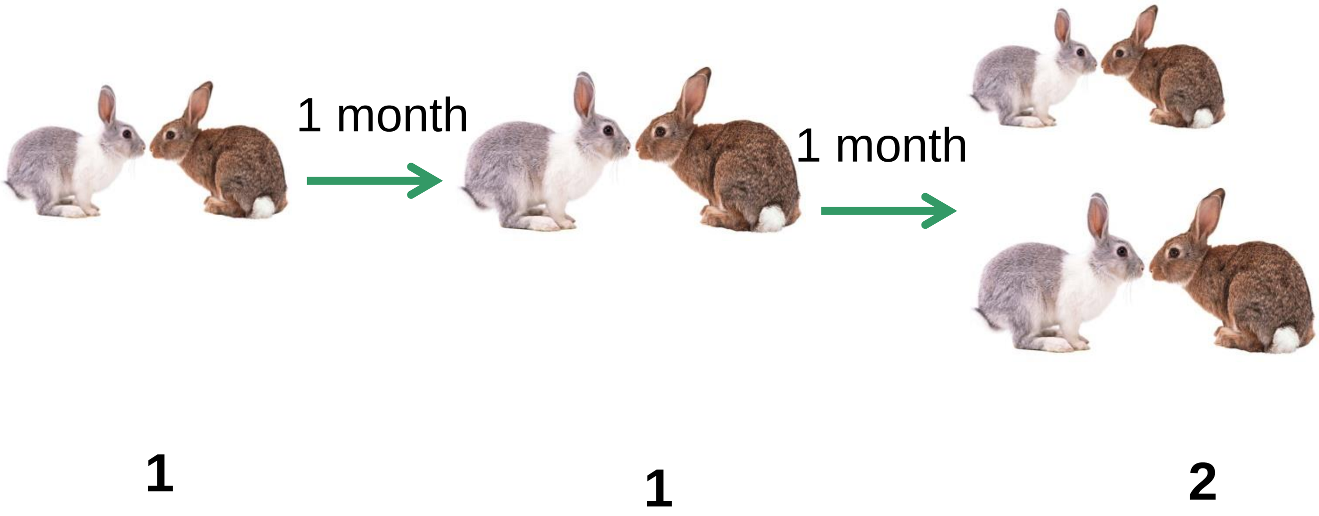


1 **1** **2**

Number of rabbits

Multiplying Rabbits Problem

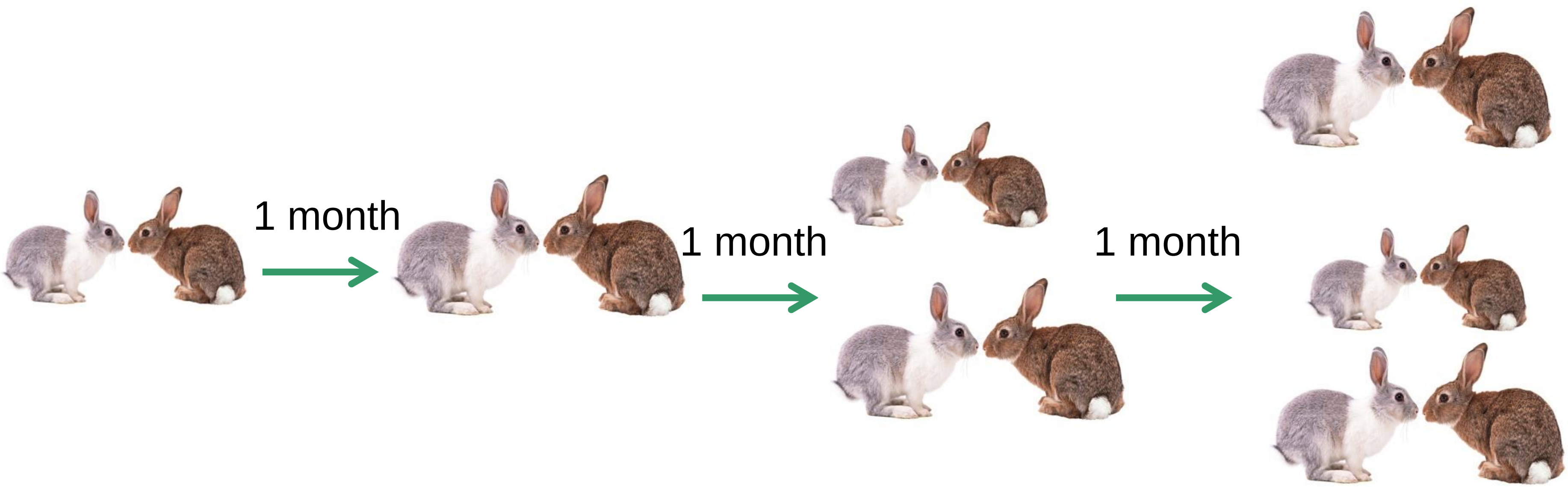
How many rabbits will there be after n months?
Rabbits can mate after 1 month and babies are born 1 month after mating



Number of rabbits

Multiplying Rabbits Problem

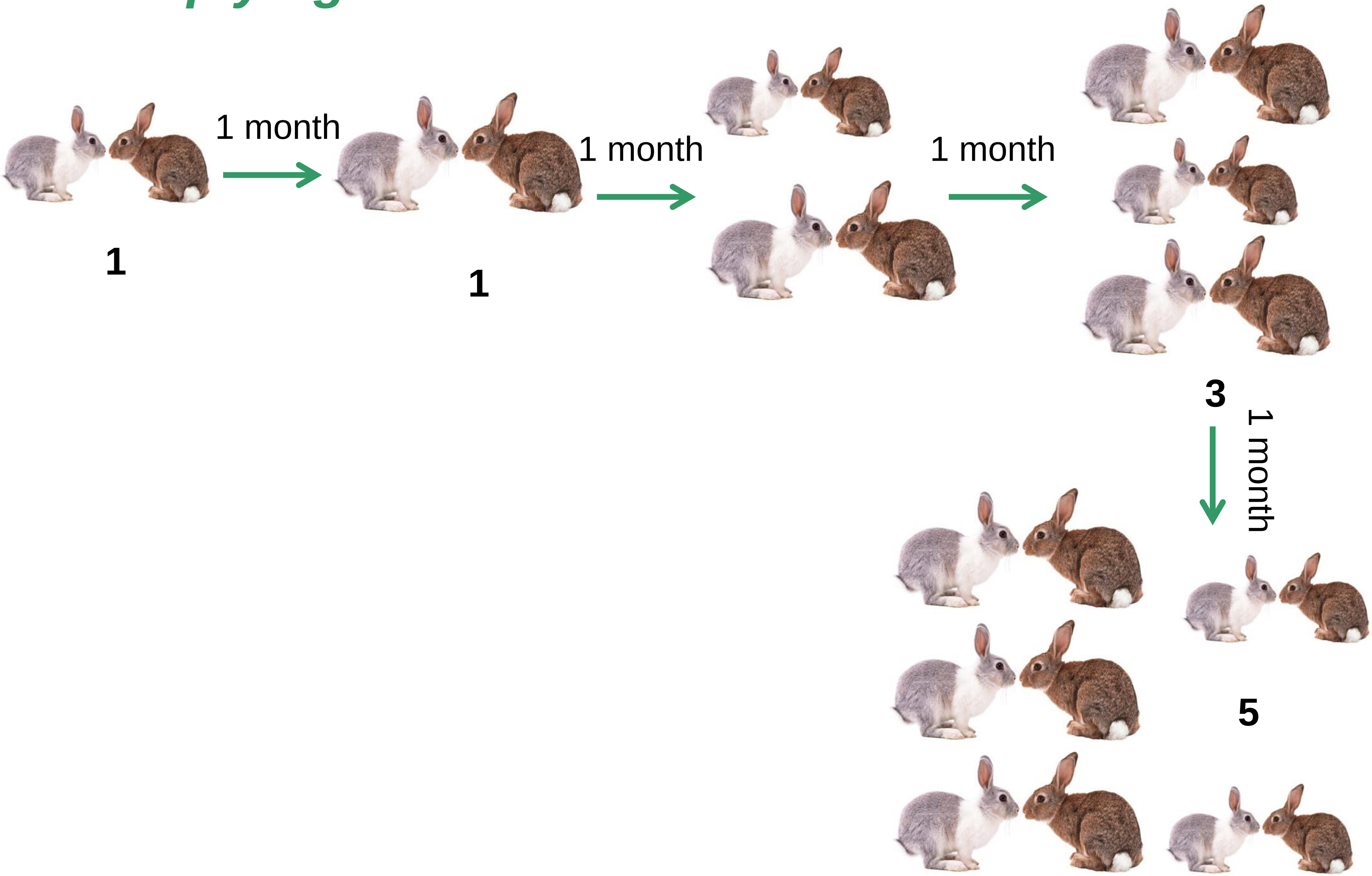
How many rabbits will there be?
Rabbits can mate after 1 month and are born after 1 month



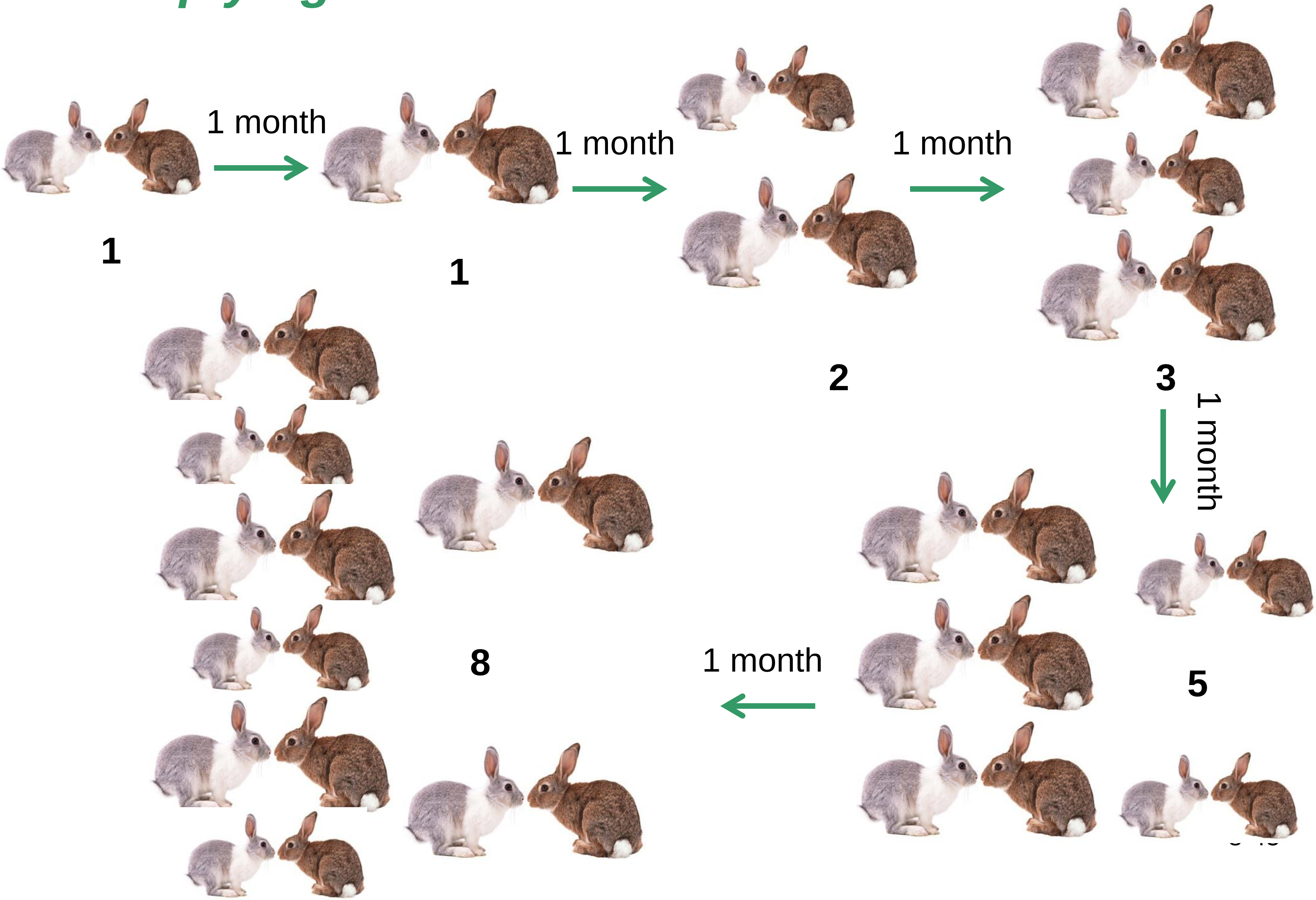
1 **1** **2** **3**

Number of rabbits

Multiplying Rabbits Problem



Multiplying Rabbits Problem



This is the number of rabbits after 1 month, 2 months, 3 months, and so on:

1, 1, 2, 3, 5, 8, …

This sequence is called the *Fibonnaci Sequence*

A Recursive Algorithm for computing 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 Computing 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;
    }
```

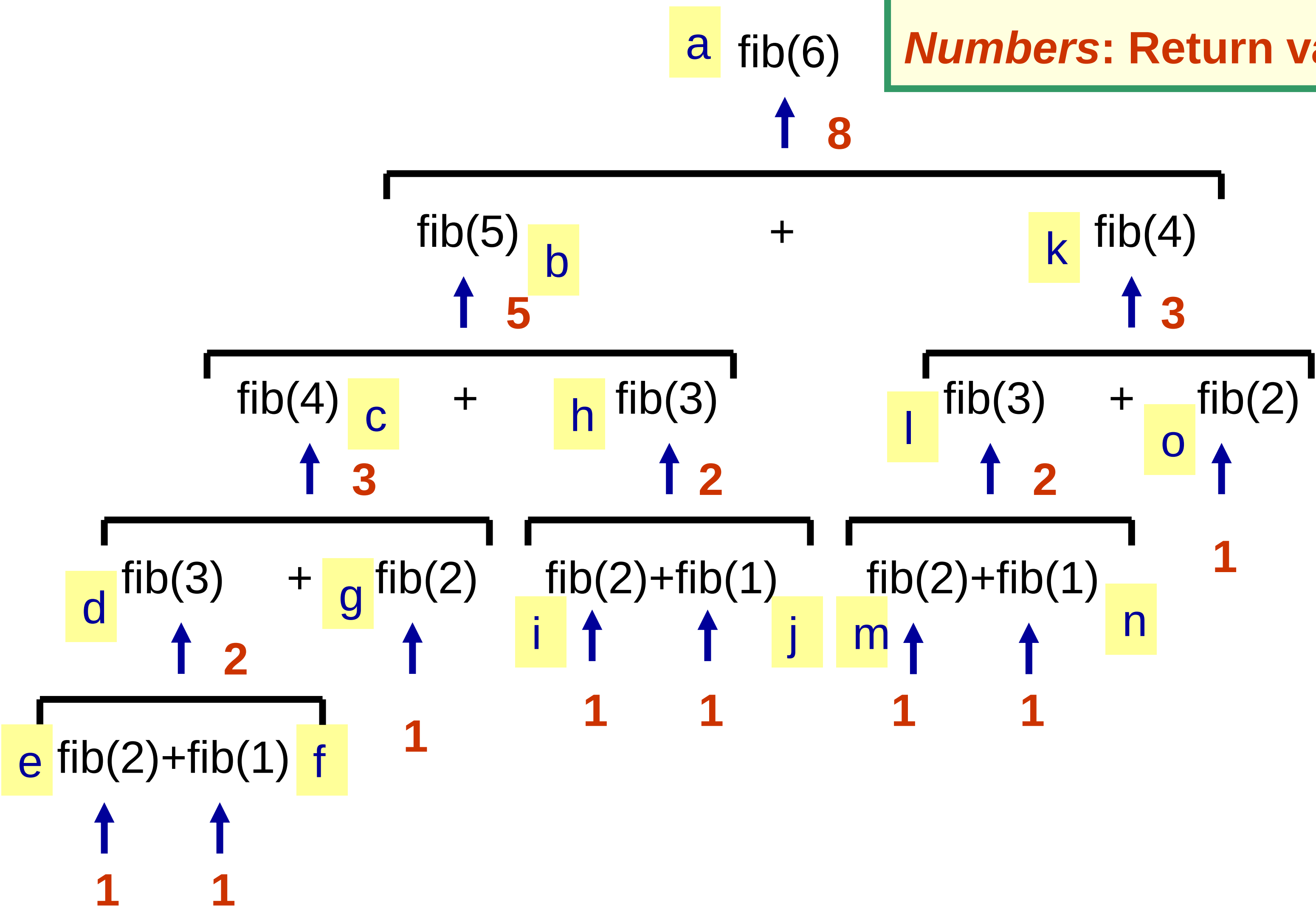
Discussion

- Which solution looks simpler, the recursive or the iterative?
- Which one is (***much***) faster?
  Why?
- ***Note***: recursive and iterative code for computing Fibonacci numbers are posted in the Sample Code page of the course's website - try running them both, and *time them*!

Evaluating fib(6)

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

***Numbers*: Return values**



Application of Recursive Algorithms

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

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

    etc.
- Evaluating algebraic expressions
- 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 consists of
  - Three vertical pegs
  - Several disks that slide onto the pegs
  - The disks are of varying sizes, initially placed on one peg with the largest disk on the bottom and increasingly smaller disks on top

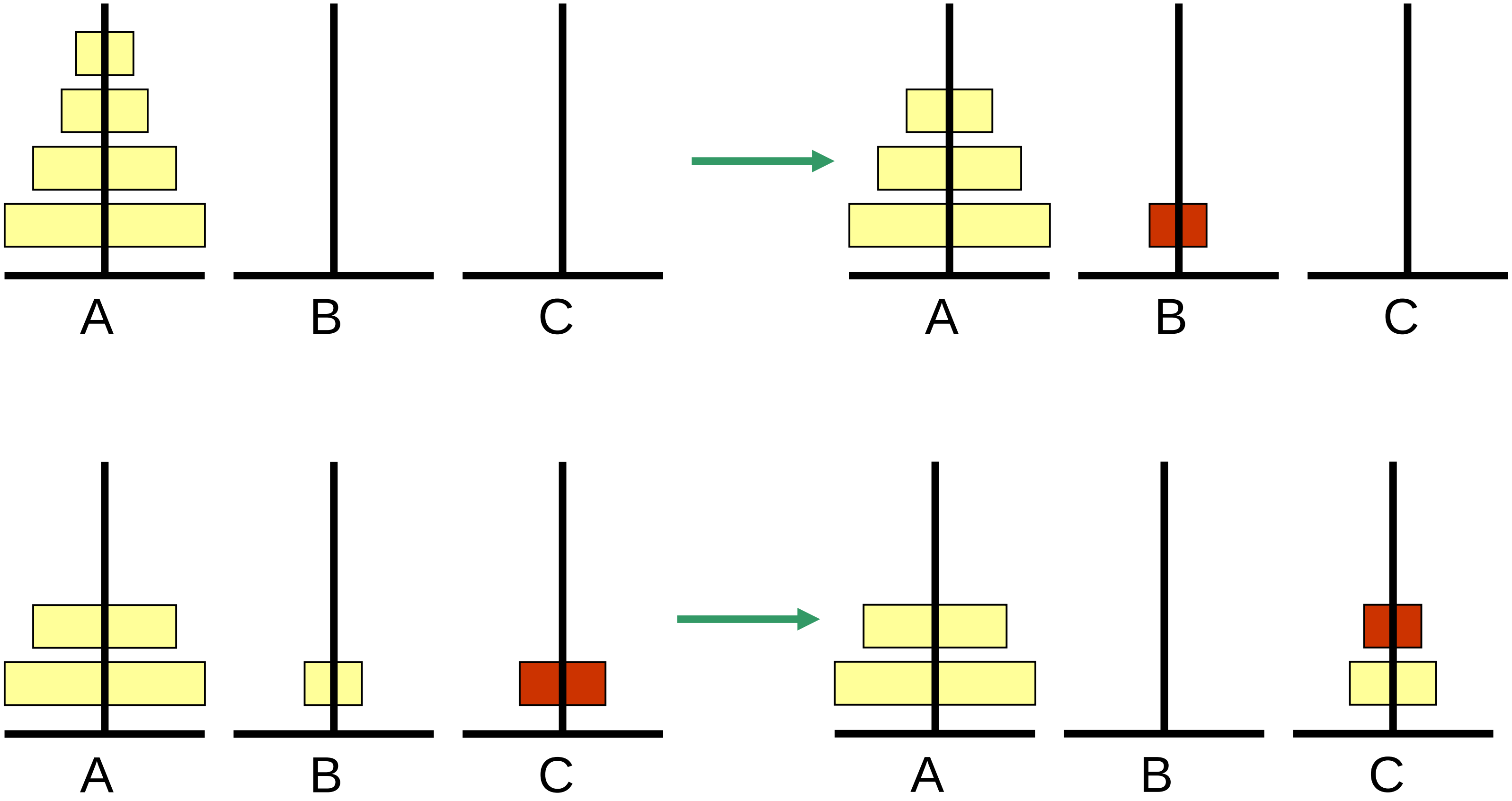
The Towers of Hanoi Puzzle

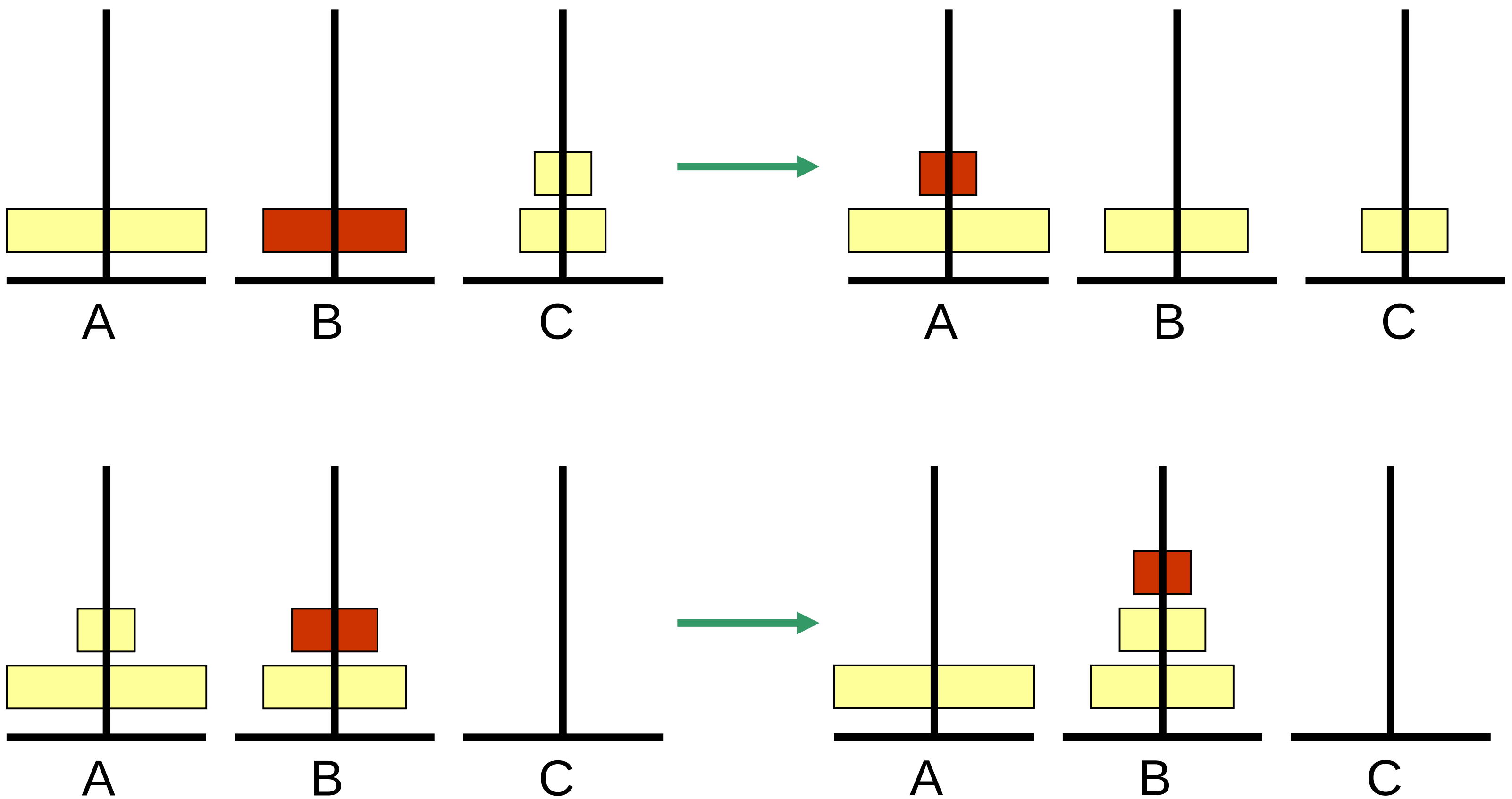
The Towers of Hanoi

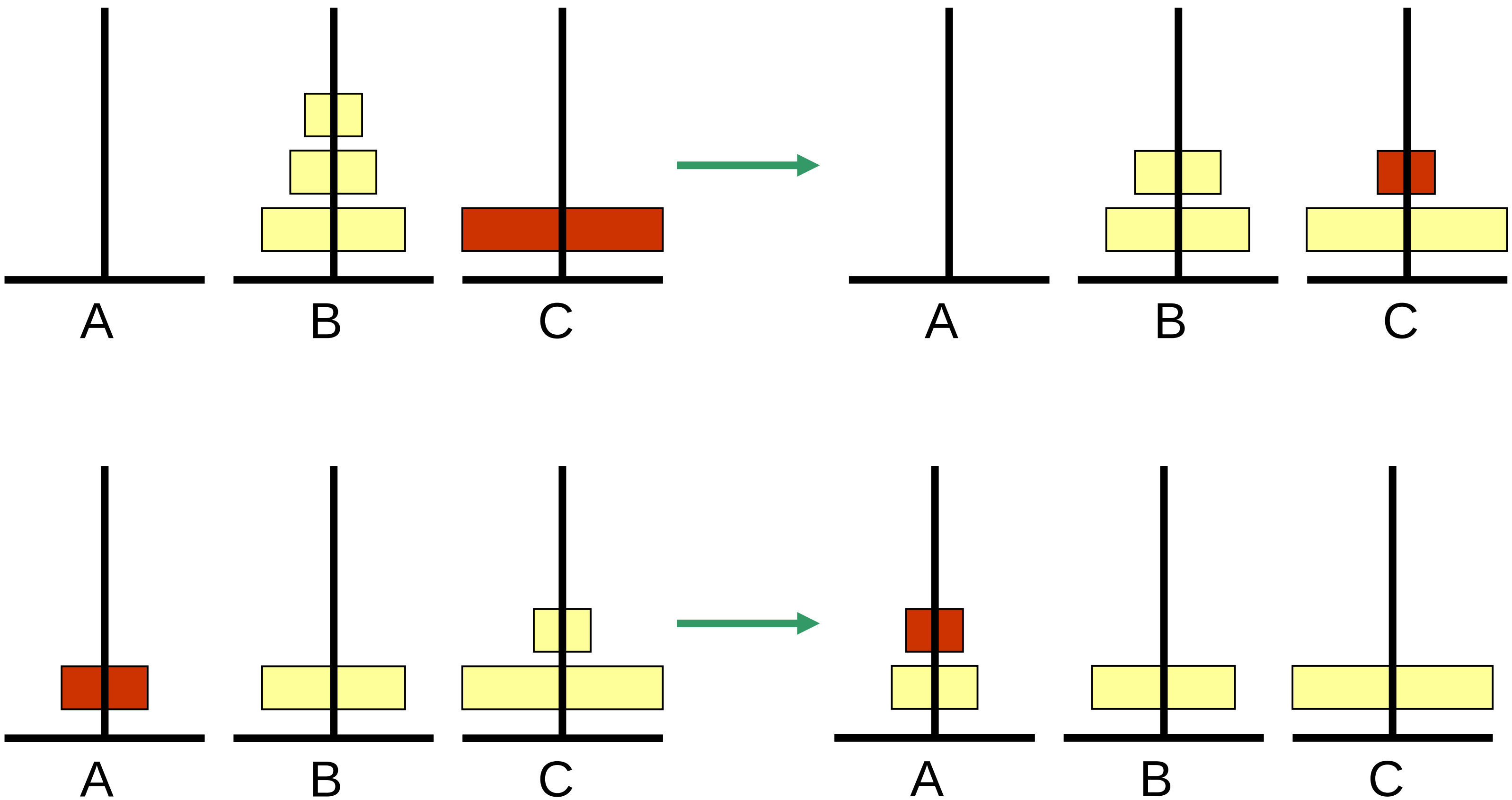
- ***Goal***: move all of the disks from the leftmost peg to the rightmost one following these rules:
  - Only ***one*** disk can be moved at a time
  - A disk ***cannot*** be placed on top of a smaller disk
  - All disks must be on some peg (except for the one in transit)

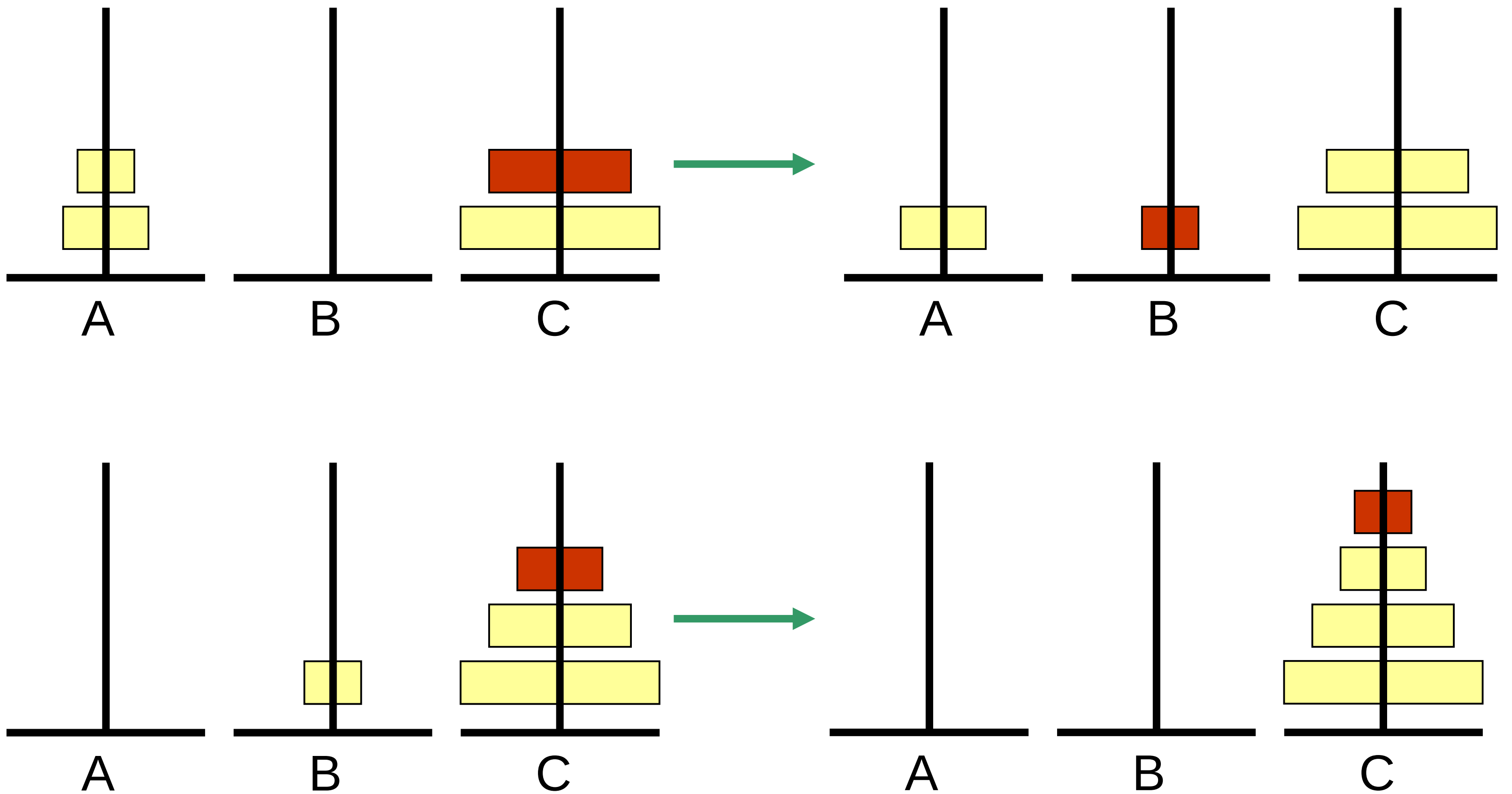
Towers of Hanoi Solution: 4 disks

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)

Algorithm hanoi(iniPeg,destPeg,tmpPeg,n)
In: initial peg, destination peg, third peg, number of disks
Out: Sequence of moves to put all disks in destPeg.
if n = 1 **then** Print (“Move disk from” iniPeg “to” destPeg)
else {
    hanoi(iniPeg,tmpPeg,destPeg,n-1)
    Print (“Move disk from” iniPeg “to” destPeg
    hanoi(tmpPeg,destPeg,tmpPeg,n-1)
}

Java Implementation

```
public void hanoi(int iniPeg, int destPeg, int tmpPeg, int n) {
if (n == 1)
    System.out.println("Move disk from " + iniPeg + " to " destPeg);
else {
    hanoi(iniPeg,tmpPeg,destPeg,n-1)
    System.out.println ("Move disk from " + iniPeg + " to " destPeg);
    hanoi(tmpPeg,destPeg,tmpPeg,n-1)
}
```

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