COMP 250

Lecture 20

recursion

A **recursive method** (or function) is a method that calls itself.

Examples we will see today:

- factorial function
- Fibonacci numbers
- reversing a list
- sorting a list
- tower of Hanoi

We will see many more examples later in the course.

Example 1: Factorial

The factorial of a positive integer is defined as follows:

$0! = 1$

$1! = 1$

$2! = 1 * 2 = 2$

$3! = 1 * 2 * 3 = 6$

…

$n! = 1 * 2 * \ldots * (n-2) * (n-1) * n$

Factorial (iterative)

$$n! = 1 * 2 * 3 * \ldots * (n-1) * n$$

```
public static int factorial (int n) {
    int result = 1;
    for (int i=2; i<=n; i++) {
        result = result * i;
    }
    return result;
}
```

Factorial (Recursive Definition)

$0! = 1$

$1! = 1$

$n! = n * (n-1) * (n-2) * (n-3) * \ldots * 1$

$= n * (n-1)!$

Factorial (Recursive)

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

Connection to Mathematical Induction ?

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

base case

induction step

Correctness

Claim: For all $n \geq 0$, the recursive `factorial(n)` algorithm returns $n!$.

Proof (by mathematical induction):

- Base case: `factorial(0)` returns 1.

- Induction step:
  - Induction hypothesis: `factorial(k)` returns $k!$ where $k \geq 0$
  - We want to prove it follows that `factorial(k+1)` returns $(k+1)!$

  - `factorial(k+1)` returns `factorial(k)` $* (k+1)$
    $= k! * (k+1)$, by induction hypothesis
    $= (k+1)!$

Example 2: Fibonacci

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

$F(0) = 0$
$F(1) = 1$
$F(n+2) = F(n+1) + F(n)$, for $n \geq 0$.

definition

Fibonacci (iterative)

```
public static int fibonacci(int n) {
      if(n==0 || n==1) {
            return n;
      }
      fib0 = 0;
      fib1 = 1;
      for (int i=2; i<=n; i++) {
            fib2 = fib0 + fib1;
            fib0 = fib1;
            fib1 = fib2;
      }
      return fib2;
}
```

Fibonacci (recursive)

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

This is simpler to express than the iterative version.

Correctness

Claim: the recursive Fibonacci algorithm is correct.

Proof:

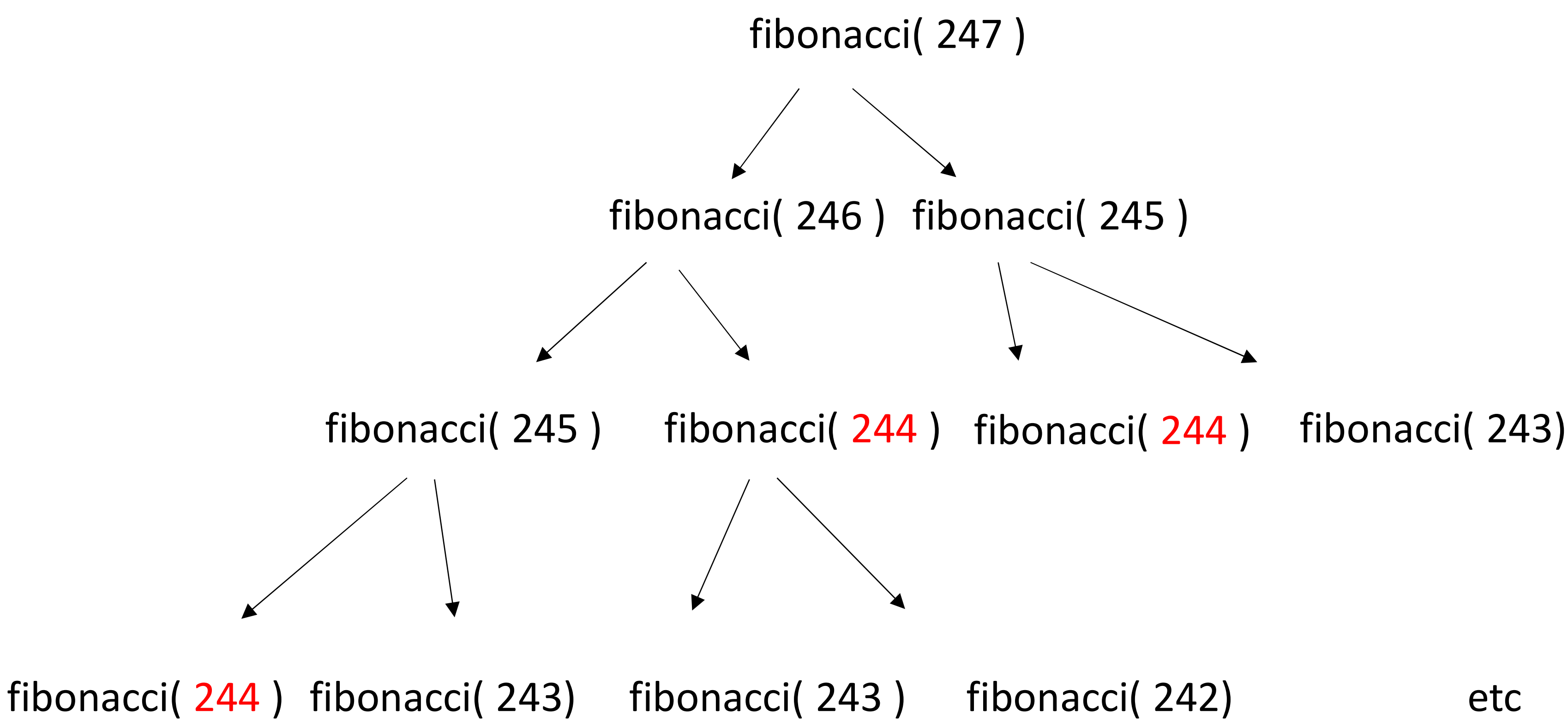
Base case(s): verify (trivial)

Induction step: (also trivial)

Let k > 1. Induction hypothesis is that fibonacci(k-1) returns F(k-1) and fibonacci(k) returns F(k).

Then fibonacci(k+1) returns F(k-1)+ F(k), which is indeed F(k+1).

Unfortunately, the recursive Fibonacci algorithm is inefficient.
It computes the same quantity many times, for example:



In COMP 251, you will learn a general technique called *dynamic programming* that avoid this inefficiency.

Example 3: Reversing a list

input (a b c d e f g h)

output (h g f e d c b a)

How to do this recursively?

Example 3: Reversing a list

input (a b c d e f g h)

output (h g f e d c b a)

How to do this recursively?

a (b c d e f g h)

(h g f e d c b) a

Example 3: Reversing a list (recursive)

```
public static void reverse(List list) {
      if(list.size()==1) {
            return;
      }
      firstElement = list.remove(0);
      reverse(list);  // this list has n-1 elements
      list.add(firstElement);
           // appends at the end of the list
}
```

Note that Java's `list.add( E )` returns a Boolean, which we ignore.

Example 4: Sorting a list (recursive)

```
public static void sort(List list) {
      if  (list.size() == 1) {
            return;
      }

```

Can we apply a similar idea ?

```
}
```

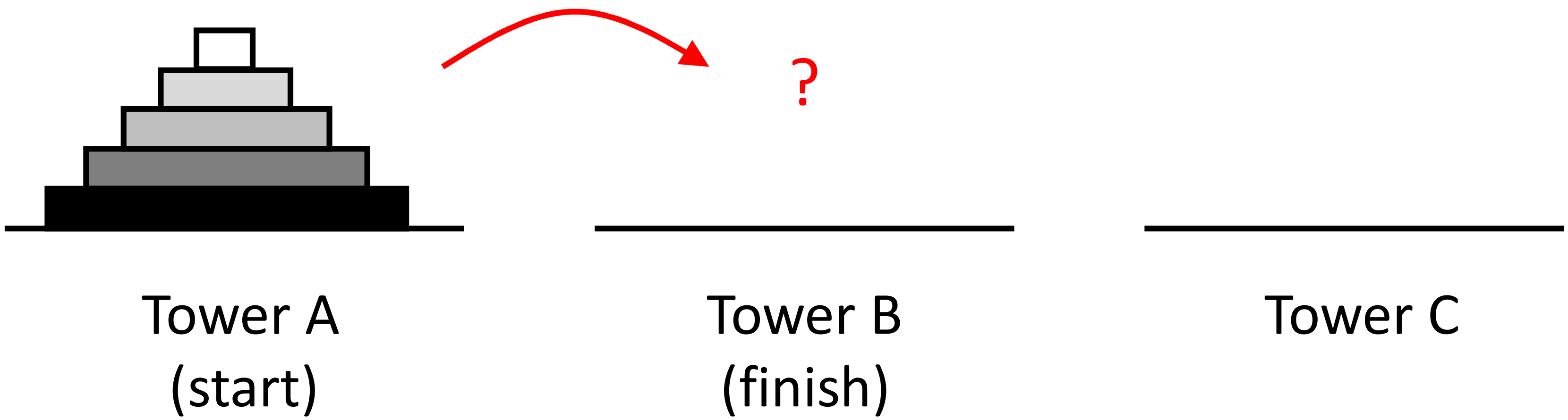
Example 4: Sorting a list (recursive)

```
public static void sort(List list) {
      if  (list.size() == 1) {
            return;
      }
      minElement = removeMinElement(list);
      sort(list); // now the list has n-1 elements
      list.add(0, minElement); // insert at front
}
```

Note that Java's `list.add(int, E )` is void. It changes the list.

You could do a similar solution by removing the max element and adding to end.

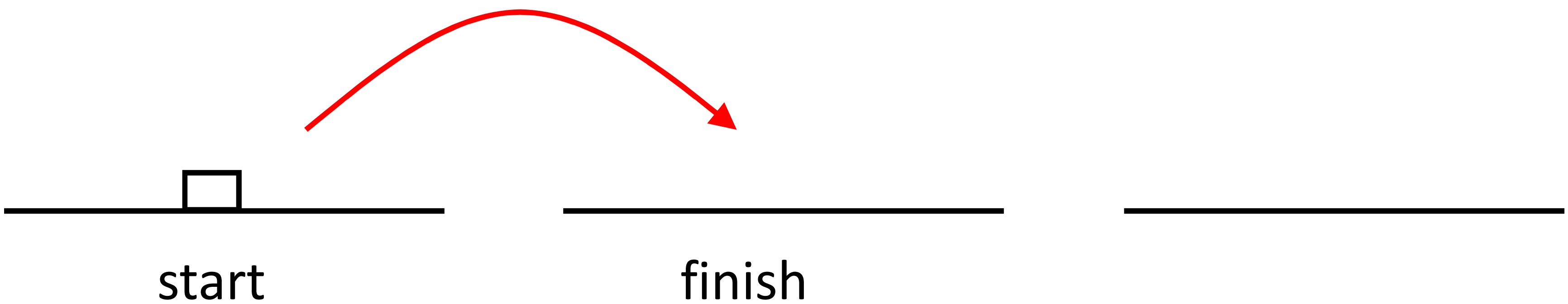
Example 5: Tower of Hanoi



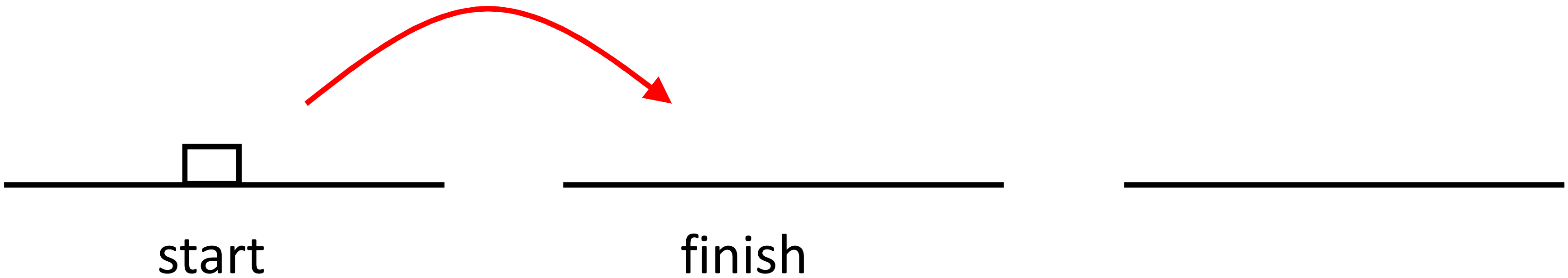
Problem: Move n disks from start tower to finish tower such that:

- move one disk at a time (pop and push)
- you can push a smaller disk on top of bigger disk (but you can't push a bigger disk onto a smaller disk)

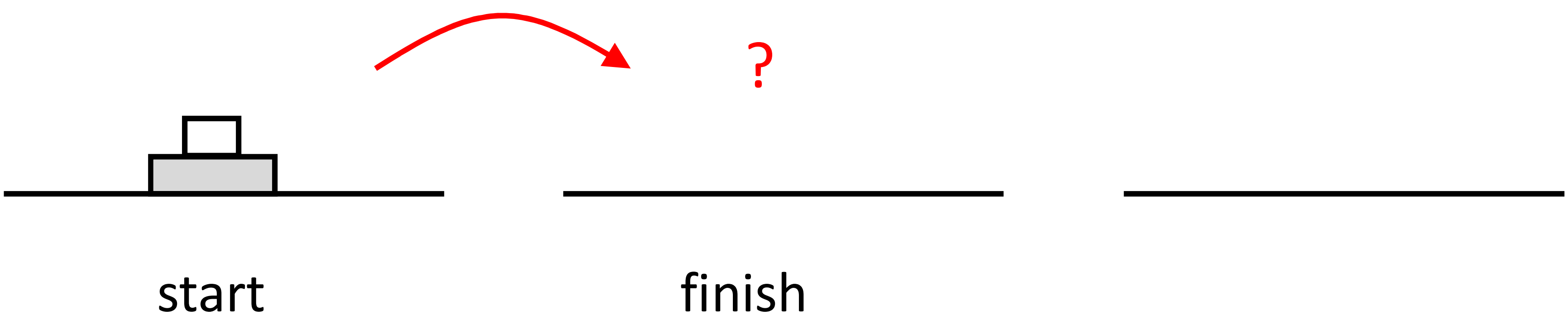
Example: $n = 1$



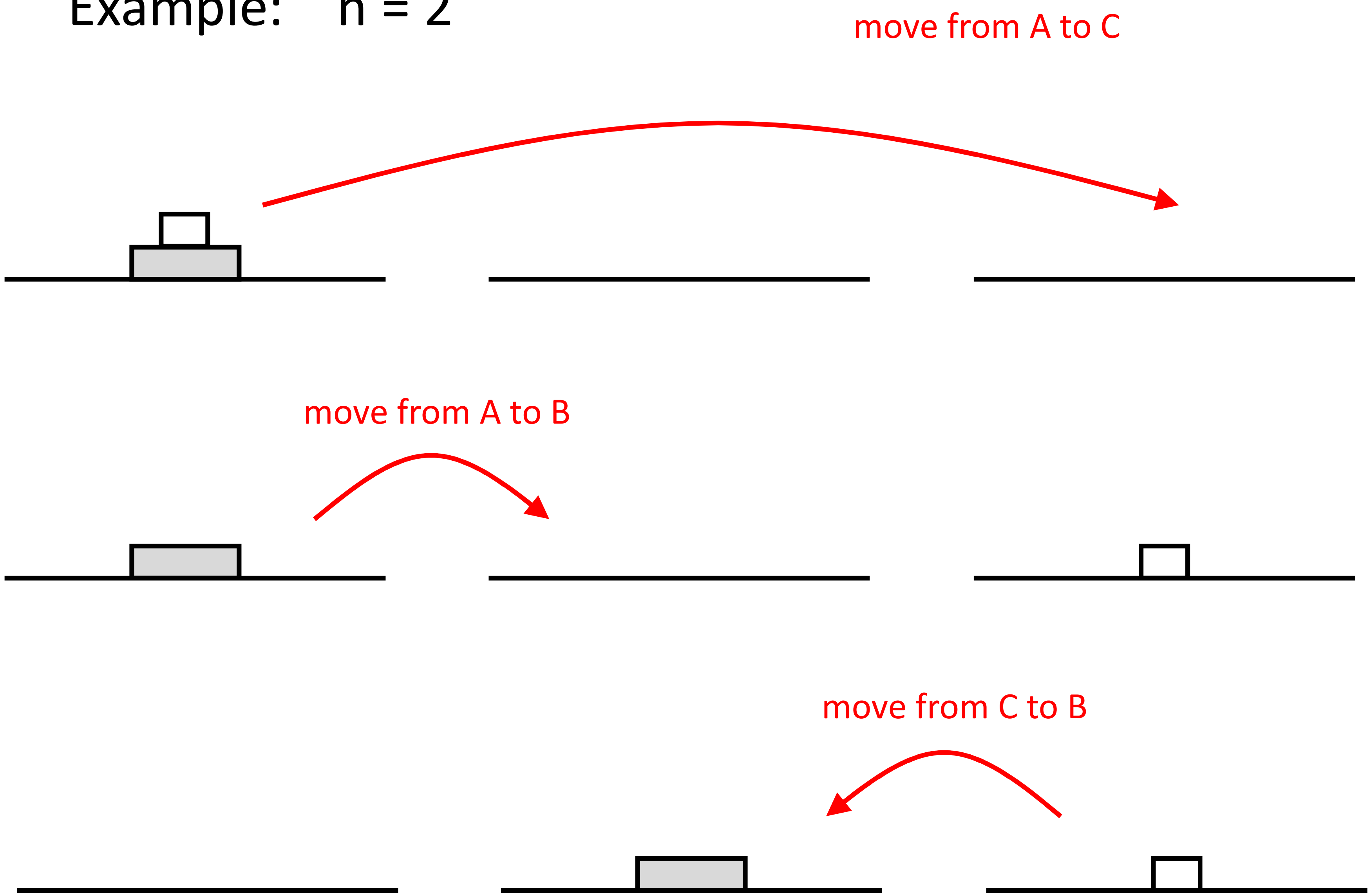
Example: n = 1



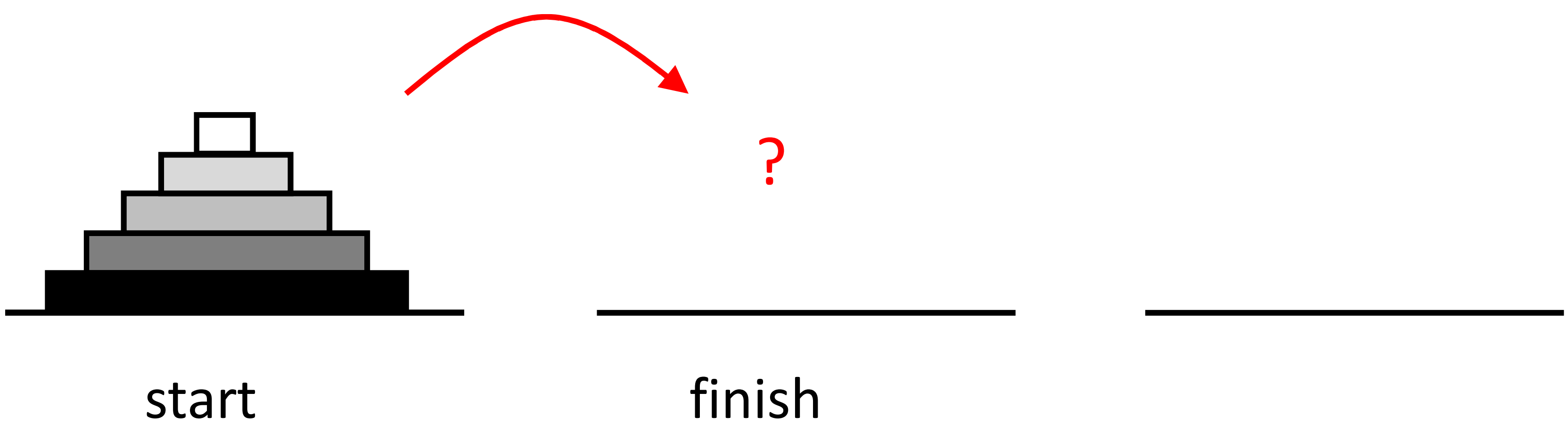
Example: n = 2



Example: n = 2

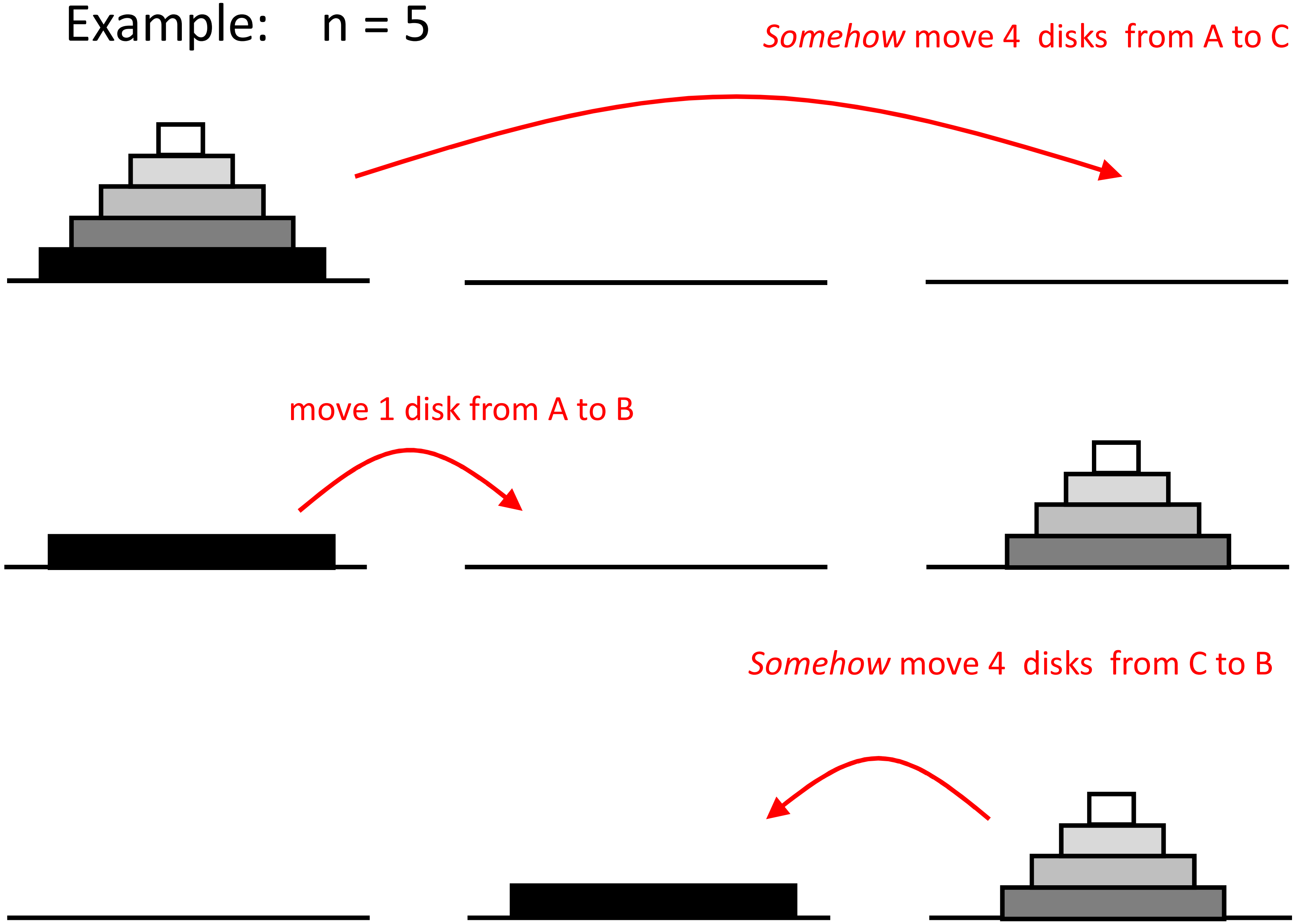


Q: How to move 5 disks from tower 1 to 2 ?



Hint: Think recursively.

Example: n = 5

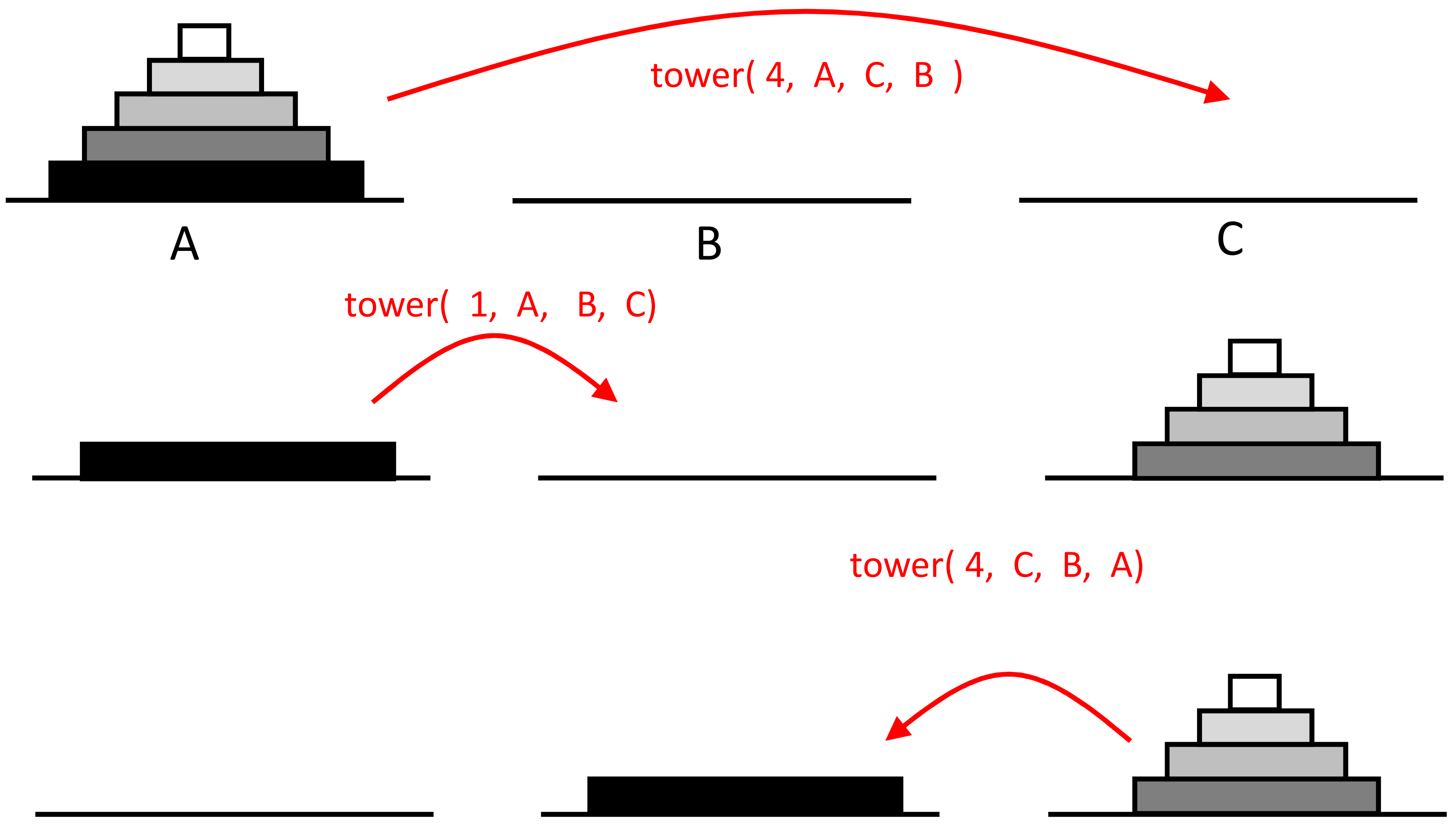

Somehow move 4 disks from A to C
move 1 disk from A to B
Somehow move 4 disks from C to B

```
tower(n, start, finish, other) {

      if (n==1) {
            move from start to finish.
      } else {
            tower(n-1, start, other, finish)
            tower(1,   start, finish, other)
            tower(n-1, other, finish, start)
      }
}
```

For example, **`tower(5,A,B,C)`**

Example: n = 5 tower(5, A, B, C)



Correctness

Claim: the tower( ) algorithm is correct, namely it moves the blocks from start to finish without breaking the two rules (one at a time, and can't put bigger one onto smaller one).

<u>Proof: (**sketch**)</u>

It doesn't matter, as long as they are different.

Base case: tower(1, *, *, *) is correct.

Induction step:

induction hypothesis

for any k >= 1, if tower(k, *, *, *) is correct
then tower(k + 1, *, *, *) is correct.
(verify by inspection of algorithm)

How many moves ?

tower(1, start, finish, other)

↓

move start
to finish

Answer: 1

How many moves ?

tower(2, start, finish, other)

tower(1, start, other, finish)

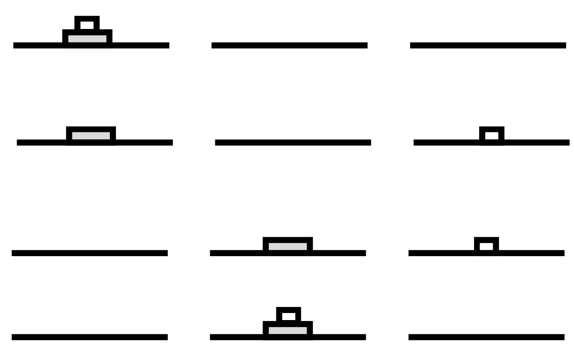
move from A to B

tower(1, other, finish, start)

move from A to C

move from C to B

Answer: 1 + 2

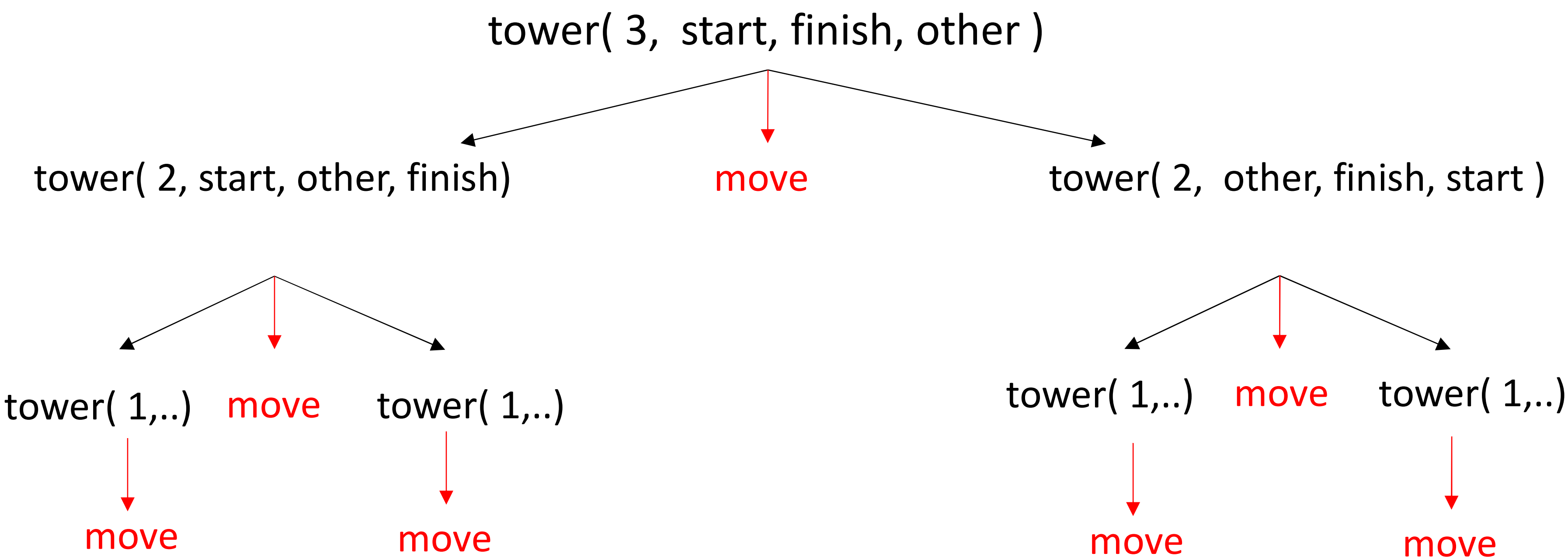


move from A to C

move from A to B

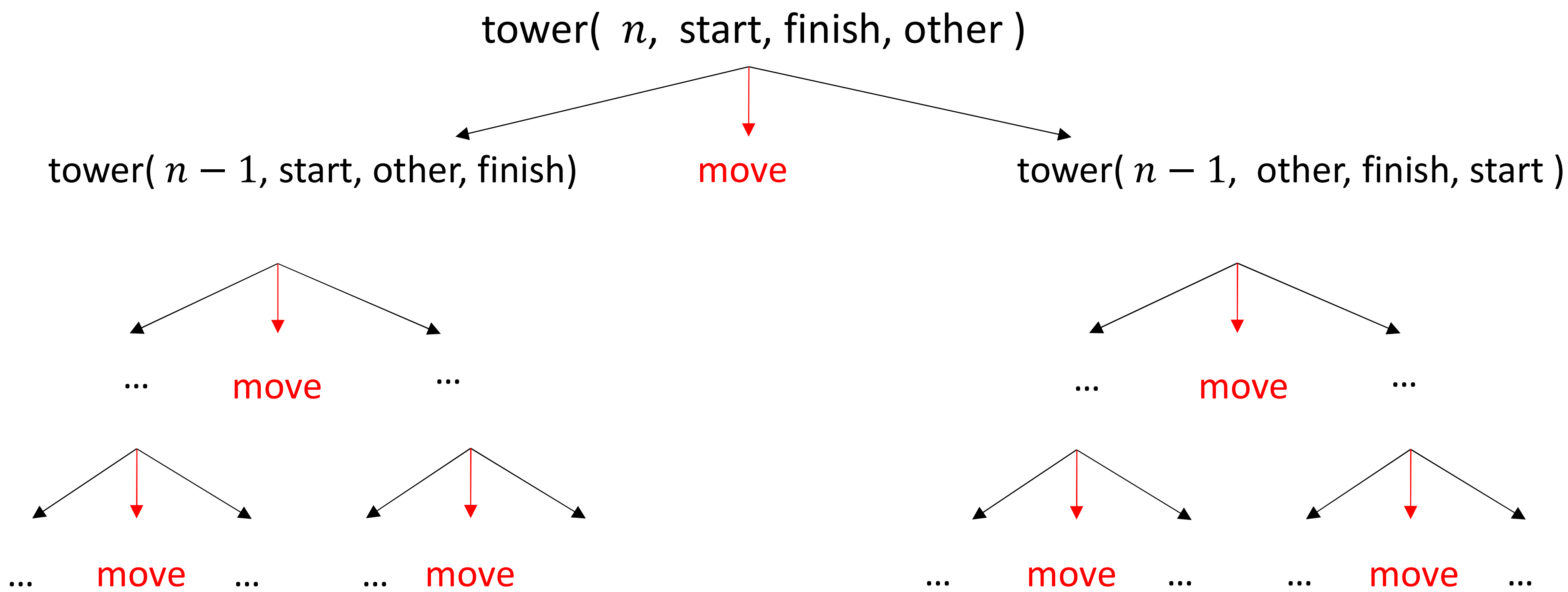
move from C to B

How many moves ?



Answer: $1 + 2 + 4 = 2^0 + 2^1 + 2^2$

How many moves ?



Answer: $1+2+4+\ldots+2^{n-1} = 2^{n} - 1$

(Geometric series. Recall lecture 3, slide 4.)

Recall (lecture 16): "call stack"

```
void   mA( ) {
            mB( );
            mC( );
       }

void  main( ){
            mA(  );
 }
```

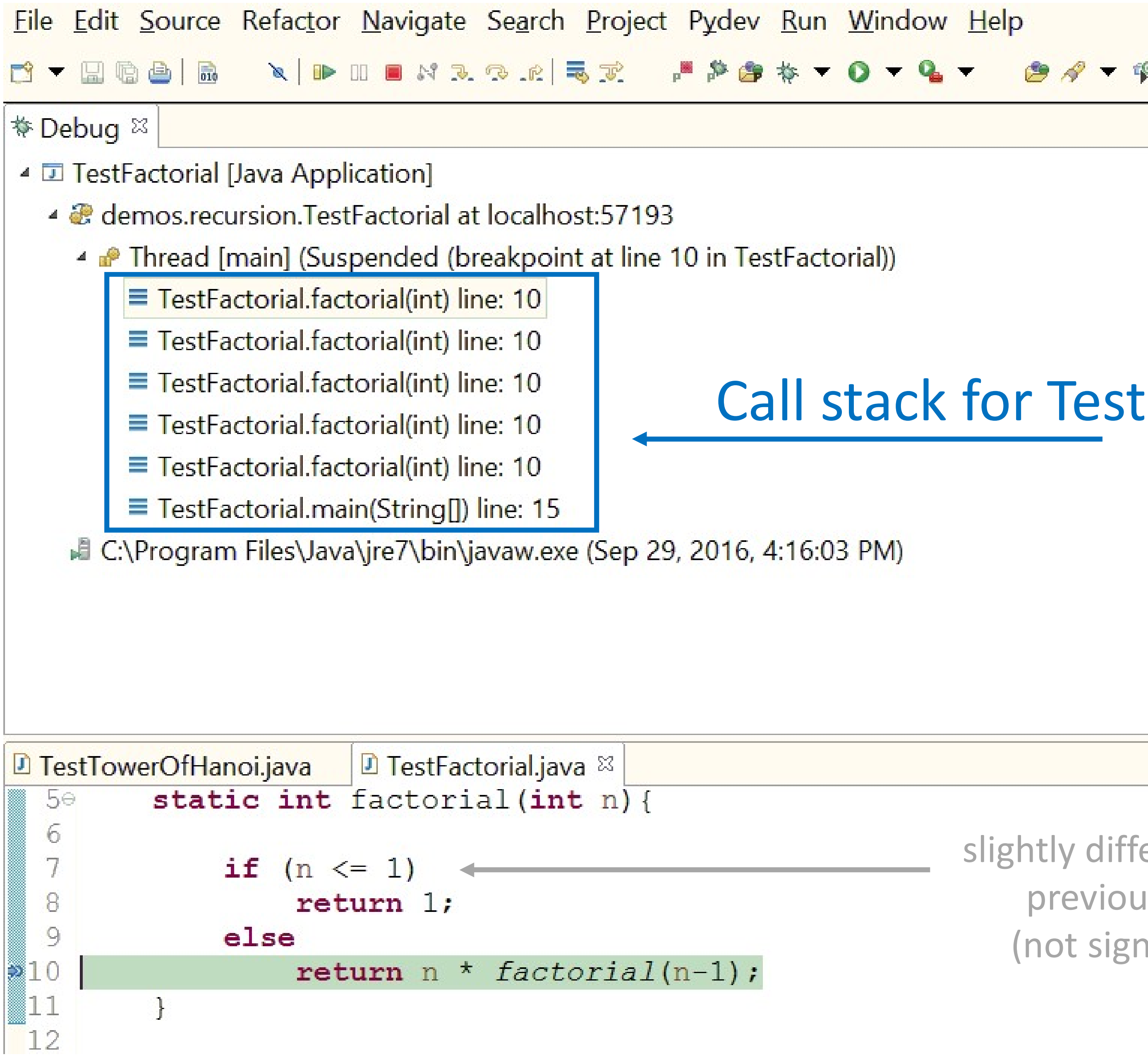
There is a single call stack for all methods.

		mB		mC		
	mA	mA	mA	mA	mA	
main	main	main	main	main	main	main

Recursive methods & Call stack

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

			factorial(0)			
		factorial(1)	factorial(1)	factorial(1)		
	factorial(2)	factorial(2)	factorial(2)	factorial(2)	factorial(2)	
main	main	main	main	main	main	main



Call stack for TestFactorial

slightly different from previous slide (not significant)

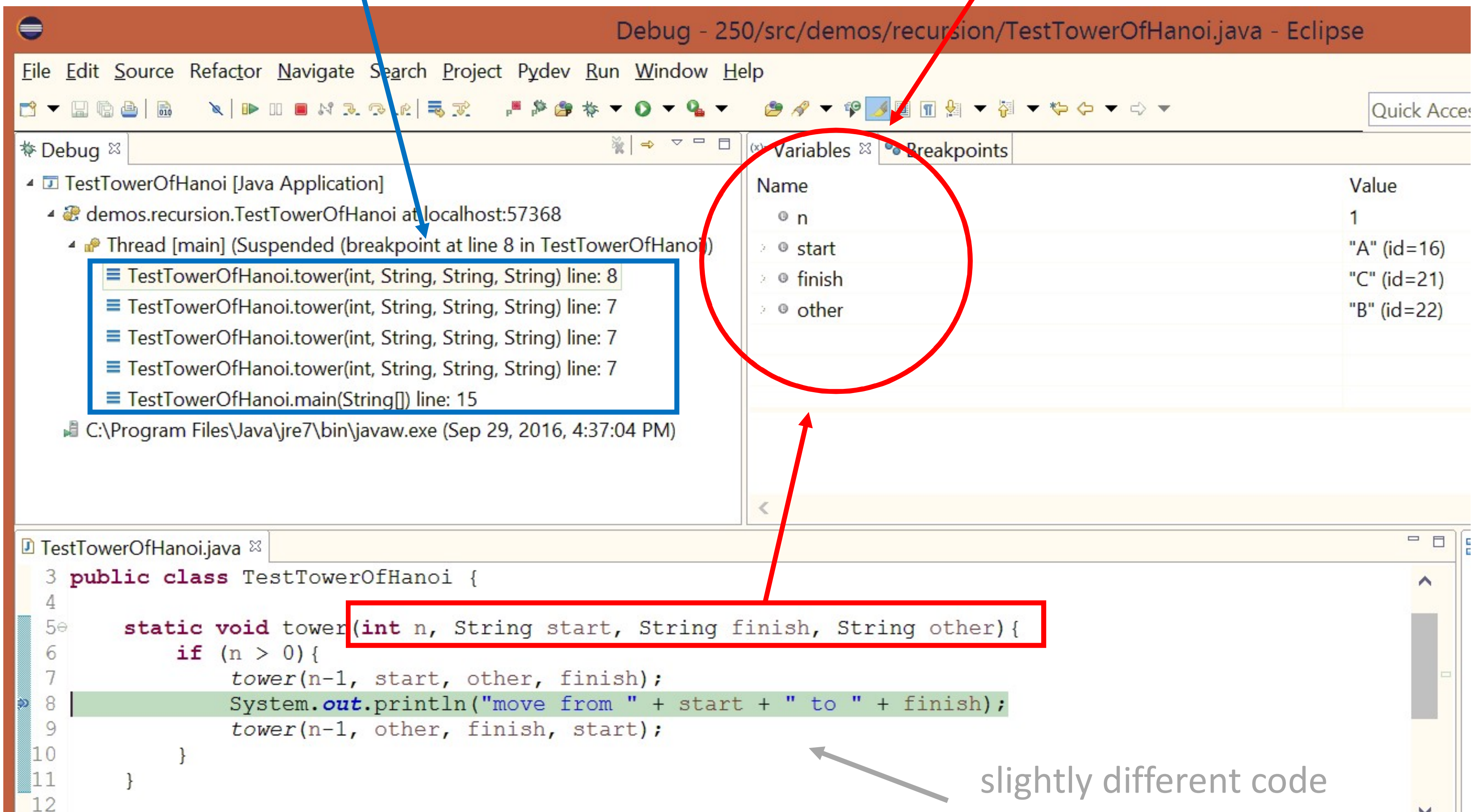
ASIDE: Stack frame
(details in COMP 273)

The call stack consists of "frames" that contain:

- the parameters passed to the method
- local variables of a method
- information about where to return ("which line number in which method in which class?")

Call stack for TestTowerOfHanoi

parameters in
current stack frame



slightly different code
from earlier slide
(not significant)

19. Induction
20. Recursion
21. **Binary Search**
22. **Mergesort & Quicksort**
23. **Trees**
24. **Tree traversal**
25. **Binary trees**
26. **Binary search trees**
27. **Heaps 1**
28. **Heaps 2**
29. Hashing 1 (maps)
30. Hashing 2
31. **Graphs 1**
32. **Graphs 2**
33. Big O 1
34. Big O 2
35. Big O 3
36. **Recurrences 1**
37. **Recurrences 2**

We will see recursive algorithms in all these lectures, and informally analyze computation complexity.

Here we will formally analyze the computation complexity of recursive algorithms.