

COMP 250

Lecture 11

recursive algorithms 1

Oct. 2, 2017

Example 1: Factorial (iterative)

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

```
factorial( n ){ // assume n >= 1
    result = 1
    for (k = 2; k <= n; k++)
        result = result * k
    return result
}
```

Factorial (recursive)

$$n! = (n - 1)! * n$$

```
factorial( n ){ // assume n >= 1
    if n == 1
        return 1
    else
        return factorial( n - 1 ) * n
}
```

Claim: the recursive factorial(n) algorithm returns $n!$.

Proof (by mathematical induction):

Base case: factorial(1) returns 1.

Induction step:

Take any $k \geq 1$.

if factorial(k) returns $k!$

then factorial($k + 1$) returns $(k + 1) !$

Example 2: Fibonacci

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

$$F(0) = 0$$

$$F(1) = 1$$

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

Fibonacci (iterative)

```
fibonacci(n){  
  if ((n == 0) | (n == 1))  
    return n  
  else{  
    fib0 = 0  
    fib1 = 1  
    for k = 2 to n{  
      fib2 = fib1 + fib0    // Fib(n+2)  
      fib0 = fib1           // Fib(n)    in next pass  
      fib1 = fib2           // Fib(n+1)  in next pass  
    }  
    return fib2  
  }  
}
```

Fibonacci (recursive)

```
fibonacci(n){ // assume n > 0
  if ((n == 0) || (n == 1))
    return n
  else
    return fibonacci(n-1) + fibonacci(n-2)
}
```

This is much simpler to express than the iterative version.

Claim: the recursive Fibonacci algorithm is correct.

Proof:

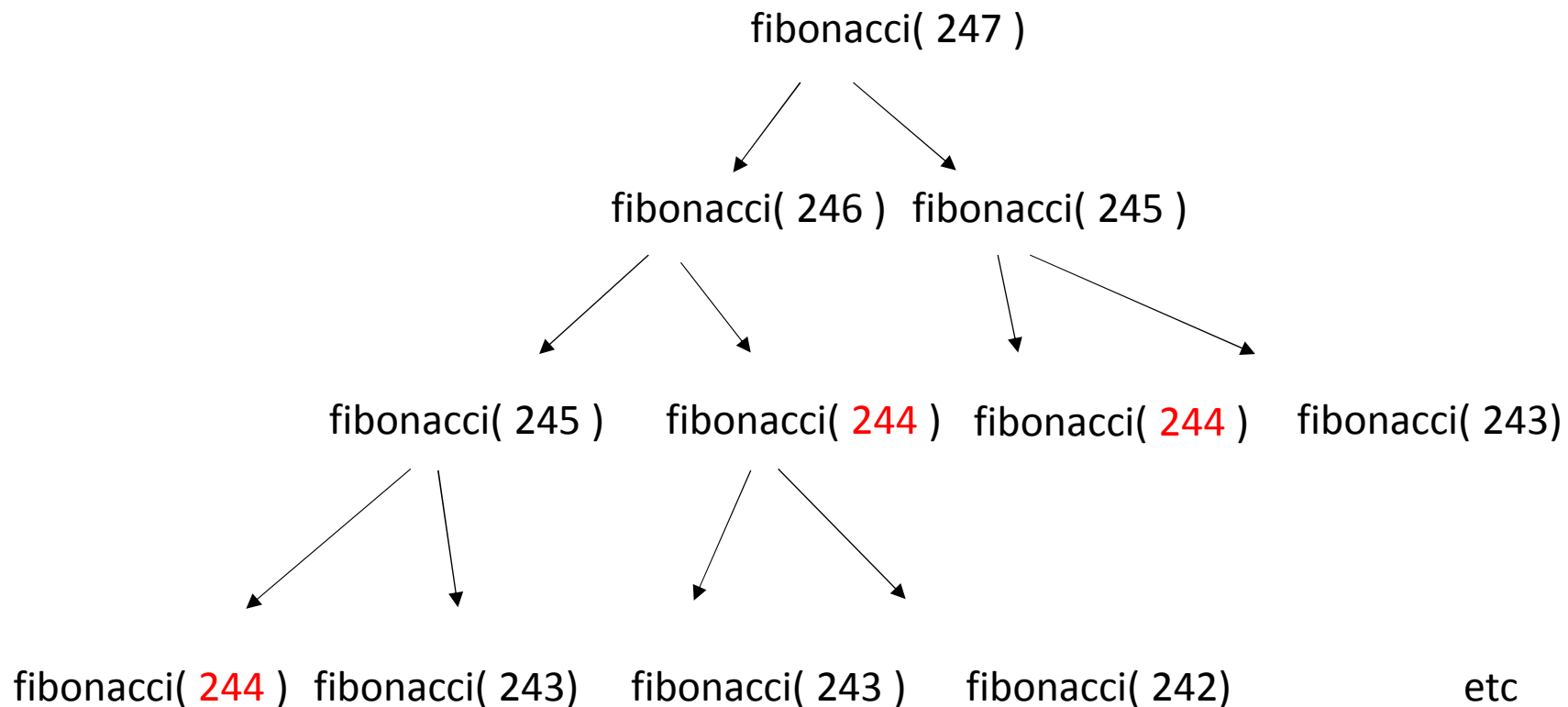
Base case: Fib (0) returns 0. Fib(1) returns 1.

Induction step:

for $k > 1$

if fibonacci($k-1$) and fibonacci(k) return $F(k-1)$ and $F(k)$
then fibonacci($k+1$) returns $F(k+1)$.

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



Example 3: Reversing a list

input (a b c d e f g h)

output (h g f e d c b a)

Example 3: Reversing a list

input (a b c d e f g h)

output (h g f e d c b a)

Idea of recursion:

 a (b c d e f g h)

 (h g f e d c b) a

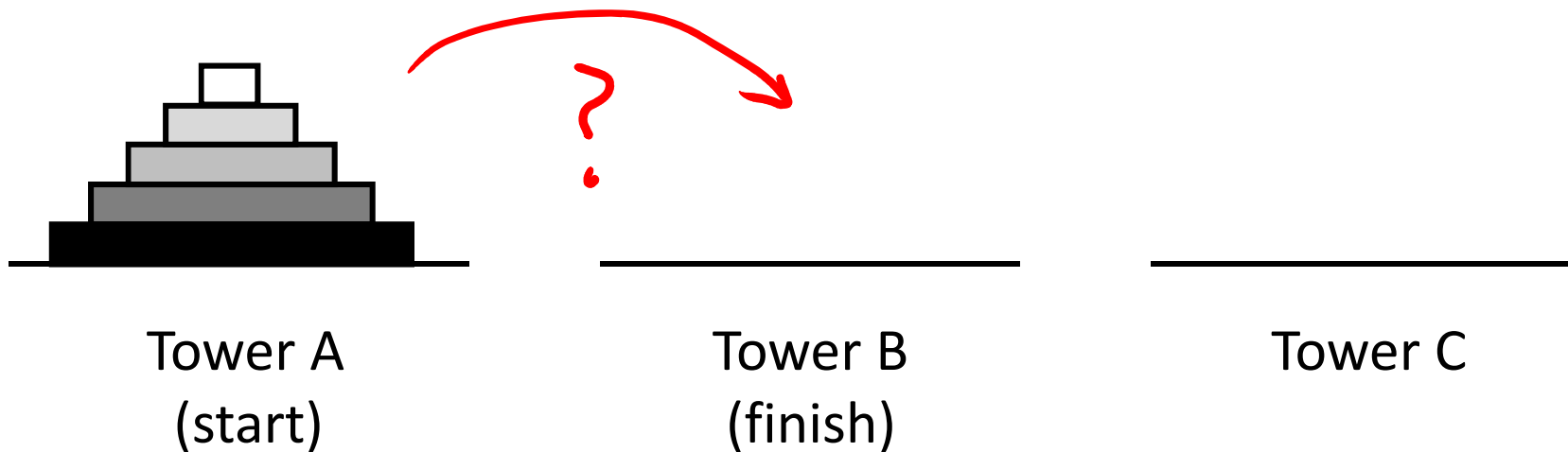
Example 3: Reversing a list (recursive)

```
reverse( list ){           // assume n > 0
    if list.size == 1      // base case
        return list
    else{
        firstElement = removeFirst(list)
        list = reverse(list) // list has only n-1 elements
        return addLast(list, firstElement )
    }
}
```

Example 4: Sorting a list (recursive)

```
sort( list ) {                                // assume size > 0
    if list.size == 1                          // base case
        return list
    else{
        minElement = removeMin(list)
        list = sort( list )                  // has n-1 elements
        return addFirst(list, minElement)
    }
}
// reminiscent of selection sort
```

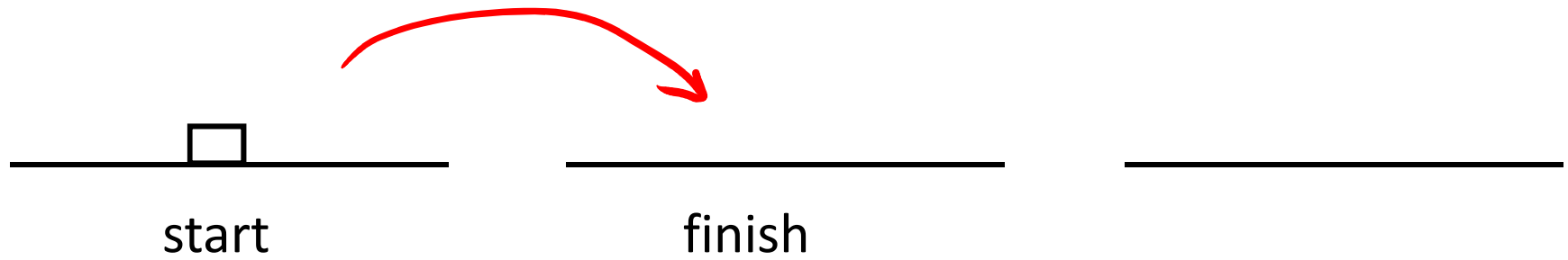
Example 5: Tower of Hanoi



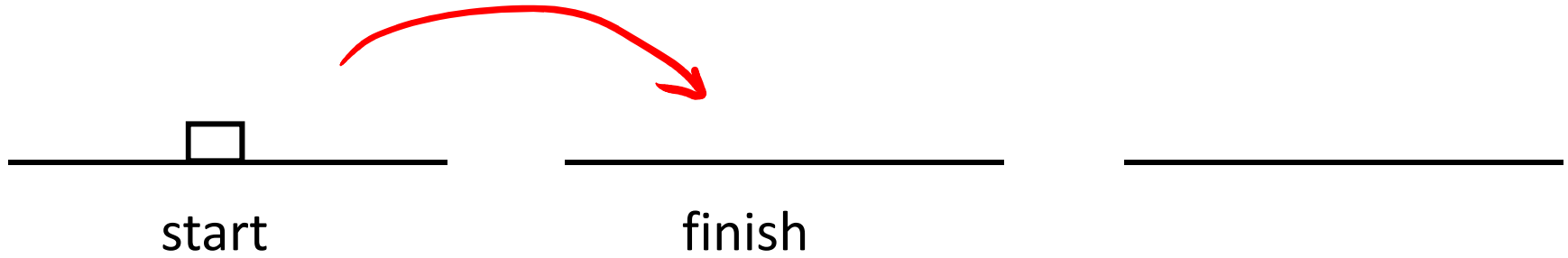
Problem: Move n disks from start tower to finish tower such that:

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

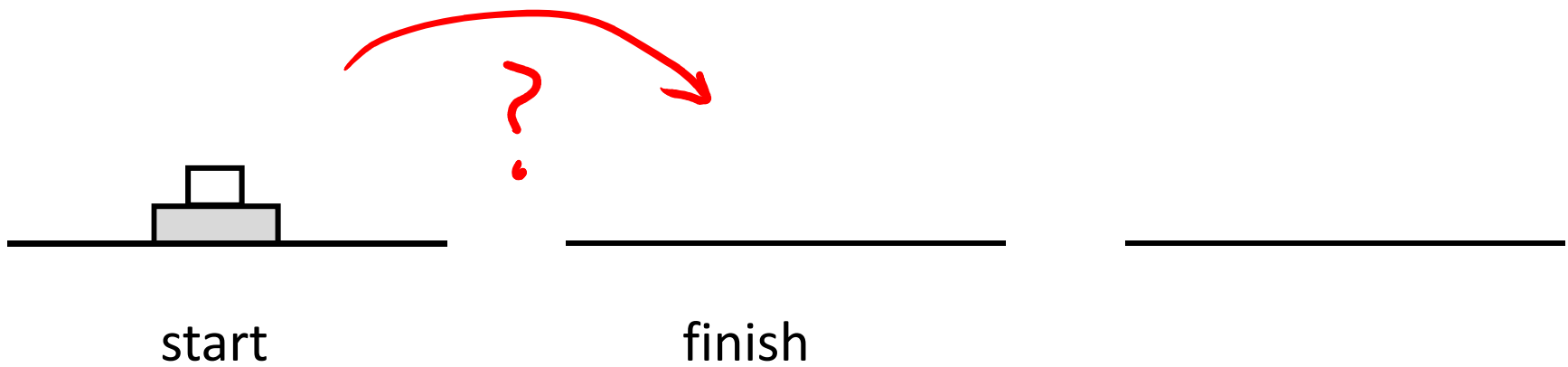
Example: $n = 1$



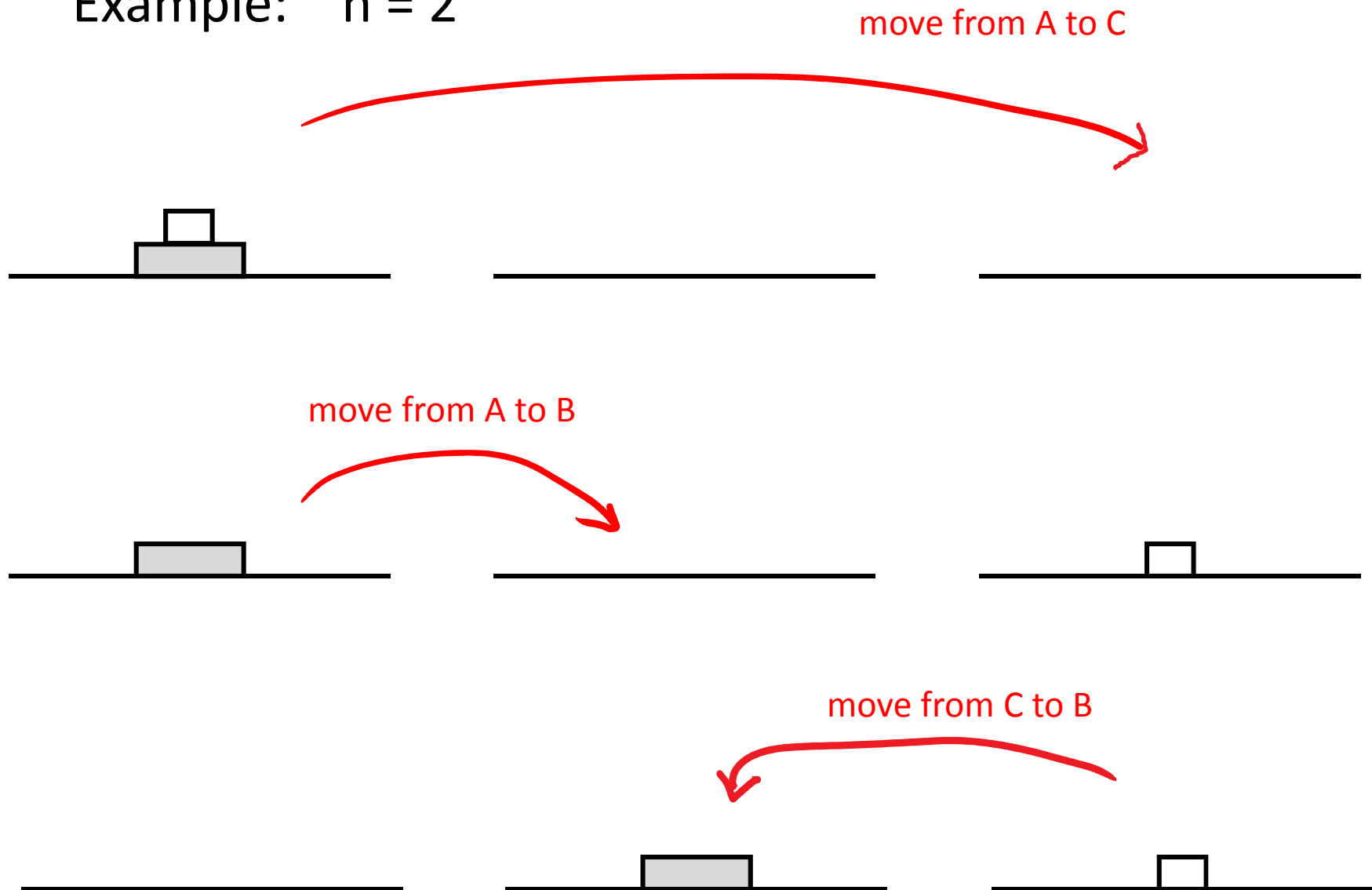
Example: $n = 1$



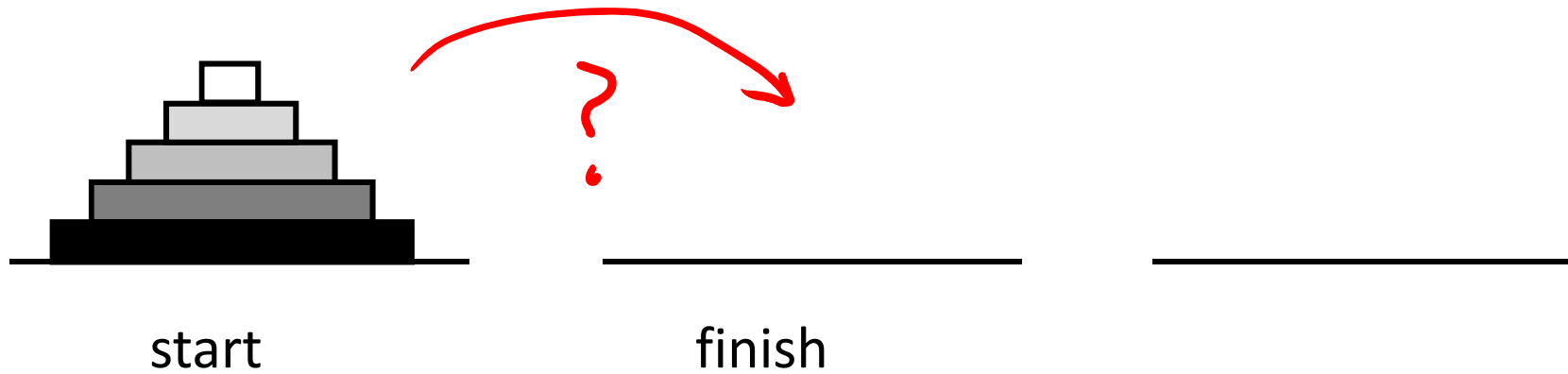
Example: $n = 2$



Example: $n = 2$



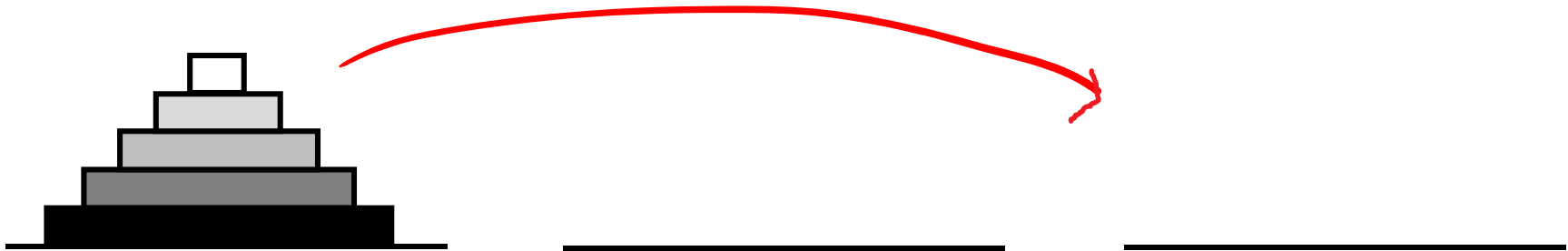
Q: How to move 5 disks from tower 1 to 2 ?



A: 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){ // e.g. tower(5, A, B, C)

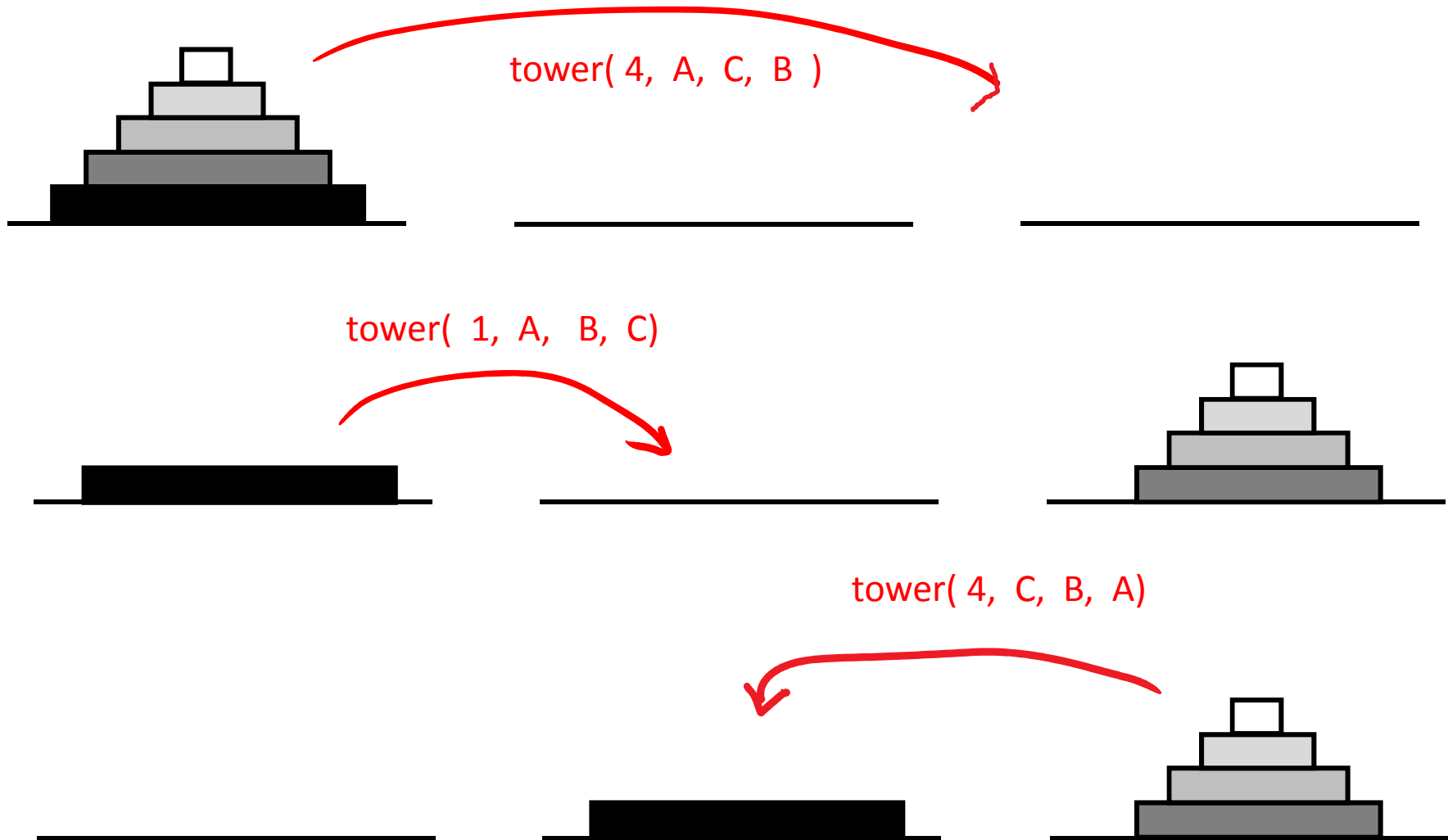
    if n > 0 {

        tower( n-1, start, other, finish)

        move from start to finish

        tower( n-1, other, finish, start)
    }
}
```

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



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).

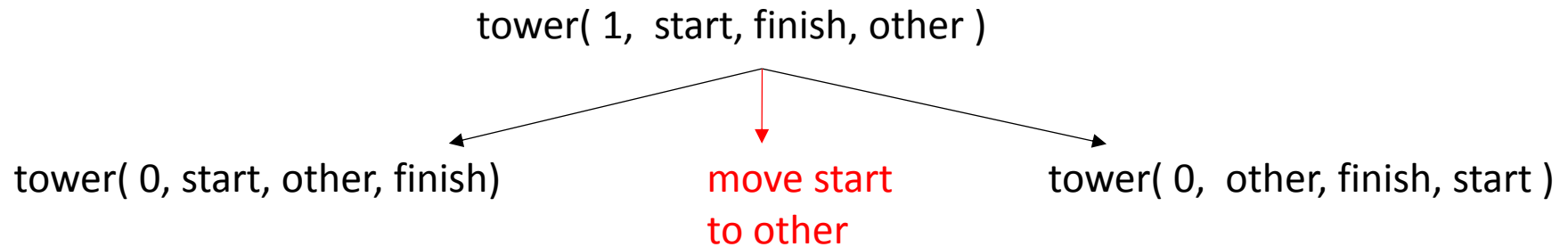
Proof: (sketch)

Base case: $\text{tower}(0, *, *, *)$ is correct.

Induction step:

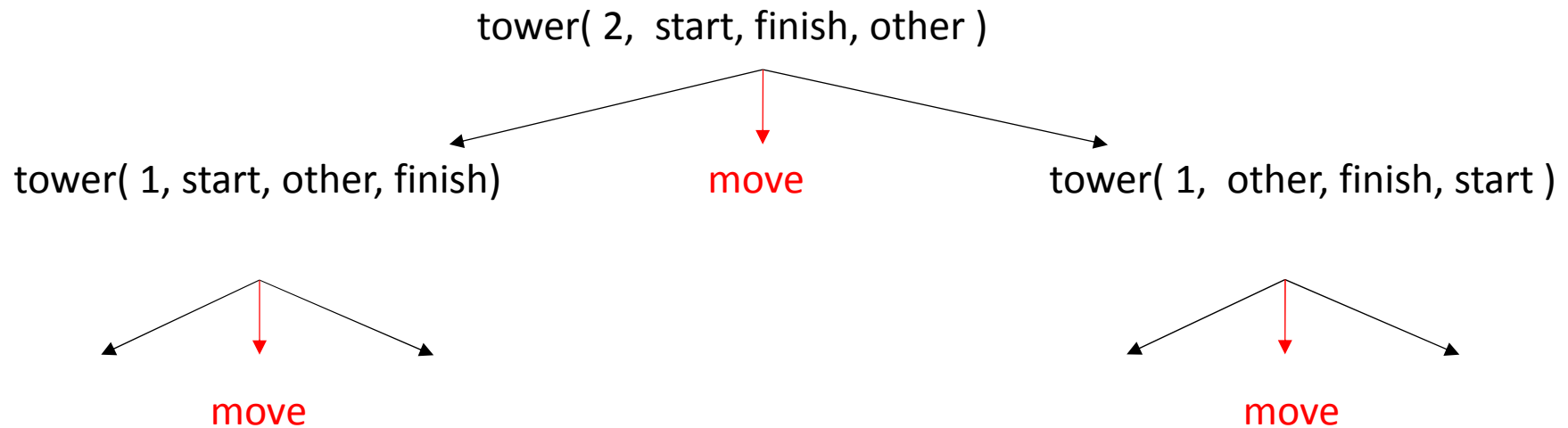
for any $k > 0$,
if $\text{tower}(k, *, *, *)$ is correct
then $\text{tower}(k + 1, *, *, *)$ is correct.

How many **moves** does tower(1, ...) make ?

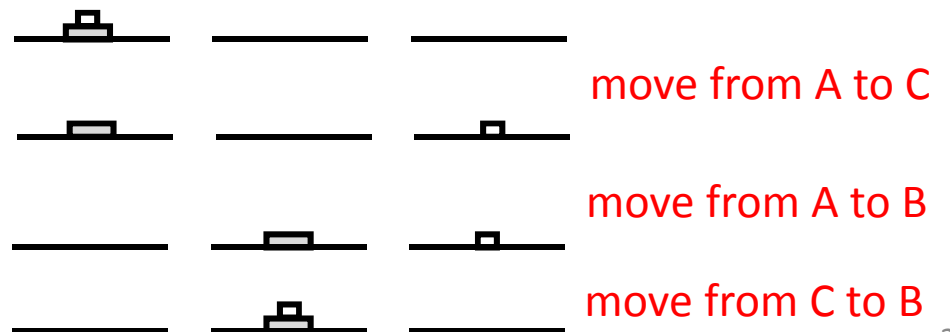


Answer: 1

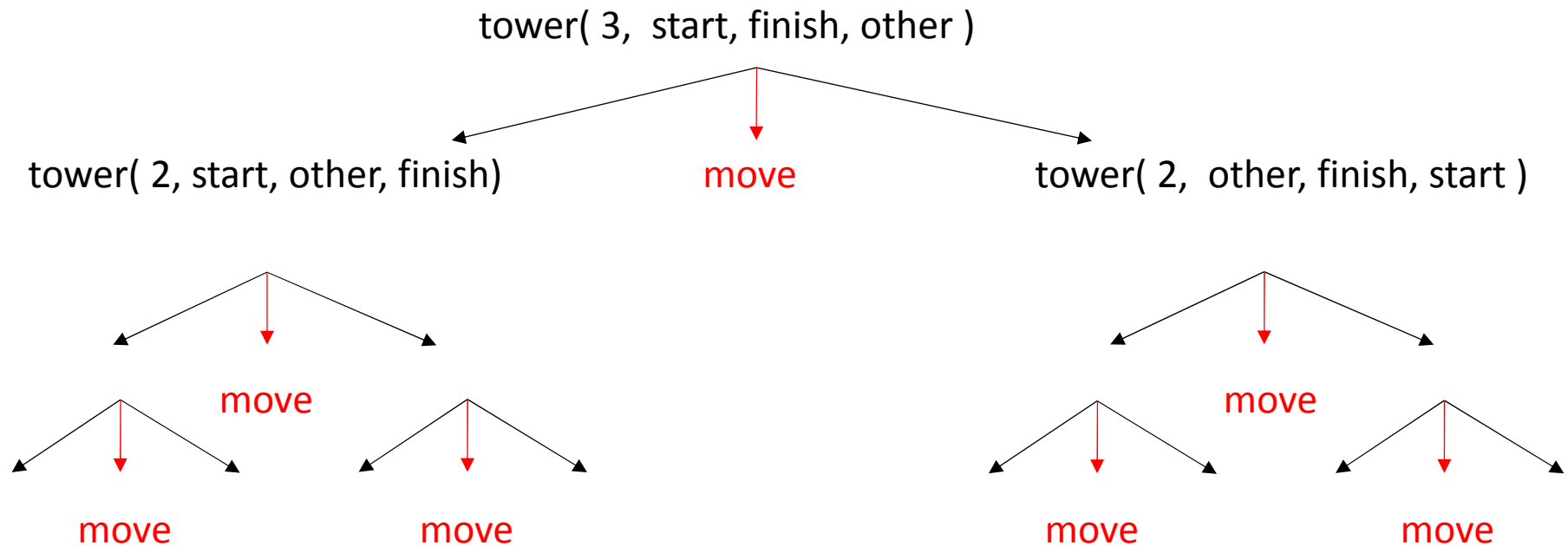
How many **moves** does tower(2, ...) make ?



Answer: 1 + 2

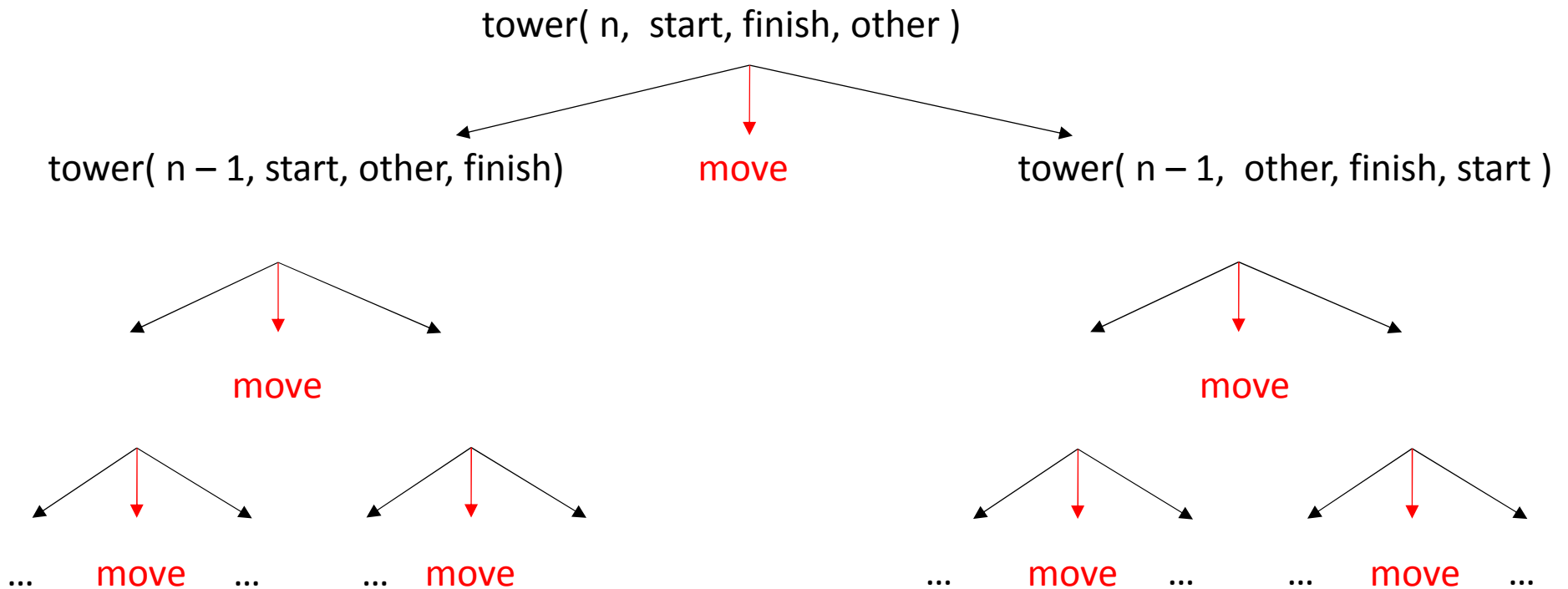


How many **moves** does tower(3, ...) make ?



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

How many **moves** does $\text{tower}(n, \dots)$ make ?



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

Recall (lecture 7): “call stack”

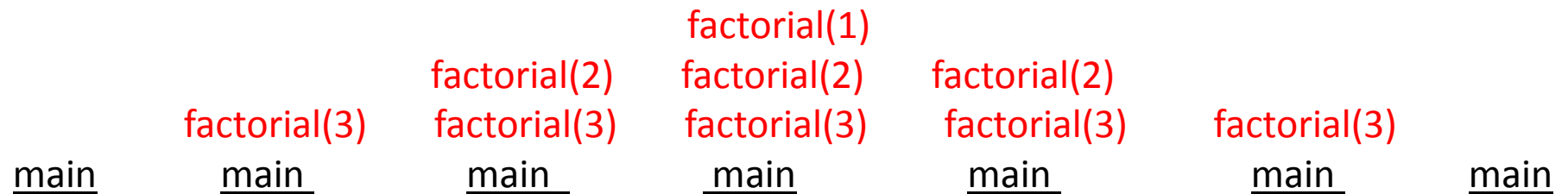
```
void mA( ) {  
    mB( );  
    mC( );  
}
```

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

		mB		mC		
	mA	mA	mA	mA	mA	
<u>main</u>	<u>main</u>	<u>main</u>	<u>main</u>	<u>main</u>	<u>main</u>	<u>main</u>

Recursive methods & Call stack

```
factorial( n ){  
    if n == 1  
        return 1  
    else  
        return factorial( n - 1 ) * n  
}
```



File Edit Source Refactor Navigate Search Project Pydev Run Window Help

Debug

- TestFactorial [Java Application]
 - demos.recursion.TestFactorial at localhost:57193
 - Thread [main] (Suspended (breakpoint at line 10 in TestFactorial))
 - TestFactorial.factorial(int) line: 10
 - TestFactorial.factorial(int) line: 10
 - TestFactorial.factorial(int) line: 10
 - TestFactorial.factorial(int) line: 10
 - TestFactorial.factorial(int) line: 10
 - TestFactorial.main(String[]) line: 15

C:\Program Files\Java\jre7\bin\javaw.exe (Sep 29, 2016, 4:16:03 PM)

Call stack for TestFactorial

TestTowerOfHanoi.java TestFactorial.java

```
5 static int factorial(int n) {  
6  
7     if (n <= 1)  
8         return 1;  
9     else  
10        return n * factorial(n-1);  
11  
12 }
```

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

The screenshot shows the Eclipse IDE in debug mode. The top toolbar includes icons for File, Edit, Source, Refactor, Navigate, Search, Project, Pydev, Run, Window, and Help. The main window displays the 'Debug' view with a call stack for 'TestTowerOfHanoi [Java Application]'. The call stack shows the following frames:

- TestTowerOfHanoi.tower(int, String, String, String) line: 8
- TestTowerOfHanoi.tower(int, String, String, String) line: 7
- TestTowerOfHanoi.tower(int, String, String, String) line: 7
- TestTowerOfHanoi.tower(int, String, String, String) line: 7
- TestTowerOfHanoi.main(String[]) line: 15

The 'Thread [main] (Suspended (breakpoint at line 8 in TestTowerOfHanoi))' is selected. The 'Variables' view on the right shows the parameters of the current stack frame:

Name	Value
n	1
start	"A" (id=16)
finish	"C" (id=21)
other	"B" (id=22)

The 'TestTowerOfHanoi.java' source code is visible at the bottom, showing the `tower` method signature and implementation. The parameters `(int n, String start, String finish, String other)` are highlighted in a red box.

```
3 public class TestTowerOfHanoi {
4
5     static void tower(int n, String start, String finish, String other) {
6         if (n > 0) {
7             tower(n-1, start, other, finish);
8             System.out.println("move from " + start + " to " + finish);
9             tower(n-1, other, finish, start);
10        }
11    }
12}
```

Call stack

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

A method can make both recursive and non-recursive calls.

There is a single call stack for all methods.