Recursion

- Recursion is more than just a programming technique. It has two other uses in computer science and software engineering, namely:
- as a way of describing, defining, or specifying things.
- as a way of designing solutions to problems (divide and conquer).

Recursive definitions

- A recursive definition is one in which something is defined in terms of itself
- Almost every algorithm that requires looping can be defined iteratively or recursively
- All recursive definitions require two parts:
  - *Base case*
  - *Recursive step*
- The recursive step is the one that is defined in terms of itself
- The recursive step must always move closer to the base case

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

```
int fact(int n) {
  if (n <= 1) return 1;
  else return n * fact(n-1);
} // fact
```

Recursion Definition

- In general, we can define the factorial function in the following way:

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} 1! &= 1 \\ 2! &= 1 \times 2 = 2 \\ 3! &= 1 \times 2 \times 3 = 6 \\ 4! &= 1 \times 2 \times 3 \times 4 = 24 \\ 5! &= 1 \times 2 \times 3 \times 4 \times 5 = 120 \end{aligned}$$

Iterative Definition

Iteration vs. recursion

- Some things (e.g. reading from a file) are easier to implement iteratively
- Other things (e.g. mergesort) are easier to implement recursively
- Others are just as easy both ways

- It can be proved that two methods performing the same task, one implementing an iteration algorithm and one implementing a recursive version, are equivalent

مضروب، Factorial

$$\text{Factorial}(5) = 5 * \overbrace{4 * 3 * 2 * 1}^{\text{Factorial }(4)}$$

$$\text{Factorial}(4) = 4 * \overbrace{3 * 2 * 1}^{\text{Factorial }(3)}$$

$$\text{Factorial}(3) = 3 * \overbrace{2 * 1}^{\text{Factorial }(2)}$$

$$\text{Factorial}(2) = 2 * \overbrace{1}^{\text{Factorial }(1)}$$

$$\text{Factorial}(1) = 1$$

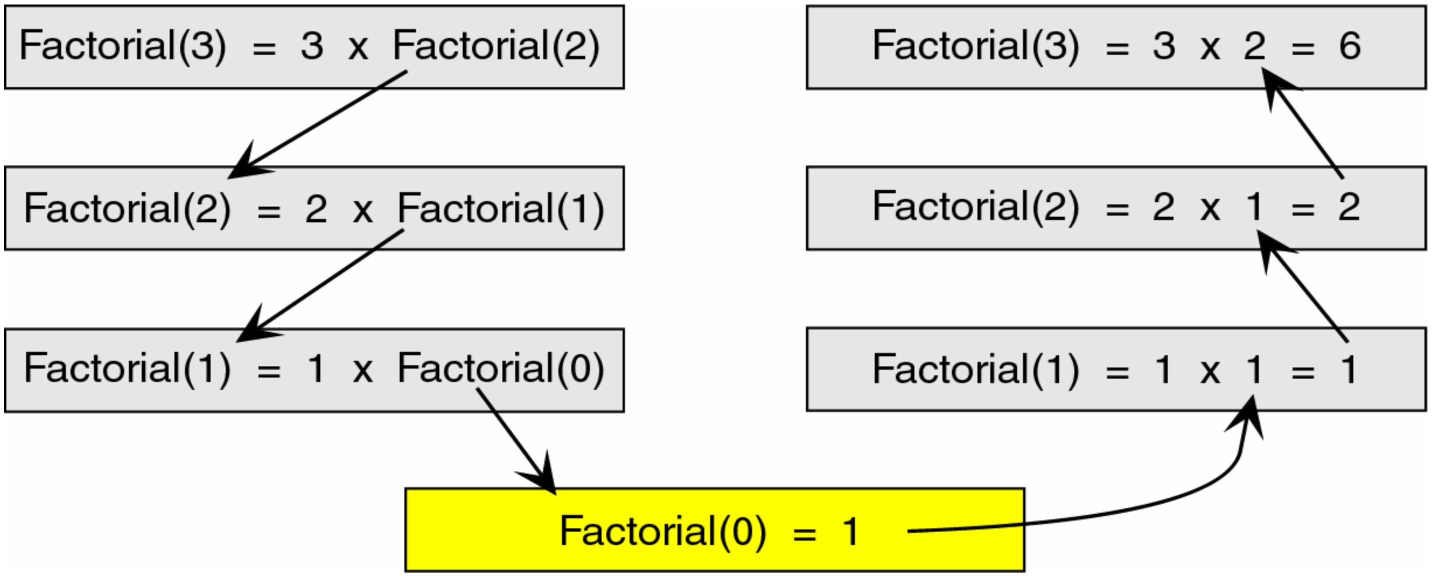
جزاء متكررة (العدد ضرب

factorial / العدد الذي قبله)

Factorial (x) = x * Factori

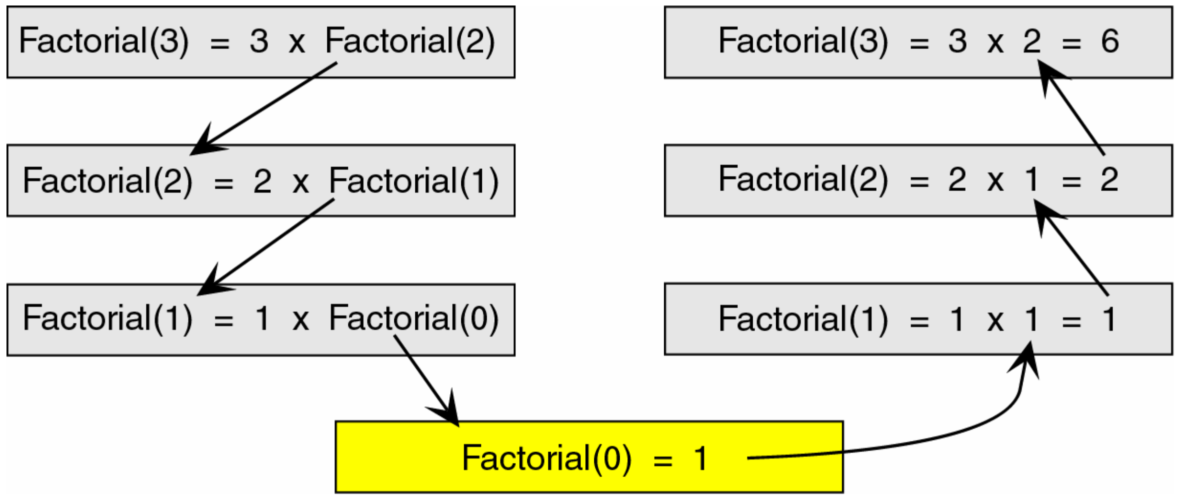
جزء يوقف فيه التكرار base

Breakdown



- Here, we see that we start at the top level, factorial(3), and simplify the problem into 3 x factorial(2).
- Now, we have a slightly less complicated problem in factorial(2), and we simplify this problem into 2 x factorial(1).

Breakdown



- We continue this process until we are able to reach a problem that has a known solution.
- In this case, that known solution is factorial(0) = 1.
- The functions then return in reverse order to complete the solution.

Breakdown

- This known solution is called the ***base case***.
- Every recursive algorithm must have a base case to simplify to.
- Otherwise, the algorithm would run forever (or until the computer ran out of memory).

Iterative Algorithm

```
factorial(n) {
    i = 1
    factN = 1
    loop (i <= n)
        factN = factN * i
        i = i + 1
    end loop
    return factN
}
```

The iterative solution is very straightforward. We simply loop through all the integers between 1 and n and multiply them together.

Recursive Algorithm

```
factorial(n) {
   if (n = 0)
        return 1
   else
        return n*factorial(n-1)
   end if
}
```

Note <u>**how much simpler the code**</u> for the recursive version of the algorithm is as compared with the iterative version ➔

we have eliminated the loop and implemented the algorithm with 1 ‘if’ statement.

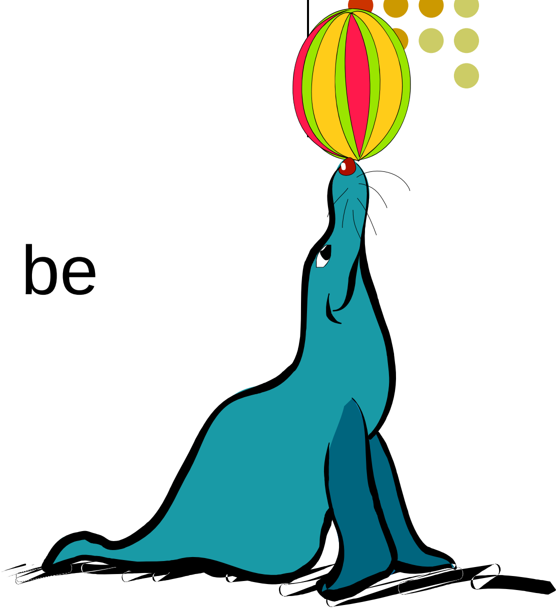
How Recursion Works

- When the function is finished, it needs to return to the function that called it.
- The calling function then ‘wakes up’ and continues processing.

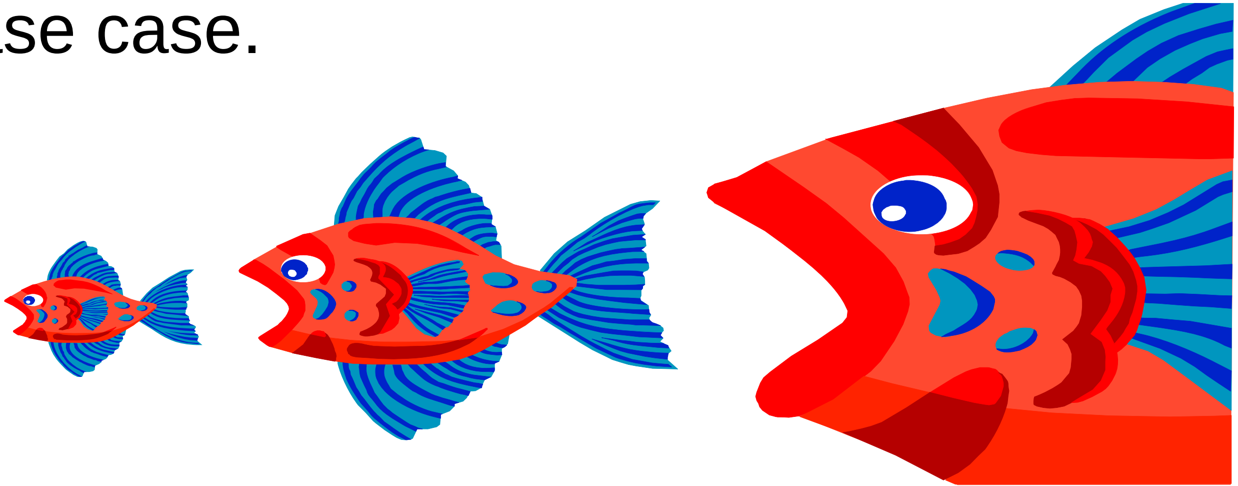
How Recursion Works

- To do this we use a stack.
- Before a function is called, all relevant data is stored in a ***stackframe***.
- This stackframe is then pushed onto the system stack.
- After the called function is finished, it simply pops the system stack to return to the original state.

Basic Recursion



- 1. Base cases:
  - Always have at least one case that can be solved without using recursion.
- 2. Make progress:
  - Any recursive call must make progress toward a base case.



Advantage and Limitations of Recursion

- Recursive solutions can be easier to understand and to describe than iterative solutions.

- Recursion works the best when the algorithm and/or data structure that is used naturally supports recursion.

- One such data structure is the tree (more to come).

- One such algorithm is the binary search algorithm that we discussed earlier in the course.

acci Numbers

0, 1, 1, 2, 3, 5, 8, 13, 21 .....

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*...*
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	...

Fibonacci (0) =0

Fibonacci (1) = 1

Fibonacci (x) = Fibonacci (x-1) + Fibonacc

Fib (7) = ?

5 4 3 2 1

rn 0

==1)

rn 1

rn Fib(x-1) + Fib (x-2)

Fib (7) = Fib (6) +

Fib (6) = Fib (5) +

Fib (5) = Fib (4) +

Fib (4) = Fib (3) +

Fib (3) = Fib (2) +

Fib (2) = Fib (1) +

Fib(1)= 1

Fib (7) = ?

5 4 3 2 1 0

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
0	1	1	2	3	5	8	13	21

rn 0

==1)

rn 1

rn Fib(x-1) + Fib (x-2)

8

13 ~~Fib (7) = Fib (6)~~ +

5

8 ~~Fib (6) = Fib (5)~~ +

3

5 ~~Fib (5) = Fib (4)~~ +

2

3 ~~Fib (4) = Fib (3)~~ +

1

2 ~~Fib (3) = Fib (2)~~ +

1

1 ~~Fib (2) = Fib (1)~~ +

1 ~~Fib(1)= 1~~

0 ~~Fib(0)= 0~~

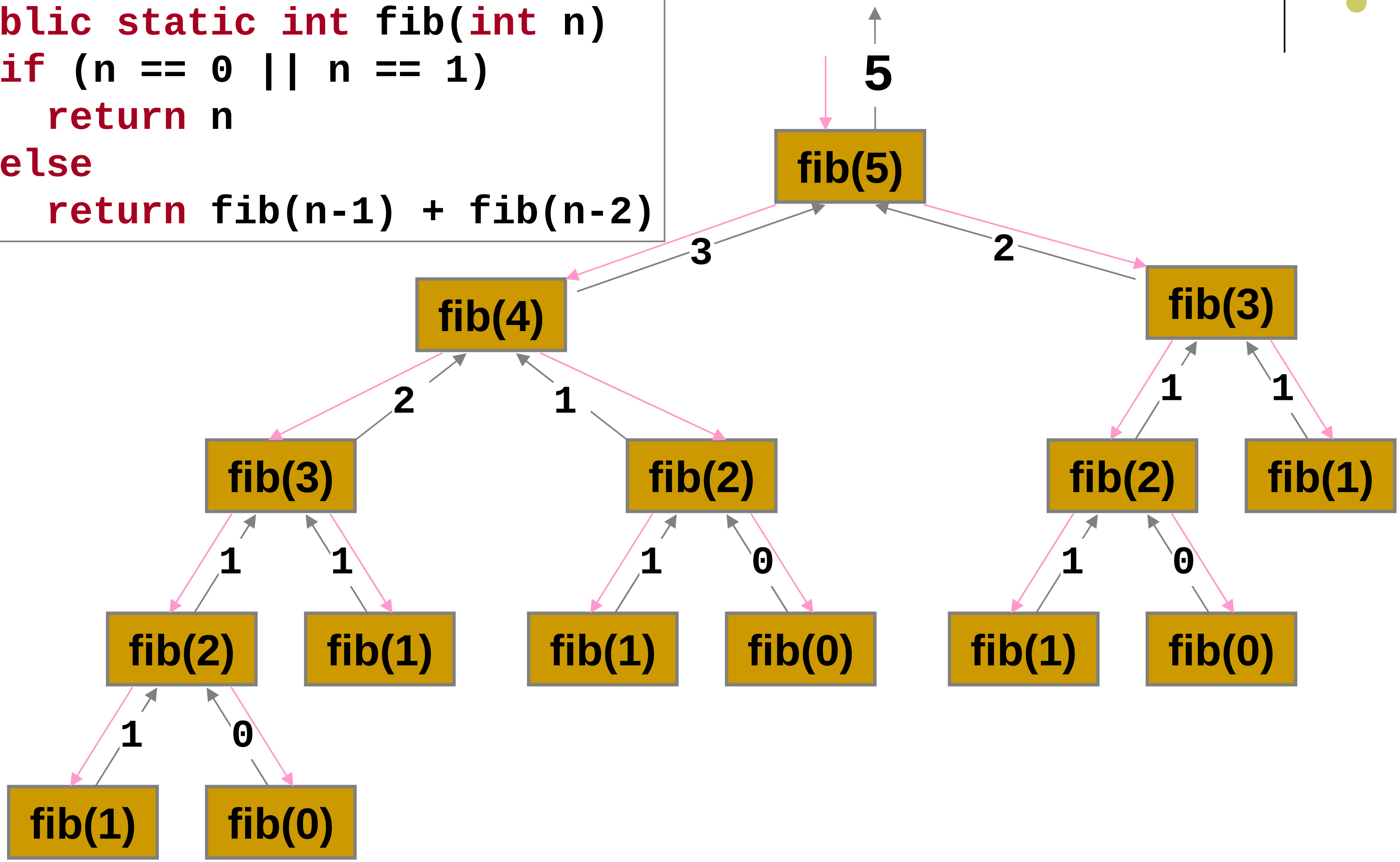
Fibonacci function:

- fibonacci(0) = 1
- fibonacci(1) = 1
- fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
- [for n>1]
- **This definition is a little different than the previous ones because It has two base cases, not just one; in fact, you can have as many as you like.**

- **In the recursive case, there are two recursive calls, not just one. There can be as many as you like.**

Function Analysis for call `fib(5)`

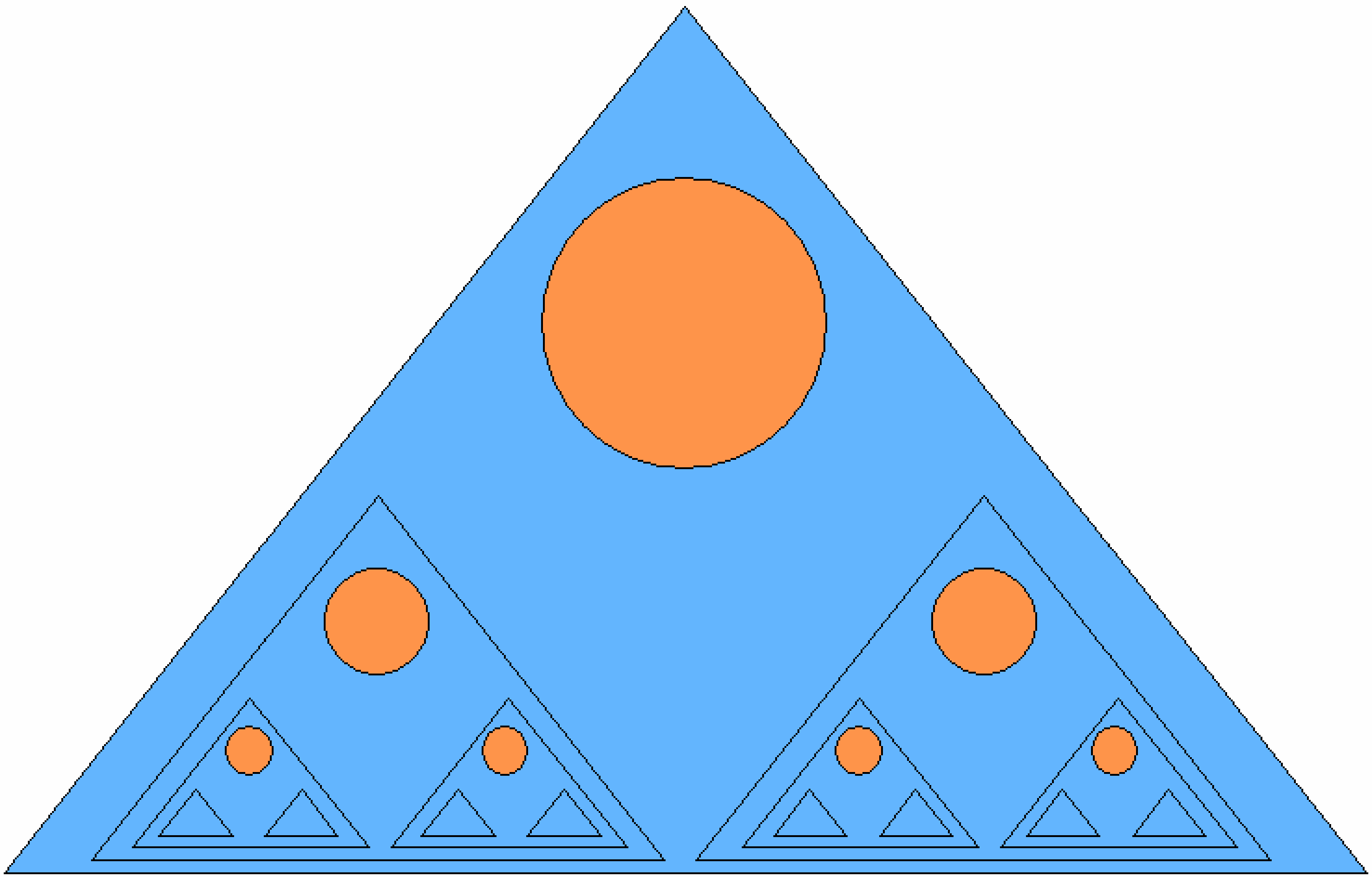
```
public static int fib(int n)
  if (n == 0 || n == 1)
    return n
  else
    return fib(n-1) + fib(n-2)
```



Conclusion

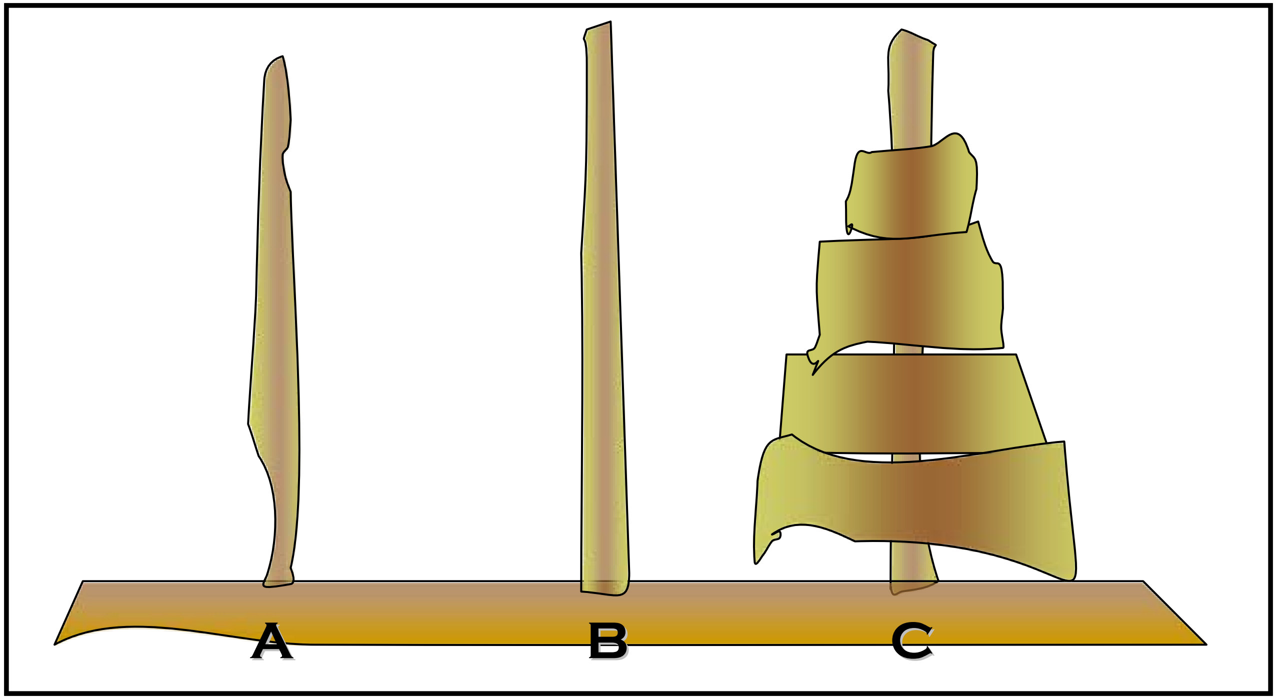
- A recursive solution solves a problem by solving a smaller instance of the same problem.
- It solves this new problem by solving an even smaller instance of the same problem.
- Eventually, the new problem will be so small that its solution will be either obvious or known.
- This solution will lead to the solution of the original problem.

A *binary tree* is either empty, or it consists of a node part (an element) and two subtree parts. Each of the subtree parts are themselves binary trees.



Towers of Hanoi

- Move n (4) disks from pole A to pole C
- such that a disk is never put on a smaller disk



- Move n (4) disks from A to C
  - Move n-1 (3) disks from A to B
  - Move 1 disk from A to C
  - Move n-1 (3) disks from B to C

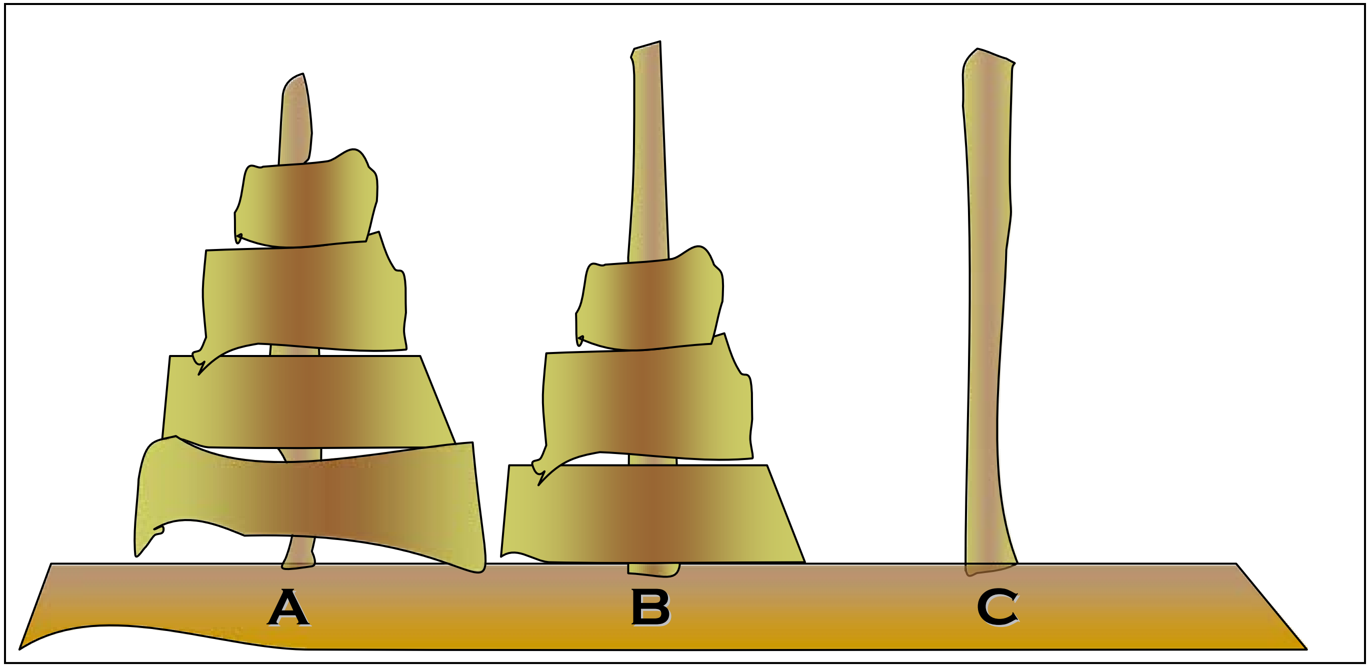


Figure 2.19a and b

a) The initial state; b) move *n* - 1 disks from *A* to *C*

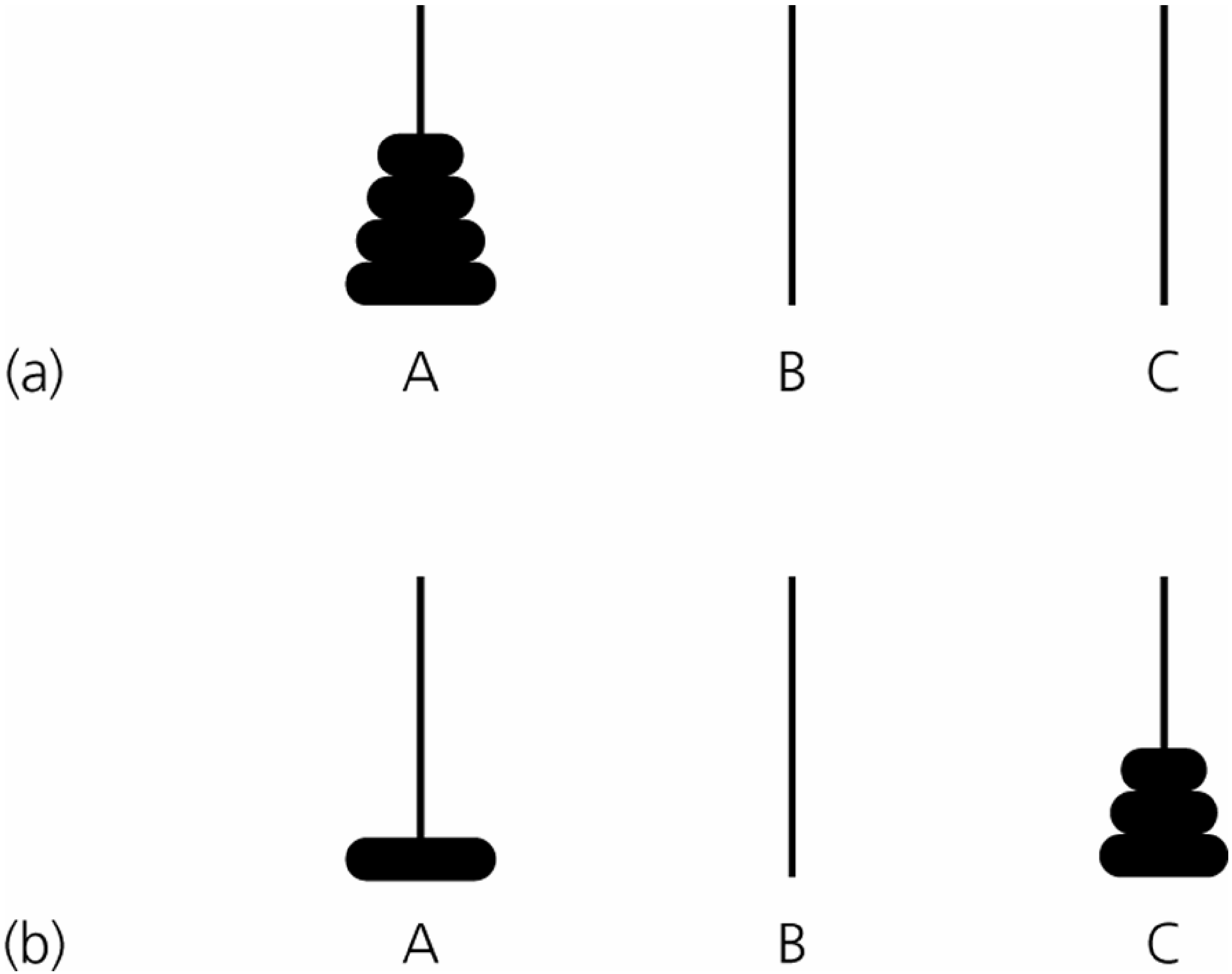
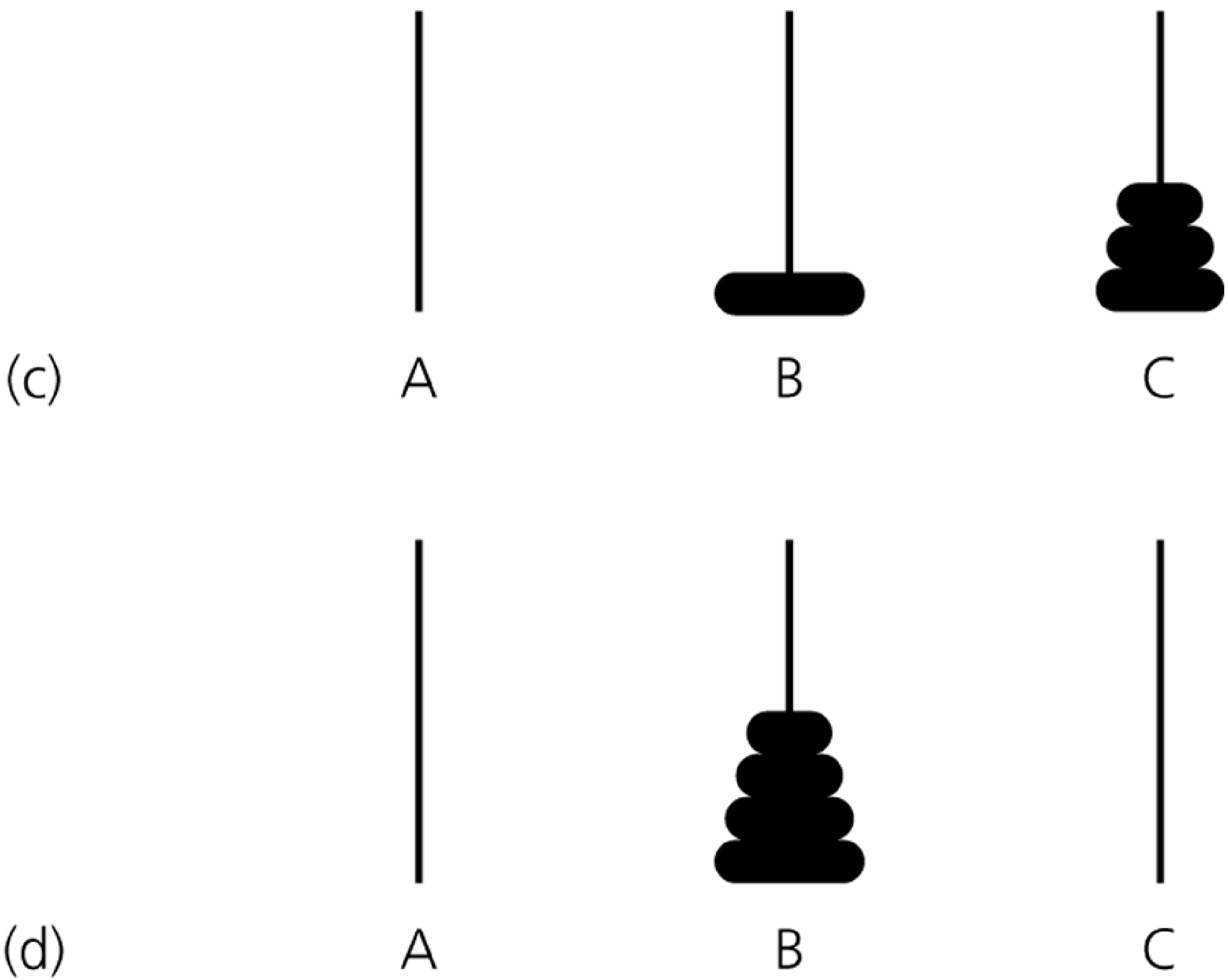


Figure 2.19c and d

c) move one disk from *A* to *B*; d) move *n* - 1 disks from *C* to *B*

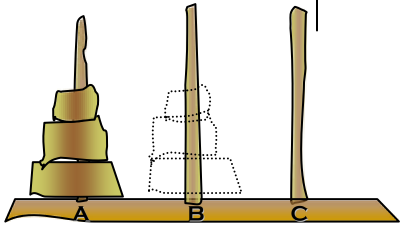


Hanoi towers

```
public static void solveTowers(int count, char source,
                               char destination, char spare) {
  if (count == 1) {
    System.out.println("Move top disk from pole " + source +
                       " to pole " + destination);
  }
  else {
    solveTowers(count-1, source, spare, destination); // X
    solveTowers(1, source, destination, spare);       // Y
    solveTowers(count-1, spare, destination, source); // Z
  }  // end if
}  // end solveTowers
```

Figure 2.21a

Box trace of ***solveTowers(3, 'A', 'B', 'C')***



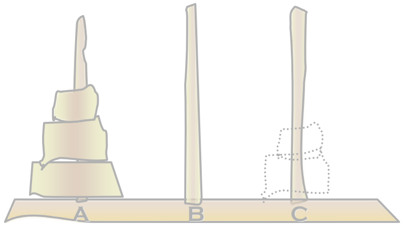
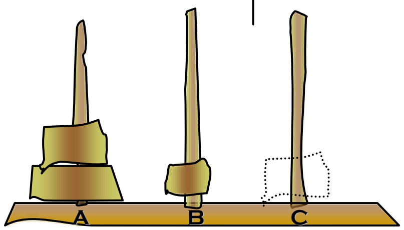


Figure 2.21b

Box trace of ***solveTowers(3, ‘A’, ‘B’, ‘C’)***



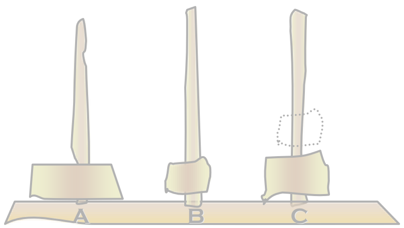
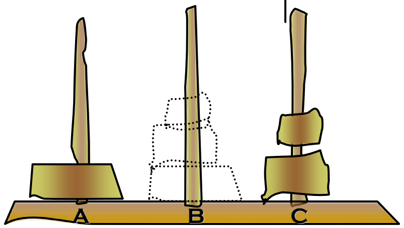


Figure 2.21c

Box trace of **_solveTowers(3, ‘A’, ‘B’, ‘C’)_**



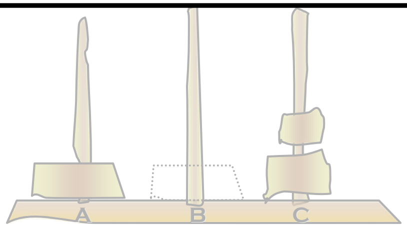
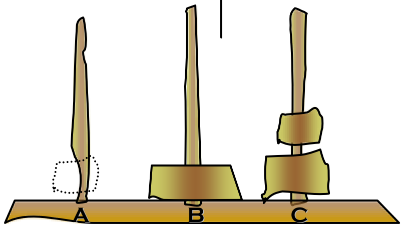


Figure 2.21d

Box trace of ***solveTowers(3, ‘A’, ‘B’, ‘C’)***



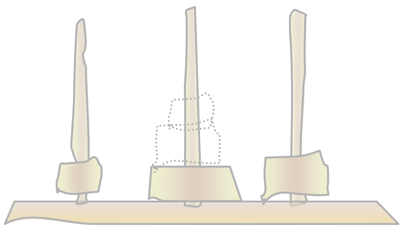
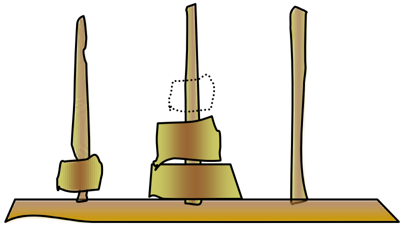
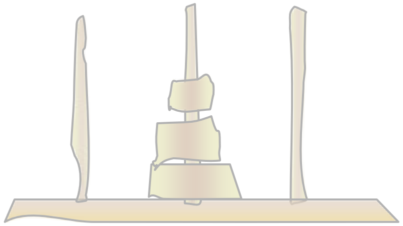


Figure 2.21e

Box trace of ***solveTowers(3, 'A', 'B', 'C')***





Cost of Hanoi Towers

- How many moves is necessary to solve Hanoi Towers problem for N disks?
- moves(1) = 1
- moves(N) = moves(N-1) + moves(1) + moves(N-1)
- i.e.
  moves(N) = 2*moves(N-1) + 1
- Guess solution and show it's correct with Mathematical Induction!