Recursion

Chapter 3.5

Divide and Conquer

- When faced with a difficult problem, a classic technique is to break it down into smaller parts that can be solved more easily.
- Recursion is one way to do this.

Recursive Divide and Conquer

- You are given a problem input that is too big to solve directly.
- You imagine,
  - "Suppose I had a friend who could give me the answer to the same problem with slightly smaller input."
  - "Then I could easily solve the larger problem."
- In recursion this "friend" will actually be another instance (clone) of yourself.

Tai (left) and Snuppy (right): the first puppy clone.

Friends & Strong Induction

Recursive Algorithm:

- Assume you have an algorithm that works.
- Use it to write an algorithm that works.

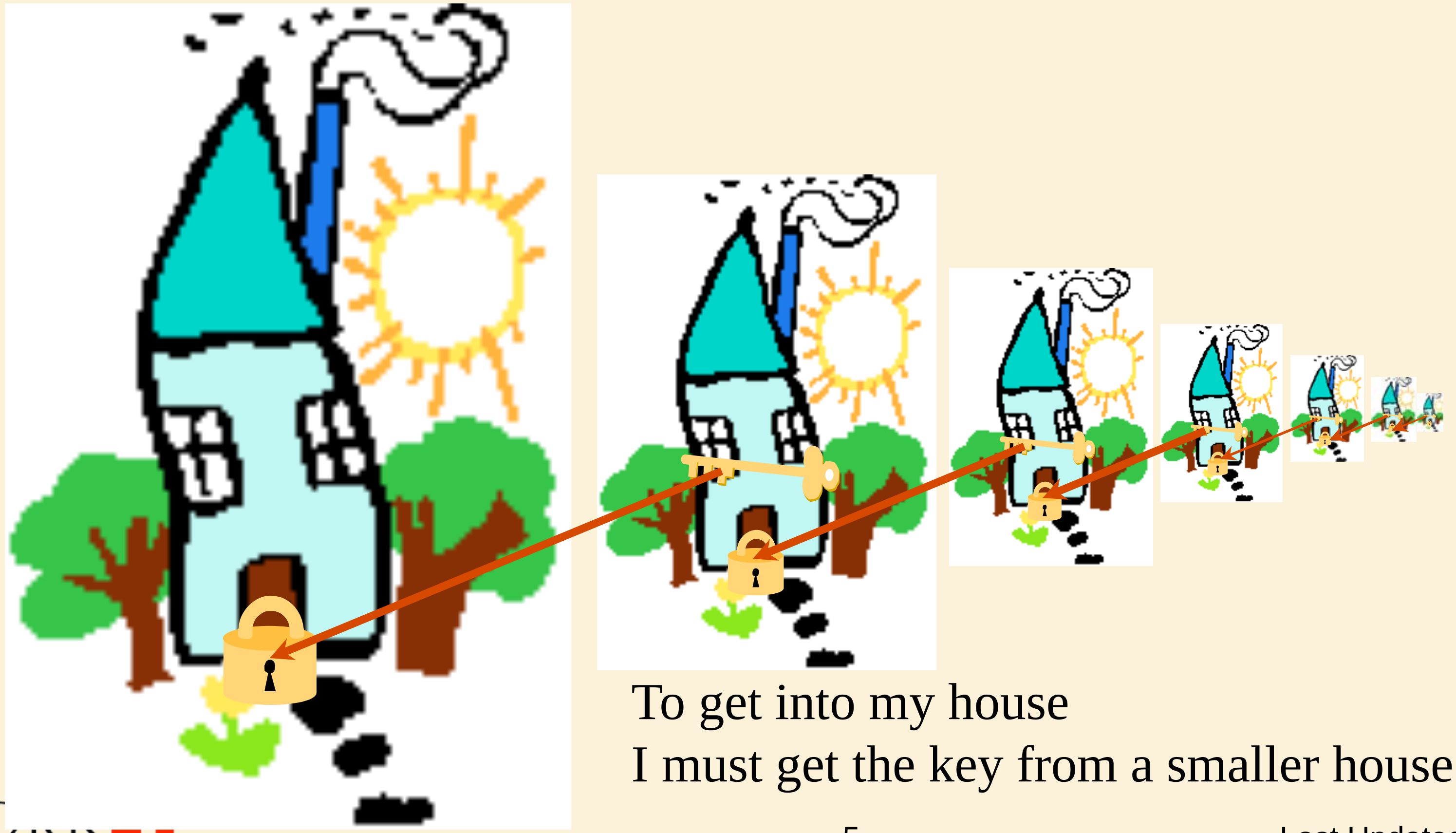
If I could get in,
I could get the key.
Then I could unlock the door
so that I can get in.

Circular Argument!

Friends & Strong Induction

Recursive Algorithm:

- Assume you have an algorithm that works.
- Use it to write an algorithm that works.

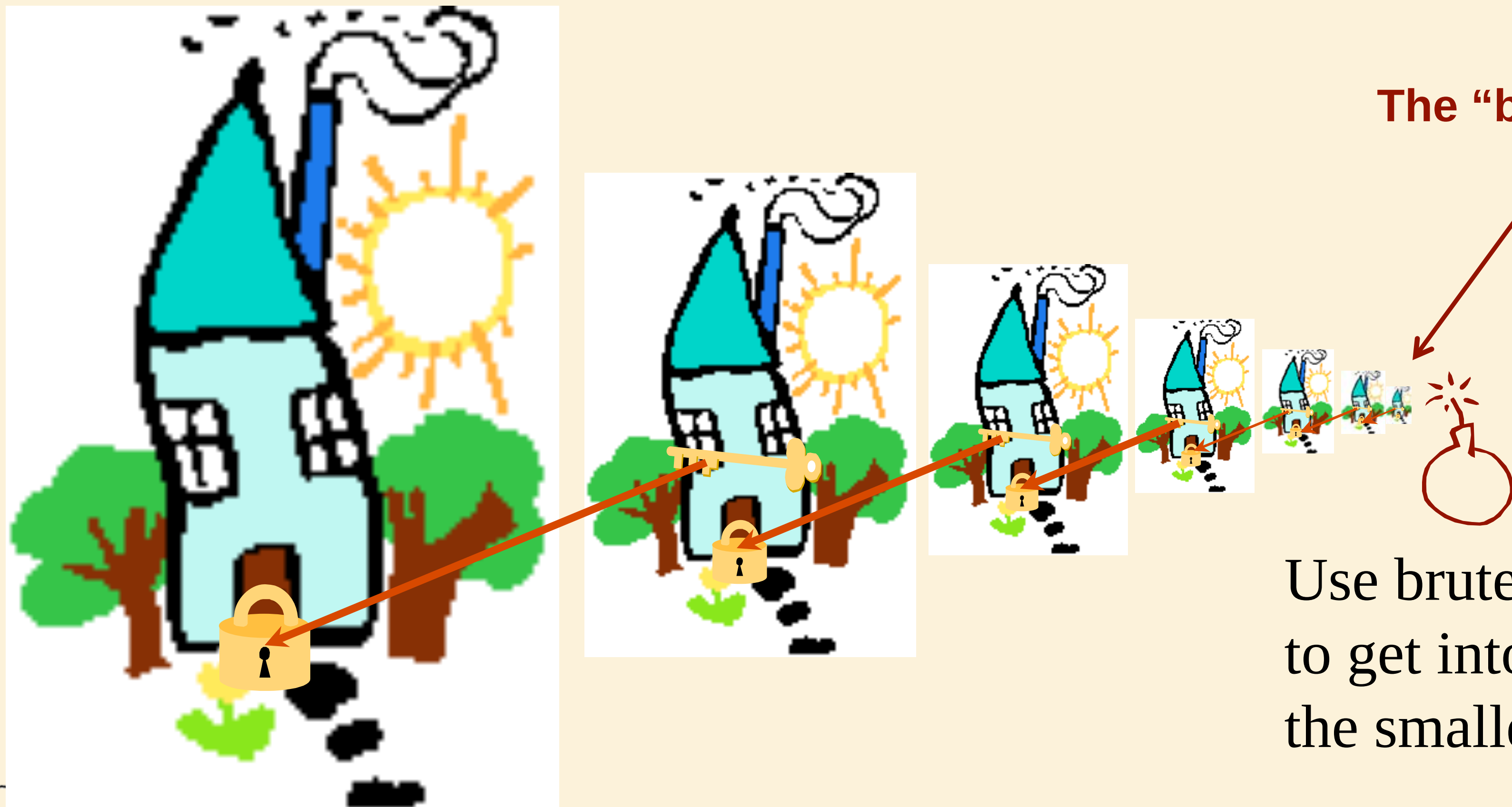


To get into my house
I must get the key from a smaller house

Friends & Strong Induction

Recursive Algorithm:

- Assume you have an algorithm that works.
- Use it to write an algorithm that works.



The "base case"

Use brute force
to get into
the smallest house.

Example

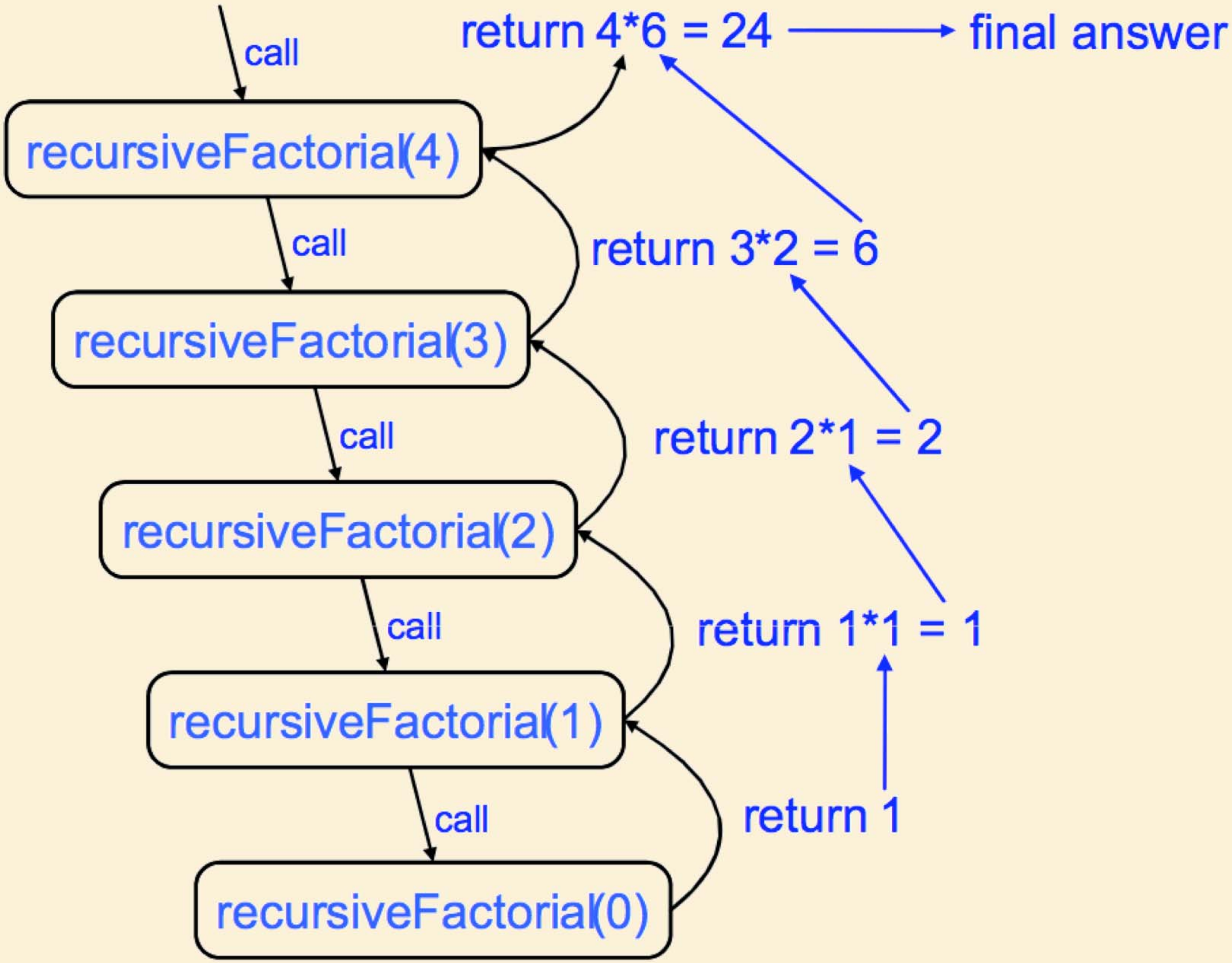
- The factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- As a Java method:

```
// recursive factorial function
public static int  recursiveFactorial(int n) {
   if  (n  ==  0)  return  1;      // base case
   else return  n  *  recursiveFactorial(n- 1); // recursive case
}
```

Tracing Recursion



Linear Recursion

- recursiveFactorial is an example of **linear** recursion: only one recursive call is made per stack frame.

```
// recursive factorial function
public static int  recursiveFactorial(int n) {
   if  (n  ==  0)  return  1;      // base case
   else return  n  *  recursiveFactorial(n- 1); // recursive case
}
```

Linear Recursion Design Pattern

- **Test for base cases**
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls **must** eventually reach a base case, and the handling of each base case should not use recursion.
- ***Recurse once***
  - Perform a single recursive call. (This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.)
  - Define each possible recursive call so that it makes **progress** towards a base case.

Another Example: Computing Powers

- The power function, **p(x,n) = x^n**, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- Assume multiplication takes constant time (independent of value of arguments).
- This leads to a power function that runs in O(n) time (for we make n recursive calls).
- We can do better than this, however.

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- For example,

$$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$

$$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$$

$$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$

$$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$$

A Recursive Squaring Method

Algorithm Power(x, n):

Input: A number x and integer n = 0

Output: The value x^n

if $n = 0$ **then**

return 1

if n is odd **then**

$y = \text{Power}(x, (n - 1)/ 2)$

return $x \cdot y \cdot y$

else

$y = \text{Power}(x, n/ 2)$

return $y \cdot y$

Analyzing the Recursive Squaring Method

Algorithm Power(x, n):

Input: A number x and integer n = 0

Output: The value x^n

if n = 0 **then**

  return 1

if n is odd **then**

  y = Power(x, (n - 1)/ 2)

  return $x \cdot y \cdot y$

else

  y = Power(x, n/ 2)

  return $y \cdot y$

Although there are 2 statements that recursively call Power, only one is executed per stack frame.

Each time we make a recursive call we halve the value of n (roughly).

Thus we make a total of log n recursive calls. That is, this method runs in O(log n) time.

The Greatest Common Divisor (GCD) Problem

- Given two integers, what is their greatest common divisor?
- e.g., gcd(56,24) = 8

Notation:

Given $d, a \in \mathbb{Z}$:

$d \mid a \leftrightarrow d$ divides $a \leftrightarrow \exists k \in \mathbb{Z} : a = kd$

Note: All integers divide 0: $d \mid 0 \forall d \in \mathbb{Z}$

Important Property:

$d \mid a$ and $d \mid b \rightarrow d \mid (ax + by) \forall x, y \in \mathbb{Z}$

Euclid's Trick

Euclid of Alexandria,
"The Father of Geometry"
c. 300 BC

Important Property:

$d \mid a$ and $d \mid b \rightarrow d \mid (ax + by) \forall x, y \in \mathbb{Z}$

Idea: Use this property to make the GCD problem easier!

Consequence: *e.g.*,

$\gcd(a,b) = \gcd(a-b,b) \longrightarrow \gcd(56,24) = \gcd(56-24,24) = \gcd(32,24)$ Good!

$\gcd(a,b) = \gcd(a-2b,b) \longrightarrow \gcd(56,24) = \gcd(56-2\times 24,24) = \gcd(8,24)$ Better!

$\gcd(a,b) = \gcd(a-3b,b) \longrightarrow \gcd(56,24) = \gcd(56-3\times 24,24) = \gcd(-16,24)$ Too Far!

$\vdots$

What is the optimal choice?

$\gcd(a,b) = \gcd(a \bmod b, b) \longrightarrow \gcd(56,24) = \gcd(56 \bmod 24, 24) = \gcd(8,24)$

Euclid's Algorithm (*circa* 300 BC)

Euclid(a,b)

<Precondition: *a* and *b* are positive integers>

<Postcondition: returns gcd(a,b)>

if $b = 0$ then

return(a)

else

return(Euclid($b, a \bmod b$))

Precondition met, since $a \bmod b \in \mathbb{Z}$

Postcondition met, since

1. $b = 0 \rightarrow \gcd(a,b) = \gcd(a,0) = a$
2. Otherwise, $\gcd(a,b) = \gcd(b, a \bmod b)$
3. Algorithm halts, since $0 \leq a \bmod b < b$

Time Complexity

Euclid(a,b)

  if $b = 0$ then

    return(a)

  else

    return(Euclid($b, a \bmod b$))

Claim: 2nd argument drops by factor of at least 2 every 2 iterations.

Proof:

Iteration	Arg 1	Arg 2
i	a	b
$i+1$	b	$a \bmod b$
$i+2$	$a \bmod b$	$b \bmod (a \bmod b)$

Case 1: $a \bmod b \le b/2$. Then $b \bmod (a \bmod b) < a \bmod b \le b/2$ ✓

Case 2: $b > a \bmod b > b/2$. Then $b \bmod (a \bmod b) < b/2$ ✓

Time Complexity

Euclid(a,b)
  if $b = 0$ then
    return(a)
  else
    return(Euclid($b, a \bmod b$))

Let k = total number of recursive calls to Euclid.

Let n = input size $\simeq$ number of bits used to represent a and b.

Then $2^{k/2} \simeq b \simeq 2^{n/2} \to k \simeq n$.

Each stackframe must compute $a \bmod b$, which takes more than constant time.

It can be shown that the resulting time complexity is $T(n) \in O(n^2)$.

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its **last** step.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Examples
  – Euclid's GCD algorithm
  – Reversing an array

Example: Recursively Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ **then**

Swap $A[i]$ and $A[j]$

ReverseArray(A, $i + 1$, $j - 1$)

return

Example: Iteratively Reversing an Array

Algorithm IterativeReverseArray(*A*, *i*, *j*):

Input: An array *A* and nonnegative integer indices *i* and *j*

Output: The reversal of the elements in *A* starting at index *i* and ending at *j*

while $i < j$ **do**

Swap $A[i]$ and $A[j]$

$i = i + 1$

$j = j - 1$

return

Defining Arguments for Recursion

- Solving a problem recursively sometimes requires passing additional parameters.
- **ReverseArray** is a good example: although we might initially think of passing only the array ***A*** as a parameter at the top level, lower levels need to know where in the array they are operating.
- Thus the recursive interface is **ReverseArray(A, i, j)**.
- We then invoke the method at the highest level with the message **ReverseArray(A, 1, n)**.

Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.
- Example 1: **The Fibonacci Sequence**

The Fibonacci Sequence

- Fibonacci numbers are defined recursively:

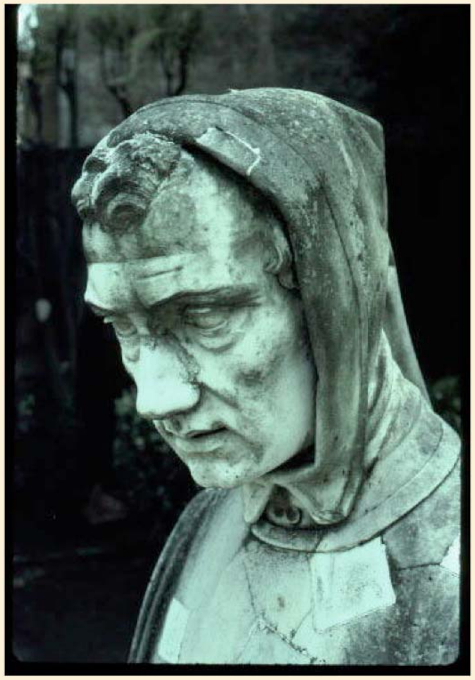
$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

The ratio F_i / F_{i-1} converges to $\varphi = \frac{1+\sqrt{5}}{2} = 1.61803398874989...$

(The "**Golden Ratio**")

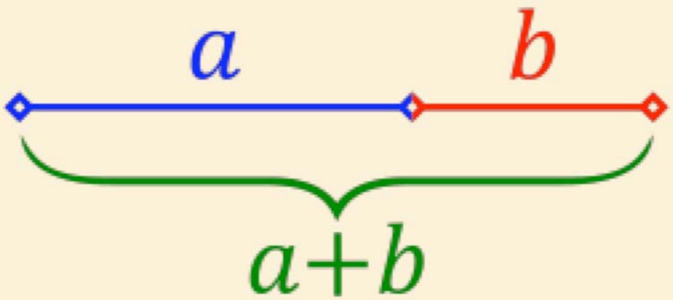


Fibonacci (c. 1170 - c. 1250)
(aka Leonardo of Pisa)

The Golden Ratio

- Two quantities are in the **golden ratio** if the ratio of the sum of the quantities to the larger quantity is equal to the ratio of the larger quantity to the smaller one.

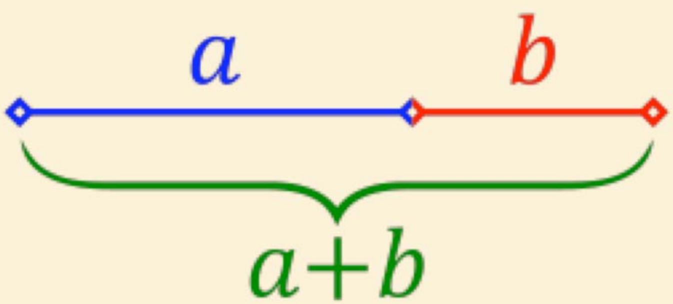
φ is the unique positive solution to $\varphi = \dfrac{a+b}{a} = \dfrac{a}{b}$.



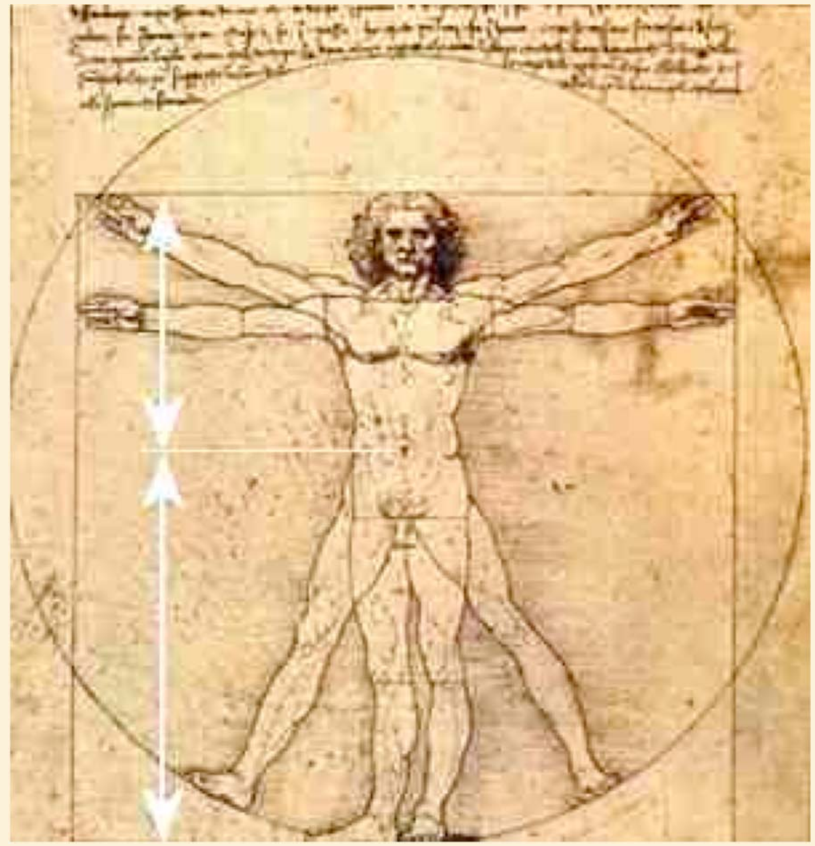
$a+b$ is to a as a is to b

The Golden Ratio

The Parthenon



$a+b$ is to a as a is to b



Leonardo

Computing Fibonacci Numbers

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- A recursive algorithm (first attempt):

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The kth Fibonacci number F_k

if k = 1 **then**

  return k

else

  return BinaryFib(k - 1) + BinaryFib(k - 2)

Analyzing the Binary Recursion Fibonacci Algorithm

- Let n_k denote number of recursive calls made by BinaryFib(k). Then
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67.$
- Note that n_k more than doubles for every other value of n_k. That is, $n_k > 2^{k/2}$. It increases exponentially!

A Better Fibonacci Algorithm

- Use **linear** recursion instead:

Algorithm LinearFibonacci(k):

Input: A nonnegative integer k

Output: Pair of Fibonacci numbers (F_k, F_{k-1})

if $k = 1$ **then**

    return $(k, 0)$

else

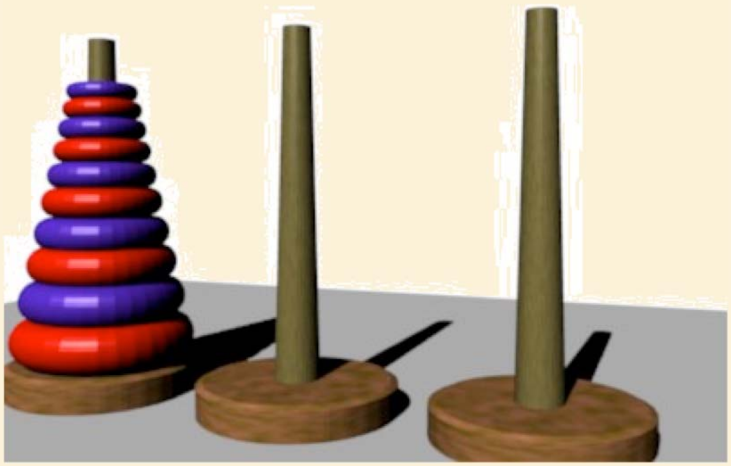
    $(i, j) = \text{LinearFibonacci}(k - 1)$

    return $(i + j, i)$

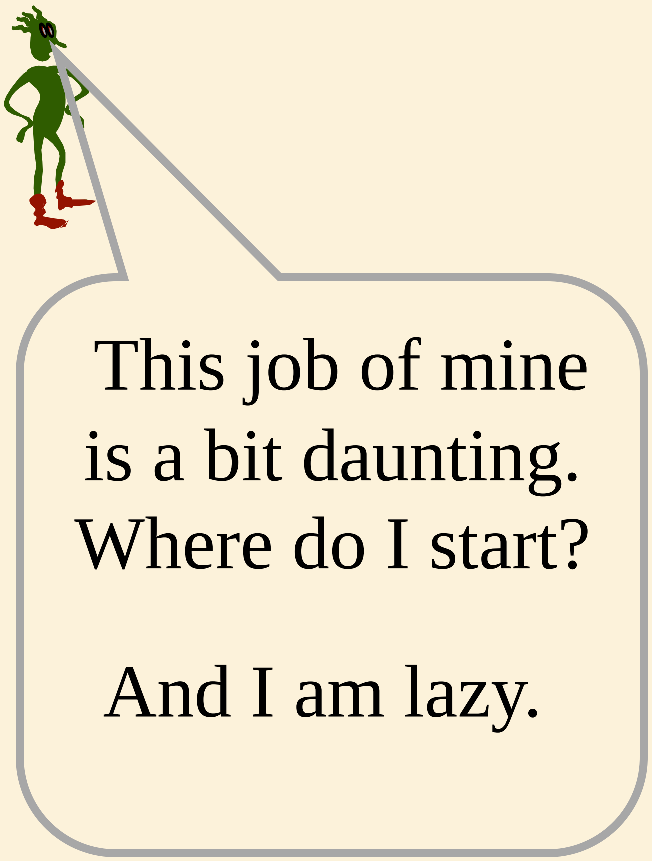
- Runs in ***O*(*k*)** time.

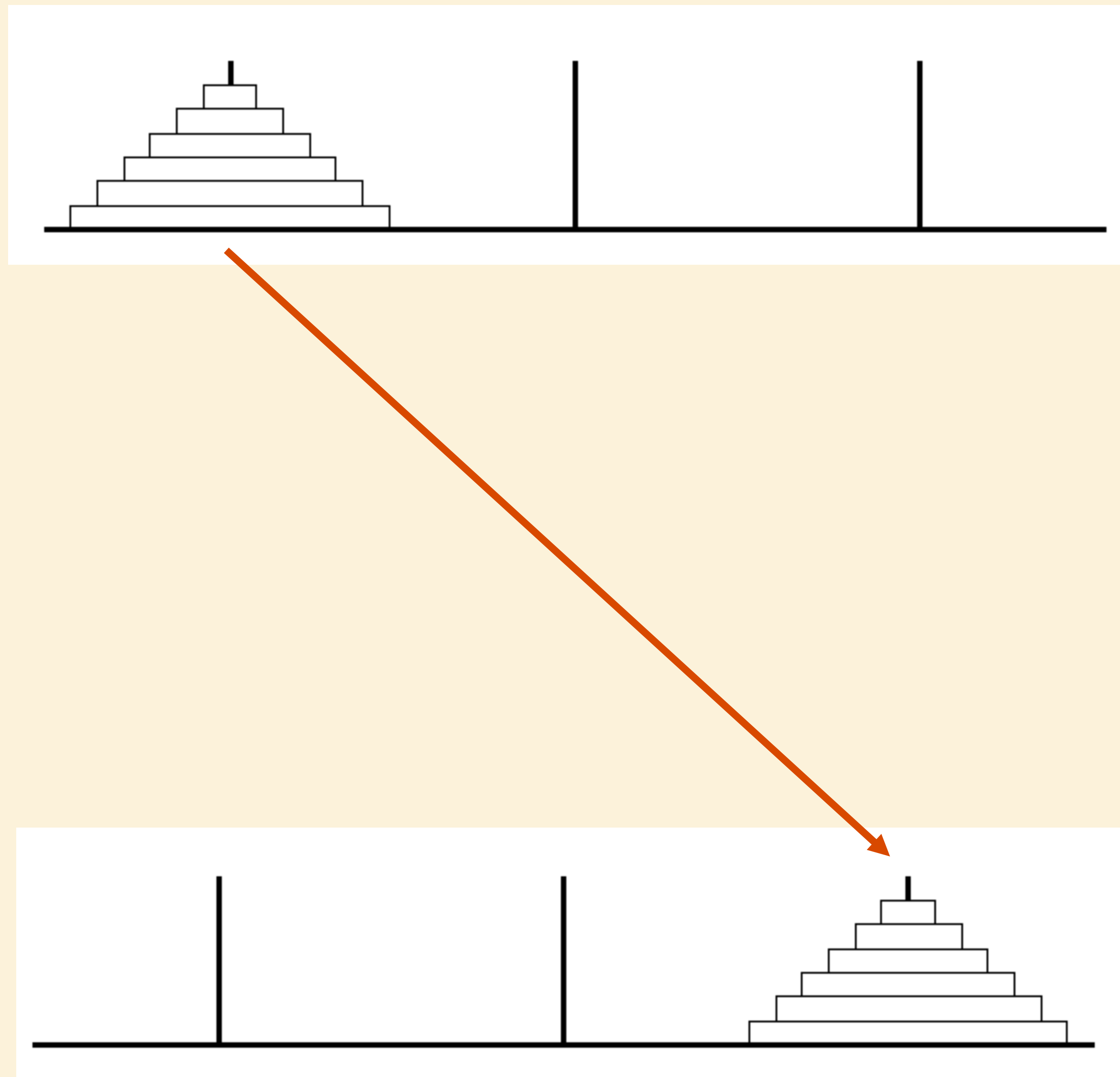
Binary Recursion

- Second Example: **The Tower of Hanoi**

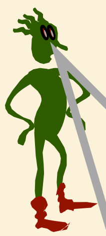


Tower of Hanoi

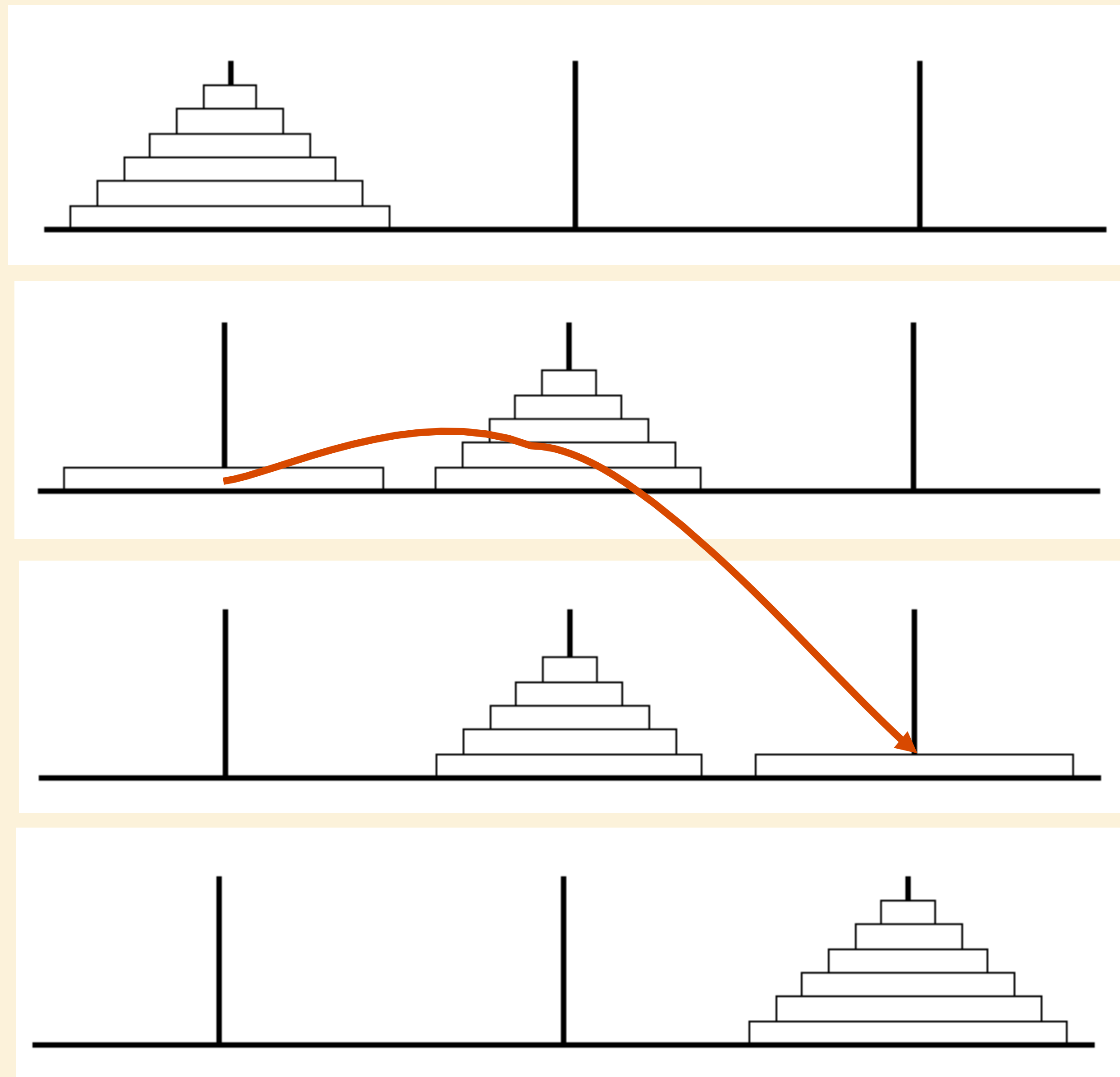




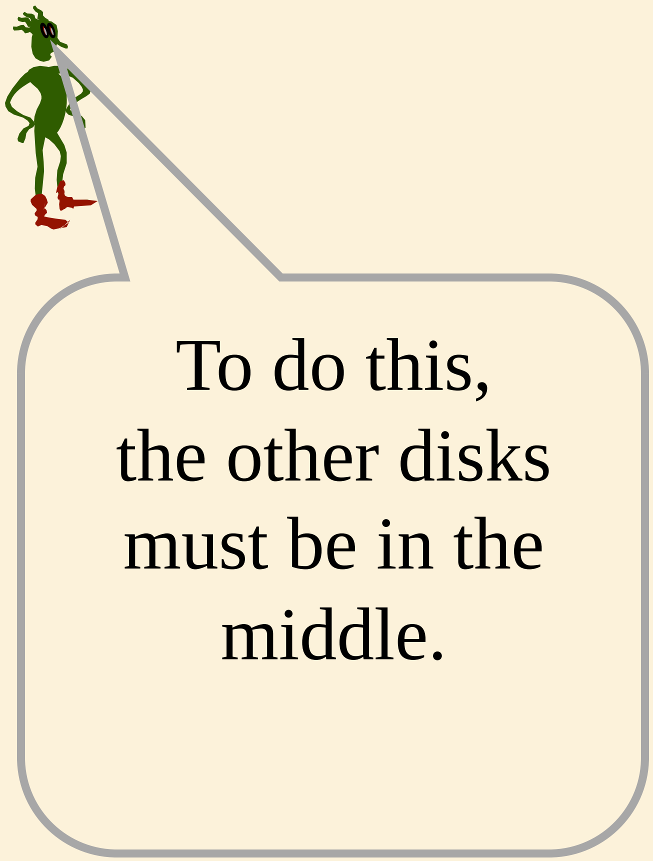
Tower of Hanoi



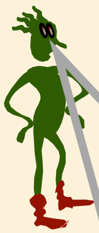
At some point, the biggest disk moves.
I will do that job.



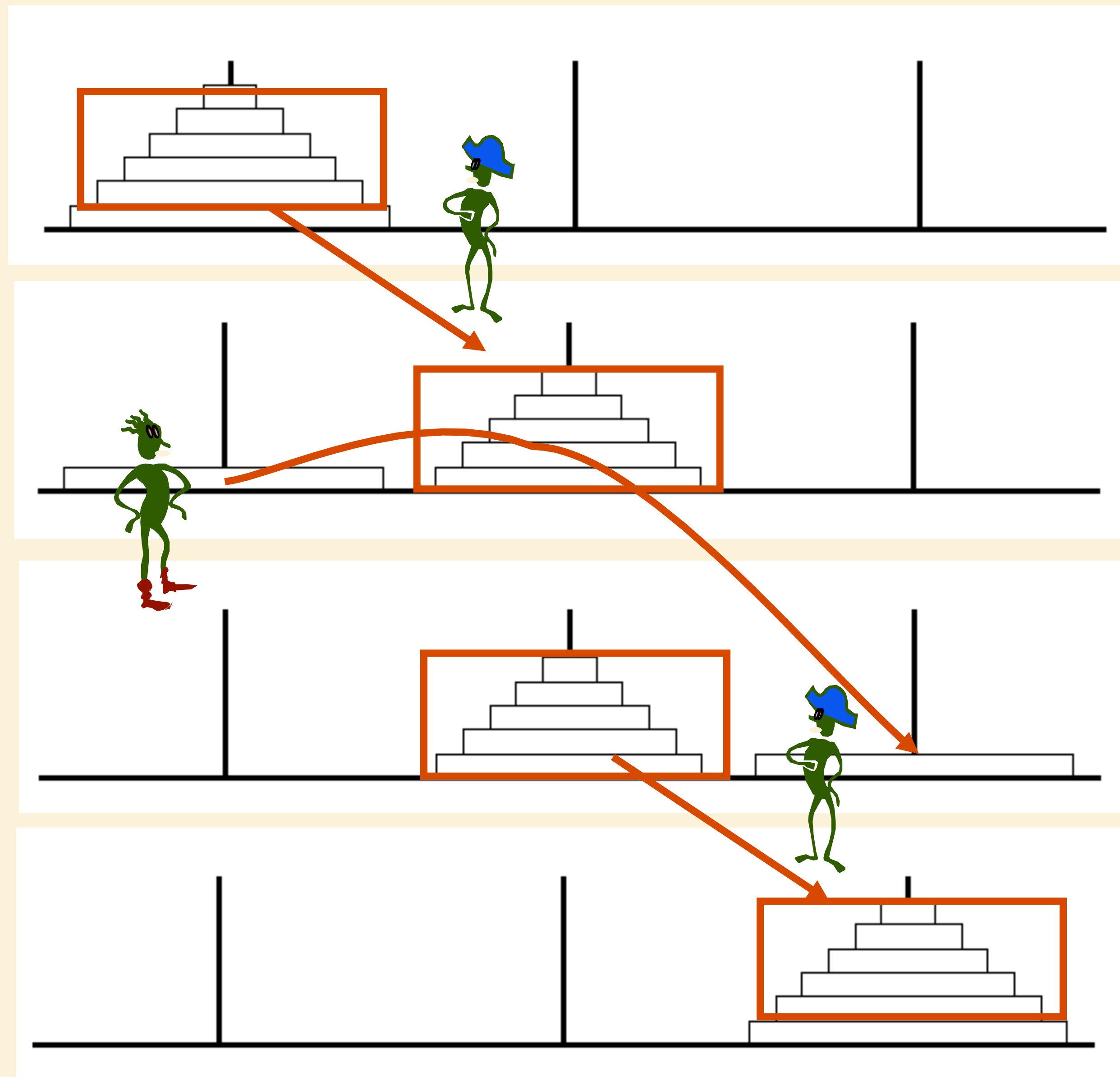
Tower of Hanoi



Tower of Hanoi



How will these move?
I will get a friend to do it.
And another to move these.
I only move the big disk.



Tower of Hanoi

Code:

algorithm $TowersOfHanoi(n, source, destination, spare)$

$\langle \boldsymbol{pre-cond} \rangle$: The n smallest disks are on $pole_{source}$.

$\langle \boldsymbol{post-cond} \rangle$: They are moved to $pole_{destination}$.

```
begin
    if(n = 1)
        Move the single disk from pole_source to pole_destination.
    else
        TowersOfHanoi(n − 1, source, spare, destination)
        Move the n^th disk from pole_source to pole_destination.
        TowersOfHanoi(n − 1, spare, destination, source)
    end if
end algorithm
```

2 recursive calls!

Tower of Hanoi

Code:

algorithm $TowersOfHanoi(n, source, destination, spare)$

$\langle \boldsymbol{pre-cond} \rangle$: The n smallest disks are on $pole_{source}$.

$\langle \boldsymbol{post-cond} \rangle$: They are moved to $pole_{destination}$.

```
begin
    if(n = 1)
        Move the single disk from pole_source to pole_destination.
    else
        TowersOfHanoi(n − 1, source, spare, destination)
        Move the n^th disk from pole_source to pole_destination.
        TowersOfHanoi(n − 1, spare, destination, source)
    end if
end algorithm
```

Time:

$T(1) = 1,$

$T(n) = 1 + 2T(n-1) \approx 2T(n-1)$

$\approx 2(2T(n-2)) \approx 4T(n-2)$

$\approx 4(2T(n-3)) \approx 8T(n-3)$

$\approx 2^i T(n-i)$

$\approx 2^n$

The Cost of Recursion

- Many problems are naturally defined recursively.
- This can lead to simple, elegant code.
- However, recursive solutions entail a **cost in time and memory**: each recursive call requires that the current process state (variables, program counter) be **pushed** onto the system stack, and **popped** once the recursion unwinds.
- This typically affects the running time **constants**, but **not** the **asymptotic time complexity** (e.g., $O(n)$, $O(n^2)$ etc.)
- Thus **recursive solutions may still be preferred** unless there are very strict time/memory constraints.

The “Curse” in Recursion: Errors to Avoid

```
// recursive factorial function
public static int  recursiveFactorial(int n) {
   return  n  *  recursiveFactorial(n- 1);
}
```

- **There must be a base condition: the recursion must ground out!**

The "Curse" in Recursion: Errors to Avoid

```
// recursive factorial function
public static int  recursiveFactorial(int n) {
    if  (n  ==  0)  return  recursiveFactorial(n); // base case
    else return  n  *  recursiveFactorial(n- 1); // recursive case
}
```

- **The base condition must not involve more recursion!**

The “Curse” in Recursion: Errors to Avoid

```
// recursive factorial function
public static int  recursiveFactorial(int n) {
   if  (n  ==  0)  return  1;      // base case
   else return  (n – 1)  *  recursiveFactorial(n);         // recursive case
}
```

- The input **must be converging** toward the base condition!