Recursion

Outline

- Induction
- Linear recursion
  - Example 1: Factorials
  - Example 2: Powers
  - Example 3: Reversing an array
- Binary recursion
  - Example 1: The Fibonacci sequence
  - Example 2: The Tower of Hanoi
- Drawbacks and pitfalls of recursion

Outcomes

- By understanding this lecture you should be able to:
  - Use induction to prove the correctness of a recursive algorithm.
  - Identify the base case for an inductive solution
  - Design and analyze linear and binary recursion algorithms
  - Identify the overhead costs of recursion
  - Avoid errors commonly made in writing recursive algorithms

Outline

- **Induction**
- Linear recursion
  - Example 1: Factorials
  - Example 2: Powers
  - Example 3: Reversing an array
- Binary recursion
  - Example 1: The Fibonacci sequence
  - Example 2: The Tower of Hanoi
- Drawbacks and pitfalls of recursion

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 uses induction to do this.

History of Induction

- Implicit use of induction goes back at least to Euclid's proof that the number of primes is infinite (c. 300 BC).
- The first explicit formulation of the principle is due to Pascal (1665).

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

Blaise Pascal, 1623 - 1662

Induction: Review

- Induction is a mathematical method for proving that a statement is true for a (possibly infinite) sequence of objects.
- There are two things that must be proved:
  1. **The Base Case**: The statement is true for the first object
  2. **The Inductive Step**: If the statement is true for a given object, it is also true for the next object.
- If these two statements hold, then the statement holds for all objects.

Induction Example: An Arithmetic Sum

- Claim: $\sum_{i=0}^{n} i = \frac{1}{2} n(n+1) \quad \forall n \in \mathbb{N}$
- Proof:

1. **Base Case.** The statement holds for $n = 0$:

$$\sum_{i=0}^{n} i = \sum_{i=0}^{0} i = 0$$

$$\frac{1}{2} n(n+1) = \frac{1}{2} 0(0+1) = 0 \qquad \checkmark$$

1. **Inductive Step.** If the claim holds for $n = k$, then it also holds for $n = k$+1.

$$\sum_{i=0}^{k+1} i = k+1+\sum_{i=0}^{k} i = k+1+\frac{1}{2}k(k+1) = \frac{1}{2}(k+1)(k+2) \qquad \checkmark$$

Recursive Divide and Conquer

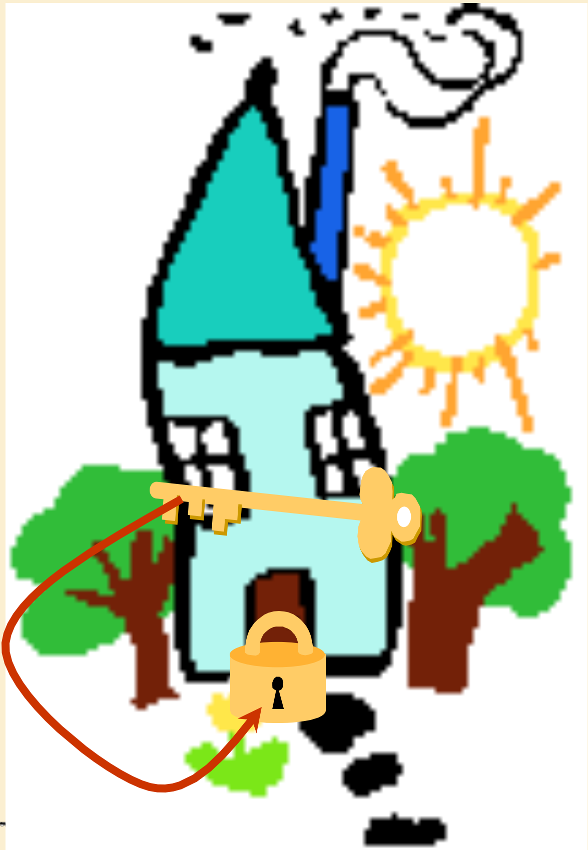
- You are given a problem input that is too big to solve directly.
- You imagine,
  - "Suppose I had a friend who could give me the answer to the same problem with slightly smaller input."
  - "Then I could easily solve the larger problem."
- In recursion this "friend" will actually be another instance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

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

Friends & 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.

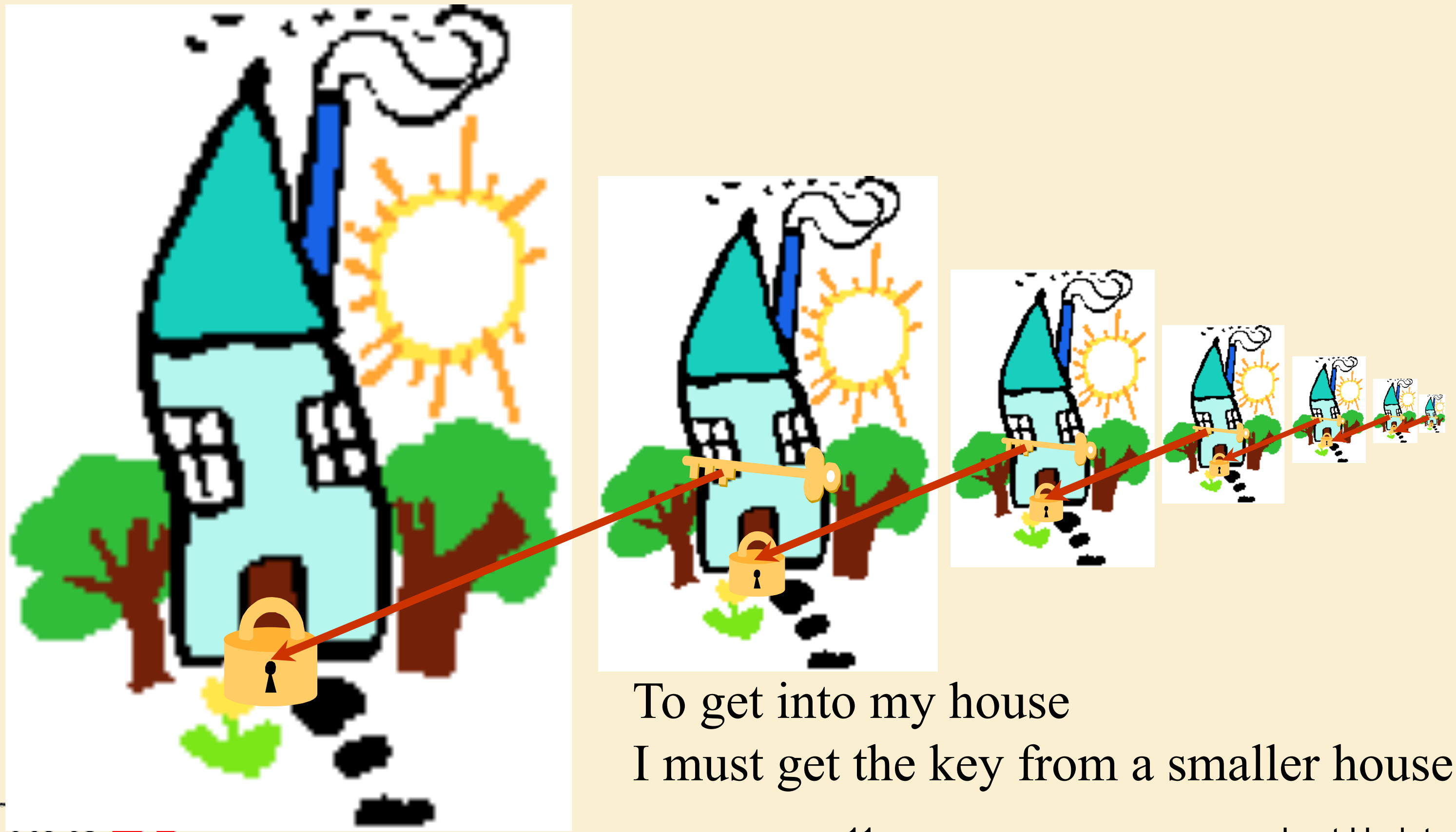
Circular Argument!

Example from J. Edmonds – Thanks Jeff!

Friends & Induction

Recursive Algorithm:

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

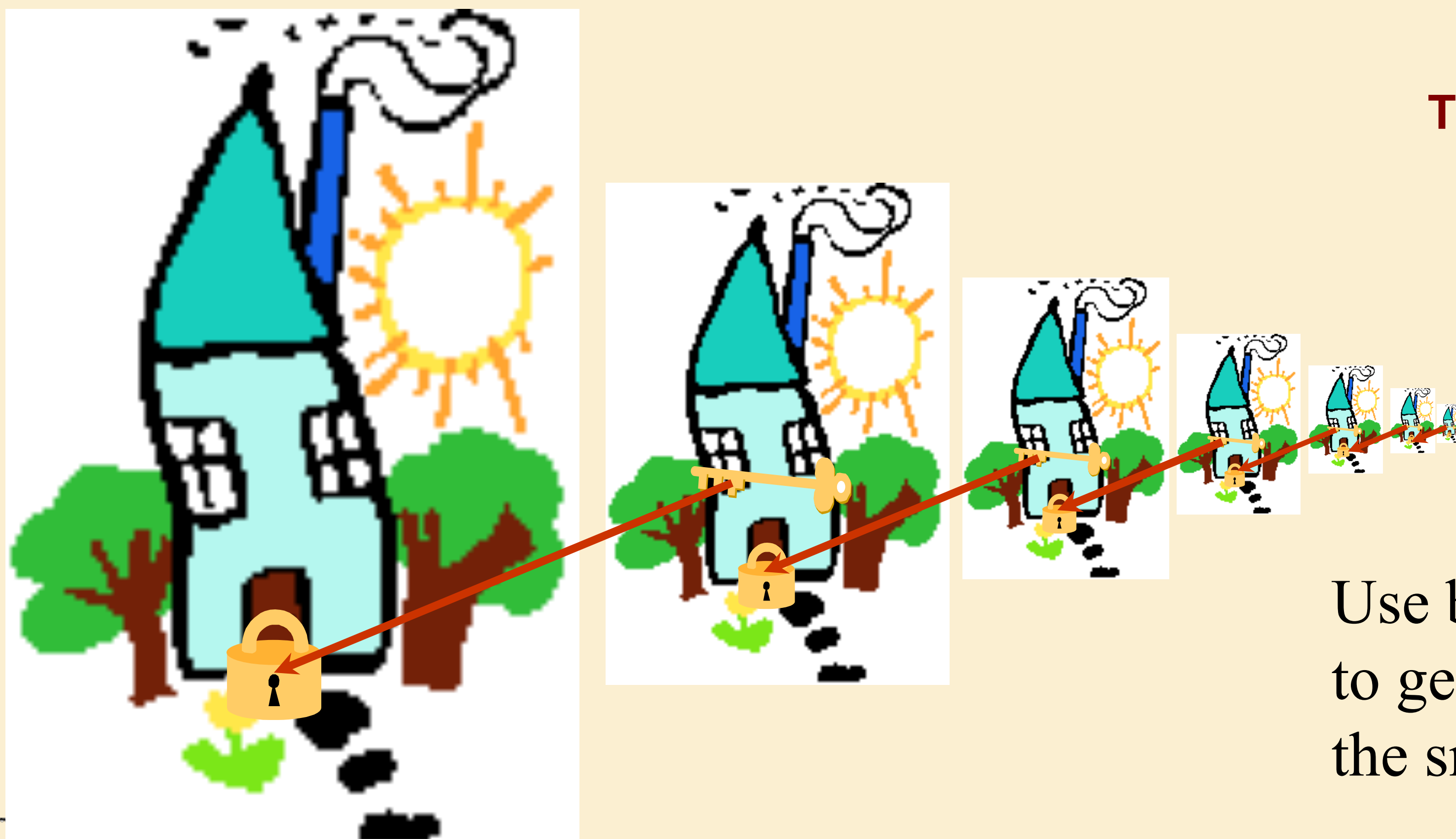


To get into my house
I must get the key from a smaller house

Friends & 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.

Outline

- Induction
- **Linear recursion**
  - Example 1: Factorials
  - Example 2: Powers
  - Example 3: Reversing an array
- Binary recursion
  - Example 1: The Fibonacci sequence
  - Example 2: The Tower of Hanoi
- Drawbacks and pitfalls of recursion

Recall: Design Pattern

- A template for a software solution that can be applied to a variety of situations.
- Main elements of solution are described in the abstract.
- Can be specialized to meet specific circumstances.

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.

Example 1

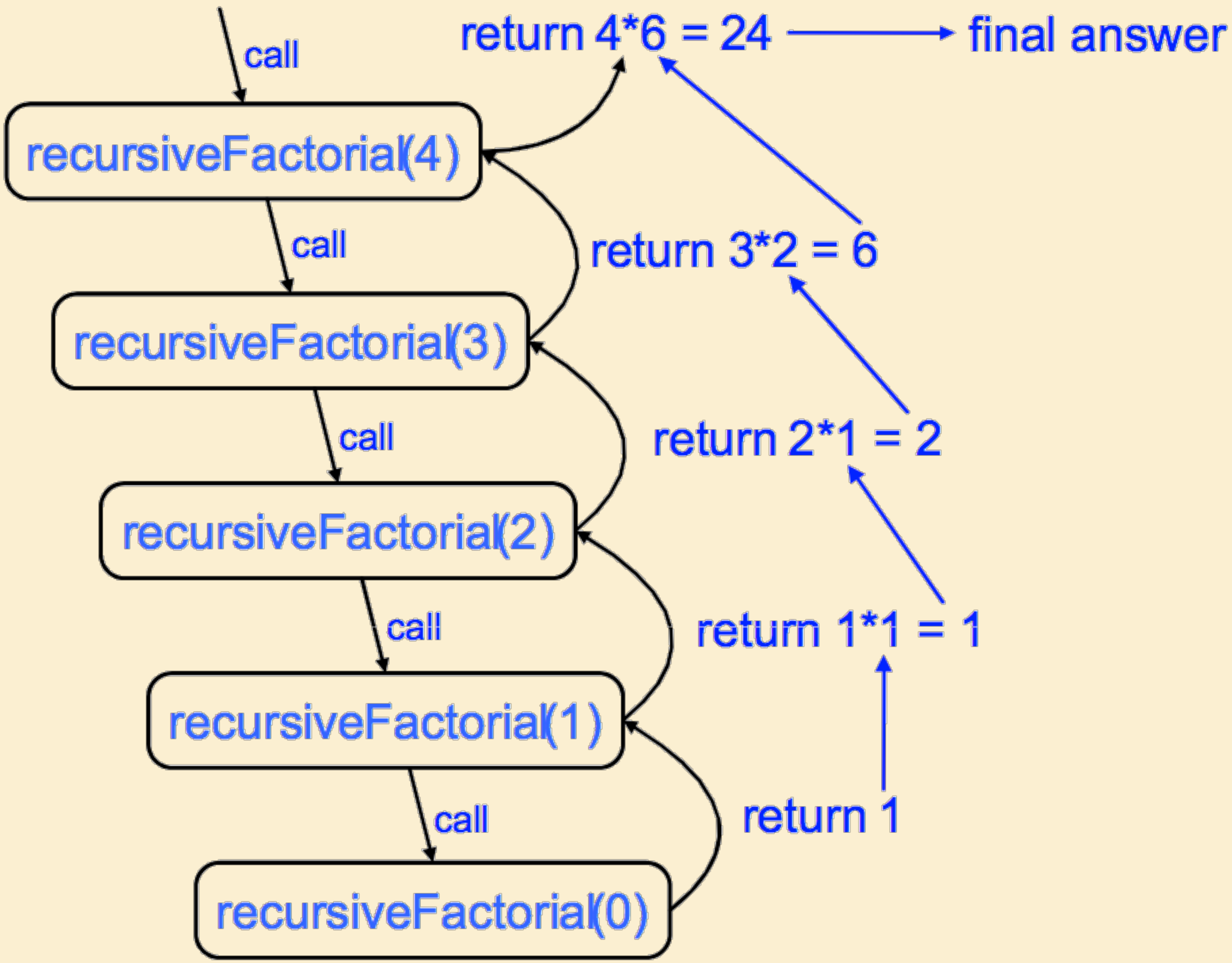
- The factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

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

- 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.
- Since there are *n* recursive calls, this algorithm has O(*n*) run time.

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

Example 2: Computing Powers

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

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

- 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).
- **Can we do better than this?**

End of Lecture

Jan 29, 2018

Recursive Squaring

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

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

Example: p(2,8)

Method 1

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

p(2,8)

= 2 x p(2,7)

= 2 x 2 x p(2,6)

= 2 x 2 x 2 x p(2,5)

= 2 x 2 x 2 x 2 x p(2,4)

= 2 x 2 x 2 x 2 x 2 x p(2,3)

= 2 x 2 x 2 x 2 x 2 x 2 x p(2,2)

= 2 x 2 x 2 x 2 x 2 x 2 x 2 x p(2,1)

= 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 x p(2,0)

= 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 x 1 ← **8 multiplies**

Method 2

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

p(2,8)

= $p(2,4)^2$

= $(p(2,2)^2)^2$

= $((p(2,1)^2)^2)^2$

= $(((2 \times p(2,0)^2)^2)^2)^2$

= $(((2 \times 1^2)^2)^2)^2$ ← **5 multiplies**

A Recursive Squaring Method

Algorithm Power(x, n):

Input: A number x and integer n

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**

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

  return $x \cdot y \cdot y$

else

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

  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.

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its **last** step.
- Such a method can easily be converted to an iterative method (which saves on some resources).

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, 0, n)**.

Outline

- Induction
- Linear recursion
  - Example 1: Factorials
  - Example 2: Powers
  - Example 3: Reversing an array
- **Binary recursion**
  - Example 1: The Fibonacci sequence
  - Example 2: The Tower of Hanoi
- Drawbacks and pitfalls of recursion

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:

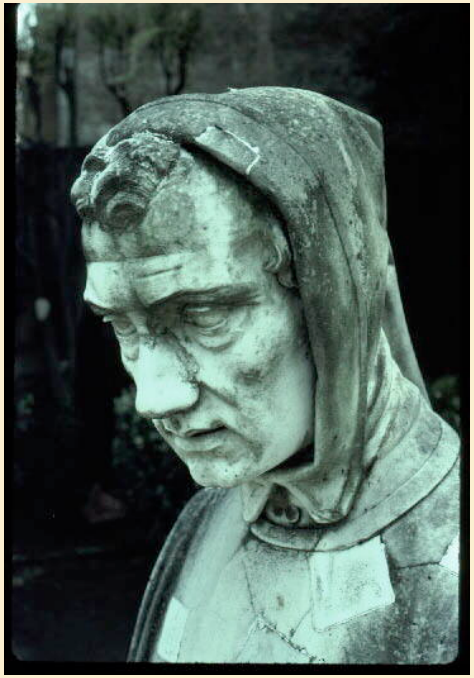
$F_0 = 0$

$F_1 = 1$

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

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

(The "**Golden Ratio**")

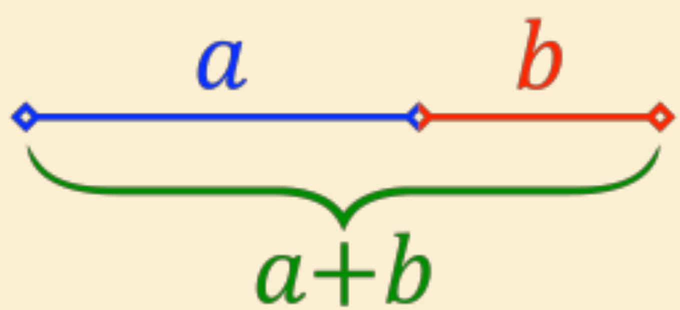


Fibonacci (c. 1170 - c. 1250)
(aka Leonardo of Pisa)

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.

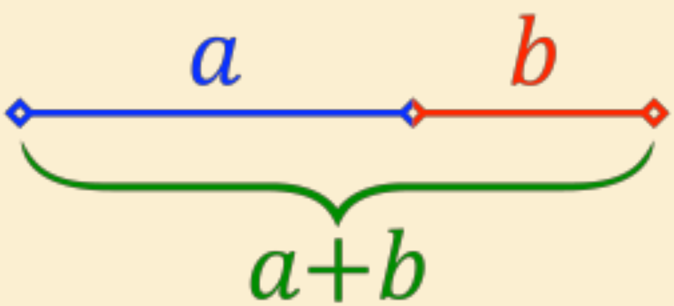
φ is the unique positive solution to $\varphi = \dfrac{a+b}{a} = \dfrac{a}{b}$.



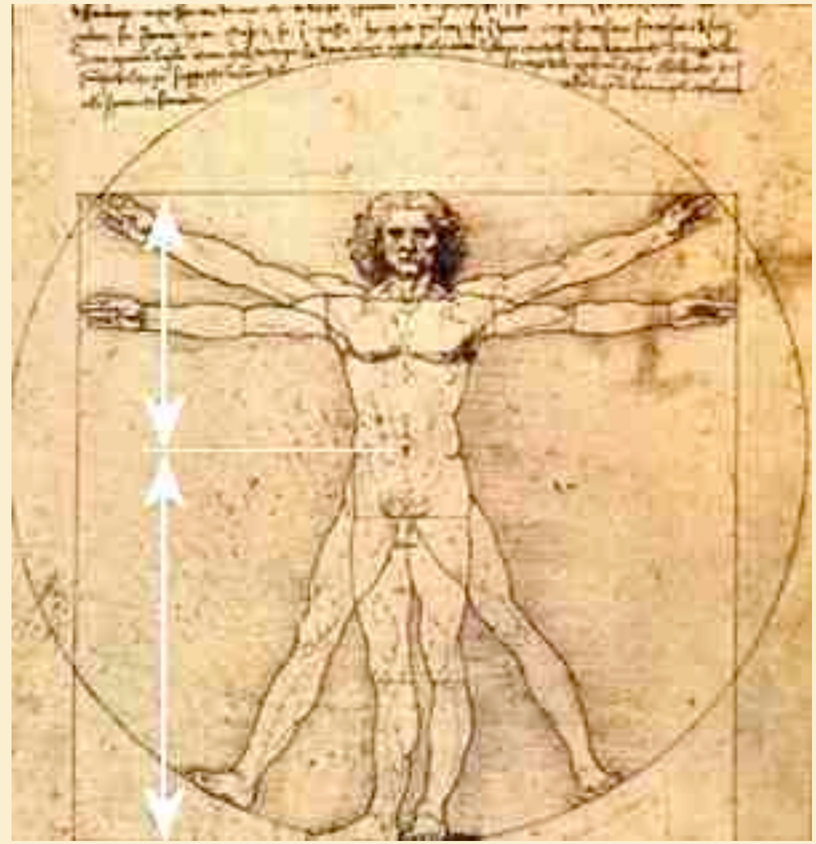
$a+b$ is to a as a is to b

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: Positive integer k

Output: The kth Fibonacci number F_k

if $k < 2$ **then**

return k

else

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

Analyzing the Binary Recursion Fibonacci Algorithm

Algorithm BinaryFib(k):

Input: Positive integer k

Output: The kth Fibonacci number F_k

if $k < 2$ **then**

return k

else

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

- 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 positive integer k

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

if $k = 1$ **then**

 return $(k, 0)$

else

 (i, j) = LinearFibonacci(k - 1)

 return $(i + j, i)$

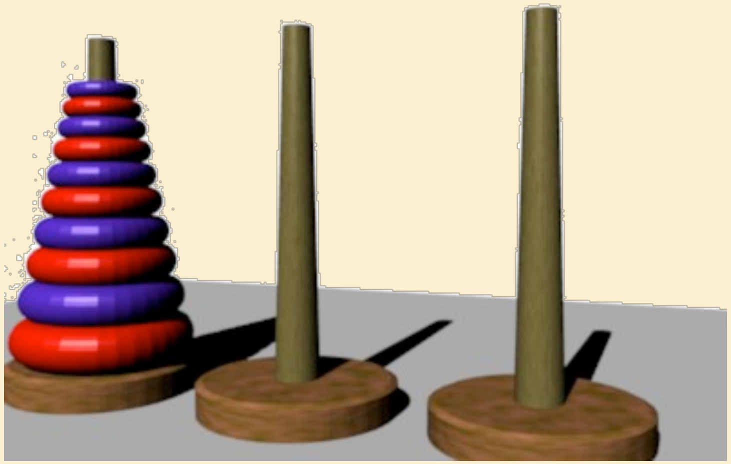
LinearFibonacci(k): F_k, F_{k-1}

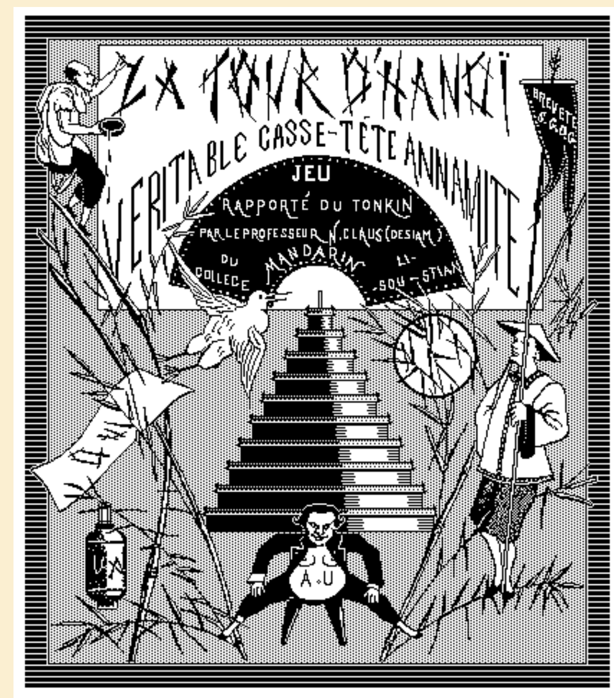
LinearFibonacci(k-1): F_{k-1}, F_{k-2}

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

Binary Recursion

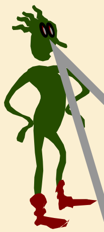
- Second Example: **The Tower of Hanoi**





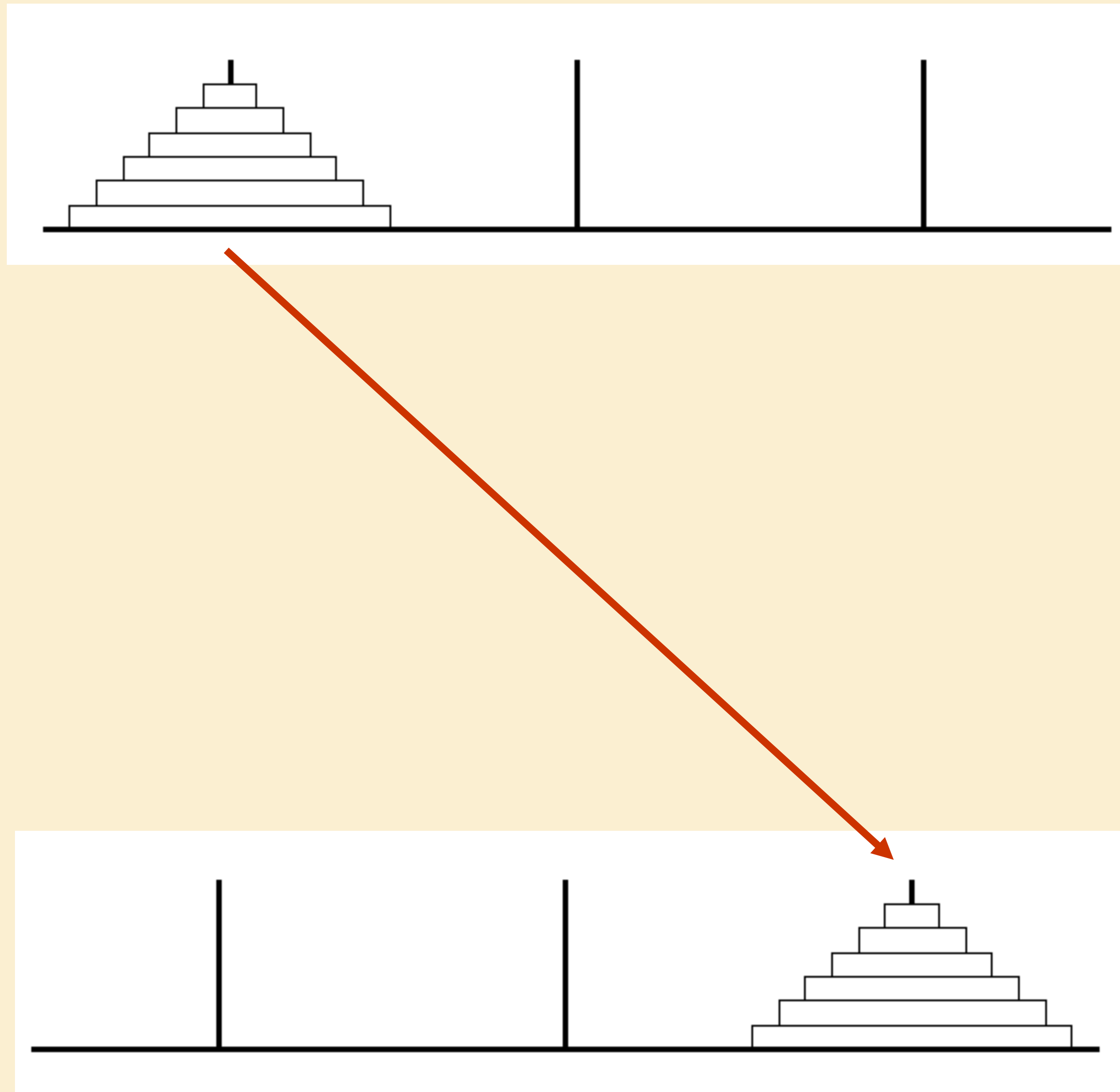
Example

Tower of Hanoi



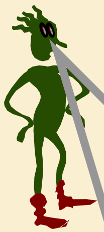
This job of mine is a bit daunting. Where do I start?

And I am lazy.

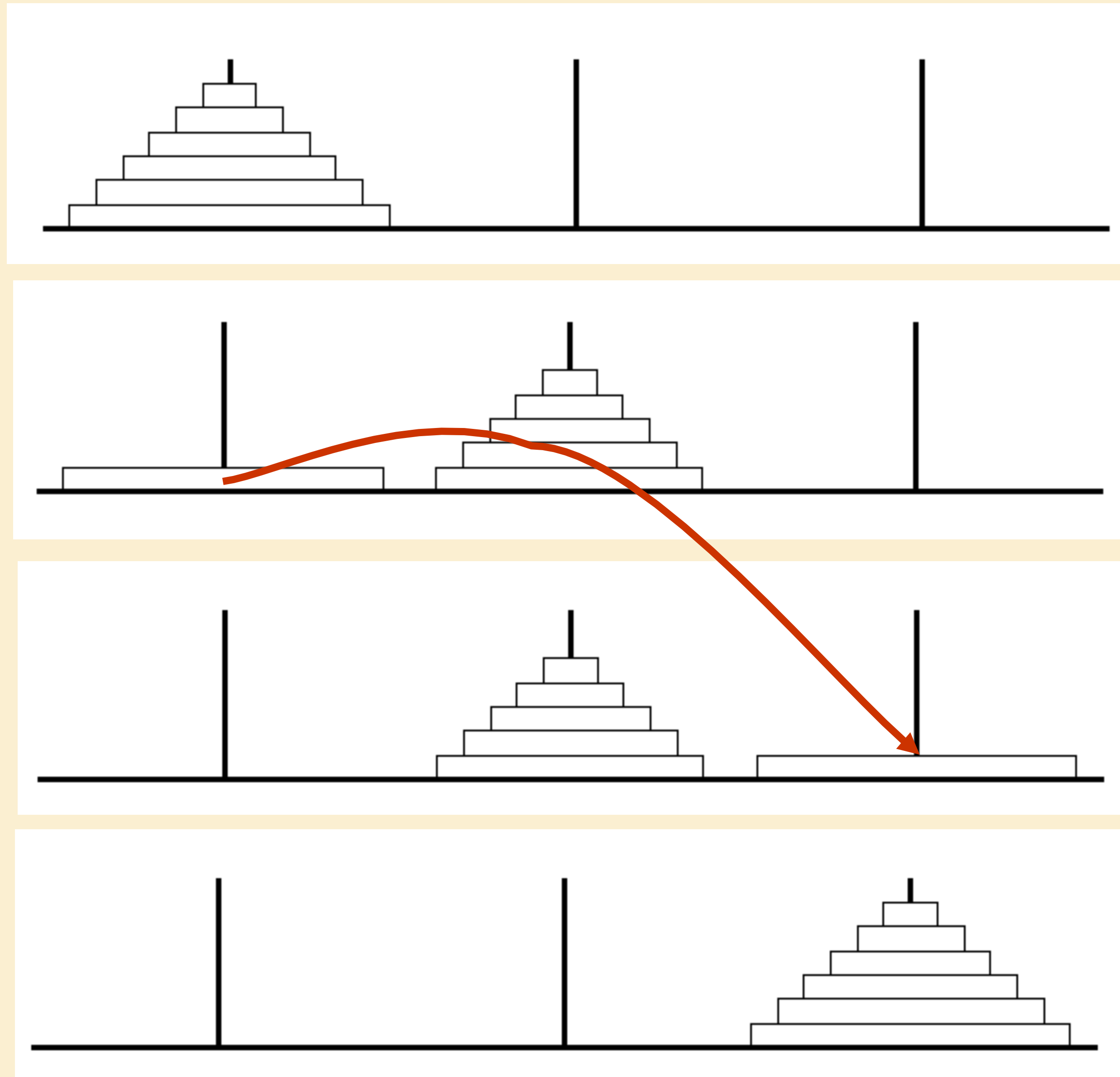


Example from J. Edmonds – Thanks Jeff!

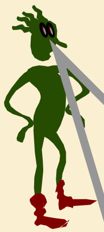
Tower of Hanoi



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

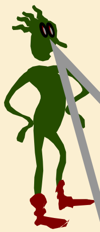


Tower of Hanoi

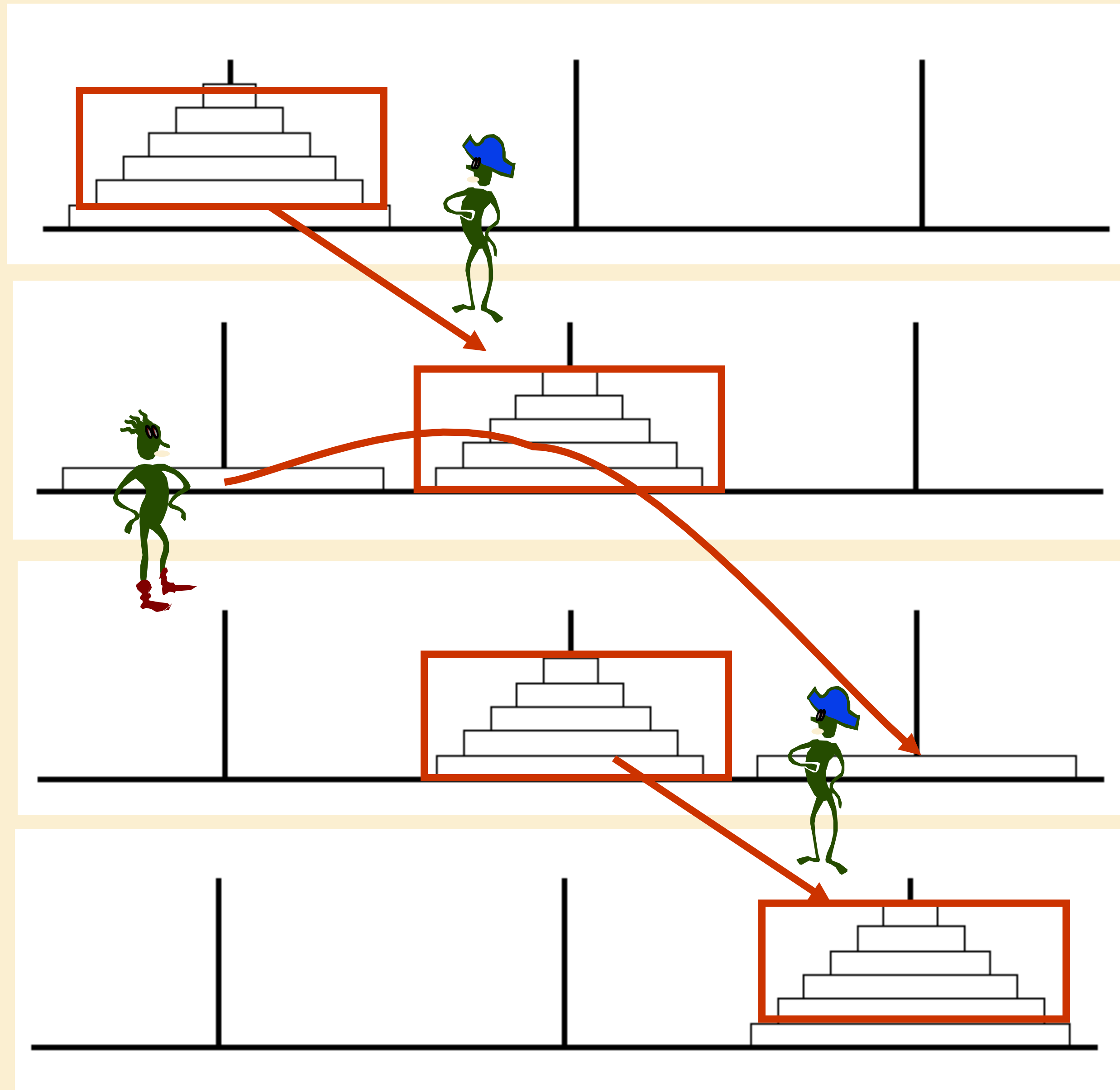


To do this, the other disks must be in the middle.

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:

$T(1) = 1,$

$$
\begin{aligned}
T(n) &= 1 + 2T(n-1) &&\approx 2T(n-1) \\
&\approx 2(2T(n-2)) &&\approx 4T(n-2) \\
&\approx 4(2T(n-3)) &&\approx 8T(n-3) \\
& &&\approx 2^i\,T(n-i) \\
& &&\approx 2^n
\end{aligned}
$$

Exponential again!!

Binary Recursion: Summary

- Binary recursion spawns an exponential number of recursive calls.
- If the inputs are only declining **arithmetically** (e.g., n-1, n-2,…) the result will be an exponential running time.
- In order to use binary recursion, the input must be declining **geometrically** (e.g., n/2, n/4, …).
- We will see efficient examples of binary recursion with geometricaly declining inputs when we discuss **heaps** and **sorting**.

Outline

- Induction
- Linear recursion
  - Example 1: Factorials
  - Example 2: Powers
  - Example 3: Reversing an array
- Binary recursion
  - Example 1: The Fibonacci sequence
  - Example 2: The Tower of Hanoi
- **Drawbacks and pitfalls of recursion**

The Overhead Costs 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!

Outline

- Induction
- Linear recursion
  - Example 1: Factorials
  - Example 2: Powers
  - Example 3: Reversing an array
- Binary recursion
  - Example 1: The Fibonacci sequence
  - Example 2: The Tower of Hanoi
- Drawbacks and pitfalls of recursion

Outcomes

- By understanding this lecture you should be able to:
  - Use induction to prove the correctness of a recursive algorithm.
  - Identify the base case for an inductive solution
  - Design and analyze linear and binary recursion algorithms
  - Identify the overhead costs of recursion
  - Avoid errors commonly made in writing recursive algorithms