COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from "Java Programming: Program Design Including Data Structures"
by Chris Kiekintveld

Recursion

Recursion

- Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first
- Recursion is a technique that solves a problem by solving a smaller problem of the same type
- A procedure that is defined in terms of itself

Recursion

- When you turn that into a program in computer, you end up with functions that call themselves:

  Recursive Functions

Review: Calling Methods

```
int x(int n) {
  int m = 0;
  n = n + m + 1;
  return n;
}

int y(int n) {
  int m = 1;
  n = x(n);
  return m + n;
}
```

What does y(3) return?

Calling Methods

- Methods can call other methods
- Can a method call itself?
- Yes! This is called a recursive method (function)

Example

Trace the execution of test(4)

```
void test(int n) {
    if (n > 0) {
        System.out.println(n);
        test(n-1);
        System.out.println(n);
    }
}
```

Example

What's behind this function ?

```
public int f(int a){
  if (a==0 || a==1)
    return(1);
  else
    return(a * f(a-1));
}
```

Example

What’s behind this function ?

```
public int f(int a){
  if (a==0 || a==1)
    return(1);
  else
    return(a * f(a-1));
}
```

It computes a! (factorial)

Factorial

Factorial:

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

Note:

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

remember:

...splitting up the problem into a smaller problem of the same type...

$$a!$$

$$a \quad * \quad (a-1)!$$

Tracing the example

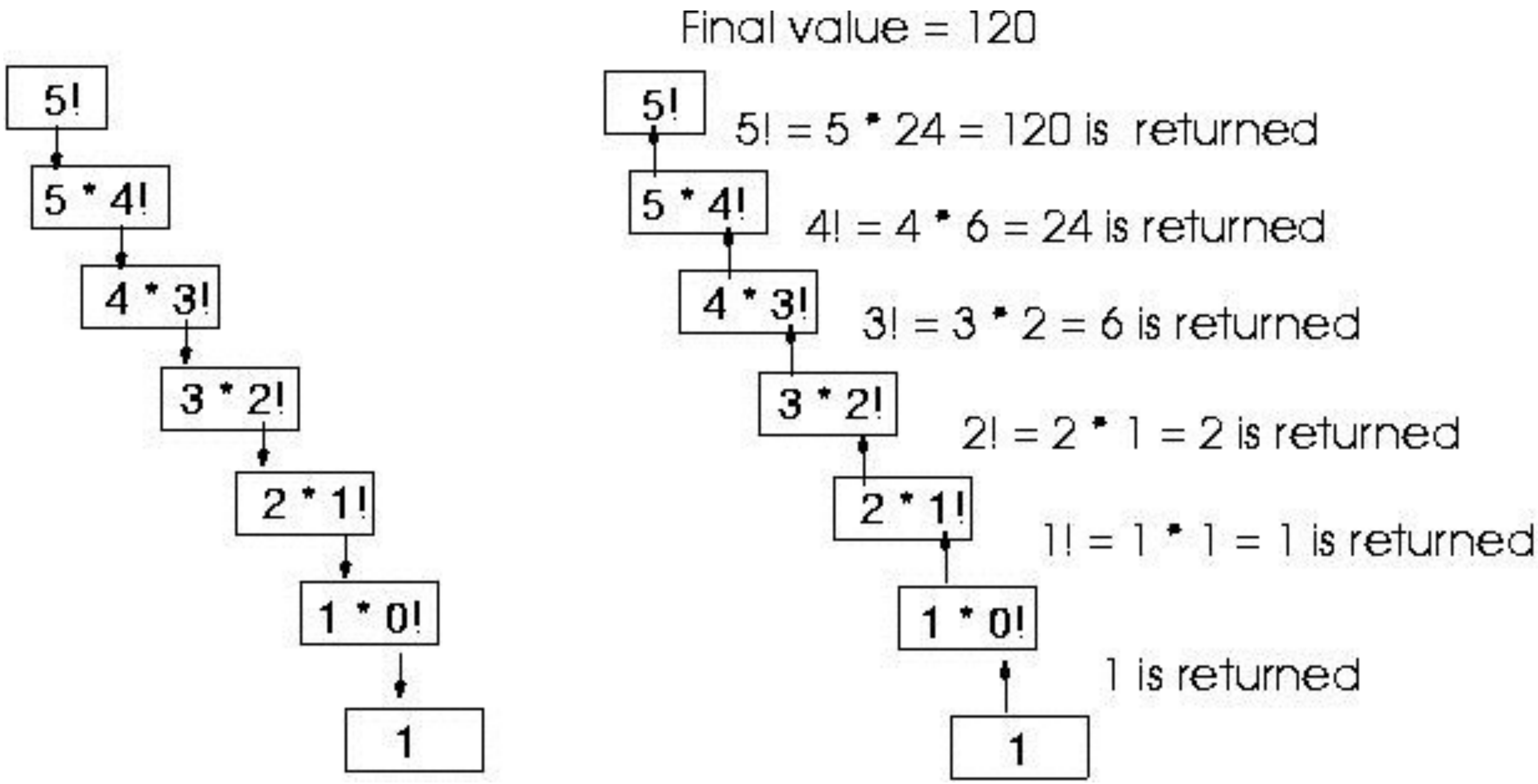
```
public int factorial(int a){
        if (a==0 || a==1)
          return(1);
        else
          return(a * factorial(a-1));
}
```

RECURSION !

Tracing the example

```
public int factorial(int a){
        if (a==0 || a==1)
            return(1);
        else
            return(a * factorial(a-1));
}
```

RECURSION !



Tracing a Recursive Method

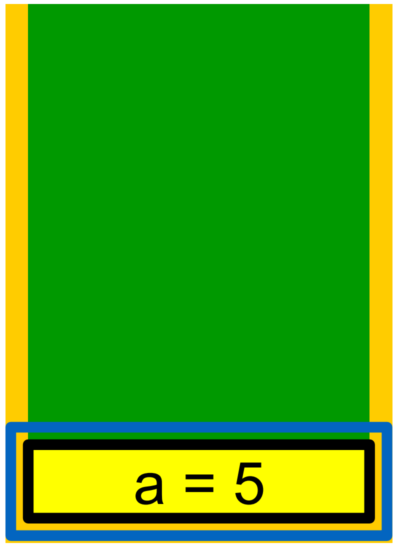
- As always, go line by line
- Recursive methods may have ***many copies***
- Every method call creates a new copy and transfers flow of control to the new copy
- Each copy has its own:
  - code
  - parameters
  - local variables

Tracing a Recursive Method

- After completing a recursive call:
  - Control goes back to the calling environment
  - Recursive call must execute completely before control goes back to the previous call
  - Execution in previous calls begins from point immediately following recursive call

Watching the Stack

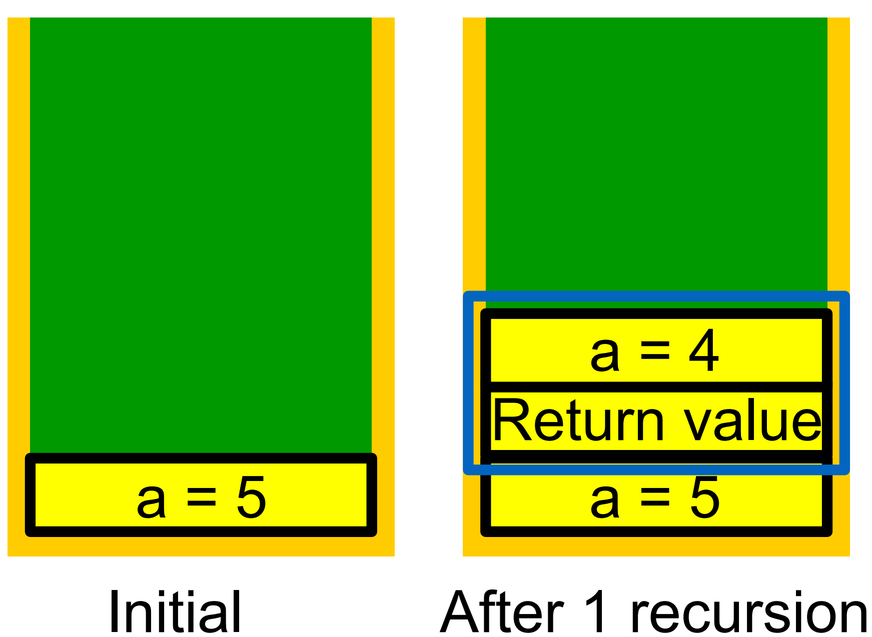
```
public int factorial(int a){
        if (a==0 || a==1)
          return(1);
        else
          return(a * factorial(a-1));
}
```



Initial

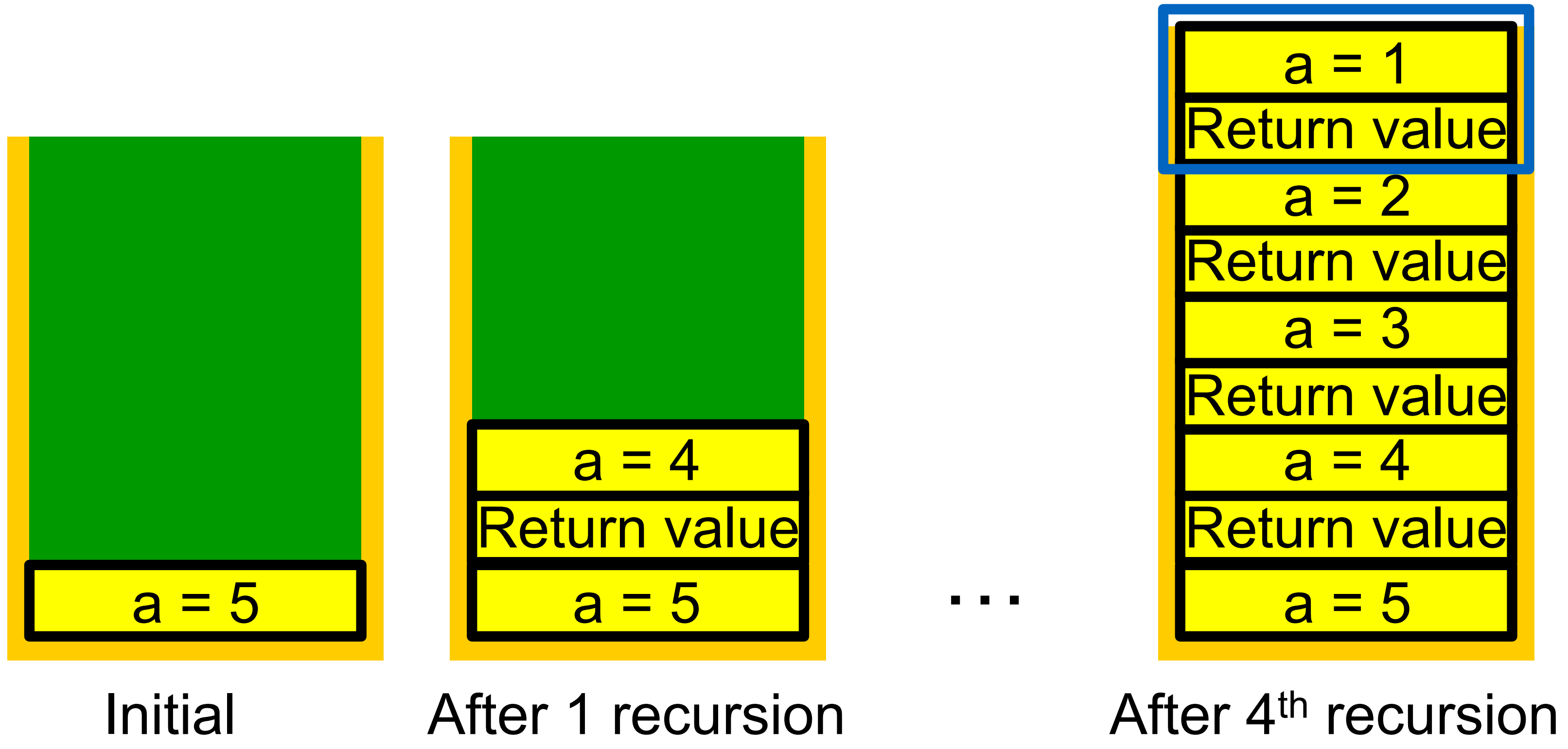
Watching the Stack

```
public int factorial(int a){
        if (a==0 || a==1)
          return(1);
        else
          return(a * factorial(a-1));
}
```



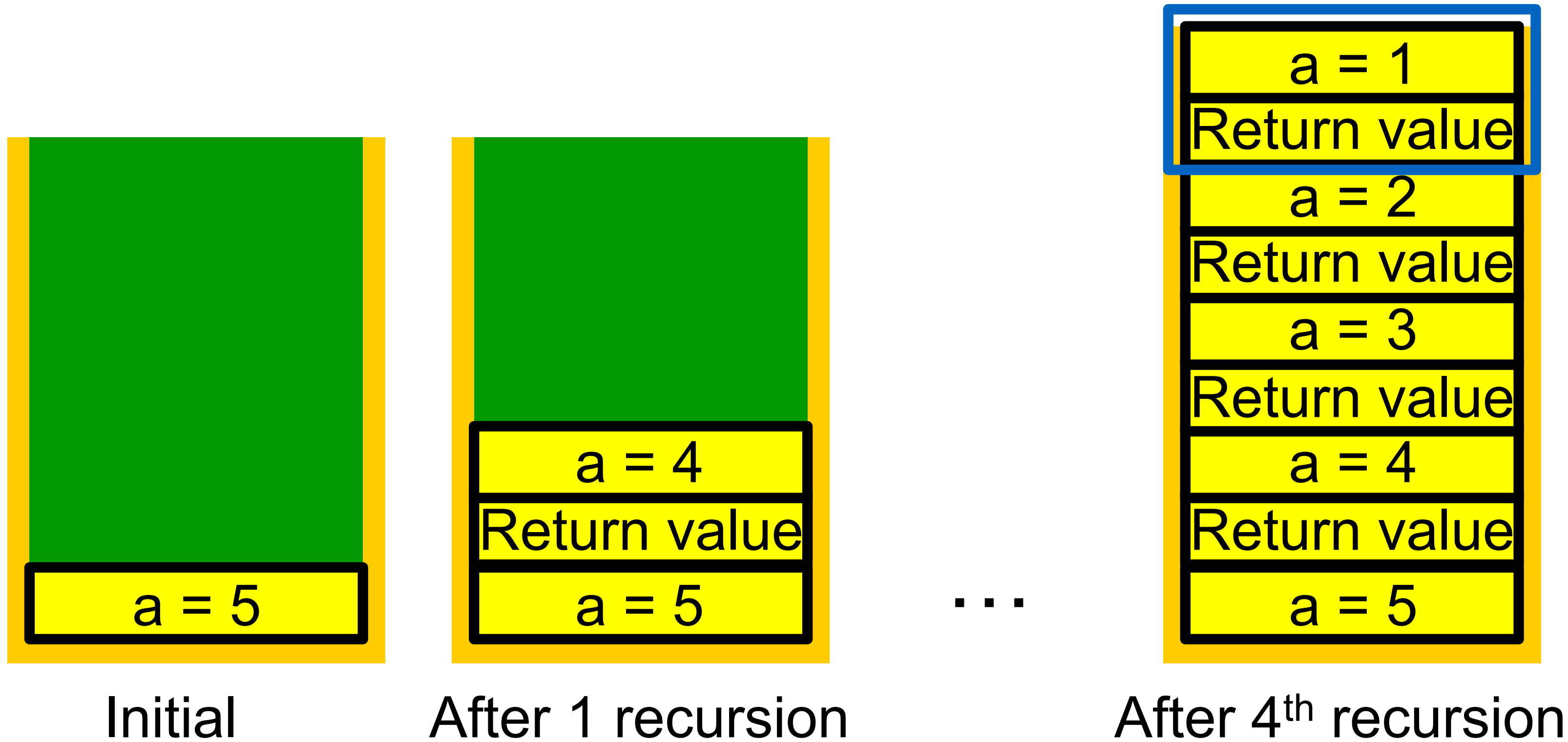
Watching the Stack

```
public int factorial(int a){
            if (a==0 || a==1)
              return(1);
            else
              return(a * factorial(a-1));
}
```



Watching the Stack

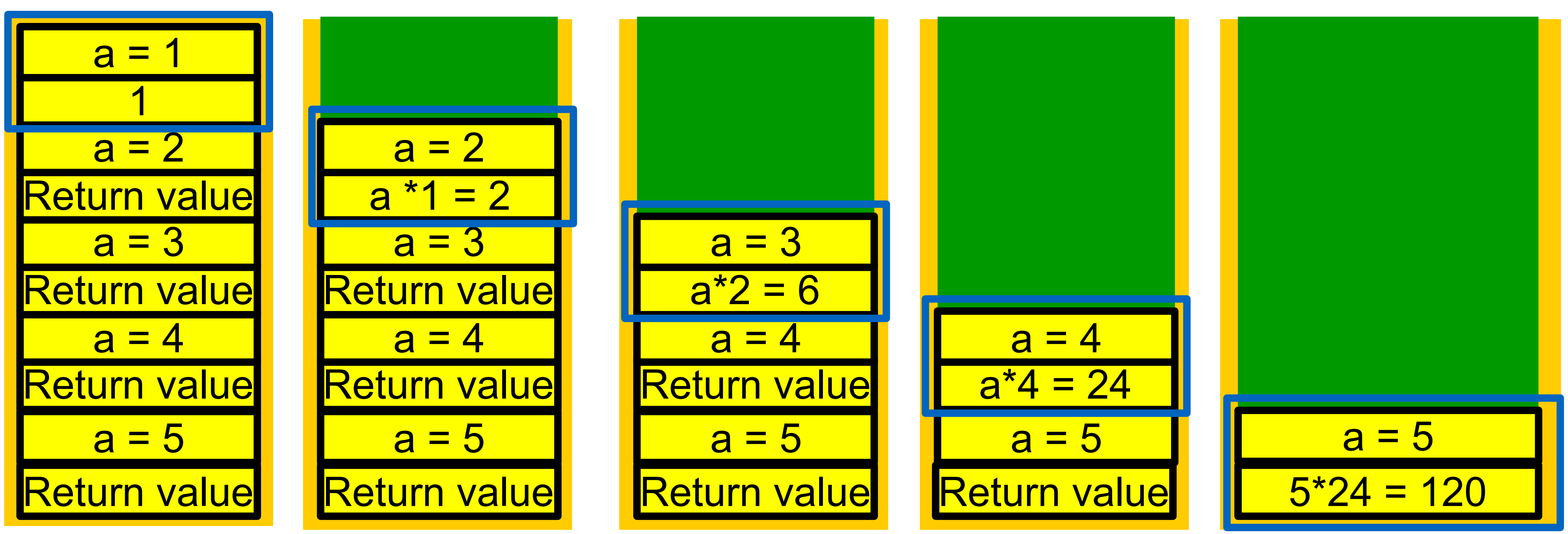
```
public int factorial(int a){
        if (a==0 || a==1)
          return(1);
        else
          return(a * factorial(a-1));
}
```



Every call to the method creates a new set of local variables (Stack Frame)!

Watching the Stack

```
public int factorial(int a){
            if (a==0 || a==1)
              return(1);
            else
              return(a * factorial(a-1));
}
```



After 4th recursion $\rightarrow$ Result

Iterative Factorial

Factorial can be write in iterative way

```
public int f(int a){
  int result = 1
  for (int i = 1; i <= a; i++)
      result *= i;
  return result;
}
```

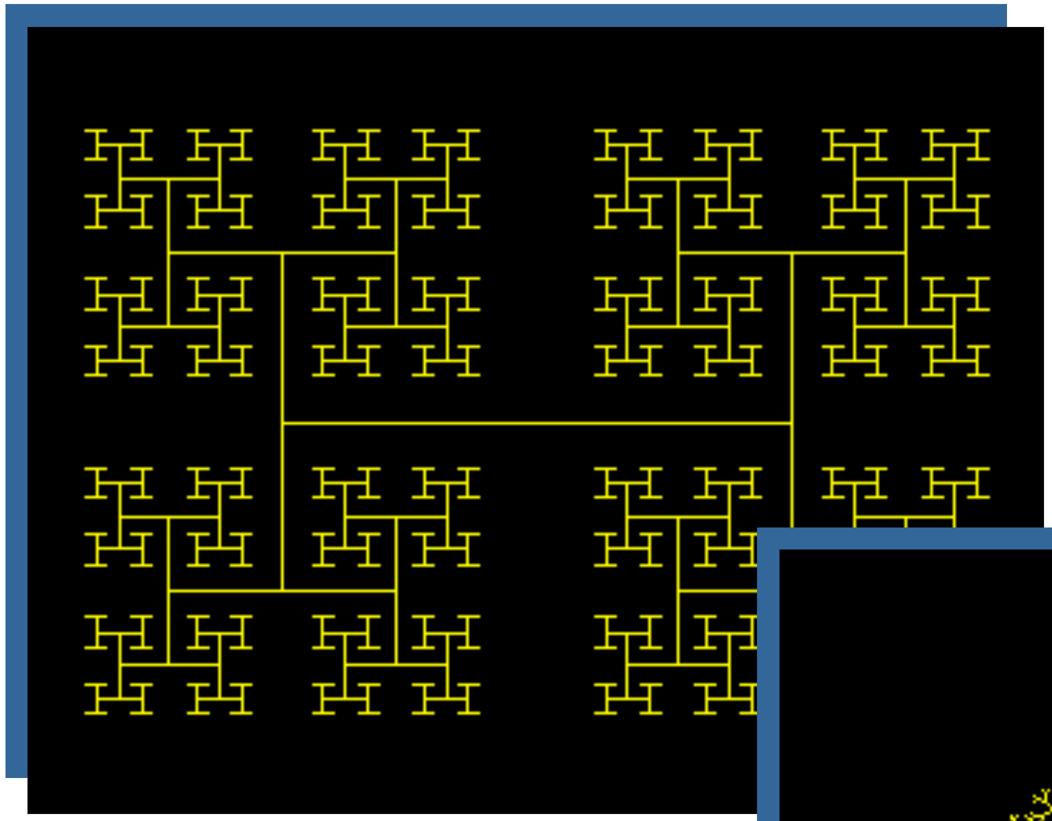
Recursion vs. Iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*
- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions
- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code
- (Nearly) every recursively defined problem can be solved iteratively
  - iterative optimization can be implemented after recursive design

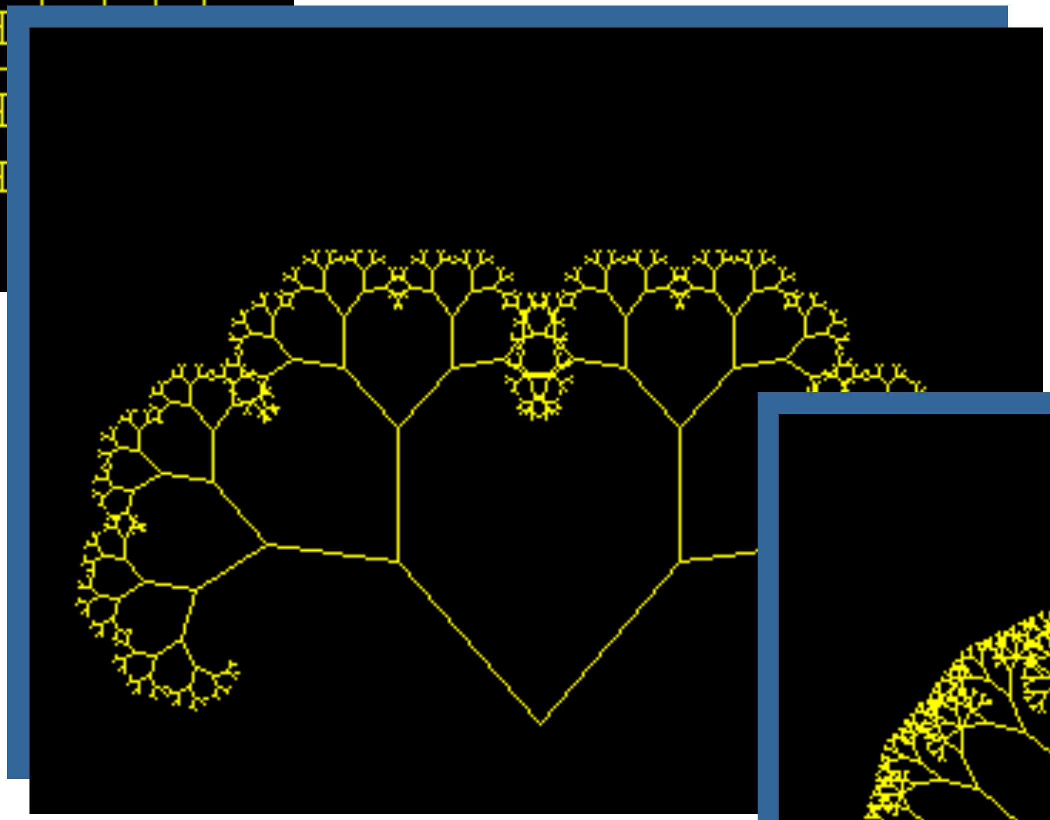
Deciding whether to use a Recursive Function

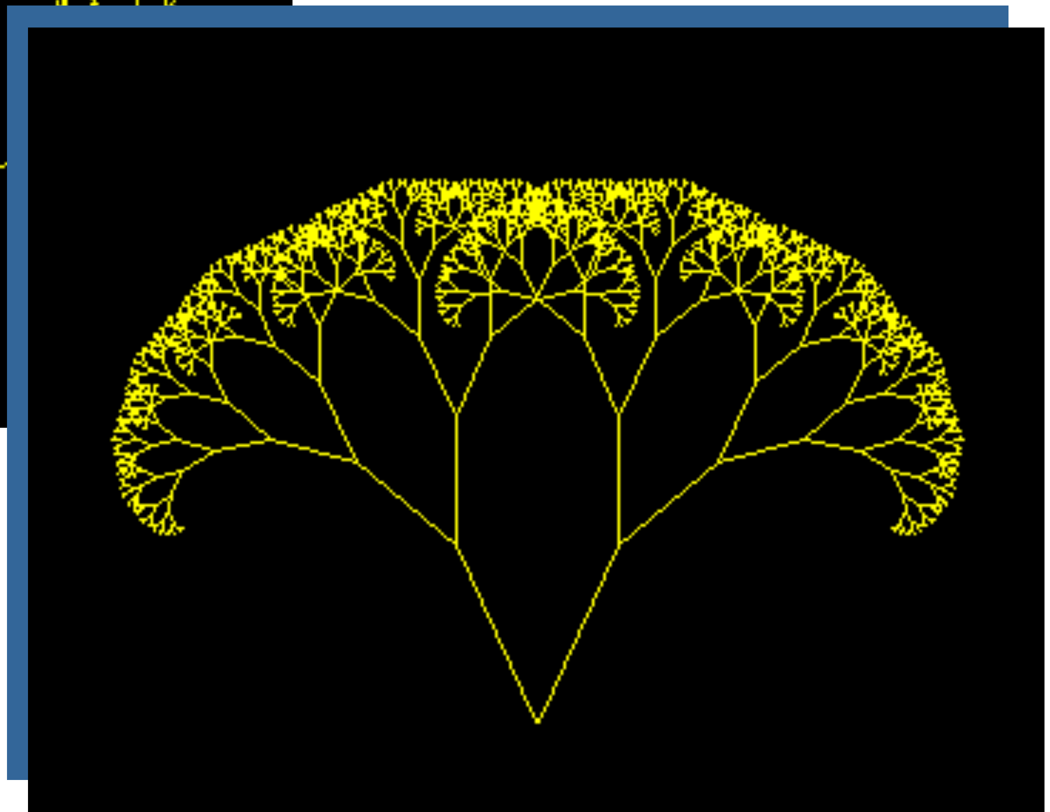
- When the depth of recursive calls is relatively "shallow"
- The recursive version does about the same amount of work as the non-recursive version
- The recursive version is shorter and simpler than the non-recursive solution

Examples: Fractal Tree



http://id.mind.net/~zona/mmts/geometrySection/fractals/tree/treeFractal.html





Design Recursive Algorithms

Designing Recursive Algorithms

- General strategy: “Divide and Conquer”
- Questions to ask yourself
  - How can we reduce the problem to smaller version of the same problem?
  - How does each call make the problem smaller?
  - What is the base case?
  - Will you ***always*** reach the base case?

Properties of Recursive Functions

Problems that can be solved by recursion have these characteristics:

- One or more stopping cases have a simple, nonrecursive solution
- The other cases of the problem can be reduced (using recursion) to problems that are closer to stopping cases
- Eventually the problem can be reduced to only stopping cases, which are relatively easy to solve

Follow these steps to solve a recursive problem:

- Try to express the problem as a simpler version of itself
- Determine the stopping cases (base case of recursion)
- Determine the recursive steps (recursive call)

Solution

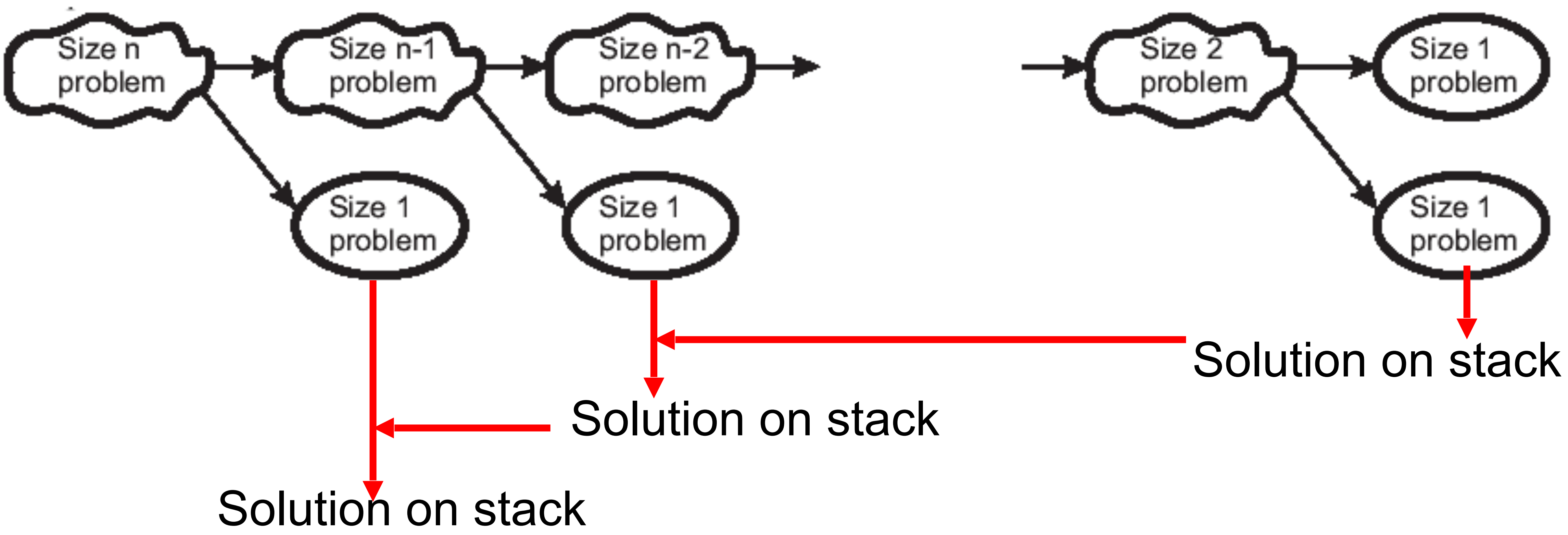
The recursive algorithms we write generally consist of an if statement:

IF

the stopping case is reached solve it

ELSE

split the problem into simpler cases using recursion



Exercise

Write a recursive function that prints the numbers 1…n in descending order:

```
public void descending(int n) {



}
```

Exercise

Write a recursive function that prints the numbers 1…n in descending order:

```
public void descending(int n) {
  if (n <= 0) return;
  System.out.println(n);
  descending(n-1);
}
```

Exercise

Write a recursive function to perform exponentiation return x^m, assuming m >= 0

```
public int exp(int x, int m) {



}
```

Exercise

Write a recursive function to perform exponentiation return x^m, assuming m >= 0

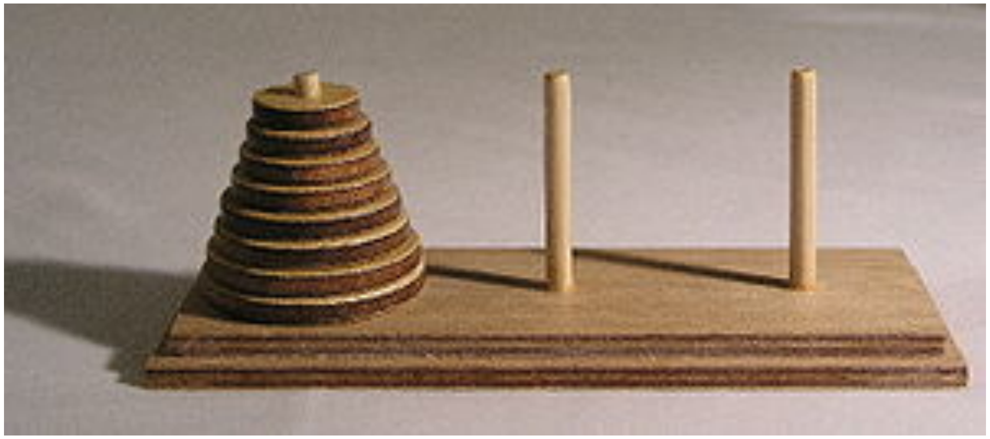
```
public int exp(int x, int m) {
  if (m == 0) { return 1; }
  if (m == 1) { return x; }
  return x * exp(x, m-1);
}
```

```
public static boolean  p(string s, int i, int f)
  if (i < f) {
    if (s[i] == s[f]) {
      return p(s, i+1, f-1);
    } else {
      return false;
    }
  } else {
    return true;
  }
}
```

```
What does  p(s,0,s.length-1) return
a) if s ="UTEP'
b) if s ="SAMS'
c) if s ="kayak"
d) if s= "ABBA"
```

Towers of Hanoi

- The legend of the temple of Brahma
  - 64 golden disks on 3 pegs
  - The universe will end when the priest move all disks from the first peg to the last



The Rules

- Only move one disk at a time
- A move is taking one disk from a peg and putting it on another peg (on top of any other disks)
- Cannot put a larger disk on top of a smaller disk
- With 64 disks, at 1 second per disk, this would take roughly 585 billion years

Towers of Hanoi: Three Disk Problem

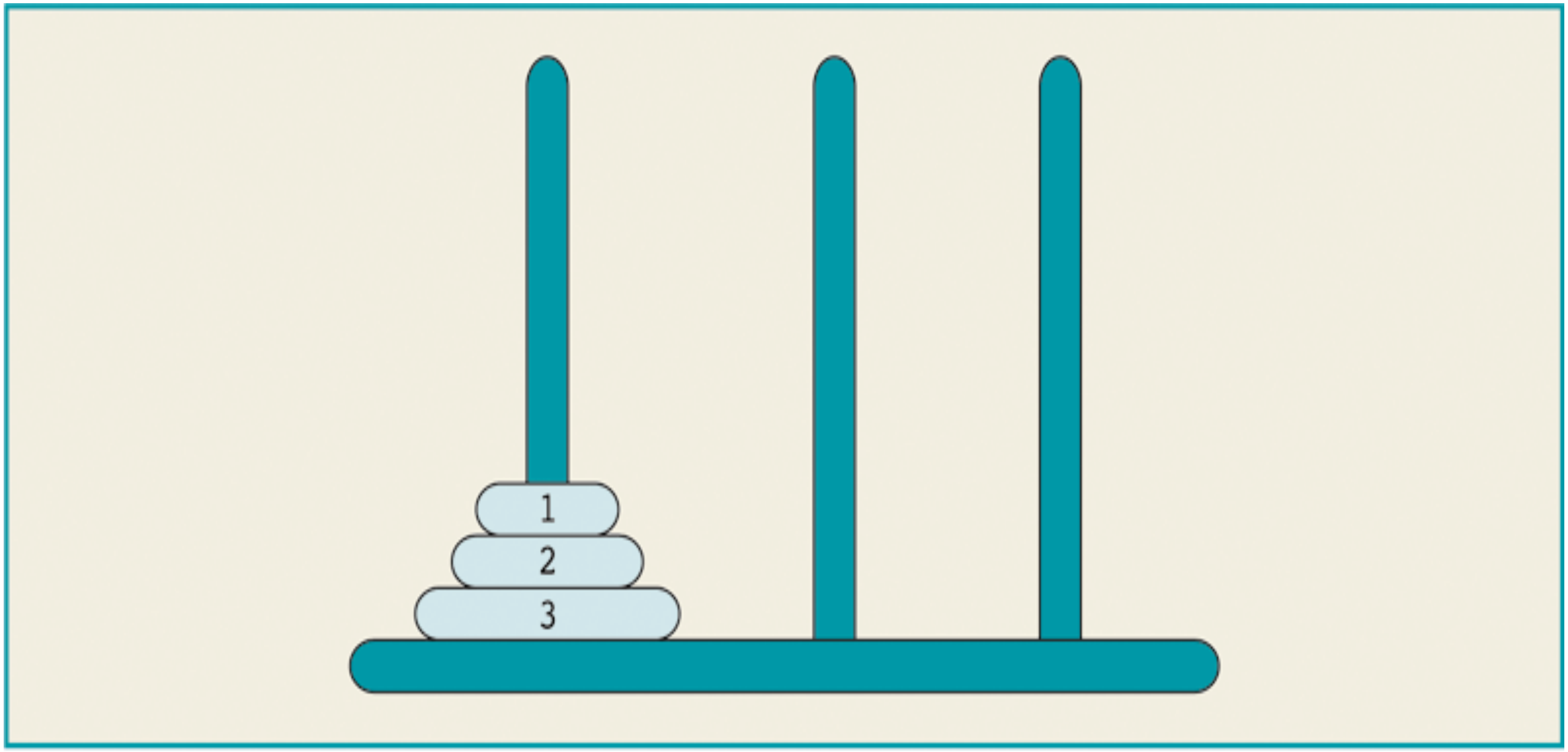
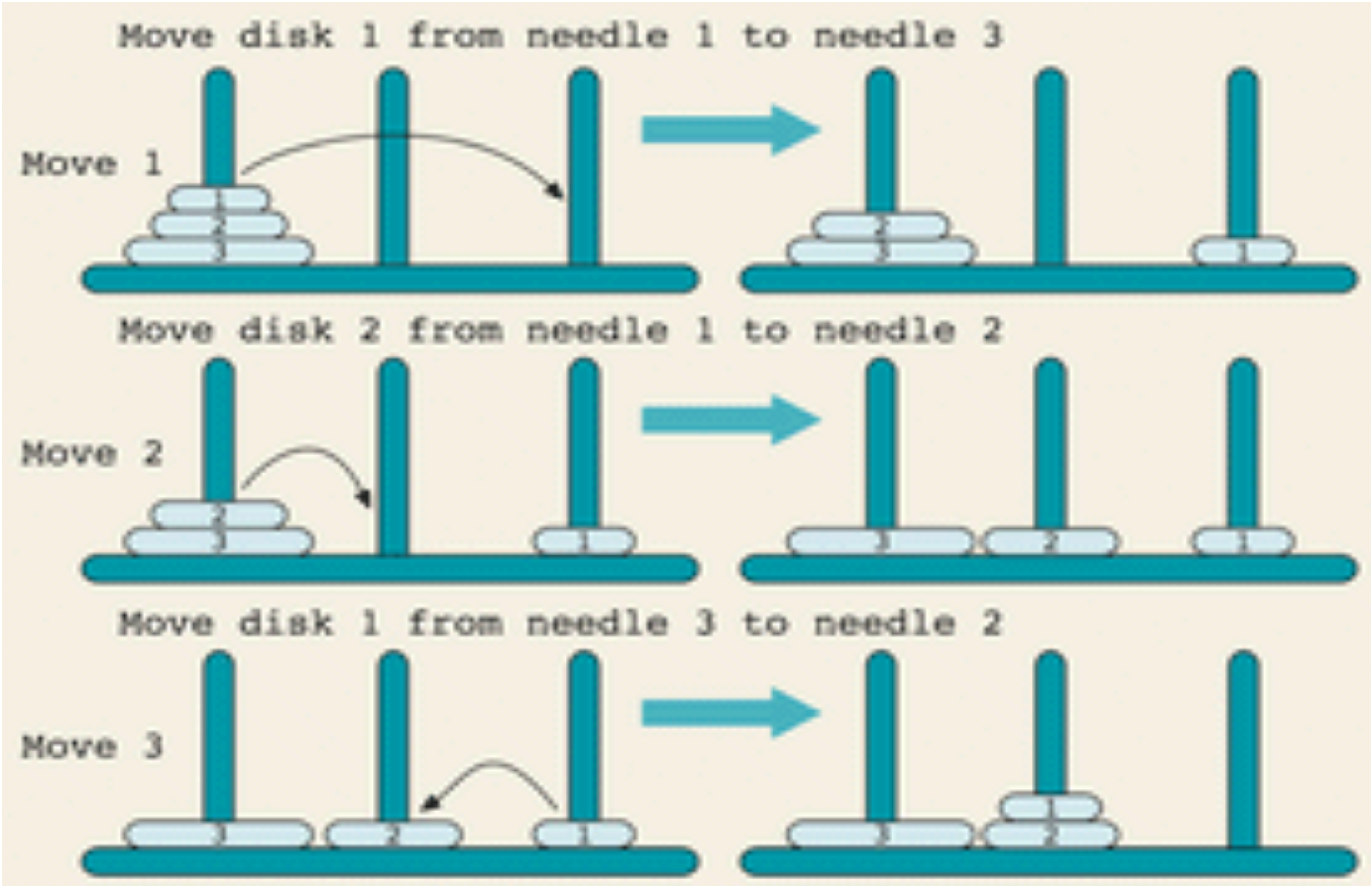
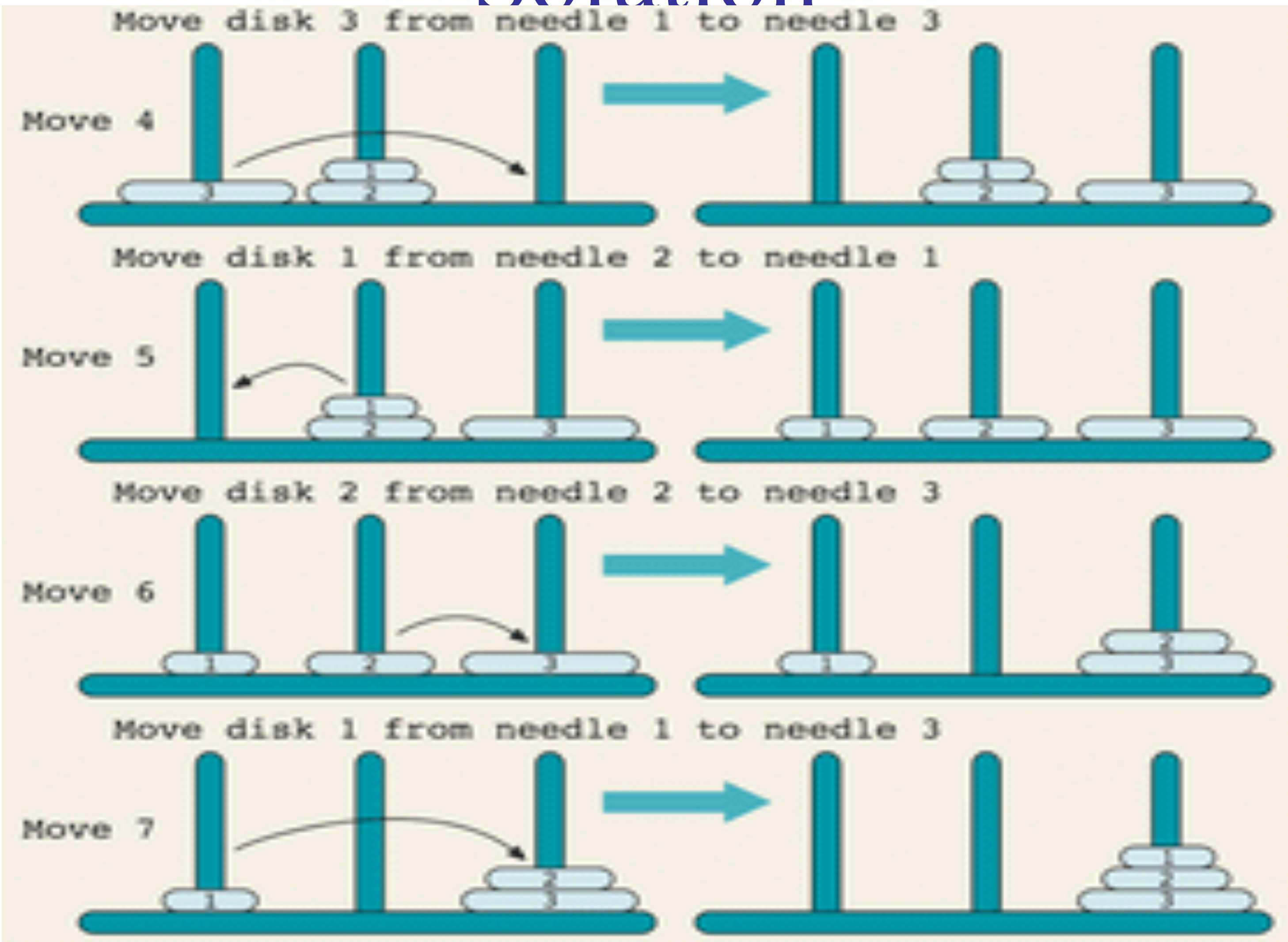


Figure 14-6 Tower of Hanoi problem with three disks

Towers of Hanoi: Three Disk Solution



Towers of Hanoi: Three Disk Solution



Four disk solution (courtesy wikipedia)

Recursive algorithm idea

- Final step is to move the bottom disk from peg 1 to peg 3
- To do this, the other n-1 disks must be on peg 2
- So, we need an way to move n-1 disks from peg 1 to peg 2
- Base case: moving the smallest disk is easy (you can always move it to any peg in one step)

Pseudocode

```
solveTowers (count, source, destination, spare) {
  if (count == 1) {
    move directly
  }
  else {
    solveTowers(count-1, source, spare, destination)
    solveTowers(1, source, destination, spare)
    solveTowers(count-1, spare, destination, source)
  }
}
```