1.7 Recursion

Objectives

- To learn the concept and usage of Recursion in C
- Examples of Recursion in C

What is recursion?

- Sometimes, the best way to solve a problem is by solving a **smaller version** of the exact same problem first

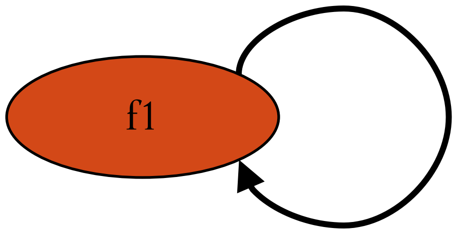
- Recursion is a technique that solves a problem by solving a **smaller problem** of the same type

- a function that calls itself

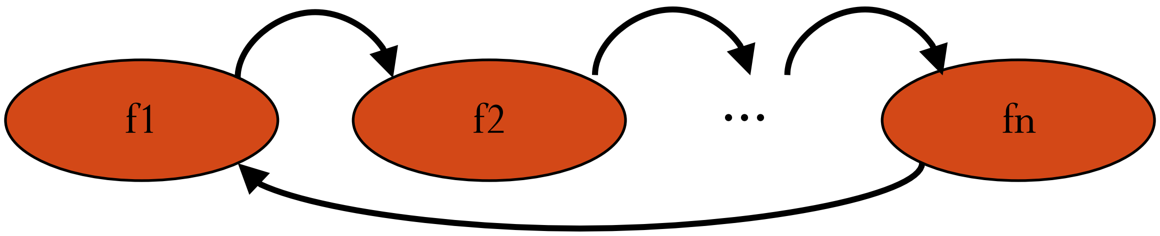
    Directly or

    Indirectly (a function that is part of a cycle in the sequence of function calls.)

Pictorial representation of direct and indirect recursive calls



Direct recursive call



Indirect recursive call

Syntax

```
function_name(parameter list)
{
        …
        // 'c' statements
        …
        function_name(parameter values)   // recursive call
        …
}
```

Problems defined recursively

- There are many problems whose solution can be defined recursively

Example: *n factorial*

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! * n & \text{if } n > 0 \end{cases}$$

(*recursive* solution)

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

(*closed form* solution)

(also called as iterative method)

Coding the factorial function

- Iterative implementation

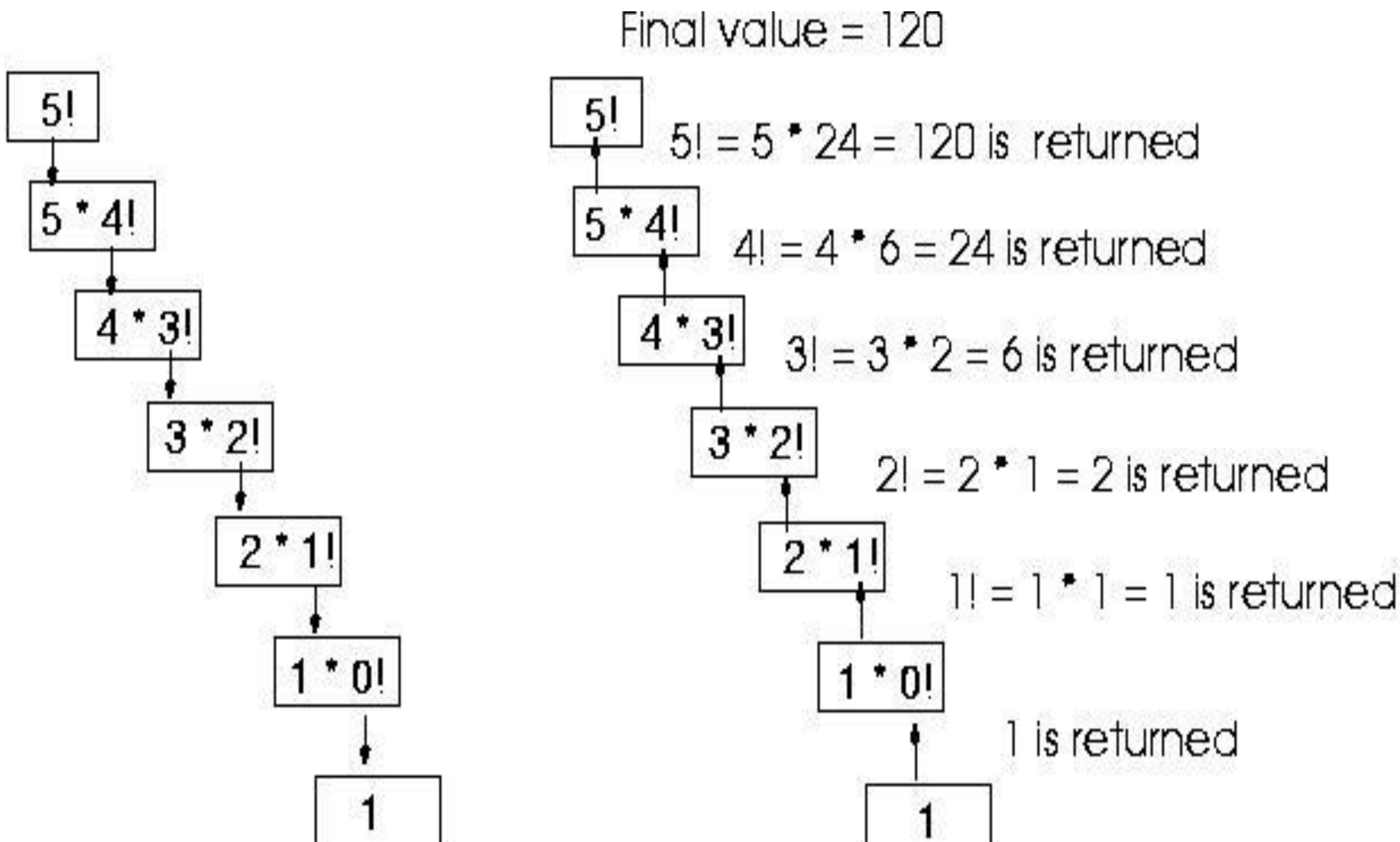
```
int Factorial(int n)
{
 int fact = 1;

 for(int count = 2; count <= n; count++)
   fact = fact * count;

 return fact;
}
```

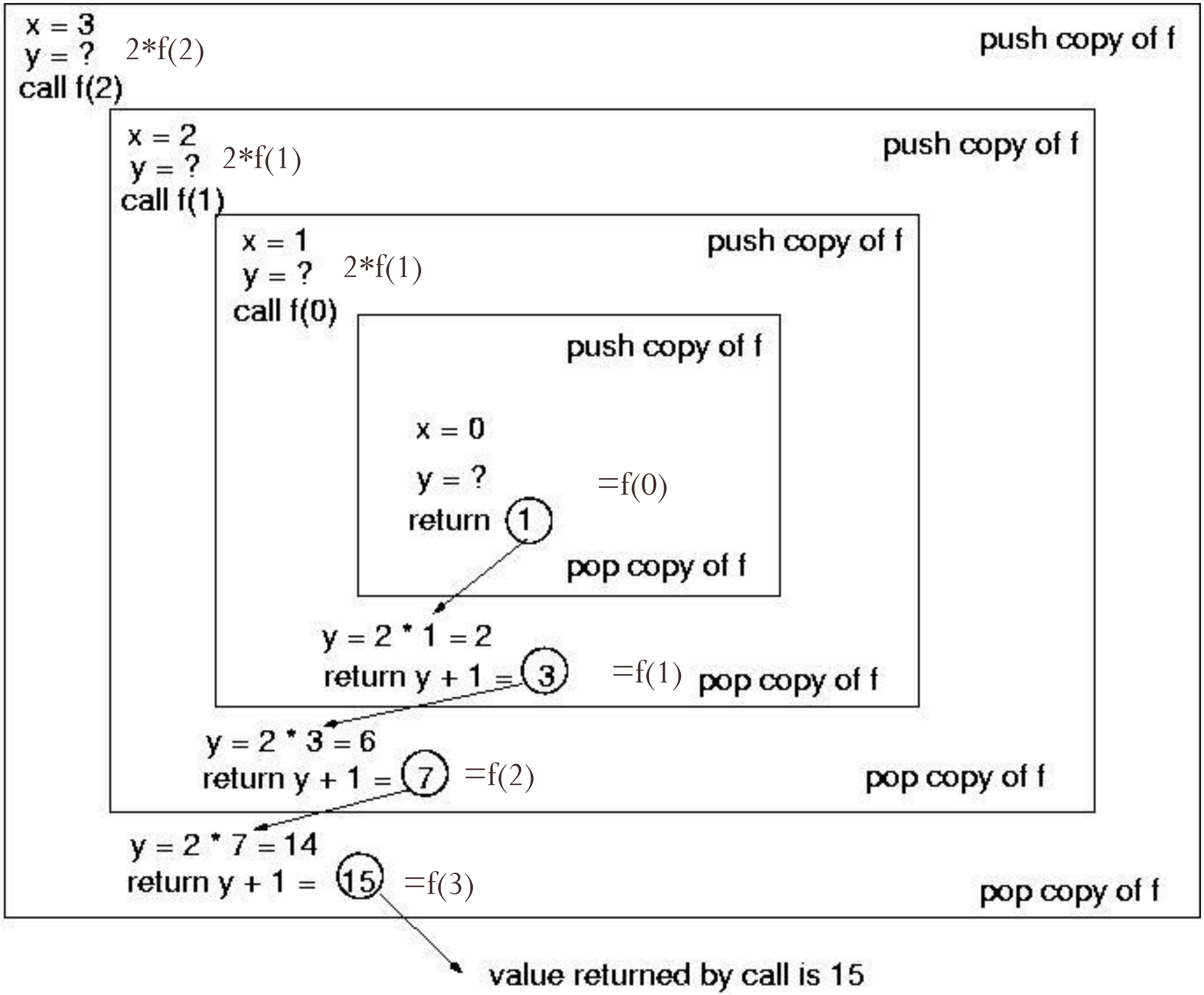
Coding the factorial function
(An Example of Recursive Call)

- Recursive implementation

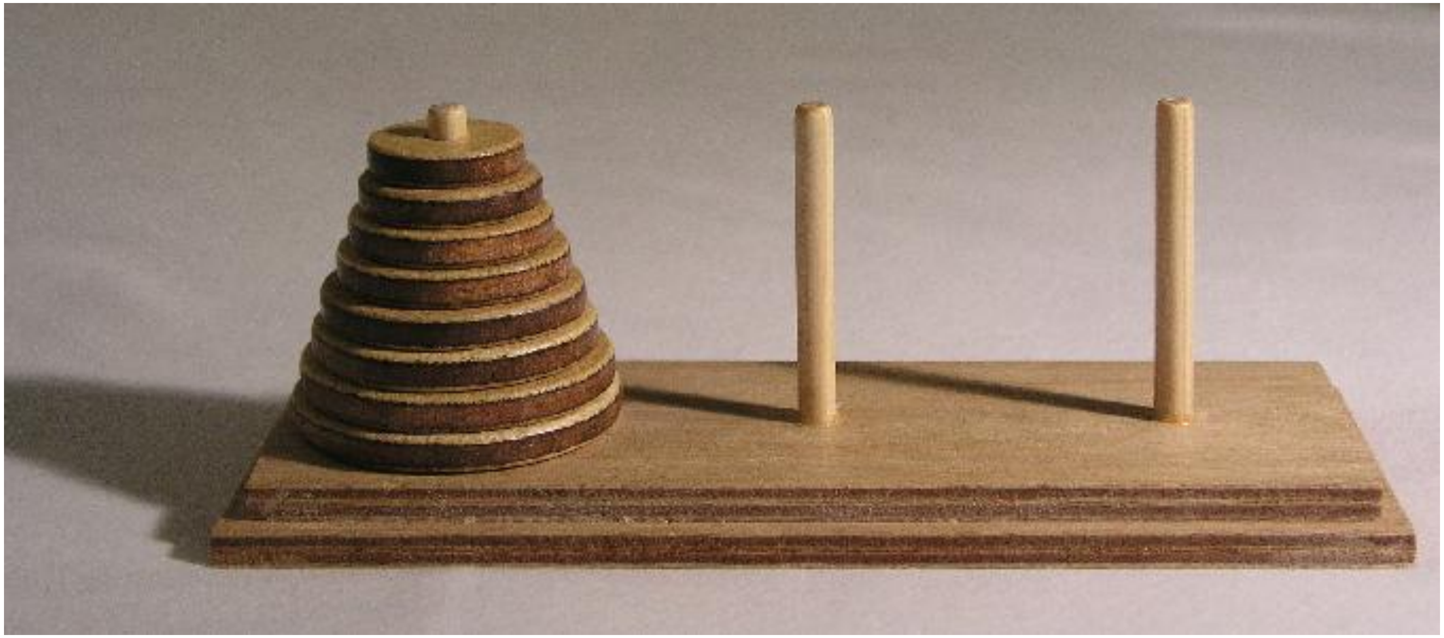
```
int Factorial(int n)
{
 if (n==0)  // base case
   return 1;
 else
   return n * Factorial(n-1);
}
```

Coding the factorial function (cont.)




x = 3
y = ? 2*f(2)
call f(2)
push copy of f
x = 2
y = ? 2*f(1)
call f(1)
push copy of f
x = 1
y = ? 2*f(1)
call f(0)
push copy of f
push copy of f
x = 0
y = ?
=f(0)
return 1
pop copy of f
y = 2 * 1 = 2
return y + 1 = 3
=f(1)
pop copy of f
y = 2 * 3 = 6
return y + 1 = 7
=f(2)
pop copy of f
y = 2 * 7 = 14
return y + 1 = 15
=f(3)
pop copy of f
value returned by call is 15

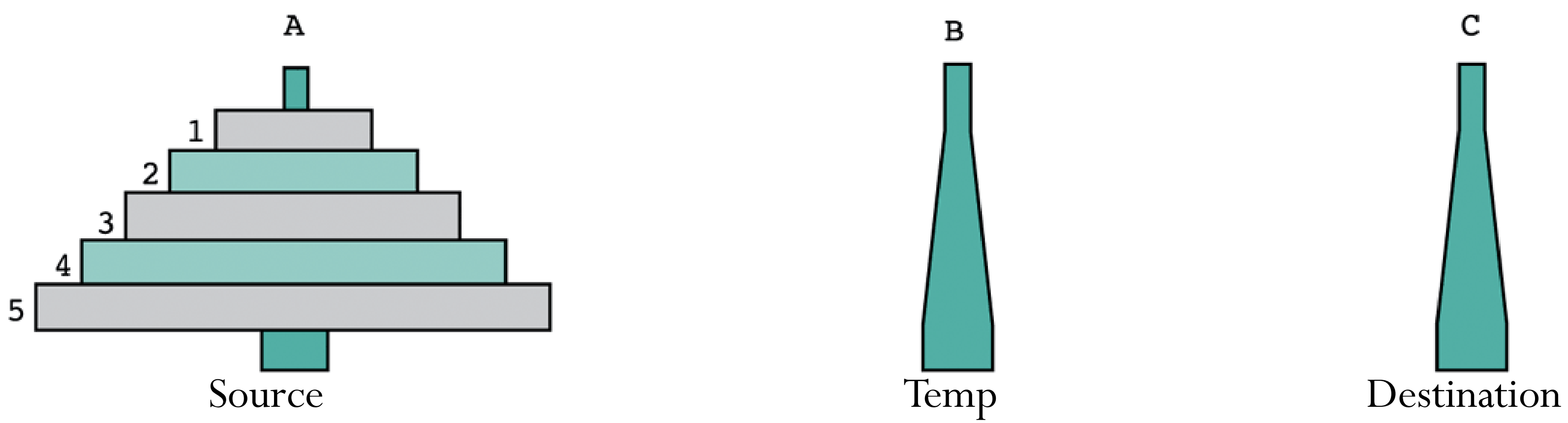
More Interesting Example
Towers of Hanoi



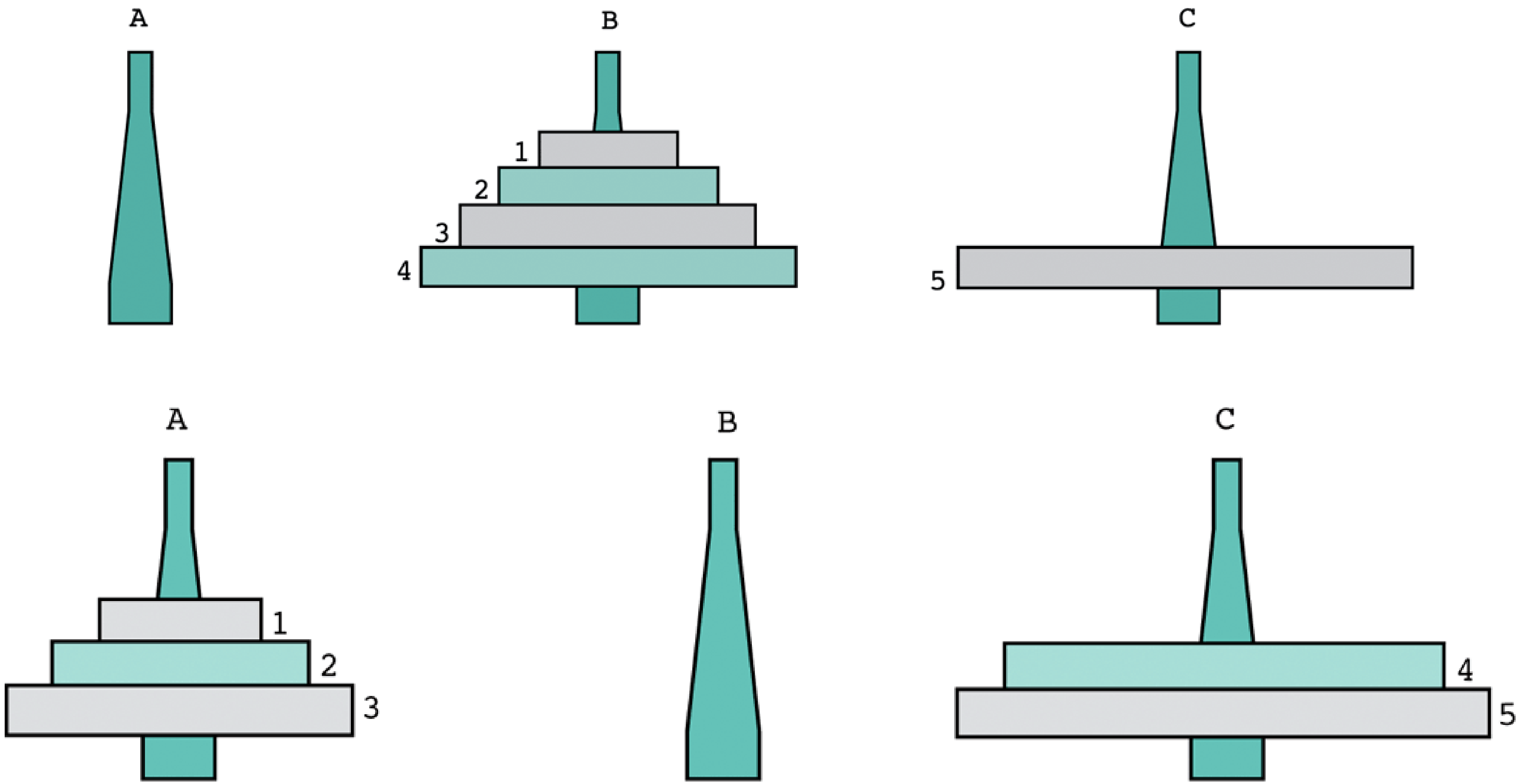
- Move stack of disks from one peg to another
- Move one disk at a time
- Larger disk may never be on top of smaller disk

A Classical Case: Towers of Hanoi

- The towers of Hanoi problem involves moving a number of disks (in different sizes) from one tower (or called “peg”) to another.
  - The constraint is that the larger disk can never be placed on top of a smaller disk.
  - Only one disk can be moved at each time
  - Assume there are three towers available.



A Classical Case: Towers of Hanoi



A Classical Case: Towers of Hanoi

- This problem can be solved easily by recursion.
- Algorithm:

  if n is 1 then

  move disk 1 from the source tower to the destination tower

  else

  1. move n-1 disks from the source tower to the temp tower.
  2. move disk n from the source tower to the destination tower.
  3. move n-1 disks from the temp tower to the source tower.

Tower of Hanoi Program

```
#include <stdio.h>

void move (int n, int a, int
   c, int b);

int main() {
  int disks;
  printf ("How many disks?");
  scanf ("%d", &disks);

  move (disks, 1, 3, 2);

  return 0;
}  // main
```

```
/* PRE:  n >= 0. Disks are arranged
    small to large on the pegs a, b,
    and c. At least n disks on peg
    a. No disk on b or c is smaller
    than the top n disks of a.
POST: The n disks have been moved
    from a to c. Small to large
    order is preserved. Other disks
    on a, b, c are undisturbed. */

void move (int n, int a, int c, int
    b) {
  if (n > 0)
    {
      move (n-1, a, b, c);
      printf ("Move one disk
    from %d to %d\n", a, c);
                     c, a);
                      0)

}  //    move
```

- Is pre-condition satisfied before this call to **move**?

Tower of Hanoi Program

```
#include <stdio.h>

void move (int n, int a, int
   c, int b);

int main() {
  int disks;
  printf ("How many disks?");
  scanf ("%d", &disks);

  move (disks, 1, 3, 2);
```

```
/* PRE:  n >= 0. Disks are arranged
   small to large on the pegs a, b,
   and c. At least n disks on peg
   a. No disk on b or c is smaller
   than the top n disks of a.
POST: The n disks have been moved
   from a to c. Small to large
   order is preserved. Other disks
   on a, b, c are undisturbed. */

void move (int n, int a, int c, int
   b) {
  if (n > 0)
   {
     move (n-1, a, b, c);
     printf ("Move one disk
  from %d to %d\n", a, c);
     move (n-1, b, c, a);
   }    // if (n > 0)

  return;
}  //    move
```

- If pre-condition is satisfied here, is it still satisfied here?

And here?

Tower of Hanoi Program

```
#include <stdio.h>

void move (int n, int a, int
   c, int b);

int main() {
  int disks;
  printf ("How many disks?");
  scanf ("%d", &disks);

  move (disks, 1, 3, 2);

  return 0;
}  // main
```

If pre-condition is true and if n = 1, does **move** satisfy the post-condition?

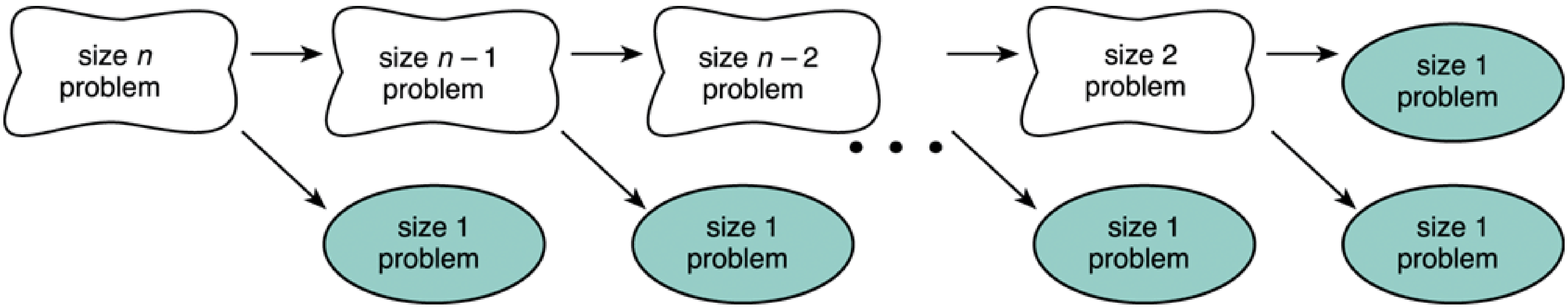
```
/* PRE:  n >= 0. Disks are arranged
   small to large on the pegs a, b,
   and c. At least n disks on peg
   a. No disk on b or c is smaller
   than the top n disks of a.
POST: The n disks have been moved
   from a to c. Small to large
   order is preserved. Other disks
   on a, b, c are undisturbed. */

void move (int n, int a, int c, int
   b) {
  if (n > 0)
   {
     move (n-1, a, b, c);
     printf ("Move one disk
   from %d to %d\n", a, c);
     move (n-1, b, c, a);
   }    // if (n > 0)

  return;
}  //   move
```

Can we reason that this program correctly plays Tower of Hanoi?

Problems Suitable for Recursive Functions

- One or more simple cases of the problem have a straightforward solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- The problem can be reduced entirely to simple cases by calling the recursive function.
  - *If this is a simple case*
    solve it
    else
    redefine the problem using recursion

Splitting a Problem into Smaller Problems



- Assume that the problem of size 1 can be solved easily (i.e., the simple case).
- We can recursively split the problem into a problem of size 1 and another problem of size n-1.

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

Recursion *vs.* Iteration (Contd…)

- Some simple recursive problems can be "unwound" into loops
  - But code becomes less compact, harder to follow!
- Hard problems cannot easily be expressed in non-recursive code
  - Tower of Hanoi
  - Robots or avatars that "learn"
  - Advanced games

Try it Yourself

- Generate a fibonacci series using recursion

  Recursive definition for $\{f_n\}$:

  INITIALIZATION: $f_0 = 0, f_1 = 1$

  RECURSION : $f_n = f_{n-1} + f_{n-2}$ for $n > 1$

- Finding the GCD
  - Euclid's algorithm makes use of the fact that

$$\gcd(x,y) = \gcd(y, x \,\mathbf{mod}\, y)$$

$$\gcd(x,y) = \begin{cases} x \text{ if} & y=0 \\ \gcd(y, x \bmod y) & \text{otherwise} \end{cases}$$

Summary

- Discussed so far the
  - What is a recursion (function) ?
  - What is the need for the recursive function?
  - Writing recursive functions using C
  - How a hard/difficult problem can be solved by recursion
  - Comparison of Recursion with Iterative method