COP 3502: Computer Science I
Spring 2004

– Day 5 –
Recursion

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

Introduction to Recursion

- The art of solving problems existed long before there were computers to assist with problem solving.
- Many of the algorithmic strategies employed in programming solutions were developed outside the realm of computing.
  - A task performed repeatedly uses iteration.
  - A decision you make exercises conditional control.
- Because repetition and decisions are fairly well-known to most humans, you are able to adapt to the use of `for`, `while`, and `if` statements in a programming language easily.

Introduction to Recursion (cont.)

- On the road to become a good problem-solver you will discover that you need more powerful problem solving strategies that what `for`, `while`, and `if` statements can do for you.
- An important one of these strategies that you will need to learn is *recursion*.
- Recursion is defined as any solution technique in which a large problem is solved by reducing it to smaller problems of the same form.

A Simple Example of Recursion

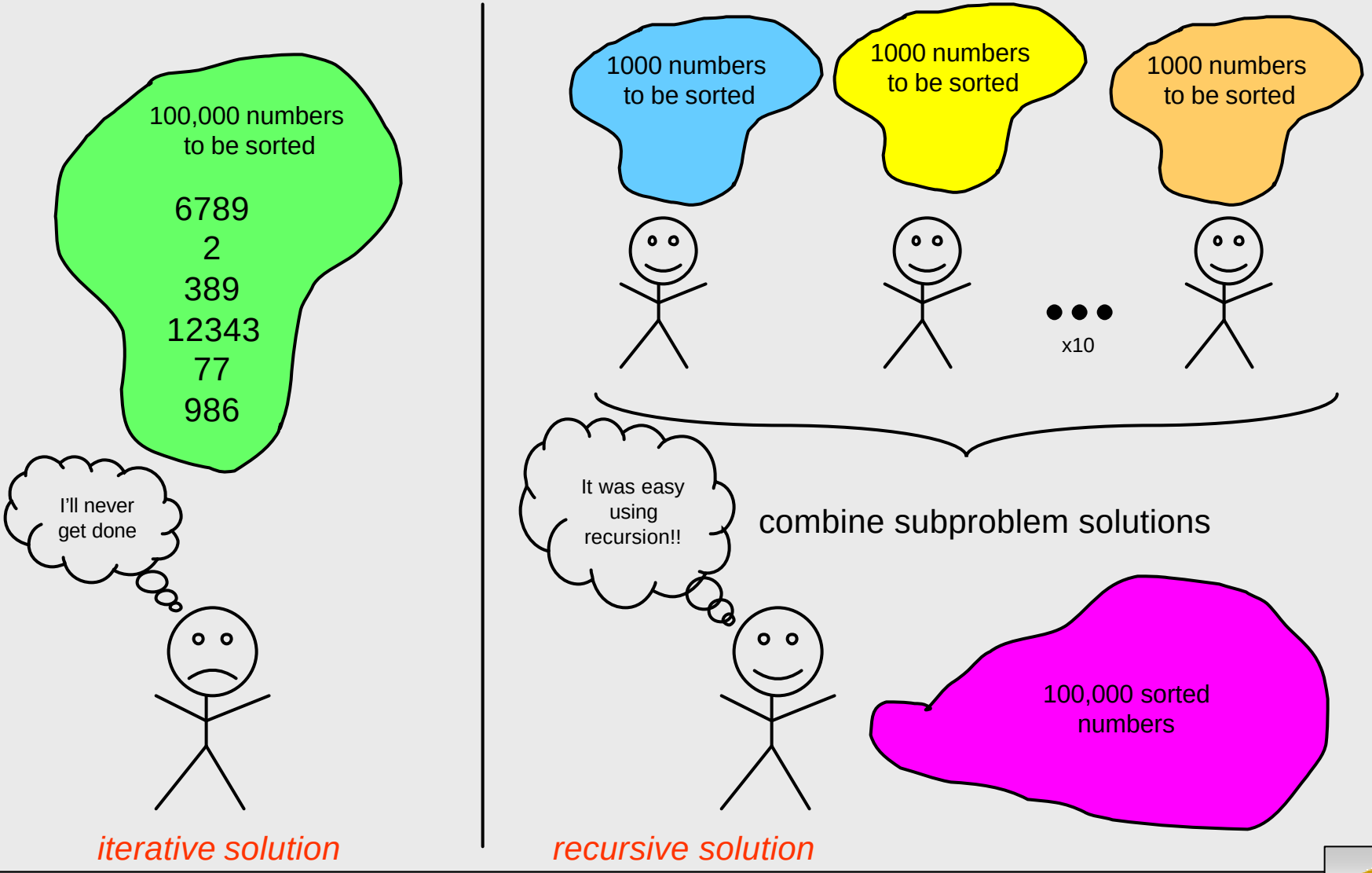
Problem: Suppose that I have a set of 100,000 numbers that I would like to have sorted.

Solution 1: I could sort the entire 100,000 numbers myself (not a good solution as I don't like to sort things!)

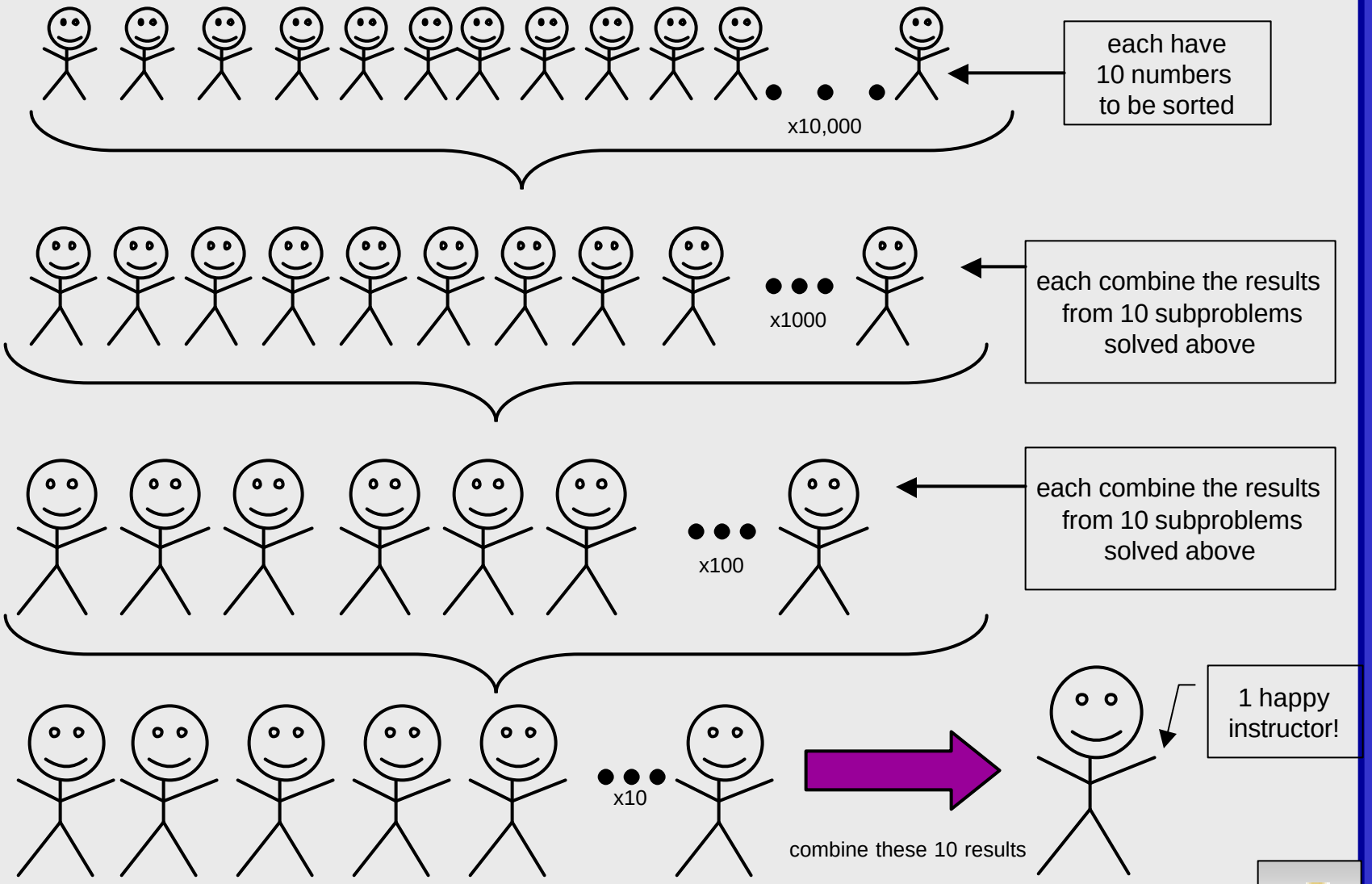
Solution 2: I could get the 100 students in the class to each sort 1000 of the 100,000 numbers. When everyone finishes sorting their batch of 1000 numbers, I will take the 100 sorted batches and reassemble them into a sorted list of 100,000 numbers (a better solution, because I don't do any sorting at all!)

- Solution 2 is a recursive solution to the problem. Each of the sub-problems is smaller than the original, but has the same form as the original problem, i.e., sorting numbers.

A Simple Example of Recursion (cont.)



A Simple Example of Recursion (cont.)



Recursive Functions

- In general, the body of a recursive function has the following form:

```
if (test for simple case)
    produce a simple solution without using recursion
else {
      break the problem into subproblems of the same form
      solve each of the subproblems by calling this function recursively
       reassemble the solutions to the subproblems into a complete solution
```


Recursive Functions (cont.)

- The form on the previous page can be used as a template for defining recursive functions and is thus referred to as the recursive paradigm.
- The recursive paradigm can be applied to any programming problem which meets the following criteria:

1. You must be able to identify the base cases (simple cases) for which an answer is easily determined.
2. You must be able to identify a recursive decomposition which allows you to break a complex instance of the problem into simpler subproblems of the same form.

Recursive Functions (cont.)

- Consider the definition of the factorial function:

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

- Let's use this recursive definition to calculate 4!.

$$\text{factorial}(4) = 4 \times \text{factorial}(3)$$

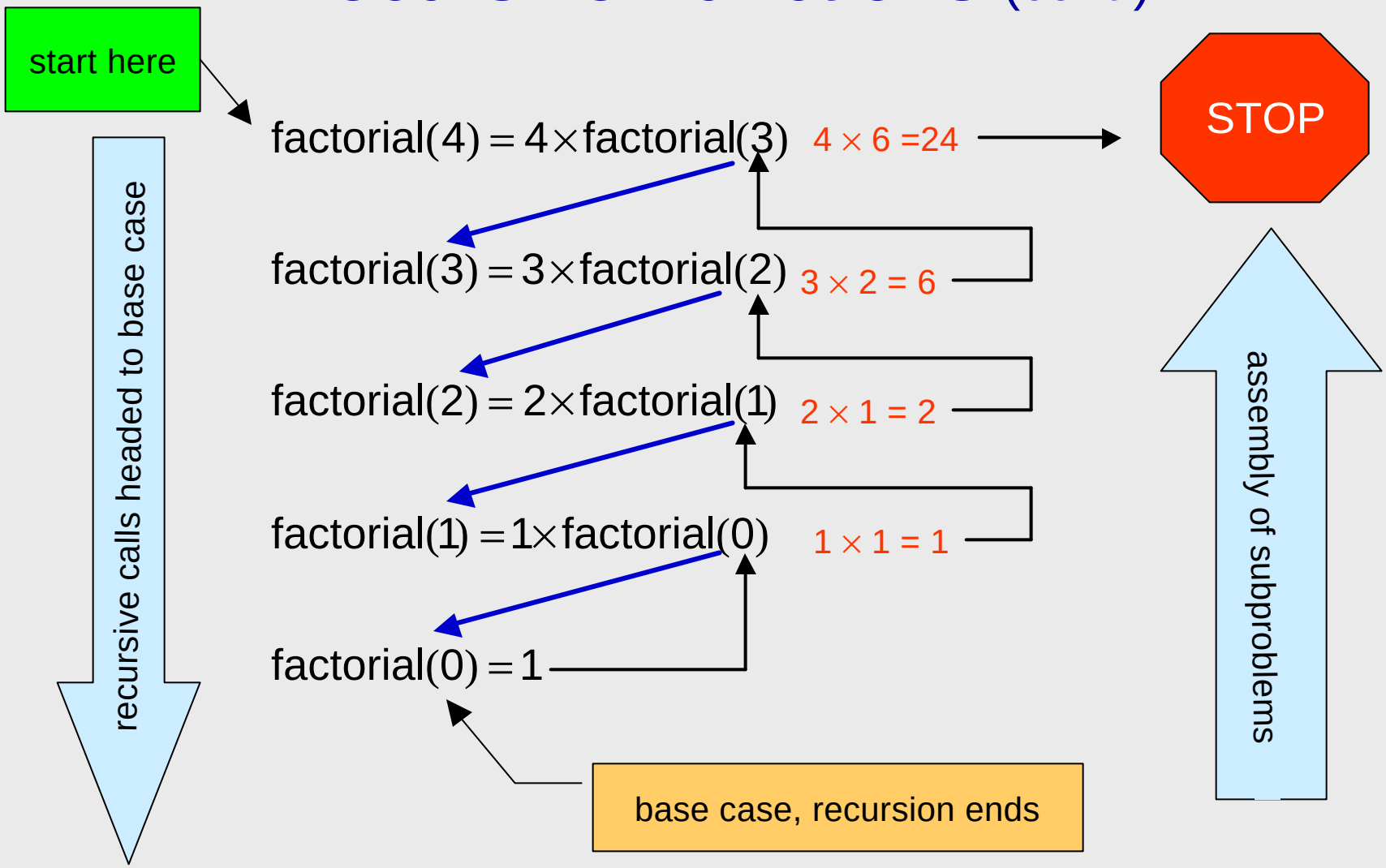
$$\text{factorial}(3) = 3 \times \text{factorial}(2)$$

$$\text{factorial}(2) = 2 \times \text{factorial}(1)$$

$$\text{factorial}(1) = 1 \times \text{factorial}(0)$$

$$\text{factorial}(0) = 1$$

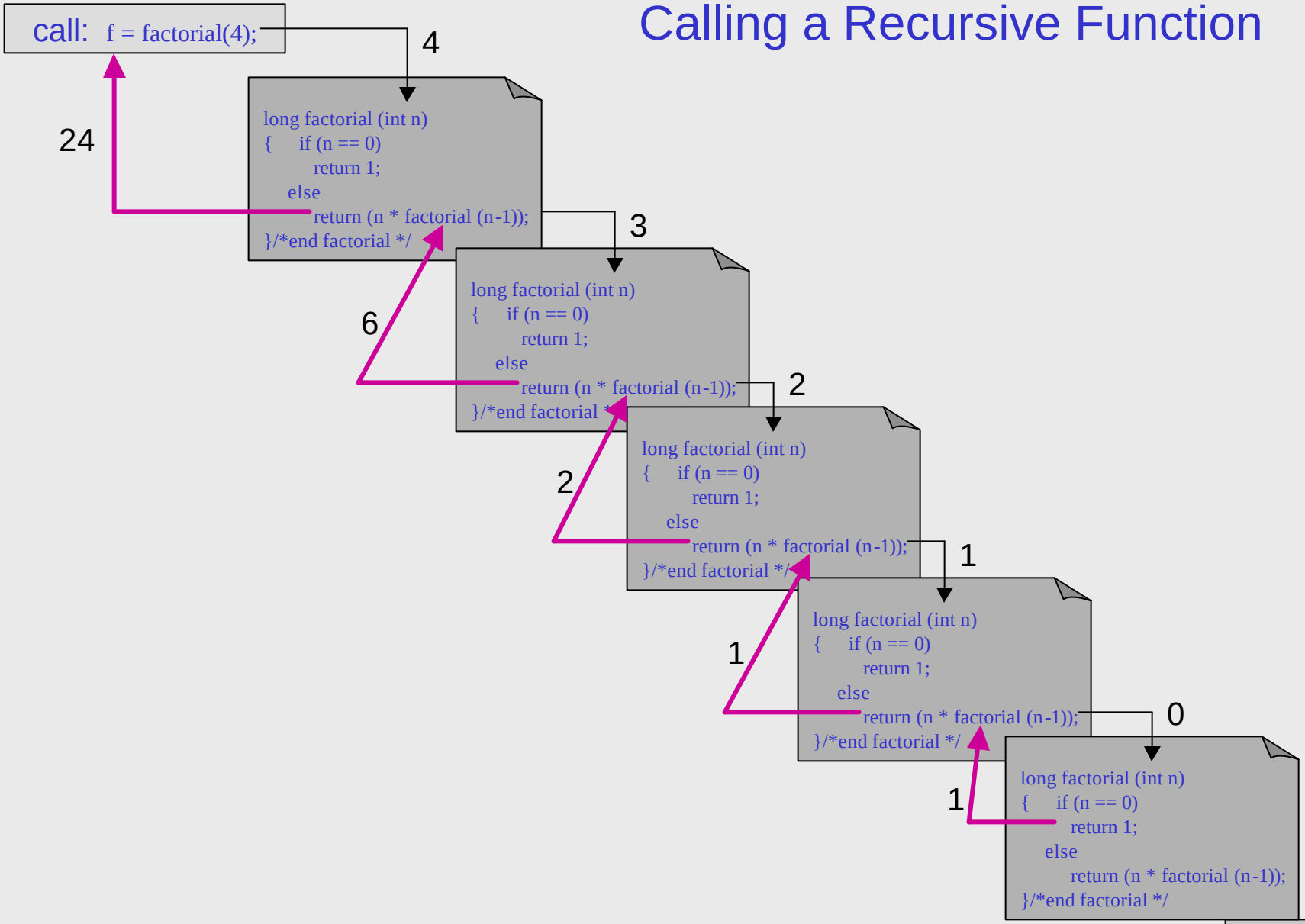
Recursive Functions (cont.)



Recursive Factorial Function in C

```
/* recursive factorial function                */
/* assumption is that result fits in a long */
long factorial (int n)
{
    if (n == 0)
        return 1;
    else
        return (n * factorial (n-1));
}/*end factorial */
```


Calling a Recursive Function



Recursive Functions (cont.)

- All recursive functions have two elements:

  1. Each call either solves one part of the problem (base case) or,
  2. It reduces the size of the problem (general case).

- In the factorial function, the statement: return 1; solves a small piece of the problem, i.e., factorial(0) = 1, while the statement: factorial(n-1); reduces the size of the problem by recursively calling the factorial function with n-1.

Fibonacci Numbers

- Fibonacci numbers can be defined by the following recursive function:

$$\text{Fibonacci}(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) & \text{if } n > 1 \end{cases}$$

- The Fibonacci numbers are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …
- The code on the next page is a C implementation of a program to print the first n Fibonacci numbers.
- Before you look at the code on the next page, try writing this function yourself.

```
/* This program prints out a Fibonacci series */
#include <stdio.h>
   /* function prototypes */
   long fib (long num);
int main (void)
{
     /* local definitions */
     int spacer;
     int seriesSize;

     /* statements */
     printf (“This program prints a Fibonacci series.\n”);
     printf (“How many numbers do you want in the series? ”);
     scanf (“%d”, &seriesSize);
     if (seriesSize <2)
          seriesSize = 2;

     printf (“First %d Fibonacci numbers are: \n”, seriesSize);
```


```
    for (spacer = 0; spacer < seriesSize; spacer++)
    {
        if (spacer % 5)
             printf (“ , %8ld”, fib(spacer));
        else
            printf (“\n%8ld”, fib(spacer));
    }
    printf (“\n”);
    return 0;
} /*end main */

/* function to calculate fibonacci numbers */
long fib (long num)
{
    if (num == 0  || num == 1)
         return num;  /*base case */
    else
        return (fib(num-1) + fib(num-2));
} /*end fib */
```


Limitations to Recursion

- All other things being equal, a recursive solution to a problem will require more time to execute than an iterative solution.
- The reason this is true is due to the overhead required to manage all of the recursive function calls. The system must maintain the "state" of your program by placing activation records on the run-time stack. An activation record contains a copy of the value of all local variables, parameters, and the return address of where program control will return when the function terminates.
- For programs the require very deep recursion (many recursive function calls) it is possible to exhaust the memory allocation before a final solution is achieved.

Limitations to Recursion (cont.)

- Consider the function we've just looked at for producing Fibonacci numbers. How many recursive calls are required to produce Fibonacci(5)?

```
fib(5) = fib(4) + fib(3)
    fib(4) = fib(3) + fib(2)
        fib(3) = fib(2) + fib(1)
            fib(2) = fib(1) + fib(0)
                fib(1) = 1
               fib(0) = 0
            fib(1) = 1
        fib(2) = fib(1) + fib(0)
            fib(1) = 1
           fib(0) = 0
    fib(3) = fib(2) + fib(1)
        fib(2) = fib(1) + fib(0)
            fib(1) = 1
           fib(0) = 0
        fib(1) = 1
```

recursive calls shown in blue

15 calls, 14 recursive + original call

Limitations to Recursion (cont.)

- The implementation of the Fibonacci series is highly inefficient.
- Look at the amount of redundant work that is performed in the calculation of Fibonacci(5).
  - fib(5) is called 1 time
  - fib(4) is called 1 time
  - fib(3) is called 2 times
  - fib(2) is called 3 times
  - fib(1) is called 5 times
  - fib(0) is called 3 times

Limitations to Recursion (cont.)

Series Size	Total Number of Calls
1	1
2	3
3	5
4	9
5	15
10	177
15	1,973
20	21,891
25	242,785
30	2,692,573
35	29,860,703
40	331,160,281

Table showing the number of calls required to calculate different length Fibonacci series

Limitations to Recursion (cont.)

- Does this mean that iterative solutions are always better than recursive solutions?

Answer: No. Many algorithms are easier to implement and maintain if they are written recursively. Recursive algorithms can be quite efficient as well. We'll look at a modified version of the Fibonacci function beginning on the next page that is much more efficient than the version we've already seen. Many data structures require recursive solutions.

Limitations to Recursion (cont.)

- You may be tempted after the previous discussion of the inefficiency of our recursive solution to the Fibonacci series to claim that all recursive functions are inefficient. However, you will be incorrect if you take this position.
- The problem with our first Fibonacci function is not with the recursion per se, but rather how the recursion is used.
- If we adopt a different recursive strategy, we can make the inefficiencies of the earlier function disappear.

Limitations to Recursion (cont.)

- As is often the case when dealing with recursion, the key to finding a more efficient solution lies in adopting a more general approach.
- In the previous algorithm, we directly implemented a recursive solution based on a recursive definition of the Fibonacci series.
- The Fibonacci series in not the only sequence whose terms are defined by the recurrence relation: $t_n = t_{n-1} + t_{n-2}$

Limitations to Recursion (cont.)

- Depending on how you select the first two terms in this series, you can generate many different sequences.
- The Fibonacci sequence is generated when the first two terms are 0 and 1.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …
- If you were to select the first two terms as 2 and 4 the sequence you would generate would be:
  - 2, 4, 6, 10, 16, 26, 42, 68, 110, 178, 288, 466,…

Limitations to Recursion (cont.)

- You can generate an extremely large number of sequences (infinite!) using this same recurrence relation.
- The general class of sequences which follow this pattern are called additive sequences.
- Using the concept of an additive sequence makes it possible to convert the problem of finding the n^{th} term in the Fibonacci sequence into the more general problem of finding the n^{th} term in an additive sequence whose first two terms are t_0 and t_1.

Limitations to Recursion (cont.)

- An additive sequence function requires three arguments: the number of terms of interest in the series, and the first two terms in the series.
- The C prototype for this function might look like the following:

```
int AdditiveSequence(int n, int t0, int t1);
```

Limitations to Recursion (cont.)

- Given such a function, generating the Fibonacci series would be straight forward as shown below:

```
int fib(int n)
{
   return (AdditiveSequence(n, 0, 1));
}
```

- The body consists of a single line of code that does nothing but call another function, passing along the additional arguments.
- Functions of this sort, are called wrapper functions. Wrapper functions are very common in recursive programming.

Limitations to Recursion (cont.)

- A implementation in C of the AdditiveSequence is given below:

```
int AdditiveSequence (int n, int t0, int t1))
{
   if (n == 0)   return (t0);
   if (n == 1)   return (t1);
   return AdditiveSequence (n-1, t1, t0+t1));
}
```

- Be sure you understand why this function properly determines the elements in an additive sequence.

Limitations to Recursion (cont.)

- Using this AdditiveSequence function, let's determine the value of Fibonacci(6).

fib(6)
= AdditiveSequence (6, 0, 1)
= AdditiveSequence (5, 1, 1)
= AdditiveSequence (4, 1, 2)
= AdditiveSequence (3, 2, 3)
= AdditiveSequence (2, 3, 5)
= AdditiveSequence (1, 5, 8)
= 8

- Notice how much more efficiently the recursion occurs using the AdditiveSequence function. The table on the next page illustrates this further.

Limitations to Recursion (cont.)

Series Size	Total Number of Calls	
	Original Function	New Function
1	1	2
2	3	3
3	5	4
4	9	5
5	15	6
10	177	11
15	1,973	16
20	21,891	21
25	242,785	26
30	2,692,573	31
35	29,860,703	36
40	331,160,281	41

Table showing the number of calls required to calculate different length Fibonacci series

A Classic Problem in Recursion

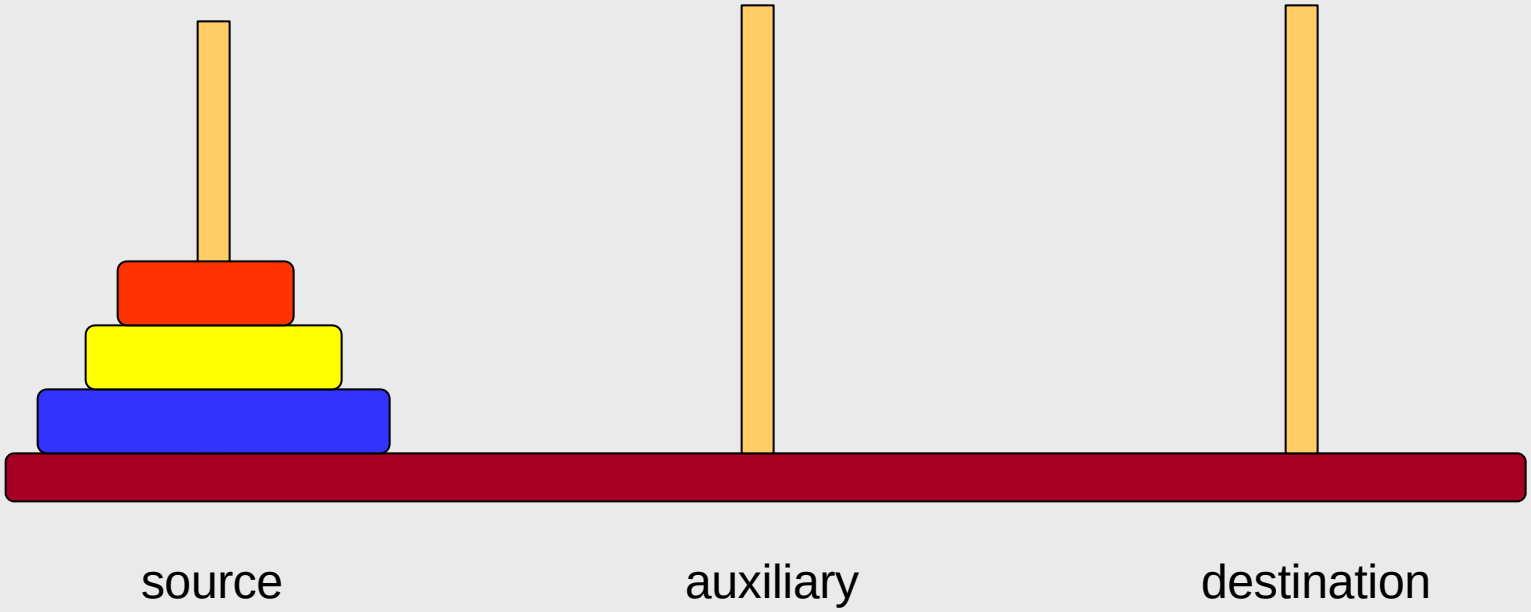
- The Towers of Hanoi is a classic problem in recursion.
- According to legend, the monks in a remote mountain monastery knew how to predict when the world would end. They had three diamond needles. Stacked on the first diamond needle were 64 gold disks of decreasing size. The monks moved one disk to another needle every hour according to the following rules:

1. Only one disk could be moved at a time.
2. A larger disk must never be stacked above a smaller one.
3. One and only one auxiliary needle could be used for the intermediate storage of disks.

A Classic Problem in Recursion (cont.)

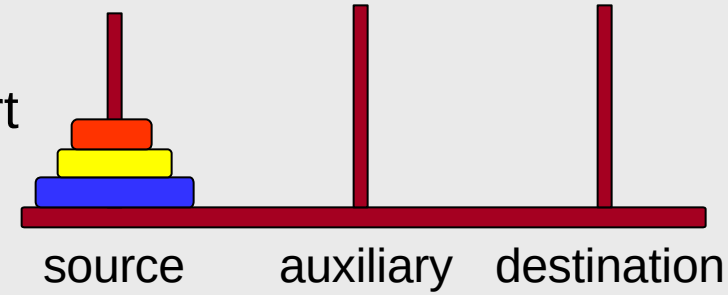
- The legend says that when all 64 disks had been transferred to the destination needle, the stars would be extinguished and the world would end.
- In other words, the world would end when $2^{64} - 1$ moves have occurred. (By the way, this will take a total of 2.1×10^{15} years to accomplish. The age of the universe is estimated at 4.6×10^{9} years! So we're safe for a while longer.)
- Since we have less time than the monks we'll examine this problem assuming there are only three disks and that the world will not end when we have completed their transfer to the destination needle.

A Simplified Towers of Hanoi Problem

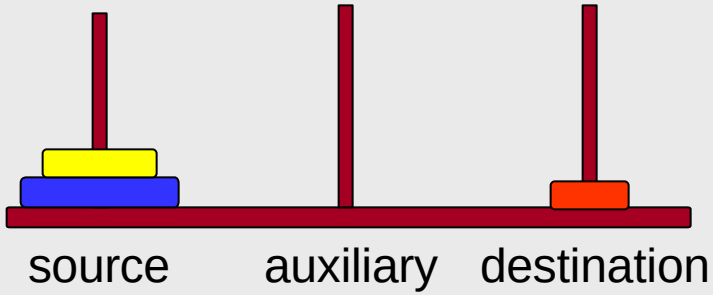


A Solution to the Towers of Hanoi Problem w/ 3 Disks

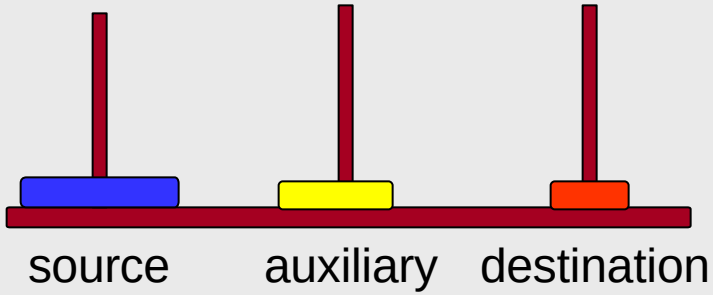
start



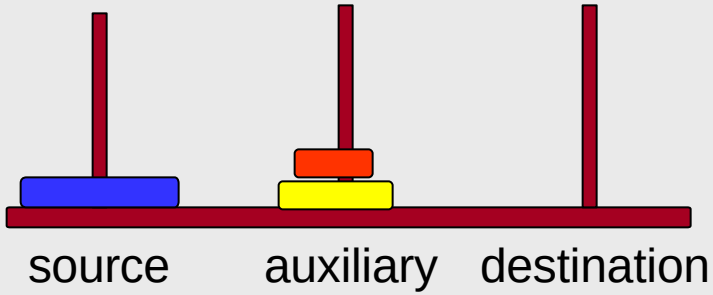
1



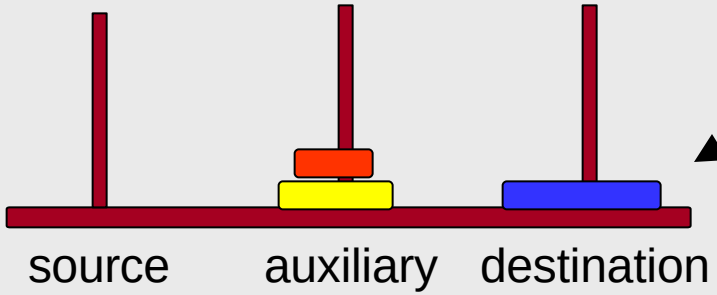
2



3



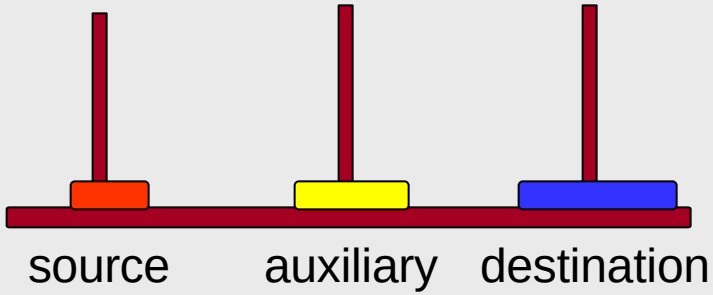
4



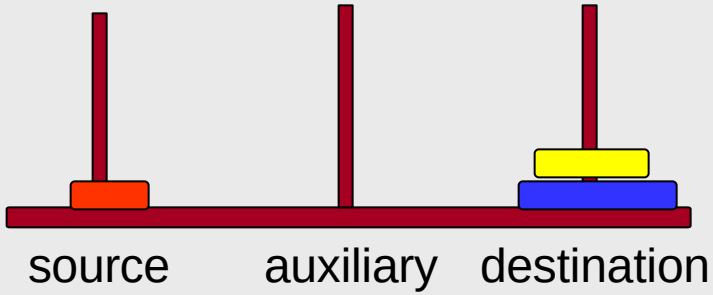
One disk has been moved from source to destination after a sequence of 4 moves.

A Solution to the Towers of Hanoi Problem w/ 3 Disks

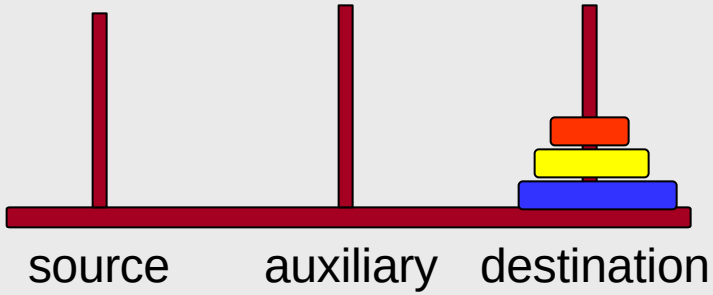
5



6



7



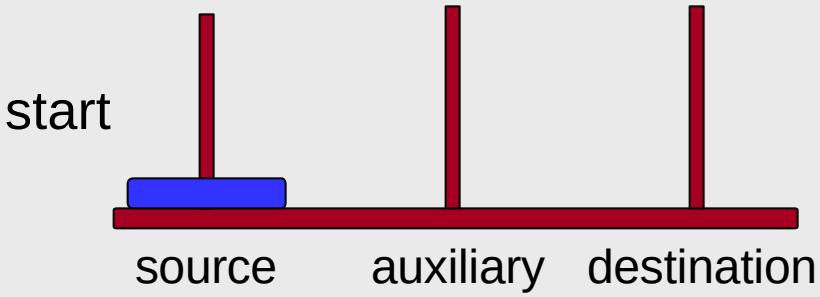
Task complete, all disks moved to destination needle. Total sequence required 7 (2^3-1) moves.

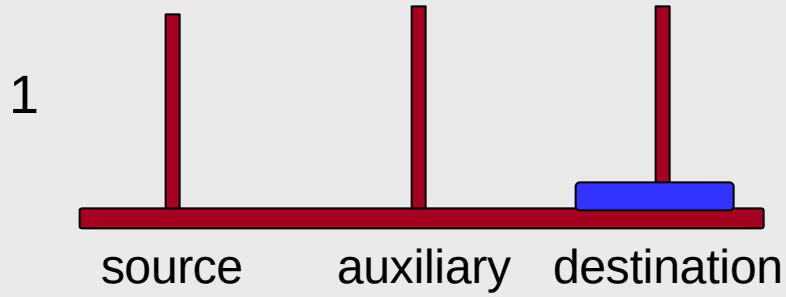
Analyzing the Recursion in the Problem

- Analyze the moves that were made to solve the simplified Towers of Hanoi problem. Can you find the pattern?
- If we are to solve this problem recursively we will need to identify the base case as well as the general case.
- In order to help identify the patterns, let's look at an even simpler form of the problem.

Analyzing the Problem (cont.)

- Let's suppose that we have only one disk.

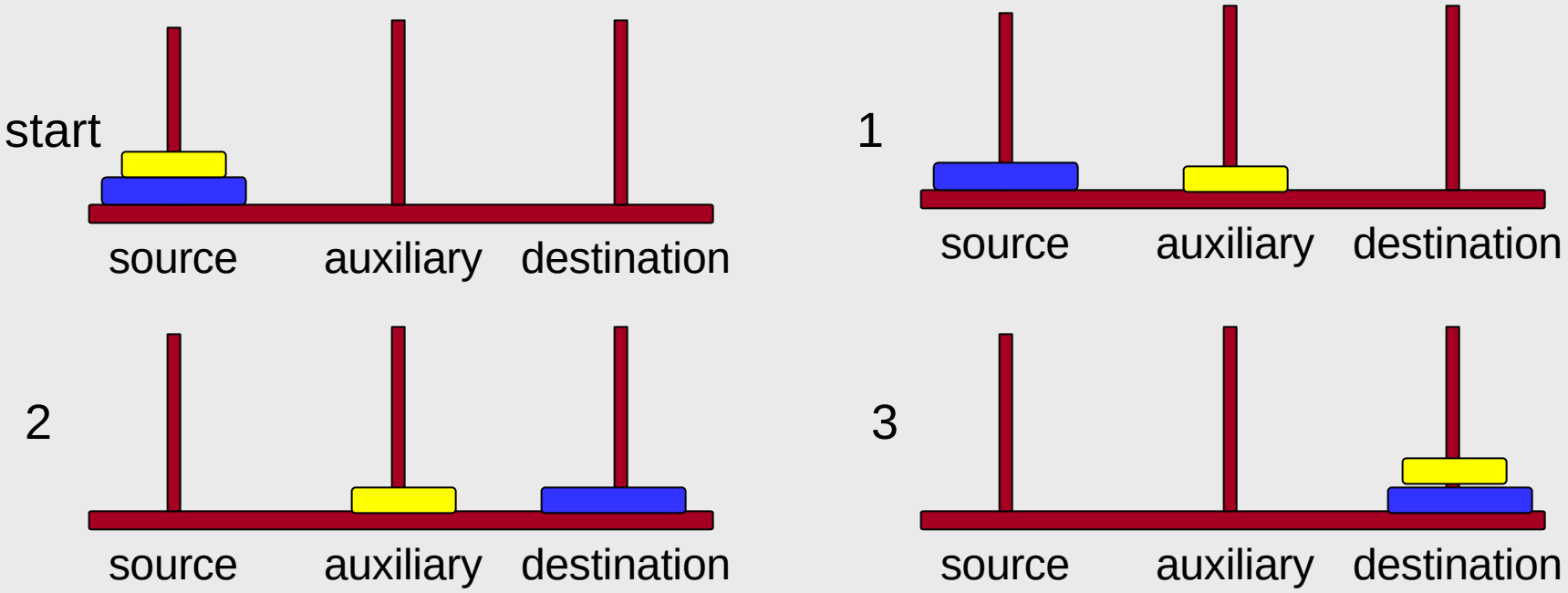




Case 1: Move one disk from source to destination needle.

Analyzing the Problem (cont.)

- Now let's suppose that we have two disks.



Case 2: Move one disk from source to auxiliary needle.
Move one disk from source to destination needle.
Move one disk from auxiliary to destination needle.

Analyzing the Problem (cont.)

- Now go back and look at how we solved the case with three disks.
- All total, we moved 2 disks from source to auxiliary, 1 disk from source to destination, and 2 disks from auxiliary to destination.

Case 3: Move two disks from source to auxiliary needle.
Move one disk from source to destination needle.
Move two disks from auxiliary to destination needle.

Analyzing the Problem (cont.)

- Hopefully, by now you begin to see the pattern in solving this problem.
- To generalize the problem we have the following:

1: Move n-1 disks from source to auxiliary needle. **(general case)**

2: Move one disk from source to destination needle. **(base case)**

3: Move n-1 disks from auxiliary to destination needle. (**general case)**

Analyzing the Problem (cont.)

- Unlike the factorial and Fibonacci problems we saw earlier, the Towers of Hanoi problem has two general cases (i.e., two recursive cases) to solve and not just one.
- This will translate into two different recursive calls when we construct our algorithm and its implementation in C.
- The solution that we'll develop will construct a function with four parameters: the number of disks to be moved, the source needle, the destination needle, and the auxiliary needle.
- In pseudocode the three cases are:

1. Call Towers (n-1, source, auxiliary, destination)
2. Move one disk from source to destination
3. Call Towers (n-1, auxiliary, destination, source)

Analyzing the Problem (cont.)

- Notice the third step: After completing the move of the first disk, the remaining disks are on the auxiliary needle. We need to move them from the auxiliary needle to the destination needle. In this case, the original source needle becomes the auxiliary needle.

- The next page shows a C function to solve the Towers of Hanoi problem. Before looking at this code in great detail, try to write the function yourself.

```
/* Solve Towers of Hanoi Problem                                    */
/* Assume: n disks with source, destination, & auxiliary needles given */
void towers (int n, char source, char dest, char auxiliary)
{
    /* local definitions */
    static int step = 0;
    /* statements */
    printf("Towers (%d, %c, %c, %c) \n", n, source, dest, auxiliary);
    if (n == 1)
        printf ("\t\t\t\tStep %3d: Move from %c to %c\n", ++step, source, dest);
    else
    {    towers (n-1,  source, auxiliary, dest);
         printf ("\t\t\t\tStep %3d: Move from %c to %c\n", ++step, source, dest);
         towers (n-1, auxiliary, dest, source);
     } /*end else */
    return;
} /*end towers */
```


Solution for 3 Disk Towers of Hanoi Problem

Calls:	**Output:**
Towers (3, S, D, A)	
Towers (2, S, A, D)	
Towers (1, S, D, A)	
	Step 1: Move from S to D
	Step 2: Move from S to A
Towers (1, D, A, S)	
	Step 3: Move from D to A
	Step 4: Move from S to D
Towers (2, A, D, S)	
Towers (1, A, S, D)	
	Step 5: Move from A to S
	Step 6: Move from A to D
Towers (1, S, D, A)	
	Step 7: Move from S to D