CS 113
DISCRETE STRUCTURES

Chapter 3: Algorithms

HOMEWORK

- **All homework is from the Exercises**
  - **No problems are from the Review Exercises**
- **Section 3.2 (p. 127): 19-21 (all)**
- **Section 3.3 (p. 131): 1-5 (all), 21-24 (all)**
- **Section 3.4 (p. 137): 8, 9, 12, 13, 27, 28**
- **Section 3.5 (p. 149): 1-12 (all), 16-25**
- **Section 3.7 (p. 160): 1-9 (all)**

ALGORITHMS

- Chapter 3 is about algorithms
- An algorithm is a finite sequence of steps where
  - The steps are precisely stated
  - The intermediate results of each step are uniquely defined
    - They depend only on the inputs and the results of the previous steps
  - The algorithm always stops (in a finite number of steps)
  - The algorithm applies to a set of inputs
  - The algorithm receives input
  - The algorithm produces output

PSEUDOCODE VS. C++

- The book uses a well-defined, strict set of pseudocode instructions
- We will follow their lead
- Some notable differences from C++ are noted in the table on the next slide
- Pseudocode doesn’t use semi-colons to end lines
    - Instead, it assumes the end of the line is like a semi-colon
- It also doesn’t use parentheses around conditions
- Also, the “for” statement can only count by 1
    - In addition, you can only count up, not down

PSEUDOCODE VS. C++: HIGHLIGHTING SOME DIFFERENCES

C++	Pseudocode
=	:=
void some_function	Procedure some_function
if (---)	If --- then (Parentheses are unnecessary)
{	begin (Most of the time)
}	end
while (---)	while --- do (Parentheses are unnecessary)
for (i=0; i<10; i++)	for i := 0 to 9 do
m++	m := m + 1
%	mod
!	not

AN EXAMPLE-ALGORITHM 3.2.2 ON P. 124

C++	Pseudocode
int max (int a, int b, int c)	procedure max (a, b, c)
{	
x = a;	x := a
if (b>x)	if b > x then
x = b;	x := b
if (c>x)	if c > x then
x = c;	x := c
return x;	return x
}	**end** max

THE END OF SECTION 3.2

- **Homework is due for Section 3.2**

- **The homework is**
- **Section 3.2 (p. 127): 19-21 (all)**

THE DIVISION ALGORITHM

- **The division algorithm states that**
  - **For any two integers x and y, where y is not 0, you can find two other integers q (the quotient) and r (the remainder) with**

    $$x=qy + r \quad \text{with } 0 \leq r < y$$

- **Back in our C++ days, we would have noted that x % y = r   x / y = q**

- **Some things to know**
  - **If x divides into y with no remainder, we write $x \mid y$**
  - **If there is a remainder in the division, we write $x \nmid y$**
  - **Also, if x divides into y with no remainder, we will write that y = kx for some integer k**
    - **We also say that there exists an integer k with y = kx**

USING THAT LAST IDEA

- **For example, if 6 divides into a number, then 3 divides into that number**
- **Formally, this is**

  If 6 | n, then 3 | n.
- **Proof:**

  Suppose that 6 | n.
  Then n = 6k for some integer k.
  Then n = 3(2k), where 2k is also an integer.
  So 3 | n.

- **From now on, assume that all letters stand for integers**

THREE IDEAS FROM THE BOOK

- **This is Theorem 3.3.4, p. 129**
- **The theorem has three parts**
- **Part 1:**

  If $c \mid m$ and $c \mid n$, then $c \mid m + n$.
- **Part 2:**

  If $c \mid m$ and $c \mid n$, then $c \mid m - n$.
- **Part 3:**

  If $c \mid m$, then $c \mid mn$ for any n.
- **The proofs of these three statements are very direct**

THE END OF SECTION 3.3

- **Homework is due for Section 3.3**

- **The homework is**
- **Section 3.3 (p. 131): 1-5 (all), 21-24 (all)**

REVISITING RECURSION

- We saw recursion in C++
- A recursive procedure is one that calls itself
- Why would you want to create a procedure like that?
  - It might be easier to program
- A good example is the math function x!
  - Here, factorial (x) [programming notation] denotes x! [math notation]

8! means 8x7x6x5x4x3x2x1

9! means 9x8x7x6x5x4x3x2x1

10! means 10x9x8x7x6x5x4x3x2x1

CHECKING OUT THE FACTORIAL FUNCTION

- **Just like before we notice that the end of 10! is 9!**
  - **And the end of 9! is 8!**
    - **It looks like 9! is just 9 x 8!**
  - **We notice that 8! is just 8 x7! too**
  - **And we keep going**
- **So, we can say n! = n x (n-1)!**
- **We can write**

```
procedure factorial (n)
    return n * factorial (n-1)
end factorial
```

- **Here are the other examples that we talked about in C++**

ANOTHER EXAMPLE: FIBONACCI NUMBERS

- **The Fibonacci Numbers are a list (sequence) of numbers**
  - **The first two Fibonacci numbers are both 1**
  - **After that, to get the next Fibonacci number, add the two before it**
- **For example, the first seven numbers in the list are**

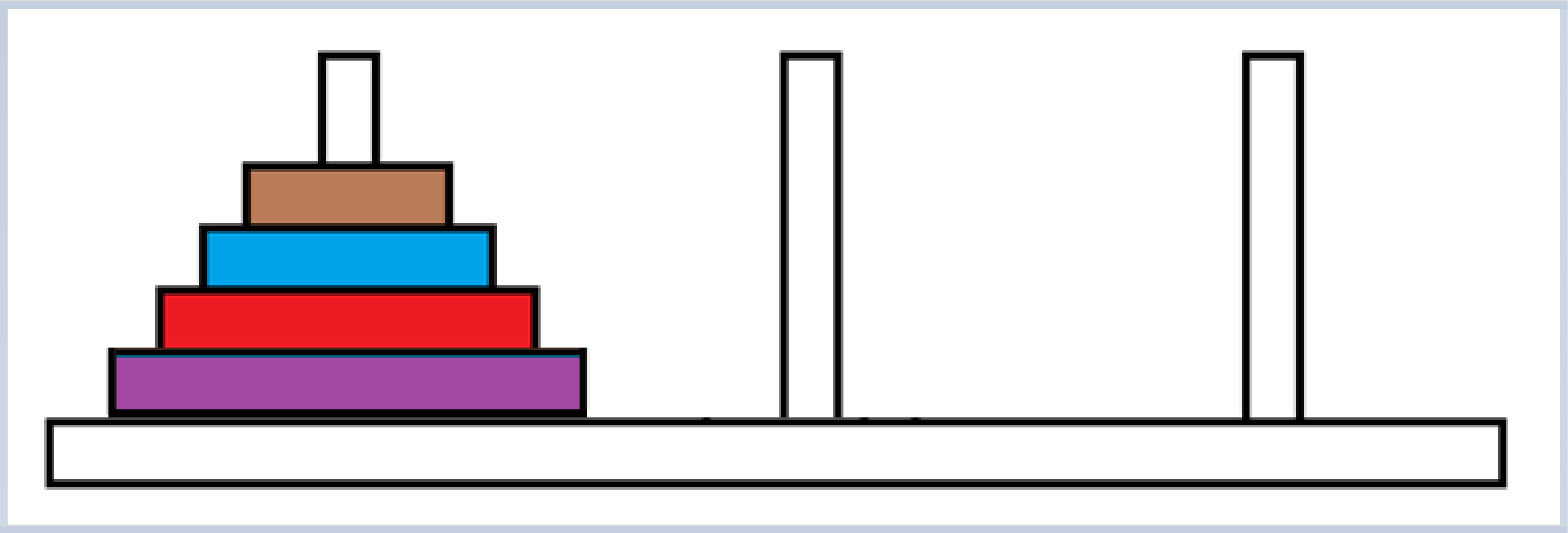
  1, 1, 2, 3, 5, 8, 13
- **This is recursive!**
- **The code is (mostly)**

```
procedure fibonacci (n)
    return fibonacci (n-1) + fibonacci (n-2)
end fibonacci
```

- **Try to write that code without recursion**

A THIRD EXAMPLE: THE TOWERS OF HANOI

- **This problem is not as mathematical as the first two**
- **It provides an excellent example of the value of recursion**
- **Here is a picture of the towers**



MOVING THE DISKS

- There are several disks on each tower
- In this case, there are four disks
  - I have colored them purple, red, blue, and brown
- Your job is to move them to the other end
- This seems simple enough

THE RULES

- **There are only two rules**
  1. **You can only move one disk at a time**
  2. **You cannot put a disk on top of a smaller disk**
- **This is much harder than it looks!**

THE SOLUTION USING RECURSION

- **Suppose there are ten disks**
- **Then, to move all ten disks to the other end**
  - **Move the top nine disks to the middle post**
    - **Really, you are just setting aside the top 9 disks**
  - **Move the remaining disk to the end post**
  - **Move the nine disks from the middle post to the end post**
    - **This puts those disks on top of the disk you just moved**
- **Wow! We have moved the disks**
- **Notice that this solution is recursive**
  - **Try to see how to do that without recursion**
  - **It is really tough**

RECURSION IN GENERAL

- **Recursion means**
  - **You turn the problem into a smaller version of the same problem**
  - **Then you call the same function to complete the solution**
- **In recursion, a function calls itself**
  - **Of course, when it calls itself, it has to be solving a smaller problem**
  - **Something like   factorial (n) = factorial (n)   is not useful**

THE CODE

- **So the pseudocode is (mostly)**

```
procedure move_disks (n, start_pole, end_pole,  spare_pole)
    move_disks (n-1, start_pole, spare_pole, end_pole)
    Move top disk from start_pole to end_pole
    move_disks (n-1, spare_pole, end_pole, start_pole)
end move_disks
```

A PROBLEM WITH RECURSION

- **Let's go back to factorials**
- **Let's calculate 3!**
- **Using the code, 3! = 3 x 2!**

  =3 x (2 x 1!)

  =3 x (2 x (1 x 0!))

  =3 x (2 x (1 x (0 x (-1)!)))
- **Does this ever stop? No!**

A STOPPING POINT

- **For recursive code, we <u>always</u> need to include a stopping condition**
  - **This is called the base case**
- **The actual pseudocode for factorials is**

```
procedure factorial (n)
    if n = 1 then
        return 1
    else
        return n * factorial (n-1)
end factorial
```

A DISADVANTAGE OF RECURSION

- **Recursion is much slower than solving the problem directly**
- **However, programming the direct solution might be a lot harder**
- **This is a trade-off to consider when using recursion**
  - **The usual decision is to use recursion if it's appropriate**
- **Coding the factorial function directly (with a for loop) is easier and more efficient**
  - **We always use the direct method when we can**
  - **This is called iteration because it uses a loop**

FACTORIAL VS. FACTORIAL

A recursive version	**A non-recursive (iterative) version**

<pre>**procedure factorial (n)**
  if n = 1 then
    return 1
  else
    return (n * factorial (n-1))
end factorial</pre>
	<pre>**procedure factorial (n)**
	  product = 1
	  for i = 1 to n
	    product = product * i
	  return (product)
	end factorial</pre>

THE FINBONACCI SEQUENCE

- The “opposite” of recursion is iteration
  - Usually this involves a loop
  - It may involve an array or a stack
- From before, the Fibonacci sequence was an example of recursion

  $$F_1 = 1,\ F_2 = 1,\ F_n = F_{n-1} + F_{n-2}$$

- This gives a clear demonstration of how recursion can be incredibly inefficient
- To check this, write out the steps needed to calculate F_4

CHECKING ON HOW RECURSION WORKS

- **See the examples**
- **fib-counts.cpp (Shows the inefficiency of Fibonacci recursion)**
- **fib-iter.cpp (Shows reiterative Fibonacci sequence)**
- **fib-recur.cpp (Typical recursive version of Fibonacci numbers)**

QUESTIONS

- Any questions?