Data Structures and Algorithms 120

Lecture 3: Recursion

Department of Computing

Curtin University

Curtin University of Technology

Last updated: 20 August 2014

Copyright Warning

This Week

– Introduction to recursion
– Simple recursive methods
– Recursion and the call stack
– Recursive algorithm design

What is Recursion?

- Recursion is where a problem is stated in terms of simpler versions of the same problem
  - A type of repetition (iteration) where the solution is arrived at by having the method repeatedly call itself to solve a simpler form
    - This self-calling continues ('iterates') until the simplest version of the problem is reached
- Some problems are much more easily solved with a recursive approach than an iterative one

Example 1: Factorial

- No error checking in either approach!
- Formal Mathematical definition:
  - N! : 0! = 1 otherwise N * (N-1)!
- Iterative solution using for loop from OOPD110

```
public long calcNFactorial( int n ) {
   long nFactorial = 1;
   for ( int ii = n; ii >= 2; ii-- )
      nFactorial *= ii;
   return nFactorial;
}
```

- Recursive solution is to solve (N-1)! and multiply by N

```
public long calcNFactorialRecursive( int n ) {
   if ( n == 0 )
      return 1;
   else
      return n * calcNFactorialRecursive( n-1 );
}
```

← Simplest case (the 'base case')

← oops – multiple returns!

← Recursive call multiplied by n, n changed to go towards base case

Removing multiple return statements

– Usually just add a local variable and set it to the return value.

```
public long calcNFactorialRecursive( int n )
{
   long factorial = 1;                                    ← declare and initialise local variable

   if ( n == 0 )                                          ← Simplest case (the 'base case')
      factorial = 1;                                      ← no more multiple returns!
   else
      factorial = n * calcNFactorialRecursive( n-1 );    ← Recursive call
   return factorial;                                      ← return local variable
   }
```

Error checking

– Throw an exception if the import is invalid

```
public long calcNFactorialRecursive( int n )
{
   long factorial = 1;                                         ← declare and initialise local variable
   if ( n < 0 )                                                ← error condition
     throw new IllegalArgumentException(“Import must not be negative");
   else if ( n == 0 )                                          ← Simplest case (the ‘base case’)
      factorial = 1;                                           ← no more multiple returns!
   else
      factorial = n * calcNFactorialRecursive( n-1 );     ← Recursive call
   return factorial;                                           ← return local variable
   }
```

Necessary Properties

– The factorial example highlights all the necessary properties of a recursive algorithm:

1) Decomposable into a simpler version of the same form
   - » N! is easy to calculate if (N-1)! is known
2) Simplest (base) case exists (and is not recursive)
   - » When n=0, no factorial is needed – just return 1
   - » The base case is the terminating condition of the recursion
     - – Thus every recursive method has an 'if' check for base case(s)
3) Base case must be reached
   - » This requires a parameter (eg: n) that MUST be changed during every recursive call (changing the value towards the base case)
   - » Otherwise the recursion will never end!

General Structure of a Recursive Algorithm

```
METHOD RecursiveAlg
IMPORT algorithm-specific-parameters
EXPORT result

Algorithm
   IF terminating_condition_1 THEN
     result ← base_case_1                   ← Usually a simple one-liner, but doesn't have to be
   ELSEIF terminating_condition_2 THEN
     result ← base_case_2
   ELSEIF                                   ← There can be any number of base cases
      ...
   ELSE
     reduce_problem_using_recursion         ← This could be one line or a whole sub-system
     result ← results_of_reduction
   ENDIF
```

Example 2: Fibonacci Number

- Calculate the Nth Fibonacci number
  - » Sequence: 0 1 1 2 3 5 8 13 21 34
  - » Mathematical Definition
  - » FIB 0 = 0, FIB 1 = 1, FIB N = FIB (N-1) + FIB (N-2)
- Iterative solution:

```
public int fibIterative(int n) {
   int fibVal = 0;                       ← value n
   int currVal = 1;                      ← value n-1
   int lastVal = 0;                      ← value n-2

   if (n == 0)
      fibVal = 0;
   else if (n == 1)
      fibVal = 1;
   else {
      for (int ii = 2; ii < n; ii++) {
         fibVal = currVal + lastVal;
         lastVal = currVal;              ← set up lastVal and currVal ready for next iteration
         currVal = fibVal;
      }
   }
   return fibVal;
}
```

Example 2: Fibonacci Sequence

– Recursive solution:

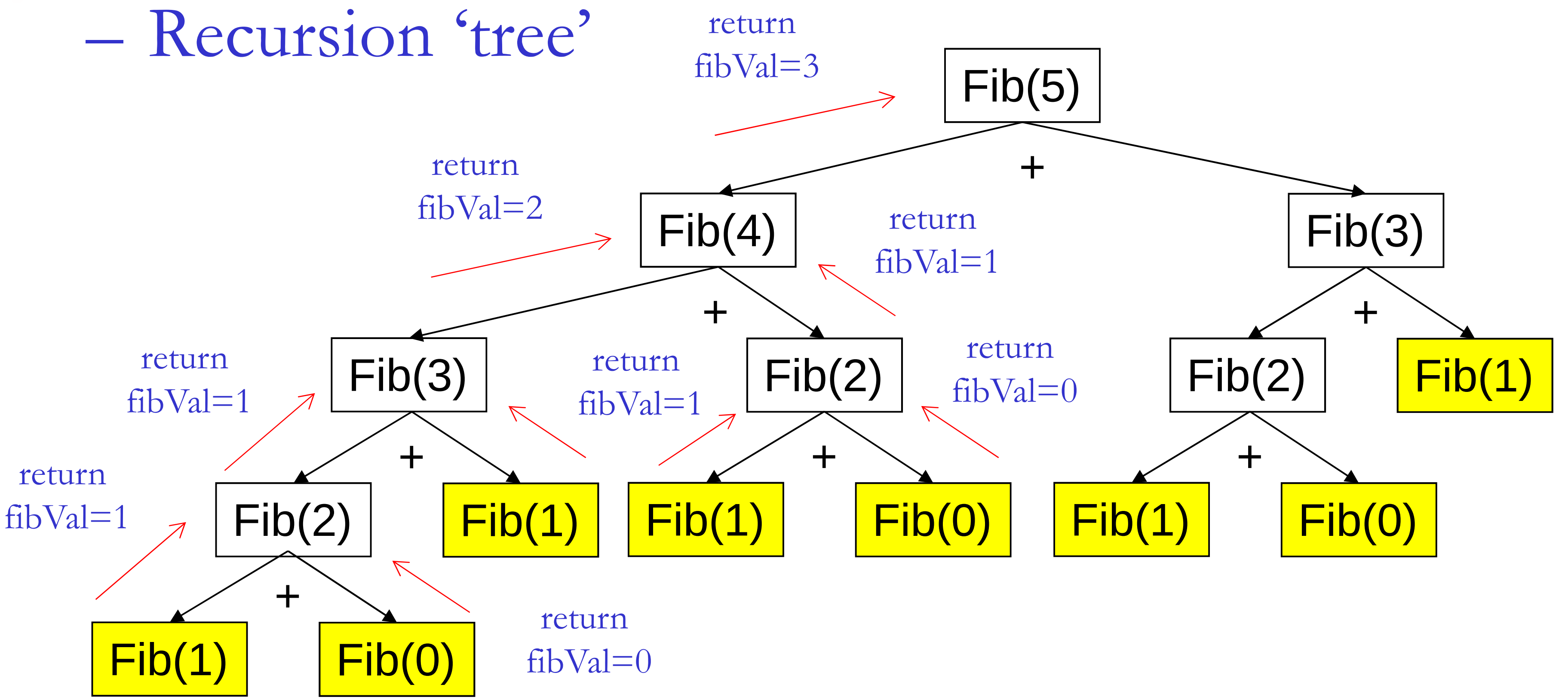
```
public int fibRecursive(int n) {
   int fibVal = 0;

   if (n == 0)
      fibVal = 0;                                         ← Base case #1
   else if (n == 1)
      fibVal = 1;                                         ← Base case #2
   else {
      fibVal = fibRecursive(n-1) + fibRecursive(n-2);     ← Two recursive calls
   }
   return fibVal;
}
```

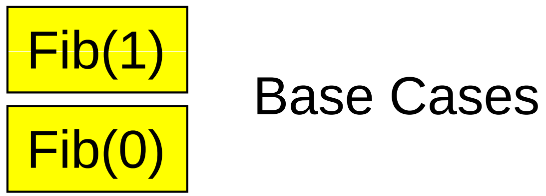
Compare with Mathematical Definition

FIB 0 = 0, FIB 1 = 1, FIB N = FIB (N-1) + FIB (N-2)

A Closer Look at Recursive Fibonacci

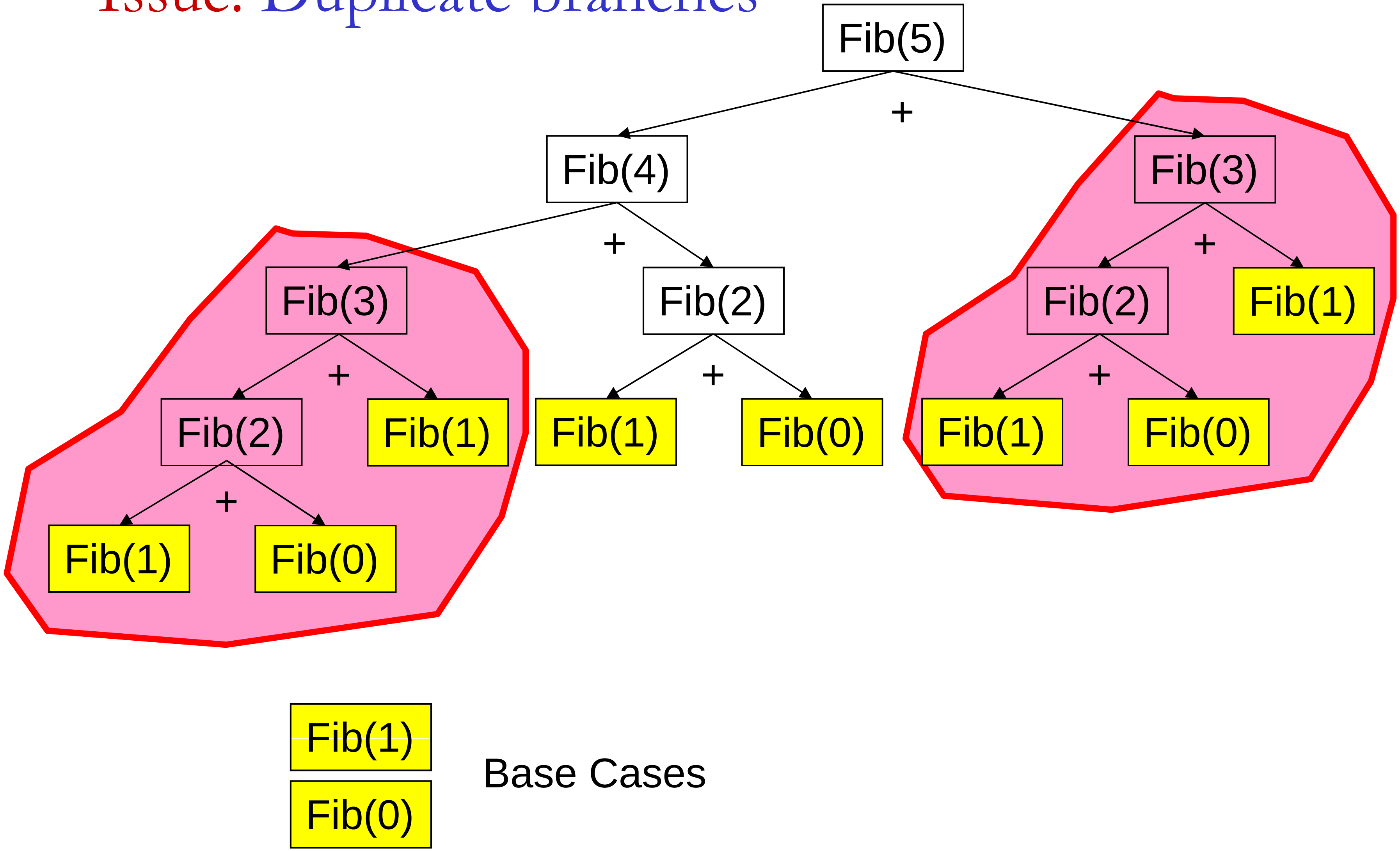
– Recursion 'tree'





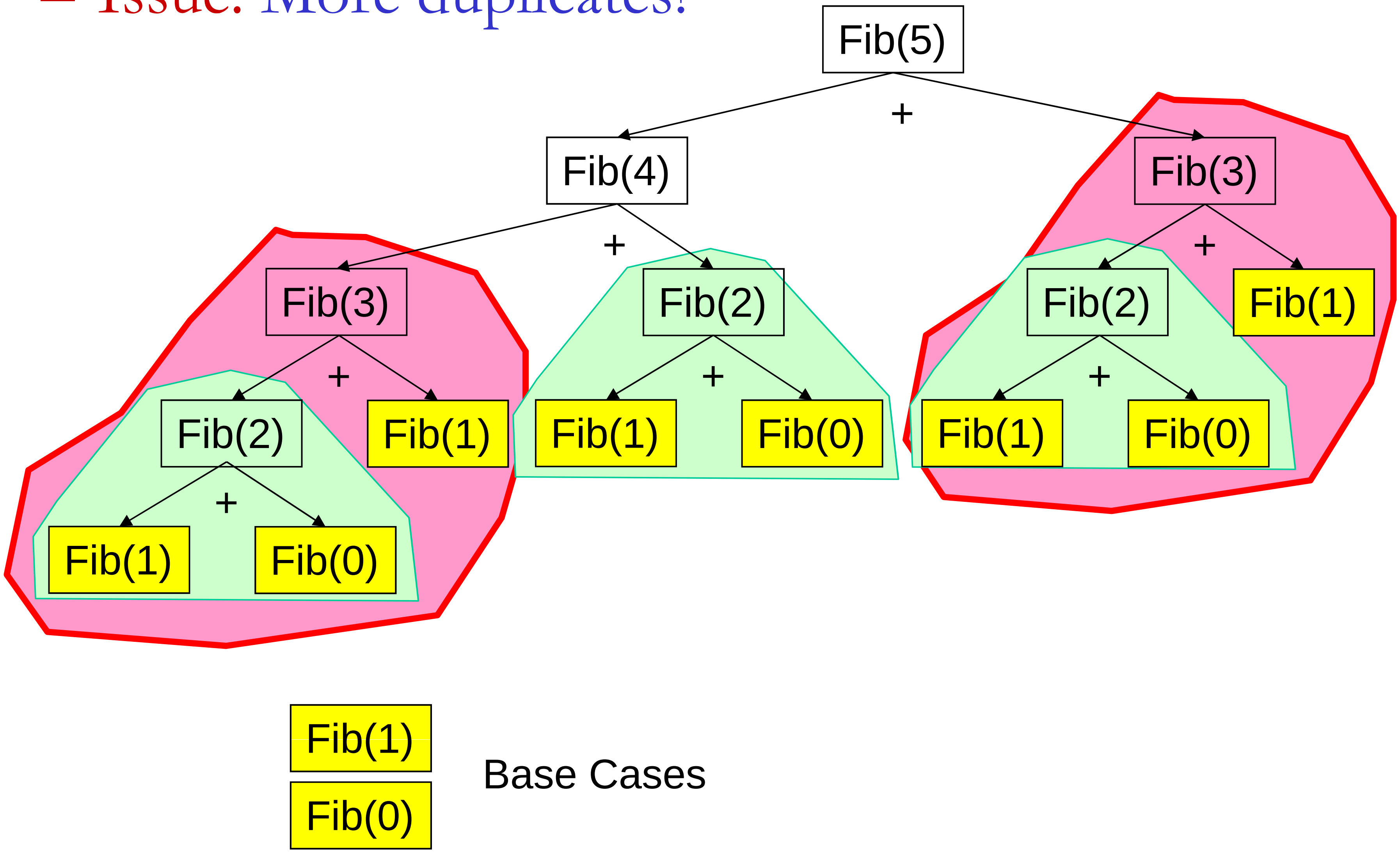
A Closer Look at Recursive Fibonacci

– Issue: Duplicate branches



A Closer Look at Recursive Fibonacci

– Issue: More duplicates!



Example 2: Fibonacci Sequence

- The recursive solution creates a recursive 'tree'
  - Due to the solution requiring *two* recursive calls
  - Thus smaller Fib(n) values are calculated *multiple* times
- Not a great example of the power of recursion
  - It is much slower than the iterative method due to duplicate calculations and function call overheads!
    - » try both with fib (200)
- But it does show how a mathematical definition is easily translated into a recursive method.

Example 3: Keyboard to Int

- When you are typing numbers at the keyboard, you are actually typing characters!
- How does the program convert this to an integer?
- Think about the sequence of input
  - » Most significant digit first
    - – which power of 10 to apply
- Two ways to solve the problem
  - » with a stack
    - – covered in lecture 4
  - » recursively
- real numbers are different – not covered

Example 3: Keyboard to Int

- Design
  - 47519 = 4*10000 + 7*1000 + 5*100 + 1*10 + 9
- But we are reading left to right
  - At each level the value is the next digit plus ten times the value at the previous level!
    - » How do we convert a character to it's decimal equivalent?
  - Implies that the value at any level must be somehow passed as a parameter to the next level of recursion.
  - Must assume initial call passes the value 0.
  - When we get to the highest level of recursion we know the total value of the integer.
  - Final value can be returned as recursion unwinds.

Example 3: Keyboard to Int

```
private int recReadToInt(int valSoFar)
{
   char nextChar;
   int value = 0, digit = 0;

   nextChar = readChar();                      ← Get the next character from keyboard

   if (nextChar >= '0' && nextChar <='9')      ← If we're dealing with a valid digit
   {                                             convert to integer and recurse
      digit = (int)nextChar - (int)'0';        ← See the ascii table
      value = recReadToInt(valSoFar * 10 + digit);
   }
   else
      value = valSoFar;                        ← Base case, no more input

   return value;
 }
```

Example 3: Keyboard to Int

```
private char readChar()
{
   char nextCh='\0';

   try {
      nextCh = (char) System.in.read();
   }
   catch (IOException e)
   {
      System.out.println(e.getMessage());
   }

   return nextCh;
}
```

← String termination character or null char
Needed because try might fail

Wrapper methods

- The Keyboard to Integer method ASSUMES an initial value of zero is imported.
- So we mark `recReadToInt()` as private.

```
private int recReadToInt(int valSoFar)
```

- We need a public method to make sure import is 0

```
public int readToInt()
{
        return recReadToInt(0);
}
```

The Importance of Terminating

- Consider the following recursive method:

```
public int endless(int n) {
   endless(n+1);
   return n;
}
```

  - Obviously, the method has no terminating condition
    - » In fact, the statement "`return n;`" is never reached
  - So what happens? An infinite loop?
    - » Actually, no – it results in a crash!
    - » Specifically, a StackOverflowException is thrown by Java
- The crash occurs as a consequence of how method calls are performed: → limits on the number of calls
  - Most (all?) modern languages have this issue

Call Stack

- Whenever a new method is called, it is necessary to store certain information related to the call
  - The method's local variables
  - A copy of the method's parameters
    - » Note: An object reference is copied, but *not the object itself*
  - Bookkeeping info, such as address (in code) to return to
    - » So that when the method finishes, the program knows where to return to in the *calling* method
- Since methods are dealt with in Last-In-First-Out (LIFO) order, this information is placed on a special *stack* in memory
- This is the Call Stack or Process Stack

Call Stack

- For each method call:
  - A stack frame is pushed onto the call stack
    - Stack frame contains local vars, params and bookkeeping info
    - If the method calls another method, it too pushes a stack frame
  - When the method returns, the stack frame is popped off the call stack
- Most (all?) languages reserve a fixed amount of memory for the call stack, limiting its max size
  - Why? Because it makes method calls very efficient – no need to check to dynamically grow/shrink the stack

Stack Overflow

- However, a recursive algorithm with hundreds of ‘iterations’ means hundreds of stack frames
  - Call stack is usually limited to around 10Mb or so
  - Thus a few hundred recursive calls could lead to the program running out of stack space
    - Faster if local vars and params take up significant space
  - If the call stack runs out of space, the program has no choice but to crash
    - Java fails with a stack overflow exception
    - C/C++ fails with a stack fault

Stack Memory vs Heap Memory

- The call stack places a limit on how many iterations a recursive algorithm can do before stack overflow
  - Exactly how many depends on the size of each stack frame
    - » …which depends on the number and size of local vars & params
  - In Java, only references and primitives exist on the stack
    - » Objects are *always* allocated on what is called the heap
      - … the heap is essentially all other memory besides the stack
    - » Heap memory is dynamically allocated using the new keyword
      - Please note: we are talking about heap *memory*, not the heap *ADT*, a totally different concept that we will explore in a later lecture
  - In C/C++ *anything* can be allocated on the stack
    - » Even objects or huge arrays – can make stack space run out fast!

Example 4: Towers of Hanoi

– Towers of Hanoi is an ancient game where a pile of disks must be moved from one peg to another

– Rules:

  - The disks can only be moved one at a time
  - A larger disk cannot be placed on top of a smaller disk

– This is a good example of where recursion is particularly useful

  - The intricate disk-shuffling turns out to be based on a surprisingly simple recursive definition

Towers of Hanoi - Algorithm

```
public void towers(int n, int src, int dest) {
   int tmp;

   if (n == 1)
      moveDisk(src, dest);
   else {
      tmp = 6 – src - dest;
      towers(n-1, src, tmp);

      moveDisk(src, dest);
      towers(n-1, tmp, dest);
   }
}
```

← Move n disks from peg src to peg dest

← Base case: move one disk from peg src to peg dest

← tmp is the 'other' (non-target) peg, since src+dest+tmp = 6

← Move all but bottom disk to temp peg tmp
This is a smaller (n-1) version of the current problem

← Move bottom disk to target peg dest

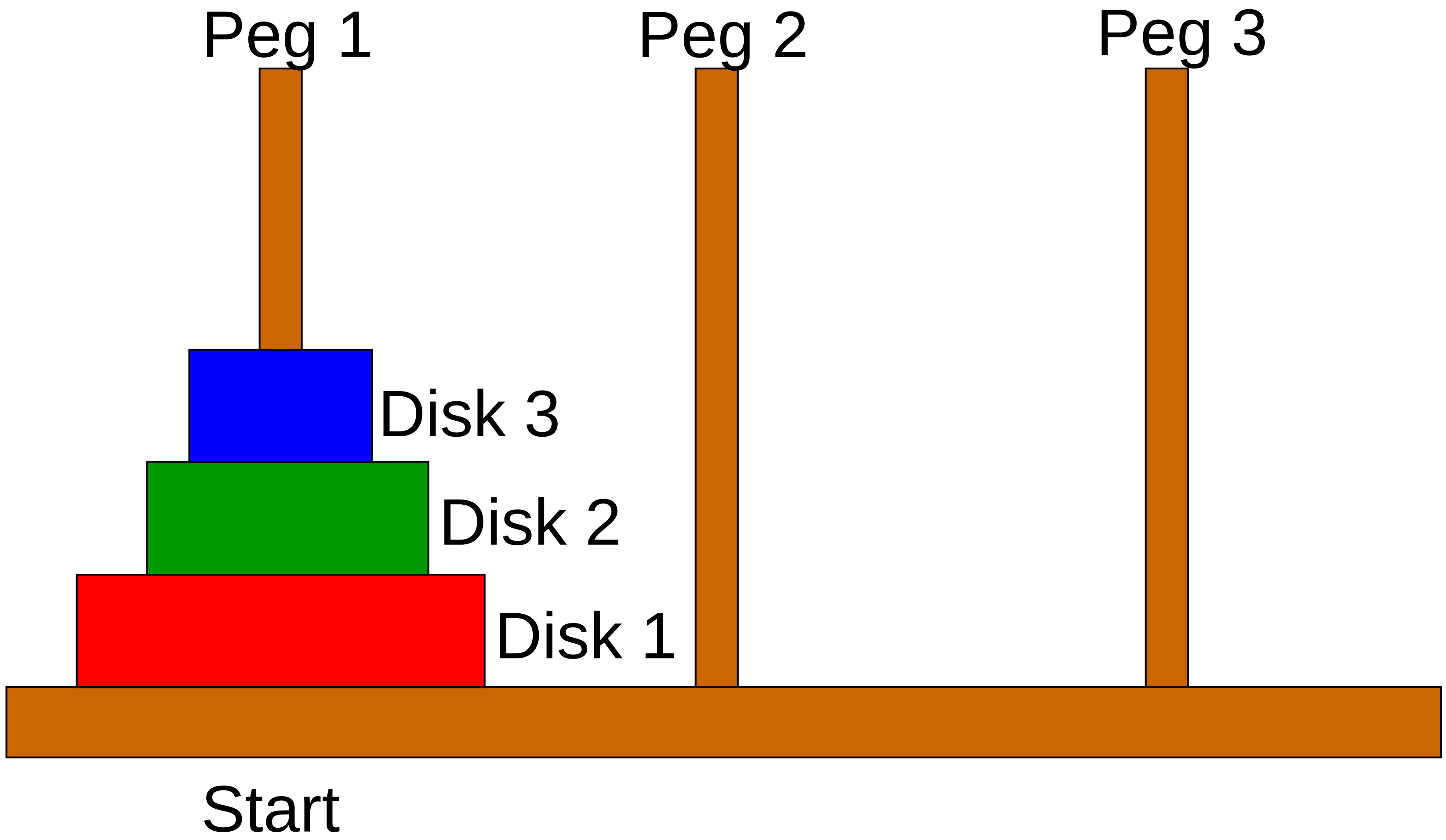
← Move the rest from temp peg tmp to target peg dest

– Note: tmp keeps changing during every recurse
  - It makes sure we choose the right temp peg at each recurse so that in the end the bottom disk will be put on the target peg dest

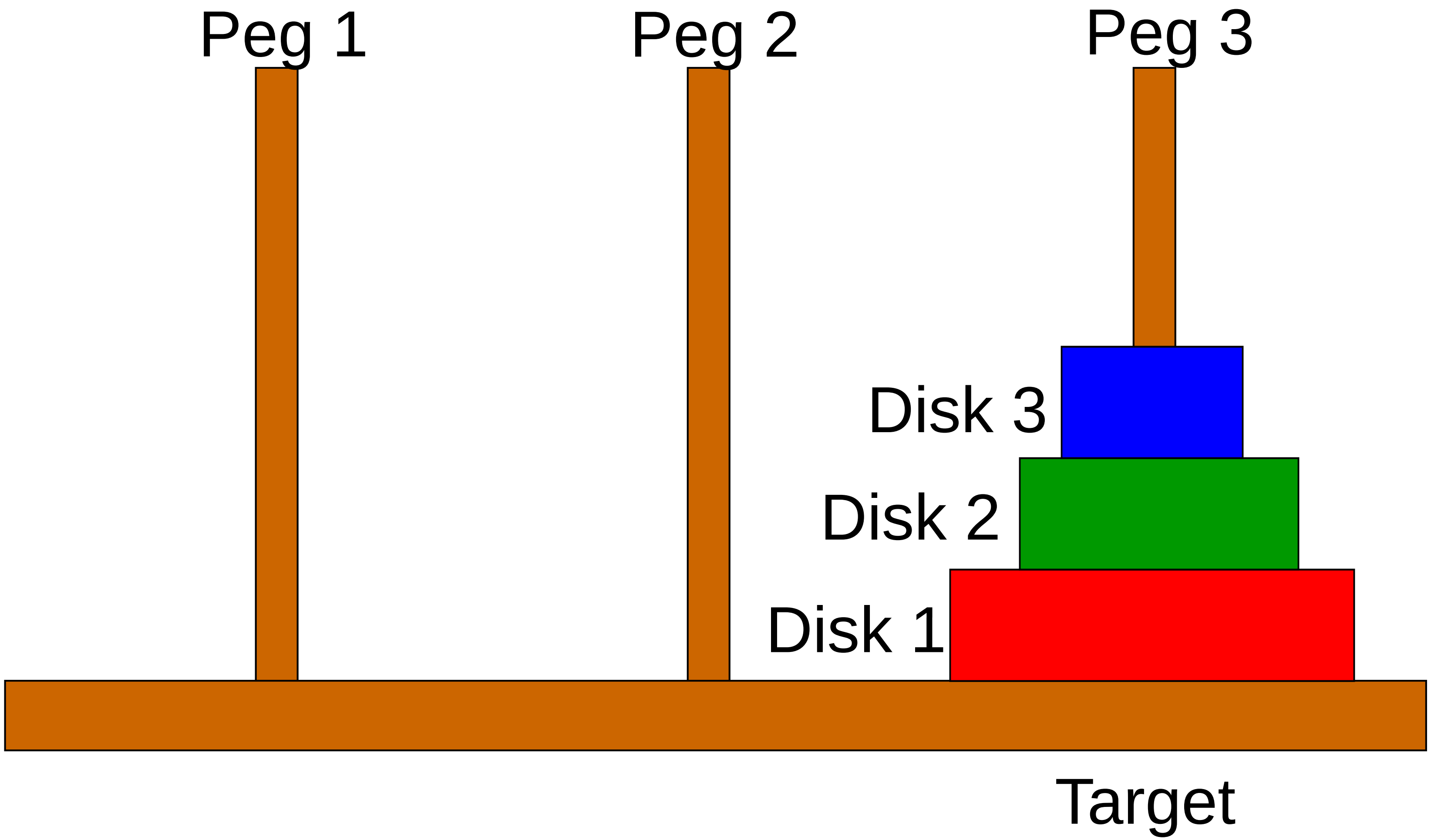
Towers of Hanoi – Step-by-Step

- Over the next few slides we will be stepping through the algorithm, explicitly showing the state of the call stack at each step
- First let's define the starting state and the target state that we want to end up in

Initial State



Target State



Stepping Through the Algorithm

towers(3, 1, 3)

tmp = 6-1-3 = 2

towers(2, 1, 2)

tmp = 6-1-2 = 3

towers(1, 1, 3)

moveDisk(1,2)

towers(1,3,2)

towers(1, 1, 3)

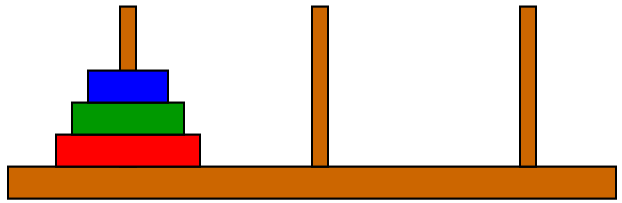
towers(2, 2, 3)

tmp = 6-2-3 = 1

towers(1, 2, 1)

moveDisk(2, 3)

towers(1, 1, 3)



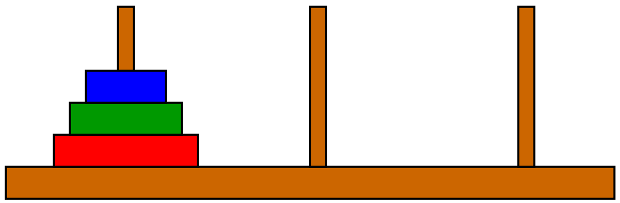
Recursion Loop

```
if (n ==1)
    moveDisk(src, dest);
else {
    tmp = 6 - src - dest;
    towers(n - 1, src, tmp);
    moveDisk(src, dest);
    towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)
    tmp = 6-1-3 = 2
    towers(2, 1, 2)
        tmp = 6-1-2 = 3
        towers(1, 1, 3)
        moveDisk(1,2)
        towers(1,3,2)
    towers(1, 1, 3)
    towers(2, 2, 3)
        tmp = 6-2-3 = 1
        towers(1, 2, 1)
        moveDisk(2, 3)
        towers(1, 1, 3)

No change

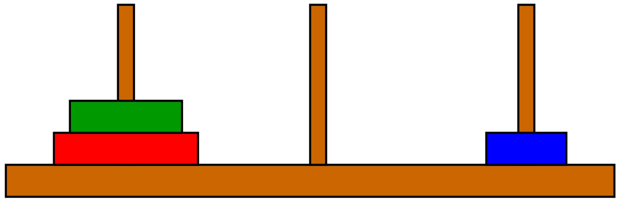


```
Recursion Loop
if (n ==1)
     moveDisk(src, dest);
else {
     tmp = 6 - src - dest;
     towers(n - 1, src, tmp);
     moveDisk(src, dest);
     towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)

    tmp = 6-1-3 = 2
    towers(2, 1, 2)
        tmp = 6-1-2 = 3
        towers(1, 1, 3)
        moveDisk(1,2)
        towers(1,3, 2)
    towers(1, 1, 3)
    towers(2, 2, 3)
        tmp = 6-2-3 = 1
        towers(1, 2, 1)
        moveDisk(2, 3)
        towers(1, 1, 3)

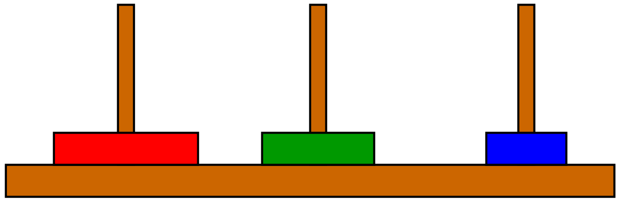


Recursion Loop

```
if (n ==1)
      moveDisk(src, dest);
else {
      tmp = 6 - src - dest;
      towers(n - 1, src, tmp);
      moveDisk (src, dest);
      towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)
- tmp = 6-1-3 = 2
- towers(2, 1, 2)
  - tmp = 6-1-2 = 3
  - towers(1, 1, 3)
  - **moveDisk(1, 2)**
  - towers(1,3, 2)
- towers(1, 1, 3)
- towers(2, 2, 3)
  - tmp = 6-2-3 = 1
  - towers(1, 2, 1)
  - moveDisk(2, 3)
  - towers(1, 1, 3)

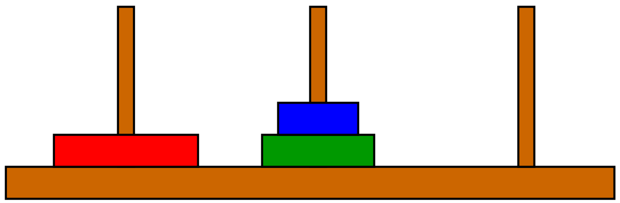


Recursion Loop

```
if (n ==1)
      moveDisk(src, dest);
else {
      tmp = 6 - src - dest;
      towers(n - 1, src, tmp);
      moveDisk (src, dest);
      towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)
- tmp = 6-1-3 = 2
- towers(2, 1, 2)
  - tmp = 6-1-2 = 3
  - towers(1, 1, 3)
  - moveDisk(1,2)
  - ☆ **towers(1,3, 2)**
- towers(1, 1, 3)
- towers(2, 2, 3)
  - tmp = 6-2-3 = 1
  - towers(1, 2, 1)
  - moveDisk(2, 3)
  - towers(1, 1, 3)



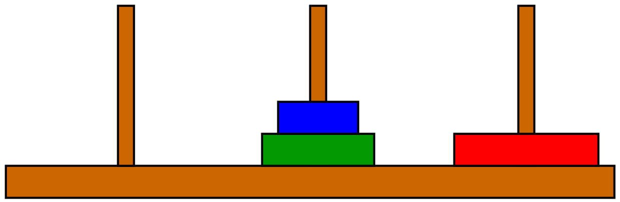
Recursion Loop

```
if (n ==1)
      moveDisk(src, dest);
else {
      tmp = 6 - src - dest;
      towers(n - 1, src, tmp);
      moveDisk (src, dest);
      towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)

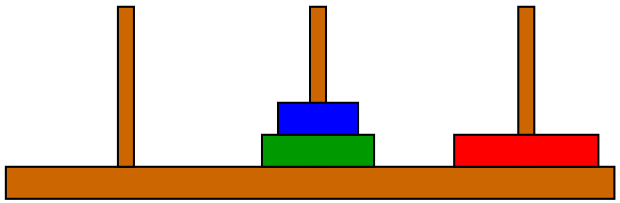
    tmp = 6-1-3 = 2
    towers(2, 1, 2)
        tmp = 6-1-2 = 3
        towers(1, 1, 3)
        moveDisk(1,2)
        towers(1,3,2)
    towers(1, 1, 3)
    towers(2, 2, 3)
        tmp = 6-2-3 = 1
        towers(1, 2, 1)
        moveDisk(2, 3)
        towers(1, 1, 3)



```
Recursion Loop
if (n ==1)
    moveDisk(src, dest);
else {
    tmp = 6 - src - dest;
    towers(n - 1, src, tmp);
    moveDisk( src, dest);
    towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)
  tmp = 6-1-3 = 2
  towers(2, 1, 2)
    tmp = 6-1-2 = 3
    towers(1, 1, 3)
    moveDisk(1,2)
    towers(1,3, 2)
  towers(1, 1, 3)
  towers(2, 2, 3)
    tmp = 6-2-3 = 1
    towers(1, 2, 1)
    moveDisk(2, 3)
    towers(1, 1, 3)



Recursion Loop

```
if (n ==1)
      moveDisk(src, dest);
else {
      tmp = 6 - src - dest;
      towers(n - 1, src, tmp);
      moveDisk(src, dest);
      towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)

  tmp = 6-1-3 = 2

  towers(2, 1, 2)

    tmp = 6-1-2 = 3

    towers(1, 1, 3)

    moveDisk(1,2)

    towers(1,3,2)

  towers(1, 1, 3)

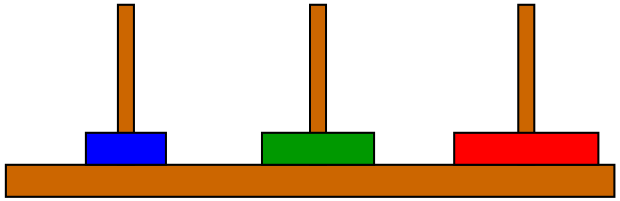
  towers(2, 2, 3)

    tmp = 6-2-3 = 1

    towers(1, 2, 1)

    moveDisk(2, 3)

    towers(1, 1, 3)



```
Recursion Loop
if (n ==1)
     moveDisk(src, dest);
else {
     tmp = 6 - src - dest;
     towers(n - 1, src, tmp);
     moveDisk(src, dest);
     towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)

  tmp = 6-1-3 = 2

  towers(2, 1, 2)

    tmp = 6-1-2 = 3

    towers(1, 1, 3)

    moveDisk(1,2)

    towers(1,3,2)

  towers(1, 1, 3)

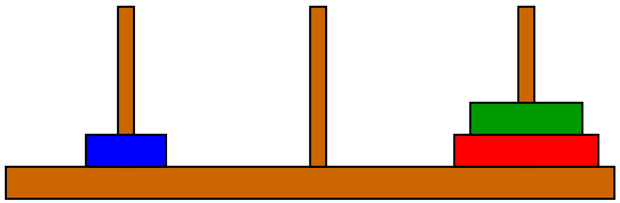
  towers(2, 2, 3)

    tmp = 6-2-3 = 1

    towers(1, 2, 1)

    moveDisk(2, 3)

    towers(1, 1, 3)



```
Recursion Loop
if (n ==1)
    moveDisk(src, dest);
else {
    tmp = 6 - src - dest;
    towers(n - 1, src, tmp);
    moveDisk(src, dest);
    towers(n-1, tmp, dest);
}
```

Stepping Through the Algorithm

towers(3, 1, 3)

  tmp = 6-1-3 = 2

  towers(2, 1, 2)

    tmp = 6-1-2 = 3

    towers(1, 1, 3)

    moveDisk(1,2)

    towers(1,3,2)

  towers(1, 1, 3)

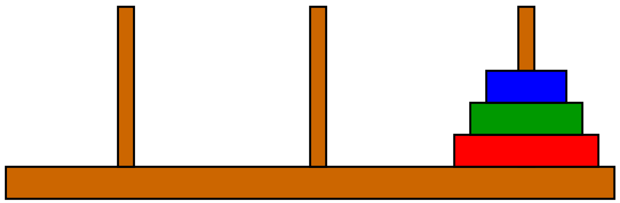
  towers(2, 2, 3)

    tmp = 6-2-3 = 1

    towers(1, 2, 1)

    towers(1, 2, 3)

    towers(1, 1, 3)



```
Recursion Loop
if (n ==1)
     moveDisk(src, dest);
else {
     tmp = 6 - src - dest;
     towers(n - 1, src, tmp);
     moveDisk(src, dest);
     towers(n-1, tmp, dest);
}
```

Recursion vs Iteration

– *Any* recursive algorithm can be re-written with an iterative (looping) solution
  - Some problems just need a different approach
    - » eg: Iterative fib(): just 'cache' the last two fib values
  - Other problems need to emulate the call stack
    - » Store data from previous iterations onto a stack ADT
      - – … just like the call stack does for params/local vars in recursion
    - » With a stack ADT (allocated on the heap), call stack limitations and stack overflows are less of an issue
      - – Although now you must maintain the stack yourself

Recursion vs Iteration

- Advantages of a recursive solution:
  - ☑ Some algorithms are *much* simpler using recursion
- Disadvantages of recursion vs iteration
  - ☒ The call stack limits the number of recursive 'iterations' that can be performed
    - » No more than a few thousand iterations before stack overflow
  - ☒ Usually slower due to method call overhead
    - » Every time a method is called, a few instructions are needed to set up the method call (eg: allocate space for local vars, etc)
    - » For small recursive methods (a few lines or less), this call overhead will become a significant factor of the processing

When to use recursion?

– When the algorithm is considerably simpler than the iterative version

- and the overheads of method calls are inconsequential
  - » towers of hanoi
  - » merge sort
  - » quick sort
  - » parsing binary trees
- and when there is little chance of a stack overflow

Next Week

- arrays
- stacks
- queues