COMP1002 DATA STRUCTURES AND ALGORITHMS

LECTURE 2: RECURSION, WRAPPERS AND EXCEPTIONS

Curtin University
Curtin Computing

Last updated: 2 March 2020

Copyright Warning

Objectives

- Exceptions
  - Discuss and apply exceptions for developing robust code
- Wrappers
  - Discuss and apply wrapper (helper) methods to insulate code
- Recursion
  - Introduce Recursion and some simple recursive methods
  - Discuss Recursion and the call stack
  - Explain Recursive algorithm design

EXCEPTION HANDLING

Errors and Exceptions

- Errors or mistakes in a program are often referred to as bugs
- They are almost always the fault of the programmer
- The process of finding and eliminating errors is called debugging
- Errors can be classified into three major groups:
  - Syntax errors
  - Runtime errors
  - Logical errors

http://python-textbok.readthedocs.io/en/1.0/Errors_and_Exceptions.html

Syntax Errors

- Python/Java will find these kinds of errors when it tries to parse your program
  - Java: compilation errors
  - Python: exit with an error message (without running anything)
- Syntax errors are mistakes in the use of the language, and are like spelling or grammar mistakes
- Common syntax errors include:
  - leaving out a keyword
  - putting a keyword in the wrong place
  - leaving out a symbol, such as a colon, comma or brackets
  - misspelling a keyword
  - incorrect indentation
  - empty block

Runtime Errors

- If a program is syntactically correct – that is, free of syntax errors – it can be run
- Some problems are only revealed when a particular line is executed
- When a program comes to a halt because of a runtime error, we say that it has crashed
- Some examples of runtime errors:
  - division by zero
  - performing an operation on incompatible types
  - using an identifier which is not defined (Python - @compile-time in Java)
  - accessing a list element, dictionary value or object attribute which doesn't exist
  - trying to access a file which doesn't exist

Logical Errors

- Logical errors are the most difficult to fix
- They occur when the program runs without crashing, but produces an incorrect result
- The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred.
- Here are some examples of mistakes which lead to logical errors:
  - using the wrong variable name
  - indenting/bracketing a block to the wrong level
  - using integer division instead of floating-point division (Python)
  - getting operator precedence wrong
  - making a mistake in a boolean expression
  - off-by-one, and other numerical errors

Handling Errors

- Error handling is a necessary task, but how can you do it elegantly?
  - Errors aren't 'normal' - you don't make a system that *expects* errors! But you must handle error situations
  - One solution: return an error code. Used in C programs

Exceptions

- O-O languages solve error handling with **exceptions**
  - An independent 'return path' designed specifically for notifying the caller of an exceptional situation (=error)
  - On an error, a method 'throws' an exception
  - The calling method can 'catch' the exception
    - If caller doesn't catch it, the exception is thrown to the next-higher caller
    - If no-one catches it, the exception causes the program to crash

Exceptions (Java)

- Java only lets objects of type Exception to be thrown
  - (actually, Exception inherits from interface Throwable, but Exception is the typical starting point for most programmers)
  - Java has a range of classes inheriting from Exception
    - eg: IllegalArgumentException, ArrayIndexOutOfBoundsException
  - You can define your own exception class, as long as it inherits from Exception (or one of it's subclasses)
    - C# is similar. C++ let's you throw pretty much anything (!!)
- Each exception should require a message (a String)
  - Why throw an error if you don't say what the problem is?
  - also use getMessage() or getLocalizedMessage()
    - may or may not be detailed enough

Exception Types (Java)

- Here are exception types relating to File I/O

- java.lang.Object
- java.lang.Throwable
- java.lang.Exception
- java.io.IOException
- java.io.FileNotFoundException

Checked vs Unchecked Exceptions (Java)

- Unchecked Exceptions
  - code will compile if not caught anywhere
  - RuntimeException, Error, and their subclasses
- Checked Exceptions
  - code will NOT compile if not caught somewhere
  - all other Exceptions
  - can use throws in method header to catch in calling method
    - see multiple catch example coming up

Exceptions (Python)

- Python only lets objects of type **Exception** or its descendants to be thrown
- Python has a range of classes descending (inheriting, extends) from Exception
  - eg: ValueError, ZeroDivisionError
- You can define your own exception class, as long as it inherits from Exception (or one of it's subclasses)
- Each exception should require a message (a String)
  - Why throw an error if you don't say what the problem is?

Exception Types (Python)

- Here are a few common exception types which we are likely to raise in our own code:
  - **TypeError**: this is an error which indicates that a variable has the wrong *type* for some operation.
    We might raise it in a function if a parameter is not of a type that we know how to handle.
  - **ValueError**: this error is used to indicate that a variable has the right *type* but the wrong *value*.
    For example, we used it when age was an integer, but the wrong *kind* of integer.
  - **NotImplementedError**: we will see in the next chapter how we use this exception to indicate that a class's method has to be implemented in a child class.

Writing your own exceptions (Java)

- inherit from Exception
  - this will make it a checked Exception

```
public class MyException extends Exception
{
    private String error;

    public MyException(String message)
    {
        super(message);
        this.error="Something bad Happened";
    }

    public MyException(String message, String error)
    {
        super(message);
        this.error=error;
    }

    public String getError(){
        return this.error;
    }
}
```

Throwing Exceptions (Java)

- Java uses the throw keyword to throw exceptions
  - So do most other O-O languages

```
public double Divide(double numer, double denom) {
    if (denom == 0) {
        throw new IllegalArgumentException("denom must be non-zero");
    }

    return numer / denom;
}
```

- Note that we throw a *new* IllegalArgumentException
  - ie: We create an object and then throw it.
  - Could also have done the following (same result):

```
IllegalArgumentException e = new IllegalArgumentException("denom must be non-zero");
throw e;
```

Writing your own exceptions (Python)

- inherit from Exception
  - this will make it a checked Exception

```
class ExampleException (Exception):    # minimal definition of exception
    pass

class Error(Exception):
    pass                                          # a more elaborate definition

class StackOverflowError(Error):
    """Exception raised if stack is overfull.

    Attributes:
        message -- explanation of the error
    """
    def __init__(self, message):
        self.message = message

class StackUnderflowError(Error):
    """Exception raised if stack is underfull.

    Attributes:
        message -- explanation of the error
    """
    def __init__(self, message):
        self.message = message
```

Raising Exceptions (Python)

- Python uses the **raise** keyword to throw exceptions
  - FYI - Java uses "throw"

```
if (invalid):
    raise ValueError("invalid import");
```

- Note that we are creating an object and then throwing it = raise

Catching Exceptions

- Exceptions from different methods in different objects are often all caught at the one place in the calling method
  - Convenient: all error handling happens in one place
- Most languages use

  try .. **except** (catch) .. [**finally**] blocks:
  - **try**: define the set of statements whose exceptions will all be handled by the catch block associated with this try
  - **except**: processing to do if an exception is thrown in the try
  - **finally**: processing to always do regardless of whether an exception occurs or not.
    - Good for clean-up, eg: closing open files
    - This block is optional and executes after the try and catch blocks

Example (Java)

```
public static void Main() {
    double fFirst, fSecond, fSum;
    try {                                  ← Try the following few statements
      fFirst = Divide(10,1);
      fSecond = Divide(2,0);               ← Will cause an exception to be thrown
      fSum = fFirst + fSecond;
      System.out.println("Sum of the divides is: " + fSum);
    }
    catch (Exception e) {                  ← Catch any exception that occurs
       System.out.println("Couldn't perform divides. Reason: " + e.getMessage());
    }
    finally {
       fSum = 0;                           ← Pointless clean-up - this is just an example!
    }
}
```

- It's possible to *only* catch IllegalArgumentExceptions:

```
catch (IllegalArgumentException e) {
   System.out.println("Couldn't perform divides. Reason: " + e.getMessage());
}
```

  - … but then other exceptions won't be caught, and if they occur they will cause the program to crash

Example (Python)

```
try:
    dividend = int(input("Please enter the dividend: "))
except ValueError:
    print("The dividend has to be a number!")

try:
    divisor = int(input("Please enter the divisor: "))
except ValueError:
    print("The divisor has to be a number!")

try:
    print("%d / %d = %f" % (dividend, divisor, dividend/divisor))
except ZeroDivisionError:
    print("The dividend cannot be zero!")
```

Example (Python)

```
try:
    age = int(input("Please enter your age: "))
    if age < 0:
        raise ValueError(str(age) + " is not valid")

except ValueError as err:
    print("You entered incorrect age input:", err)

else:
    print("I see that you are", age, " years old.")

finally:
    print("It was really nice talking to you.\
           Goodbye!")
```

Error checks v's exception handling (Python)

```
# with checks
n = None
while n is None:
    s = input("Enter an integer: ")
    if s.lstrip('-').isdigit():
        n = int(s)
    else:
        print("%s is not an integer." % s)
```

```
# with exception handling
n = None
while n is None:
    try:
        s = input("Enter an integer: ")
        n = int(s)
    except ValueError:
        print("%s is not an integer." % s)
```

Exceptions vs Error Codes

- Advantages over error codes:
  - ☑ Separate return path for separate issues
  - ☑ Caller can deal with all exceptions in batch & in one place
    - Makes the code a lot easier to follow
  - ☑ Exceptions must be handled or the program will crash
    - Stops errors from accidentally going unnoticed
  - ☑ Lets a constructor raise an error just like any other method
    - Constructors don't have a return value!
- Disadvantages:
  - ☒ Need language support for throwing/catching exceptions

Multiple catch clauses

- previous example caught <u>all</u> exceptions
- what if we want to handle various exceptions in a different way?
  - *e.g.,* if a file doesn't exist, ask the user for a different filename, but if anything else goes wrong with the file, terminate the program.
- every try clause must be followed by one or more catch clauses
  - order is important!

Multiple catch example

```
private void readFileExample() throws IOException  {          ← Required as Checked Exception, must be caught in caller
   String inFilename;
   //file code omitted
   boolean noFile;
   do {
      noFile = true;                                          ← file doesn't exist yet,
      inFileName = ConsoleInput.readLine("Please enter the filename");
      try {
       //file code omitted - see lecture 2                    ← Open file – if it doesn't exist, FileNotFoundException
          lineNum = 0;
      //file code omitted                                     ← Read the first line
          while (line != null){                               ← While not end-of-file, process and read lines
             lineNum++;                                        ← IOException may be thrown
             processLine(line);                               ← Some other exception may be thrown here
          }
         noFile = false;                                      ← if we get here, everything worked fine
      }//end try
      catch (FileNotFoundException e){                        ← can catch sub of IOExceptions
         System.out.println("Couldn't find your file " + e.getMessage() + "try again!");
      }                                                       ← will continue loop
      catch (IOException e){                                  ← MUST catch checked IOExceptions
         throw e;
      }                                                       ← re throw IOException to exit method
      catch (Exception e){                                    ← can catch all other Exceptions
         throw new Exception("I am a bad programmer: " , e);
                                                              ← re throw Exception to exit method
      }
   } while(noFile);                                           ← If the file doesn't exist, repeat the loop
}
```

Order is important!

- Exceptions are part of a class hierarchy
- E.g. in Java:

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.io.IOException
                java.io.FileNotFoundException
```

- With inheritance, FileNotFoundException **isA** IOException and IOException **isA** Exception
- If you catch/except the **parent** first, the code will never get a chance to give a specific response to any **child!**

Exceptions in Real Life

- Exceptions aren't just merely 'error checking'
  - They are also incredibly useful as assertion checks, ensuring that the program is always in a valid state
  - Why bother? Because if you mess up somewhere in the code your mistake will quickly (or not so quickly) result in an exception
    - … alerting you to the existence of your mistake
  - So PUT EXCEPTIONS EVERYWHERE
    - The closer the exception is to the bug, the easier it is to find
    - EXCEPTIONS WILL SAVE YOU OVER & OVER

Recursion

https://giphy.com/gifs/queen-bohemian-rhapsody-4ZcVk52nDpaliEVhqN

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
    - The simpler version of the problem reduces its problem and calls the method again
      - The simpler version of the problem reduces its problem and calls the method again
        - The simpler version of the problem reduces its problem and calls the method again
          - The simpler version of the problem reduces its problem and calls the method again
            - The simpler version of the problem reduces its problem and calls the method again
              - The simpler version of the problem reduces its problem and calls the method again
                - 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 (Java)

- Formal Mathematical definition:
  - N! : 0! = 1 otherwise N * (N-1)!
- Iterative solution using for loop from OOPD

```
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 )                                  ← Simplest case (the 'base case')
      return 1;                                   ← oops – multiple returns!
   else
      return n * calcNFactorialRecursive( n-1 );  ← Recursive call multiplied by n,
}                                                    n changed to go towards base case
```

  - No error checking in either approach!

Example 1: Factorial (Python)

- Formal Mathematical definition:
  - N! : 0! = 1 otherwise N * (N-1)!
- Iterative solution using for loop

```
def calcNFactorial( n ):
      nFactorial = 1
      for ii in range( n, 1, -1 ):
        nFactorial = nFactorial * ii
      return nFactorial
```

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

```
def calcNFactorialRecursive( n ):
   if ( n == 0 ):                          ← Simplest case (the 'base case')
      return 1                             ← oops – multiple returns!
   else:
      return n * calcNFactorialRecursive( n-1 )   ← Recursive call multiplied by n,
                                                    n changed to go towards base case
```

  - *Note: No error checking in either approach!*

Removing multiple return statements (Python)

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

```
def calcNFactorialRecursive( n ):
   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
```

Removing multiple return statements (Java)

- 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 (Java)

- 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
 }
```

Error checking (Python)

- Throw an exception if the import is invalid

```
def calcNFactorialRecursive( n ):
    factorial = 1                          ← declare and initialise local variable
    if ( n < 0 ):                          ← error condition
        raise ValueError("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) One or more base case(s) 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 (e.g., 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 not always
   ELSEIF terminating_condition_2 THEN
     result ← base_case_2
   ELSEIF                          ← There can be any number of base cases
      ...
   ELSE
     reduce_prob_using_recursion   ← This could be one line or a whole sub-system
     result ← results_of_reduction
   ENDIF
```

Example 2: Fibonacci Numbers (Java)

- 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 Numbers (Python)

- 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:

```
def fibIterative( n ):
    fibVal = 0                  ← value n
    currVal = 1                 ← value n-1
    lastVal = 0                 ← value n-2

    if (n == 0):
        fibVal = 0
    else if (n == 1):
        fibVal = 1
    else:
        for ii in range(2, n):
            fibVal = currVal + lastVal
            lastVal = currVal           ← set up lastVal and currVal ready for next iteration
            currVal = fibVal
    return fibVal
```

Example 2: Fibonacci Sequence (Java)

- 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)

Example 2: Fibonacci Sequence (Python)

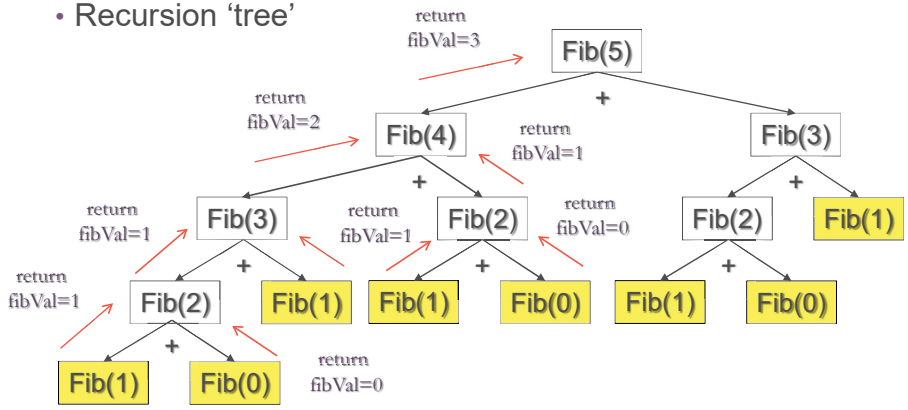
- Recursive solution:

```
def fibRecursive( n ):
    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'

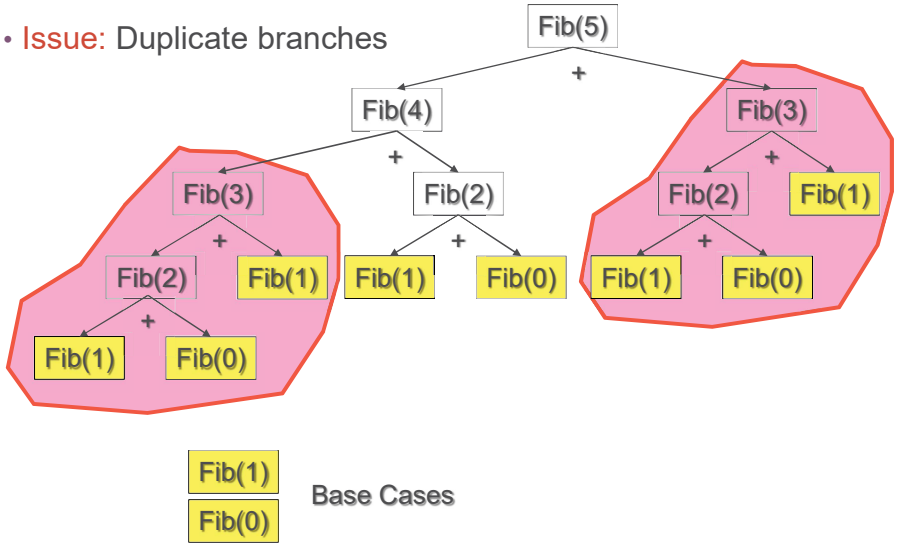


Fib(1)
Fib(0)

Base Cases

A Closer Look at Recursive Fibonacci

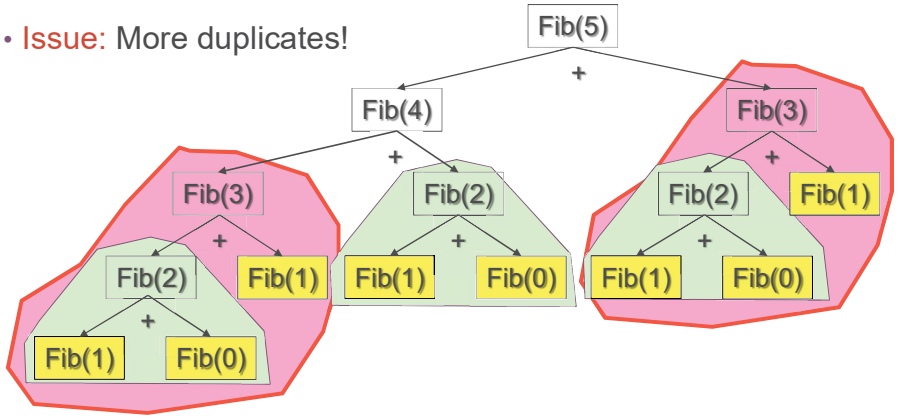
- Issue: Duplicate branches

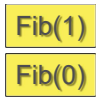


Base Cases

A Closer Look at Recursive Fibonacci

- Issue: More duplicates!





Base Cases

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: Binary Search

- Discussed last lecture… iterative solution:

```
METHOD BinarySearchIter IMPORT sortedArr, searchTgt
                        EXPORT matchIdx, which will be -1 if not found
matchIdx ← -1                                          ← Assume failure to find the target
lowerBd ← 0                                               could throw exception
upperBd ← sortedArr.length

WHILE (NOT found) AND (lowerBd <= upperBd) DO
   chkIdx ← (lowerBd + upperBd) / 2
   IF (sortedArr[chkIdx] < searchTgt) THEN
      lowerBd ← chkIdx+1                               ← target must be in the upper half
   ELSEIF (sortedArr[chkIdx] > searchTgt) THEN
      upperBd ← chkIdx-1                               ← target must be in the lower half
   ELSE
      matchIdx ← chkIdx                                ← found our target
      found ← TRUE
   ENDIF
ENDWHILE
```

Example 3: Binary Search

- And now a recursive solution:

```
METHOD BinarySearchRec IMPORT sortedArr, searchTgt, lowerBd, upperBd
                       EXPORT matchIdx, which will be -1 if not found
matchIdx ← -1                                          ← Assume failure to find the target
                                                         or could throw exception

chkIdx ← (lowerBd + upperBd) / 2
IF (sortedArr[chkIdx] < searchTgt) THEN
      matchIdx = BinarySearch ← (chkIdx+1, upperBd)    ← target must be in the upper half
ELSEIF (sortedArr[chkIdx] > searchTgt) THEN
      matchIdx = BinarySearch ← (lowerBd, chkIdx-1)    ← target must be in the lower half
ELSE
      matchIdx ← chkIdx                                ← found our target
      found ← TRUE
ENDIF
```

Wrapper methods

- The BinarySearchRec method assumes valid lower/upper bounds.
- To protect it we mark `binarySearchRec()` as private.

```
private int binarySearchRec(int[] sortedArr…) // Java…

# Python – underscore is convention for "private"
def _binarySearchRec(sortedArr…):
```

- Then add a public wrapper method to validate/set up arguments

```
public int binarySearch(int[] sortArr, int target) //Java
    {
    return binarySearchRec(sortArr, target, 0, sortArr.length-1);
    }
def binarySearch(sortArr, target):                    # Python
    return binarySearchRec(sortArr, target, 0, len(sortArr)-1)
```

The Importance of Terminating

- Consider the following recursive method:

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

```
def endless(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

- Each successive call puts another stack frame on the stack
- This happens with all function/method calls, not just recursive calls

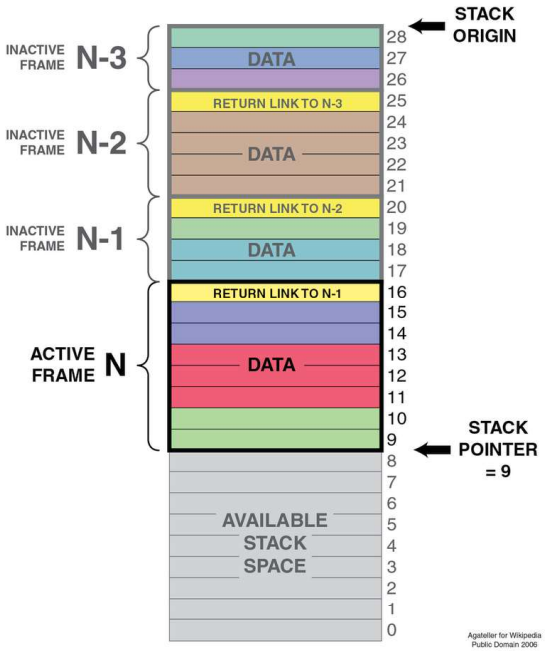


Image from : https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

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 error
    - Python failes with a RecursionError
    - 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/Python, 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!~~
    - *UPDATE – C++ now puts references on the stack. This text left in to flag the "answer" to an old exam question. It is now out of date.*

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

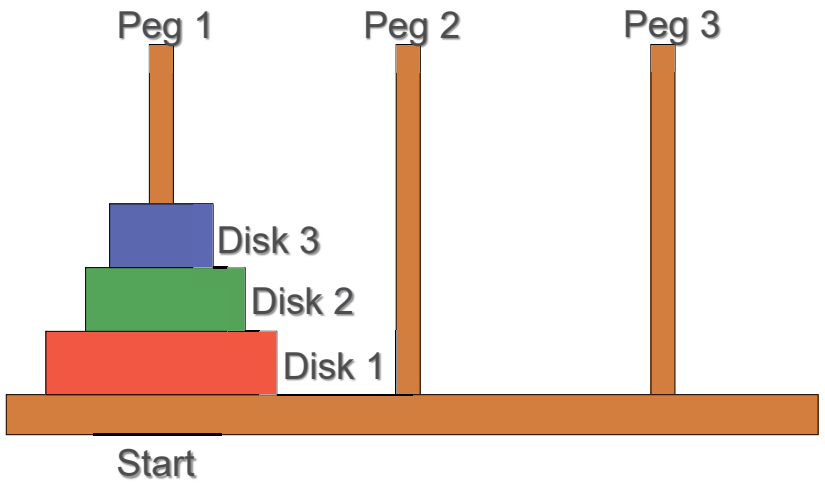
```
METHOD towers IMPORT n, src, dest   ← Move n disks from peg src to peg dest

   if (n == 1)
      moveDisk ← (src, dest)          ← Base case: move one disk from peg src to peg dest
   else
      tmp = 6 - src - dest            ← tmp is the 'other' (non-target) peg,
                                            since src+dest+tmp = 6
      towers ← (n-1, src, tmp)        ← Move all but bottom disk to temp peg tmp
                                        This is a smaller (n-1) version of the current problem
      moveDisk ← (src, dest)          ← Move bottom disk to target peg dest
      towers  ← (n-1, tmp, 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

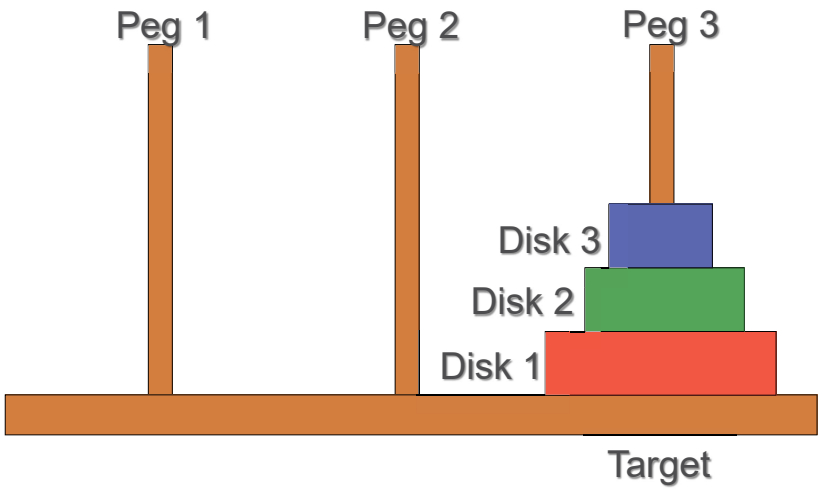
Towers of Hanoi – Step-by-Step

- 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

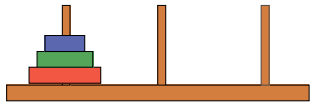


Target State



Stepping Through the Algorithm

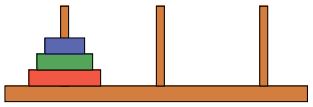
towers(3, 1, 3)
- tmp = 6-1-3 = 2
- towers(2, 1, 2)
  - tmp = 6-1-2 = 3
  - towers(1, 1, 3)
  - moveDisk(1,2)
  - towers(1,3,2)
- moveDisk(1, 3)
- towers(2, 2, 3)
  - tmp = 6-2-3 = 1
  - towers(1, 2, 1)
  - moveDisk(2, 3)
  - towers(1, 1, 3)



```
Recursion Code
if (n ==1)
    moveDisk(src, dest)
else
    tmp = 6 - src – dest
    towers(n - 1, src, tmp)
    moveDisk(src, dest)
    towers(n-1, tmp, dest)
```

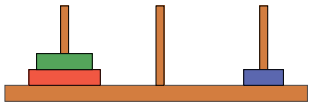
towers(3, 1, 3)
- tmp = 6-1-3 = 2
- **towers(2, 1, 2)**
  - tmp = 6-1-2 = 3
  - towers(1, 1, 3)
  - moveDisk(1,2)
  - towers(1,3,2)
- moveDisk(1, 3)
- towers(2, 2, 3)
  - tmp = 6-2-3 = 1
  - towers(1, 2, 1)
  - moveDisk(2, 3)
  - towers(1, 1, 3)

No change



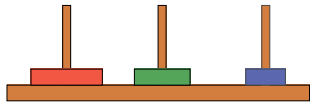
```
Recursion Code
if (n ==1)
    moveDisk(src, dest)
else
    tmp = 6 - src – dest
    towers(n - 1, src, tmp)
    moveDisk(src, dest)
    towers(n-1, tmp, dest)
```

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



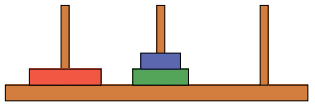
```
Recursion Code
if (n ==1)
    moveDisk(src, dest)
else
    tmp = 6 - src – dest
    towers(n - 1, src, tmp)
    moveDisk(src, dest)
    towers(n-1, tmp, dest)
```

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



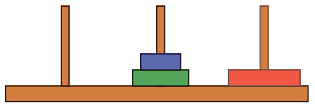
```
Recursion Code
if (n ==1)
    moveDisk(src, dest)
else
    tmp = 6 - src – dest
    towers(n - 1, src, tmp)
    moveDisk(src, dest)
    towers(n-1, tmp, dest)
```

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



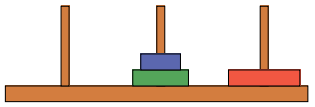
```
Recursion Code
if (n ==1)
    moveDisk(src, dest)
else
    tmp = 6 - src – dest
    towers(n - 1, src, tmp)
    moveDisk(src, dest)
    towers(n-1, tmp, dest)
```

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



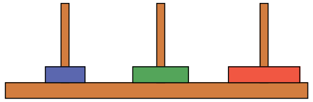
```
Recursion Code
if (n ==1)
    moveDisk(src, dest)
else
    tmp = 6 - src – dest
    towers(n - 1, src, tmp)
    moveDisk(src, dest)
    towers(n-1, tmp, dest)
```

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



```
Recursion Code
if (n ==1)
    moveDisk(src, dest)
else
    tmp = 6 - src – dest
    towers(n - 1, src, tmp)
    moveDisk(src, dest)
    towers(n-1, tmp, dest)
```

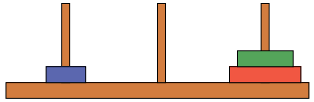
towers(3, 1, 3)
tmp = 6-1-3 = 2
towers(2, 1, 2)
tmp = 6-1-2 = 3
towers(1, 1, 3)
moveDisk(1,2)
towers(1,3,2)
moveDisk(1, 3)
towers(2, 2, 3)
tmp = 6-2-3 = 1
★ **towers(1, 2, 1)**
moveDisk(2, 3)
towers(1, 1, 3)



Recursion Code

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

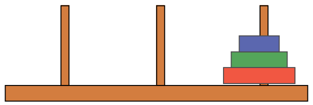
towers(3, 1, 3)
tmp = 6-1-3 = 2
towers(2, 1, 2)
tmp = 6-1-2 = 3
towers(1, 1, 3)
moveDisk(1,2)
towers(1,3,2)
moveDisk(1, 3)
towers(2, 2, 3)
tmp = 6-2-3 = 1
towers(1, 2, 1)
★ **moveDisk(2, 3)**
towers(1, 1, 3)



Recursion Code

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

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



Recursion Code

```
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
    - *e.g.,* 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

- **Towers of Hanoi** – a few lines of code recursively.
  6 pages of code iteratively!

Recursion vs Iteration

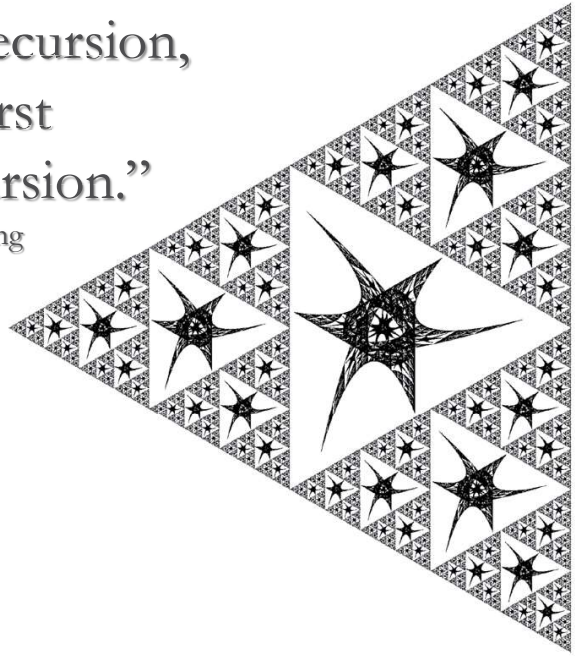
- Advantages of a recursive solution:
  - ☑ Some algorithms are *much* simpler using recursion
- Disadvantages of recursion vs iteration
  - ☒ The call stack limits the number of recursive 'iterations' that can be performed
    - No more than a few thousand iterations before stack overflow
  - ☒ Usually slower due to method call overhead
    - Every time a method is called, a few instructions are needed to set up the method call (e.g., 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

"To understand recursion, one must first understand recursion."

— Stephen Hawking



Next Week

- Objects (Revision)
- Stacks
- Queues