

Recursion

- A recursive function, as you saw in CS100, is one that calls itself.
- Classic textbook example: factorial

Mathematical definition:

$$0! = 1$$

$$N! = N \times (N - 1)! \text{ if } N > 0$$

How would you implement this?

Recursion vs. Iteration

- Roughly speaking, recursion and iteration perform the same kinds of tasks:
 - Solve a complicated task one piece at a time, and combine the results.
- Emphasis of iteration:
 - keep repeating until a task is “done”
 - e.g.*, loop counter reaches limit,
linked list reaches null pointer,
`instream.eof()` becomes true
- Emphasis of recursion:
 - Solve a large problem by breaking it up into smaller and smaller pieces until you can solve it; combine the results.
 - e.g.*, recursive factorial function

Which is Better?^a

- No clear answer, but there are known trade-offs.
- “Mathematicians” often prefer recursive approach.
 - Solutions often shorter, closer in spirit to abstract mathematical entity.
 - Good recursive solutions *may* be more difficult to design and test.
- “Programmers”, esp. w/o college CS training, often prefer iterative solutions.
 - Somehow, it seems more appealing to many.
 - Control stays local to loop, less “magical”.

Compare iterative to recursive versions of factorial!

^aSome of these statements are personal opinion.

Which Approach Should You Choose?

- Depends on the problem.
 - The factorial example is pretty artificial; it’s so simple that it really doesn’t matter which version you choose.
- Many ADTs (*e.g.*, trees) are simpler & more natural if methods are implemented recursively.
- Recursive isn’t always better, ’tho:

```
// Recursively compute nth Fibonacci number.  
// assumes n>=0  
public static int fib (int n) {  
  
  
  
  
  
  
  
  
  
}
```

→ This takes $O(2^n)$ steps! Unusable for large n .

```
// Iteratively compute nth Fibonacci number.
// assumes n>=0
public static int ifib (int n) {

}
}
```

→ This iterative approach is “linear”; it takes $O(n)$ steps.

Moral: No substitute for careful thought.

Moral: “Obvious” and “natural” solutions aren’t always practical.

Basic Idea of Recursion

1. Know how to solve a problem immediately for “small” of trivial cases.

→ These are called the *basis* cases.

→ Often there are only one or two of these.

2. If input is non-trivial, can break it up smaller and smaller until chunks until you reach something you know how to deal with.

Eventually, you make enough recursive calls that the input reaches a “basis case”.

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

Problem: Infinite Recursive Calls

- If you are not careful with the program logic, you may miss a basis case and go off into an infinite recursion.
- This is similar to an infinite loop!
- Example: call to factorial with $N < 0$
 - Either you must ensure that factorial is never, ever called with a negative N , or you must build in a check somehow.
- Moral: When you are designing your recursive calls, make sure that at least one of the basis cases MUST be reached eventually.
 - This is often pretty hard!

More Recursive Examples

```
class RecursionTest {
    // Find max value in an unsorted array of ints.
    public static int findMax (int [] A, int startIndex) {
        if (startIndex == A.length - 1) {
            return A[startIndex];
        } else {
            return Math.max (A[startIndex],
                findMax(A, startIndex+1)) ;
        }
    }

    public static void main (String[] args) {
        int [] A = new int[5];
        A[0] = 3; A[1] = 37; A[2] = -5;
        A[3] = 12; A[4] = 7;
        System.out.println ("Max = " + findMax (A, 0));
    }
}
```

Recursive Implementations of BST Routines

- Can implement many BST routines recursively.
- Recursive method implementations are more elegant than iterative, but no more or less efficient:
 - Recursion is a big win for printing full BSTs.
 - Search is a little nicer.
 - Insert would be nicer recursively ... if only Java allowed changes to parameters to percolate back to the caller.
 - Delete still complicated; left to student.
- Assume BSTNode class is the same as before. We will implement a new class called `recBST` for recursive BSTs.
- Again, will assume only integers values are stored. Can make BSTs of more interesting entities if they support ideas of `lessThan/greaterThan`.
 - BST routines will have to be altered slightly, but the basic ideas are the same.

```
public class recBST {
    BSTNode root;

    public recBST () {
        root = null;
    }

    // Public sees this function, which starts
    // a recursive search at the root of the tree.
    public boolean search (int value) {
        return rSearch (value, root);
    }

    private boolean rSearch (int value, BSTNode curNode) {
        if (curNode == null) {
            return false;
        } else if (curNode.value == value) {
            return true;
        } else if (value < curNode.value) {
            return rSearch (value, curNode.left);
        } else {
            return rSearch (value, curNode.right);
        }
    }

    ...
}
```

An **Incorrect** Approach to insert

```
public void insert (int value) {
    rInsert (value, root);
}

private void rInsert (int value, BSTNode node) {
    if (node == null) {
        node = new BSTNode (value);
    } else if (value < node.value) {
        rInsert (value, node.left);
    } else if (value > node.value) {
        rInsert (value, node.right);
    }
}
```

- This approach will work in some programming languages ... but not Java.
-
-

Correct Recursive BST insert

```
public void insert (int value) {
    root = rInsert (value, root);
}

private BSTNode rInsert (int value, BSTNode node) {
    if (node == null) {
        node = new BSTNode (value);
    } else if (value < node.value) {
        node.left = rInsert (value, node.left);
    } else if (value > node.value) {
        node.right = rInsert (value, node.right);
    }
    return node;
}
```

- Solution is to send back a reference to the new node as the value of the function!
- Somewhat unintuitive.

Printing a BST

- Want to print all values stored in a tree in increasing order.
- Recall BST property:
→
- Thinking recursively ... what should the print routine look like?
- Again, if node has interesting value, may want to invoke `node.toString` or another customized routine instead of `System.out.println(node.value)`

```
public void print () {
    printSubtree(root);
}

private void printSubtree (BSTNode node) {

}
```

Example: Searching Through a List

- Want to search through an array for a sought element.
- Again, we assume integers for simplicity of example. In “real world”, we would be searching through an array of generic `Sortables`.
i.e., use `equals` instead of `==`,
use `lessThan` instead of `<`
- I hope you have already seen iterative versions of linear and binary search in CS100!

Linear Search

Simplest approach: linear search

- Start at beginning, keep going until you find the element.
- Works with sorted and unsorted lists
[If list is sorted, can exit once elements become larger than sought value.]

```
// Returns an index in array of soughtVal, if it occurs.
// Returns -1 is soughtVal is not present in array.
// Note: Array A need not be sorted.
public static int iterLinearSearch (int [] A,
    int soughtVal) {
    for (int i=0; i<A.length; i++) {
        if (A[i] == soughtVal) {
            return i;
        }
    }
    return -1;
}
```

Recursive Linear Search

Binary Search

Idea:

- Have a window or range of values we are currently considering.
[Initially, the window is the whole array.]
- Look at midpoint in range and compare to soughtValue.
 - If $A[\text{mid}] == \text{soughtValue}$, we're found it.
 - If $A[\text{mid}] < \text{soughtValue}$, discard first half.
 - If $A[\text{mid}] > \text{soughtValue}$, discard second half.
- Keep halving the list until either you find it or your sublist has no elements.
- Computational complexity of binary search is number of times you can halve the list.

Iterative Binary Search

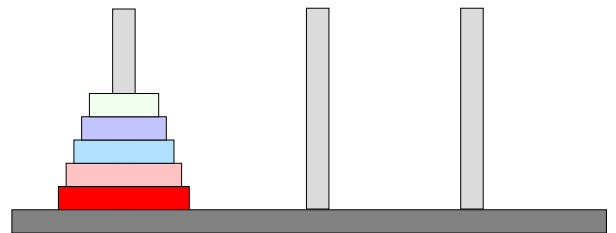
```
// Returns an index in array of soughtVal, if it occurs.
// Returns -1 if soughtVal is not present in array.
// Note: Array A MUST be sorted.
public static int iterBinarySearch (int [] A,
    int soughtVal) {
    int lo=0, hi=A.length-1;

    // Exit loop when lo>hi. This will happen just
    // after the sublist has been reduced to one
    // element (lo==hi) and then we reset lo or hi
    // because we didn't find soughtVal there.
    while (lo <= hi) {
        // note integer division.
        final int mid = (lo + hi)/2;
        if (A[mid] == soughtVal) {
            return mid;
        } else if (A[mid] < soughtVal) {
            // discard first half
            lo = mid +1;
        } else {
            // discard second half
            hi = mid -1;
        }
    }
    return -1;
}
```

Recursive Binary Search

```
// Returns an index in array of soughtVal, if it occurs.
// Returns -1 if soughtVal is not present in array.
// Note: array A must be sorted.
// Also note extra parameters lo and hi. Clients will
// make initial call with lo==0 and hi==A.length-1
public static int recBinarySearch (int [] A,
    int soughtVal, int lo, int hi) {
}
}
```

Example: Towers of Hanoi



Classic ancient problem:

- N rings in increasing size.
- 3 poles.
- Rings start stacked on pole 1. Goal is to move rings so that they are stacked on pole 3 ... BUT
 - Can only move one ring at a time.
 - Can't put larger ring on top of smaller.
- Iterative solution is "powerful ugly"; recursive solution is "elegant".

Towers of Hanoi Solution

```
public class hanoi {
    public static void move (int N, int src, int dest) {
        if (N>0) {
            // neat trick to get index of other pole.
            final int temp = 6 - src - dest;
            move (N-1, src, temp);
            System.out.println ("Move ring from pole "
                + src + " to pole " + dest);
            move (N-1, temp, dest);
        }
    }

    public static void main (String[] args) {
        // Move two rings from pole 1 to pole 3.
        System.out.println ("\nSoln for two rings:\n");
        move (2, 1, 3);
        // Move three rings from pole 1 to pole 3.
        System.out.println ("\nSoln for three rings:\n");
        move (3, 1, 3);
    }
}
```

How Recursion is Implemented by the Compiler

- Run-time *stack* is used to keep track of pending function calls (parameters and local variables).
- Storage for *objects* comes from another part of memory called the *heap*.
 - However, params and local vars that refer to these objects are stored within the run-time stack somewhere.
- static variables are stored somewhere else.
- The set of params and local vars for a function call is stored in an *activation record* (AR).

- If one function calls another, a new AR is created and pushed onto the run-time stack. AR has storage for params and local vars PLUS remembers where to return to when done.
- When a function call finishes, the AR is popped off the stack and (eventually) destroyed. Return to appropriate spot and return the value of the function (if not void).

Tracing Through Hanoi