Recursion

A recursive method is a method that calls itself

Recursion can be used to solve problems that can be reduced to simpler instances of itself

A recursive method has two kinds of cases

  One or more stopping or base cases that solve the problem without any recursive calls
  One of more cases that include a recursive call (involving a simpler problem)

If a method continually calls itself without returning, eventually a “StackOverflowError” will be thrown
When to use recursion

A recursive solution is natural and easy to understand
A recursive solution doesn’t result in excessive duplicate computation
The equivalent iterative solution is too complex
Examples of Recursion

  Towers of Hanoi
  Factorial
  Domino Tiling and other geometrical puzzles
  Fibonacci numbers
  Flood filling a region
  Exploring a maze
  Operating on trees and graphs
Towers of Hanoi

Problem:
There are three pegs and one of them has \( n \) disks on it, each disk larger than the one above it. The problem is to move all of the disks to another peg by moving one disk at a time without ever placing a larger disk on top of a smaller disk.

Key observation:
In order to move a disk it is necessary for all smaller disks to be on a single, different peg.
Disks larger than the one we are trying to move can be anywhere.
In the picture, to move disk #7 to the destination peg, all of the smaller disks have to be on the remaining peg.
Towers of Hanoi Algorithm:

Base case:
Moving zero disks can be done by doing nothing.

Recursive case:
To move n pegs to a target peg with n > 0, first move the n - 1 smaller pegs to the other peg, move the n disk to the target peg, then move the n - 1 smaller pegs to the target peg.

Notice that we could have made the base case be moving the smallest disk. This works but the algorithm is slightly more complicated:

Base case:
Moving the smallest disk can be done by simply moving it to the destination peg.

Recursive case:
To move n pegs to a target peg with n > 1, first move the n - 1 smaller pegs to the other peg, move the n disk to the target peg, then move the n - 1 smaller pegs to the target peg.

General computer science hint: algorithms can be simplified and made more general by carefully considering the zeroth or empty cases.
Towers of Hanoi Solution

public class Hanoi {

    public static void main( String args[] ) {
        if ( args.length != 1 ) {
            System.err.println( "Usage: java Hanoi n" );
        } else {
            int n = Integer.parseInt( args[0] );

            moveDisks( n, "A", "C", "B" );
        }
    }

    public static void moveDisks( int n,
                                   String from,
                                   String to,
                                   String other ) {

        if ( n == 0 ) {
            return;
        } else {
            moveDisks( n - 1, from, other, to );
            System.out.println( "Move disk " + n
                                 + " from " + from
                                 + " to " + to );
            moveDisks( n - 1, other, to, from );
            return;
        }
    }
}

Towers of Hanoi Solution with count of moves:

```java
public class Hanoi {

    public static void main( String args[] ) {
        if ( args.length != 1 ) {
            System.err.println( "Usage: java Hanoi n" );
        } else {
            int n = Integer.parseInt( args[0] );
            System.out.println( "Total moves " + moveDisks( n, "A", "C", "B" ) );
        }
    }

    public static int moveDisks( int n, String from, String to, String other ) {
        if ( n == 0 ) {
            return 0;
        } else {
            int m1, m2;
            m1 = moveDisks( n - 1, from, other, to );
            System.out.println( "Move disk " + n + " from " + from + " to " + to );
            m2 = moveDisks( n - 1, other, to, from );
            return m1 + 1 + m2;
        }
    }
}
```
Output for Towers of Hanoi:

swm[86] % java Hanoi 0
Total moves 0

swm[87] % java Hanoi 1
Move disk 1 from A to C
Total moves 1

swm[88] % java Hanoi 2
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Total moves 3

swm[89] % java Hanoi 3
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Total moves 7

swm[90] % java Hanoi 4
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
Move disk 4 from A to C
Move disk 1 from B to C
Move disk 2 from B to A
Move disk 1 from C to A
Move disk 3 from B to C
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Total moves 15
Factorial

Definition

\[
0! = 0 \\
n! = n \times (n - 1)!
\]

Observation: This definition works for all non-negative integers. (Factorial is not defined for negative integers.)

Both kinds of recursive cases

The first case is the stopping or base case.

The second case reduces the problem to a simpler case.

Direct implementation:

```java
public class Factorial {
    public static void main( String args[] ) {
        System.out.println( "Factorial(5) = " "+ factorial(5) );
    }

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

How many ways can dominos be placed in a 2 x n rectangle?

What is the general rule?

Width 1 can be filled in 1 way

Width 2 can be filled in 2 ways

Width 3 can be filled in 3 ways
Width 4 can be filled in 5 ways
Domino Tiling - continued

Problem: How many ways can a 2 x n rectangle be tiled with dominos?

How can we break this problem into simpler ones?

The domino in the upper left corner must be horizontal or vertical

- If it is vertical then there remains a 2 x (n-1) rectangle to tile
- If it is horizontal then there must be another horizontal domino below it and this leaves a 2 x (n-2) rectangle left to tile

Sketch of solution

- If (amount already tiled) = n then we have no more to tile and we are done - print solution and return 1 (the solution we just printed)

Otherwise the number of ways to tile is the sum of the number of ways by starting with a vertical domino plus the the number of ways by starting with a horizontal domino

To tile starting with a vertical domino we check if the length is at least 1 and then put a vertical domino in the puzzle and then tile the resulting 2 x (n-1) puzzle.

To tile starting with a horizontal domino we check if the length is at least 2 and then put two horizontal dominos in the puzzle and then tile the resulting 2 x (n-2) puzzle.
public class Dominos {
    static char [] dominos;
    static void main( String args[] ) {
        if ( args.length != 1 ) {
            System.err.println( "Usage: java Dominos n" );
        } else {
            dominos = new char[Integer.parseInt( args[0] )];
            System.out.println("Total ways is " + fill(0));
        }
    }
    static int fill(int n) {
        if ( n == dominos.length ) {
            System.out.println( dominos );
            System.out.println( dominos );
            System.out.println();
            return 1;
        } else {
            return fillv( n ) + fillh( n );
        }
    }
    static int fillv( int n ) {
        if ( n < dominos.length ) {
            dominos[n] = '|';
            return fill( n + 1 );
        } else { return 0; }
    }
    static int fillh( int n ) {
        if ( n < dominos.length - 1 ) {
            dominos[n] = '-';
            dominos[n+1] = '~';
            return fill( n + 2 );
        } else { return 0; }
    }
}
Output from Domino program:

```
swm[71] % java Dominoes 5

Total ways is 8
```
Fibonacci numbers

Definition

\[ \text{fib}(0) = 1 \]
\[ \text{fib}(1) = 1 \]
\[ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2) \]

Observation: This definition works for all non-negative integers

Both kinds of recursive cases

The first two cases are the stopping or base case.

The third case reduces the problem to simpler cases.

Note: there exists a closed-form formula for Fibonacci numbers:

\[ \text{fib}(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{2\sqrt{5}} \]

(The number of ways to tile a 2 x n rectangle with Dominos is \( \text{fib}(n) \) - can you see why?)
Fibonacci numbers solution:

```java
public class Fibonacci {

    static void main( String args[] ) {
        if ( args.length != 1 ) {
            System.err.println( "Usage: java Fibonacci n" );
        } else {
            int n = Integer.parseInt( args[0] );
            System.out.println( "Fibonacci(" + n + ") = " + fib( n ) );
        }
    }

    public static int fib( int n ) {
        if ( n == 0 || n == 1 ) {
            return 1;
        } else {
            return fib( n - 1 ) + fib( n - 2 );
        }
    }
}
```
Call chart for Fibonacci numbers

Fibonacci(6) = 13
fib(6) = 13
  fib(5) = 8
    fib(4) = 5
      fib(3) = 3
        fib(2) = 2
          fib(1) = 1
          fib(0) = 1
        fib(1) = 1
    fib(2) = 2
      fib(1) = 1
      fib(0) = 1
  fib(3) = 3
    fib(2) = 2
      fib(1) = 1
      fib(0) = 1
    fib(1) = 1
fib(4) = 5
  fib(3) = 3
    fib(2) = 2
      fib(1) = 1
      fib(0) = 1
    fib(1) = 1
fib(2) = 2
  fib(1) = 1
  fib(0) = 1
Flood filling a region

Given a rectangular array of pixels with some pixels designated as "boundary" pixels and a pixel to start filling, the problem is to fill all pixels connected to the designated pixel by a path that doesn’t go through a boundary pixel with a designated color.

```java
public class FloodFill {

    int m, n, a[][];

    public FloodFill( int m, int n ) {
        this.m = m;
        this.n = n;
        a = new int[m][n];
    }

    public static void main(String args[]) {
        FloodFill f = new FloodFill( 10, 10 );
        System.out.println( f.floodFill( 2, 3, 7 ) );
    }

    public int floodFill( int x, int y, int newColor ) {
        return flood( x, y, a[x][y], newColor );
    }

    private int flood( int x, int y,
            int oldColor, int newColor ) {
        if ( x < 0 || y < 0 || x >= m || y >= n) {
            return 0;
        }
        if ( a[x][y] != oldColor ) return 0;
        a[x][y] = newColor;
        return 1 + flood( x, y+1, oldColor, newColor )
            + flood( x, y-1, oldColor, newColor )
            + flood( x+1, y, oldColor, newColor )
            + flood( x-1, y, oldColor, newColor );
    }
}
```
Exploring a maze (general idea)

General way to explore:

Have you been to where you are now?
   If so then return - you have either explored this place or are exploring it now

Mark this place as explored or being explored

Enumerate all the possible ways to go from where you are now

For each way
   Explore it and return to this place

You are done exploring this place - return
Exploring a maze (Main pp 391-397)

Explore ahead of you:

If there is a wall in front of you then you are done - return
If your name is written on the ground in the square in front of you then you are done - return
Write your name in the square in front of you.
Step forward one square (you are now standing on the name you just wrote)
Turn left 90°
“Explore ahead of you”
Turn right 90°
“Explore ahead of you”
Turn right 90°
“Explore ahead of you”
Turn right 90°
Step forward one square and turn 180°
You are done - return
Operating on trees and graphs

Each node of a tree or graph is similar so there are several recursive algorithms that are called with a node of a tree or graph as an argument and do a recursive call on some of the neighboring nodes. These algorithms are very similar to exploring a maze.
How deeply can we recurse?

Different programming environments enable different amounts of recursion.

My lisp environment was happy to recursively call a function a million times.

We can write a Java program to find out how deeply we can recurse in Java:

```java
public class Recursion {

    static int n = 0;

    public static void main(String args[]) {
        try {
            f();
        } catch (StackOverflowError e) {
            System.out.println("Depth = " + n);
        }
    }

    public static void f() {
        n++;
        f();
    }

    Depth = 4332 at RIT
    Depth = 7641 at home
```
How deep with lots of local variables:

```java
public class Recursion {
    static int n = 0;
    public static void main(String args[]) {
        try {
            f();
        } catch (StackOverflowError e) {
            System.out.println("Depth = " + n);
        }
    }
    public static void f() {
        int a = 234;
        int b = 234;
        int c = 234;
        int d = 234;
        int e = 234;
        int f = 234;
        int g = 234;
        int h = 234;
        int i = 234;
        int j = 234;
        int k = 234;
        int l = 234;
        int m = 234;
        int q = 234;
        int o = 234;
        int p = 234;
        n++;
        f();
    }
}
```

Depth = 2682 at RIT
Depth = 2953 at home