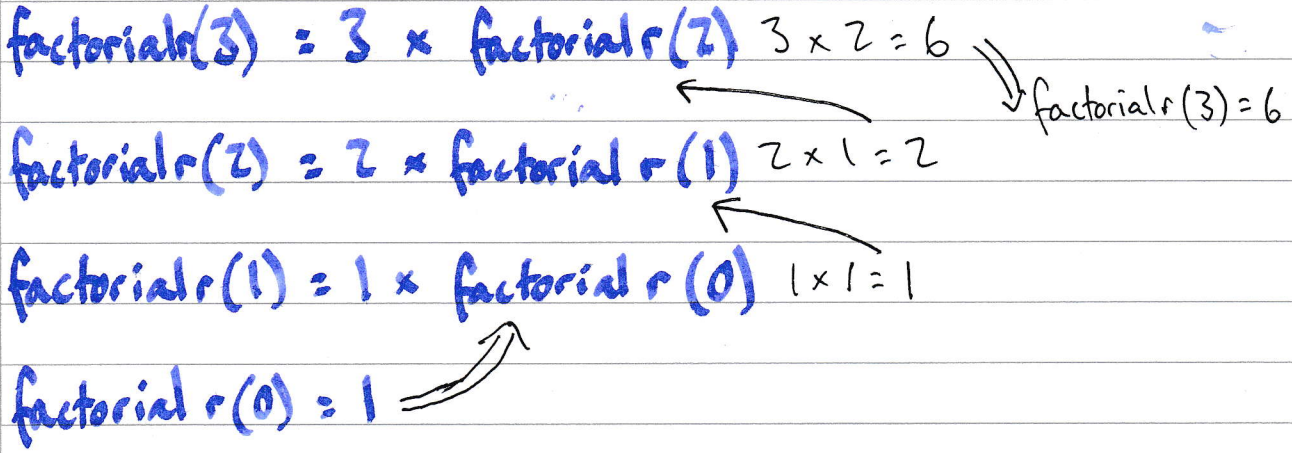
Recursion

- Method that calls itself with a smaller version. It cannot be infinite. Must reach BASE CASE
  - stops recursion

Tree Diagram



Tracing through with recursive tree diagram

decToBinary(14) = print 0 (4)

decToBinary(7) = print 1 (3)

decToBinary(3) = print 1 (2)

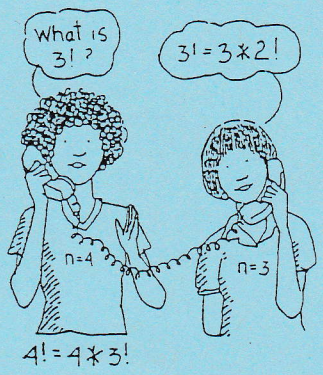
decToBinary(1) = print 1 (1)

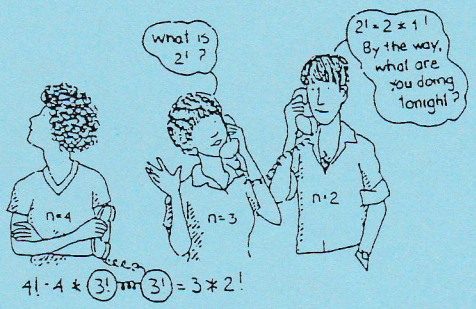
} 1110

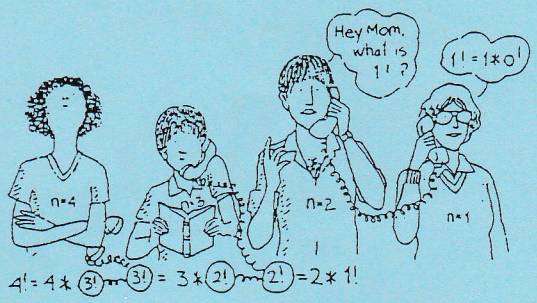
decToBinary(0) = Stop. (rewind)
Base case reached.

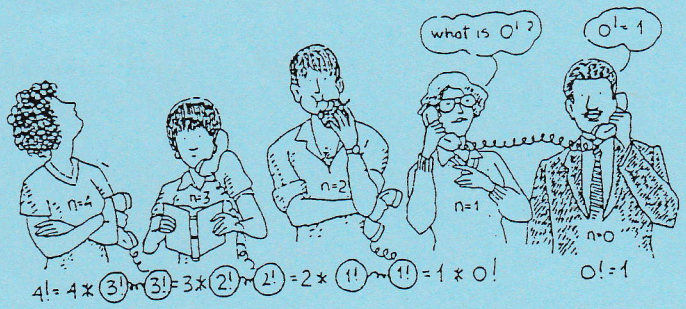
Tail recursion - a series of recursive calls that upon rewind, has nothing left to do

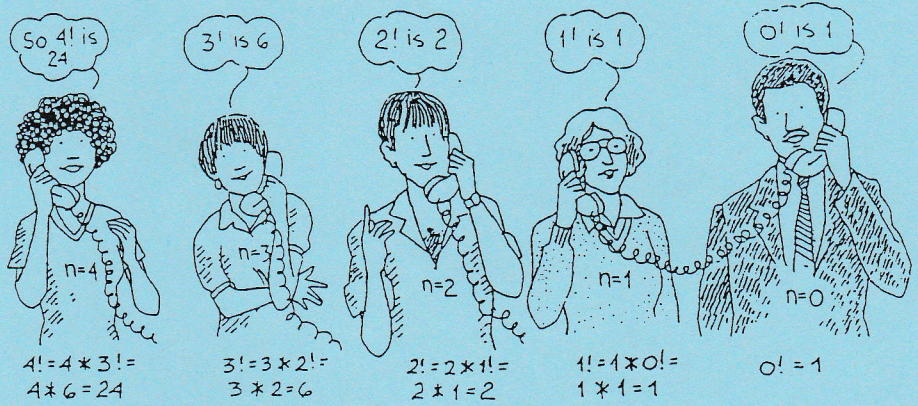
Recursion In Action... Solving n! where n=4



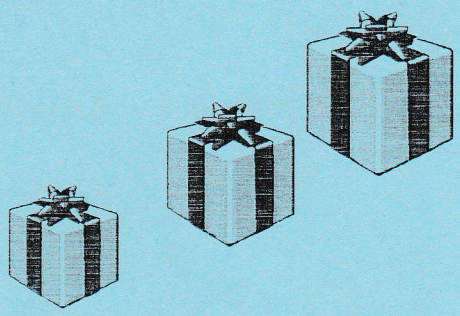








Our First Recursion Program



```
public class Factorial
{
        public static void main(String[] args)
        {
                System.out.println(factorialr(3));
                System.out.println(factoriali(3));
        }

        private static int factorialr (int n)
        {                                                    recursive
                if (n == 0)
                        return 1;
                else
                        return n * factorialr(n-1);
        }

        private static int factoriali (int n)//iterative solution
        {

                int result = 1;

                for (int i = n;  i >= 2; i--)
                        result = result * i;

                return result;
        }
}
```

Writing Recursion

1) You must figure out how to ~~base case~~ write a base case.
Most trivial case, the one you know the answer to.
The base case will stop recursion.

2) Use actual examples, redefine your problem using a smaller version of itself.

$$\sum_{x=1}^{4} = (2^1+1) + (2^2+1) + (2^3+1) + (2^4+1)$$
$$= 34$$

3) Combine results and try to use
```
if(base case)
else
//recursion
```

Lab 9

```
public static double recurSum(int n)
  if(n < 1)
  return 0;
  else
  return Math.pow(2, n) + 1 + recurSum(n-1)
```

palindrome("RADAR") = 'R' == 'R' && palindrome("ADA") → true

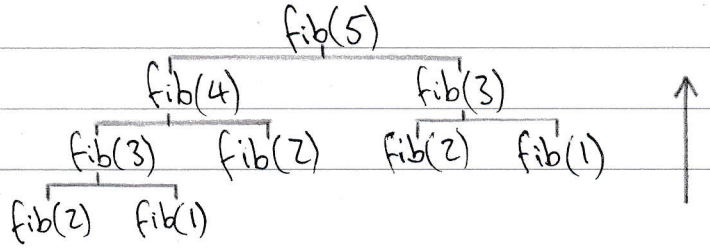
palindrome("ADA") = 'A' == 'A' && palindrome("D") → true : true

palindrome("D") : true → true : true ✓

palindrome("D") : true ✓

RADAR

Recursive Fibonacci



CSC 205 Recursion Worksheet

1. A palindrome is a number, word, or phrase that reads the same backwards as forwards. Examples include EVE, 8715178, and RADAR.

   Shown below is a recursive method that determines whether a given string is a palindrome. The base case of this method is a string of length 1 which is, of course, a palindrome. The recursive case strips and checks the outer letters and then sends the sends the word wihout the outer letters off to be checked as a palindrome.

   Trace through the palindrome method below with the string RADAR and prove that it is indeed a palindrome.

A Recursive Palindrome Checker Using a String

```
private static boolean palindrome (String word)
{
        if (word.length() <= 1)
                return true;
        else
        {
            char first = word.charAt(0);
            char last  = word.charAt(word.length()-1);
            String chopWord = word.substring(1,word.length()-1);
            return ((first == last) && palindrome(chopWord));
        }
}
```

base case

Here is an iterative version of this same algorithm that you may enjoy comparing the above method to. Remember that all iterative algorithms can be converted to recursive algorithms, and vice versa.

An Iterative Palindrome Checker Using A String Object

```
private static boolean palindrome (String word)
{
        int i = 0;
        int last = word.length()-1;
        while (i <= word.length()/2)
        {
                if (word.charAt(i) != word.charAt(last))
                        return false;
                i++;  last--;
        }
        return true;
}
```

2. The Fibonacci numbers are defined as the following sequence :

0 1 1 2 3 5 8 13 21 34 55 ...

Notice that except for the first two numbers, each number is the sum of the two preceding numbers.

i) Determine the base case and the recursive case of a method that returns the *n*th Fibonacci number where *n* is a parameter.
ii) Trace through the recursive Fibonacci method shown below with fib(5).
iii) Notice the iterative reversion of the Fibonacci method which is also shown below. Which version is more obvious or easier to understand?

Recursive Fibonacci

```
private static int fib (int n)
{
        if (n == 1)
                return 0;
        else if (n == 2)
                return 1;
        else
                return (fib(n-1) + fib(n-2));
}
```

Iterative Fibonacci - Much more efficient

```
private static int fib (int n)
{
        int sum = 0;
        int pre1 =  0;
        int pre2 =  1;

        if (n == 1)
                return 0;
        else if (n == 2)
                return 1;
        else
        {
                for (int i = 3;  i <= n ; i++)
                {
                        sum  = pre1 + pre2;
                        pre1 = pre2;
                        pre2 = sum;
                }
                return sum;
        }
}
```

The Oil Spill Program

Using Recursion To Peek In All Directions Within A Matrix

*	*	*	*	*	@
*	@	*	*	*	*
*	@	*	*	@	@
*	@	*	*	*	@
*	@	*	*	*	*
*	@	@	@	@	@

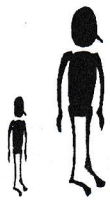
3 oil spills

```
public class OilSpill
{
  public static void main (String[] args)
  {
        int size1 = 8, size2 = 8;   int oilpockets = 0;
        char[][] field = new char[100][100];

        // Input The Data For Field Matrix (Pad Around Sides With '*')
        inputMethod (size1, size2, field);
        for (int i = 1; i <= size1-2 ; i++)
              for (int j = 1; j <= size2-2 ; j++)
                    oilpockets = checkCurrBlock(field, i, j, oilpockets,
                                                size1, size2);
        System.out.println("Total Oil Pockets = " + oilpockets);
  }

  private static int checkCurrBlock (char[][] field, int i, int j,
                                     int oilpockets, int size1, int size2)
  {
        if (field[i][j] == '@')
        {
                checkSurroundings(field, i, j, size1, size2);
                oilpockets++;
        }
        return oilpockets;
  }

  private static void checkSurroundings (char[][] field, int i, int j,
                                         int size1, int size2)
  {
        if (field[i][j] == '@')
        {
                field[i][j] = '*';
                checkSurroundings (field, i-1, j, size1, size2);
                checkSurroundings (field, i-1, j-1, size1, size2);
                checkSurroundings (field, i-1, j+1, size1, size2);
                checkSurroundings (field, i, j-1, size1, size2);
                checkSurroundings (field, i, j+1, size1, size2);
                checkSurroundings (field, i+1, j-1, size1, size2);
                checkSurroundings (field, i+1, j+1, size1, size2);
                checkSurroundings (field, i+1, j, size1, size2);
        }
  }
}
```

checking for out of bounds so creates a "guard"

8 recursive because 8 different directions to look

A Whole New Way To Do A Binary Search Using Recursion



```
public class Binary
{
    public static void main(String[] args)
    {
        String[] s = {"Karen", "John", "Matt", "Suzy"};
        System.out.println(binarySearch(s,0,s.length-1,"Matt");
    }

    public static int binarySearch(Object[] a, int first, int last,
                                   Object key)
    {
        if (first <= last)
        {
            int mid = (first + last) / 2;

            Object midVal = a [mid];

            int comp = ((Comparable)midVal).compareTo (key);

            if (comp < 0)
                return binarySearch (a, mid + 1, last, key);

            if (comp > 0)
                return binarySearch (a, first, mid - 1, key);

            return mid;  // key found

        } // if first <= last
        return -first - 1; // key not found; belongs at a [first]

    } // method binarySearch
}
```

built in compareTo

Array.out(s)

built in interface

purely abstract class

make sure if objects have compareTo method