

```
class Fibonacci
{
    // this solves problem P5.19 in the textbook,
    // in two ways:  recursively (bad), normal (good).

    public static int fibonacci( int n )
    {
        if( n <= 0 ) return 0;

        int fib = 1;
        int fibPrev    = 1;    // fib(1),  fib(i-2)
        int fibPrevPrev = 1;    // fib(2),  fib(i-1)

        for( int i=3; i <= n; i++ ) {
            fib = fibPrev + fibPrevPrev;

            // update the prev values
            fibPrevPrev = fibPrev;
            fibPrev     = fib;
        }

        return fib;
    }

    // this is a recursive version of fibonacci.
    // it is very inefficient because it makes
    // numerous repetitive calls to itself.
    // for example, fib(9) calls fib(8) and fib(7),
    // but fib(8) calls fib(7) and fib(6).  notice
    // how fib(7) was called twice there.
    // as the fibonacci proceeds, it gets even
    // more redundant.  See section 12.9 in textbook.
    private static int fibonacciRecursiveCalls;
    public static int fibonacciRecursive( int n )
    {
        fibonacciRecursiveCalls++;
        if( n <= 0 )
            return 0;
        if( n <= 2 )
            return 1;
        else
            return fibonacciRecursive(n-1) + fibonacciRecursive(n-2);
    }

    public static void main( String[] args )
    {
        // compute up to fib(30) as an example
        for( int i=1; i <= 30; i ++ ) {
            int fib = fibonacci( i );

            fibonacciRecursiveCalls = 0;
            int fibR = fibonacciRecursive( i );

            System.out.print( "fibonacci(" + i + ") = " + fib );
            System.out.print( "\tfibonacciRecursive(" + i + ") = " + fibR );
            System.out.println( "\tin " + fibonacciRecursiveCalls + " calls" );
        }
    }
}
```

```
class Find
{
    // this class solves problem P5.24 in the textbook
    // in two ways: one with recursion, one without.

    // this method looks for the string 't' inside
    // the supposedly larger string 's'
    public static int find( String s, String t )
    {
        int si=0, ti=0;
        while( si <= s.length() - t.length() )
        {
            // look inside String s, at position si
            String partOfS = s.substring( si, si+t.length() );
            // start ^^, ^^^^^^^^^^^^^^^^^ pastEnd

            if( partOfS.equals( t ) ) // compare strings
                return si; // found at position si

            si++;
        }
        return -1; // not found
    }

    // this method looks for the string 't' inside
    // the supposedly larger string 's'
    public static int findRecursive( String s, String t )
    {
        if( s.length() < t.length() )
            return -1; // base case, s is too small; not found

        // look at first part of S
        String firstPartOfS = s.substring( 0, t.length() );

        // check if String s starts off with String t
        if( firstPartOfS.equals( t ) ) // compare strings
            return 0; // found at position 0 of s
        else {
            // Recurse by removing first character from s,
            // and looking for t in the remainder of s.
            // If we found it, we must find the position in
            // s by counting as the recursion unrolls.
            int found = findRecursive( s.substring(1), t );
            if( found < 0 )
                return -1; // didn't find it, keep -1
            else
                return 1+found; // count the positions
        }
    }

    public static void main( String[] args ) {
        // use 4 simple test cases. more should be used.
        // first case: returns 1, the first match (multiple matches)
        // second case: returns 0, matching at beginning of s
        // third case: returns 9, matching at end of s
        // fourth case: returns -1, no match

        System.out.println( find( "Mississippi", "is" ) );
        System.out.println( find( "Mississippi", "Miss" ) );
        System.out.println( find( "Mississippi", "pi" ) );
        System.out.println( find( "Mississippi", "hip" ) );

        System.out.println( findRecursive( "Mississippi", "is" ) );
        System.out.println( findRecursive( "Mississippi", "Miss" ) );
        System.out.println( findRecursive( "Mississippi", "pi" ) );
        System.out.println( findRecursive( "Mississippi", "hip" ) );
    }
}
```

```
class Gcd
{
    public static int gcd( int m, int n )
    {
        System.out.println( "finding gcd("+m+", "+n+")" );
        if( n==0 || m==0 )
            return 0;
        else if ( m < n )    // always want n to be smaller
            return gcd(n,m);
        else if( m%n == 0 ) // if n evenly divides m, n is GCD
            return n;
        else                // reduce the problem
            return gcd(n,m%n);
    }

    public static void main( String[] args )
    {
        boolean done;

        do {
            int x, y;
            x = Stdin.getInt();
            y = Stdin.getInt();

            int g = gcd( x, y );

            System.out.print ( "GCD("+x+", "+y+") = " );
            System.out.println( g );

            done = ( x==0 && y==0 );
        } while( !done );
    }
}
```

```
class Hanoi
{
    // this solves Problem P5.21 of the textbook.

    public static void hanoi( int fromPeg, int toPeg, int n )
    {
        int tmpPeg = 1 + 2 + 3 - fromPeg - toPeg;
        // pegs are numbered 1,2,3 ... compute our tmp peg

        if( n == 1 ) {
            System.out.print ( "Move disk from " + fromPeg );
            System.out.println( " to " + toPeg );
        } else {
            // move n-1 disks from fromPeg to tmpPeg
            // move nth disk from fromPeg to toPeg
            // move n-1 disks from tmpPeg back to toPeg
            hanoi( fromPeg, tmpPeg, n-1 );
            hanoi( fromPeg, toPeg, 1 );
            hanoi( tmpPeg, toPeg, n-1 );
        }
    }

    public static void main( String[] args )
    {
        // move 4 disks from peg 1 to peg 2
        hanoi( 1, 2, 4 );
    }
}
```

```
class Permute
{
    // this is problem P9.13 from the textbook
    // it is similar (but more difficult than)
    // the phone number mapping quiz question.

    public static void permuteHelper( int[] prefix, int[] toPermute )
    {
        // this method must consider all orders of numbers
        // in "toPermute" and append them to "prefix".  it
        // does this recursively.
        if( toPermute.length == 0 ) {
            printArray( prefix );
        } else {
            int[] newPrefix = new int[prefix.length+1];
            int[] newPermute = new int[toPermute.length-1];

            // copy from prefix to newPrefix
            for( int i=0; i < prefix.length; i++ ) {
                newPrefix[i] = prefix[i];
            }

            for( int i=0; i < toPermute.length; i++ ) {
                // try all elements in toPermute

                // add each one to newPrefix
                newPrefix[prefix.length] = toPermute[i];

                // copy all elements except toPermute[i]
                // into newPermute array
                int k=0;
                for( int j=0; j < toPermute.length; j++ ) {
                    if( j != i )
                        newPermute[k++] = toPermute[j];
                }

                // call ourselves recursively
                permuteHelper( newPrefix, newPermute );
            }
        }
    }

    public static void permute( int n )
    {
        int[] prefix = new int[0];
        int[] toPermute = new int[n];

        // initialize the array of values to permute
        for( int i=0; i<n; i++ )
            toPermute[i] = i;

        permuteHelper( prefix, toPermute );
    }

    // this method prints the contents of an array
    public static void printArray( int[] a )
    {
        for( int i=0; i < a.length; i++ ) {
            System.out.print( a[i] + " " );
        }
        System.out.println(); // go to next line
    }

    public static void main( String[] args )
    {
        permute( 4 );
    }
}
```