Problem 2.1 CES 512 - D. Bozarth - 13 March 2005

REQUIREMENT:

Java Runtime Engine. (Developed on JDK 1.4.2).

INVOCATION:

```
"java Ex_2_1 <enter>"
```

LIST OF FILES

```
Ex_2_1$Pair.class         class Pair         (bytecode file)
Ex_2_1.class              class Ex_2_1       (bytecode file)
Ex_2_1.java               class Ex_2_1       (source file)
Ex_2_1_Description.txt    (this file)
outfile.txt               (to be created by program)
```

CAPABILITIES

Define H(n) as the number of ways to arrange any combination of 1x1, 2x1, and 1x2 tiles, to exactly fill an L-shaped region with identical arms, each of which arm has width 1 and length n.

The program fulfills two separate requirements, selectable by user input:

1. Given a single integer input "n", report the exact value of H(n).

2. Given a single integer input "n", report an estimate for the value of H(n!).

The program provides two additional features:

1. Echoes program output to a file.

2. Offers the user a choice to report only H(n), or to successively report (k and H(k)) for every (0 <= k <= n).

CLASS STRUCTURE

Class Ex_2_1 provides the basic functionality to calculate H(n) for integer values of n. Its contained Class Pair structures a pair of BigIntegers. The method g(int) is called by the method h(int), and returns such a Pair, for use by h(int).

Also in use are the static method fac(int) which returns the BigInteger factorial of any integer; the static method getNumberDigitsInH_Fac(int) which returns a BigInteger estimate of the value of H(n!), and the static method main(String args[]).

A derived class Ex_2_1_Big was built and used in an attempt to calculate H(n) for unlimited sizes of n. This didn't work out; one attempt to calculate H(10!) continued for about 10 hours without terminating.

THEORY OF OPERATION

Define G(n) as the number of ways to arrange any combination of 1x1, 2x1, and 1x2 tiles, to exactly fill a single linear region of width 1 and length n. Then H(n) can be related to G(n) as follows:

```
       H(n) =  (number of ways to first place a single 1x1 at the vertex of the "L", and then
fill the remaining space)
               +
               (number of ways to first place a single 2x1 at the vertex of the "L", and then
fill the remaining space)
               +
               (number of ways to first place a single 1x2 at the vertex of the "L", and then
fill the remaining space)


   =>  H(n) =  ( 1 * pow(G(n-1), 2) )  +  ( 1 * G(n-1)*G(n-2) )  +  ( 1 * G(n-1)*G(n-2) )

   =>  H(n) =  { pow(G(n-1), 2)  +  2 * G(n-1)*G(n-2) , 1 < n

                 n                                    , n = 0, 1
                }
```

The running time of this recurrence increases exponentially with n.

A straightforward modification reduces the running time to order n. Note that

```
       G(n) =   (number of ways to first place a single 1x1 at the end of a 1xn column, then fill
the remaining space)
               +
               (number of ways to first place a single 2x1 at one end of a 1xn column, then fill
the remaining space)


   =>  G(n) = { G(n-1) + G(n-2),  n > 1
                 1             ,  n = 0, 1
               }
```

This is the Fibonacci sequence.

Thus it is necessary only to calculate and store successive pairs of Fibonacci numbers with index pairs (0, 1) through (n-2, n-1) - using the stored pair from the previous step to find the current pair - then perform 3 multiplications and one addition.

This can be done in linear time. The rendering of Fibonacci pairs is implemented by the method g(int), and the final calculations are done by h(int).

The resulting code can rapidly provide the value of H(1000):

```
4224696333392304878706725602341482782579852840250681098010280137314308584370130707224123599639141
5110884460875389096036076401947116435960292719833125987373262535558026069915859152294924539049987
2225679531698287448247299226390183371677806060701161549788671987985831146887087626459736908672288
4023654422295243347964480139515349562972087652656069529806499841977448720155612802665404554171717
881930324025204312082516817125
```

This number has 418 digits.

In order to estimate the number of digits in H(n) for very large n - specifically H(100!) - I found H(k!) for small values of k, and made a list of the number of digits:

k	#digits
0	1
1	1
2	1
3	3
4	10
5	50
6	301
7	2107
8	16853
9	151675

As mentioned above, I found it impractical to calculate H(10!). This is a very interesting list, though, because it shows that if

```
M(k) = [ #digits in H(k!) for 0 <= k <= 9 ], then

M(k) = { 1                    , k < 3
         3                    , k = 3
         k * M(k - 1) + c     , 3 < k, -3 <= c <= 1
       }
```

Since c is known for each k in this range, M(k) is exact, and

```
M'(k) = [ #digits in H(k!) for 9 < k <= max(int) ]
```

can be estimated as

```
M'(k)  ≈  k * M(k - 1)
```

Thus it was straightforward to build a linear-time algorithm to estimate M'(k). Specifically, since (100!) is within the range of Java integers, M'(100) was estimated as

```
3900808456641377137895924838383005896420123590856504198701729993057520248835192749665508308141407
603358556446247097983777635904061440000000000000000000000000 digits,
```

which is about

```
4 x 10e157 digits.
```