Chapter 73 Programming Exercises

Exercise 1

Write a program that implements this definition of ***cube numbers*** (for positive integers):

```
cube(1) = 1
cube(N) = cube(N-1) + 3*(square(N)) - 3*N + 1
```

Implement the `square()` method using this definition given in the exercises for chapter 71 (also for positive integers):

```
square(1) = 1
square(N) = square(N-1) + 2*N - 1
```

Make a complete program similar to *PyramidTester.java* given in the chapter.

Extra: The math-like definition of square(N) does not work for negative N because the recursive step asks for square(N-1), which for negative integers is a harder problem than the original. For example, square(-5) asks for square(-6).

For a definition of square(N) that works for negative integers (only) do this:

$$(N+1)^2 = N^2 + 2N + 1$$

$$N^2 = (N+1)^2 - 2N - 1$$

Create a similar definition for cube(N) that works for negative integers. Write cube(N) that works for negative integers (only).

Extra, Extra: Write cube(N) that works for all integers, negative, positive, and zero. Hint: use an if-statement and three base cases.

Click here to go back to the main menu.

Exercise 2

Consider this definition of the sum of the elements in an integer array:

```
sum( array, index ) = 0, if index == array.length

sum( array, index ) = array[index] + sum( array, index+1), if index < array.length
```

Write a Java method that implements this definition and a program to test it. The method should look something like:

```
int sum ( int[] array, int index )
{
 . . .
}
```

The testing program will call `sum( testArray, 0 )`.

Click here to go back to the main menu.

Exercise 3

Improve the previous program by extending the definition of `sum()` so that the user does not need to include that odd-looking zero in the first call to the function:

```
sum( array ) = sum( array, 0 )

sum( array, index ) = 0, if index == array.length

sum( array, index ) = array[index] + sum( array, index+1), if index < array.length
```

To implement this, write a second method `sum( int[] array)` that overloads the method of exercise 1. The testing program will call `sum( testArray )`.

Click here to go back to the main menu.

Exercise 4

Write your own recursive definition of the **maximum element** in an array of integers. Then, implement your definition in Java and test it with a testing program. (Do exercise 2 before you do this one. You will need one base case and two recursive cases.)

Click here to go back to the main menu.

Exercise 5

Consider this definition of the Perrin sequence:

```
perrin(0) = 3
perrin(1) = 0
perrin(2) = 2

perrin(N) = perrin(N-2) + perrin(N-3)
```

Write a Java method that implements this definition and a program to test it.

Click here to go back to the main menu.

Exercise 6

A combination is a way of selecting several items out of a group of items where order does not matter. For example, say that you have five colors, Red, Orange, Yellow, Green, and Blue and you want to select two of them. There are 10 ways this can be done:

```
(R, O)  (R, Y)  (R, G)  (R, B)
        (O, Y)  (O, G)  (O, B)
                (Y, G)  (Y, B)
                        (G, B)
```

Often you want to know how many combinations are possible when N things are selected from M items. In the above example, N is 2, and M is 5. A formula for this is;

```
Comb( M, N ) = M! / (N!(M-N)!)
```

Unfortunately, the factorial function quickly gets very big. If you try to use this formula directly, you will likely discover that M! "blows up" even when the final value of `Comb( M, N )` is within the range of data type int. For example,

```
15! = 1 307 674 368 000
```

well beyond the range of `int`. However,

```
Comb( 15, 5 ) = 3003
```

which easily fits into an `int`. To use the formula, you need to be fairly clever to get common factors in the numerator and denominator to cancel.

A recursive formula is:

- if `M < N, Comb( M, N ) = 0`
- if `N < 0, Comb( M, N ) = 0`
- `Comb( M, 0 ) = 1`
- otherwise, `Comb( M, N ) = Comb( M-1, N-1 ) + Comb( M-1, N )`

This formula does not use any multiplication at all!

Write a program that implements the recursive method of `Comb(M,N)`. If you want, implement the non-recursive formula as well and see how the two methods compare. You may wish to use data type `long` in an attempt to avoid overflow.

Click here to go back to the main menu.
