The SDP Primer

Exercise sheet 03

Session: 2016-17

1 Purposes of these exercises

- To give you more practice with Scala, methods in particular
- To stress the importance of following specifications exactly

2 General idea of the exercises

For each of the numbers 1 through 100, determine some characteristics of those numbers. Print out the numbers, one per line, along with a list of its characteristics (on the same line). Your output should look approximately like this:

```
1   c, h, t, s, not sm, ht, not pr, d
2   p, u, not t, not s, sm, ht, pr, d
3   p, u, t, not s, not smug, ht, not pr, d
4   c, u, not t, s, not sm, ht, not pr, d
5   p, u, not t, not s, sm, dis, not pr, d
6   c, u, t, not s, not sm, dis, pr, per
...
```

where

```
c = composite
d = deficient
dis = dishonest
h = happy
ht = honest
per = perfect
p = prime
pr = pronic
sm = smug
s = square
t = triangular
u = unhappy
```

3 Details

3.1 General

Create an object `NumberPersonalities` in a file `NumberPersonalities.scala`. Within the `NumberPersonalities` object, declare a variable as `val limit = 100`. Within the `NumberPersonalities` object, write a

```
def main(args: Array[String])
```

method to test the numbers `1` through `limit`, inclusive, and print out the results in the form shown above. In `main`, do *not* assume that `limit` is 100, but write your program to handle *any* positive `Int`. This makes it easy to change the limit later.

`main` should probably be the *last* method you write, after you have written *and tested* all the methods that it will need to call.

A *predicate* is a method that returns a `Boolean` value (`true` or `false`). Within the `NumberPersonalities` object, write the following predicates:

- `isPrime(n: Int): Boolean`
- `isHappy(n: Int): Boolean`
- `isTriangular(n: Int): Boolean`
- `isSquare(n: Int): Boolean`
- `isSmug(n: Int): Boolean`
- `isHonest(n: Int): Boolean`
- `isPronic(n: Int): Boolean`
- `isDeficient(n: Int): Boolean`
- `isPerfect(n: Int): Boolean`
- `isAbundant(n: Int): Boolean`

And also write the following method:

- `sumOfPositiveDivisorsOf(n: Int): Int`

You should provide tests for all your methods.

- Testing does *not* depend on the parameter names (`n` in each of the above). You can use whatever parameter names you like. For these methods, however, `n` is as good a name as any.
- In most cases, Scala can deduce the return type (`Boolean` in all but one of the above methods), and you don't have to specify it.

3.2 Prime and composite numbers

A positive number is **prime** if its only positive divisors are itself and one. For example, `7` is prime because it is evenly divisible by `1` and `7`, but not by any number in between (in particular, it is not evenly divisible by `2`, `3`, `4`, `5`, or `6`).

A positive number is **composite** if it is not prime. For example, `10` is composite because it is evenly divisible by `2` and `5`; `12` is composite because it is evenly divisible by `2`, `3`, `4`, and `6`. As a special case, `1` is considered to be composite.

> **Programming note.** You can test whether a number `n` is prime by dividing it by each of the numbers `2` through `n-1`, and testing if the remainder is zero (use the `%` operator). A zero remainder means `n` was evenly divisible by that divisor, hence is composite.

3.3 Happy and unhappy numbers

Repeatedly apply the following procedure to a number:

1. Square each of the digits of the number.
2. Add the squares together.

If by doing this you eventually get to `1`, then the number is **happy**. For example, if you start with `19`:

- `12 + 92 = 1 + 81 = 82`
- `82 + 22 = 64 + 4 = 68`
- `62 + 82 = 36 + 64 = 100`
- `12 + 02 + 02 = 1 + 0 + 0 = 1` and the number is happy.

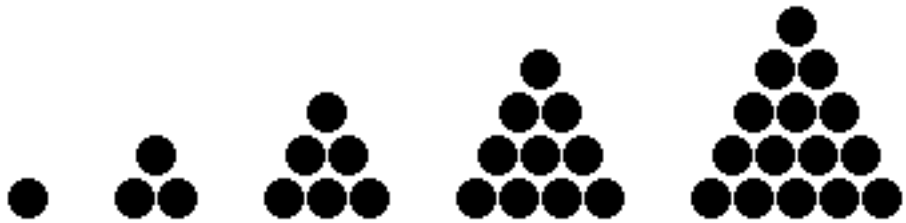
If instead you get into an infinite loop, the number is **unhappy**. So if your program runs forever, the number is unhappy. This isn't a very useful test, however. Fortunately, every unhappy number will eventually get into this cycle:

```
4, 16, 37, 58, 89, 145, 42, 20, 4, ...
```

so if you get any one of these numbers, say `4`, then you can stop and conclude that the number is unhappy.

> **Programming note.** You can get the last digit of a number `n` with the expression `(n % 10)`. You can subsequently discard this digit by doing an "integer division" by 10; when you get a zero, there are no more digits.
>
> Example: Taking the number `1024`:
>
> ```
>                  1024          % 10 = 4          1024
> / 10 = 102, 102 % 10 = 2          102 / 10 = 10,   10
> % 10 = 0           10 / 10 = 1,     1 % 10 = 1            1
> / 10 = 0 (stop)
> ```

3.4 Triangular numbers

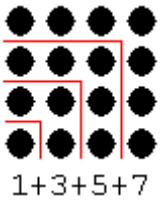


A **triangular number** is a number of the form `1 + 2 + 3 + ... + n`, for some positive `n`. These numbers are called triangular because, if you have that many objects, you can arrange them in an equilateral triangle (see figure). The first few triangular numbers are `1, 3, 6, 10, 15...`.

> **Programming note.** You can test if a number is triangular by subtracting the number 1 from it, then subtracting 2, then subtracting 3, etc. If at some point you get zero, then the starting number is triangular; but if you first get a negative number, then the number is not triangular.
>
> Example:
>
> `6 - 1 = 5` `5 - 2 = 3` `3 - 3 = 0` and the number is triangular.

3.5 Square numbers

A **square number**



is a number of the form `1 + 3 + 5 + 7 + ... + n`, for some odd positive `n`. (The figure should help you understand why this definition is the same as the usual definition of square numbers.)

> **Programming note.** To test if a number is **square**, do the following. For each of the odd numbers `1 + 3 + 5 + 7 + ...`, subtract that number, and see if you get zero (square) or a negative number (not square).

3.6 Smug numbers

A number is **smug** if it is the sum of two square numbers. The first few smug numbers are `2` (1+1), `5` (1+4), `8` (4+4), `10` (1+9), `13` (4+9), etc.

> **Programming note.** To test if a number is smug, do the following. For each of the square numbers (which you can get as `1*1, 2*2, 3*3`, etc.), subtract that number, and see if the remainder is also a square number. You can stop if the remainder is less than the square number you subtracted.

3.7 Honest and dishonest numbers

A number is **dishonest** if it "pretends" to be a square number, but isn't one. Specifically, n is dishonest if there is a number `k` such that `n/k = k` (using integer division), but `k*k` is not `n`. A number is **honest** if it is not dishonest.

> **Programming note.** To find `k`, try the numbers 1, 2, 3, ... until you find a `k` such that either `n/k = k` or `n/k` $<$ `k`.

3.8 Pronic numbers

A number is **pronic** if it is the product of two consecutive numbers. The first few pronic numbers are `2` (1*2), `6` (2*3), `12` (3*4), `20` (4*5), etc.

> **Programming note.** Starting with `k = 1`, compute `k * (k + 1)` until the result either equals or exceeds the number you are testing.

3.9 Deficient, perfect, and abundant numbers

Every positive number is either **deficient**, **perfect**, or **abundant**. To determine which it is, add up all the positive numbers (including `1`) that divide it evenly.

- If the sum of the divisors is less than the number, the number is **deficient**.
- If the sum of the divisors exactly equals the number, the number is **perfect**.
- If the sum of the divisors is greater than the number, the number is **abundant**.

For example, `24` is evenly divisible by `1`, `2`, `3`, `4`, `6`, `8`, and `12`. Because `1+2+3+4+6+8+12 = 36`, and `36` $>$ `24`, the number `24` is abundant.

A fundamental rule of good programming is the **DRY Principle**: *Don't Repeat Yourself!* In each of three methods (`isDeficient`, `isPerfect`, and `isAbundant`), you need to compute the sum of the divisors of a number. It's poor style to write this same code in all three methods; instead, write a `sumOfPositiveDivisorsOf` method that each of the three methods can use.