Recursion is one of the key concepts in computer science – the idea that methods (recursive functions) or data (recursive data structures) can be described in terms of themselves. The most important thing to remember is that these definitions must not be circular or they cannot be resolved. The definitions must always have:

- a simple base case or cases (non-recursive), and
- rules that unambiguously reduce all other cases towards the base case.

For instance, in mathematics, a function on the natural numbers might be defined in terms of a given value at 0 (the base case), and some rules for computing its value at a natural number n in terms of its values at smaller numbers (see, the factorial example from the lectures).

In this lab you will first gain some experience with basic recursive functions, and then explore a simple recursive data structure.

Problem description

In the first part we'll be looking at functions that are related to the digits of a number. Specifically, given an integer (represented as a **`long`**) we want to write functions that: determine the number of digits it has, and determine the sum of those digits. It is easy to describe the function that computes the number of digits of a *positive* integer n recursively:

```
digits($n$):
  **if** $n < 10$ **then**
    return 1
  **end if**
  return $1 + \text{digits}(n/10)$
```

In the pseudocode above, the division $n/10$ should of course be "integer division" as Java does it, i.e., the result is the integer k such that $10 \times k \leq n < 10 \times (k+1)$. A small modification is needed to ensure that it works properly on negative integers as well. The function that takes the sum of the digits of n (e.g., $\text{sumOfDigits}(257) = 14$) has a very similar recursive description.

In the second part we'll begin with a recursive data structure that represents a tower of blocks. We represent such a **`Tower`** by its top block which for convenience we'll just take to be a **`char`** representing its colour, i.e., something like **`'B'`**, **`'G'`**, **`'R'`** or **`'Y'`**, together with another **`Tower`** representing the rest of the blocks. The basic constructors and some utility functions are supplied. Your job will be to work out how to compute the height of a tower and the number of blocks in a tower of a given colour.

Part one (1%)

Create an application file called **`RecursiveApp.java`** that contains the following two functions:

`digits(long n)` Returns a **`long`** equal to the number of digits of its argument

`sumOfDigits(long n)` Returns a **`long`** equal to the sum of the digits of **`n`** modified by the sign of **`n`**. That is, **`sumOfDigits(257)`** should return 14, whereas **`sumOfDigits(-257)`** should return -14.

Part two (1%)

Copy the file **`Tower.java`** from the `/home/cshome/coursework/241/pickup/03/` directory into your `~/241/03/` directory, and then add the following methods to it:

`height()` A method that returns an **`int`** equal to the height, i.e., number of blocks, in the tower.

`count(char c)` A method that returns an **`int`** equal to the number of blocks equal to **`c`** in the tower.

Marking

Check that your programs work correctly, and then use the command `241-check` to make sure they pass all of our tests. If all is well then you can submit using `241-submit` as usual. If you only complete part one or part two then `241-submit` will allow you to submit the completed part. Both of your programs should be in the package `week03`.

Reflection and extension

- How could we write a method **`take(int k)`** that takes the top **`k`** blocks from the tower, discarding the ones below them? What about **`remove(int k)`** that removes the top **`k`** blocks?

- In the **`Tower`** class we use a space character to represent "no block" and hence an empty tower. Suppose that we have an existing tower, **`t`**. What happens if we do something like:
  `t = t.add(' ')`?
  Assuming you think this is undesirable (you should), how could we address this issue? There are several possibilities ...
- How could you write a method **`stack(Tower other)`** that stacks another tower on top of this one?
- How could the recursive methods you've defined be replaced by non-recursive ones? Would this be a sensible thing to do?