This week you'll continue to do some timing experiments using some recursive methods. Also, you'll begin to familiarise yourselves with the union-find code needed for doing Assignment 1.

Some hints:

- Use a loop to automate testing across a range of n. Ranging n linearly is easy and often makes sense, although sometimes it can be better to (say) double it each iteration to cover a greater range more quickly.
- Consider adding an additional loop to compute the mean time taken for each value of n. As we saw last time, a single sample can be quite noisy, especially when the times are short.
- Structure your timing output for ease of import into a spreadsheet (tab- or comma-separated values, line-separated records). For example, output each value of n, $fib(n)$, and the time taken on a single line.
- Use a spreadsheet (or other tool) to plot and analyse the results.
- If your program is taking too long to run, you might need to kill it. Use Ctrl-C in the terminal or press the Stop button in your IDE.

Experiment 1

This experiment illustrates the very poor behaviour of recursive algorithms when they require more than one recursive call to cases that aren't much smaller. The standard example for illustrating this is the Fibonacci numbers, and who am I to argue with standards? So, recall that the Fibonacci numbers are the sequence:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

where each number in the sequence (after the pair of 1's) is the sum of the two preceding numbers. That is, (indexing from 0):

$$f_0 = f_1 = 1,\ f_n = f_{n-1} + f_{n-2} \text{ for } n > 1.$$

The skeleton outline for this experiment includes a basic method for computing the Fibonacci numbers using this as a recursive definition as well as some other methods.

1. Look at the times required to compute `fibRec(n)` for various values of n. How do these times behave? How large a value of n can you (realistically) apply this method to?
2. Do the other implementations produce the same sequence of values? Aside from "running the code and checking" are you *confident* of this (based on reading the code)?

3. How large a value of n can you apply the other methods to in a reasonable length of time? What's the limiting condition?
4. Which of the other methods is "best"?

Experiment 2

Implementations of the different union-find algorithms that we've discussed in lectures are provided in the `cosc201.unionfind` package. The main point of this experiment is to think about ways of testing them. This is really a warm-up for your work on Assignment 1. It also introduces the idea of using an input text file as a sort of script for testing purposes.

1. Read the code of `UF1`, `UF2`, and `UF3` to confirm to your satisfaction that they should do what is suggested.
2. Write a class that can read a file (or from `System.in`) and carry out various union-find tasks as indicated below. You can do this from scratch, or extend the skeleton provided in `Lab2Exp2`.

   The first line of the input file consists of a single integer n. In repsonse to this, your program should create a union-find instance, `X`, of size n, and then output `Size <n>` (with the obvious replacement). Each remaining line of input is of one of the following forms:

   - `Union a b`
     Echo (i.e., print as output) the input and then perform `X.union(a,b)`.
   - `Find a`
     Echo the input, then a colon, a space, and the result of calling `X.find(a)`.
   - `All a`
     Echo the input followed by a colon, a space, and all the elements of a's current set in increasing order.
   - `Summary` (Stretch goal)
     Echo the input, followed by a sequence of lines containing each of the sets exactly once (and each in increasing order). Ideally, the set containing 0 should be printed first, then the set containing the smallest element not in 0's set, and so on.

   Don't worry too much about efficiency here at this point, but you will probably find (particularly with the `Summary` item) that you would like to have access to some more sophisticated data structures for handling this. We'll be seeing some of those later on in the paper! To help you along, pseudocode for (a possible) implementation of `All` and `Summary` is given at the end of this document.

3. If you have different (but correct) implementations of union-find, for which of the output-producing operations above must they give the same result?

Reflection and extension

- Do any of the alternative methods for computing Fibonacci numbers suggest general principles for making (some) recursive computations more efficient? (We'll be examining this later in the semester.)
- What currently happens with your code for experiment 2 if the input is badly formatted? How could you make it more robust?

Example for Experiment 2

This is just a very basic example illustrating the intended behaviour.

If the input file is:

```
5
Union 2 3
Union 1 4
Find 1
Union 4 0
All 0
Find 4
Summary
```

Then the output could be (some might depend on the exact choice of algorithm or implementation in union-find):

```
Size 5
Union 2 3
Union 1 4
Find 1: 4
Union 4 0
All 0: 0 1 4
Find 4: 0
Summary
0 1 4
2 3
```

Pseudocode for `All` and `Summary`

The pseudocode below describes one way of coding the two extensions you'll need to add - these are perfectly suitable for smaller examples.

```
All(x):
  Initialise an empty ArrayList for the result to be returned
  Let r = find(x)
  for i from 0 to n-1:
    if find(i) is equal to r:
      add i to the result
  return the result
```

```
Summary:
  Initialise an empty arraylist "seen"
  (for the representatives that have been seen)
  for i from 0 to n-1:
    Let r = find(i)
    if r is not in seen:
      Add r to seen
      Print All(i)
```