CSCM12 – Software concepts and efficiency
February 17th 2023

Tonicha Crook & Cécilia Pradic

# Lab 2: recursive functions

- No need to complete lab 0 before you do lab 2; there is no dependency between the two.
- For sign-off, if the exercise is code, you are expected to show one working copy of the code. Otherwise, write down a worked solution (i.e., as you would in an exam) to show us.
- The questions marked as **Challenge** are not required for signing off, but I would encourage you to look at them if and only if you have time to spare.

1. **Assessing time complexity** For each of the java function below, assess its asymptotic time complexity in the worst case scenario with a $\mathcal{O}$:

   (a)
   ```
   static void func1(int[] a, int[] r)
   {
     int n = a.length;
     for(int i = 0; i < n; i++)
       for(int j = 0; j < n; j++)
         r[(i+1)*(j+1)-1] = a[i] * a[j];
   }
   ```

   (b)
   ```
   static void func2(int[] a, int[] r)
   {
     int n = a.length;
     for(int i = 0; i < Math.sqrt(n); i++)
         r[i] = a[i*i];
   }
   ```

   (c)
   ```
   static double naivePow(double a, int n)
   {
     if(n == 0)
       return 1;
     else if(n < 0)
       return 1/naivePow(a,-n);
     else
       return a * naivePow(a, n - 1);
   }
   ```

   (d)
   ```
   static double evalPoly(double[] p, double v)
   {
     int n = p.length;
     double r = 0;
     for(int i = 0; i < n; ++i)
       r += p[i] * naivePow(v, i);
     return r;
   }
   ```

   (e)
   ```
   static int horner(ArrayList<Integer> p, int v)
   {
   ```

```java
    if(p.size() <= 0)
      return 0;
    else
    {
      final int p0 = p.remove(p.size());
      // you can assume that p.remove(p.size()) is O(1)
      return p0 + v * horner(p, v);
    }
  }
```

(f) What do the last two function compute? (hint: think of $p$ as representing $x \mapsto \sum_{i=0}^{n-1} p_i x^i$)

(g) **Challenge** Prove that the $\mathcal{O}$s you have are actually $\Theta$s.

2. **Fun with fractals** Fractals are geometric shapes that are often nice to describe using recursion. For this question, we will be drawing some fractals using the graphics library from java. First, download the file `Fractals.java` from canvas. It comes pre-filled with a bit of boilerplate code for displaying stuff and a `Turtle` class[1].

   The basic idea is that we may pretend that we are drawing thanks to a turtle walking on the screen. The turtle is always located somewhere, represented by the $x$ and $y$ attribute, and facing some direction represented by `orientation`, which stores the angle with the $x$-axis (in radians). The method `turnLeft` can be used to change orientation, and `walk` to make the turtle move forward in the current direction by a certain distance, printing a line in the process. The method `fly` is the same as `walk`, but it will not induce any drawing.

   In the boilerplate code, I expect you to modify the `paint` method of `MyCanvas` and to add methods to `Turtle`[2]. You should not need to modify attributes directly in those new methods.

   (a) First run the code and explain the `walkEquilateralTriangle` method.

   (b) Implement a `turnRight` method similar to `turnLeft`, but which makes the turtle turn to the right. Also implement a method `turnAround`.

   (c) Implement a method for `Turtle` with signature

   ```java
   static void randomWalk(double dist, int nbSteps)
   ```

   which implements the following random walk: at each step, the turtle rotates randomly[3] in one of the four cardinal directions and move over `dist` pixels. Try out your function on reasonably large values. For `nbSteps = 50`, does you turtle ever loop?

   (d) The *Koch* curve is perhaps the most popular example of a first fractal. It can be defined as the limit of the sequence of curves which can be defined recursively as follows:

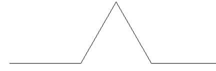
   - at order 0, just draw a segment

[1] This is inspired from the Logo language `https://en.wikipedia.org/wiki/Logo_(programming_language)`.

[2] Unless there are some quick fixes to be made in the made function owing to screen size or something of that nature, but I expect things should work as-is.

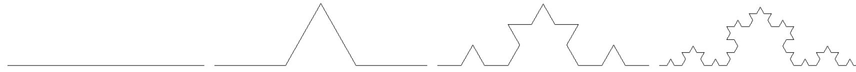
[3] If you do not remember how to draw a random number, please look at the provided code, it has some hints

- at order $(n + 1)$, consider the following shape

  where the middle part is an equilateral triangle minus its base. Take that and replace every segment by the Koch curve at order $n$.

The first few iterations can be represented as follows:



This can easily be done using the turtle by rephrasing the above process recursively

i. First, write a method

   ```
   static void walkKoch1(double dist)
   ```

   that prints out the curve at order 1; each atomic segment should have size $\frac{\texttt{dist}}{3}$, so that the overall straight line distance covered by the turtle should be `dist` (hint: recall that all the inner angles of an equilateral triangle are $\frac{\pi}{3}$

ii. Now write the general function

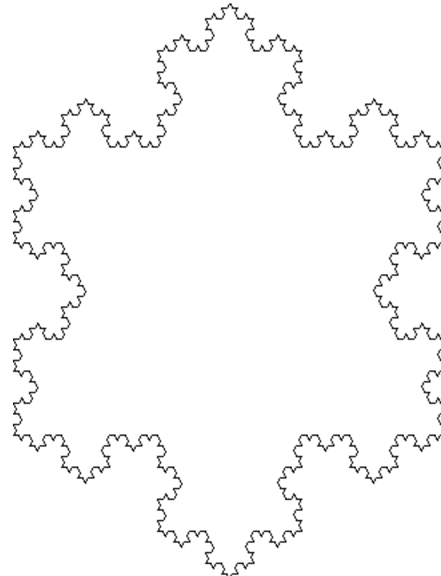
   ```
   static void walkKoch(double dist, order n)
   ```

   that prints out the Koch curve at an arbitrary order. For the recursive case, I suggest to start from the code from `walkKoch1`, which can presumably be adapted by replacing calls of the `walk` method by suitable recursive calls.

iii. Finally write a function

   ```
   static void walkKochFlake(double dist, order n)
   ```

   that outputs the following

(e) **Challenge** draw some other fractals; one nice one would be the dragon curve which may be obtained using a mutual recursion:

- At order 0, just draw a segment.
- At order $n+1$, draw a dragon curve at order $n$, turn $\frac{\pi}{2}$ to the left, and draw a *flipped* dragon curve at order $n$.

The flipped dragon curve of order $n$ is drawn by:

- Drawing a segment at order 0
- At order $n+1$, first drawing a non-flipped dragon curve of order $n$, turning $\frac{\pi}{2}$ **to the right** and then drawing a flipped dragon curve of order $n$.

You may find inspiration from wikipedia, or you can have fun by varying the angles, the distance and adding a modicum of randomness! Feel free to look at the documentation of `Graphics` to do fancier stuff too.

3. **Our first sorting algorithm** Let us try to introduce our first algorithm to sort arrays. This one is known as *bubble sort*. You can use either recursion or loops (or mix them) to solve this exercise.

   (a) Write a function

   ```
   static void bubbleDownStep(int[] arr, int i)
   ```

   that takes as input an array $a$ and an index $i$ of that array, and that swaps the elements of index $i$ and $i+1$ if $a(i) > a(i+1)$. This function should run in constant time ($\mathcal{O}(1)$).

   (b) Deduce a function

   ```
   static void bubbleDown(int[] arr, int i)
   ```

   that takes as input an array $a$ and an index $i$ of that array, and assuming that `arr` is sorted up to index $i$, applies `bubbleDownStep` a number of times so that `arr` is sorted up to index $i+1$. This function should run in linear time ($\mathcal{O}(n)$).

   (c) Deduce a function

   ```
   static void bubbleSort(int[] arr)
   ```

   which sorts the array `arr`. What is its asymptotic complexity?