CSCM12 – Software concepts and efficiency February 3rd 2023
Tonicha Crook & Cécilia Pradic

Lab 0: getting started

- As mentioned in the first lecture, the class session is not marked per se, but will net you easy bonus points for the coursework if you engage. Further, I am not sure it will even be possible to assimilate the lecture materials further down the line if you do not practice, so I would *strongly* recommend you do them.
- You can work in groups of up to four; I would suggest you attempt to at most pair up to solve the exercises and then discuss with the other pair of your group once you both have a solution to compare maybe? Working alone is fine too.
- 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.

The rough idea for this lab sheet is to brush up a bit your java skills, expose you to pseudo-code, some toy problems in algorithmic complexity, and introduce some notions that will be useful for the next lecture and the subsequent labs. I hope that you won't be bored and that you'll have fun :) (this brand new lab sheet was written fairly late at night, I hope it does not contain too many bugs; but it is probably long. Do not worry if you cannot finish everything; try to call us for sign-offs at the 90min marks if possible).

1. **Warm-up** Write pseudo-code for a function that takes as input an array of integers and outputs the minimal value. Then, write the same thing in java. You may assume that the array is non-empty.

   Challenge: show that your code does the minimal number of comparisons possible between elements of the arrays.

2. **Addition in base 10** Let us consider the long sum addition algorithm that we recalled on Monday, but written out to be executed by a computer (Algorithm 1, whose pseudo-code you may find on the next page).

   Run this algorithm by hand on the inputs $[7, 7]$ and $[1, 5, 2]$ by hand, detailing at each step of the for loop the values of X, Y, R and i.

   Then, write the corresponding java code and test it on the same example (hint: you can use `x % 10` to compute the last digit of `x`).

   Challenge: write the pseudo-code and the java code for long multiplication in base 10.

3. **Maths: revisiting limits** Just a couple of exercises about limits of functions when their parameters tend to infinity. If you struggle, feel free to ask for help, either from us or your classmates; this is not about testing you, but merely to ease you into some discussions we are going to have on Monday.

Input: Two arrays $X = [x_0, \ldots, x_n]$ and $Y = [x_0, \ldots, y_k]$ and with all $x_i, y_i \leq 9$ for all i representing two numbers in base 10
Output: An array representing the sum of the two numbers in base 10

```
SumBase10(X, Y)
    Resize X or Y so that they have the same size n > 0
    Create an array of integers R of size n + 1 with 0s in it
    for i from 0 to n − 1 do
        R[i] ← R[i] + X[i] + Y[i]
        if R[i] ≥ 10 then
            R[i] ← the last digit of R[i]
            R[i + 1] ← 1
        end
    end
    return R
```

Algorithm 1: Long addition in base 10

For a function $f : \mathbb{R} \to \mathbb{R}$ (and, mutatis mutandis, for a sequence $f : \mathbb{N} \to \mathbb{R}$), we say that

- the limit of f at $+\infty$ is some $L \in \mathbb{R}$ if f can stay arbitrarily close to L after some point. Formally, this is the case when for every small $\varepsilon > 0$, there exists a large enough $B \in \mathbb{N}$ such the distance between $f(x)$ and L for $x \geq B$ is inferior to ε. In symbol soup:

$$\forall \varepsilon > 0.\ \exists B.\ \forall x \geq B.\ |f(x) - L| < \varepsilon$$

- the limit of f at $+\infty$ is $+\infty$ if f can remain above an arbitrarily high threshold after some point. Formally, it means that for any big N, there is a large enough B such that f remains above N from that point on. In symbol soup:

$$\forall N.\ \exists B.\ \forall x \geq B.\ f(x) > N$$

- the case of $-\infty$ being a limit is similar

In any case, when ℓ is the limit of a function f at $+\infty$, we write $f(x) \xrightarrow{x \to +\infty} \ell$ or $\lim\limits_{x \to +\infty} f(x) = \ell$.

A word of caution: not every function has a limit in general. For instance the sequence $n \mapsto (-1)^n$ does not converge. However, in the context of this module, we are only going to need functions that are ultimately increasing (i.e. such that $x \leq y \Rightarrow f(x) \leq f(y)$) and these always have limits.

Furthermore, while the definitions above may seem a bit scary, we will ultimately only need to compute rather simple limits, for which rather simple rules may be used. For the most part, we are going to consider quotients of products and sums of polynomials, exponential functions like $x \mapsto 2^x$ and logarithms. I will not attempt to give those rules here, but point to e.g. `https://en.wikipedia.org/wiki/List_of_limits` for some formulas you can use, along with the standard way you can simplify polynomials/exponentials/logarithms.

Try to compute the following limits:

- $\lim_{n\to+\infty} \sqrt{n}$
- $\lim_{n\to+\infty} 2^{-n}$
- $\lim_{n\to+\infty} \frac{n^2}{1+n}$
- $\lim_{n\to+\infty} \lceil 2^{-n} \rceil + \lfloor 2^{-n} \rfloor$ ($\lfloor x \rfloor$ is the biggest integer smaller than x while $\lceil x \rceil$ is the smallest integer bigger than x)
- $\lim_{n\to+\infty} \frac{2^{-n}}{2^{-2n}}$
- $\lim_{n\to+\infty} \frac{(n+5)n(n+3)}{n^4+\frac{1}{n}}$
- $\lim_{n\to+\infty} \frac{n}{\log(n)+n}$

Challenge:

- $\lim_{n\to+\infty} u_n$ for $u_0 = 1$ and $u_{n+1} = (-2)^{2n} + 3u_n$
- $\lim_{n\to+\infty} \frac{x^{\sqrt{x}}}{\sqrt{x}^x}$

4. **Finding a maximum and a minimum simultaneously** For this exercise, please use the template code provided in the file `minAndMax.java` on canvas (or in the appendix of the sheet).

   (a) Write a naive java function that takes as input an array of numbers and returns a pair of numbers where the first component is the minimum element of the array and where the second component is the maximum.

   (b) How many time will you invoke the comparison operator if you input an example of size n? (don't hesitate to run the provided code to make conjectures)

   (c) Now, consider an (alternative most probably) algorithm that works as follows: first group elements of the array in pairs. Declare auxiliary arrays Top and Bot of size $\frac{n}{2}$ Compare all elements pairwise; for each pair, put the maximal element in Top and the minimal one in Bot. Then, compute naively the minimal element m in Bot and the maximal element m' in Top and return (m, m').

      i. Write this procedure in java; you will need to code the subprocedures picking a maxmium and a minimum as auxiliary functions.

      ii. How many times will you invoke the comparison operator if you input an example of size $2n$? Is it better than your previous algorithm.

      iii. Note that the above procedure works when n is even. Adapt your algorithm so that it works when n may be odd. How many comparisons do you need then for inputs of size $2n + 1$?

   Challenge (hard!): prove that the algorithm that was given as a final solution is optimal in number of comparisons (i.e., that any algorithm doing strictly less comparisons necessarily gives wrong outputs).

   (Comment: I am not aware of practical applications for this problem, but maybe there is; as we shall see later, when it comes to writing java, this is a bit of a frivolous question. But I hope it's somewhat fun at least!)

5. **Challenge task for higher-order functions likers** Implement the dichotomy search seen in the monday lecture in Java, i.e. the algorithm that takes as input a function $f : [0,1] \to [0,1]$ with $f(0) = 0$ and $f(1) = 1$ as well as some ε, and returns a x such that $|f(x) - \frac{1}{2}| < \varepsilon$. For the type of the input function f, you can use the `DoubleUnaryOperator` interface from `java.util.function`.

Appendix

```
// Class of comparable elements; the idea is that you should avoid manipulating
// elements of this class directly as integers, but only use the provided
// compare method; that way the total number of comparisons ran during the
// program will be tallied up in nComparisons, and we will be able to use that
// to compare the efficiency of various algorithms.
//
// (we will have a similar methodology to compare sorting algorithms later on)
class Comparable
{
  static private int nComparisons = 0; // the total number of comparisons made
                                       // in the program
  final private int val;

  Comparable(int x)
  {
    val = x;
  }

  static boolean compare(Comparable x, Comparable y)
  {
    nComparisons++;
    return x.val <= y.val;
  }

  // Please do not use this function to bypass compare!
  public String toString()
  {
    return Integer.toString(val);
  }

  static int nComparisons()
  {
    return nComparisons;
  }
}

// A (family of) classes for objects that can hold pairs of elements,
// parameterized by two classes T and U which are the type of the first and
// second component respectively.
//
// (if you want to look up the java concept for this notion of class
// parameterized by other classes, these are called generics)
class Pair<T,U>
{
  final public T fst;
  final public U snd;

  Pair(T x, U y)
```

```
  {
    fst = x;
    snd = y;
  }
}

public class MinAndMax
{
   // Input: an array in of Comparables of length >= 1
   // Output: a pair of Comparables
   static Pair<Comparable,Comparable> minAndMax(Comparable[] in)
   {
     // Put here your code; you can assume that the input array in
     // has length >= 1
     return null;
   }

   public static void main()
   {
     int[] exVals = {4,86,42,6,4,7,8,9};
     Comparable[] ex = new Comparable[exVals.length];

     for(int i = 0; i < exVals.length; ++i)
       ex[i] = new Comparable(exVals[i]);

     Pair<Comparable,Comparable> res = minAndMax(ex);

     System.out.printf("The minimum of %s is %s and the maximum %s.\n",
                       ex.toString(), res.fst.toString(), res.snd.toString());
     System.out.printf("This was computed using %d comparisons.\n",
                       Comparable.nComparisons());
   }
}
```