CSCM12 – Software concepts and efficiency February 2nd 2024
Manjiri Joshi, Cécilia Pradic

Lab 1

The goal of this lab is to reacclimate yourself with basic Java and try to impress upon you that some pieces of code may be faster than others. Try to complete Questions 1 and 2. If you do not have for Question 3, please give it some thought before the next lecture.

The **Challenge** tasks go beyond our expectations of you. Kudos if you can do them, but it is fine if you don't engage with those.

In one question you are asked to produce some graphs. There is a script that we provide you on canvas that can help you with this, we would recommend you do this if you are not familiar with excel, but that script requires that `matplotlib` (a python library) is installed on the machine you are using. **It is not installed by default on the lab machine, so you need to install it as follows if you are using a lab machine:** go to ZenWorks and install a version of Jupyter notebook (any should do). Then open a terminal[1], type `pip install matplotlib` and hit enter. It will take a few minutes, but you can start doing the lab sheet/other things while it's installing.

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.

2. **Some experiments** Open the file `Lab1.java` provided to you on Canvas.

   (a) Write a function that counts the number of pairs (i, j) such that `a[i] == a[j]` and $i < j < n$, where n is the argument. You should complete the following definition in `Lab1.java`:

   ```
   static int countDup1(int n)
   ```

   Note that `a` is a large array of integers which is defined globally in the class defined in the file . You can always assume that n is smaller or equal to the size of the array. Please do not look at the code for the functions `countDup2`, `countDup3` and `countDup4` yet :)

   (b) `Lab1.java` contains three other implementations `countDup2`, countDup3 and `countDup4` that you may now look at. They should do the same thing, but with varying efficiency. What the main function does is that measures the running time of those implementations on various inputs and output some data in a file called `Lab1.csv`. Take a quick read of the whole program, compile it, run it and try to understand how it roughly works (note that it tries to create/modify a file `Lab1.csv`). Modify the program[2] so that it also mesures outputs a spreadsheet containing time measurements for all of the four `countDup` functions.

[1] You can do this by going to windows start menu and searching for "command line" or "terminal". Otherwise you can use the shortcut windows key + R on your keyboard, type `cmd` in the textbox and run that.

[2] You might need to use the syntax for lambdas as in the example. If you want to understand the details you may look at `https://dev.java/learn/lambdas/`, but you do not need to understand the nitty-gritty to complete the lab sheet.

(c) Plot the different time complexity of the four functions. You can do this by downloading the file `Lab1.py` provided to you on canvas, putting it in the same folder as your data file `Lab1.csv` and double-clicking on it. If you do this, please make sure beforehand that `matplotlib` is installed on the machine you are using (c.f. start of the lab sheet). Alternatively, you could open `Lab1.csv` in a spreadsheet editor like excel and generate the plots by hand, up to you!

(d) Looking at the plot, which implementation do you find the more efficient?

(e) Try to match the curves you obtain with the following three functions of n:

$$n\log(n) \qquad n^2 \qquad n^3$$

If using the python script, to do that, you can uncomment lines 31 – 34 (by removing the # character and the space afterwards) and re-run the script; it will plot the graph of those functions (approximately) as well as your previous data.

3. Write a function

```
static int exp(int n)
```

that computes $2^n = \underbrace{2 \times \ldots \times 2}_{n \text{ times}}$ (you can assume n is a positive natural number). Then, compare it for efficiency with the following alternative implementation:

```
static int expOtherImplementation(int n)
{
  if(n == 0)
    return 1;
  int r = expOtherImplementation(n / 2);
  return (n % 2 == 0) ? r * r : r * r * 2;
}
```

How does it work? Is it more efficient than your implementation or less? Try to justify your answer if you can!

Challenge task: Finding a maximum and a minimum simultaneously

For this exercise, please use the template code provided in the file `minAndMax.java` on canvas. It contains some helpful functions that will allow you to test your functions (feel free to spend some time with it to understand how it works).

(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 $2n + 1$?

Super-Challenge: 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).