CSCM12 – Software concepts and efficiency February 2nd 2024
Arno Pauly & 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. It would be nice if you engage with the first two questions. The last one is a bit more recreational and there to get you in the mood of designing your own efficient algorithms (as we are going to see later, the sort of optimizations this last question asks you to consider would be barely noticeable on most practical examples, so it is not a huge issue if you do not complete it (but it's fun!)).

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

   (a) In `Lab1.java`, there is a prototype to be completed:

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

   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.

   (b) The file contains three other implementations `countDup2`, countDup3 and `countDup4` that should do the same thing, but with varying efficiency. Take a read of the whole program, compile it, run it and try to assess what it does. Modify the program[1] so that it outputs a spreadsheet containing time measurements for all of the four `countDupx` functions.

   (c) Using a spreadsheet editor, plot the different time complexity of the four functions. Which is the one which looks more efficient for large inputs?

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

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

   (e) Write a function

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

   that computes 2^n. Then, compare it for efficiency with the following alternative implementation:

   ```
   static int expOtherImplementatin(int n)
   {
     if(n == 0)
       return 1;
   ```

[1] 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://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`, but you do not necessarily need to understand the nitty-gritty.

```
    int r = exp3(n / 2);
    return (n % 2 == 0) ? r * r : r * r * 2;
  }
```

Which of these function is the more efficient? Modify the code of `Lab1.java` to compare these functions and give your answer.

3. **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 $2n+1$?

   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!)