Luleå University of Technology
Computer Science
Håkan Jonsson

January 20, 2017
D0010E

# Lab 1
## – Two Dilemmas –

## 1 How to pass this assignment

This assignment consists of 10 tasks. The purpose is to introduce the basic programming language constructs used in Java and to learn about some issues involving computations. When you work on some of the tasks you will create, and incrementally add code to, two Java classes named `LifeLength` and `Raise`. Other tasks asks for handwritten answers on paper. During your work with the tasks there will be lab assistants around in the lab to help you, if you get stuck.

When you have completed all 10 tasks, you should have your work checked. First, print and fill in the grading document available from the web. Then find a lab assistant and present your work. At such presentation you should:

1. Show (on screen), demonstrate (run tests), explain, and discuss your class `LifeLength`.
2. Show, explain, and discuss your handwritten answer to Problem 5 and 7.
3. Show (on screen), demonstrate (run tests), explain, and discuss your class `Raise`.
4. Show, explain, and discuss your handwritten answer to Problem 10.

You pass if everything is correct and you manage to explain your work in detail so it is clear that you understand what you have done. If something is incorrect or you can not explain everything, you fail and get a couple of days to correct your solutions and present it all again.

## 2 Part 1: Collatz conjecture

### 2.1 Small steps

In this lab assignment we will study sequences of natural numbers. The sequences are generated by starting with an arbitrary natural number $a_0$, and then producing $a_1 = f(a_0)$, $a_2 = f(a_1)$, $a_3 = f(a_2)$, etc where

$$f(n) = \begin{cases} 1, & \text{if } n = 1, \\ n/2, & \text{if } n \text{ is even, and} \\ 3n+1 & \text{otherwise.} \end{cases} \tag{1}$$

For instance, $a_0 = 3$ gives the sequence (you should check this)

3, 10, 5, 16, 8, 4, 2, 1, 1, 1, ...

while $a_0 = 42$ gives

42, 21, 64, 32, 16, 8, 4, 2, 1, 1, 1, ...

We say that the function $f$ "takes one step" in the sequence.

**Task 1.** Write a class `LifeLength` with a static method

```
public static int f1(int a0)
```

that implements $f$ (as defined in Eq. 1) and a static method `main` of the kind that always is included in Java programs. Printing is only allowed in `main`. It is forbidden in `f1`.

The program (`main`), should take an integer $a_0$ as argument, call `f1` to compute $f(a_0)$, and print the result ($f(a_0)$). To access an argument given to a Java program, and to convert a string to an integer, follow the example below.

The `main`-method in Java always takes an array of strings as an argument. Each string contains an argument given to the program. You will learn about arrays later in the course, but for now you can write the following in order to convert the `String`s to `int`s:

```
public static void main(String[] args){
  int first = Integer.parseInt(args[0]);
  int second = Integer.parseInt(args[1]);
}
```

Use '2' for the third argument, and so on. Remember that forgetting the arguments when you run the above program will lead to an error, and the program will crash. The expression `args.length` gives the number of arguments as an integer, and can be used in an `if`-statement if you like to test how many arguments are present (and maybe add your own error-checking). Using `LifeLength` we can now generate sequences one number at a time[1]:

```
> java LifeLength 42
21
> java LifeLength 21
64
> java LifeLength 64
32
>
```

and so on.

## 2.2 Taking larger steps

In the sequences above, the number 1 is eventually reached after which only ones are generated. In the sequence that started with $a_0 = 3$ this happened after 7 steps. In the second sequence it took 8 steps.

Let the *life length* $\ell(x)$ of a number $x$ be the number of steps it takes before we reach 1 in the sequence that starts with $x$. So, $\ell(3) = 7$ while $\ell(42) = 8$. We will now study the life length of some other numbers. However, to not exhaust ourselves, we define some practical functions that can take more than (just) one step at a time in a sequence.

**Task 2.** Add, *without using loops* and *without extensively repeating code*, the static methods `f2`, `f4`, `f8`, `f16`, and `f32` that takes 2, 4, 8, 16, and 32 steps respectively in sequence to the class `LifeLength` such that

- `f2(`$a_0$`)` returns $a_2$,
- `f4(`$a_0$`)` returns $a_4$,
- `f8(`$a_0$`)` returns $a_8$,
- `f16(`$a_0$`)` returns $a_{16}$, and
- `f32(`$a_0$`)` returns $a_{32}$.

Extend the output printed in `main` to include also `f2(`$a_0$`)`, `f4(`$a_0$`)`, `f8(`$a_0$`)`, `f16(`$a_0$`)`, and `f32(`$a_0$`)`. Carefully format the output so that it is clear what the printed numbers are:

```
> java LifeLength 42
f1=21  f2=64  f4=16  f8=1  f16=1  f32=1
>
```

Hint: Define each function in terms of another function that takes slightly fewer steps.

[1] This example assumes that Java is run the old-fashioned way at a command prompt in a UNIX-system, and that arguments to the program are given as options on the command line. If you program in `Eclipse`, which we recommend in this course, there is a dialog box (the tab "Arguments" under item "Run..." in the menu "Run") in which you can specify the arguments to your program.

We are now better equipped to determine life lengths by testing. Moreover, if we like to take larger steps in a sequence we could define new functions `f64`, `f128`, and so on.

Still, it will be a bit cumbersome to take a number of steps that is not a power of two. In fact, we would be better of with a function `iterateF` that would take not only $a_0$ as argument but also the number of steps that should be taken. All functions that takes a specific number of steps can then be substituted for this *generalized* function, which can be implemented with a `for`-loop that iterates from 0 to $n$.

**Task 3.** Add a static method

```
public static int iterateF(int a0, int n)
```

to `LifeLength` that, given a start value $a_0$ and a number $n \geq 0$, takes $n$ steps in the sequence that starts with $a_0$.

Add code in `main` that make the following calls and prints the results: `interateF(3,5)`, `interateF(42,3)` and finally `interateF(1,3)`.

Practical tip: Make comments out of all already existing lines of code in `main` when you add the new code. Then you can run your new code without running the old code.

Turing code into comments, as suggested above, inactivates the code but keeps the text of the code in the class so that it can easily be activated into code again later on, if desired. This is a common, practical, and very simple technique used by programmers to dynamically handle their code while writing programs.

## 2.3 Life length

Looking back, what we are interested in is life lengths, not taking steps in sequences. Probably, especially while working on Task 3, you might have been thinking of simply defining a function that automatically computes the life length of a number. We will now do this in two different ways.

### 2.3.1 Computation Pattern 1: Iteration

Iteration is a powerful technique to repeat a computation. We used iteration to solve Task 3 and we will here see another situation where it can be used. This example shows how $x^k$, for integers $k > 0$, can be computed using iteration.

```
public static double iterRaise(double x, int k) {
    int count = 0;
    double result = 1.0;

    while (count < k) {
        result = result * x;
        count = count + 1;
    }
    return result;
}
```

Note how the iteration in the `while`-loop continues as long as the condition (`count < k`) is true. In each iteration the variable `result` is changed and `count` is incremented. Eventually, `count` becomes greater than `k` and we leave the loop. At that moment, `result` contains $x^k$ as desired.

To compute a life length using this pattern we instead repeatedly apply $f$ to generate one number after another in the sequence, and do so until 1 is obtained. The corresponding condition in the `while`-loop checks that the number we have reached in the sequence is not 1. The number of times $f$ is applied is then the life length, so we need a variable to keep track of the number of iterations and be careful to update it within the loop.

**Task 4.** Add yet a static method

```
public static int iterLifeLength(int a0)
```

to the class `LifeLength` that computes the life length of a given number using a loop. Also, add a static method `intsToString` that, given a value `X` and the life length `Y` of `X` as arguments, returns the string

```
"The life length of " + X + " is " + Y + "."
```

Change `main`, by turning code into comments again, so it computes the life lengths of the numbers from 1 to 15 using calls to `iterLifeLength` and only printing what `intsToString` returns.

### 2.3.2 Computation Pattern 2: Recursion

Another method to compute life lengths is based on recursion over natural numbers. In general, such a function calls itself and makes use of the result of that call to compute the final result. For example, the function that computes $x^k$ for $k > 0$ could work as like this: For large $k$, it first make a recursive call to raise $x$ to the power of $k - 1$, and then multiplies the result by $x$. This works since

$$x^k = xx^{k-1}, k > 0.$$

On the other hand, for k=0 the result is always 1, and no recursion is needed. This gives us the complete equation:

$$x^k = \begin{cases} 1, & k = 0 \\ xx^{k-1}, & \text{otherwise.} \end{cases} \tag{2}$$

If you find it hard to locate the function itself in Eq. 2, this is because it is implicit in the way we write powers in mathematics. It lacks a notation of its own like a symbol or name. If we introduce the convention that $\text{power}(x, k) = x^k$, Eq. 2 turns into

$$\text{power}(x, k) = \begin{cases} 1, & k = 0 \\ x \cdot \text{power}(x, k - 1), & \text{otherwise.} \end{cases} \tag{3}$$

Based on this, we can now implement the function in Java as follows:

```
public static double recRaiseOne(double x, int k) {
    if (k==0) {
        return 1.0;
    } else {
        return x * recRaiseOne (x, k-1);
    }
}
```

Eq. 2 is the key to this recursive solution as it relates $x^k$ and $x^{k-1}$ to each other. To come up with a recursive function that computes life lengths we need to formulate a similar equation that relates life lengths to each other.

**Task 5.** Give a recursive equation that relates $\ell(x)$ to $\ell(f(x))$ in the case where

1. $x = 1$ (this is easy)
2. $x \neq 1$ (here, $f$ is the function defined on page 1).

Hint: Write down a list of the numbers $a_0, a_1, a_2, \ldots$ of a generated sequence. Also, write the life length beside each number. Can you see the pattern?

Note: You must *write down* your answer to this problem as a valid mathematical equation as in the example above. It is not enough to just be able to phrase it verbally.

The recursive equation tells you how the computation should be carried out. What you also need is a name of the function that can serve as a good name of the method in Java. Two candidates are $\ell$ and `l` (lower-case `L`) that are traditionally used in mathematics but who are both unsuitable when programming computers.

The first is not a symbol found on ordinary keyboards so it can not be entered by neither you nor programmers who like to make use of your method. The other is sometimes, in some fonts, very similar to the number 1 so one can easily be mistaken for the other. More importantly, the name `l` does not say what the method does and computes.

**Task 6.** Add to your class `LifeLength` a static method `recLifeLength` that computes the life length of a number using your answer to Task 5.

Change the program so that it outputs the life lengths computed by both `iterLifeLength` and `recLifeLength` side-by-side, and find once again the life lengths of all numbers $a_0 \in \{1, \ldots, 15\}$.

We have not taken much time to freshen up on recursive definitions so maybe they feel hard to understand. The lecture slides from Lecture 2 contains a part that is devoted to this topic and that you should read them if you need more on recursion.

A practical way to learn about recursive computations is to carry them out by hand, with pen and paper, and with concrete input. The computation of $2^4$, for instance, starts with

$$\begin{aligned}2^4 &= 2 \cdot 2^3\\ &= 2(2 \cdot 2^2)\\ &= 2(2(2 \cdot 2^1))\\ &= 2(2(2(2 \cdot 2^0)))\end{aligned}$$

Now, since $2^0 = 1$, we continue with

$$\begin{aligned}&= 2(2(2(2 \cdot 1)))\\ &= 2(2(2 \cdot 2))\\ &= 2(2 \cdot 4)\\ &= 2 \cdot 8\\ &= 16.\end{aligned}$$

The important thing is not to have arrived at the correct answer (we already knew by heart that $2^4 = 16$). Rather, it is the mechanical procedure performed to get the answer that is interesting. All recursive computations can, in principle, be traced like this.

**Clean-Up Task: Part 1.** Do the following to finalize your class `LifeLength`.

1. Activate all commented lines of code in `main` and divide them up in chunks according to which of the tasks 1, 2, 3, 4, and 6 they belong to.
2. For each chunk: Write a static method and move the chunk into that method, so that the lines of code in the chunk get executed when the method is called. Name methods so it is clear to which tasks they belong. Note that some methods, like that one for Task 1, might need parameters.
3. Make sure `main` is now empty. Then add an integer variable $n$ and a `switch`-statement in which the branching depends on the value of $n$.
4. Program the `switch`-statement so that if $n \in \{1, 2, 3, 4, 6\}$, the method for Task $n$ is called (and no other method). If $n \notin \{1, 2, 3, 4, 6\}$, the program should just print an error message pointing out the fact that $n$ has an invalid value.
5. Make sure you can now run your program and demonstrate that it works for different values of $n$, that is that you have workable solutions to tasks 1, 2 , 3, 4, and 6. You will be asked to do this at the grading.

## 2.4 A Dilemma

The sequences that we are dealing with have for several decades been the subject of intensive study by mathematicians, who above all have focused in on the following problem:

Does every start value have a finite life length?

The strange thing is that despite all the effort spent on this problem no one knows the answer. At least one "proof" has been presented that claimed that the answer is *yes*, but the proof turned out to be wrong. By testing different start values, like you have done in this lab assignment, it is known that the answer is indeed *yes* for all start values up to around $10^{60}$. However, as you know, this is no proof for general start values. There is actually an opportunity here to become famous!

**Task 7.** It could still be the case that there exists some large start value for which the answer is *no*. Explain in a few sentences, and in your own words, what it would mean for your program if one would run it on such an input.

Note: You must *write down* your answer.

## 3 Part 2: Computational efficiency

In this second part of the assignment we will study program efficiency. It is usually possible to find more then one method to solve a given computational problem. In Section 2.3, for instance, we learned about two ways to compute $x^k$. In this Section, we will learn about a third way that is based on the following equation:

$$x^k = \begin{cases} 1, & \text{if } k = 0, \\ x^{\lfloor \frac{k}{2} \rfloor} x^{\lfloor \frac{k}{2} \rfloor}, & \text{if } k \text{ is even, and} \\ x \cdot x^{\lfloor \frac{k}{2} \rfloor} x^{\lfloor \frac{k}{2} \rfloor}, & \text{otherwise.} \end{cases} \tag{4}$$

Here $\lfloor x \rfloor$ denotes "floor(x)". Eq. 4 is recursive in that it contains two recursive cases in which $x^k$ is defined in terms of $x^{\lfloor \frac{k}{2} \rfloor}$.

**Task 8.** Write a class `Raise` with a static method

```
public static double recRaiseHalf(double x, int k)
```

that computes $x^k$ using Eq. 4. Include a method `main` from which you test your method. The method `recRaiseHalf` will be used in Task 9 below.

Important: Make sure that you, when needed, assign $x^{\lfloor \frac{k}{2} \rfloor}$ to a local variable and then use this variable in the multiplications that gives the final result.

We now have two different methods, `recRaiseOne` and `recRaiseHalf`, that both employ recursion to compute $x^k$. Their correctness follows directly by the ordinary power-laws but how do the two compare as far as running time is concerned? What we like is to know roughly how long these methods will take to finish as a function of the size of their input (that is, $x$ and $k$).

It is cumbersome to perform a formal study and prove bounds on the precise amount of machine instructions the methods will spend. Also, the actual time depends on what computer you use and how it happens to be programmed internally. There are entire courses devoted to this at our department. Instead we will carry out a series of experiments to establish the relation between the number of recursive calls and the size of the input. Since the computation within each method only involves constant time operations like comparisons, small expressions, assignments, and calls to methods, the number of recursive calls will at least tell us about how the running time grows with larger input.

**Task 9.** Add to the class `Raise`, and above `recRaiseHalf`, the method `recRaiseOne` on page 4. Re-write the two methods, and add code, so that it is possible to count and report the number of recursive calls made separately for the two methods.

Add a test where you compute $x^k$ with the two methods, and print their results as well as the number of recursive calls each of them make, for $x = 1.5$ and $k \in \{1, 2, \ldots, 15\}$.

We are now ready to perform an experimental study of the efficiency of the methods `recRaiseOne` and `recRaiseHalf`.

**Problem 10.**

Note: You must *write down* your answers below, either by hand or using computer.

1. What do you think influences the running time the most, the size of $x$ or the size of $k$? Why? Explain using your own words and in at most a few sentences.
2. Choose a small value $1 < x < 1.001$ and compute the number of recursive calls made by the two methods when $k$ vary from 1 to 10 000. Make suitable changes to `Raise` so that all computations are done in a single run.

   Let $N_{\text{one}}(k)$ be the number of recursive calls made by `recRaiseOne`, and let $N_{\text{half}}(k)$ be the number of recursive calls made by `recRaiseHalf`. Draw graphs that show $N_{\text{one}}(k)$ and $N_{\text{half}}(k)$. It is alright to make a drawing by hand but you may also use a graph-drawing program, if you prefer. Make two plots, one per method. Be careful to mark values $k$ where something interesting happens in the graphs, if there are such $k$.
3. Formulate a hypothesis (an educated guess) about what kind of function $N_{\text{one}}(k)$ is. Then, run more experiments on larger values $k$ to see if your hypothesis seem to be correct. Continue until you manage to formulate a function that is correct for $1 < k \leq 75\ 000$.
4. Do the same for $N_{\text{half}}(k)$, that is formulate a hypothesis, run more experiments etc.

   This function is harder to formulate. In addition to the ordinary arithmetic operators you probably also need functions like ceiling, floor and base-2 logarithms ($\log_2$).

   Hint: Check out the sequence of numbers $2^i$ for $i = 0, 1, 2, 3 \ldots$.

## 4 Another Dilemma

The two solutions you have studied have dramatically different running times. The reason for this lies in the very way they are formulated; the algorithm used. A closely related question is what a computer program really computes? When we program a computer we have a specific general solution to a specific problem in mind. The solution is an *algorithm* and the program is an *implementation* of the algorithm.

It is relatively easy to prove small and simple algorithms correct but the correctness of larger ones can be very difficult to prove. In fact, most computer programs are so large and complicated that they defy all attempts to prove them correct, why it is not possible to formally assure that they really compute what they are supposed to compute.

The practical solution to this dilemma is to perform extensive tests to see if the program works. However, in real contexts involving commercial software, such tests can never cover all possible situations and inputs. A test only indicates the correctness in *specific* cases, not the absence of errors in *all* cases.

Also, programs are subject to numerical problems. Our program that computes $x^k$ by iteration is correct in the sense that it performs the right operations. However, we use variables of type `double` that have bounded accuracy so what happens when we repeatedly perform thousands of multiplications of intermediate values stored i doubles? Is the final result stored in such a double really $x^k$..?