# Compulsory assignment 1 - INF 102 - Autumn 2017

Deadline: **Septemer 29th 2017, 16:00**

## Organizational notes

This compulsory assignment is an individual task, however you are allowed to work together with at most 1 other student. If you do, remember to write down both your name and the name of team member on everything you submit (code + answer text). In addition to your own code, you may use the entire Java standard library, the booksite and the code provided in the github repository associated with this course.

The assignments are pass or fail. If you have made a serious attempt but the result is still not sufficient, you get feedback and a new (short, final) deadline. You need to pass all (3) assignments to be admitted to the final exam.

Your solutions (including all source code and textual solutions in PDF format) must be submitted to the automatic submission system accessible through the course before Sep 29th 2017, 16:00. Instructions on how to submit will follow shortly. Independent of using the submission system, you should always keep a backup of your solutions for safety and later reference.

If you have any questions related to the exercises, send an email to:
`knutandersstokke@gmail.com`
In addition you have the opportunity to ask some questions in the Tuesday review session and at the workshops.

## 1 From Reverse Polish to Infix

This expression is written in *reverse polish* notation:

$$1\ 3\ +\ 2\ 4\ 2\ *\ +\ * \tag{1}$$

It's a postfix notation, and is computer friendly when it comes to calculating the answer. However, this notation is hard for the human eye to read. Implement a program which takes a reverse polish expression and converts it to an expression with infix notation. In this case, the output would be:

$$(1 + 3) * (2 + (4 * 2)) \tag{2}$$

## 2 Timeline

A popular feature of every well known web browser is the ability to navigate back and fourth through the last websites visited. This timeline is often a single track, so if you go back a couple of websites and decides to go to a new website from there, you erase "the future".

Implement a program which takes a number N, and then N lines. The lines contain either a website title ("reddit: the front page of the internet", "cats - YouTube", etc.) or a command `*back*` or `*forward*` to indicate that the user hits backward or forward respectively. For each line, print out the current title of the browser. If the user wants to go past the first or last website in the timeline, print "You are currently on the first/last website".

This is an example of an input and its corresponding output:

**Input**:

```
11
Facebook
Twitter
Google
*back*
*back*
*forward*
YouTube
*forward*
LinkedIn
*back*
*back*
```

**Output**:

```
Facebook
Twitter
Google
Twitter
Facebook
Twitter
YouTube
You are currently on the last website
LinkedIn
YouTube
Twitter
```

## 3 Triplicates in four lists

Given four lists of N names, devise a linearithmic (O(N*log(N))) algorithm to determine if there is any name which exist in exact three of the four lists, and if so, return the lexicographically first such name. Implement your algorithm in Java and do a (theoretical) runtime analysis that explains how you arrive at the linearithmic order of growth.

## 4 When is sorting profitable?

An easy way to find an element in an array is to just loop through the array until you hit the element or conclude that the key does not occur in the array. This method is fast for a few searches, but when the number of searches increase you might want to sort the table first, and then use binary search to find the element you're looking for.

In this exercise we want to know for how many searches sorting first would be profitable. Use an array of size $N = 10e6$ with integers between 0 and 999, and compare linear searches in the unsorted array against binary searches (you can use the binary search algorithm provided in the algs4 library) of the sorted array. Include the time it takes to sort the array when measuring binary search. Test for S = 1, 2, ..., 100 searches. Do one experiment where some of the numbers are outside the range (average runtime) and one experiment where every number is outside the range (worst case runtime).

Find the average search time for every S and answer these questions:

1. How did you performed the experiment? If you performed the experiment by using other values or techniques than in the description, explain what you did different and why.
2. Which S would give sorting first the fastest average search time?
3. Is there any difference between average and worst case runtime? Why?
4. If you run the experiment multiple times, does the result differ? If so, how could you get a more accurate result?

Provide the code for your experiment as well.

Also, the Java optimizer tends to skip certain tasks when the values of a computation are not used for anything. To turn the optimizer off use the argument `-Djava.compiler=NONE` when running.
(In Eclipse, go to *Run* → *Run Configurations...* → *Arguments* and type in the argument under *VM arguments*).