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 your team member in all the files you submit. 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 **September 29th 2017, 16:00**.
For each assignment, you will receive an invitation to a git-archive through the submission system (`https://retting.ii.uib.no`) which you will use to submit your solutions. You can use the same user credentials as for your mitt.uib.no account.

Your **textual** solutions (everything except source code) must be submitted in a single pdf-file named `hand-in.pdf` (case-sensitive) and placed in the root folder of your git-archive. All **source code** must be placed inside the `src` folder that is distributed with the archive. To submit your solutions, push to the remote archive (remember to `-add` and `-commit` your files first).

Independent of using the submission system, you should always keep a backup of your solutions for safety and later reference. After submission, your code will be compiled by the system and compilation errors will be reported. You can view the results of these tests in your browser under the following url: `http://retting.ii.uib.no:81/me`. Only code that can be compiled on this system will be accepted.

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

The following is an example of an expression 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 to read for the human eye. Implement a java-program `rpTranslator` which takes an arbitrary reverse polish expression and converts it to a fully bracketed expression in infix notation. In this case, the output would be:

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

You may assume that the input is a sequence of strings separated by whitespaces. Strings ”*” and ”+” represent the operators.

2 Timeline

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

Implement a java-program `siteBrowser` which takes a number N, and then N lines. The lines contain either a website title (a string) 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 website. If the user wants to go past the first or last website in the timeline, the program should print a warning and stay at the current 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
[Warning: 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 occurs in exact three of the four lists. If so, return the lexicographically first such name. Implement your algorithm in Java (`exactTriplicates`) 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 the position of an element in an array is to just loop through the array until you find the element, or conclude that the element does not occur. This method is fast for a few searches, but when the number of searches increase you might want to sort the array 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 O(N*log(N)) sorting becomes profitable. Use an array of size $N = 10e6$ of random integers and compare linear searches in the unsorted array against binary searches in the sorted array. If the search is successful, the position of the element in the array should be printed. You can use the binary search algorithm provided in the algs4 library. When evaluating binary search, take the total time of all searches plus the time it takes to sort the array. Test first for S = 5, 10, 20, 40,... random searches, until linear search becomes slower than binary search. You will find an S for which linear search is fastest, such that for 2S binary search is fastest. Then interpolate between S and 2S to find some break-even point.

Answer these questions:

1. Based on your experiments, what is the break-even S?
2. 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 (`profitSorting`).