Compulsory assignment 2 - INF 102 - Autumn 2017

Deadline: **October 20th 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 Oct 20th 2017, 16:00. Independent of using the submission system, you should always keep a backup of your solutions for safety and later reference.

The project will be a maven project (and not Eclipse project) so that students can use their IDE of choice. If you're using Eclipse, you should either clone the repository first and then import it as *existing maven project*, or import it directly using this guide.

Some of the problems are covered by tests. Be aware that they are just an indication of how the structure of your code should look like. Passed tests does not necessarily mean that you have solved the problem. Though the TAs won't take your own written tests into account, we highly recommend you to write tests as you write code.

If you have any questions related to the exercises, send an email to:
`knut.stokke@student.uib.no`
In addition you have the opportunity to ask some questions in the Tuesday review session and at the workshops.

Comparing Ternary Search and Binary Search

In this exercise we want you to compare searching through an unbalanced 2-3-tree (Ternary Search) to searching through an unbalanced binary tree (Binary Search) on the number of compares.

You may use (clickable links) `UBST.java` and `UTST.java` from the repository and modify them according to your needs. `UTST.java` only has a method `put()` for the purpose of this experiment. If you choose to use these programs for your experiment you should modify their `put()` and `get()` methods so that they also count the number of compares.

Test your programs on small files like `algs4-data/tinyTale.txt`. Then compare the number of compares (pun not intended) of both programs with sufficiently large files with random keys. You should use the same keys for both trees.

Perform the experiment and describe the results. How does the number of inserted values and the number of searches affect the difference between the number of compares?

Minimise number of disks

You have a set of files with different sizes, and you want write all the files to disks. Each disk has (decimal) 1GB of storage, and you want to minimise the number of disks needed. Finding the best solution is not efficient (if you find an efficient way, please let us know), so instead we want you to try two different approaches which often give an ok solution:

- One solution is to write the files to the disk in the same order as they are represented. Each file should be written on the disk with most space left. If no disk has enough space, you get a new empty disk and write it there.
- Another solution is to sort the files (comparing the sizes) in descending order, and then write them using the same approach as the first solution.

The input is a file with strings, representing the file sizes of the fictive files in the following (newline-separated) format: ”`File-size1\nFile-size2\nFile-size3`”
You can assume that the file sizes are in the range 1-999MB. Output the number of disks used and a list of all the disks. For each disk print all the file sizes of the ‘files’ that were written to the disk. E.g. if the input consists of 500MB, 100MB, 500MB and 600MB, then the first solution would give:

```
3
Disk 1 : [500MB, 100MB]
Disk 2 : [500MB]
Disk 3 : [600MB]
```

Use input files with random filesizes between 1B and 1GB, and with different numbers of such file sizes. Submit the code you used for the comparison and describe the results in the pdf-file.

(Hint: Is there a clever data structure that could be used to store the disks to easily find the disk with most space left? Think of `MultiWayMerge`!)

Huffman

The idea behind the Huffman encoding is to replace long words that occur often by shorter strings. Thus the size of a text file can often be reduced, at the cost of en-/decoding (space/time trade-off).

Your program should take a filename (`*.txt`) as input. Using a symbol-table, do a frequency analysis of word counts for the provided file. Sort your symbol-table on the counts.

Implement a sensible translation of long, frequent words to shorter strings in another symbol table. These strings should only contain ASCII-characters. Translate the file according to this symbol table, and output the result to a file in this format:

```
symbol-table + "\n" + "****" + "\n" + compressed-text
```

The filename of the latter file should be `*.txt.cmp`. The symbol-table should contain the neccessesary information for the decoder to decode the file.

Your implementation should also be able to reversely translate an encoded input file. The decode-method should return the decoded string.

You may assume that words are delimited by whitespaces and that the file does not contain binary strings.

Apply your program to `leipzig1M.txt` (provided in `algs4-data`) and find out whether your program performs lossless encoding with regard to words (after en- and decoding the input file, the original words are restored). If not lossless, what is the problem?

The **best solution** for this exercise will be awarded a **symbolic prize**. We will evaluate solutions based on the size of the compressed file (which includes both the symbol-table and the original input text) on disk. To qualify for a winning solution, the compression must be lossless with regard to the words occuring in the input file and the en- and decoding computation must finish within a reasonable (user-friendly) amount of time. If several solutions match these criteria and achieve equally good compression factors, we will consider additional measures such as compression speed to determine the winner.