Compulsory assignment 2 - INF 102 - Autumn 2016

Deadline: **Oct 28th, 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 28th, 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:
`gunnar.schulze@ii.uib.no`
In addition you have the opportunity to ask some questions in the Friday review session and at the workshops.

1 Sink-and-swim heapsort in maximum priority queues

Modify the sorting procedure in `Heap.java` (provided in `algs4.jar`) to avoid the check (call to compareTo) whether a key has already reached is correct position (greater than both its children in a MaxPQ). In this version, the key that is currently considered sinks to the bottom, successively trading places with the larger of its children. After that, it `swim()`s up again to reach its correct position.

Perform experiments using random arrays of 10000 string keys. Vary the length of keys that get sorted in each experiment (choose keys of 2, 10, 20, 40 and 50 characters) and compare the performance of this sorting method with the regular `Heap.sort()` method (as given in `Heap.java`) in terms of its runtime (use repetitions).
Do the same experiments for already sorted arrays.

Document your results in a table and and answer the following question: (When) does the modified version of heapsort compare favourably to the regular `Heap.sort()` method?

2 Programming contest: `Blooming` E.coli

Motivation

In the field of bioinformatics a common task is to determine which organism (e.g. a virus or bacterium) is present in a biological sample. This can be done by sequencing the DNA of the sample. In such experiments, so-called *sequencers* take short fragments of the DNA contained in a sample and output a sequence of the alphabet {A,T,C,G}, representing the DNA code contained in that fragment. These sequences can then be compared to a reference, a set of longer sequences of the same alphabet that are specific for every organism. If one or many such short sequences are found to as substrings of a reference of a particular organism, it is likely that this organism was present in the sample. In practice, for each analyzed sample, there will be tens of millions of such short sequences and thus the amount of data produced is very large. As more and more such experiments are conducted, an important challenge is to find efficient methods to compress and store the data without loosing important information.

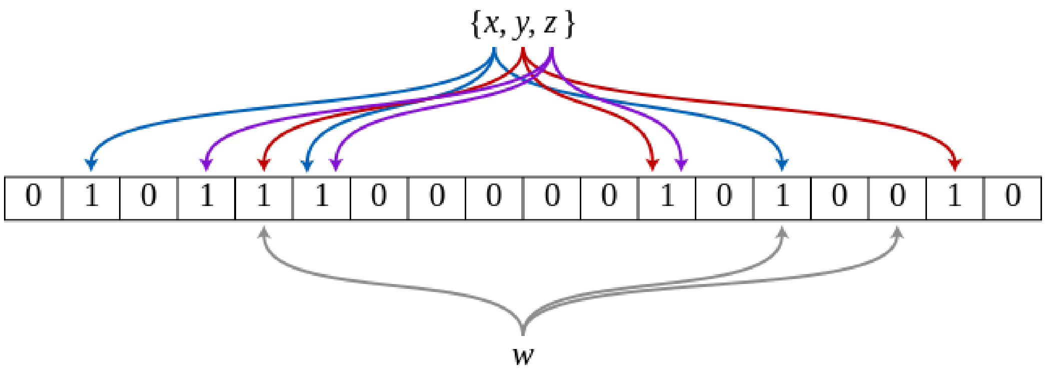

Figure 1: ref.: David Eppstein, `https://commons.wikimedia.org/w/index.php?curid=2609777`

One way to compress short sequences efficiently is through the use of so-called **Bloom filters**. Bloom filters are essentially hash-based data structures that use an array and a series of hash-functions to input or extract information (see fig.1 for an example). In a bloom filter, an array of size M is initialized with all 0 values. Given a set of input sequences, several hash functions h are evaluated separately on each sequence. Each hash function returns an index into the array and the array will be set to 1 at this index. In the example depicted in fig.1 the set of input sequences is given by {x,y,z} and each sequence is evaluated by the same 3 hash functions (colored arrows), yielding 3 indices for each sequence. When used efficiently, such an array can be much smaller (in terms of storage size) than the set of initial input sequences while still containing the information to determine whether a certain sequence was observed.

After constructing the Bloom filter, it can be queried for a given sequence (e.g. w in the example) to determine if it was present in the original set. Using the same hash functions as before the indices for a given **query** sequence are looked up in the array. If all are marked with 1 the query sequence was contained in the original set of input strings. Otherwise the conclusion is that it was not present. In the example (fig.1), w is correctly identified as "not-contained", since one of the hash functions yields an index `i` for which `array[i]==0`.

A **weak point** of Bloom filters is that the result of querying sometimes *also* returns `true` for sequences that were not contained in the original set. This is known as a *false positive* (FP). False positives can for example occur when the array is very "full" (many positions are set to 1). Thus, the challenge when constructing a Bloom filter is to keep the number of false positives as low as possible (to avoid mistakes when querying) while achieving a good compression level (ratio of N/M as high as possible).

Tasks

You are given two sets S_1 and S_2 of $N_1 = 1,000,000$ and $N_2 = 2,000,000$ short strings of length $L = 36$, respectively, which are substrings of the E.coli reference genome. S_1 is fully contained in S_2, so that $S_1 \subseteq S_2$. All sequences in S_1 and S_2 are unique (occur only once within each set). The sequences can be downloaded here: `https://drive.google.com/open?id=0ByaLRKEzgXVHZTJrUzY4eHhoRlU`

A) Write a java class `BloomFilter` that can compress the set S_1 of short strings using an array of size M (input parameter) and write the compressed array to a file.

B) Read in and query the compressed file generated in A) with the set of sequences provided in S_2.

C) Estimate the rate of false positives in your compression by comparing the number of sequences in S_2 for which the BloomFilter instance returns *true* (P) with the actual number of sequences contained in S_1. **Hint:** A Bloom filter always returns true for a true positive (TP), which in this setting comprises all sequences contained in S_1 (N_1). Thus, false positives are given by $FP = P - N_1$ and the false positive rate (FPR) is given by $FPR = \frac{FP}{FP+TP}$.

Evaluation of results

(1) Best trade-off between compression level and false-positive rate, according to the formula:

$$\frac{FS_i}{FS_c} * (1 - (S_{ratio} * FPR)) \tag{1}$$

where
FS_i is the file size of the input file (on disk), FS_c is the compressed file size (on disk),
S_{ratio} is $\frac{|S_2|}{|S_1|} = \frac{N_2}{N_1}$(=2, for the given sets),
FPR is the false positive rate (FPR) as described in C)

Constraint: The FPR should always be ≤ 0.1 (10%). In general, if FPR reaches its maximum, the term $(1 - (S_{ratio} * FPR))$ and thus the total score will become 0.

(2) Speed of compression and querying.

The solution that compares favourably (highest score) in criterion (1) and respects the additional constraint will win the prize!
If there are several very close solutions, criterion (2) will additionally be considered. Note that we will compare the programs on similar, but not necessarily the same inputs as provided in the exercise!

3 Exam exercise

Design (and solve) an interesting exercise that you think could be part of the final exam. Submit both the exercise text and the solution (possibly a small program). The task should be solvable given the lectures/book content covered in the course so far (chapters 2.5-3.5 in the textbook).