**IN1002 Coursework**

This task will require you to produce Java implementations of several search algorithms on an ordered collection of strings. The most relevant lectures are weeks 2 and 4. It is due at 5pm on Sunday 26th March 2017, to be submitted via Moodle. It is worth 30% of the module marks. (The final exam accounts for the other 70%.)

**Starting point**

Download the Search NetBeans project from Moodle. The project contains some same data files and four Java files:

interface StringList

This is an interface to a simple array-like collection of strings. You may assume that the strings are in ascending dictionary order, e.g.

"multiple" < "never" < "no" < "nobody" < "other" < "others" < "otter"

Some of the strings may be duplicated. The methods provided are

size()

returns the number of strings in the collection

get(int i)

returns the string at position i in the collection

You must **not** change anything in this file.

class FileStrings implements StringList

An implementation of StringList, initialized from one of the files in the data folder. There are also some additional methods for instrumentation that you can use in your testing.

You must **not** change anything in this class.

class Search

A class of methods, whose bodies you have to fill in. This is what you'll be submitting. You can add extra methods if you like, but there is no need for fields. You must not change the signatures of the methods supplied.

SearchTest

A main program that you can use for testing the methods you write in the Search class. Change this as much as you like. You won't be submitting it; I'll be using my own test program.

**The task**

You are to implement the following methods. The cost measure will be the number of calls to the get() method. Include an invariant (as a comment) with each loop.

public int countUnique(StringList a)

Return the number of unique elements in the list. The cost should be *n*, where *n* is the size of the list.

public String mostCommon(StringList a)

Return the element that occurs the most times in the list. If two or more are equally common, return the one that comes earliest. (We are comparing whole strings here, not substrings.) This should also cost *n*.

public int findEqual(StringList a, String k)

Return the index of an element in the list that is equal to k, or -1 if it is not in the list. using the first version of binary search discussed in week 4. This should cost log(*n*) in the worst case, but less in other cases.

public int countLess(StringList a, String k)

Return the number of elements in the list that are less than k in dictionary order. This is the index where the first entry equal to k would be found, it present, so you can do this with the second version of binary search discussed in week 4. This should cost log(*n*).

public int countLessOrEqual(StringList a, String k)

Return the number of elements in the list that are less than or equal to k in dictionary order. This can be done with a small variation of the algorithm for the previous part: look at the invariant and think about how to change it. This should cost also log(*n*).

public int countBetween(StringList a, String k1, String k2)

Return the number of elements in the list that lie between k1 and k2 inclusive. (Your method should cope still work whether or not either or both of these strings is in the list.) This should cost no more than 2\*log(*n*) (but see the extension task).

public int countMatches(StringList a, String prefix, String suffix)

Return the number of elements in the list that have the specified prefix and suffix. (They may overlap.) This should cost log(*n*) + *k*, where *k* is the number of strings in the list that have that prefix.

You may use standard Java classes. In particular, you may find the following methods of the [String](http://docs.oracle.com/javase/8/docs/api/java/lang/String.html) class useful: equals, compareTo, startsWith, endsWith.

**Assessment criteria**

The marks will be divided as follows:

Implementation

Each of the methods will be assessed for correctness and for achieving the specified cost. Be sure to consider boundary cases. Unlike the Java module, we will not be assessing Java style. The marks allocated to each method are:

* countUnique: 15%
* mostCommon: 20%
* findEqual: 10%
* countLess: 10%
* countLessOrEqual: 10%
* countBetween: 5%
* countMatches: 15%

Invariants

The correctness and completeness of the invariants will count for 10% of the mark.

Extension task

The task for the final 5% of the mark calls for additional ingenuity, and I expect only a few students will achieve it. If you don't do this, you can still get a very good mark on the rest, so I'd encourage you to get them perfect first.

In the countBetween task, the required cost was 2\*log(*n*), and this is easy to implement. The task is to do better.

**Submission**

You are to submit a single file called Search.java as described above.

All submissions will be compiled and run by an automated process, so your class **must** build with the classes I have supplied. Do not submit versions of the other files; your class must compile with my originals.