Searching and Sorting

November 11th & 12th 2019

Notes last updated: October 11th 2019, at 11.18am

1 The Code Package

There are quite a few files in this week's code package. Don't despair! Most of these are auxilliary helper files that are not essential to the exercises, such as tester and timer classes, or they define auxilliary data types, the implementations of which are not relevant (e.g. the `ArrayGenerator` and `Scope` classes). The main relevant packages are

- `arraySorter` for the questions in section 3, and hence for the logbook question;
- `stringSearcher` for the questions in section 2.

For the remaining packages and classes, probably all you need to know is the behaviour of the concrete classes in the `arrayGenerator` package.

An array generator is an object that can be used to generate arrays of a specified size, containing (random selections from) a range of values. For example

```
ArrayGenerator <Character> characterGenerator =
  new CharacterArrayGenerator("abc");
```

will create an array generator that can be used to generate arrays that will contain a random sequence of the `Character`s 'a', 'b', and 'c' (and only these), while

```
ArrayGenerator<Character> integerGenerator =
  new IntegerArrayGenerator(new IntegerScope(1,8));
```

will create an array generator that can be used to generate arrays that will contain a random sequence of the values from 1 to 8 (inclusive). Arrays can be created by calling the `getArray(int size)` method. For example after

```
Character[] characterArray = characterGenerator.getArray(5);
```

`characterArray` might be the array `['b','b','a','c','a']`, and after

```
Integer[] integerArray = integerGenerator.getArray(8);
```

`integerArray` might be the array `[4,2,7,7,3,6,8,2]`.

2 Searching in Strings

Read the following problem description, and then answer the question in section 2.6.5. Note that there are further exercises, on sorting, in section 3, and that the logbook question is question 4 in that section.

2.1 Introduction

A common searching problem is to find a given substring in a larger "superstring". Sometimes these can be very large strings, for example in biochemistry when you may be searching for a certain sequence in a large DNA or protein database. It can be important to find efficient algorithms to perform such searches.

2.2 The Problem

Given: A string to search in (`superstring`) and a string to search for (`substring`)

Preconditions:

- `superstring.length` > `substring.length`
- `substring.length` > 0

Aim: To determine if the substring is present in the superstring. Routine should return the index position of first start of the substring when the substring is present, or report that the substring is not present, if this is the case.

Algorithm: Sequential search of a string for a given substring

Input: A String superstring and a String substring

Output: Index into the superstring or a `NotFound` exception

2.3 Examples

- search for "fred" in "Alfred the Great" should return 2
- search for "cap" in "The incapable captain capsized the boat" should return 6
- search for "absent" in "The cab sent a message to base" should throw a `NotFound` exception

2.4 Code

2.4.1 StringSearcher

We can define an abstract class defining the required functionality, without specifying the implementation. Note that we represent our string as arrays of `char`s. This is for efficiency reasons. Accessing the n^{th} character as `array[n]`, rather than `string.charAt(n)` is significantly faster.

```
public abstract class StringSearcher {

   // This string searcher
   private char[] string;

   public StringSearcher(char[] string) {
      this.string = string;
   }

   public StringSearcher(String string) {
      this.string = string.toCharArray();
   }



/**
   * @return this StringSearch's String object
   */


   public char[] getString() {
      return string;
   }



/**
   * Check for an occurrence of this StringSearch's string in a
   * superstring
   * @param superstring the superstring to be searched
```

```
    * @return the index of the leftmost occurrence of this StringSearch's
    * string in the superstring, if this exists
    * @throws NotFound if no such occurrence can be found
    */

    public abstract int occursIn(char[] superstring) throws NotFound;

    public int occursIn(String superstring) throws NotFound {
        return occursIn(superstring.toCharArray());
    }
}
```

2.5 Obvious Solution

The obvious solution is to start at the beginning of the superstring and perform a sequential search through it, looking for the substring.

2.5.1 First Design

```
public int occursIn(char[] superstring) throws NotFound {


/**
    * Perform a sequential search for this string in the superstring
    * for each index in the superstring
    * if this string occurs at that index
    * return the index
    *
    *
    * no occurence found, so fail
    */


    for (int index = 0; index < superstring.length - getString().length; index++) {
        if (occursAt(superstring,index)) {// does this string occur at this index?
            return index; // yes, so return the index
        }
    }
    throw new NotFound(); // reached the end of the superstring without finding this string
}
```

The only part of this code that still needs to be defined is the `occursAt` method, which reports if the string occurs at a given position in the superstring. How *do* we check if the substring occurs at some position in the superstring?

We can do this by checking the characters one by one. If we reach the end of the target string without finding a mismatch we have found an instance of the substring.

```
private boolean occursAt(char[] superstring,int index) {


/**
   * Try to match the characters in this string, one by one, to the superstring.
   * for each character in the superstring, starting at the given index
   * if this character does not match the corresponding character in this string
   * mismatch - return false
   *
   *
   * reached end of string without a mismatch - return true
   */


   for (int i = 0; i < getString().length; i++) {
      if (getString()[i] != superstring[i+index]) {
         return false;
      }
   }
   return true;
}
```

This implementation of a simple sequential search is provided with this week's code.

2.5.2 Example

The straightforward algorithm above essentially shifts the substring along by one position each time an attempted match fails. For example, if we are looking for the target string "abbab" in the source string "abbcaabbdaabacbababababbababbd" the simple algorithm would start by comparing the first characters of the source and target strings:

```
abbcdabbdaabacbababababbababbd
abbab
↑
```

These match, so we compare the next two characters:

```
abbcdabbdaabacbababababbababbd
abbab
 ↑
```

We have another match, so move along another character:

```
abbcdabbdaabacbababababbababbd
abbab
  ↑
```

Yet another match, so try to match the next character:

```
abbcdabbdaabacbababababbababbd
abbab
   ↑
```

We have a mismatch, so the substring is moved along one space, and the process starts again:

```
abbcdabbdaabacbababababbababbd
 abbab
 ↑
```

But this is is a bit silly, for two reasons. Firstly, we already know that the second letter in the superstring is a 'b' and that the substring starts with an 'a', so there is no way that there can be a match at this position. Secondly, our mismatch character in the superstring was a 'c', and there is no 'c' anywhere in the substring so there cannot be a match with the substring at any position that includes the 'c' — we may as well shift the substring past the 'c'.

2.6 The Boyer-Moore Algorithm

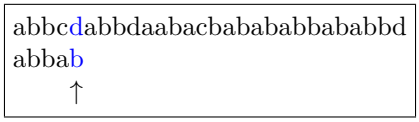
The Boyer-Moore algorithm is an algorithm that aims to avoid the problems of the straightforward algorithm by making use of the sort of analysis of possible shifts discussed above. These are known as "bad character shifts" and "good suffux shifts", and are explained below. You should note that the Boyer-Moore algorithm is based on trying to find a match starting at the *end* of the substring.

2.6.1 Bad Character Shifts

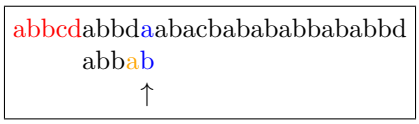
A bad character shift is based on the second reason given at the end of section 2.5.2. If you come across a mismatched character in the superstring there are two possibilities, which are discussed below.

Mismatched character does not appear in the substring: If the mismatched character in the superstring *does not* appear anywhere in the substring, as here (there is no 'd' in the substring), you shift the substring to start immediately after the mismatched character in the superstring.

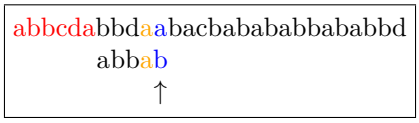
For example, using the same superstring and substring as above, the Boyer-Moore algorithm would start in this state:



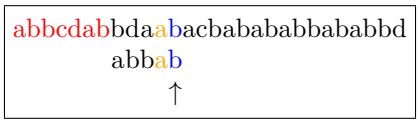
The mismatched character 'd' does not appear anywhere in the substring, so we shift the substring to start immediately to the right of the 'd', and start the matching again:



The mismatched character does appear in the substring: If the mismatched character *does* appear in the substring, as here, the *rightmost* occurrence of that character in the substring is identified (here shown in orange), and the substring is shifted to align this with the mismatched character in the superstring, and the matching starts again.

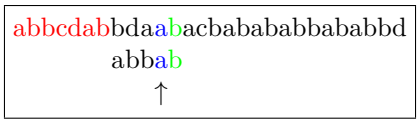


Note that we have an identical situation here, and we get another shift of one to match the two 'a's:

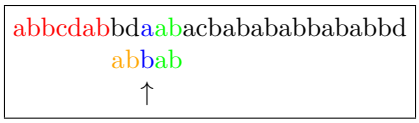


2.6.2 Good Suffix Shifts

Good suffix shifts are based on the first reason given at the end of section 2.5.2. If we continue the search from the point above we see that the first character matches. The algorithm will then try to match the next character:

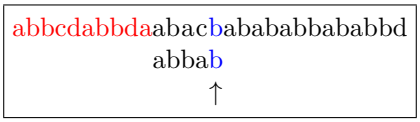


This character also matches:

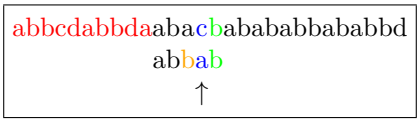


We now have a character mismatch. We could use a bad character shift but, since the *last* occurrence in the substring of the mismatched character 'a' is to the right of the current position we would then actually shift the substring back to the left. Instead we take the suffix of the substring that we have been able to match (in this case "ab"), and look for another occurrence, in the substring, of this sequence to the left of the current position. Again, there are two possibilities.

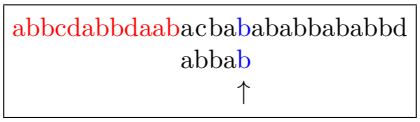
Matched suffix appears elsewhere in the substring: If the suffix that has been recognised does appear elsewhere in the substring the substring is shifted to align the rightmost (but non-final) occurrence of this substring with the sequence recognised in the superstring. In this case that is a shift of three:



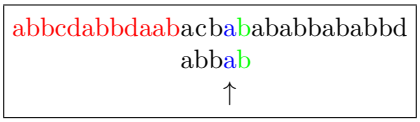
Here we have a character match:

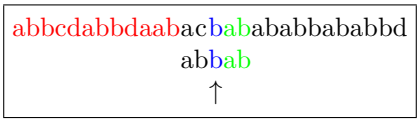


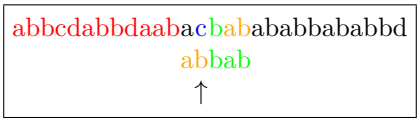
A bad character shift here would again shift the substring to the left. The rightmost non-terminal occurrence of the recognised sequence "b" is shown in orange, and a good suffix shift gives us a shift of two:



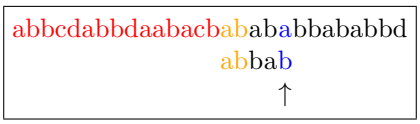
The next three steps in the matching are simple character matches:







Matched suffix does not appear anywhere in the string: We are now in a situation where we have a matched suffix "bab" that does not appear anywhere else in the substring than at the end. In this case we identify the longest prefix of the substring that will match the end of the recognised suffix (here the two "ab" sequences shown in orange), and the substring is shifted to align these sequences:

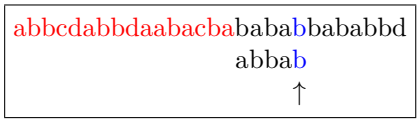


Note that we could also have done a bad character shift here, to shift the substring past the 'c', but that would only have given us a shift of two.

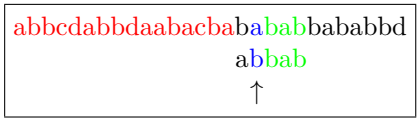
2.6.3 Completing the Search

The search would continue as follows:

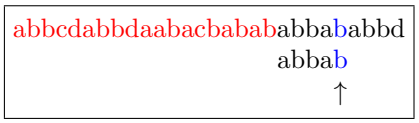
- We have a character mismatch, which gives us a bad character shift of one:



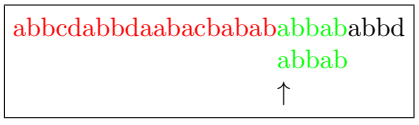
- We then get three character matches:



- A good suffix shift shifts the substring three places:



- Five character matches identify the occurrence of the substring in the superstring:



2.6.4 Boyer-Moore Shift Tables

The important thing about all the shifts described above is that they can all be calculated in advance, without any knowledge of the superstring. The Boyer-Moore algorithm will construct shift tables telling it what distance to shift for a bad character shift or for a good suffix shift. Once these tables are constructed they can be used to provide efficient searching for the substring. At each mismatch the Boyer-Moore algorithm will check both tables to find which gives the better shift, and use this shift.

2.6.5 Questions

Answer the following questions:

1. Implement the Boyer-Moore algorithm. Your implementation should **implement** the `StringSearch` **interface**.
2. Use your algorithm, and the implementation of the sequential substring search algorithm provided to perform an empirical comparison of the efficiency of the two alogorithms.
3. Express the efficiency of the two algorithms, expressed in terms of the sizes of both the string and the substring. Justify your answers.
4. Does the size of the alphabet the strings are generated from affect the efficiency of the algorithms?

You can use the `CharacterArrayGenerator`s to generate arrays of `Character`s, which are then fairly straightforward to turn into arrays of `char`s.

3 Sorting

to help you with the following questions this week's code bundle includes a `RandomIntegerArray` class that will generate an array of random `Integer`s, within a given range. You may use this, if you wish, to generate test data.

1. *Model question* Run and time the bubble sort algorithm a number of times for arrays of different sizes. Plot the timing results on a graph. Try to arrive at an (approximate) formula relating the time taken to the size of the array.
2. Implement the selection sort algorithm. Your implementation should **implement** the `ArraySort` **interface**.
3. Implement the quicksort algorithm. Your implementation should **implement** the `ArraySort` **interface**.

4. *Logbook question* Use your implementations to time the execution of (at least) these two sorting algorithms for various sizes of array, and plot the results on a graph. Can you arrive at (approximate) formulæ for how the execution times vary in relation to the data size?