COMP122 Assessment 2

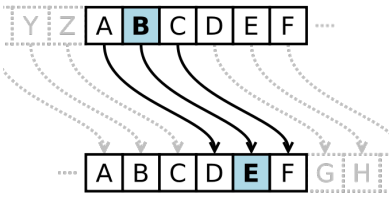
Interfaces and Exceptions

Due 2019-04-05 at 5pm

The Caesar cipher

The Caesar cipher is an ancient method of encrypting text, i.e. attempting to transform text into a format that is unreadable by others.

The Caesar cipher is a 'rotation cipher' and operates by translating each letter into the one that is shifted along the alphabet by a fixed distance. This distance is called the *shift*. It is the same for all letters in the alphabet and therefore can be seen as the secret *key* to encrypt and decrypt: To encrypt your text using a given shift, you translate letters by that many places later in the alphabet. A Caesar cipher with shift 3 can be illustrated as follows.



For example, if your text to encrypt is 'Meet me at midnight under the bridge' and your shift is 3, the encrypted text is 'Phhw ph dw plgqljkw xqghu wkh eulgjh', as the letter 'b' gets translated into an 'e', and 'e' gets translated into 'h' and so on. We 'wrap around' at the end of the alphabet, so a 'z' gets changed to a 'c' given a shift of 3. We can interpret a negative value for the shift as translating letters backwards (e.g. an 'f' gets encrypted as the letter 'b' if the shift is -4).

It is believed that Julius Caesar actually used such a cipher for his correspondence. Unfortunately for him, this type of cipher is easily broken using a frequency analysis method. This method is outlined below and your assignment is to implement it in Java.

Cracking the Caesar cipher.

Suppose we are given a *cipher text*, i.e. text that has already been encrypted with some (unknown) shift, and we want to determine the original unencrypted text (typically referred to as the *plaintext*).

To reconstruct the original text we could decode with each of the 26 possible shifts, and take the result that looks 'closest' to an English sentence.

How do we measure 'closeness'? This is where letter frequencies and a small statistical trick comes in. First of all, we know how often each letter occurs on average in English text. For instance, 'e' is the most common letter, then 't', and so on. To decode our cipher text, we can compute the frequencies of each letter as they appear in the text. To measure how 'close' these are to the known English letter frequencies, we use the χ^2-score (that is the Greek letter 'chi', so it's the 'chi-squared score'). This score is defined as

$$\chi^2 = \sum_{\alpha=a}^{z} \frac{(\text{freq}_\alpha - \text{English}_\alpha)^2}{\text{English}_\alpha}$$

where freq_α denotes the frequency of a letter α (the number of occurrences divided by the total number of letters in the text), and English_α is the corresponding English letter frequency. In other words, we sum the fraction over all 26 possible letters to determine this score.

The χ^2 score will be *lower* when the frequencies are closer to English. Note that when we do this, we are ignoring the case of letters (we want to treat upper and lower case equally for our purposes).

We will be using the following known frequencies for the letters in English, given here in a Java-style array arranged in the usual alphabetical order (see `frequencies.java`, you're welcome).

```
double[] freq = {0.0855, 0.0160, 0.0316, 0.0387, 0.1210, 0.0218,
                 0.0209, 0.0496, 0.0733, 0.0022, 0.0081, 0.0421,
                 0.0253, 0.0717, 0.0747, 0.0207, 0.0010, 0.0633,
                 0.0673, 0.0894, 0.0268, 0.0106, 0.0183, 0.0019,
                 0.0172, 0.0011};
```

Requirements

For this exercise you will develop two programs. The first one rotates a given plain text for a given shift. The second one will crack a given cipher text and display the original plain text on the screen.

a. **(15%)** Write a Java Interface for rotation ciphers. The interface should be called `RotationCipher` and declare the following **`public`** methods.

   - `rotate`, which should take a string `s` and an integer `n` as parameters and return the string `s` rotated by shift `n`.
   - `decipher`, that translates a (cipher text) string to a (plain text) string.
   - `frequencies`, which computes the letter frequencies in a given string (as size-26 arrays of **`double`**, as displayed above).
   - `chiSquared`, which returns the χ^2-score (a **`double`**) for two given sets of frequencies

b. **(25%)** Implement your interface in a class called `Caesar`. Note the hints and comments below to help you in case you're lost.

   You may assume that the original plain text is in English and that the cipher text has been produced using a Caesar cipher by some (unknown) shift. This shift has only been applied to letters, not anything else that may appear in the text, so spaces are spaces, any punctuation is left untouched, etc. Lower case letters are transformed to lower case letters. When computing letter frequencies you need not distinguish between lower case and capital letters.

c. **(15%)** Write an application called `Rotate`, which rotates a given (plain text) string by a given shift. This should make use of your `Caesar` class from part b). The shift as well as the input

string should be read as the first and second command line parameters, respectively. The (only!) output should be the rotated string in case all goes well. See below for sample outputs.

```
$ java Rotate 3 "The ships hung in the sky in much the same way that bricks don't."
Wkh vklsv kxqj lq wkh vnb lq pxfk wkh vdph zdb wkdw eulfnv grq'w.

$ java Rotate -13 "The ships hung in the sky in much the same way that bricks don't."
Gur fuvcf uhat va gur fxl va zhpu gur fnzr jnl gung oevpxf qba'g

$ java Rotate 13 The ships hung in the sky in much the same way that bricks don't.
Too many parameters!
Usage: java Rotate n "cipher text"

$ java Rotate 13
Too few parameters!
Usage: java Rotate n "cipher text"
```

d. **(15%)** Write an application called `BreakCaesar`, which finds out and prints a plain text message for a given cipher text string. As in part c), the input string should be read as (the only) command line parameter. Sample output below.

```
$ java BreakCaesar "Vg vf n zvfgnxr gb guvax lbh pna fbyir nal znwbe ceboyrzf whfg jvgu
    cbgngbrf."
It is a mistake to think you can solve any major problems just with potatoes.

$ java BreakCaesar
Too few parameters!
Usage: java BreakCaesar "cipher text"

$ java BreakCaesar Too Many Parameters
Too many parameters!
Usage: java BreakCaesar "cipher text"
```

e. **(10%)** Draw an UML class diagram that includes all classes and interfaces that you've written. This should be part of your report.

f. **(10%)** Document your code (i.e., all interfaces, classes and methods) using javadoc comments. Generate HTML documentation and put the generated files in a separate directory called 'docs'.

g. **(5%)** Caesar would have written in Latin instead of English. What would we do differently if we know the language we're examining isn't English but some other language?

h. **(5%)** Suppose we (somehow) know that the person doing the encryption uses one shift value for lower case letters, and a different shift value for upper case letters. What would we have to do differently? How would that affect our calculations, or how would we have to alter our program/calculations to account for this?

Parts g) and h) are bonus questions and require no code to be written. Just comment on these questions in your report.

For parts c) and d), make sure that your program handles the case where a user provides unexpected input (no, too few, too many, not the right kind of parameters) and complains with an appropriate error message. Test your application appropriately and document expected and observed behaviour in your report.

Hints and Comments

1. In Java, **char** and **int** variables are (more or less) interchangeable. A Java statement like

```
int diff = 'e' - 'b';
```

   is perfectly legal, i.e. Java can interpret the 'difference of two letters' with no problem, and this will give an integer value. If the two letters are of the same case, then this will give a value between -25 and 25. In particular, if `ch` is a lower case (**char**) letter, then

```
int diff = ch - 'a';
```

   tells you how many places after 'a' that letter is in the alphabet. (The value of `diff` will be between 0 and 25 inclusive). We can use this idea in encrypting/decrypting letters. Assuming that is a nonnegative integer, we can encrypt the (lower case) letter `ch` by doing the following:

```
char newChar = (char) ((ch - 'a' + shift) % 26  + 'a');
```

   What is this doing? First we find out the number of characters after 'a' for the letter `ch`, and add the shift. The % operator is 'mod', so we get the remainder left over after dividing by 26. That is doing the 'wrap around'. Then we turn this back into a by adding the letter `a` and typecasting to a **char** variable.

2. When translating letters, notice that If `ch-'a'` is between 0 and 25 (inclusive), then `ch` is a lower case letter, and we encrypt as above. Alternatively, if `ch-'A'` is between 0 and 25 (inclusive), we encrypt `ch` similarly to get a new upper case letter. You may also use helper methods `isLetter`, `isLowerCase` etc from the `Character` class.

3. In order to translate whole strings, recall the Java `String` methods we discussed in the practicals. In particular, if `str` is a `String`, then `str.charAt(i)` gives you the **char** at index `i` in `str`.

4. When counting letter frequencies remember that for this exercise, we consider upper and lower case to be the same and do not consider spaces and punctuation characters. For example, in the string `"Mississippi moon!"`, the frequency of the letter 'm' is $2/15$, while the frequency of the letter 's' is $4/15$.

Submission Instructions

- Deadline: Friday, 5 April 2019, 5:00pm
- Submission server: https://sam.csc.liv.ac.uk/COMP/Submissions.pl

Your submission should be a single compressed `.zip` file called **`comp122_assessment_2_SID.zip`** where SID should be replaced by your student ID. This file should include:

1. Your implementation. This must include the `.java` source files, *not* **`.class`** files! Put everything into a subfolder called 'src'
2. The generated HTML documentation from part f). This should be in a subfolder called 'docs'.
3. A report in `.pdf` format. Include sections for each relevant part of the exercise.

Marking Scheme

Each part counts towards the final mark of this coursework as follows.

a	b	c	d	e	f	g	h
15%	25%	15%	15%	10%	10%	5%	5%

If your code does not compile and run **on departmental computer systems** you will suffer a penalty of 5%. The same applies if your submission does not exactly follow the submission instructions regarding file name and type.

Your submission is an individual piece of work. **No collaboration with other students is allowed!** Expect that plagiarism detection software will be run on your submission to compare your work to that of other students. Late submissions and plagiarism/collusion are subject to the University Code of Practice on Assessment, available here[1]. In particular, since these are online submissions, a late penalty of 5% applies **for every calendar day**, starting on and including Saturday, 6 April.

Notice that **I will not grant extensions requested after the submission deadline on 05/04/2019!**

[1] https://www.liverpool.ac.uk/media/livacuk/tqsd/code-of-practice-on-assessment/code_of_practice_on_assessment.pdf