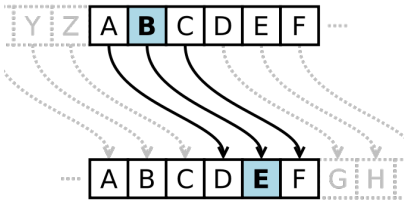
The Caesar Cipher

The Caesar cipher is an ancient method of encrypting text, i.e. to transform text into a format that is unreadable for anyone without a secret key. 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 outlined below. Your assignment is to implement this in Java.

Part 1: Encrypting and Decrypting

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 that "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).

Requirements

In a file called `Caesar.java`, implement the following (**`public static`**) methods.

- a method called `rotate` that rotates a single character. It should have two arguments: an integer (**`int`**) shift and a **`char`** to rotate, and return the character rotated by the given shift, as a **`char`**.

  Lower-case characters should be translated into lower-case characters, capitalised ones into capitalised ones, and all other characters should remain untouched.

- another method called `rotate` that rotates a whole string. It should have two arguments: an integer (`int`) shift and a `String` to rotate and return the string rotated by the given shift, as a `String`.
- a `main` method, that allows to encode/decode text, as follows. This should interpret the first two The first argument is interpreted as an integer shift and the second one as a (string) message to be rotated. The *only* output printed by the program should be the rotated string output.

  Moreover, the main method should check if it was called with exactly two arguments and complain otherwise. See below for example outputs. Your program needs to print the *exact* same output to be considered correct.

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

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

Part 2: 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, which is a single number.

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).

You are provided with a file called `Brutus.java`, which already defines letter frequencies in English texts. This is given as array of doubles, in alphabetical order:

```
public static final double[] english = {
    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
};
```

Accordingly, the frequency of the letter “a”, the probability that a randomly chosen letter in an English text is an “a”, is $\text{English}_a = 0.0855$ and can be accessed as `english[0]`. Similarly, $\text{English}_b = 0.0160$ and so on.

Requirements

In `Brutus.java`, write

- a method called `count` that takes a single `String` parameter and returns a length-26 integer array whose values reflect how often each character occurred. You should not make a difference between upper and lower case letters and the returned array should be in alphabetical order. This way, if `letterCounts` is an array resulting from your method then `letterCounts[25]` is the number of times the letter “z” or “Z” occurs.
- a method called `frequency` that takes a single `String` and returns a length-26 array of **`double`** s whose values correspond, in alphabetical order, to the frequency of the letter. This way, if `letterFreq` is an array resulting from this method then `letterFreq[24]` is the number of times the letter “x” or “X” occurs, divided by the length of the string input.
- a method called `chiSquared`, which returns the χ^2-score (a **`double`**) for two given sets of frequencies. That is, it should take *two* parameters, both of type **`double`**`[]`, and return a single **`double`** value that tells us how close these two arrays are. You may assume that the two inputs are always of length 26.
- a `main` method that can be used to decipher Caesar-encoded English cryptotext without the key. Of course, you should be using your `chiSquared` method as well as the given English letter frequencies.

The string that is to be deciphered should be read from the first command line argument and your program should ensure that it gets exactly this one argument and complain otherwise. Sample output below.

```
$> java Brutus "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 Brutus
Too few parameters!
Usage: java Brutus "cipher text"

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

Hints and Comments

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

   ```
   1  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

   ```
   1  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).

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 that you can access the individual characters in a string using the `charAt` method: 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 $4/15$.

5. Make sure you that your code is readable and appropriately documented. There will be points for javadoc comments that explain what each method does and how its parameters are interpreted.

6. You can run a (partial) automarker to check that your submission is in the correct format.

```
$> check50 liv-ac-uk/comp122/2021/problems/caesar
```

Submission

Submit you solution using `submit50` just like the lab exercises.

```
$> submit50 liv-ac-uk/comp122/2021/problems/caesar
```

You can submit multiple times and only the latest version (and it's submission time) will be considered for grading.

Submissions are subject to UoL's Code of Practice on Assessment and the usual late penalties (-5% per for each 24 hour period immediately following the deadline) apply. If you require an extension due to extenuating circumstances please get in touch with the CS student office (csstudy@liv.ac.uk) *before* the submission deadline. We will not grant any extensions afterwards.