24.3 *Lab:* Cryptogram Solver

Almost everyone is sooner or later tempted to either send a message in code or to decode an encrypted message. In a simple cryptogram puzzle, a small fragment of text is encrypted with a substitution cipher in which each letter is substituted with another letter. Something like: "Puffm, Cmafx!" To solve a cryptogram we usually look for familiar patterns of letters, especially in short words. We also evaluate frequencies of letters, guessing that in English the most frequent letters stand for 'e' or 't', while the least frequent letters stand for 'j' or 'q'. The purpose of this lab is to solve a cryptogram and to master the use of lookup tables and distributions in the process.

Our *Cryptogram Solver* program is interactive. After opening an encrypted text file, the user sees the encrypted text on the left side of the screen and the decoded text on the right. Initially, nothing is decoded — the decoded text has only dashes for all the letters. The user then enters substitutions for one or several letters, clicks the "Refresh" button, and immediately sees an updated version of the decoded text. In addition, the program can offer decoding hints based on the frequencies of letters in the encrypted text.

Cryptogram Solver can also create cryptograms. If you enter random substitutions for all letters (or click the "Encode" menu item) and then enter your text fragment on the left side (by typing it in, loading it from a text file, or cutting and pasting it from another program), then the text shown on the right side will be an encrypted version of your text.

Your task is to write a class `Enigma`, named after the famous German encryption machine.[enigma] Enigma was invented in 1918 and was first used in the banking business, but it very quickly found its way into the military. Enigma's codes were considered "unbreakable," but they were eventually broken, first by Polish codebreakers in 1932 and later, during WWII, by the codebreakers of the Bletchley Park project in Great Britain, led by Alan Turing,[turing] one of the founders of modern computer science. The battle between Enigma codemakers and codebreakers lasted through WWII, and the dramatic successes of Allied cryptanalysts provided invaluable intelligence information.

Your `Enigma` class should maintain a lookup table for substitutions for the letters 'A' through 'Z'. The easiest way to store the table is simply in a string of text. Initially the lookup table contains only dashes. For example:

```
private static String lookupTable = "--------------------------";
```

As decryption proceeds, the table is gradually filled and updated. You can change your guess for a letter as often as you want.

The `getNumericValue(char ch)` static method of the `Character` class returns consecutive integers for letters 'A' through 'Z' (10 for 'A', 11 for 'B', etc.; 0 though 9 are returned for digits between '0' and '9'). Therefore, if `ch` is an upper-case letter, then

```
int i = Character.getNumericValue(ch) -
                              Character.getNumericValue('A');
```

sets `i` to an integer in the range from 0 to 25, which can be used as an index into our lookup table (array or string) of 26 letters.

Your `Enigma` class should define three static methods:

```
void setSubstitutions(String subs);
String decode(String text);
String getHints(String text, String lettersByFrequency);
```

The first two methods support decoding (or encoding) of text; the last one supports computer-generated hints based on letter counts.

The `setSubstitutions(String subs)` method assumes that `subs` contains 26 characters and saves it as the lookup table.

The `decode(String text)` method decodes all the letters in `text` according to the current lookup table. `decode` leaves all characters that are not letters unchanged and preserves the upper or lower case of letters. It returns the decoded string, which has the same length as `text`.

The `getHints(String text, String lettersByFrequency)` method returns computer-generated hints for each letter in the encrypted text. It works as follows. First it counts the number of occurrences for each of the letters 'a' through 'z' in `text` (case blind) and saves these 26 counts in an array. Write a separate private method for that:

```
private static int[] countLetters(String text)
```

You should count all letters in one sweep over `text`. Start with zeroes in all counts, then increment the appropriate count for each letter.

After getting the counts for all letters, `getHints` creates and returns a `String` of "hints." The returned string `hints` should hold a permutation of letters 'A' through 'Z'; `hint.charAt(k)` will be displayed as a computer-generated hint for decoding the *k*-th letter of the alphabet. The hints should be based on comparing the order of letters by frequency in letter counts in encrypted text with the order of letters by frequency in plain (unencrypted) text. The `lettersByFrequency` parameter contains the letters 'A' through 'Z' arranged in increasing order of their frequencies in a sample text file. Suppose `lettersByFrequency`, passed to `getHints` is `"JQXZKBVWFUYMPGCLSDHROANITE"`. Then 'J' should be the hint for the least frequent letter in `text`, 'Q' for the second least frequent letter, and so on. The method's code is quite short once you figure out the algorithm. Try to figure it out yourself or read the (encrypted) hint below.

Mlk ze zvcyqutbf lufuvzq ty "lyqtuec ps hymetuec," nklhqupkn ue Hbzwtkq 18. Jyq kzhb kvkfket `hymetl[d]` ue tbk vkttkq hymetl zqqzs, juen tbk emfpkq yj kvkfketl `hymet[r]` lmhb tbzt `hymet[r] < hymet[d]` yq `hymet[r] == hymet[d]` zen `r < d`. Tbul emfpkq (hzvv ut `qzed`) ul tbk qzed yj tbk d-tb vkttkq yj tbk zvwbzpkt ue tkqfl yj utl jqkimkehs ue tbk kehqswtkn tkot. Ly `buetl[d]` lbymvn pk lkt ty `vkttkqlPsJqkimkehs.hbzqZt(qzed)`.

(The above paragraph is also available in `JM\Ch24\Cryptogram\hint.txt` file. You can use *Cryptogram Solver* on it even before you get the hints part working correctly. Just use a "stub" method for `getHints` that returns an arbitrary string of 26 characters.)

We generated our `lettersByFrequency` string by counting occurrences for the 26 letters in the file `sample.txt`. Therefore, if you load `sample.txt` (a plain text file) into the program, the hint displayed for each letter should be that same letter. This is an easy way to test your `countLetters` and `getHints` methods.

Combine your `Enigma` class with the `Cryptogram` and `CryprogramMenu` classes located in `JM\Ch24\Cryptogram`. Test your program with `sample.txt`, then try to decode `secret.txt`. Both these files are in `JM\Ch24\Cryptogram`, too.

Unfortunately, as you can see, our computer-generated hints turn out to be entirely unhelpful, except for the most frequent letter 'e'. Apparently we need a more sophisticated tool for solving cryptograms automatically — perhaps counting 2-D distributions for all pairs of adjacent letters, or even 3-D distributions for all triplets of letters, or a way to look for other patterns in the text.