Coursework: Permutation Cipher Deadline: Friday 12 April 5pm

A **permutation cipher** is an encryption that uses any permutation of the alphabet to encrypt a message. It resists brute-force attacks (trying all possible keys) because there are $26!$ keys. This is rather a lot:

$$26! = 403291461126605635584000000$$

The weakness (which we will obviously exploit!) of permutation ciphers is that they encrypt each letter individually, and uniformly (i.e., in a given encrypted message, the same character replaces 'D' everywhere, another stands for 'A' everywhere, etc). This means we can use patterns in natural language to guess what the key is.

In this coursework we will build a tool for working with permutation ciphers. We will be able to encrypt and decrypt messages, and we will attack an encrypted message with a **frequency attack**. This uses the relative frequency of each letter in the English language to guess a key. Only very rarely will this give the exact key — but on many occasions it is close enough that the partially deciphered text gives enough hints on how to complete the key. We will include some simple tools to help finish the job.

Part 1 (30%): We will first build a class to store keys, and the functions we will use with them. A key is any permutation of the alphabet. We will store a key internally as a permutation of the numbers 1 to 26, in a row vector of length 26, but make the display function such that it shows the corresponding letters. The identity permutation, the alphabet, is stored as the vector `1:26`, and you can generate a random permutation with `randperm(26)`. The coursework files include a script `tests.m` with useful tests and test data. The tests use random keys, so you could run them multiple times to be extra sure.

Build a class `Key` that stores and handles keys. It should have the following:

Properties:

- A property `perm` that stores a key as a permutation on the numbers 1 to 26. It should **not** store it as a string of letters.

Methods:

- A **constructor** method that takes a permutation `p` of the numbers 1 to 26 and creates a `Key` storing that permutation as its `perm` property.
- A **display** method that shows a `Key` as a permutation of the alphabet.
- A method `mtimes(l,m)` that takes two keys and gives their **composition**:

$$(l \circ m)\,(i) = l\,(m\,(i))\,.$$

Using the name `mtimes` means you can write `l * m` for $l \circ m$.

- A method `invert(k)` that gives the **inverse** of the key `k`. For every number `x` between 1 and 26, if `K.key(x) = y` then `K.invert.key(y) = x`.
- A method `encrypt(k,m)` that encrypts a message (a character vector) `m` with the key `k`. First, turn the message into all uppercase. Then apply the key to each uppercase letter in the message, leaving other characters as is. Make sure to convert from the ASCII indices (65–90) of the alphabet to the permutation indices (1–26) and back.
- A method `decrypt(k,m)` that decrypts a message `m` with the key `k`, by encrypting with the inverse key.

Test cases and examples are at the end of this document.

A **frequency attack** works as follows. Suppose our ciphertext has these letter counts:

27	85	106	83	42	54	6	249	220	427	147	93	1	252	290	270	75	57	143	198	9	91	271	334	2	151
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Sorting this by occurrence count (from small to large), we get:

1	2	6	9	27	42	54	57	75	83	85	91	93	106	143	147	151	198	220	249	252	270	271	290	334	427
M	Y	G	U	A	E	F	R	Q	D	B	V	L	C	S	K	Z	T	I	H	N	P	W	O	X	J

We can compare this against the frequency of letters in English (from rare to common):

Z	Q	X	J	K	V	B	P	Y	G	F	W	M	U	C	L	D	R	H	S	N	I	O	A	T	E

This tells us that a likely key used to encrypt the ciphertext message is, starting from the most frequent letter, the one taking E $\rightarrow$ J, T $\rightarrow$ X, A $\rightarrow$ O, etc. To help you implement this, you are provided with a function `permutation` that, given the letter counts (the first line above), returns the matching permutation of 1 to 26 (second line above). [1]

Part 2 (35%): We will build a class to try and decrypt a permutation-cipher encoded message. Write a class `Attack` with the following properties and methods.

[1] The `permutation` function works by combining the letter counts with the identity permutation `1:26` in a 2-dimensional array, sorting this according to letter count, and returning the resulting permutation on 1 to 26.

Properties:

- A property `ciphertext` that stores the encrypted message.
- A property `key` that stores a `Key` object.

Methods:

- A **constructor** method that takes an encrypted message as input, stores it as the `ciphertext` property, and initializes the `key` with the identity permutation `1:26`.
- A **display** method that shows the stored key and the first 300 characters of the message, decrypted with the current key. The message is probably too long to display in its entirety (short messages are hard to decode), and displaying it (partially) decrypted shows the progress made thus far in breaking the cipher. Format it nicely, indicating the key and the message (you can use `char(13)` to create a line break).
- A method `lettercount` that counts the occurrences of each letter of the alphabet in the ciphertext, and returns them as a 1×26 array. Make sure to pre-allocate the return array. You should **not** use built-in counting functions such as `find` or `histcount`.
- A method `attack` that carries out a frequency attack:

  1. allocate an array with English letter frequencies (see above);
  2. use `lettercount` to retrieve the letter frequency in the cyphertext;
  3. use the `permutation` function to sort the alphabet by (reverse) letter count;
  4. combine (1) and (3) in the way described above to guess a key.

  Your method should return the new `Attack` object, with the same ciphertext but the new key. You can approach (4) above in two ways: by directly computing the new key (easiest), or by using the `invert` and composition (`*`) methods of the `Key` class (slightly harder but very satisfying).

Tests and examples are at the end of this document. The ciphertext that we will be attacking is in the script `ciphertext.m`; running it assigns the variable `secret`.

As was to be expected, the letter frequency of our ciphertext is not exactly that of English in general. The key generated by our frequency attack is off, and the message is still very much illegible. But at the same time, several letters are spot on: in particular the word “the” is complete. We will finish the job by creating a method `sample` to inspect a random sample of the current decoding in search of familiar words, and a method `swap` to swap two letters in the key.

Part 3 (15%):

- Add a method `sample` to your `Attack` class that displays a random piece of the ciphertext, decrypted with the current key, of 300 characters in length. Make sure that the sample text is not cut short by reaching the end of the ciphertext. For efficiency, **first** select the sample and **then** decrypt it, instead of the other way around (which would decrypt the whole ciphertext). Use the Matlab function `randi(n)` to generate a random integer between 1 and n (inclusive).
- Add a method `swap` to your `Key` class to swap two letters in the key. The function should take a key and two characters as input, and swap the two characters on the **input** side of the key. That is because we're using the key to **decrypt** the message, and we want to swap two letters in the decrypted ciphertext to create recognizable words. So, if `k` is the key given by the first permutation below, swapping H and P, `swap(k,'H','P')`, should give the second key below.

```
                  ↓               ↓
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Y J M T X N I L W U H K F A D V E Q B G C Z P O S R

                  ↓               ↓
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Y J M T X N I V W U H K F A D L E Q B G C Z P O S R
```

  You may compute the resulting key directly, or by giving the permutation that swaps the two given letters and then composing with the original key using (`*`).
- Add a corresponding method `swap` to your `Attack` class, that swaps two letters in the `key` property by calling the `swap` method of the `Key` class.

Part 4 (20%): To wrap up, we will polish our functions and add some convenient functionality. One thing we'll add is **undo** functionality for the `swap` function, using a list to store previous swaps. A list class is included with the files you were given. You may re-use code from the tutorials, and any solutions you are given on Moodle, to implement this.

- Edit the constructor for the `Key` class so that, in addition to the current functionality, it accepts a character array as input, and calling `Key` without arguments creates a random permutation key.
- Add a property `past` to your `Attack` class, and let the constructor initialize it with the empty list.

- Edit the `swap` method of the `Attack` class so that it stores appropriate "undo" information in the `past` property. Edit the `attack` method so that it erases any "undo" information.
- Add an `undo` method that reverts the previous swap, or if there is no undo information, writes an appropriate message and does nothing.
- Use the tools you have built to decrypt the secret message in `ciphertext.m`. Put the key you've found in a script `solution.m` as the variable `theKey`. It should be an object of the `Key` class, and it should be the only variable declared in the script.

Submission: submit the relevant Matlab files on Moodle:

`Key.m` `Attack.m` `solution.m`

Do not submit the other files, and make sure your code works with the unedited files. Either upload your files directly, or as a zip-file named `cw2.zip` (zip the files directly, and not the folder containing them). **You should also submit a .txt file with your team's information. One submission per team.**The deadline is:

Friday 12 April 2019, 5pm

Marking: marking will be done in two stages: first, by a number of automated tests; second, by hand. If your code works as required and looks reasonable to the eye, with (at least) minimal styling and commenting, you will receive full marks. For automated tests that fail, partial marks may still be given, but only if style and comments make it possible to examine the code properly.[2] In principle, and within reason, your code will not be marked for efficiency.

[2]This is meant to reflect the real world: perfectly working code can live undocumented and unmaintained for long periods of time; broken but well-documented code may be fixed; broken, undocumented code is a source of utter despair and will be discarded where possible...

Tests and examples

Part 1:

```
>> tests
l * k : passed
k * k.invert : passed
k.invert * k : passed
k-1 * l-1 = (l*k)-1 : passed

>> k = Key(p)                    % Permutation p given by tests.m
k =
CVWKXAYBTPOESJLHIFQZDRGMNU

>> l = Key(q)                    % Permutation q given by tests.m
l =
PDQRIXYJVFCLKBGZUAWOHESNMT

>> k * l
ans =
HKIFTMNPRAWEOVYUDCGLBXQJSZ

>> l * k
ans =
QESCNPMDOZGIWFLJVXUTRAYKBH

>> k.invert
ans =
FHAULRWPQNDOXYKJSVMIZBCEGT

>> encrypt(k,'The North remembers')
ans =
ZBX JLFZB FXSXSVXFQ

>> decrypt(k,ans)
ans =
THE NORTH REMEMBERS
```

Part 2:

```
>> ciphertext
>> A = Attack(secret)     % String "secret" given by ciphertext.m
A =
Key:
ABCDEFGHIJKLMNOPQRSTUVWXYZ

Partial decoding:
XIJ PSAJHXKTJ OB XIJ SPHCWHF VJH

IOZVJN IPS RJJH NJPXJS BOT NOVJ IOKTN WH NWZJHCJ QWXI IWN ZOHF,
XIWH RPCE CKTAJS OAJT P CIJVWCPZ AJNNJZ WH QIWCI IJ QPN RTJQWHF
P DPTXWCKZPTZL VPZOSOTOKN DTOSKCX. IWN IJPS QPN NKHE KDOH IWN
RTJPNX, PHS IJ ZOOEJS BTOV VL DOWHX OB AWJQ ZWEJ P NXTPHFJ, ZPHE
RWTS, Q

>> A.lettercount
ans =
  Columns 1 through 10
    27    85   106    83    42    54     6   249   220   427
  Columns 11 through 20
   147    93     1   252   290   270    75    57   143   198
  Columns 21 through 26
     9    91   271   334     2   151

>> A = A.attack
A =
Key:
OFSZJBDIPUAKLNWRYTHXCEVGQM

Partial decoding:
THE ICKESTLRE AF THE CISUOSB WES

HADWEN HIC PEES NEITEC FAR NAWE HALRN OS NODESUE YOTH HON DASB,
THOS PIUV ULRKEC AKER I UHEWOUID KENNED OS YHOUH HE YIN PREYOSB
I GIRTOULDIRDM WIDACARALN GRACLUT. HON HEIC YIN NLSV LGAS HON
PREINT, ISC HE DAAVEC FRAW WM GAOST AF KOEY DOVE I NTRISBE, DISV
PORC, Y
```

Part 3:

```
>> A.sample

N EJUEGT YOTH THLRNTAS.
4. MAL TADC WE, FALR YEEVN IBA, THIT THLRNTAS HIC IS AGTOAS AS
NAWE NALTH IFROUIS GRAGERTM YHOUH YALDC EJGORE OS I WASTH, ISC
YHOUH HE CENOREC MAL TA NHIRE YOTH HOW. 5. MALR UHEXLE-PAAV ON
DAUVEC OS WM CRIYER, ISC MAL HIKE SAT INVEC FAR THE VEM. 6. MAL
CA SAT GRAGANE TA O

>> A.key.swap('A','B')
ans =
FOSZJBDIPUAKLNWRYTHXCEVGQM

>> A = A.swap('A','O')
A =
Key:
WFSZJBDIPUAKLNORYTHXCEVGQM

Partial decoding:
THE ICKESTLRE OF THE CISUASB WES
% ... truncated
```

Part 4:

```
>> Key('GHCWISQDOETMZNBULJRKPYVFXA')
ans =
GHCWISQDOETMZNBULJRKPYVFXA

>> Key
ans =
YOIDWZUJGAVNRXTSLHPKQFBMCE

>> A = A.attack;
>> A = A.swap('A','O');
>> A = A.undo
A =
Key:
OFSZJBDIPUAKLNWRYTHXCEVGQM

Partial decoding:
THE ICKESTLRE AF THE CISUOSB WES
% ... truncated

>> A = A.undo
No undo information

A =
Key:
OFSZJBDIPUAKLNWRYTHXCEVGQM

Partial decoding:
THE ICKESTLRE AF THE CISUOSB WES
% ... truncated
```