Compléments d'informatique

Projet 2 : Enigma

22 octobre 2024

L'objectif concret de ce projet est d'implémenter une version de la machine Enigma, une machine de chiffrement de l'information utilisée par l'Allemagne nazie pendant la seconde guerre mondiale [1], ainsi que d'implémenter un algorithme permettant de *cracker* un message crypté via une machine Enigma. L'objectif pédagogique est de vous faire pratiquer la programmation modulaire, le programme complet étant constitué de plusieurs modules séparés interagissant entre eux. Ce projet est à réaliser **seul ou à deux**. La date limite de remise est précisée sur eCampus et sur Gradescope.

1 Description de la machine

La machine Enigma permet de chiffrer des messages (uniquement composés de lettres) et ensuite de les déchiffrer. Pour ce faire, elle reçoit les lettres une par une et les encrypte en d'autres lettres. Le message généré fait donc la même longueur que le message initial, et est également composé uniquement de lettres. Ce qui rend la machine efficace est sa propriété d'encrypter différemment la même lettre. Ainsi, encrypter le message `aaaa` ne mènera pas à un code composé de quatre fois la même lettre. Pour comprendre ceci, il faut examiner les composants de la machine.

Description des composants. Une machine Enigma est composée de n *rotors* (historiquement, $n = 3$ mais nous resterons général) et d'un *reflector*. Ces composants permettent tous d'encoder une lettre en une autre. La spécificité du *reflector* est que son mappage est symétrique : s'il encode `a` en `p`, alors `p` sera encodé en `a`. Lorsqu'une lettre doit être encodée, elle passe par tous les *rotors*, puis par le *reflector*, et puis repasse par tous les *rotors*. À chaque étape, une nouvelle lettre est générée, il y a donc au final $2n + 1$ changements de lettre. Chaque composant a sa propre table de mappage, lui indiquant, pour chaque lettre,

1. `https://fr.wikipedia.org/wiki/Enigma_(machine)`

celle qu'il devrait générer. De plus, lors du trajet *retour* à travers les *rotors*, ce sont les tables de mappage symétriques qui sont utilisées. La figure 1 montre un exemple d'encodage de la lettre `a` où l'on ne considère que les quatre premières lettres de l'alphabet et où la machine ne contient qu'un *rotor*. Dans ce cas, la machine sort `c`. Remarquez que si l'on avait donné `c` en entrée, on aurait obtenu un `a`. Cette symétrie mène à une autre propriété de la machine : les processus d'encryption et de decryption sont identiques. Ainsi, pour décoder un message, il suffit de prendre une machine Enigma avec le même état que celle utilisée pour le coder (nous reviendrons plus tard sur cette notion d'état) et de donner les caractères du message codé en entrée.

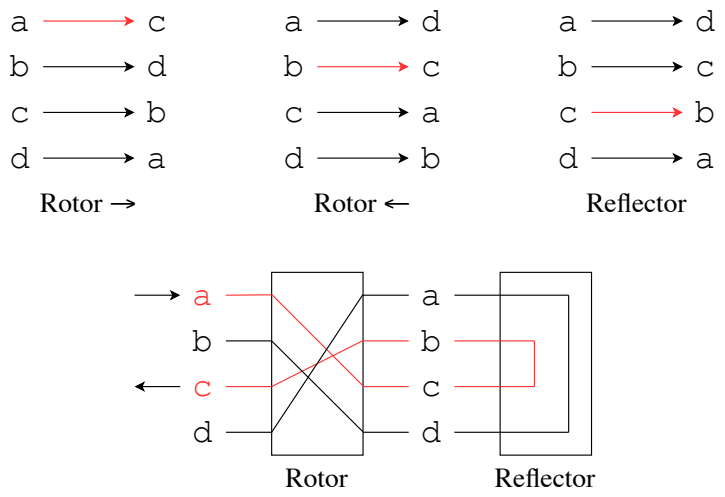

FIGURE 1 – Simulation d'une machine Enigma avec un *rotor* et avec seulement quatre lettres. En haut se trouvent les trois tables de mappage : respectivement celle du *rotor* lors de l'*aller*, celle du *rotor* lors du *retour* et celle du *reflector*. En bas se trouve une illustration graphique du *chemin* pris dans la machine lorsque la lettre `a` est donnée en input.

Rotation des *rotors*. Nous savons maintenant comment encoder des messages. Cependant, si nous appliquons simplement cette méthode, l'encodage d'une même lettre sera toujours la même lettre, or ce n'est pas le cas avec les machines Enigma. C'est ici qu'intervient la notion de *rotation* ou de *shift* : au fur et à mesure que des lettres sont données en entrée, les *rotors* vont tourner, ce qui va changer le mappage. Pour expliquer ceci, il est important de voir les tables de mappage comme des *sauts* entre les lettres. Ainsi, encoder `a` en `c` revient à faire `2` sauts dans le tableau des lettres. Il est donc possible de représenter les tables de mappage comme des tableaux d'entiers indiquant, pour chaque lettre, le nombre de sauts à faire (voir figure 2).

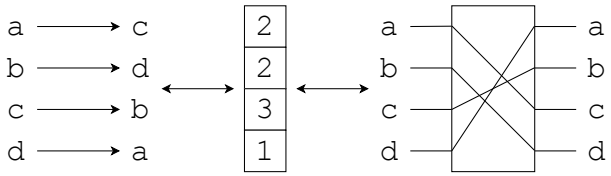

FIGURE 2 – Différentes manières de visualiser une table de mappage.

À chaque fois qu'un lettre est donnée à la machine, ce tableau d'entiers est décalé de une case de manière circulaire (la dernière case devenant la première). Dès lors, si l'on donne deux `a` de suite à la machine, celle-ci sortira un `c` et ensuite un `d` (voir figure 3). Notons que la rotation des *rotors* se fait **avant** de générer une nouvelle lettre.

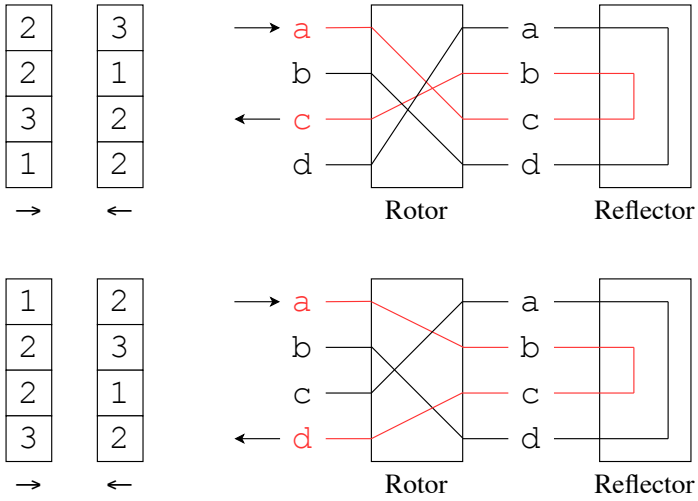

FIGURE 3 – Simulation de l'encodage de deux lettres par une machine Enigma ne contenant qu'un seul *rotor*. Les tables de mappage du *rotor* sont décalées après la première lettre.

Ajouter des *rotors*. Une fois que le principe est compris pour une machine avec un seul *rotor*, il est assez simple de passer à des machines en contenant plusieurs. La subtilité réside dans quels *rotors* faire tourner lorsqu'une lettre est donnée, car tous ne tournent pas à chaque fois. En effet, seulement le premier (appelé *rotor rapide*) tourne à chaque lettre donnée. Les suivants ne tournent que lorsque le précédent a fait un tour complet (c'est-à-dire 26 rotations si l'on considère toutes les lettres). Le second tournera donc lorsque le premier fera sa 26ème rotation, c'est-à-dire lorsque la machine recevra la 26ème lettre. Le troisième *rotor* tournera quant à lui lorsque la machine recevra la $26^2 = 676$ ème lettre. Et

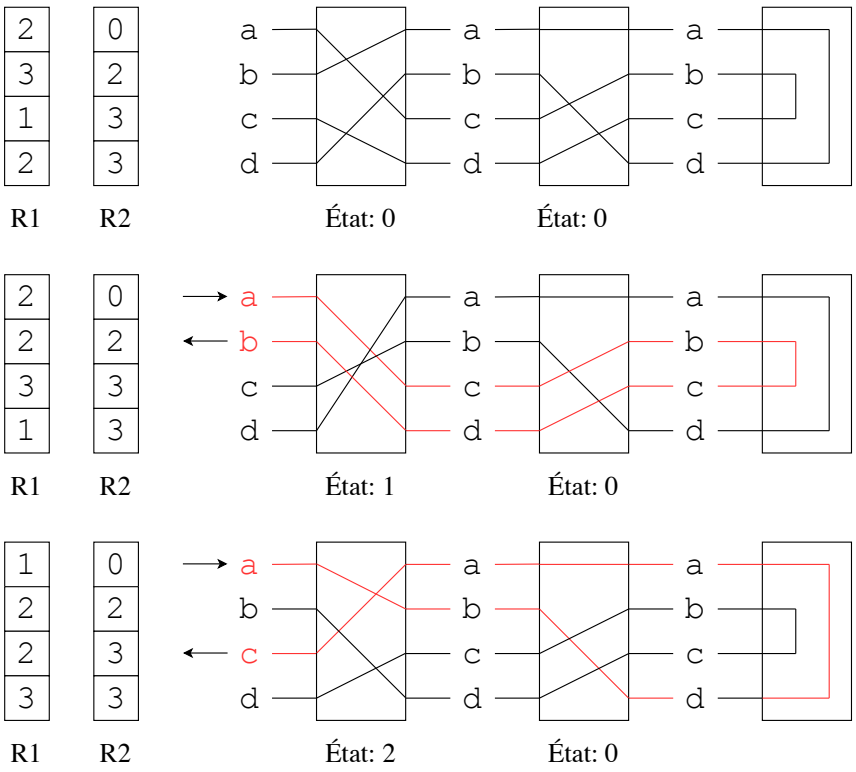

FIGURE 4 – Simulation d'une machine Enigma à deux *rotors* avec deux lettres en entrée.

ainsi de suite pour les *rotors* suivants. Le *reflector*, lui, reste toujours fixe.

Lorsqu'une lettre est donnée en entrée, elle passe par les *rotors* par **ordre décroissant** de rapidité. Elle passe ensuite par le *reflector*, et puis repasse dans les *rotors* dans le sens inverse (voir figure 4).

Machine Enigma et notion d'état. Une machine Enigma particulière est donc définie d'une part par les tables de mappage de ses *rotors* et de son *reflector* mais également par la position initiale des *rotors* à utiliser lors de l'encrytion, c'est-à-dire de combien de positions chacun des *rotors* doit être décalé avec de commencer l'encryption ou la décryption. Cette position initiale constitue une sécurité supplémentaire de la machine. Un utilisateur qui connaîtrait la table de mappage des *rotors* mais pas la position initiale des *rotors* ne serait pas capable de décrypter un message intercepté. On appellera l'ensemble des positions courante des *rotors* d'une machine son *état* (de même l'état d'un *rotor* sera sa position courante). L'état de la machine sera représenté par n valeurs entières entre 0 et 25 pour

une machine à n *rotors*. Dans le domaine de la cryptographie, l'état initiale de la machine constitue ce qu'on appelle une *clé de chiffrement*, parce que sa connaissance est nécessaire pour décrypter une message. Pour l'implémentation des rotations, la mise à leur état initial des *rotors* doit être prise comme si elle résultait de la génération d'autant de caractères. Ainsi, chaque *rotor* intérieur devra être décalé quand le *rotor* précédent (à sa gauche) passera de l'état 25 à l'état 0.

2 Implémentation de la machine

L'implémentation de la machine est basée sur trois modules que vous devrez implémenter :

- `mapper.c/.h` implémentant une table de mappage.
- `rotor.c/.h` implémentant un *rotor*.
- `enigma.c/.h` implémentant une machine Enigma.

On supposera dans ce projet que le machine encrypte des textes ne contenant que les 26 lettres minuscules de `a` à `z` (et donc aucun signe de ponctuation, espace ou lettres accentuées).

2.1 Fichiers `mapper.c` et `mapper.h`

Ces fichiers implémentent une table de mappage, qui vous servira à implémenter le *reflector* ainsi que les *rotors*. Vous devrez choisir une structure pour le type `Mapper` et implémenter les fonctions de l'interface.

Fonctions de l'interface. Les fonctions à écrire sont les suivantes :

`Mapper *mapperCreate(char *mapping_table, int state)` : crée et initialise une table de mappage sur base d'un tableau de caractères `mapping_table` et d'un état `state`. Le tableau `mapping_table` est une chaîne de caractères (terminée par un caractère nul) de longueur 26 contenant une permutation des lettres de `a` à `z` qui représente la table de mappage initiale (cette chaîne serait par exemple `"cdba"` pour la table de mappage en haut à gauche de la figure 1). L'argument `state` est un entier entre 0 et 25 représentant le décalage initial à appliquer pour obtenir le mappage effectif. Cet argument vous sera utile pour l'implémentation du *rotor* (pour le *reflector*, cet argument devrait rester à 0). La chaîne `mapping_table` peut être libéré par la fonction appelante et ne doit donc pas être stockée telle quelle dans la structure.

`void mapperFree(Mapper *mapper)` : libére la table de mappage de la mémoire.

`char mapperMap(Mapper *mapper, char input)` : renvoie la lettre correspondant à `input` dans la table de mappage en prenant en compte l'état courant.

`bool mapperShift(Mapper *mapper)` : effectue une rotation de la table de mappage, ce qui incrémente son état de 1(modulo 26). La fonction doit renvoyer `true` si l'état passe de 25 à 0, `false` sinon.

`int mapperGetState(Mapper *mapper)` : renvoie l'état (c'est-à-dire le décalage) courant de la table de mappage.

`void mapperSetState(Mapper *mapper, int state)` : fixe l'état de la table de mappage à la nouvelle valeur `state`.

2.2 Fichiers `rotor.c` et `rotor.h`

Ces fichiers permettent d'implémenter un *rotor* tel que décrit plus haut. Vous devrez choisir une structure pour le type `Rotor` et implémenter les fonctions de l'interface. On vous demande d'utiliser dans votre implémentation le type `Mapper`. Un *rotor* devrait utiliser deux tables de mappage, respectivement pour les codages aller et retour à travers le *rotor*.

Fonctions de l'interface. Les fonctions à écrire sont les suivantes :

`Rotor *rotorCreate(char *forward_table, int state)` : crée et initialise un *rotor*. La table `forward_table` a le même format que l'argument `mapping_table` de `mapperCreate` et correspond à la table de mappage aller. `state` est le décalage initial entre 0 et 25 du rotor.

`void rotorFree(Rotor *rotor)` : libère le *rotor* de la mémoire.

`char rotorMapForward(Rotor *rotor, char input)` : renvoie la lettre correspond à la lettre `input` lors de l'encodage *aller* par le *rotor*.

`char rotorMapBackward(Rotor *rotor, char input)` : renvoie la lettre correspond à la lettre `input` lors de l'encodage *retour* par le *rotor*.

`bool rotorShift(Rotor *rotor)` : effectue une rotation du *rotor* d'une position à partir de son état courant. Renvoie `true` lorsque l'état passe de 25 à 0, `false` sinon.

`Mapper *rotorGetForwardMapper(Rotor *rotor)` : renvoie une table de mappage correspondant à l'étape de codage aller du rotor (prenant en compte l'état courant).

`Mapper *rotorGetBackwardMapper(Rotor *rotor)` : renvoie une table de mappage correspondant à l'étape de codage retour du *rotor* (prenant en compte l'état courant).

`int rotorGetState(Rotor *rotor)` : renvoie l'état du *rotor*.

`void rotorSetState(Rotor *rotor, int state)` : fixe l'état du *rotor* à `state`.

2.3 Fichiers `enigma.c` et `enigma.h`

Ces fichiers permettent d'implémenter une machine Enigma telle que décrite plus haut. Vous devrez choisir une structure pour le type `Enigma` et implémenter les fonctions de l'interface. On vous demande d'utiliser dans votre implémentation le type `Rotor` pour les *rotors* et le type `Mapper` pour le *reflector*.

Fonctions de l'interface. Les fonctions à écrire sont les suivantes :

```
Enigma *enigmaCreate(int num_rotors, char **rotor_tables,
                     int *rotor_states, char *reflector_table)
```

Crée une machine Enigam avec `num_rotors` *rotors*. Le tableau `rotor_tables` contient les tables de mappage initiale de chacun des *rotors* du plus rapide au plus lent et le tableau `reflector_table` contient la table de mappage du *reflector*. Le tableau `rotor_states` de taille `num_rotors` contient l'état initial de chacun des *rotors*.

`void enigmaFree(Enigma *enigma)` : libère la machine de la mémoire.

`void enigmaEncrypt(Enigma *enigma, char *input, char *tofill)` : encrypte le message `input` avec la machine `enigma`. Le message encrypté doit être placé dans le tableau `tofill` préalloué (et devra se terminer par un `'\0'`). L'état de la machine ne doit pas être réinitialisé après encryption.

`void enigmaReset(Enigma *enigma, int *rotor_states)` : si `rotor_states` est `NULL`, la machine est réinitialisée à son état initial (défini à sa création). Sinon, l'état des *rotors* est fixés aux nouvelles valeurs présentes dans le tableau `rotor_states` et cet état devient le nouvel état initial de la machine.

`int enigmaGetNumRotors(Enigma *enigma)` : renvoie le nombre de *rotors* de la machine.

`Rotor *enigmaGetRotor(Enigma *enigma, int index)` : renvoie le *rotor* d'indice `index` de la machine (entre 0 et `num_rotors` $-$ 1).

`Mapper *enigmaGetReflector(Enigma *enigma)` : renvoie le *reflector* de la machine.

`void enigmaGetRotorStates(Enigma *enigma, int *tofill)` : renvoie dans le tableau

préalloué `tofill` l'état des *rotors*.

2.4 Fichiers `main.c` et `mappings.h`

Le fichier `main.c` vous fournit du code pour tester votre implémentation. Une fois compilé par le `Makefile`, un exécutable nommé `enigma` vous permet d'encrypter/décrypter un message avec l'option `-e`. Le fichier `mappings.h` définit une série de tables de mappage pour les *rotors* et pour le *reflector* que vous pouvez utiliser pour vos tests. Par exemple, la commande :

```
./enigma -e helloworld 0 10 1 11 2 12 0
```

encrypte le message `helloworld` en utilisant les trois premiers mappages de `mappings.h` pour les *rotors* avec comme valeur de décalage (état) respectif pour ces *rotors* 10, 11 et 12 et le premier mappage de `mappings.h` pour le *reflector*. Lancez la commande `./enigma` dans un terminal pour avoir des informations sur la syntaxe des arguments.

3 Crackage du code par analyse statistique

Une fois la machine implémentée, on aimerait implémenter un algorithme pour la "cracker". Le crackage d'un algorithme d'encryption consiste à décrypter un message sans avoir connaissance de la clé de chiffrement. On considéra deux niveaux de crackage de la machine. Dans le premier, on supposera les *rotors* et le *reflector* connus et seule l'état initial de la machine ayant encrypté le message devra être déterminé. Dans le second, il s'agira en plus de retrouver les *rotors* et le *reflector* utilisés en supposant qu'ils ont été choisis parmi un ensemble de table de mappage candidates.

Crackage de l'état initial. Dans ce mode, on suppose connaître parfaitement les *rotors* et le *reflector* de la machine et il s'agit de trouver l'état initial des *rotors* qui a permis d'obtenir un message encrypté donné. Une fois cet état déterminé, on pourra décrypter le message, ainsi que d'autres messages encryptés par la même machine.

Une solution naïve, appelée attaque par force brute, consiste à énumerer tous les états possibles, à décrypter le message avec chacune d'eux et à s'arrêter dès que le message décrypté semble être le message original. Il y a deux problèmes avec cette approche : (1) le nombre d'états potentiels peut être trop important pour permettre leur énumération en un temps raisonnable, (2) si on veut automatiser la procédure, il n'est pas possible de regarder chaque message décrypté pour détecter que la décryption a réussi. Pour le premier point, dans le cas d'Enigma, si le nombre de *rotors* n'est pas trop élevé, l'énumération de tous les états possibles (il y en a 26^n pour n *rotors*) reste faisable avec un ordinateur moderne [2]. Pour le deuxième point, il y a plusieurs techniques possibles. L'approche qu'on vous propose

2. La machine Enigma date d'une époque où les ordinateurs n'étaient pas très puissants.

d'implémenter n'a besoin que de savoir dans quelle langue le message original est écrit. Sur base de cette information, il est en effet possible d'attribuer un score de vraisemblance au message décrypté en regardant si la fréquence d'apparition de paires de lettres consécutives, appelés des bigrammes, correspond à la fréquence d'apparition de ces bigrammes dans la langue en question. Plus formellement, si on note $S = s_1 s_2 \dots s_m$ le message décrypté de longueur m, le score de ce message sera calculé par l'expression suivante :

$$Score(S) = \sum_{i=1}^{m-1} \ln(f_{s_i,s_{i+1}}), \tag{1}$$

où $f_{s_i,s_{i+1}} \in [0,0;1,0]$ est la fréquence d'apparition du bigramme $s_i s_{i+1}$ dans la langue du message. Le crackage du code consistera alors à déterminer l'état de la machine qui donne le message décrypté de score maximum. Plus le message encrypté est long, plus le crackage a des chances de bien fonctionner.

Crackage complet. Dans la machine Enigma originale, pour rendre le crackage plus compliqué, les n *rotors* de la machine n'étaient pas fixés à l'avance mais étaient choisis parmi un ensemble de m *rotors* candidats, un même *rotor* pouvant être utilisé plusieurs fois au sein de la même machine. De même le *reflector* était également choisi parmi un ensemble prédéfini de l *reflector*s. Etant donné le nombre n de *rotors* de la machine, le crackage demande donc de déterminer le *reflector*, chacun des n *rotors* et l'état initial de chacun des n *rotors*. L'approche de crackage consiste toujours à énumérer toutes les configurations possibles de la machine pour rechercher celle qui maximise le score du message décrypté. Le temps de calcul est cependant fortement augmenté, d'autant plus si m et l sont grands.

4 Implémentation du crackage (`crack_enigma.c/.h`)

L'algorithme de crackage devra être implémenté dans les fichiers `crack_enigma.c` et `crack_enigma.h`. Ce module n'implémente pas de nouveau type. Vous devrez implémenter uniquement les deux fonctions suivantes :

`double score_text(char *text, double **bigrams)` : calcule le score du message `text` selon la formule (1). Le tableau `bigrams` est un tableau à deux dimensions contenant les fréquences des bigrammes dans la langue anglaise. L'indexation suivant l'ordre alphabétique (par exemple, `bigrams[0][5]` est la fréquence du bigramme `'af'`).

`void crack_enigma_state(char *ciphertext, double **bigrams, Enigma *enigma)` : cette fonction implémente le crackage de l'état d'une machine Enigma donnée selon la procédure force brute décrite plus haut. Cette fonction doit mettre la machine en argument dans l'état initial qui a servi à encrypter le message `ciphertext` (via la fonction `enigmaReset`).

`void crack_enigma_full(char *ciphertext, double **bigrams, int num_rotors)` : cette fonction implémente le crackage complet d'une machine Enigma selon la procédure décrite ci-dessus, en supposant que cette machine contient `num_rotors it` choisi parmi ceux définis dans le tableau `rotor_tables` et un réflecteur choisi parmi ceux définis dans le tableau `reflector_tables` du fichier `mappings.h`. La fonction doit renvoyer une machine Enigma correctement configurer pour pouvoir décrypter le message `ciphertext`.

Le fichier `main.c` vous permettra de tester votre implémentation au moyen de la commande `-c`. Lancez `./enigma` dans un terminal pour obtenir une description des arguments de cet exécutable. Nous vous fournissons deux messages à décrypter. Pour le premier, vous savez qu'il a été encrypté avec une machine composé des 3 premiers *rotors* et du premier *reflector* dans le fichier `mappings.h`. Pour le second, vous savez seulement qu'il a été encrypté avec une machine à 3 *rotors*.

5 Conseils d'implémentation

Nous vous conseillons logiquement d'implémenter les fichiers `mapper.c`, `rotor.c` et `enigma.c` dans cet ordre en testant individuellement chacun des fichiers. Pour `mapper.c`, essayez de minimiser autant que possible les complexités des fonctions `mapperMap` et `mapperShift` (qui devront être appelées souvent pour le crackage). Une manière de faire est de pré-calculer une fois pour toute le tableau de décalage (voir figure 2) à partir de la table de mappage et d'éviter d'explicitement le décaler lors d'un appel `mapperShift`. L'implémentation de `mapperMap` est sans doute l'opération la plus compliquée (surtout à comprendre, pas à implémenter). Prenez le temps d'y réfléchir sur feuille avant de vous lancer. Comme indiqué plus haut, le *rotor* devrait être implémenté sur base de deux structures de type `Mapper`, une pour l'aller et l'autre pour le retour, les deux pouvant être pré-calculées également une fois pour toute. On peut montrer en effet que décaler les deux tables de mappage aller et retour initiales donne bien le résultat escompté. La machine Enigma se basera finalement sur n structures de type `Rotor` et une structure de type `Mapper` pour le *reflector*. La gestion des rotations devraient être aisée en se basant sur le retour de la fonction `rotorShift`.

Une fois que votre machine fonctionne, vous pouvez passer au crackage. La fonction `score_text` n'appelle pas de commentaire particulier. La fonction `crack_enigma_state` demande d'itérer sur tous les états possibles. Il possible de le faire avec une boucle principale dans laquelle vous devez passer d'un état au suivant. La fonction `crack_enigma_full` est plus compliquée. Vous devez énumérer toutes les configurations possibles de la machine en termes de *rotors* et *reflector*. Une implémentation récursive pourrait être plus simple pour faire cette énumération. Vous pouvez vous inspirer pour ça de la solution de l'exercice 5 du TP1.

6 soumission et système de cotation

Vous devez soumettre (uniquement) les fichiers suivants sur Gradescope :

— `mapper.c`, `rotor.c`, `enigma.c`, et `crack_enigma.c`.

— `rapport.txt`

Dans ce dernier fichier vous sont demandés les contributions de chacun au projet, des analyses de complexités pour certains fonctions, et le décryptage que vous avez obtenu avec les deux techniques de crackage.

Pour information, si vous n'implémentez que la machine Enigma mais pas du tout le crackage, votre cote maximale pour ce projet sera 14/20. Si vous implémentez le crackage de l'état mais pas le crackage complet, votre cote maximale sera 17/20. Dans ce dernier cas, il suffira que votre fonction `crack_enigma` renvoie `NULL` dans le cas où l'argument `enigma` est `NULL`.

Bon travail !