ENIGMA ET LA BOMBE CRYPTOLOGIQUE

Jonathan CERTES

1 Environnement de travail

1.1 Travail dans une machine virtuelle

Télécharger Oracle VirtualBox :
`https://www.virtualbox.org/wiki/Downloads`
Installer VirtualBox sur sa machine personnelle :
`https://www.virtualbox.org/manual/UserManual.html#installation`
Télécharger la machine virtuelle fournie par l'enseignant.
Importer la machine virtuelle dans VirtualBox :
`https://www.virtualbox.org/manual/UserManual.html#ovf-import-appliance`
Démarrer la machine virtuelle :
`https://www.virtualbox.org/manual/UserManual.html#intro-starting`
Tout le TP sera réalisé dans la machine virtuelle.

1.2 Environnement

La machine virtuelle contient un interpréteur `Python3` et un ensemble de bibliothèques utiles à la simulation d'Enigma. Des scripts Python permettent d'instrumenter ces bibliothèques pour réaliser le chiffrement et le déchiffrement de messages. Ces scripts peuvent être modifiés à l'aide d'environnements de développement ou d'éditeurs de texte tels que `nano`, `vim`, `geany`, etc.

L'ensemble de ces scripts est présent dans la machine virtuelle. Pour procéder à leur lancement, ouvrir un terminal et se placer dans le dossier `tp-enigma` :

```
debian@myhostname:~$ cd tp-enigma/
debian@myhostname:~/tp-enigma/$ ls
```

Le mot de passe pour le compte utilisateur est : `debian`.

1.3 Exécuter un script Python

Lorsque vous écrivez un script Python `exemple.py`, vous pouvez l'exécuter en le passant comme argument à l'interpréteur `python3`. Par exemple :

```
debian@myhostname:~/tp-enigma/$ python3 exemple.py
```

Une autre possibilité est d'ajouter un *shebang* [1] à votre script. Le *shebang*, représenté par `#!`, est un en-tête de fichier texte qui indique au système d'exploitation (de type Unix) que ce fichier n'est pas un fichier binaire mais un script (ensemble de commandes) ; sur la même ligne est précisé l'interpréteur permettant d'exécuter ce script.

Ensuite, vous pourrez rendre votre script exécutable et l'exécuter comme un programme à part entière. Par exemple :

```
debian@myhostname:~/tp-enigma$ cat exemple.py
#!/usr/bin/python3
print("Hello world")

debian@myhostname:~/tp-enigma$ chmod +x exemple.py
debian@myhostname:~/tp-enigma$ ./exemple.py
```

1. `https://fr.wikipedia.org/wiki/Shebang`

2 Objectifs

Le premier objectif de ces séances est de comprendre le fonctionnement de la machine Enigma. Cette tâche est assurée par une cryptanalyse et une ingénierie inverse de la bibliothèque Python `enigma`.

Le second objectif est l'attaque de la machine à l'aide des Bombes Cryptologiques : respectivement la *bomba kryptologiczna* de Marian Rejewski et la Bombe d'Alan Turing et Gordon Welchman.

C'est parti !

3 Introduction

Le **chiffrement** est un procédé de cryptographie grâce auquel on souhaite rendre la compréhension d'un document impossible à toute personne qui n'a pas la clef de (dé)chiffrement. Ce principe est généralement lié au principe d'accès conditionnel.

Bien que le chiffrement puisse rendre secret le sens d'un document, d'autres techniques cryptographiques sont nécessaires pour communiquer de façon sûre. Pour vérifier l'intégrité ou l'authenticité d'un document, on utilise respectivement un code d'authentification de message ou une signature numérique. [...] La sécurité d'un système de chiffrement doit reposer sur le secret de la clef de chiffrement et non sur celui de l'algorithme. Le principe de Kerckhoffs suppose en effet que l'ennemi (ou la personne qui veut déchiffrer le message codé) connaisse l'algorithme utilisé.

Source : `https://fr.wikipedia.org/wiki/Chiffrement`

Enigma est une machine électromécanique portative servant au chiffrement et au déchiffrement de l'information. Elle fut inventée par l'allemand Arthur Scherbius, reprenant un brevet du Néerlandais Hugo Koch, datant de 1919. Enigma fut utilisée principalement par l'Allemagne nazie (*Die Chiffriermaschine Enigma*) pendant la Seconde Guerre mondiale.

Le terme "Enigma" désigne en fait toute une famille de machines, car il en a existé de nombreuses et subtiles variantes, commercialisées en Europe et dans le reste du monde à partir de 1923. Elle fut aussi adoptée par les services militaires et diplomatiques de nombreuses nations.

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

Une **Bombe Cryptologique** (dérivé du polonais *bomba*, "bombe") est une machine à usage spécifique conçue vers octobre 1938 par le cryptologue polonais Marian Rejewski du *Biuro Szyfrów* (Bureau du chiffre du renseignement militaire polonais) pour déchiffrer les codes allemands de la machine Enigma. L'engin est baptisé de ce nom parce qu'il fait tic-tac lorsqu'il fonctionne.

Une machine inspirée de celle-ci, nommée Bombe, est utilisée par les cryptologues britanniques et américains au cours de la Seconde Guerre mondiale dans le même but.

Source : `https://fr.wikipedia.org/wiki/Bombe_cryptologique`

4 Fonctionnement d'Enigma

Dans cette section, nous allons réaliser une cryptanalyse et ingénierie inverse de la machine Enigma à partir de la bibliothèque Python nommée `enigma`. Nous allons reconstruire la machine à partir d'une simulation de pièces détachées. La figure 1 montre deux photographies d'une machine Enigma : capot fermé, et capot ouvert.

Enigma chiffre les informations en faisant passer un courant électrique à travers une série de composants. Ce courant est transmis **en pressant une lettre sur le clavier** ; il traverse un réseau complexe de fils puis **allume une lampe** qui indique la lettre chiffrée.

Le premier composant du réseau est une série de roues adjacentes, appelées "**rotors**", qui contiennent les fils électriques utilisés pour chiffrer le message. Les rotors tournent, modifiant la configuration complexe du réseau chaque fois qu'une lettre est tapée. Enigma utilise habituellement une autre roue, appelée "**réflecteur**", et un composant appelé "**tableau de connexions**", ce qui permet de rendre plus complexe encore le processus de chiffrement.

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

Notons que le clavier des machines Enigma ne possédaient pas de touche "espace". Les messages écrits avec Enigma ne comportaient donc aucun espace. Dans ce TP, nous travaillons donc avec un alphabet de 26 lettres, composé seulement des majuscules de l'alphabet latin.

FIGURE 1 – Clavier d'Enigma, capot fermé (gauche) et ouvert (droite)

4.1 Les rotors

Lorsqu'une lettre est tapée au clavier de la machine Enigma, un courant électrique traverse la machine pour allumer la lampe qui indique la lettre chiffrée. Le chemin de ce courant est dépendant de la position des rotors de la machine. La bibliothèque Python `enigma` fournit un modèle des rotors disponibles dans les machines Enigma. Nous allons étudier ces modèles et analyser la méthode de chiffrement employée par Enigma.

Question 1 Le script `tp-enigma/01/rotor.py` contient un squelette prêt pour analyser le fonctionnement des rotors. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/01
debian@myhostname:~/tp-enigma/01$ ./rotor.py
```

Dans la suite de cette section, nous allons modifier ce script pour effectuer notre cryptanalyse.

Question 2 La méthode `enigma.rotors.factory.create_rotor()` permet d'instancier un objet de type `rotor` que nous pouvons étudier. En particulier, cet objet possède un attribut `entry_map` qui donne la table des substitutions effectuées par le rotor.
Instancier le rotor `"III"` et observer la table des substitutions qu'il effectue.

```
theRotor = enigma.rotors.factory.create_rotor("III")
print("SUBSTITUTIONS: " + str(theRotor.entry_map))
```

Lorsqu'une lettre de l'alphabet traverse un rotor, un décalage est appliqué sur cette lettre en fonction de son index dans la liste. Egalement, les rotors ont la possibilité de tourner. Dans ce cas, l'index de la lettre dans la liste est modifié en fonction de la position du rotor.

Question 3 Pour simplifier notre étude, nous allons commencer par observer le comportement du rotor lorsque celui-ci n'a pas tourné.
Déclarer une variable qui indiquera la position du rotor.

```
theRotorPos = 0
print("ROTOR POSITION: " + str(theRotorPos))
```

Question 4 Nous allons désormais tenter de définir quel décalage est appliqué aux lettres par le rotor.
Pour la lettre `A`, déterminer son index dans l'alphabet et l'index de la lettre qui va la substituer. Afficher la lettre `A` et son image fournie par le rotor.

```
theLetterIn  = "A"
theIndexIn   = theAlphabet.find(theLetterIn)
theIndexOut  = theRotor.entry_map[theIndexIn]
theLetterOut = theAlphabet[theIndexOut]
print(theLetterIn + " -> " + theLetterOut)
```

Question 5 Réaliser la même substitution à l'aide du rotor, cette fois-ci avec la lettre `B`.
Expliquer le comportement observé (il est possible de tester avec d'autres lettres). Quel chiffrement réalise le rotor lorsque celui-ci ne tourne pas ?

```
theLetterIn  = "B"
theIndexIn   = theAlphabet.find(theLetterIn)
theIndexOut  = theRotor.entry_map[theIndexIn]
theLetterOut = theAlphabet[theIndexOut]
print(theLetterIn + " -> " + theLetterOut)
```

4.1.1 Position du rotor

Question 6 Lorsque le rotor ne tourne pas, il réalise un chiffrement par substitution monoalphabétique. Nous devons désormais définir l'impact de la rotation du rotor sur cette substitution. Nous allons de nouveau, pour la lettre `A`, déterminer son index dans l'alphabet et l'index de la lettre qui va la substituer.
Modifier la valeur de la position du rotor à 1 et définir sur quelle connexion la lettre `A` va être réalisée. Ajouter l'impact de la position sur l'entrée du rotor.

```
theRotorPos = 1
print("ROTOR POSITION: " + str(theRotorPos))

theLetterIn  = "A"
theIndexIn   = theAlphabet.find(theLetterIn)
theInputPin  = (theIndexIn + theRotorPos) % len(theRotor.entry_map)
```

Question 7 Nous savons sur quelle entrée du rotor la lettre `A` va être connectée. Appliquer la substitution réalisée par le rotor pour obtenir l'index de la sortie de celui-ci.

```
theOutputPin = theRotor.entry_map[theInputPin]
print(theOutputPin)
```

La question est désormais de savoir par quelle lettre la lettre `A` va être substituée lorsque le rotor n'est pas en position 0. Pour le déterminer, nous pouvons nous fier à la figure 2, qui représente les 5 premières substitutions du rotor 3 pour les positions 0, 1 et 2.

Dans les questions précédentes, nous avons vu qu'en position 0, les lettres `A` et `B` étaient respectivement substituées par `B` (connexion à la sortie 1) et `D` (connexions à la sortie 3).

Question 8 En position 1, la substitution de la lettre `A` provoque une connexion vers la sortie 3.
Déduire l'index de la lettre résultante en fonction de la connexion de sortie et de la position du rotor.
Appliquer la substitution et vérifier que la lettre `A` est substituée par un `C` (comme sur la figure 2).

```
theIndexOut  = (theOutputPin - theRotorPos) % len(theRotor.entry_map)
theLetterOut = theAlphabet[theIndexOut]
print(theLetterIn  + " (" + str(theInputPin)  + ") -> " + \
      theLetterOut + " (" + str(theOutputPin) + ")")
```

Question 9 Nous avons donc déterminé un algorithme pour simuler le comportement des rotors.
Effectuer des vérifications pour plusieurs lettres et plusieurs positions : s'assurer que nous avons le même comportement que celui décrit sur la figure 2.

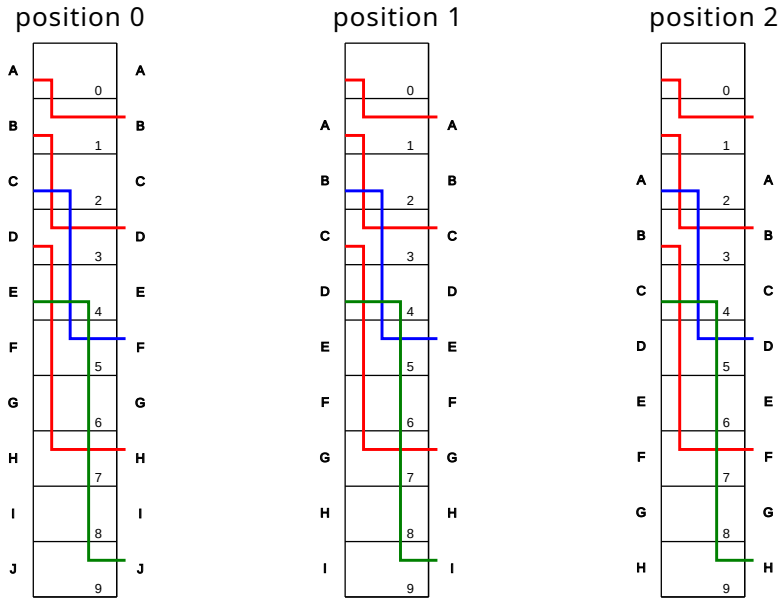

FIGURE 2 – Substitutions du rotor 3 pour les positions 0, 1 et 2

```
# verifications:
for theRotorPos in [1, 2]:
  print("")
  print("ROTOR POSITION: " + str(theRotorPos))
  for theLetterIn in "ABC":
    theIndexIn   = theAlphabet.find(theLetterIn)
    theInputPin  = (theIndexIn + theRotorPos) % len(theRotor.entry_map)
    theOutputPin = theRotor.entry_map[theInputPin]
    theIndexOut  = (theOutputPin - theRotorPos) % len(theRotor.entry_map)
    theLetterOut = theAlphabet[theIndexOut]
    print(theLetterIn  + " (" + str(theInputPin)  + ") -> " + \
          theLetterOut + " (" + str(theOutputPin) + ")")
```

Les rotors sont des roues adjacentes qui ont la possibilité de changer de position. Un changement de position implique une modification des substitutions pour les lettres qui les traversent.

Question 10 Dans la machine Enigma, la position des rotors est modifiée à chaque fois qu'une lettre est tapée. Observer le chiffrement par le rotor `"III"` de plusieurs lettres, par exemple `ABCDE`, lorsque la position du rotor est incrémentée à chaque lettre.

```
theRotorPos = 0
for theLetterIn in "ABCDE":
  theRotorPos += 1 # here is the rotation
  theIndexIn   = theAlphabet.find(theLetterIn)
  theInputPin  = (theIndexIn + theRotorPos) % len(theRotor.entry_map)
  theOutputPin = theRotor.entry_map[theInputPin]
  theIndexOut  = (theOutputPin - theRotorPos) % len(theRotor.entry_map)
  theLetterOut = theAlphabet[theIndexOut]
  print(str(theRotorPos) + ": " + \
        theLetterIn  + " (" + str(theInputPin)  + ") -> " + \
        theLetterOut + " (" + str(theOutputPin) + ")")
```

4.1.2 Sens de traversée du rotor

Question 11 Nous avons observé la table des substitutions dans le rotor "III". Il s'avère que chaque rotor peut être traversé dans les deux sens. La table des substitutions est alors différente : elle permet d'obtenir la transformation inverse. L'objet de type rotor obtenu par la méthode enigma.rotors.factory.create_rotor() possède un attribut exit_map qui donne la table des substitutions effectuées par le rotor dans l'autre sens : de la sortie vers l'entrée.
Observer la table des substitutions que le rotor "III" effectue lorsque celui-ci est traversé dans l'autre sens.
Pourquoi la valeur à l'index 1 est elle égale à 0 (la réponse est valable pour toutes les valeurs à tous les indexes) ?

```
print("SUBSTITUTIONS: " + str(theRotor.exit_map))
```

A l'index 1, la valeur est égale à 0 car dans la table de substitution dans le premier sens, nous avions la valeur 1 à l'index 0. A chaque index, la valeur donnée indique la substitution dans l'autre sens par rapport à la première table.
Nous pouvons nous aider de la figure 2 pour déduire certaines des premières valeurs. Par exemple, à l'index 3, nous avons la valeur 1 (D est substitué par B en position 0) ; à l'index 5, nous avons la valeur 2 (F est substitué par C en position 0).

Question 12 Placer le rotor dans la position 0. Effectuer un chiffrement où la position est incrémentée à chaque lettre et où le rotor est traversé en sens inverse.
Chiffrer le résultat obtenu précédemment à partir des lettres "ABCDE". Que peut-on observer ?
Que se passe-t-il si la position de départ n'est pas la même ?

```
theRotorPos = 0
for theLetterOut in "CFILO":
  theRotorPos += 1 # here is the rotation
  theIndexOut  = theAlphabet.find(theLetterOut)
  theOutputPin = (theIndexOut + theRotorPos) % len(theRotor.entry_map)
  theInputPin  = theRotor.exit_map[theOutputPin]
  theIndexIn   = (theInputPin - theRotorPos) % len(theRotor.entry_map)
  theLetterIn  = theAlphabet[theIndexIn]
  print(theLetterOut + " (" + str(theOutputPin) + ") <- " + \
        theLetterIn  + " (" + str(theInputPin)  + ")")
```

Si l'on a positionné le rotor de la même manière que lorsque nous avons chiffré ABCDE dans un sens, alors chiffrer le résultat dans l'autre sens réalise en réalité un déchiffrement.

Si l'on a positionné le rotor de manière différente, alors nous effectuons un second chiffrement. Il faudra donc traverser deux fois le rotor, dans le sens approprié et avec une position initiale correcte, pour déchiffrer.

La bibliothèque Python `enigma`

Jusqu'à présent, nous avons implémenté nous-même le fonctionnement des rotors à partir de leurs tables de substitution. La bibliothèque Python enigma fournit toutes les fonctions pour effectuer les substitutions et ne pas avoir besoin de ré-implémenter comme nous l'avons fait.

En particulier, les méthodes signal_in() et signal_out() fournissent les indexes des lettres de l'alphabet après substitution (dans les deux sens pour la traversée du rotor). De plus, la méthode rotate() permet de modifier la position du rotor et de simuler une rotation sans avoir à mémoriser la position courante.

La méthode enigma.rotors.factory.create_rotor(), qui permet d'instancier un objet de type rotor, a pour effet de remettre à zéro la position.

Question 13 Utiliser ces fonctionnalités de la bibliothèque Python enigma pour chiffrer et déchiffrer un message.

```
print("")
theRotor = enigma.rotors.factory.create_rotor("III")
for theLetterIn in "ABCDE":
  theRotor.rotate()
  theLetterOut = theAlphabet[theRotor.signal_in(theAlphabet.find(theLetterIn))]
  print(theLetterIn  + " -> " + theLetterOut)

print("")
theRotor = enigma.rotors.factory.create_rotor("III") # reset the position
```

```
for theLetterOut in "CFILO":
  theRotor.rotate()
  theLetterIn = theAlphabet[theRotor.signal_out(theAlphabet.find(theLetterOut))]
  print(theLetterOut  + " <- " + theLetterIn)
```

A partir de maintenant, nous allons utiliser les fonctionnalités de cette bibliothèque pour étudier le comportement d'Enigma.

Question 14 Nous souhaitons également déterminer l'algorithme de chiffrement réalisé par le rotor.
Chiffrer un message composé de 100 fois le même caractère avec le rotor "III". Que peut-on observer ?

```
theRotor = enigma.rotors.factory.create_rotor("III")
for theLetterIn in ("A" * 100):
  theRotor.rotate()
  theLetterOut = theAlphabet[theRotor.signal_in(theAlphabet.find(theLetterIn))]
  print(theLetterOut, end="")
print("")
```

Question 15 Reproduire l'expérience avec un autre caractère. Que peut-on conclure sur le chiffrement réalisé par le rotor et la taille de la clef ?

```
theRotor = enigma.rotors.factory.create_rotor("III")
for theLetterIn in ("B" * 100):
  theRotor.rotate()
  theLetterOut = theAlphabet[theRotor.signal_in(theAlphabet.find(theLetterIn))]
  print(theLetterOut, end="")
print("")
```

Chaque rotor effectue donc 26 chiffrements par clef : 26 chiffres de Vigenère. La clef a une longueur telle que celle-ci peut prendre 26 valeurs possibles. Chaque valeur de la clef assure un décalage dans un tableau de substitutions, modulo 26. Si le message clair est composé du même caractère répété en boucle, alors le message chiffré se répète tous les 26 caractères. Changer de caractère répété en boucle implique utiliser une clef différente (obtenue par décalage de la première clef). Si le message clair est composé de caractères différents, alors il y a de fortes chances que nous n'observons pas de répétition.

Dans la machine Enigma, plusieurs rotors sont positionnées à la suite (généralement, 3 rotors). Chaque rotor réalise une substitution. De ce fait, il est nécessaire de connaitre les positions de tous les rotors pour déterminer la substitution effective.

La figure 3 donne une représentation simplifiée des trois rotors connectés au clavier. Sur cet exemple, une lettre A est substituée par un D, puis par un F et de nouveau par un D. De même, une lettre B est substituée par un A, puis par un E et par un F. Etc.

Question 16 La machine Enigma que nous étudions possède 3 rotors de 26 caractères. Combien de possibilités y a-t-il pour la position initiale des 3 rotors.

```
print(26 * 26 * 26)
```

Question 17 Instancier les rotors I, II et III en position 0. Visualiser la table de substitution de chacun des rotors.

```
theRotor1 = enigma.rotors.factory.create_rotor("I")
theRotor2 = enigma.rotors.factory.create_rotor("II")
theRotor3 = enigma.rotors.factory.create_rotor("III")
#
print("I   : " + str(theRotor1.entry_map))
print("II  : " + str(theRotor2.entry_map))
print("III: " + str(theRotor3.entry_map))
```

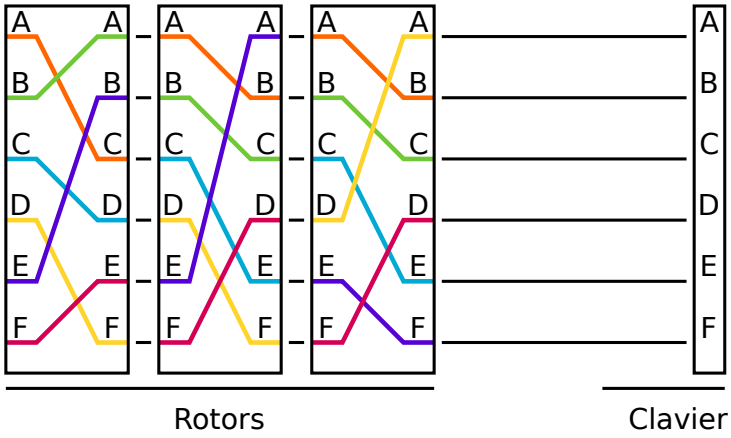

FIGURE 3 – Connexions entre les rotors et le clavier

Question 18 Pour les positions 1, 2 et 3 du rotor `III`, visualiser la table de substitution de l'assemblage des trois rotors. Nous considérons que les rotors sont traversés dans l'ordre suivant : `III`, `II` puis `I`.

```
for thePosition in range(0, 3):
  theRotor3.rotate()
  theSub = list()
  for i in range(0, 26):
    theSub.append(theRotor1.signal_in(theRotor2.signal_in(theRotor3.signal_in(i))))
  #
  print("x3 (" + str(thePosition) + "): " + str(theSub))
```

Conclure sur l'entropie obtenue à partir de l'assemblage des trois rotors lors d'une seule variation de position.

4.2 Le réflecteur

Lorsqu'une lettre est tapée au clavier de la machine Enigma, un courant électrique traverse plusieurs rotors, semblables au rotor "`III`" que nous venons d'étudier. Cet assemblage de plusieurs rotors permet d'obtenir une table de substitution variant, à chaque changement de position, avec une entropie similaire à celle d'un rotor seul. La différence est que la taille de la clef pour les différents chiffres de Vigenère est alors de $26 \times 26 \times 26$.

Une fois les rotors traversés, le courant traverse également un réflecteur. Le réflecteur réalise la même fonction qu'un rotor : il est équivalent à un tableau de substitution. La différence avec les rotors et que celui ne tourne pas et qu'il est construit de telle sorte que le courant qui le traverse, après substitution, soit réfléchi vers les rotors. Le courant va alors traverser de nouveau les rotors en sens inverse. La figure 4 donne une représentation simplifiée du réflecteur connecté aux trois rotors.

L'objectif du réflecteur est qu'une fois tous les rotors traversés, une nouvelle substitution est appliquée avant de les traverser de nouveau en sens inverse. Le nombre de substitutions appliquées à chaque lettre entrée au clavier, pour une machine Enigma à n rotors, est donc de $2 \times n + 1$.

Au niveau du clavier, une ampoule est positionnée à chaque lettre, de sorte à ce que la substitution allume celle qui correspond à la lettre résultante. La figure 5 illustre le cas où la lettre `A` est pressée au clavier : celle-ci est alors substituée par un `F`.

Question 19 Le script `tp-enigma/02/reflector.py` contient un squelette prêt pour analyser le fonctionnement du réflecteur. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/02
debian@myhostname:~/tp-enigma/02$ ./reflector.py
```

Dans la suite de cette section, nous allons modifier ce script pour effectuer notre cryptanalyse.

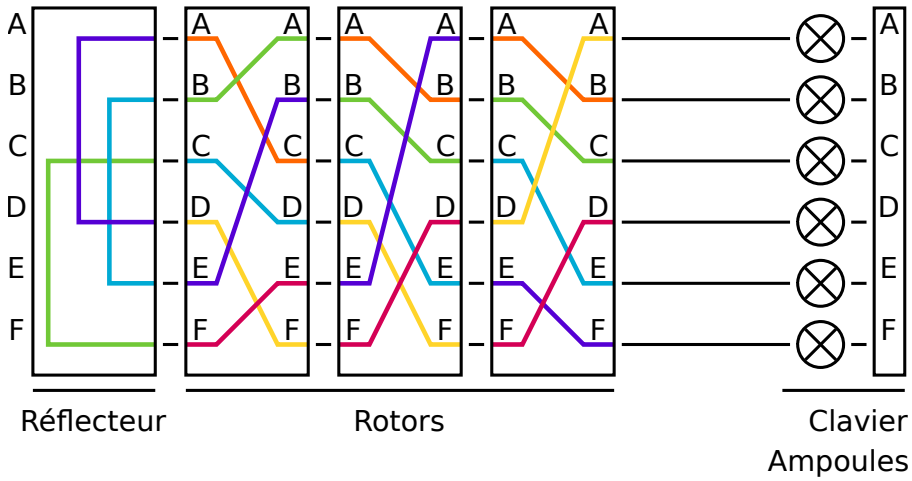

FIGURE 4 – Connexions entre les rotors et le réflecteur

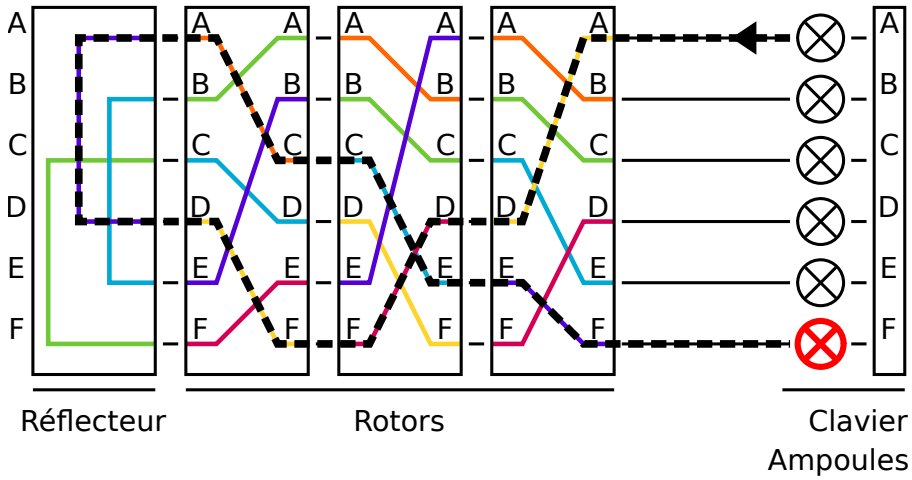

FIGURE 5 – Ampoule allumée après substitution de la lettre A

Question 20 La méthode `enigma.rotors.factory.create_reflector()` permet d'instancier un objet de type `rotor` que nous pouvons étudier (dans cette bibliothèque, le réflecteur et les rotors sont définis avec des objets du même type). Ici aussi, cet objet possède un attribut `entry_map` qui donne la table des substitutions effectuées par le réflecteur.
Instancier le réflecteur "B" et observer la table des substitutions qu'il effectue.

```
theReflect = enigma.rotors.factory.create_reflector("B")
print("SUBSTITUTIONS: " + str(theReflect.entry_map))
```

Question 21 Pour chacune des lettres de l'alphabet, calculer son index, la substituer avec le réflecteur et afficher la lettre résultante.

```
for theLetterIn in theAlphabet:
  theIndexIn   = theAlphabet.find(theLetterIn)
  theIndexOut  = theReflect.entry_map[theIndexIn]
  theLetterOut = theAlphabet[theIndexOut]
  print(theLetterIn  + " (" + str(theIndexIn)  + ") -> " + \
        theLetterOut + " (" + str(theIndexOut) + ")")
```

Le réflecteur ne tourne pas. En conséquence, les substitutions que celui-ci réalise seront toujours identiques à celles que nous venons d'afficher. Par contre, l'ajout du réflecteur va provoquer une nouvelle traversée des rotors qui composent la

machine Enigma. Nous allons donc désormais observer le comportement d'un assemblage de trois rotors et du réflecteur.

Question 22 Instancier les rotors `I`, `II` et `III` en position 0 ainsi que le réflecteur `B`.

```
theRotor1  = enigma.rotors.factory.create_rotor("I")
theRotor2  = enigma.rotors.factory.create_rotor("II")
theRotor3  = enigma.rotors.factory.create_rotor("III")
theReflect = enigma.rotors.factory.create_reflector("B")
```

Question 23 Nous allons simuler le comportement de la machine Enigma lorsque la lettre `A` est tapée au clavier. Calculer l'index de la lettre `A` dans l'alphabet.

```
theLetterIn = "A"
theIndex = theAlphabet.find(theLetterIn)
print(theLetterIn + " (" + str(theIndex) + ")")
```

Question 24 Appliquer l'ensemble des sept substitutions réalisées par l'assemblage des trois rotors et du réflecteur. Afficher le résultat de chaque substitution.
Note : le clavier est positionné à droite des trois rotors, nous allons donc d'abord traverser le rotor `III`, puis le rotor `II` et enfin le rotor `I`.

```
theIndex = theRotor3.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
theIndex = theRotor2.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
theIndex = theRotor1.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
theIndex = theReflect.signal_in(theIndex) ; print(" [  (" + str(theIndex) + ")")
theIndex = theRotor1.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
theIndex = theRotor2.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
theIndex = theRotor3.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
```

Question 25 Finalement, déterminer quel est le résultat d'une substitution de la lettre `A` par la machine Enigma avec les trois rotors en position 0.

```
theLetterOut = theAlphabet[theIndex]
print("= " + theLetterOut)
```

Question 26 Sans modifier la position des rotors, effectuer une traversée identique, cette fois-ci pour substituer la lettre `U`.
Que peut-on observer ? La machine Enigma réalise-t-elle un chiffrement symétrique ou asymétrique ?

```
theLetterIn = "U"
theIndex = theAlphabet.find(theLetterIn)
print(theLetterIn + " (" + str(theIndex) + ")")
#
theIndex = theRotor3.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
theIndex = theRotor2.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
theIndex = theRotor1.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
theIndex = theReflect.signal_in(theIndex) ; print(" [  (" + str(theIndex) + ")")
theIndex = theRotor1.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
theIndex = theRotor2.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
theIndex = theRotor3.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
#
theLetterOut = theAlphabet[theIndex]
print("= " + theLetterOut)
```

Pour une position donnée de trois rotors, toute lettre entrée au clavier est substituée 7 fois. Le résultat de cette substitution peut être transmis de nouveau à la machine Enigma, dans le cas où les rotors sont dans la même position, pour obtenir la lettre d'origine. Le chiffrement réalisé par Enigma est donc **symétrique** : l'opération inverse est utilisée pour déchiffrer un message.

Considérons le schéma représenté sur la figure 5, où une lettre `A` est substituée par un `F`. Pour la même position des rotors, déchiffrer une lettre `F` permet d'obtenir un `A` comme représenté sur la figure 6.

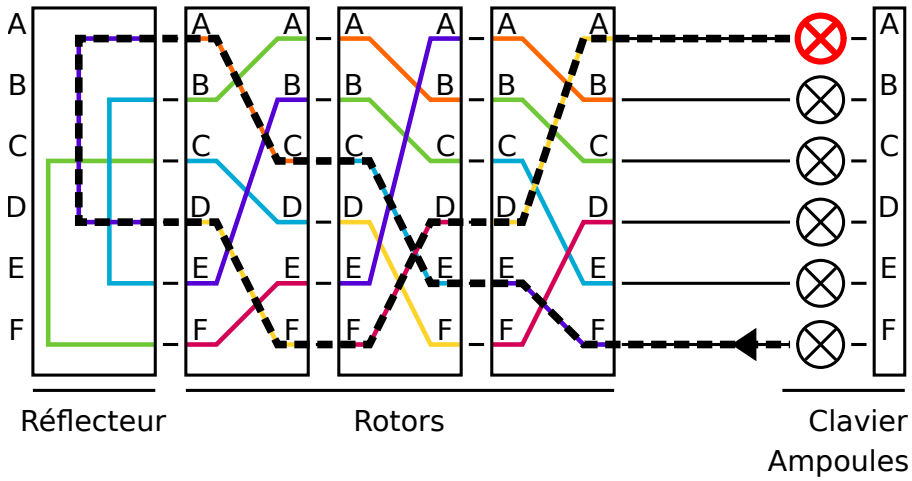

FIGURE 6 – Ampoule allumée après substitution de la lettre F

Rappelons également que les rotors ont la possibilité de tourner et de changer de position. La position des rotors est modifiée à chaque frappe au clavier. Comme le chiffrement est symétrique, que l'on tape le message clair ou le message chiffré, si l'on part de la même position de départ pour les trois rotors, alors le changement de position sera identique. Chaque frappe au clavier entraîne une rotation de la même manière.

Question 27 Réaliser une rotation du rotor `III` et exécuter de nouveau le code précédent (pour le chiffrement de la lettre `A` et le déchiffrement du résultat).
Vérifier que le chiffrement est toujours symétrique.

```
theRotor3.rotate()
```

4.2.1 Modèle mathématique d'Enigma

Dans le cas que nous venons de réaliser, les rotors sont positionnés de telle sorte que le premier rotor traversé soit le rotor III, puis le rotor II et finalement le rotor I. Pour toute lettre α tapée au clavier,

— soit n_1 la position du rotor I, n_2 la position du rotor II et n_3 la position du rotor III,

— soit $R_1(n_1,\alpha)$ la substitution réalisée par le rotor I, $R_2(n_2,\alpha)$ la substitution réalisée par le rotor II, $R_3(n_3,\alpha)$ la substitution réalisée par le rotor III et $R_B(\alpha)$ la substitution réalisée par le réflecteur B,

Alors la substitution $E(n_1,n_2,n_3,\alpha)$ appliquée par Enigma est représentée par la transformation mathématique suivante :

$$\begin{aligned} &\forall\, n_1,n_2,n_3 \in [0:26[,\\ &\quad \forall\, \alpha \in [A-Z],\\ &\qquad E(n_1,n_2,n_3,\alpha) = R_3^{-1}(n_3,R_2^{-1}(n_2,R_1^{-1}(n_1,R_B(R_1(n_1,R_2(n_2,R_3(n_3,\alpha))))))) \end{aligned} \tag{1}$$

L'ajout du réflecteur implique qu'un changement de position d'un rotor modifie deux substitutions : une fois dans un sens et une fois dans l'autre sens. En effet, si nous incrémentons la valeur de n_3, alors nous changeons le résultat des transformations $R_3(n_3,x)$ mais aussi de $R_3^{-1}(n_3,x)$.

Comme Engima est symétrique, alors appliquer le résultat d'une substitution sur une nouvelle substitution avec les mêmes valeurs de n_1,n_2,n_3 permet d'obtenir la lettre d'origine. De ce fait, pour deux lettres de l'alphabet α et β quelconques et toute valeur de n_1,n_2,n_3 :

$$\begin{aligned} &\forall\, n_1,n_2,n_3 \in [0:26[,\\ &\quad \forall\, \alpha,\beta \in [A-Z],\\ &\qquad E(n_1,n_2,n_3,\alpha) = \beta \iff E(n_1,n_2,n_3,\beta) = \alpha \end{aligned} \tag{2}$$

4.3 Mécanisme d'entraînement

Afin d'éviter de réaliser un simple chiffrement par substitution monoalphabétique, chaque touche pressée sur le clavier provoque l'entraînement d'au moins un rotor, changeant ainsi la substitution alphabétique utilisée. Cela assure que la substitution est différente pour chaque nouvelle frappe sur le clavier, il deux façon équivalente de visualiser ce comportement :

— Enigma produit un chiffrement par substitution polyalphabétique différent pour chaque lettre de l'alphabet tapée au clavier.
— Enigma produit un chiffrement par substitution monoalphabétique différent après chaque rotation (à chaque lettre de l'alphabet tapée au clavier).

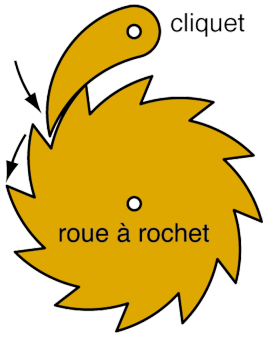

FIGURE 7 – Fonctionnement d'une roue à rochets

C'est un mécanisme à base de roue à rochets qui était le plus fréquemment utilisé [La figure 7 représente le fonctionnement d'une roue à rochets.]. A cette roue à 26 dents était associé un fin anneau métallique à encoches, solidaire du rotor. Chaque frappe de touche, via les cliquets, poussait sur la roue de chaque rotor pour la faire avancer d'un cran. Ce sont les anneaux métalliques qui permettaient ou non l'avancée des rotors. **Pour qu'un rotor avance, il fallait que l'anneau du rotor [précédent] présente une encoche alignée avec son cliquet**. Le premier rotor n'étant pas bloqué par un anneau, il avançait d'un cran à chaque frappe. L'avancée des autres rotors dépendait du nombre d'encoches : une seule pour les rotors de type `I` à `V`, deux pour les rotors de type `VI` à `VIII`.

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

Question 28 Le script `tp-enigma/03/rotations.py` contient un squelette prêt pour réaliser une analyse du mécanisme d'entraînement. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/03
debian@myhostname:~/tp-enigma/03$ ./rotations.py
```

Dans la suite de cette section, nous allons modifier ce script pour réaliser notre étude.

Avec le fonctionnement décrit précédemment, la rotation d'un rotor va entraîner le rotor suivant une ou deux fois tous les 26 caractères. Dans la machine Enigma, les différents rotors n'ont pas leur encoche positionnée au niveau de la lettre `A`, donc l'entraînement du rotor suivant ne se fait pas lorsque l'alphabet a été parcouru de `A` à `Z` (mais potentiellement lorsqu'il a été parcouru de `E` à `D`).

Nous allons désormais analyser les rotors afin de déterminer l'entraînement de ceux-ci à chaque frappe de touche au clavier. La première étape consiste à déterminer l'emplacement de l'encoche sur les rotors que nous utilisons (ou des encoches pour les rotors de type `VI` à `VIII`) . Nous utilisons les rotors `I` à `III`, donc une seule encoche est présente.

Question 29 La bibliothèque Python `enigma` fournit l'emplacement des encoches sur chaque rotor. Cet emplacement est accessible au travers de l'attribut `step_set`.
Afficher l'emplacement des encoches sur les rotors `I` à `III`.

```
print("Rotor III step at: " + str(theRotor3.step_set))
print("Rotor II  step at: " + str(theRotor2.step_set))
print("Rotor I   step at: " + str(theRotor1.step_set))
```

Question 30 Nous allons désormais implémenter une rotation des rotors dans des itérations :
- le rotor `I` doit tourner lorsque le rotor `II` présente une encoche alignée avec son cliquet ;
- le rotor `II` doit tourner lorsque le rotor `III` présente une encoche alignée avec son cliquet ;
- le rotor `III` doit tourner à chaque itération.

Réaliser 50 itérations et appliquer les rotations des rotors selon ces conditions. A chaque itération, afficher les positions des trois rotors. La position d'un rotor est accessible via l'attribut `pos`.

```
for i in range(0, 50):
  if ( theRotor2.pos == theAlphabet.find("E") ):
    theRotor1.rotate()
  if ( theRotor3.pos == theAlphabet.find("V") ):
    theRotor2.rotate()
  theRotor3.rotate()
  #
  print("Positions [I, II, III]: " + str([theRotor1.pos, theRotor2.pos, theRotor3.pos]))
```

A l'aide des encoches, le rotor `II` tourne bien toutes les 26 rotations du rotor `III`. A priori, le rotor `I` tourne également toutes les 26 rotations du rotor `II`. Cependant, cette implémentation pose un problème lorsque le rotor `II` est à la position de la lettre `E`.

Question 31 Placer les positions des rotors `I`, `II` et `III` pour correspondre aux indexes des lettres `A`, `D` et `A`. Effectuer de nouveau 50 itérations et observer les rotations des trois rotors.
Que peut-on remarquer ?

```
theRotor1.pos = theAlphabet.find("A")
theRotor2.pos = theAlphabet.find("D")
theRotor3.pos = theAlphabet.find("A")
print("")
for i in range(0, 50):
  if ( theRotor2.pos == theAlphabet.find("E") ):
    theRotor1.rotate()
  if ( theRotor3.pos == theAlphabet.find("V") ):
    theRotor2.rotate()
  theRotor3.rotate()
  #
  print("Positions [I, II, III]: " + str([theRotor1.pos, theRotor2.pos, theRotor3.pos]))
```

Lorsque le rotor `II` est à la position de la lettre `E`, le rotor `I` tourne en même temps que le rotor `III`. Ce comportement n'a pas satisfait les concepteurs de la machine Enigma. Un mécanisme supplémentaire a donc été ajouté pour empêcher le rotor `I` de tourner à chaque touche tapée : celui de la *double avancée*.

4.3.1 Double avancée

Le mécanisme incluait également "**une double avancée**". L'alignement du cliquet avec une encoche lui permettait de pousser sur la roue à rochet de son rotor, mais lui permettait aussi de pousser sur l'encoche de l'anneau solidaire du précédent rotor. Ce sont donc à la fois le rotor du cliquet et le rotor de l'anneau qui avançaient d'un cran. Pour une machine à trois rotors, seul le second était affecté par cette double avancée, le premier rotor avançant à chaque frappe de toute façon. [...]

Avec trois rotors et une seule encoche pour les rotors I et II, une Enigma a une période de $26 \times 25 \times 26 = 16900$ (et non pas $26 \times 26 \times 26$ à cause de la double avancée). Historiquement, les messages étaient limités à quelques centaines de lettres évitant toute chance de répétition de combinaisons des rotors, empêchant par là même les cryptanalystes d'accéder à des indices importants.

La frappe d'une touche provoquait d'abord une avancée sur les rotors et ensuite seulement une connexion électrique.

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

Concrètement, avec ce mécanisme, cela signifie que le rotor `II` pouvait effectuer une rotation sous deux conditions :
- soit le rotor `III` présentait une encoche au niveau de son cliquet (ce que nous avons implémenté dans la question précédente) ;

— soit le rotor II lui-même présentait une encoche au niveau de son propre cliquet.

Question 32 Ajouter cette nouvelle condition à la rotation du rotor II. Placer de nouveau les positions des rotors I, II et III pour correspondre aux indexes des lettres A, D et A. Effectuer de nouveau 50 itérations et observer les rotations des trois rotors.
Vérifier que le rotor II effectue bien une double avancée et que le rotor I ne tourne plus en même temps que le III.

```
theRotor1.pos = theAlphabet.find("A")
theRotor2.pos = theAlphabet.find("D")
theRotor3.pos = theAlphabet.find("A")
print("")
for i in range(0, 50):
  if ( theRotor2.pos == theAlphabet.find("E") ):
    theRotor1.rotate()
  if ( theRotor3.pos == theAlphabet.find("V") or theRotor2.pos == theAlphabet.find("E") ):
    theRotor2.rotate()
  theRotor3.rotate()
  #
  print("Positions [I, II, III]: " + str([theRotor1.pos, theRotor2.pos, theRotor3.pos]))
```

Nous avons désormais terminé d'implémenter les rotations telles qu'elles étaient réalisées par la machine Enigma. Nous allons désormais réaliser un chiffrement afin de vérifier le comportement que nous venons d'implémenter.

Question 33 Placer les positions des rotors I, II et III pour correspondre aux indexes des lettres A, D et A. Réaliser le chiffrement d'une chaîne de caractères composée de 50 fois le caractère A et observer le résultat.
Pour rappel, la frappe d'une touche provoquait d'abord une avancée sur les rotors et ensuite la substitutions des lettres du message.

```
thePlaintext  = "A" * 50
print("PLAIN TEXT    = " + thePlaintext)

theRotor1.pos = theAlphabet.find("A")
theRotor2.pos = theAlphabet.find("D")
theRotor3.pos = theAlphabet.find("A")
theCipherText = ""
for theLetterIn in thePlaintext:
  # rotate before cipher:
  if ( theRotor2.pos == theAlphabet.find("E") ):
    theRotor1.rotate()
  if ( theRotor3.pos == theAlphabet.find("V") or theRotor2.pos == theAlphabet.find("E") ):
    theRotor2.rotate()
  theRotor3.rotate()
  # cipher:
  theIndex = theAlphabet.find(theLetterIn)
  theIndex = theRotor3.signal_in(theIndex)
  theIndex = theRotor2.signal_in(theIndex)
  theIndex = theRotor1.signal_in(theIndex)
  theIndex = theReflect.signal_in(theIndex)
  theIndex = theRotor1.signal_out(theIndex)
  theIndex = theRotor2.signal_out(theIndex)
  theIndex = theRotor3.signal_out(theIndex)
  theCipherText += theAlphabet[theIndex]
#
print("CIPHER TEXT   = " + theCipherText)
```

Nous allons désormais vérifier notre implémentation en tentant de chiffrer le même message clair avec une machine Enigma complète fournie par la bibliothèque Python `enigma`.

— La fonction `EnigmaMachine.from_key_sheet()` construit une machine Enigma avec des rotors et un réflecteur choisi. Nous pouvons sélectionner les rotors à l'aide du paramètre `rotors` et le réflecteur l'aide du paramètre `reflector`. Les valeurs sont des chaînes de caractères où chaque nom de rotor est séparé par un espace.

— La méthode `EnigmaMachine.set_display()` permet de choisir la position des rotors. La valeur est une chaîne de 3 caractères où chaque lettre définit la position du rotor (I puis II puis III).

— La méthode `EnigmaMachine.process_text()` réalise le chiffrement/déchiffrement d'une chaîne de caractères et retourne le résultat.

Question 34 Instancier une machine Enigma utilisant les rotors `I`, `II` et `III` ainsi que le réflecteur `B`. Placer les rotors sur la position `ADA` et chiffrer une chaîne de caractère composée de 50 fois le caractère `A`.
Vérifier que nous obtenons le même résultat qu'avec notre précédente implémentation.

```
theMachine    = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B")
theMachine.set_display("ADA")
#
thePlaintext  = "A" * 50
print("PLAIN TEXT    = " + thePlaintext)
theCipherText = theMachine.process_text(thePlaintext)
print("CIPHER TEXT   = " + theCipherText)
```

Question 35 A l'aide de la même machine Enigma, placer de nouveau les rotors sur la position `ADA` et déchiffrer le résultat du chiffrement précédent.
Vérifier que nous obtenons bien une chaîne de caractère composée de 50 fois le caractère `A`.

```
theMachine.set_display("ADA")
theDecipherText = theMachine.process_text(theCipherText)
print("DECIPHER TEXT = " + theDecipherText)
```

4.3.2 Position des rotors : anneaux et affichages

Un dernier point concernant le mécanisme d'entraînement est que l'encoche provoquant l'entraînement des rotors était positionnée sur un anneau. Si toutefois cet anneau était solidaire du rotor, la position de celui-ci était réglable. Il était donc possible de déplacer la ou les encoches sur les 26 crans de la roue à rochets.

Un déplacement de l'anneau a deux effets sur le fonctionnement de la machine Enigma :

— le premier est une modification de la position du rotor, exactement de la même manière que le réglage que nous avons fait précédemment ;

— le second est une modification de la position de l'encoche, qui provoquera la rotation du rotor suivant. La rotation du rotor suivant apparaitra donc plus tôt ou plus tard la première fois, mais toujours avec la même fréquence (tous les 26 caractères si le rotor ne possède qu'une seule encoche).

Dans cette section, nous considérons que la méthode `set_display()` modifie l'affichage du rotor (visible sur le capot de la machine Engima). Notre objectif est de déterminer **comment la position des rotors varie en fonction de leur affichage et de leur position d'anneau**. Egalement, nous souhaitons observer la modification de l'entraînement du rotor suivant.

Question 36 Lors de l'instanciation d'une machine Enigma complète, le paramètre `ring_settings` permet de modifier la position des anneaux des trois rotors. La valeur de ce paramètre est une liste de trois nombres entiers compris entre 0 et 25 (ou quatre, si la machine Enigma possède 4 rotors).
Instancier la machine Enigma en positionnant les 3 anneaux sur la position 0. Régler l'affichage des trois rotors sur `ADA` et chiffrer un texte composé de 5 caractères `A`. Que peut-on observer par rapport au chiffrement précédent ?

```
theMachine = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
theMachine.set_display("ADA")
print(theMachine.process_text("A" * 5))
```

Question 37 Nous obtenons le même résultat que précédemment car nous n'avons pas modifié les positions des anneaux. Désormais, réaliser de nouveau le même chiffrement, cette fois-ci en positionnant l'anneau du rotor `III` sur la position 1.
Quelle est la différence avec un positionnement de l'anneau du rotor `III` sur 0 ?

```
theMachine = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 1])
theMachine.set_display("ADA")
print(theMachine.process_text("A" * 5))
```

Question 38 Toujours en positionnant l'anneau du rotor `III` sur la position 1, régler cette fois-ci l'affichage des trois rotors sur `ADB`.
Chiffrer un texte composé de 5 caractères `A`. Que peut-on observer par rapport aux deux derniers chiffrements ?

```
theMachine = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 1])
theMachine.set_display("ADB")
print(theMachine.process_text("A" * 5))
```

Cette expérience permet d'observer l'effet d'une modification de position de l'anneau des rotors sur leur position : appliquer un décalage sur l'anneau a le même effet qu'une modification de l'affichage, choisi avec la méthode `set_display()`, mais en sens inverse. En effet, deux configurations donnent le même résultat lors d'un chiffrement de 5 lettres :

— placer l'anneau du rotor `III` sur la position 0 et positionner l'affichage du rotor sur la lettre `A` ;
— placer l'anneau du rotor `III` sur la position 1 et positionner l'affichage du rotor sur la lettre `B`.

Placer l'anneau du rotor `III` sur la position 1 et positionner l'affichage du rotor sur la lettre `A` donne aussi le même résultat que placer l'anneau du rotor `III` sur la position 0 et positionner l'affichage du rotor sur la lettre `Z`.

Modèle mathématique de la position des rotors

Dans notre modèle mathématique d'Enigma, nous avons modélisé les positions des rotors, respectivement n_1, n_2 et n_3, à l'aide de trois nombres entiers compris dans l'intervalle $[0:26[$. D'après l'expérience que nous venons de réaliser :

— soit r_1 le réglage de l'anneau (*ring*) du rotor I, r_2 le réglage de l'anneau du rotor II et r_3 le réglage de l'anneau du rotor III,
— soit d_1 le réglage de l'affichage (*display*) du rotor I, d_2 le réglage de l'affichage du rotor II et d_3 le réglage de l'affichage du rotor III,

Alors les positions n_1, n_2 et n_3 des rotors sont définies par les équations suivantes :

$$\begin{aligned} &\forall\, n_1,n_2,n_3 \in [0:26[, \\ &\forall\, r_1,r_2,r_3 \in [0:26[, \\ &\forall\, d_1,d_2,d_3 \in [0:26[, \\ &\qquad n_1 = (d_1 - r_1)\ mod\ 26 \\ &\qquad n_2 = (d_2 - r_2)\ mod\ 26 \\ &\qquad n_3 = (d_3 - r_3)\ mod\ 26 \end{aligned} \tag{3}$$

Question 39 Nous souhaitons désormais observer l'effet d'une modification de position de l'anneau des rotors sur la position des encoches.
Instancier deux machines Enigma avec les mêmes rotors dans le même ordre ainsi que le même réflecteur.

— sur la première machine, positionner les anneaux sur $(0,0,0)$ et l'affichage sur `AAA`,
— sur la seconde machine, positionner les anneaux sur $(0,0,1)$ et l'affichage sur `AAB`.

Ces deux machines ont donc la même position initiale, malgré des réglages différents.

```
theMachine1 = EnigmaMachine.from_key_sheet(
              rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
theMachine1.set_display("AAA")
#
theMachine2 = EnigmaMachine.from_key_sheet(
              rotors="I II III", reflector="B", ring_settings=[0, 0, 1])
theMachine2.set_display("AAB")
```

Question 40 Sur les deux machines, chiffrer 30 fois le caractère `A` ; à chaque caractère, afficher le résultat du chiffrement ainsi que la position des trois rotors de chaque machine Enigma. La liste des rotors de la machine est accessible via l'attribut `rotors`.
Pour chacune des deux machines, quelle est la position du rotor `III` lorsque le rotor `II` effectue sa première avancée ?
Quelle différence obtient on pour le chiffrement ? Combien de caractères sont différents et pour quelles positions ?

```
thePlaintext  = "A" * 30
for theChar in thePlaintext:
  theCipherText1 = theMachine1.process_text(theChar)
  theCipherText2 = theMachine2.process_text(theChar)
  print(theCipherText1 + " [%(0)02d %(1)02d %(2)02d]" % {
      "0": theMachine1.rotors[0].pos,
      "1": theMachine1.rotors[1].pos,
      "2": theMachine1.rotors[2].pos
    } + " / "
      + theCipherText2 + " [%(0)02d %(1)02d %(2)02d]" % {
      "0": theMachine2.rotors[0].pos,
      "1": theMachine2.rotors[1].pos,
      "2": theMachine2.rotors[2].pos
    }
  )
```

Question 41 Afin de confirmer notre intuition, modifier le réglage initial de la seconde machine Enigma (garder identique celui de la première machine). Cette fois-ci :

— sur la seconde machine, positionner les anneaux sur $(0,0,5)$ et l'affichage sur `AAF`.

Ici aussi, les deux machines ont la même position initiale, malgré des réglages différents.

```
theMachine1 = EnigmaMachine.from_key_sheet(
              rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
theMachine1.set_display("AAA")
#
theMachine2 = EnigmaMachine.from_key_sheet(
              rotors="I II III", reflector="B", ring_settings=[0, 0, 5])
theMachine2.set_display("AAF")
```

Question 42 De nouveau, chiffrer 30 fois le caractère `A` sur les deux machines ; à chaque caractère, afficher le résultat du chiffrement ainsi que la position des trois rotors de chaque machine Enigma.
Pour chacune des deux machines, quelle est la position du rotor `III` lorsque le rotor `II` effectue sa première avancée ? Quelle différence obtient on pour le chiffrement ? Combien de caractères sont différents et pour quelles positions ?

```
thePlaintext  = "A" * 30
for theChar in thePlaintext:
  theCipherText1 = theMachine1.process_text(theChar)
  theCipherText2 = theMachine2.process_text(theChar)
  print(theCipherText1 + " [%(0)02d %(1)02d %(2)02d]" % {
      "0": theMachine1.rotors[0].pos,
      "1": theMachine1.rotors[1].pos,
      "2": theMachine1.rotors[2].pos
    } + " / "
      + theCipherText2 + " [%(0)02d %(1)02d %(2)02d]" % {
      "0": theMachine2.rotors[0].pos,
      "1": theMachine2.rotors[1].pos,
      "2": theMachine2.rotors[2].pos
    }
  )
```

Cette expérience permet d'observer l'effet d'une modification de position de l'anneau des rotors sur la position de l'encoche : appliquer un décalage sur l'anneau modifie l'index qui provoquera une rotation du rotor suivant. En effet, par défaut, le rotor `III` pousse le rotor `II` lorsque sa position atteint l'index 21 (index de la lettre `V`). Si nous décalons son anneau de 1 cran, alors il pousse le rotor `II` lorsque sa position atteint l'index 20.

Seulement, ceci impacte le texte chiffré pendant un nombre fixe de caractères : tous les 26 caractères, nous retrouvons la même position quelque soit le réglage des anneaux. C'est-à-dire, lorsque pour chaque rotor :

$$(d-r) \; mod \; 26 = (d-r)$$

Nous avons terminé notre cryptanalyse du **premier modèle de la machine Enigma**, commercialisé en Europe et dans le reste du monde à partir de 1923. Dans la suite de ce TP, nous allons étudier les évolutions de la machine Enigma et les Bombes Cryptologiques qui ont permis de décrypter les communications chiffrées avec Enigma. Nous utiliserons seulement les fonctionnalités apportées par la bibliothèque Python `enigma` pour implémenter la machine.

L'armée allemande adopte la machine Enigma dès 1926 pour chiffrer ses communications. Seulement, le modèle utilisé était modifié par rapport à celui que nous venons d'étudier. En effet, celui-ci comportait des rotors dont les substitutions étaient différentes de la version commerciale. Egalement un **tableau de connexions** était présent, situé entre les rotors et le clavier.

4.4 Le tableau de connexions

Les machines des armées, et non les machines commerciales, avaient un tableau de connexions à l'avant de la machine, composé de l'alphabet. Avec des simili-prise jack, des câbles se connectent face à un alphabet pour permuter les deux lettres interconnectés. Quand une touche est pressée, le courant électrique passe d'abord par le câble de la lettre échangée, avant de traverser les rotors, qui fonctionnent normalement. [Plusieurs] paires de lettres sont ainsi permutées chaque jour. C'est la partie de la machine qui possédait les possibilités de connexions les plus élevés, bien plus que les rotors.

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

Question 43 Le script `tp-enigma/04/connect.py` contient un squelette prêt pour réaliser une analyse du tableau de connexions. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/04
debian@myhostname:~/tp-enigma/04$ ./connect.py
```

Dans la suite de cette section, nous allons modifier ce script pour réaliser notre étude.

La bibliothèque Python `enigma` fournit un modèle du tableau de connexions que nous pouvons instancier. La méthode `enigma.plugboard.Plugboard.from_key_sheet()` permet de construire ce modèle à partir d'une liste de bigrammes. Chaque bigramme définit une connexion entre deux lettres de l'alphabet. L'attribut `wiring_map` fournit le tableau de substitutions réalisées par le tableau de connexions.

Question 44 La méthode `signal()` permet d'appliquer les substitutions à partir de l'index d'une lettre de l'alphabet. Avec le tableau de connexions instancié dans le script, afficher le résultat d'une substitution de la lettre `A`.

```
theLetterIn  = "A"
theIndexIn   = theAlphabet.find(theLetterIn)
theIndexOut  = thePlugboard.signal(theIndexIn)
theLetterOut = theAlphabet[theIndexOut]
print(theLetterIn  + " (" + str(theIndexIn)  + ") -> " + \
      theLetterOut + " (" + str(theIndexOut) + ")")
```

Question 45 Toujours avec le même tableau de connexions, afficher le résultat des substitutions de toutes les lettres de l'alphabet.
Est il possible d'utiliser le tableau de connexions pour substituer la lettre `A` par un `B`, la lettre `B` par un `C`, et la lettre `C` par un `A` ? Conclure sur le fonctionnement du tableau de connexions.

```
for theLetterIn in theAlphabet:
  theIndexIn   = theAlphabet.find(theLetterIn)
  theIndexOut  = thePlugboard.signal(theIndexIn)
  theLetterOut = theAlphabet[theIndexOut]
  print(theLetterIn  + " (" + str(theIndexIn)  + ") -> " + \
        theLetterOut + " (" + str(theIndexOut) + ")")
```

Le tableau de connexions ne permet que de **permuter les lettres deux à deux** lors des substitutions. Donc si la lettre `A` est substituée par un `B`, alors la lettre `B` est forcément substituée par un `A`.
Nous allons désormais instancier une machine Enigma complète, sur laquelle nous utilisons le tableau de connexions, et observer l'impact de celui-ci.

Question 46 Instancier deux machines Enigma utilisant les rotors `I`, `II` et `III` ainsi que le réflecteur `B`. Conserver une position des anneaux sur $(0,0,0)$. Régler l'affichage des rotors sur `AAA`.
Sur la première machine Enigma, ne pas utiliser le tableau de connexions.
Sur la seconde machine Enigma, configurer le tableau avec la connexion suivante : `BD`.
Le paramètre `plugboard_settings` permet de définir la configuration du tableau de connexions.

```
theMachine1 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B")
theMachine1.set_display("AAA")
#
theMachine2 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", plugboard_settings="BD")
theMachine2.set_display("AAA")
```

Question 47 A l'aide de ces deux machines Enigma, chiffrer le texte clair composé de 50 fois le caractère `A` et observer le résultat (en particulier, les trois premières lettres).
Sur la première machine, la première lettre est un `B`. Sur la deuxième machine, la lettre `B` est elle substituée par un `D` ?
Sur la première machine, la deuxième lettre est un `D`. Sur la deuxième machine, la lettre `D` est elle substituée par un `B` ?
A quel niveau est donc positionnée la permutation dans la machine Enigma ?

```
thePlaintext  = "A" * 50
theCipherText = theMachine1.process_text(thePlaintext)
print("CIPHER TEXT 1 = " + theCipherText)
theCipherText = theMachine2.process_text(thePlaintext)
print("CIPHER TEXT 2 = " + theCipherText)
```

Dans le cas que nous venons de réaliser, la machine Engima chiffre la lettre `A` par un `B` en première lettre. Or, sur le tableau de connexions, la lettre `B` est permutée avec `D`. Nous observons donc une lettre `D` à chaque fois où nous devrions observer un `B`. De la même manière, vice-versa : nous observons un `B` lorsque nous avons un `D` sans le tableau de connexions.
Le tableau de connexions est donc placé **à la sortie** de la machine Enigma.

Question 48 Ré-itérons l'expérience. Cette fois-ci, sur la seconde machine Enigma, configurer le tableau avec les connexions suivantes : `BD AM`.

```
theMachine1 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B")
theMachine1.set_display("AAA")
#
theMachine2 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B",
                plugboard_settings="BD AM")
theMachine2.set_display("AAA")
```

Question 49 De nouveau, à l'aide de ces deux machines Enigma, chiffrer le texte clair composé de 50 fois le caractère `A` et observer le résultat (en particulier, les trois premières lettres).
Sur la première machine, la première lettre est un `B`. Sur la deuxième machine, la lettre `B` est elle substituée par un `D` ?
Que peut-on observer sur les autres lettres ? Pourquoi ?

```
thePlaintext  = "A" * 50
theCipherText = theMachine1.process_text(thePlaintext)
print("CIPHER TEXT 1 = " + theCipherText)
theCipherText = theMachine2.process_text(thePlaintext)
print("CIPHER TEXT 2 = " + theCipherText)
```

Le tableau de connexions ne réalise pas seulement une substitution à la sortie de la machine Enigma. Il réalise deux substituions : une lorsque la lettre est tapée au clavier et une avant que l'ampoule indiquant le résultat ne s'allume. En réalité, dans notre deuxième expérience, nous n'avons pas chiffré 50 fois la lettre `A` mais 50 fois la lettre `M` (comme `A` est permutée avec `M` sur le tableau de connexions).

En effet, le tableau de connexions est positionné entre les rotors et l'ensemble clavier/ampoules. La figure 8 donne une représentation simplifiée de l'emplacement du tableau de connexions dans la machine Enigma. Afin de comprendre ce qu'il s'est réellement passé lors de l'ajout du tableau de connexions, en particulier pourquoi nous obtenons le même

résultat pour la première lettre, nous devons reconstruire la machine Enigma pour pouvoir observer les substitutions internes.

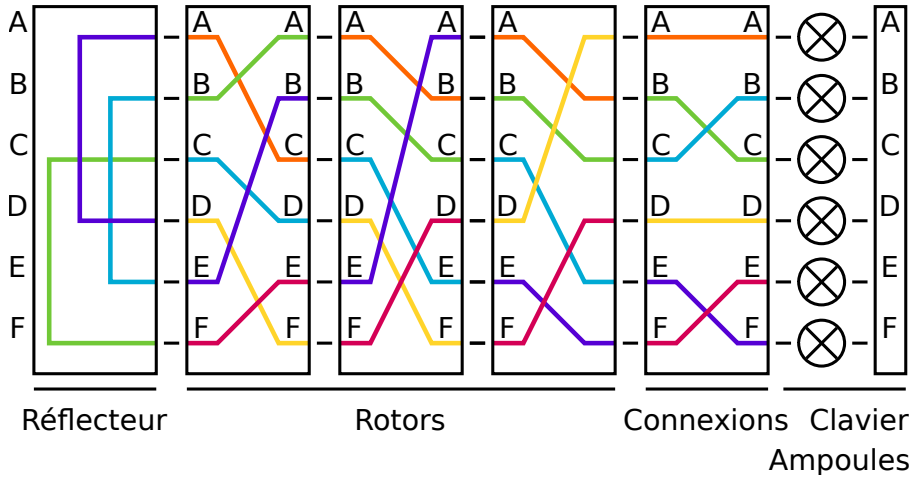

FIGURE 8 – Emplacement du tableau de connexions dans la machine Enigma

Question 50 Instancier les trois rotors `I`, `II` et `III` seuls, ainsi que le réflecteur `B`. Pour cela, utiliser les méthodes `create_rotor()` et `create_reflector()`.
Egalement, modifier la position des rotors (à l'aide de l'attribut `pos`) pour former la position initiale `AAA`.

```
theRotor1  = enigma.rotors.factory.create_rotor("I")
theRotor2  = enigma.rotors.factory.create_rotor("II")
theRotor3  = enigma.rotors.factory.create_rotor("III")
theReflect = enigma.rotors.factory.create_reflector("B")
#
theRotor1.pos = theAlphabet.find("A")
theRotor2.pos = theAlphabet.find("A")
theRotor3.pos = theAlphabet.find("A")
```

Question 51 Comme dans la section précédente, réaliser le chiffrement d'une chaîne de caractères composée de 3 fois le caractère `A`.
A chaque traversée de rotor/réflecteur, afficher l'index du résultat de la substitution. Une fois le chiffrement terminé, vérifier que nous obtenons bien les mêmes 3 premières lettres qu'avec la machine Enigma complète (sans le tableau de connexions).

```
thePlaintext  = "A" * 3
theCipherText = ""
for theLetterIn in thePlaintext:

  # rotate before cipher:
  if ( theRotor2.pos == theAlphabet.find("E") ):
    theRotor1.rotate()
  if ( theRotor3.pos == theAlphabet.find("V") or theRotor2.pos == theAlphabet.find("E") ):
    theRotor2.rotate()
  theRotor3.rotate()

  # no plugboard:
  theIndex = theAlphabet.find(theLetterIn)
  print(theLetterIn + " (" + str(theIndex) + ")")
  theIndex = theRotor3.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
  theIndex = theRotor2.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
  theIndex = theRotor1.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
  theIndex = theReflect.signal_in(theIndex) ; print(" [  (" + str(theIndex) + ")")
  theIndex = theRotor1.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
  theIndex = theRotor2.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
  theIndex = theRotor3.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
```

```
  theCipherText += theAlphabet[theIndex]
  print("= " + theAlphabet[theIndex])

print("CIPHER TEXT   = " + theCipherText)
```

Question 52 Nous allons comparer les substitutions intermédiaires avec celles de la machine Enigma équipée du tableau de connexions.
Instancier le tableau de connexions avec les mêmes permutations que précédemment et positionner de nouveau les rotors pour former la position initiale AAA.

```
thePlugboard = enigma.plugboard.Plugboard.from_key_sheet("BD AM")
#
theRotor1.pos = theAlphabet.find("A")
theRotor2.pos = theAlphabet.find("A")
theRotor3.pos = theAlphabet.find("A")
```

Question 53 Réaliser de nouveau le chiffrement d'une chaîne de caractères composée de 3 fois le caractère A et afficher l'index du résultat de la substitution à chaque traversée de rotor/réflecteur. Une fois le chiffrement terminé, vérifier que nous obtenons bien les mêmes 3 premières lettres qu'avec la machine Enigma complète (avec le tableau de connexions).

Important ! La traversée du tableau de connexions est effective deux fois, avant et après la traversée de tous les rotors / réflecteur.

```
theCipherText = ""
for theLetterIn in thePlaintext:

  # rotate before cipher:
  if ( theRotor2.pos == theAlphabet.find("E") ):
    theRotor1.rotate()
  if ( theRotor3.pos == theAlphabet.find("V") or theRotor2.pos == theAlphabet.find("E") ):
    theRotor2.rotate()
  theRotor3.rotate()

  theIndex = theAlphabet.find(theLetterIn)
  print(theLetterIn + " (" + str(theIndex) + ")")
  theIndex = thePlugboard.signal(theIndex)  ; print(" #  (" + str(theIndex) + ")")
  theIndex = theRotor3.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
  theIndex = theRotor2.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
  theIndex = theRotor1.signal_in(theIndex)  ; print(" <- (" + str(theIndex) + ")")
  theIndex = theReflect.signal_in(theIndex) ; print(" [  (" + str(theIndex) + ")")
  theIndex = theRotor1.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
  theIndex = theRotor2.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
  theIndex = theRotor3.signal_out(theIndex) ; print(" -> (" + str(theIndex) + ")")
  theIndex = thePlugboard.signal(theIndex)  ; print(" #  (" + str(theIndex) + ")")
  theCipherText += theAlphabet[theIndex]
  print("= " + theAlphabet[theIndex])

print("CIPHER TEXT   = " + theCipherText)
```

Question 54 Observer les résultats des substitutions intermédiaires et expliquer pourquoi les deux machines Enigma fournissent la même image (B) pour la première lettre A tapée.
Faire de même pour les deux lettres A tapées en suivant. Pourquoi les deux machines donnent des résultats différents (D et Z pour la première contre Y et K pour la seconde) ?

Ceci conclut notre cryptanalyse de la machine Enigma telle qu'elle était utilisée par l'armée allemande avant et pendant la seconde guerre mondiale. Maintenant que nous savons comment les substitutions sont réalisées, nous pouvons en conclure que l'intuition et la cryptanalyse manuelle sur texte chiffré seul seront insuffisantes pour "casser" Enigma (c'est-à-dire décrypter les messages futurs).

Pour réaliser leur attaque sur la machine Enigma, Marian Rejewski et Alan Turing ont tous deux conçu une **Bombe Cryptologique** : une machine électromécanique permettant de simplifier le déchiffrement des messages lors d'une attaque à clair connu.

5 Attaque d'Enigma : étude préliminaire

Marian Adam Rejewski est un mathématicien et cryptologue polonais né le 16 août 1905 à Bromberg et mort le 13 février 1980 à Varsovie. Il est à l'origine de la première attaque cryptanalytique sur la machine de chiffrement Enigma au début des années 1930.

Les réalisations de Rejewski et de ses collègues cryptologues Jerzy Różycki et Henryk Zygalski permettent aux Britanniques de commencer à lire des messages chiffrés en allemand passant par Enigma au début de la Seconde Guerre mondiale [...]. Les renseignements obtenus grâce à ces déchiffrements font partie du programme Ultra et contribuent, peut-être de manière décisive, à la défaite du Troisième Reich.

Source : `https://fr.wikipedia.org/wiki/Marian_Rejewski`

Dans cette section, nous allons, de la même manière que Marian Rejewski, réaliser une étude préliminaire sur Enigma pour identifier les classes d'attaque que nous pouvons réaliser. Nous allons étudier les procédures opérationnelles des utilisateurs, évaluer les aspects vulnérables aux attaques par force brute et les mettre en équations.

5.1 Un peu d'Histoire...

En 1926, la marine de guerre allemande adopte l'usage d'**une machine Enigma modifiée** ; en 1928, l'armée de terre allemande fait de même.

L'après-midi du dernier samedi de janvier 1929, un colis d'Allemagne est bloqué par la douane de Varsovie. D'après le manifeste, il contient de l'équipement radio. Le représentant de l'entreprise allemande exige que ce colis, expédié par erreur hors valise diplomatique, soit renvoyé en Allemagne avant de passer en douane. Alertés par son insistance, les douaniers préviennent le Bureau du chiffre (*Biuro Szyfrów*) [...]. Mandatés par le Bureau du chiffre, Ludomir Danilewiecz et Antoni Palluth ouvrent avec mille précautions la boîte qui contient, non pas une radio, mais une machine à chiffrer. Du samedi soir au lundi matin, ils examinent l'engin sous toutes les coutures, avant de le remballer soigneusement. Il s'agit d'une **machine à chiffrer commerciale, de marque "Enigma"**. Le Bureau du chiffre s'empresse d'en acheter un exemplaire, par des voies tout à fait légales, auprès de son inventeur allemand le Dr Arthur Scherbius, qui la commercialisait dès 1923.

Pour décrypter les messages Enigma, trois informations sont nécessaires :

— une compréhension générale du fonctionnement d'Enigma,
— le câblage des rotors,
— et les réglages quotidiens (l'ordre des rotors, les positions des anneaux, l'affichage initial des rotors et les permutations sur le tableau de connexions).

Rejewski ne dispose que du premier élément, basé sur des informations déjà acquises par le *Biuro Szyfrów*.

Rejewski aborde donc le problème de la découverte du câblage des rotors [sur le modèle militaire de la machine Enigma : la machine modifiée]. Pour ce faire, selon l'historien David Kahn, il est le premier à utiliser les mathématiques pures dans l'analyse cryptographique. Les méthodes précédentes ont largement exploité les schémas linguistiques et les statistiques des textes en langage naturel : l'analyse fréquentielle des lettres. Rejewski applique des techniques de la théorie des groupes (des théorèmes sur les permutations) dans son "attaque" sur Enigma. Ces techniques mathématiques, combinées aux éléments fournis par le chef du renseignement radio français Gustave Bertrand, à partir de données de l'informateur allemand Hans-Thilo Schmidt, permettent à Rejewski de reconstituer les câblages internes des rotors et du réflecteur non rotatif de la machine.

"*La solution*", écrit Kahn, "*est le véritable exploit de Rejewski, qui l'élève au panthéon des plus grands cryptanalystes de tous les temps*". Rejewski utilise un théorème mathématique selon lequel deux permutations sont conjuguées si et seulement si elles ont la même structure de cycle, que le professeur de mathématiques et co-éditeur du journal de cryptologie *Cryptologia*, Cipher A. Deavours, décrit comme "*le théorème qui a remporté la Seconde Guerre mondiale*". [...] En décembre 1932, Marian Rejewski fait l'un des plus grands bonds en avant de l'histoire de la cryptologie : **en employant des mathématiques pures, la théorie des permutations et groupes, il reconstitue les interconnexions des rotors et du réflecteur**.

Source : `https://fr.wikipedia.org/wiki/Marian_Rejewski`

Dans ce TP, nous ne détaillons pas les travaux mathématiques de Marian Rejewski qui ont permis la reconstitution des câblages des rotors. Nous admettons que nous sommes en possession d'une machine Enigma possédant les câblages corrects et nous nous concentrons sur la troisième problématique : **déterminer les réglages quotidiens**. C'est-à-dire, pour un jour donné, nous devons déterminer l'ordre des rotors utilisés, leur position initiale et les permutations réalisées sur le tableau de connexions. Pour cette problématique, nous reproduisons les travaux réalisés par Marian Rejewski.

5.2 Procédures opérationnelles des utilisateurs

Pour réaliser l'attaque permettant de déterminer les réglages quotidiens, Marian Rejewski a exploité une faiblesse des procédures opérationnelles de l'armée allemande. Dans cette section, nous allons suivre cette procédure afin de chiffrer des messages avec Enigma de la même manière que l'armée allemande entre 1926 et 1939.

Procédure

Avant une communication, les machines Enigma des opérateurs devaient être configurées avec un réglage global. Ce réglage stipulait l'ordre des rotors `I`, `II` et `III` (seulement 3 rotors étaient utilisés), les positions des anneaux des rotors et les permutations à appliquer sur le tableau de connexions (entre 3 et 6). Tous les opérateurs avaient connaissance de ce réglage global, nous l'appellerons le *secret partagé* (*shared secret*). Il était **inconnu des adversaires** comme Marian Rejewski.

Chaque message était chiffré en utilisant un affichage différent pour la position de départ des rotors. Cet affichage était sélectionné par l'opérateur. Nous appellerons *clef de session* (*session key*) les trois lettres composant l'affichage utilisées comme position initiales pour chiffrer le message.

Pour transmettre la clef de session à l'opérateur destinataire, l'opérateur expéditeur commence le message par un indicateur à neuf lettres : **trois lettres non-chiffrées** fournissant un affichage temporaire pour obtenir la clef de session **ainsi qu'une répétition de la clef de session, chiffrée** depuis cet affichage temporaire. Cette répétition avait pour objectif de limiter les erreurs dues aux problèmes de transmission (réalisées en morse et dont les équipements n'étaient pas exempts de défauts).

Pour chiffrer :

1. L'opérateur d'envoi applique les réglages conformément au secret partagé. Il place les trois rotors aux emplacements définis, règle ensuite la position des anneaux des rotors et applique les permutations sur le tableau de connexions.
2. L'opérateur d'envoi choisit un affichage pour les trois rotors qu'il joint au message (en clair).
3. L'opérateur d'envoi choisit une clef de session, qu'il compte utiliser pour chiffrer le message. Il tape deux fois cette clef de session sur la machine Enigma et joint le résultat au message. Nous appelons ce résultat **l'identifiant**.
4. L'opérateur d'envoi modifie l'affichage de sa machine Enigma sur la clef de session puis tape son message.

Pour déchiffrer :

1. L'opérateur destinataire applique les réglages conformément au secret partagé. Il place les trois rotors aux emplacements définis, règle la position des anneaux des rotors et applique les permutations sur le tableau de connexions.
2. L'opérateur destinataire lit les trois premières lettres du message : il règle l'affichage des rotors sur ces trois lettres.
3. L'opérateur destinataire tape l'identifiant (les six premières lettres du message) reçu et observe le résultat :
   — Si le résultat est composé de deux fois le même trigramme, alors ce trigramme constitue la clef de session.
   — Si le résultat est composé de deux trigrammes différents, alors une erreur s'est produite et l'opérateur destinataire doit tester les deux trigrammes comme clef de session.
4. L'opérateur destinataire modifie le réglage de sa machine Enigma de sorte que l'affichage soit égal à la clef de session puis tape la suite du message.

Dans l'armée allemande, l'intervalle entre les modifications du secret partagé était trimestriel avant 1936, puis devient mensuel en février 1936, et enfin quotidien en octobre 1936.

Question 55 Le script `tp-enigma/05/procedure.py` contient un squelette prêt pour implémenter l'algorithme de chiffrement et déchiffrement d'un message. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/05
debian@myhostname:~/tp-enigma/05$ ./procedure.py
```

Dans la suite de cette section, nous allons modifier ce script pour implémenter notre algorithme.

Question 56 Nous partons du principe que tous les opérateurs utilisent un modèle de la machine Engima comportant le réflecteur `B`. Egalement, nous partons du principe que le secret partagé, valable pour la journée est le suivant :

- ordre des rotors : `II` puis `I` puis `III`
- position des anneaux : 12, 22, 21
- liste des permutations : `AL DF GP QM`

Instancier une machine Enigma et effectuer les réglages globaux conformément à notre secret partagé.

```
theRotorsOrder = "II I III"
theRing        = [12, 22, 21]
thePlugboard   = "AL DF GP QM"
#
theMachine = EnigmaMachine.from_key_sheet(
                rotors=theRotorsOrder, reflector="B",
                ring_settings=theRing, plugboard_settings=thePlugboard)
```

Question 57 Pour créer un indicateur de 9 lettres, choisir un affichage pour la position initiale des rotors (par exemple : `GBL`) et l'ajouter en clair au message.
Configurer Enigma pour que l'affichage des rotors soit sur cette position.

```
theClearPos   = "GBL"
theCipherText = theClearPos
theMachine.set_display(theClearPos)
```

Question 58 A l'aide des réglages globaux que nous venons de réaliser, chiffrer deux fois une clef de session (par exemple `KYG`) pour obtenir l'indicateur et ajouter le résultat au message.

```
theSessionKey  = "KYG"
theCipherText += theMachine.process_text(theSessionKey * 2)
```

Question 59 Modifier le réglage de la machine Enigma pour que l'affichage soit égal à la clef de session et chiffrer un message. Afficher ensuite le résultat.
Le message peut comporter des espaces mais ceux-ci seront remplacés par la lettre `X` lors de l'appel de la méthode `EnigmaMachine.process_text()`. Ceci reflète le comportement de certains opérateurs d'Enigma durant la seconde guerre mondiale, qui utilisaient la lettre `X` en guise d'espace (même si la consigne était d'omettre les espaces).

```
theMachine.set_display(theSessionKey)
thePlainText   = "VOICI UN MESSAGE CHIFFRE AVEC ENIGMA"
theCipherText += theMachine.process_text(thePlainText)
print("CIPHER MESSAGE   = " + theCipherText)
```

Pour plus de lisibilité, nous pouvons afficher le message en introduisant un espace après les troisième et neuvième caractères :

```
print("CIPHER MESSAGE   = " + theCipherText[0:3] + " " \
                             + theCipherText[3:9] + " " + theCipherText[9:])
```

Le message que nous venons d'obtenir est celui qui est transmis à l'opérateur destinataire. Nous allons désormais déchiffrer ce message comme si nous étions cet opérateur.

Question 60 Instancier une machine Enigma et effectuer les réglages globaux conformément à notre secret partagé.

```
theRotorsOrder = "II I III"
theRing        = [12, 22, 21]
thePlugboard   = "AL DF GP QM"
theMachine = EnigmaMachine.from_key_sheet(
                rotors=theRotorsOrder, reflector="B",
                ring_settings=theRing, plugboard_settings=thePlugboard)
```

Question 61 Observer les 3 premières lettres du message et positionner l'affichage des rotors sur ces trois lettres.

```
theClearPos = theCipherText[0:3]
theMachine.set_display(theClearPos)
```

Question 62 Déchiffrer les 6 caractères suivants du message à l'aide de la machine Enigma. En déduire la clef de session.

```
theIndicator = theMachine.process_text(theCipherText[3:9])
print("INDICATOR = " + theIndicator)
theSessionKey = theIndicator[0:3]
```

Question 63 Positionner l'affichage des rotors conformément à la clef de session et déchiffrer le reste du message.

```
theMachine.set_display(theSessionKey)
theDecipherText = theMachine.process_text(theCipherText[9:])
print("DECIPHER MESSAGE = " + theDecipherText)
```

Cette procédure d'échange de messages chiffrés a été employé par l'armée allemande entre 1926 et 1939. L'avantage de cette procédure est que l'affichage utilisé pour chiffrer est différent à chaque message, ce qui augmente la durée des attaques statistiques si un adversaire souhaite tester plusieurs secrets partagés. L'inconvénient est que cette procédure est lourde pour les utilisateurs légitimes (les correspondants) car deux réglages de l'affichage doivent être réalisés.

5.3 Ensemble des possibles

En admettant que cette procédure opérationnelle soit connue de l'adversaire (ce qui n'était pas supposé être le cas mais le fut après espionnage) alors, même si celle-ci ne communique pas l'ordre des rotors et leurs positions initiales, elle communique des informations sur le contenu du message. Avant de commencer notre attaque, décrivons par des équations les informations inconnues de l'adversaire.

Question 64 Soit o un nombre entier positif décrivant l'ordre des rotors utilisés. Le nombre o est inconnu de l'adversaire. Il existe trois rotors différents à placer dans n'importe quel ordre :

- combien de rotors peuvent êtres choisis pour être placé dans le premier emplacement ?
- une fois le premier rotor placé, combien de rotors peuvent êtres choisis pour être placé dans le second emplacement ?
- une fois les deux premiers rotors placés, combien de rotors peuvent êtres choisis pour être placé dans le troisième emplacement ?

En déduire l'intervalle des valeurs possibles pour o.

```
o = 3 * 2 * 1
print("Ordre des 3 rotors: o appartient a [0:" + str(o) + "[")
```

Question 65 Admettons que le nombre de rotors différents à placer soit augmenté de trois à cinq. La machine Enigma possède toujours trois emplacements pour les rotors.
Calculer l'intervalle des valeurs possibles pour o.

```
o = 5 * 4 * 3
print("Ordre des 5 rotors: o appartient a [0:" + str(o) + "[")
```

La valeur maximale pour o peut être calculée depuis une formule plus générale. Soit m un nombre entier représentant le nombre total de rotors que nous pouvons placer dans une Enigma à trois emplacements :

$$\forall m \geq 3,\ max(o) = \frac{!m}{!(m-3)} \tag{4}$$

Question 66 Soient n_1, n_2 et n_3 trois nombres entiers positifs, représentant respectivement les positions initiales des rotors placés aux emplacements 1, 2 et 3. Nous décrivons l'ensemble des trois positions sous forme d'un seul nombre entier positif n tel que :

$$n = n_1 \times 26 \times 26 + n_2 \times 26 + n_3$$

Chaque rotor peut avoir une position initiale parmi 26 :
- combien de possibilités existe-t-il pour la position initiale n_1 du premier rotor ?
- combien de possibilités existe-t-il pour la position initiale n_2 du second rotor ?
- combien de possibilités existe-t-il pour la position initiale n_3 du troisième rotor ?

En déduire l'intervalle des possibilités pour n.

```
n = 26 * 26 * 26
print("Position des rotors: n appartient a [0:" + str(n) + "[")
```

Dans la suite de ce TP, nous décrivons la position initiale des rotors avec ce nombre entier positif n et nous pourrons déduire n_1, n_2 et n_3 tels que :
- $n_3 = (n)\ mod\ 26$
- $n_2 = (n/26)\ mod\ 26$
- $n_1 = (n/26/26)\ mod\ 26$

De la même manière, nous décrivons la position des anneaux des rotors avec un nombre entier positif r et nous pourrons déduire r_1, r_2 et r_3 (positions des anneaux pour chaque rotor) tels que :
- $r_3 = (r)\ mod\ 26$
- $r_2 = (r/26)\ mod\ 26$
- $r_1 = (r/26/26)\ mod\ 26$

Idem pour l'affichage initial des rotors, avec un nombre entier positif d et nous pourrons déduire d_1, d_2 et d_3 (affichage initial pour chaque rotor) tels que :
- $d_3 = (d)\ mod\ 26$
- $d_2 = (d/26)\ mod\ 26$
- $d_1 = (d/26/26)\ mod\ 26$

En conséquence, nous pouvons directement écrire que la position initiale des rotors est liée à la position des anneaux et à l'affichage initial tel que :

$$n = (d - r)\ mod\ (26 \times 26 \times 26) \tag{5}$$

Question 67 Si nous considérons que la clef de session (l'affichage initial, c'est-à-dire d) est connu de l'adversaire, le nombre de possibilités pour trouver la position initiale n est-il réduit ?

Réponse : non, étant donné que la position des anneaux est inconnue, nous ne connaissons pas le résultat de la différence $d - r$.

Question 68 Soit p un nombre entier positif décrivant les permutations réalisées sur le tableau de connexions. Admettons qu'une seule paire de lettres est permutée,
- combien de possibilités existe-t-il pour choisir la première lettre à permuter ?
- une fois la première lettre choisie, combien de possibilités existe-t-il pour choisir la seconde lettre avec laquelle permuter ?

En déduire l'intervalle des valeurs possibles pour p.
Attention ! Permuter `A` avec `B` revient au même que permuter `B` avec `A`.

```
p = int(26 * 25 / 2)
# (AB and BA are the same, hence a division by 2)
print("Avec  1 paire  de lettres: p appartient a " + "[0:" + str(p) + "[")
```

Question 69 Même question si deux paires de lettres sont échangées.
Attention ! Permuter d'abord la première paire puis ensuite la seconde revient au même que permuter d'abord la seconde paire puis ensuite la première.

```
p = int( (26*25 / 2) * (24*23 / 2) / 2 )
# ("AB CD" and "CD AB" are the same, hence a new division by 2)
print("Avec  2 paires de lettres: p appartient a " + "[0:" + str(p) + "[")
```

Question 70 Même question si trois paires de lettres sont échangées.

```
p = int( (26*25 / 2) * (24*23 / 2) * (22*21 / 2) / (3*2) )
# (pairs of letters can be in any order)
print("Avec  3 paires de lettres: p appartient a " + "[0:" + str(p) + "[")
```

Question 71 Généraliser la formule et calculer pour connaître le nombre de possibilités si 10 paires de lettres sont échangées.

```
thePairs = 10
p = int(
  math.factorial(26)/math.factorial(26-2*thePairs) / math.pow(2, thePairs) / math.factorial(thePairs)
)
print("Avec " + str(thePairs) + " paires de lettres: p appartient a " + "[0:" + str(p) + "[")
```

Question 72 Les machines Enigma que nous souhaitons attaquer peuvent contenir jusqu'à :
- 5 rotors à placer dans 3 emplacements,
- 10 paires de lettres permutées sur le tableau de connexions.

Calculer le nombre de réglages possibles, incluant la position initiale des rotors, dans ce cas précis.

```
print("Total: " + str(o * n * p))
```

6 La Bombe Cryptologique de Rejewski

Marian Rejewski a conçu un appareil électromécanique, la Bombe Cryptologique (*bomba kryptologiczna* en polonais), qui permet d'exploiter une vulnérabilité que l'on trouvait dans les communications militaires allemandes entre 1926 et 1939. L'objectif de cet appareil est d'extraire l'ordre des rotors et leur position initiale (anneaux et affichage) ; les permutations sur le tableau de connexions ne sont pas considérées. L'engin est baptisé de ce nom parce qu'il fait tic-tac lorsqu'il fonctionne.

Dans cette section, nous allons implémenter l'algorithme de la Bombe Cryptologique de Marian Rejewski en procédant par étapes :
- d'abord nous concevons un modèle capable d'attaquer une machine Enigma sans tableau de connexions,
- ensuite nous considérons l'attaque d'une machine Enigma avec un tableau de connexions partiel,
- finalement, nous considérons l'attaque d'une machine Enigma complète.

Voici nos hypothèses :
- Nous partons donc du principe que **nous sommes en possession d'un grand nombre de messages chiffrés en suivant la procédure** décrite dans la section précédente, que nous connaissons.
- Nous considérons que **nous sommes en possession d'une réplique de la machine Enigma militaire** telle que Marian Rejewski l'a conçue et nous ne nous intéressons pas à la méthode de conception.
- La machine Enigma que nous attaquons possède trois emplacements pour les rotors et **seulement trois rotors peuvent être placés** (dans n'importe quel ordre).
- Egalement, nous considérons que nous sommes en possession d'une très faible puissance de calcul, c'est-à-dire de l'ordre des capacités humaines (les ordinateurs n'existaient pas dans les années 1930) : lors de notre attaque, **nous ne pouvons taper que deux caractères par seconde**.
- Pour finir, **l'emplacement des rotors et les positions des anneaux changent quotidiennement**. Nous devons donc trouver un moyen d'obtenir ces informations en moins d'une journée de travail (8h).

A partir de ces hypothèses, nous allons établir une stratégie d'attaque et construire une Bombe Cryptologique. Nous allons introduire de nouvelles hypothèses, basées sur les modèles mathématiques simplifiés d'Enigma que nous considérons. Ces nouvelles hypothèses peuvent être valides ou invalides. A chaque étape, nous allons calculer le **taux de réussite** (probabilité que nos hypothèses soient valides) et la **durée de l'attaque**. L'objectif de notre étude est, au maximum, d'augmenter le taux de réussite et réduire la durée de l'attaque.

6.1 L'oracle

Avant de concevoir la Bombe, nous devons établir une stratégie d'attaque. Dans un premier temps, nous considérons que nous attaquons une machine Enigma dépourvue du tableau de connexions. L'idée proposée par Marian Rejewski est de réaliser une attaque par force brute sur certaines clefs qu'il considère comme probables. En effet, son hypothèse est qu'il est possible, à l'aide d'une machine, d'**écarter de l'attaque par force brute des clefs qui sont improbables**. Pour cela, il est nécessaire de trouver un oracle (ou distingueur) qui nous permet d'identifier les clefs probables.

Question 73 Le script `tp-enigma/06/rejewski.py` contient un squelette prêt pour implémenter une Bombe Cryptologique. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/06
debian@myhostname:~/tp-enigma/06$ ./rejewski.py
```

Question 74 Nous sommes en possession d'un message chiffré avec Enigma, où l'opérateur a suivi la procédure décrite précédemment :

```
GBL TXQJZS WFTWLCMDBAIDQLQBSUDUNIYMUZUJVVVEABII
```

Admettons que nous ne savons pas quelle lettre l'opérateur a tapée pour obtenir la première lettre `T` : appelons cette lettre α_0 (*alpha* zéro). Si la procédure a été correctement suivie, quelle lettre l'opérateur a tapée pour obtenir la première lettre `J` ?

Si la procédure a été correctement suivie, alors nous pouvons déduire des informations du message. En effet, les six caractères aux indexes 3 à 8 constituent la répétition d'un trigramme : ceci est donc une faiblesse des procédures opérationnelles. En l'occurrence, la première et la quatrième lettre de cette répétition résultent de **deux chiffrements différents de la même lettre**. Si l'opérateur a tapé α_0 pour obtenir la première lettre `T`, alors il a aussi tapé α_0 pour obtenir la première lettre `J`.

Question 75 Soit $E(n,\alpha)$ la fonction mathématique qui décrit la transformation réalisée par la machine Enigma, sans le tableau de connexions, pour une position $n \in [0 : 26\times 26\times 26[$ des trois rotors et une lettre $\alpha \in [A-Z]$ tapée au clavier. Admettons que nous ne connaissons pas la position des rotors avant la première écriture : appelons cette position n_0. Nous pouvons donc écrire que :

$$E(n_0,\alpha_0) = \texttt{T} \tag{6}$$

Emettons l'hypothèse que le second rotor n'a pas tourné entre l'écriture du premier et du quatrième caractère. Conformément à la question précédente, quelle autre équation pouvons nous écrire ?

Réponse : Si le second rotor n'a pas tourné et qu'Enigma se trouve à la position n_0 avant l'écriture du premier caractère, alors nous pouvons en déduire qu'Enigma se trouve trouve à la position n_0+3 avant l'écriture du quatrième caractère. Or nous savons que l'opérateur a tapé la lettre α_0 pour obtenir le quatrième caractère `J`, nous pouvons donc en déduire que :

$$E(n_0+3,\alpha_0) = \texttt{J} \tag{7}$$

Pour rappel, la machine Enigma est **symétrique**, ce qui signifie que, pour une position n donnée, entrer dans Enigma une lettre d'un message chiffré donne la lettre dans le message clair. Donc, pour deux lettres de l'alphabet α et β quelconques, nous pouvons écrire le théorème suivant :

$$\begin{aligned} &\forall\, n \in [0 : 26\times 26\times 26[,\\ &\quad \forall\, \alpha,\beta \in [A-Z],\\ &\qquad E(n,\alpha)=\beta \iff E(n,\beta)=\alpha \end{aligned} \tag{8}$$

Question 76 Appliquer ce théorème sur l'équation 6, quelle nouvelle équation obtenons nous ?

Question 77 Ce que nous obtenons est une expression de α_0 ; remplacer α_0 par cette expression dans l'équation 7. Quelle nouvelle équation obtenons nous ? La lettre α_0 est elle présente dans cette équation ?

Question 78 Dans le cas de notre Enigma sans tableau de connexions, nous cherchons à réaliser une attaque par force brute sur les clefs probables : c'est-à-dire sur des positions initiales des rotors que nous avons identifiées. De la dernière équation, déduire un oracle qui nous permet d'identifier une clef probable.

Solution :
Si nous appliquons le théorème sur l'équation 6, nous obtenons :

$$E(n_0, \mathtt{T}) = \alpha_0$$

En remplaçant l'expression de α_0 dans l'équation 7, nous obtenons :

$$E(n_0+3, E(n_0, \mathtt{T})) = \mathtt{J} \tag{9}$$

Cette expression est indépendante de α_0. En conclusion, sans savoir quelle lettre l'opérateur tapée deux fois (aux indexes 0 et 3), nous pouvons déterminer un oracle qui nous permet d'identifier une clef probable : *la position initiale des rotors utilisée pour chiffrer le message est telle que, si nous chiffrons la lettre* `T` *à cette position puis chiffrons le résultat à la position* $n+3$*, alors nous obtenons la lettre* `J`.

Question 79 Cet oracle n'est valide que dans le cas où le second rotor n'a pas tourné entre les positions n_0 et n_0+3. Calculer le **taux de réussite** pour que l'établissement de notre oracle soit valide.

```
success = (26 - 3)/26
print("Probabilite que l'oracle soit valide: " + str(success))
```

Question 80 Pour l'ordre des rotors `I II III` et la position des anneaux (0, 0, 0), instancier deux machines Enigma que nous allons utiliser pour réaliser deux fonctions $E(n, \alpha)$.

```
theRotorsOrder = "I II III"
theRing        = [0, 0, 0]
thePlugboard   = ""
theMachine1 = EnigmaMachine.from_key_sheet(
                rotors=theRotorsOrder, reflector="B",
                ring_settings=theRing, plugboard_settings=thePlugboard)
theMachine2 = EnigmaMachine.from_key_sheet(
                rotors=theRotorsOrder, reflector="B",
                ring_settings=theRing, plugboard_settings=thePlugboard)
```

Question 81 A partir de ces deux machines Enigma, réaliser une attaque par force brute sur la valeur n_0, trouver et compter les valeurs qui vérifient l'équation 9.
Attention ! Comme cette équation n'est valide que dans le cas où le second rotor n'a pas tourné, s'assurer que l'addition $+3$ ne soit réalisée que sur la position du troisième rotor (celui qui tourne à chaque frappe).

```
theCount = 0
for n0 in range(0, 26*26*26):
  n3 = int(n0 % 26)
  n2 = int((n0 / 26) % 26)
  n1 = int((n0 / (26*26)) % 26)

  theDisplay1 = theAlphabet[n1] + theAlphabet[n2] + theAlphabet[n3]
  theMachine1.set_display(theDisplay1)
  alpha_0 = theMachine1.process_text("T")

  theDisplay2 = theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 3) % 26]
  theMachine2.set_display(theDisplay2)
  if ( theMachine2.process_text(alpha_0) == "J" ):
    print("clef probable = " + theDisplay1)
```

```
    theCount += 1

print("nombre de clefs probables: " + str(theCount))
```

Cette attaque sur la valeur n_0 constitue l'essence même de la Bombe de Rejewski. L'idée est de réaliser un test simple sur la position des rotors et écarter de l'attaque par force brute des clefs qui sont improbables. Ensuite, une fois le filtrage terminé, nous pouvons manuellement tester les clefs probables en tentant de déchiffrer le corps du message que nous attaquons. L'oracle que Marian Rejewski a choisi est obtenu en assemblant plusieurs machines Enigma avec des positions de départ différentes.

Notons que cette attaque donne la position de départ (n) pour les rotors. Or les trois premières lettres du message sont en clair et indiquent l'affichage (d) qui doit être visible sur la machine Enigma. Ce que nous recherchons, c'est le réglage des anneaux (r). Pour déduire cette valeur, il suffit simplement de réaliser, **pour chaque rotor**, une soustraction entre l'affichage attendu et la position de départ (n) de notre clef probable. En effet, selon notre définition :

— $r_1 = (d_1 - n_1)\ mod\ 26$
— $r_2 = (d_2 - n_2)\ mod\ 26$
— $r_3 = (d_3 - n_3)\ mod\ 26$

Question 82 Admettons que faire une attaque par force brute sur les clefs probables requiert de déchiffrer le message que nous attaquons (sans les 3 premiers caractères, qui sont en clair) en tapant 2 caractères par secondes.
Combien de clés probables avons nous obtenues ? Combien de **temps va durer l'attaque** sur ces clefs probables ?

```
theTime = int((len(theCipherText) - 3) / 2) * theCount
print("duree de l'attaque brute force [s]:   " + str(theTime))
print("duree de l'attaque brute force [min]: " + str(theTime / 60))
print("duree de l'attaque brute force [h]:   " + str(theTime / 3600))
```

Faire une attaque par force brute sur ces clefs probables prend quelques heures. Cependant, pour la machine Enigma que nous attaquons, nous n'avons pas listé l'intégralité des clefs probables. En effet, nous avons seulement listé celles où l'ordre des rotors est `I II III`. Or, les possibilités pour l'ordre des rotors s'élève, pour m rotors à placer, à :

$$\frac{!m}{!(m-3)}$$

Question 83 Admettons que nous ayons construit une Bombe Cryptologique qui extrait toutes les clefs probables pour l'ordre des rotors `I II III`. Que pouvons nous faire pour les ordres différents ?

Réponse : nous devons construire plusieurs Bombes Cryptologiques, une pour chaque ordre des rotors possible. Jusqu'en décembre 1938, il n'existait que trois rotors différents à placer dans Enigma. Les cryptanalystes du Bureau du chiffre ont donc construit 6 Bombes.

6.1.1 Schéma de l'attaque

Résumons par un schéma le moyen d'obtention d'un oracle pour notre attaque sur Enigma. Nous savons que l'opérateur a tapé la lettre α_0 deux fois : lorsque Enigma est à la position n_0 et à la position $n_0 + 3$. A ces positions, nous avons obtenu deux lettres : `T` et `J`.

L'idée consiste donc à utiliser deux machines Enigma dont la position est décalée de 3, d'en retourner une (Enigma est symétrique) et de les connecter. Peu importe la lettre α_0 qui a été tapée deux fois, si la position n_0 est correcte, alors transformer deux fois la lettre `T` doit donner un `J`. La figure 9 schématise cette stratégie permettant d'obtenir notre oracle.

Résumé :
— taux de réussite : 88.45%
— durée de l'attaque : $\approx 6 \times 4$h

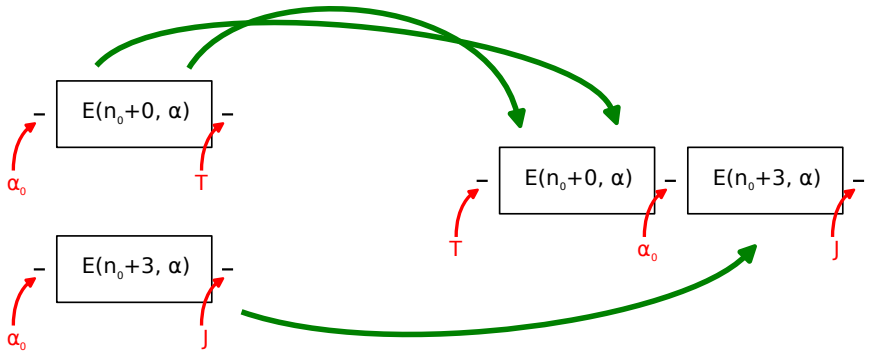

FIGURE 9 – Oracle : connexions de machines Enigma

6.2 Architecture de la Bombe

Lors de la conception de la Bombe Cryptologique, l'idée consiste à assembler astucieusement plusieurs répliques de machine Enigma dont la position de départ est différente. Une source de courant est connectée à l'entrée de la première machine, toutes les sorties de cette machine sont connectées à toutes les entrées de la machine suivante, puis un relai est connecté à une sortie de la machine suivante. Si le test est réussi, alors la Bombe a détecté une clef probable : le relai stoppe le tic-tac de la Bombe et l'opérateur peut lire la dite clef sur celle-ci. Le schéma de la figure 10 représente les connexions à réaliser entre deux machines Enigma pour détecter les clefs probables en fonction d'un oracle. Sur ce schéma, $E(n,\alpha_0) = \texttt{C}$ et $E(n+3,\alpha_0) = \texttt{E}$.

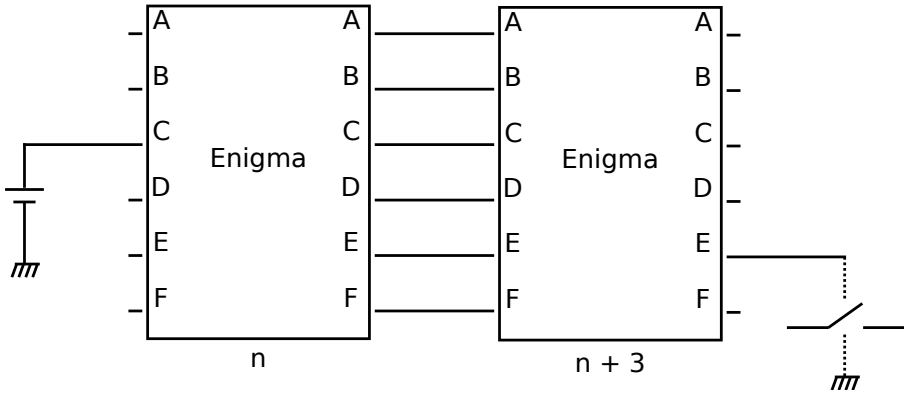

FIGURE 10 – Schéma électrique permettant de détecter les clefs probables

L'ennui est que, pour réaliser ce schéma, nous ne pouvons pas assembler des machines Enigma complètes. En effet, celles-ci comportent un clavier et 26 ampoules. De plus les mêmes fils de connexion sont utilisés en entrée et en sortie d'Enigma. Pour réaliser sa Bombe Cryptologique, Marian Rejewski a du faire concevoir des répliques d'Enigma que l'on peut connecter entre elles. C'est-à-dire des circuits électromécaniques comportant 26 entrées et 26 sorties.

Dans ces répliques, le réflecteur est traversant (mais il réalise toujours les mêmes substitutions) et les rotors sont doublés : trois rotors en entrée et les trois mêmes rotors en sortie (en miroir). Le schéma de la figure 11 représente une réplique d'Enigma telle qu'elle a été introduite dans la Bombe. A chaque itération, une rotation d'un rotor sur la vraie machine Enigma se traduit ici par une rotation de deux rotors (celui qui doit tourner et son miroir).

Question 84 Nous attaquons toujours le même message chiffré avec Enigma, où l'opérateur a suivi la procédure décrite précédemment :

```
GBL TXQJZS WFTWLCMDBAIDQLQBSUDUNIYMUZUJVVVEABII
```

Appelons α_0, α_1 et α_2 les trois lettres que l'opérateur a tapée deux fois dans l'indicateur. Proposer une solution pour améliorer notre oracle et écarter de nouvelles clefs improbables.

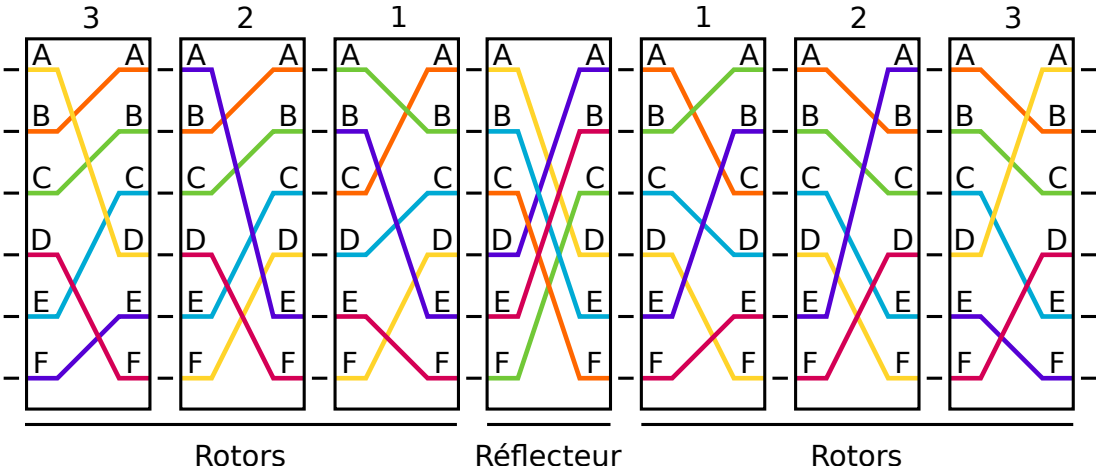

FIGURE 11 – Réplique d'Enigma dans la Bombe Cryptologique

Comme l'opérateur a répété un trigramme dans son indicateur, nous avons en réalité trois répétitions. Le résultat de ces répétitions après chiffrement est `TXQJZS`. Exactement comme dans notre attaque précédente où nous avons chiffré deux fois un `T` (à n_0 puis à n_0+3) pour obtenir un `J`, nous pouvons en plus :

— chiffrer deux fois un `X` (à n_0+1 puis à n_0+4) pour obtenir un `Z`,
— chiffrer deux fois un `Q` (à n_0+2 puis à n_0+5) pour obtenir un `S`.

Question 85 Durant notre première attaque, notre oracle n'était valide que dans le cas où le second rotor n'a pas tourné entre les positions n_0 et n_0+3. Pour cette nouvelle attaque, entre quelles positions le second rotor ne doit pas tourner pour que notre oracle reste valide ?

Question 86 Calculer le **taux de réussite** pour que l'établissement de notre nouvel oracle soit valide.

```
success = (26 - 5)/26
print("Probabilite que l'oracle soit valide: " + str(success))
```

Question 87 Pour l'ordre des rotors `I II III` et la position des anneaux (0, 0, 0), instancier six machines Enigma que nous allons utiliser pour réaliser six fonctions $E(n,\alpha)$.

```
theMachine1 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
theMachine2 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
theMachine3 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
theMachine4 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
theMachine5 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
theMachine6 = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B", ring_settings=[0, 0, 0])
```

Question 88 A partir de ces six machines Enigma, réaliser une attaque par force brute sur la valeur n_0, trouver et compter les valeurs qui vérifient trois équations semblables à l'équation 9.
Attention ! Comme cette équation n'est valide que dans le cas où le second rotor n'a pas tourné, s'assurer que l'addition $+3$, $+4$ ou $+5$ ne soit réalisée que sur la position du troisième rotor (celui qui tourne à chaque frappe).

```
theCount = 0
for n0 in range(0, 26*26*26):
  n3 = int(n0 % 26)
  n2 = int((n0 / 26) % 26)
  n1 = int((n0 / (26*26)) % 26)

  theDisplay1 = theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 0) % 26]
  theDisplay2 = theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 1) % 26]
  theDisplay3 = theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 2) % 26]
  theDisplay4 = theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 3) % 26]
  theDisplay5 = theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 4) % 26]
```

```
  theDisplay6 = theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 5) % 26]

  theMachine1.set_display(theDisplay1)
  theMachine2.set_display(theDisplay2)
  theMachine3.set_display(theDisplay3)
  theMachine4.set_display(theDisplay4)
  theMachine5.set_display(theDisplay5)
  theMachine6.set_display(theDisplay6)

  if ( (theMachine4.process_text(theMachine1.process_text("T") == "J") and \
       (theMachine5.process_text(theMachine2.process_text("X") == "Z") and \
       (theMachine6.process_text(theMachine3.process_text("Q") == "S") \
     ):
    print("clef probable = " + theDisplay1)
    theCount += 1

print("nombre de clefs probables: " + str(theCount))
```

Question 89 Le nombre de clefs probables obtenu est suspect. Quelles sont les deux raisons possibles qui peuvent mener à un tel résultat ?

Réponse : Avec notre attaque sur la répétition du trigramme, notre itération ne trouve aucune clef probable. Pourtant, notre algorithme est correct. C'est forcément qu'une de nos hypothèses est fausse. Nous avons émis deux hypothèses :

— le second rotor n'a pas tourné entre les positions n_0 et n_0+5 : cette hypothèse possède un taux de réussite de 80%.

— l'ordre des rotors est `I II III` : cette hypothèse a un taux de réussite de $1/6$.

Pour le bien de ce TP, le texte que nous attaquons a été spécifiquement choisi pour que le second rotor ne tourne pas entre les positions n_0 et n_0+5. La raison pour laquelle notre algorithme ne trouve pas de clef probable est que l'ordre des rotors n'est pas correct. Si nous avons plusieurs Bombes Cryptologiques qui effectuent les même tests en parallèle, alors l'une d'entre elles trouve (au moins) une clef probable.

Question 90 Sur les six machines Enigma que nous avons instanciées, modifier l'ordre des rotors en `I III II` et ré-exécuter l'algorithme. Combien de clefs probables obtenons nous ?

Question 91 A partir de l'affichage attendu, en clair dans le message, et de la clef probable obtenue, réaliser trois soustractions modulo 26 pour obtenir le réglage des anneaux des trois rotors.

```
r1 = (theAlphabet.find("G") - theAlphabet.find("A")) % 26
r2 = (theAlphabet.find("B") - theAlphabet.find("F")) % 26
r3 = (theAlphabet.find("L") - theAlphabet.find("Q")) % 26
print([r1, r2, r3])
```

Question 92 Instancier une machine Enigma avec les rotors dans l'ordre `I III II` et le réglage des anneaux que nous venons de trouver. Modifier l'affichage tel que décrit en clair dans l'entête du message que nous attaquons.

```
theMachine = EnigmaMachine.from_key_sheet(
                rotors="I III II", reflector="B", ring_settings=[r1, r2, r3])
theMachine.set_display("GBL")
```

Question 93 Déchiffrer les 6 caractères suivants du message. Vérifier que, une fois déchiffrés, ceux-ci contiennent bien une répétition d'un trigramme.

```
theSessionKey = theMachine.process_text(theCipherText[3:9])
print(theSessionKey)
```

Question 94 Modifier l'affichage sur la machine Enigma conformément au trigramme répété. Déchiffrer ensuite le corps du message. La clef probable que nous avons identifiée est-elle correcte ?

```
theMachine.set_display(theSessionKey[0:3])
theClearText = theMachine.process_text(theCipherText[9:])
print(theClearText)
```

Question 95 Toujours dans le cas où déchiffrer le message que nous attaquons (sans les 3 premiers caractères, qui sont en clair) requiert de taper 2 caractères par secondes, combien de **temps va durer l'attaque** sur les clefs probables que nous avons obtenues ?

```
theTime = int((len(theCipherText) - 3) / 2) * theCount
print("duree de l'attaque brute force [s]: " + str(theTime))
```

Félicitations ! Vous venez de déterminer l'architecture de la Bombe de Rejewski et décrypter votre premier message chiffré avec Enigma ! La Bombe de Rejewski est donc constituée de six répliques d'Enigma (comme représentées sur la figure 11) connectées deux à deux (comme représenté sur la figure 10). Trois relais, à la sortie de chaque paire de répliques, sont connectés en série et arrêtent la Bombe lorsque la position des rotors, définissant la valeur d'un nombre entier n, vérifient trois équations semblables à l'équation 9. Trois sources de courant, à l'entrée de chaque paire de répliques, sont connectées en parallèle afin de fournir de quoi actionner les trois relais.

6.2.1 Modèle mathématique

Résumons l'attaque réalisée à partir d'un modèle mathématique. Soient $\alpha_0, \beta_0, \gamma_0 \in [A-Z]$ trois lettres **inconnues** tapées aux clavier par l'opérateur. Soient $\alpha_1, \beta_1, \gamma_1, \alpha_2, \beta_2, \gamma_2 \in [A-Z]$ six lettres **connues** qui composent l'identifiant dans le message chiffré. Soit $n_0 \in [0 : 26 \times 26 \times 26[$ la position initiale d'Enigma que nous recherchons.
Voici nos hypothèses :

$$\begin{cases} \textbf{H1 :} & E(n_0, & \alpha_0) = \alpha_1 \\ \textbf{H2 :} & E(n_0+1, & \beta_0) = \beta_1 \\ \textbf{H3 :} & E(n_0+2, & \gamma_0) = \gamma_1 \\ \textbf{H4 :} & E(n_0+3, & \alpha_0) = \alpha_2 \\ \textbf{H5 :} & E(n_0+4, & \beta_0) = \beta_2 \\ \textbf{H6 :} & E(n_0+5, & \gamma_0) = \gamma_3 \end{cases}$$

En appliquant le théorème 8 sur les hypothèses **H1**, **H2** et **H3**, nous obtenons :

$$\begin{cases} \textbf{H1 :} & E(n_0, & \alpha_1) = \alpha_0 \\ \textbf{H2 :} & E(n_0+1, & \beta_1) = \beta_0 \\ \textbf{H3 :} & E(n_0+2, & \gamma_1) = \gamma_0 \end{cases}$$

En remplaçant les expressions de α_0, β_0 et γ_0 dans les hypothèses **H4**, **H5** et **H6**, nous obtenons :

$$\begin{cases} \textbf{H4 :} & E(n_0+3, E(n_0, & \alpha_1)) = \alpha_2 \\ \textbf{H5 :} & E(n_0+4, E(n_0+1, & \beta_1)) = \beta_2 \\ \textbf{H6 :} & E(n_0+5, E(n_0+2, & \gamma_1)) = \gamma_2 \end{cases}$$

La Bombe Cryptologique teste toutes les valeurs possibles pour n_0 et détecte une clef probable lorsque les équations **H4**, **H5** et **H6** sont vérifiées.

6.3 Permutations et taux de réussite

Nous avons atteint une durée d'attaque suffisamment réduite. Notre attaque fonctionne sans tableau de connexions. Plus particulièrement, notre attaque fonctionne si aucune des lettres intervenant dans nos équations n'est permutée. L'objectif de cette section est d'introduire les permutations causées par le tableau de connexions et d'observer l'impact sur le taux de réussite. Egalement, l'objectif est d'améliorer le taux de réussite malgré la présence de permutations.

Question 96 Le script `tp-enigma/07/swaps.py` contient un squelette prêt pour observer l'effet des permutations sur la Bombe Cryptologique. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/07
debian@myhostname:~/tp-enigma/07$ ./swaps.py
```

Ce script contient deux fonctions :
- la fonction `procedure()` permet de générer un message en suivant la procédure décrite dans la section 5.2.
- la fonction `bomba_simpl()` permet de réaliser l'attaque que nous avons implémentée dans la section précédente avec la Bombe de Rejewski. Cette fonction comporte une légère différence avec notre attaque : le test n'est pas appliqué sur la position d'Enigma mais sur le réglage des anneaux. Il est donc nécessaire de lui fournir l'affichage (par exemple `GBL`) : les soustractions sont réalisées dans son algorithme. De ce fait, cette fonction retourne directement le réglage des anneaux, pas besoin de le calculer.

Question 97 Appeler la fonction `procedure()` avec les mêmes réglages que précédemment pour obtenir un message chiffré identique.

```
procedure("I III II", "B", [6, 22, 21], "", "GBL", "MDR")
```

Question 98 A partir de l'affichage initial et de l'identifiant de 6 lettres que nous attaquons, appeler la fonction `bomba_simpl()` pour réaliser la même attaque que précédemment.
Vérifier que la clef que nous recherchons est présente parmi les clefs probables trouvées.

```
bomba_simpl("I III II", "B", "GBL", "TXQJZS")
```

Notre attaque fonctionne si aucune des lettres intervenant dans nos équations n'est permutée. Désormais, nous ajoutons des permutations sur le tableau de connexions. A chaque expérience, nous appelons la fonction `procedure()` pour obtenir l'identifiant que nous souhaitons attaquer puis la fonction `bomba_simpl()` pour observer si la clef recherchée est trouvée ou non.

Nous identifions deux cas qui peuvent poser problème :

1. le premier est le cas où l'opérateur a tapé une lettre (en entrée d'Enigma) qui est permutée.
2. le second est le cas où la lettre obtenue dans l'identifiant (à la sortie d'Enigma) est permutée.

6.3.1 Permutations dans le message clair (lettres tapées)

Question 99 Dans notre message, l'affichage chiffré deux fois par l'opérateur est `MDR`. Permuter ces lettres et chiffrer de nouveau le message pour obtenir un identifiant.
Attention ! Choisir les lettres permutées telles que le résultat du chiffrement ne fasse pas apparaitre de lettre permutée dans l'identifiant (nous souhaitons décorréler les deux problèmes).

```
procedure("I III II", "B", [6, 22, 21], "MD RE", "GBL", "MDR")
```

Question 100 Réaliser l'attaque avec la Bombe de Rejewski sur l'identifiant obtenu. La clef que nous cherchons est elle présente parmi les clefs probables ?

```
bomba_simpl("I III II", "B", "GBL", "QJYAIG")
```

Question 101 Réitérer l'expérience avec d'autres permutations de `M`, `D` et `R` (toujours sans permuter le contenu de l'identifiant). La clef que nous cherchons est elle présente parmi les clefs probables ? Qu'est-ce qui peut expliquer un tel comportement ?

```
procedure("I III II", "B", [6, 22, 21], "MD RF", "GBL", "MDR")
bomba_simpl("I III II", "B", "GBL", "QJOAIZ")
```

Explication par le schéma

La clef que nous cherchons est toujours présente parmi les clefs probables trouvées par la Bombe. La raison est que **les permutations dans les lettres tapées par l'opérateur n'ont aucun impact** sur l'attaque que nous réalisons. Afin d'expliquer ce comportement nous pouvons réaliser un schéma qui résume notre attaque.

Nous savons que l'opérateur a tapé la lettre α_0 deux fois : lorsque Enigma est à la position n_0 et à la position n_0+3. Dans notre cas, cette lettre est permutée : α_0 devient $P(\alpha_0)$ et, à ces positions, nous avons obtenu deux lettres : Q et A. Les lettres Q et A ne sont pas permutées. Le schéma équivalent de la machine réalisant la transformation est donc une cascade d'un tableau de connexions en entrée et d'une machine Enigma : pas de tableau de connexions à la sortie.

Pour obtenir notre oracle, comme précédemment, nous devons utiliser deux fois ce schéma équivalent : une fois retourné (Enigma avec son tableau de connexions est symétrique), une fois dans le même sens, et les connecter. La figure 12 schématise cette stratégie permettant d'obtenir notre oracle.

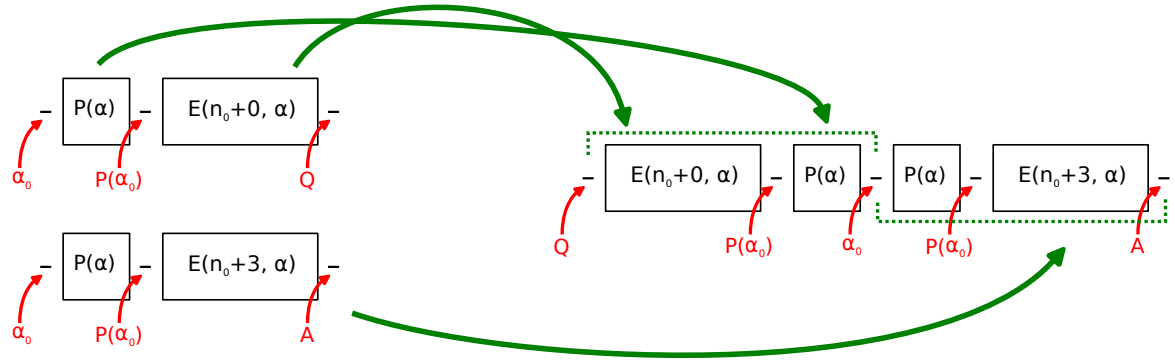

FIGURE 12 – Oracle : connexions de machines Enigma, tableau de connexions en entrée

Or, il s'avère qu'avec une telle connexion, nous sommes amenés à positionner deux fois le tableau de connexions à la suite. Et, comme effectuer deux fois la même permutation revient à ne rien permuter du tout : nous pouvons retirer les deux tableaux de connexions car leur effet s'annule. En conséquence, le schéma obtenu est identique à celui représenté sur la figure 9. Nous obtenons donc bien notre oracle : $E(n_0, \mathtt{Q})$ nous donne $P(\alpha_0)$ et $E(n_0+3, P(\alpha_0))$ nous donne A.

Explication par les mathématiques

Une autre manière d'expliquer ce comportement est de se tourner vers les mathématiques.

Soit $\alpha, \beta \in [A-Z]$ deux lettres de l'alphabet. Soit $P(\alpha)$ la fonction mathématique qui décrit la transformation réalisée par le tableau de connexions (*plugboard*) pour une lettre $\alpha \in [A-Z]$ quelconque tapée au clavier.

$$\forall\ \alpha, \beta \in [A-Z],\ P(\alpha) = \begin{cases} \alpha & \text{si } \alpha \text{ n'est pas permutée} \\ \beta \neq \alpha & \text{si } \alpha \text{ est permutée} \end{cases}$$

Egalement, une permutation échange les lettres deux à deux. En conséquence, la fonction $P(\alpha)$ est symétrique. Nous pouvons donc écrire :

$$\forall\ \alpha, \beta \in [A-Z],\ P(\alpha) = \beta \iff P(\beta) = \alpha \tag{10}$$

et en déduire que :

$$\forall\ \alpha \in [A-Z],\ P(P(\alpha)) = \alpha \tag{11}$$

Soient $\alpha_0, \beta_0, \gamma_0 \in [A-Z]$ trois lettres **inconnues** tapées aux clavier par l'opérateur. Soient $\alpha_1, \beta_1, \gamma_1, \alpha_2, \beta_2, \gamma_2 \in [A-Z]$ six lettres **connues** qui composent l'identifiant dans le message chiffré. Soit $n_0 \in [0 : 26 \times 26 \times 26[$ la position initiale d'Enigma que nous recherchons.

Voici nos hypothèses :

$$\begin{cases} \textbf{H1 :} & E(n_0, & P(\alpha_0)) = \alpha_1 \\ \textbf{H2 :} & E(n_0+1, & P(\beta_0)) = \beta_1 \\ \textbf{H3 :} & E(n_0+2, & P(\gamma_0)) = \gamma_1 \\ \textbf{H4 :} & E(n_0+3, & P(\alpha_0)) = \alpha_2 \\ \textbf{H5 :} & E(n_0+4, & P(\beta_0)) = \beta_2 \\ \textbf{H6 :} & E(n_0+5, & P(\gamma_0)) = \gamma_3 \end{cases}$$

En appliquant le théorème 8 sur les hypothèses **H1**, **H2** et **H3**, nous obtenons :

$$\begin{cases} \textbf{H1 :} & E(n_0, & \alpha_1) = P(\alpha_0) \\ \textbf{H2 :} & E(n_0+1, & \beta_1) = P(\beta_0) \\ \textbf{H3 :} & E(n_0+2, & \gamma_1) = P(\gamma_0) \end{cases}$$

Appliquons le théorème 10 sur chacune des équations.

$$\begin{cases} \textbf{H1 :} & P(E(n_0, & \alpha_1)) = \alpha_0 \\ \textbf{H2 :} & P(E(n_0+1, & \beta_1)) = \beta_0 \\ \textbf{H3 :} & P(E(n_0+2, & \gamma_1)) = \gamma_0 \end{cases}$$

En remplaçant les expressions de α_0, β_0 et γ_0 dans les hypothèses **H4**, **H5** et **H6**, nous obtenons :

$$\begin{cases} \textbf{H4 :} & E(n_0+3, P(P(E(n_0, & \alpha_1)))) = \alpha_2 \\ \textbf{H5 :} & E(n_0+4, P(P(E(n_0+1, & \beta_1)))) = \beta_2 \\ \textbf{H6 :} & E(n_0+5, P(P(E(n_0+2, & \gamma_1)))) = \gamma_2 \end{cases}$$

Finalement, nous appliquons le théorème 11 au coté gauche des équations.

$$\begin{cases} \textbf{H4 :} & E(n_0+3, E(n_0, & \alpha_1)) = \alpha_2 \\ \textbf{H5 :} & E(n_0+4, E(n_0+1, & \beta_1)) = \beta_2 \\ \textbf{H6 :} & E(n_0+5, E(n_0+2, & \gamma_1)) = \gamma_2 \end{cases}$$

Ces équations sont identiques à celles de la section précédente. Pour rappel, la Bombe Cryptologique teste toutes les valeurs possibles pour n_0 et détecte une clef probable lorsque les équations **H4**, **H5** et **H6** sont vérifiées.

6.3.2 Permutations dans le message chiffré (identifiant)

Question 102 Dans notre messages, lorsque les lettres tapées par l'opérateur ne sont pas permutées, l'identifiant est `TXQJZS`. Permuter certaines de ces lettres et chiffrer de nouveau le message pour obtenir un nouvel identifiant.

```
procedure("I III II", "B", [6, 22, 21], "TX", "GBL", "MDR")
```

Question 103 Réaliser l'attaque avec la Bombe de Rejewski sur l'identifiant obtenu. La clef que nous cherchons est elle présente parmi les clefs probables ? Qu'est-ce qui peut expliquer un tel comportement ?

```
bomba_simpl("I III II", "B", "GBL", "XTQJZS")
```

Explication par le schéma

La clef que nous cherchons est désormais absente des clefs probables trouvées par la Bombe. La raison est que **les permutations dans le message chiffré empêche la réussite de l'attaque** que nous réalisons. Afin d'expliquer ce comportement nous pouvons réaliser un schéma qui résume notre attaque.

Nous savons que l'opérateur a tapé la lettre α_0 deux fois : lorsque Enigma est à la position n_0 et à la position n_0+3. Dans notre cas, cette lettre n'est pas permutée et, à ces positions, nous avons obtenu deux lettres qui sont potentiellement permutées. Après permutation, nous obtenons `X` et `J`. Le schéma équivalent de la machine réalisant la transformation est donc une cascade d'une machine Enigma et d'un tableau de connexions en sortie : pas de tableau de connexions à l'entrée.

Pour obtenir notre oracle, comme précédemment, nous devons utiliser deux fois ce schéma équivalent : une fois retourné (Enigma avec son tableau de connexions est symétrique), une fois dans le même sens, et les connecter. La figure 13 schématise cette stratégie permettant d'obtenir notre oracle.

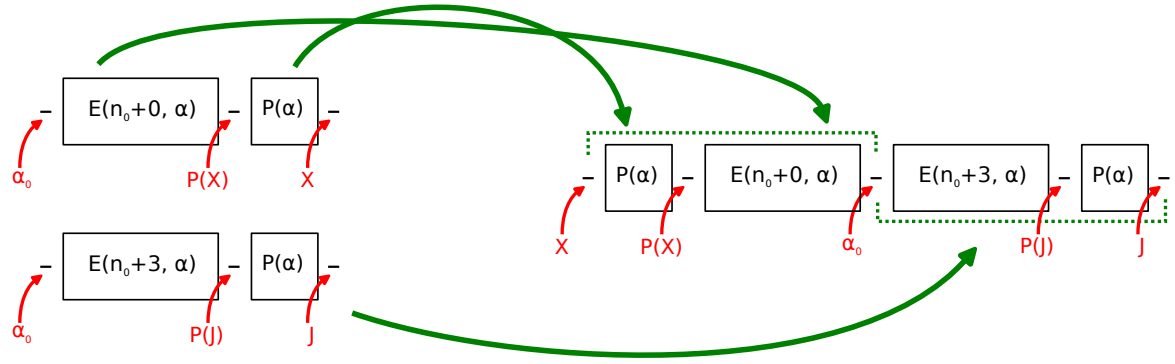

FIGURE 13 – Oracle : connexions de machines Enigma, tableau de connexions en sortie

Or, il s'avère qu'avec une telle connexion, nous devons entrer la lettre `P(X)` en entrée de la première machine Enigma et tester que nous obtenons bien `P(J)` en sortie de deuxième machine Enigma. L'ennui est que l'attaque que nous avons implémentée, dont le schéma est représenté sur la figure 9, ne teste pas ces lettres : elle teste que `X` devient `J`. L'attaque ne peut pas fonctionner car nous n'avons pas ajouté les tableaux de connexions comme sur la figure 13. Malheureusement, les permutations sur le tableau de connexions sont inconnues.

Explication par les mathématiques

Une autre manière d'expliquer ce comportement est de se tourner vers les mathématiques.

Soient $\alpha_0, \beta_0, \gamma_0 \in [A-Z]$ trois lettres **inconnues** tapées aux clavier par l'opérateur. Soient $\alpha_1, \beta_1, \gamma_1, \alpha_2, \beta_2, \gamma_2 \in [A-Z]$ six lettres **connues** qui composent l'identifiant dans le message chiffré. Soit $n_0 \in [0 : 26 \times 26 \times 26[$ la position initiale d'Enigma que nous recherchons.
Voici nos hypothèses :

$$\begin{cases} \textbf{H1 :} & E(n_0, & \alpha_0) = P(\alpha_1) \\ \textbf{H2 :} & E(n_0+1, & \beta_0) = P(\beta_1) \\ \textbf{H3 :} & E(n_0+2, & \gamma_0) = P(\gamma_1) \\ \textbf{H4 :} & E(n_0+3, & \alpha_0) = P(\alpha_2) \\ \textbf{H5 :} & E(n_0+4, & \beta_0) = P(\beta_2) \\ \textbf{H6 :} & E(n_0+5, & \gamma_0) = P(\gamma_3) \end{cases}$$

En appliquant le théorème 8 sur les hypothèses **H1**, **H2** et **H3**, nous obtenons :

$$\begin{cases} \textbf{H1 :} & E(n_0, & P(\alpha_1)) = \alpha_0 \\ \textbf{H2 :} & E(n_0+1, & P(\beta_1)) = \beta_0 \\ \textbf{H3 :} & E(n_0+2, & P(\gamma_1)) = \gamma_0 \end{cases}$$

En remplaçant les expressions de α_0, β_0 et γ_0 dans les hypothèses **H4**, **H5** et **H6**, nous obtenons :

$$\begin{cases} \textbf{H4 :} & E(n_0+3, E(n_0, & P(\alpha_1))) = P(\alpha_2) \\ \textbf{H5 :} & E(n_0+4, E(n_0+1, & P(\beta_1))) = P(\beta_2) \\ \textbf{H6 :} & E(n_0+5, E(n_0+2, & P(\gamma_1))) = P(\gamma_2) \end{cases}$$

Pour rappel, la Bombe Cryptologique teste toutes les valeurs possibles pour n_0 et détecte une clef probable lorsque les équations **H4**, **H5** et **H6** sont vérifiées. Or les valeurs $P(\alpha_1)$, $P(\beta_1)$, $P(\gamma_1)$, $P(\alpha_2)$, $P(\beta_2)$ et $P(\gamma_2)$ sont inconnues. Nous ne savons donc pas comment vérifier ces équations et nous ne pouvons pas détecter une clef probable.

6.3.3 Améliorer le taux de réussite

Nous avons vu que les permutations dans le message chiffré empêchent la réussite de l'attaque que nous réalisons. Cependant, l'ajout du tableau de connexions ne permute pas toutes les lettres. En effet, d'après notre modèle mathématique du tableau de connexions :

$$\forall\, \alpha,\beta \in [A-Z],\ P(\alpha) = \begin{cases} \alpha & \text{si } \alpha \text{ n'est pas permutée} \\ \beta \neq \alpha & \text{si } \alpha \text{ est permutée} \end{cases}$$

Il est donc possible que notre attaque réussisse même si un tableau de connexions est présent : dans le cas où aucune lettre dans l'identifiant (message chiffré) n'est permutée. Dans cette section nous allons calculer le taux de réussite et tenter de l'améliorer.

Afin de simplifier les calculs, plaçons nous dans le pire cas et admettons trois hypothèses (fausses) :

— nous attaquons un identifiant où les 6 lettres sont choisies aléatoirement avec une distribution uniforme.
— les permutations sur le tableau de connexions sont choisies aléatoirement avec une distribution uniforme.
— la présence des lettres dans l'identifiant et les permutations sur le tableau de connexions sont des phénomènes décorrélés.

Question 104 Dans le cas où nous avons une seule paire de lettres permutée sur le tableau de connexions, quelle est la probabilité pour que **la première lettre de l'identifiant** ne soit pas permutée ?

```
print(" 1 paire:  " + str((26 - 2) / 26))
```

Question 105 Même question pour deux paires de lettres permutées sur le tableau de connexions.

```
print(" 2 paires: " + str((26 - 2*2) / 26))
```

Question 106 Même question pour trois paires de lettres permutées sur le tableau de connexions.

```
print(" 3 paires: " + str((26 - 2*3) / 26))
```

Question 107 Avec trois paires de lettres permutées sur le tableau de connexions, quelle est la probabilité pour que **les deux premières lettres de l'identifiant** ne soient pas permutées ?

```
print(" 3 paires / 2 lettres differentes: " + str(math.pow((26 - 2*3) / 26, 2)))
```

Question 108 Même question pour les trois premières lettres de l'identifiant.

```
print(" 3 paires / 3 lettres differentes: " + str(math.pow((26 - 2*3) / 26, 3)))
```

Question 109 Généraliser la formule et calculer, pour 0 à 10 paires de lettres permutées sur le tableau de connexions, la probabilité que les 6 lettres de l'identifiant ne soient pas permutées.

```
print("6 lettres differentes:")
for thePairCount in range(0, 10 + 1):
  print("%2d" % thePairCount + " paires: " + str(
    math.pow((26 - 2*thePairCount) / 26, 6)
  ))
```

L'attaque que nous réalisons possède un faible taux de réussite lorsque plusieurs paires de lettres sont permutées sur le tableau de connexions. Marian Rejewski propose une solution pour améliorer ce taux de réussite en modifiant légèrement la Bombe. L'idée est d'**attaquer plusieurs messages en même temps**, ceci est rendu possible si **les six répliques de machines Enigma présentes dans la Bombe ont un affichage paramétrable**. Ainsi, nous pouvons astucieusement choisir les messages que nous attaquons pour réduire le nombre de lettres utilisées.

Question 110 Générer 10 messages en appelant la fonction `procedure()` et observer le contenu des identifiants. Chercher des identifiants où l'on retrouve les mêmes lettres pour différents indexes n et $n+3$.
Voici un exemple où nous forçons une *seed* et utilisons les fonctions de génération aléatoire :

```
random.seed(3)
for i in range(0, 10):
  theClearPos   = ""
  theSessionKey = ""
  for j in range(0, 3):
    theClearPos   += random.sample(list(theAlphabet), k=1)[0]
    theSessionKey += random.sample(list(theAlphabet), k=1)[0]
  print(str(i) + ": ", end="")
  procedure("I III II", "B", [6, 22, 21], "MD RE TX", theClearPos, theSessionKey)
```

Dans cet exemple, nous obtenons trois messages où les lettres `L`, `B` et `O` peuvent être utilisées :

```
print("1: PST I(L)FZ(B)O ZQEXQCWBGNUFJWMODZQMLPCURKWNOVFMHTFN")
print("6: YBY (L)YB(O)XT QDYXNIMLDPJQJBTKJVLXBQEYRPKUWNUHONGF")
print("8: ZMZ G(O)YF(B)U BUFYUEXIAESJDIVDESVBTVGIUCUAMUNHDRDN")
```

L'idée proposée par Marian Rejewski est donc que nous pouvons attaquer trois identifiants en même temps dans la Bombe, en ne choisissant qu'une seule équation parmi **H4**, **H5** et **H6**. Pour chacune des trois équations, nous remplaçons la position n_0 que nous faisons varier par n_1, n_2 ou n_3 selon si nous attaquons le premier, le deuxième ou le troisième identifiant. Ainsi, nous pouvons écrire :

$$\begin{cases} \textbf{H4}: & E(n_0+3, E(n_0,\ P(\alpha_1))) = P(\alpha_2) \\ \textbf{H5}: & E(n_0+4, E(n_0+1,\ P(\beta_1))) = P(\beta_2) \\ \textbf{H6}: & E(n_0+5, E(n_0+2,\ P(\gamma_1))) = P(\gamma_2) \end{cases} \iff \begin{cases} \textbf{H4}: & E(n_1+3, E(n_1,\ P(\alpha_1))) = P(\alpha_2) \\ \textbf{H5}: & E(n_2+3, E(n_2,\ P(\beta_1))) = P(\beta_2) \\ \textbf{H6}: & E(n_3+3, E(n_3,\ P(\gamma_1))) = P(\gamma_2) \end{cases}$$

L'idée est que la Bombe réalise une attaque par force brute sur une seule valeur : la position des anneaux. Or les trois identifiants que nous attaquons sont issus de messages interprétés le même jour : la position des anneaux utilisée pour ces trois messages est donc identiques (seul l'affichage diffère). Appelons r_0 la position des anneaux utilisée pour les trois messages et d_1, d_2, d_3 la valeur de l'affichage utilisé pour chaque message. Voici les nouvelles équations que la Bombe doit vérifier en faisant varier r_0 :

$$\begin{cases} \textbf{H4}: & E(d_1-r_0+3, E(d_1-r_0,\ P(\alpha_1))) = P(\alpha_2) \\ \textbf{H5}: & E(d_2-r_0+3, E(d_2-r_0,\ P(\beta_1))) = P(\beta_2) \\ \textbf{H6}: & E(d_3-r_0+3, E(d_3-r_0,\ P(\gamma_1))) = P(\gamma_2) \end{cases} \tag{12}$$

Pour rappel, les valeurs de l'affichage d_1, d_2 et d_3 sont données en clair en préfixe du message chiffré. Dans l'exemple précédent, ces valeurs sont les images des affichages `PST`, `YBY` et `ZMZ`.

Question 111 Avec cette nouvelle stratégie, nous n'utilisons que 3 lettres différentes dans les identifiants que nous attaquons. Calculer, pour 0 à 10 paires de lettres permutées sur le tableau de connexions, la probabilité que ces 3 lettres ne soient pas permutées.

```
print("3 messages / 3 lettres differentes:")
for thePairCount in range(0, 10 + 1):
  print("%2d" % thePairCount + " paires: " + str(
    math.pow((26 - 2*thePairCount) / 26, 3)
  ))
```

Question 112 Instancier 6 machines Enigma qui serviront de répliques pour notre nouvelle version de la Bombe de Rejewski. Choisir les rotors dans l'ordre `I III II` et $(0,0,0)$ pour la position des anneaux.

```
theMachine1 = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B", ring_settings=[0, 0, 0])
theMachine2 = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B", ring_settings=[0, 0, 0])
theMachine3 = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B", ring_settings=[0, 0, 0])
theMachine4 = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B", ring_settings=[0, 0, 0])
theMachine5 = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B", ring_settings=[0, 0, 0])
theMachine6 = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B", ring_settings=[0, 0, 0])
```

Question 113 Nous attaquons les messages suivants :

```
PST I(L)FZ(B)O ZQEXQCWBGNUFJWMODZQMLPCURKWNOVFMHTFN
YBY (L)YB(O)XT QDYXNIMLDPJQJBTKJVLXBQEYRPKUWNUHONGF
ZMZ G(O)YF(B)U BUFYUEXIAESJDIVDESVBTVGIUCUAMUNHDRDN
```

Configurer la Bombe pour réaliser une attaque par force brute sur la position des anneaux, où l'affichage des 6 répliques d'Enigma correspond aux valeurs fournies par les affichages de ces messages (attention à bien incrémenter la valeur si la lettre n'est pas à l'index 0 après configuration de l'affichage).

```
theCount = 0
for r0 in range(0, 26*26*26):
  r3 = int(r0 % 26)
  r2 = int((r0 / 26) % 26)
  r1 = int((r0 / (26*26)) % 26)

  theDisplay1 = theAlphabet[(theAlphabet.find("P") - r1) % 26] + \
                theAlphabet[(theAlphabet.find("S") - r2) % 26] + \
                theAlphabet[(theAlphabet.find("T") - r3 + 1) % 26]
  theDisplay4 = theAlphabet[(theAlphabet.find("P") - r1) % 26] + \
                theAlphabet[(theAlphabet.find("S") - r2) % 26] + \
                theAlphabet[(theAlphabet.find("T") - r3 + 4) % 26]

  theDisplay2 = theAlphabet[(theAlphabet.find("Y") - r1) % 26] + \
                theAlphabet[(theAlphabet.find("B") - r2) % 26] + \
                theAlphabet[(theAlphabet.find("Y") - r3 + 0) % 26]
  theDisplay5 = theAlphabet[(theAlphabet.find("Y") - r1) % 26] + \
                theAlphabet[(theAlphabet.find("B") - r2) % 26] + \
                theAlphabet[(theAlphabet.find("Y") - r3 + 3) % 26]

  theDisplay3 = theAlphabet[(theAlphabet.find("Z") - r1) % 26] + \
                theAlphabet[(theAlphabet.find("M") - r2) % 26] + \
                theAlphabet[(theAlphabet.find("Z") - r3 + 1) % 26]
  theDisplay6 = theAlphabet[(theAlphabet.find("Z") - r1) % 26] + \
                theAlphabet[(theAlphabet.find("M") - r2) % 26] + \
                theAlphabet[(theAlphabet.find("Z") - r3 + 4) % 26]
```

Question 114 Positionner l'affichage correctement sur les 6 répliques et réaliser le test d'obtention des 3 lettres en émettant l'hypothèse que celles-ci ne sont pas permutées. Compter les clefs probables trouvées et les afficher. Vérifier que la clef que nous recherchons est parmi celles-ci.

```
  # toujours dans la boucle:
  theMachine1.set_display(theDisplay1)
  theMachine2.set_display(theDisplay2)
  theMachine3.set_display(theDisplay3)
  theMachine4.set_display(theDisplay4)
  theMachine5.set_display(theDisplay5)
  theMachine6.set_display(theDisplay6)
  if ( (theMachine4.process_text(theMachine1.process_text("L")) == "B") and \
       (theMachine5.process_text(theMachine2.process_text("L")) == "O") and \
       (theMachine6.process_text(theMachine3.process_text("O")) == "B") \
     ):
    print("clef probable = " + str([r1, r2, r3]))
    theCount += 1

print("nombre de clefs probables: " + str(theCount))
```

Remarque sur le taux de réussite

Nous avons calculé le taux de réussite en fonction des lettres présentes dans les identifiants que nous attaquons et le nombre de paires de lettres permutées dans le tableau de connexions. Comme dans l'attaque précédente (sans permutation), cette nouvelle attaque ne fonctionne que dans le cas où le second rotor ne tourne pas entre les positions n et $n+3$. Ceci est valable pour les positions n_1, n_2 et n_3, respectivement pour les trois messages attaqués.

La probabilité que le second rotor ne tourne pas est donc de $(26-3)/26$ pour chaque message. Donc pour trois positions, nous avons $((26-3)/26)^3$.

En choisissant astucieusement nos trois messages, nous pouvons également améliorer ce taux de réussite. Pour cela, il suffit de choisir trois messages où la dernière lettre de l'affichage est la même. En effet, comme la position des anneaux est la même pour les trois messages, alors le second rotor tournera après le même nombre de rotations du premier rotor. Notre taux de réussite sera alors toujours de $(26-3)/26$.

Note : si nous attaquons aux indexes $n+1$ et $n+4$ dans un message, alors il faudra que l'affichage soit à la lettre précédente dans l'alphabet. Par exemple, avec ces trois messages, la probabilité que le second rotor ne tourne pas est de $(26-3)/26$:

```
PST I(L)FZ(B)O
YBU (L)YB(O)XT
ZMT G(O)YF(B)U
```

6.4 *Okulary* : les messages spéciaux

Dans la section précédente, nous avons vu comment réduire le nombre de lettres présentes dans l'identifiant que nous attaquons et ainsi augmenter le taux de réussite. Dans cette section, nous allons voir comment augmenter davantage ce taux de réussite.

Question 115 Générer 10 messages en appelant la fonction `procedure()` et observer le contenu des identifiants. Chercher des messages où l'on retrouve les mêmes lettres pour les indexes n et $n+3$ dans le même message.
Voici un exemple où nous forçons une *seed* et utilisons les fonctions de génération aléatoire :

```
random.seed(4)
for i in range(0, 10):
  theClearPos   = ""
  theSessionKey = ""
  for j in range(0, 3):
    theClearPos   += random.sample(list(theAlphabet), k=1)[0]
    theSessionKey += random.sample(list(theAlphabet), k=1)[0]
  print(str(i) + ": ", end="")
  procedure("I III II", "B", [6, 22, 21], "MD RE TX", theClearPos, theSessionKey)
```

Dans cet exemple, nous obtenons deux messages où les lettres `R`, et `E` correspondent à ces critères.

```
print("4: IAZ AD(R)LZ(R) AVDTFQLCNZAOIYXWJHXVWTQHJDTPIAYWKYVU")
print("9: JWY P(E)KU(E)J EKQNMGHWZSZZNJCQFISUCGVNDBWIYIOPBUTP")
```

Une information importante est que la machine Enigma possède une particularité : il peut arriver des cas où **une lettre est substituée par la même autre lettre à une position n et à une position $n+3$**. Dans ces conditions, nous pouvons réduire davantage le nombre de lettres présentes dans l'identifiant que nous attaquons.

Les cryptanalystes du Bureau du chiffre ont baptisé ces messages *okulary* : un mot polonais qui signifie "lunettes". Ce sont les messages où une lettre $\alpha \in [A-Z]$ de l'identifiant, pour une position $n_0 \in [0, 26\times 26\times 26[$, vérifie l'équation suivante :

$$E(n_0,\alpha) = E(n_0+3,\alpha) \tag{13}$$

Question 116 Admettons que nous sommes en possession d'un grand nombre de messages et que nous pouvons trouver 3 identifiants comportant un *okulary* de la même lettre. Pour 0 à 10 paires de lettres permutées sur le tableau de connexions, calculer la probabilité que cette lettre de l'identifiant ne soit pas permutée.

```
print("3 fois 1 lettre en okulary:")
for thePairCount in range(0, 10 + 1):
  print("%2d" % thePairCount + " paires: " + str(
    math.pow((26 - 2*thePairCount) / 26, 1)
  ))
```

Ces valeurs représentent le taux de réussite de notre attaque pour chaque valeurs du nombre de paires de lettres permutées sur le tableau de connexions.

Question 117 Avec une machine Enigma utilisant les rotors dans l'ordre `I III II`, parcourir toutes les positions et, pour chaque lettre de l'alphabet, compter le nombre d'*okulary* que nous avons.

```
theMachine = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B")
theCount = 0
for n in range(0, 26*26*26):
  n3 = int(n % 26)
  n2 = int((n / 26) % 26)
  n1 = int((n / (26*26)) % 26)
  for theLetter in theAlphabet:
    theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[n3])
    alpha = theMachine.process_text(theLetter)
    theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 3) % 26])
    if ( theMachine.process_text(theLetter) == alpha ):
      #print("okulary (" + theLetter + ") pour n = " + str(n))
      theCount += 1
```

Question 118 Nous nous plaçons dans le cas où l'affichage et les lettres tapées par l'opérateur sont choisies de manière aléatoire uniforme. Comme un identifiant est le chiffrement d'une succession de 3 lettres, pour chaque message, quelle est la probabilité que celui-ci contienne un *okulary* ?

```
print("Probabilite d'avoir un okulary: " + str(3*theCount/(26*26*26*26)))
```

A ce stade de l'aventure, nous sommes en possession de plusieurs clefs probables (ordre des rotors et position des anneaux) pour les réglages de la machine Engima mais n'avons pas d'information sur les permutations réalisées par le tableau de connexions. En effet, la Bombe de Rejewski ne donne aucune information à ce sujet.

Pour la suite de l'attaque, les équipes du Bureau du chiffre se partageaient les différentes clefs probables et poursuivaient les travaux manuellement. L'objectif était d'obtenir les réglages du tableau de substitution. Pour cela, une réplique d'Enigma avec les réglages trouvés était utilisée et des messages étaient déchiffrés par itérations successives.

Une solution consistait à faire apparaitre des mots dans la langue du message (en l'occurrence, en allemand) et de tester des permutations sur le tableau de connexions jusqu'à obtenir un texte clair sans erreur. Bien évidemment, il était possible d'attaquer plusieurs messages en même temps.

Une autre solution consistait à déchiffrer les identifiants d'autres messages et réaliser des permutations sur le tableau de connexions pour faire apparaitre une répétition d'un trigramme. En choisissant astucieusement les identifiants, il était possible d'arriver à un résultat rapidement.

6.4.1 Quelques mots sur les permutations

Nous pouvons noter, en attaquant un identifiant de type *okulary*, que l'utilisateur tape deux fois la même lettre et obtient deux fois la même (autre) lettre. De ce fait, il est envisageable d'attaquer le tableau de connexion dans le cas où les lettres présentes dans l'identifiant sont permutées.

Pour cela une solution consiste à tester toutes les permutations possibles pour une seule lettre : celle présente dans les identifiants de type *okulary*. Si nous obtenons la même lettre en sortie des deux machines Enigma, alors nous obtenons une clef probable avec la permutation. Imaginons que l'utilisateur a tapé deux fois la lettre α_0 et que celle-ci présente un *okulary* avec la lettre α_1. Alors, nous pouvons tester toutes les lettres possibles $P(\alpha_1)$ en entrée de la Bombe (pour une position donnée des rotors) et arrêter la Bombe lorsque nous obtenons de nouveau $P(\alpha_1)$ en sortie. La figure 13 schématise cette stratégie permettant d'obtenir un tel oracle.

Cette stratégie a été envisagée par les cryptanalystes du Bureau du chiffre. Cependant, même si celle-ci est valide, elle n'était pas réalisable en termes de puissance de calcul. En effet, afin de tester les 26 possibilités pour $P(\alpha_1)$, il était nécessaire :

— soit de modifier les Bombes Cryptologiques et ainsi multiplier par 26 leur durée de fonctionnement.
— soit construire de nouvelles Bombes Cryptologiques pour en obtenir 26 fois plus et tester une permutation sur chaque Bombe.

Dans son ouvrage *The essential Turing*, Jack Copeland explique que la Bombe de Rejewski fonctionnait pendant une durée approximative de deux heures pour parcourir les $26 \times 26 \times 26$ positions des rotors. Multiplier par 26 cette durée de fonctionnement ne présentait pas une solution viable car la clef de chiffrement changeait toutes les 24 heures. Construire de nouvelles Bombes n'était pas non plus envisageable compte tenu des capacités financières réduites du Bureau du chiffre.

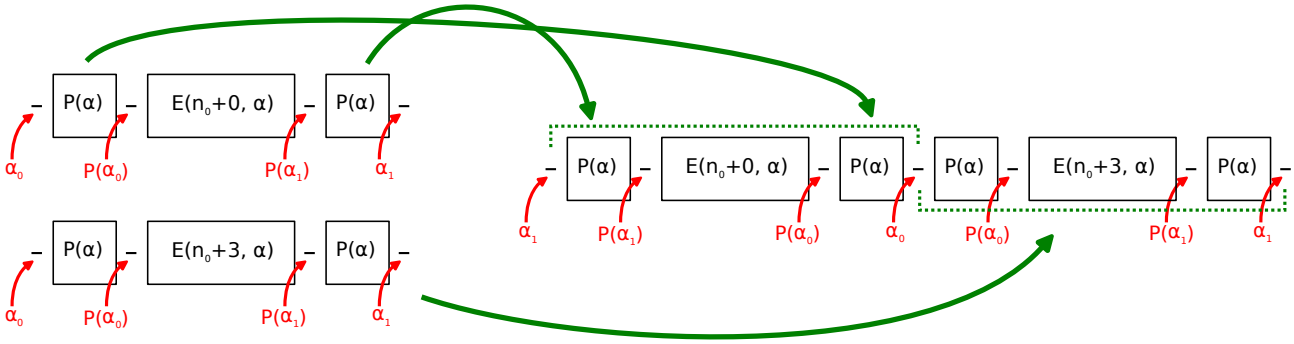

FIGURE 14 – Oracle : connexions de machines Enigma pour un identifiant *okulary*

Cette section met un terme à l'étude de la Bombe Cryptologique de Rejewski. En effet, le 15 décembre 1938, une autre modification apportée à la machine Enigma complique l'application de la Bombe. Les Allemands avaient fourni à leurs opérateurs deux rotors supplémentaires pour compléter les trois d'origine, ce qui rend le décryptage dix fois plus complexe ($!5/!(5-3) = 60$). Construire dix fois plus de Bombes Cryptologiques dépasse les capacités du Biuro Szyfrów : ces nombreuses Bombes auraient coûté quinze fois son budget annuel d'équipement.

Il devient évident que la guerre est imminente et que les ressources financières polonaises sont insuffisantes pour suivre l'évolution du chiffrement avec Enigma. L'état-major polonais et le gouvernement polonais décident d'initier leurs alliés occidentaux aux secrets du décryptage de la machine. Les méthodes polonaises sont révélées aux représentants des services de renseignement français et britanniques lors d'une réunion à Pyry le 25 juillet 1939.

7 La Bombe Cryptologique de Turing/Welchman

Alan Mathison Turing, né le 23 juin 1912 à Londres et mort le 7 juin 1954 à Wilmslow, est un mathématicien et cryptologue britannique, auteur de travaux qui fondent scientifiquement l'informatique. [...]

Durant la Seconde Guerre mondiale, il joue un rôle majeur dans la cryptanalyse de la machine Enigma utilisée par les armées allemandes : l'invention de machines usant de procédés électroniques, les Bombes, fera passer le décryptage à plusieurs milliers de messages par jour. Mais tout ce travail doit forcément rester secret, et ne sera connu du public que dans les années 1970. Après la guerre, il travaille sur un des tout premiers ordinateurs, puis contribue au débat sur la possibilité de l'intelligence artificielle, en proposant le test de Turing.

Source : `https://fr.wikipedia.org/wiki/Alan_Turing`

William Gordon Welchman, né le 15 juin 1906 et mort le 8 octobre 1985, est un mathématicien britannique et cryptologue de la Seconde Guerre mondiale, à Bletchley Park.

Welchman est nommé à la tête de la sixième section des services du chiffre : la Hutte 6. Alan Turing était le chef de la section huit : Hutte 8. La Hutte 6 était chargée de percer le chiffre des machines Enigma de l'Armée de terre (Heer) et de l'Armée de l'air allemandes.

Source : `https://fr.wikipedia.org/wiki/William_Gordon_Welchman`

Dans cette section, nous allons étudier le fonctionnement de la Bombe de Cryptologique d'Alan Turing et Gordon Welchman, qui permet de réaliser une attaque à clair connu sur les communications militaires allemandes interceptées à partir de 1940. Notons que la Bombe de Turing permettait d'obtenir, pour un ordre de rotors donné, une position initiale (image de la différence entre la position des anneaux et des celle des rotors : pas les valeurs exactes de chacune) ainsi que les permutations appliquées sur le tableau de connexions.

7.1 Un peu d'Histoire...

A partir du 15 décembre 1938, **le nombre de rotors que l'on pouvait placer dans la machine Enigma passe de 3 à 5** : `I`, `II`, `III`, `IV` et `V`. Un nouveau réflecteur, le réflecteur `C`, est introduit en 1940 mais fut peu utilisé. Un second nouveau réflecteur, le réflecteur `D`, a été introduit plus tard, sa première utilisation fut observée à partir de janvier 1944.

Fin 1938, après les accords de Munich, la Grande-Bretagne développe ses armements. Turing fait partie des jeunes cerveaux appelés à suivre des cours de chiffre et de cryptanalyse à la Government Code and Cypher School (GC&CS). Juste avant la déclaration de guerre, il rejoint le centre secret de la GC&CS à Bletchley Park. Il est affecté aux équipes chargées du déchiffrement des messages codés avec Enigma. Ce travail profite initialement des percées effectuées par les services secrets polonais du Biuro Szyfrów et du renseignement français au PC Bruno, que Turing visite entre décembre 1939 et les premiers mois de 1940 et d'où il rapporte des copies des feuilles de Zygalski.

Peu après une rencontre à Varsovie (1939) où le Bureau du chiffre polonais explique aux Français et aux Britanniques le câblage détaillé des rotors d'Enigma et la méthode polonaise de décryptage des messages associés, Turing se met au travail sur une approche moins spécifique du problème. En effet, Turing a deux objectifs en tête :

— trouver une attaque qui fonctionne quelque-soit les permutations sur le tableau de connexion.
— trouver une attaque qui fonctionne même si la répétition d'un trigramme est supprimée des procédures de communications.

En janvier 1940, **le nombre de paires de lettres permutées sur le tableau de connexions passe à 10**. En mai 1940, les Allemands **abandonnent la présence de l'identifiant qui constitue la répétition d'un trigramme**. Turing participe aux recherches qui permettent de pénétrer les réseaux de l'armée de terre et de l'aviation. Il conçoit des méthodes mathématiques et des versions améliorées de la Bombe polonaise.

7.2 Architecture de la Bombe

Tout comme la Bombe de Marian Rejewski, la Bombe d'Alan Turing est composée de rotors assemblés pour constituer des répliques d'Enigma. Là où la Bombe de Rejewski possédait une architecture fixe composée de 6 répliques connectées deux à deux (trois fois le schéma représenté sur la figure 10), la Bombe de Turing possède 36 répliques d'Enigma non connectées. Il n'y a donc pas d'architecture à proprement parler mais des répliques à connecter soi-même en fonction de la ou des attaque(s) réalisée(s). Les répliques sont identiques à celle représentée sur la figure 11.

L'idée d'un tel nombre de répliques est de pouvoir, avec seulement une seule Bombe, reproduire les 6 Bombes de Rejewski utilisées en parallèle pour attaquer les 6 ordres possibles des rotors (dans le cas où seulement 3 rotors sont à

placer). Il faut donc 10 Bombes de Turing pour paralléliser 60 attaques sur les 60 ordres possibles des rotors dans le cas où 5 rotors sont à placer (tant que l'identifiant est toujours présent dans les communications allemandes).

L'idée d'avoir des répliques à connecter soi-même est de rendre possible une attaque plus générale, paramétrable sans aucune restriction. En effet, Alan Turing souhaite utiliser la Bombe de manière identique à celle de Rejewski mais également pour réaliser une attaque à clair connu sur n'importe quel message : le système est donc modulaire.

Dans l'ouvrage *The essential Turing* de Jack Copeland, Patrick Mahon explique que la Bombe de Turing fonctionnait pendant une durée approximative de 20 minutes pour parcourir les $26 \times 26 \times 26$ positions des rotors.
La figure 15 montre une photographie de la Bombe de Turing.

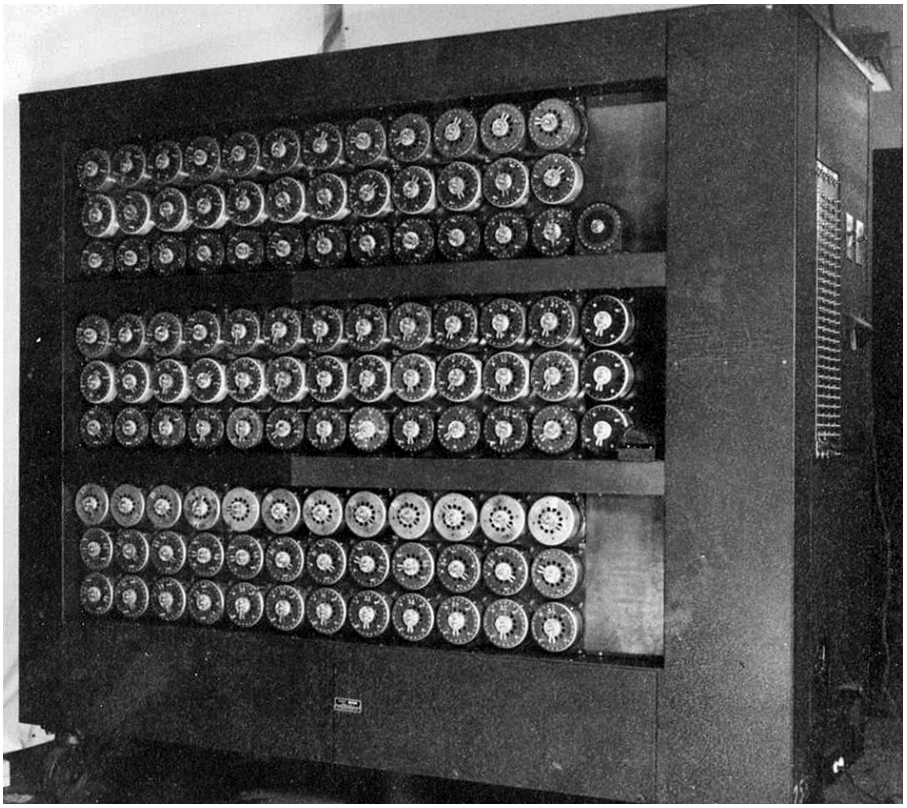

FIGURE 15 – Bombe de Turing/Welchman vue de devant

7.3 Attaque du tableau de connexions

Sur la Bombe de Turing, il est possible de connecter les répliques d'Engima de sorte à obtenir le même comportement que la Bombe de Rejewski. L'ennui est que cette dernière ne permet pas une attaque de l'identifiant lorsque les lettres présentes dans le message chiffré sont permutées. Turing propose une évolution de cette stratégie dans le but d'obtenir les permutations sur le tableau de connexions. L'objectif est de rendre l'attaque de Rejewski résistante aux permutations **sans augmenter la durée de l'attaque**.

Pour procéder à son attaque, Alan Turing repart des travaux de Marian Rejewski lors d'une attaque d'un identifiant *okulary*. En particulier, il réutilise le schéma représenté sur la figure 14. L'idée est de tester toutes les lettres possibles $P(\alpha_1)$ en entrée de la Bombe (pour une position donnée des rotors). Sa proposition est la suivante :

1. nous pouvons connecter les répliques d'Enigma de manière à ce que tester une lettre possible $P(\alpha_1)$ crée une réaction en chaîne pour tester toutes les possibilités.
2. nous pouvons trouver une condition pour identifier la valeur de la lettre $P(\alpha_1)$ après cette réaction en chaîne.
3. nous pouvons trouver un oracle qui nous permet de distinguer si les positions des rotors sont correctes et arrêter la Bombe.

Dans cette section, nous étudions la propositions d'Alan Turing sur l'attaque d'un identifiant.

Question 119 Le script `tp-enigma/08/plugboard.py` contient un squelette prêt pour réaliser l'attaque proposée par Alan Turing sur le tableau de connexions. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/08
debian@myhostname:~/tp-enigma/08$ ./plugboard.py
```

Dans la suite de cette section, nous allons modifier ce script pour réaliser l'attaque.

Question 120 Tout comme Marian Rejewski, nous attaquons plusieurs identifiants *okulary* obtenus avec les mêmes réglages. Cette fois-ci, nous utilisons 10 permutations sur le tableau de connexions.
Instancier une machine Enigma à l'aide des réglages suivants :

- ordre des rotors : `I` puis `III` puis `II`
- réflecteur : `B`
- position des anneaux : 0, 0, 0
- liste des permutations : `AC BJ DM IK LZ ER NO PQ TX SV`

```
theMachine = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B",
   ring_settings=[0, 0, 0], plugboard_settings="AC BJ DM IK LZ ER NO PQ TX SV")
```

Question 121 Cette machine présente un *okulary* sur la lettre `N` pour les positions n suivantes :

- 3403,
- 9885,
- 14287

Vérifier la présence des *okulary* en chiffrant deux fois la lettre `N` pour ces positions n et $n+3$. Attention à bien appliquer l'addition $+3$ seulement sur la position du rotor 3 (cas où le rotor 2 ne tourne pas durant le chiffrement de l'identifiant).

```
for n in [3403, 9885, 14287]:
  n3 = int(n % 26)
  n2 = int((n / 26) % 26)
  n1 = int((n / (26*26)) % 26)
  theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[n3])
  alpha_1 = theMachine.process_text("N")
  theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 3) % 26])
  alpha_0 = theMachine.process_text("N")
  print("n = %5d, N -> " % n + alpha_1 + "; n = %5d + 3, N -> " % n + alpha_0)
print("")
```

L'idée proposée par Alan Turing pour connecter les paires de répliques d'Enigma est de prendre les 26 sorties de la seconde machine et de les connecter aux 26 entrées de la première machine. De ce fait, cela crée une réaction en chaine si la lettre $P(\alpha_1)$ obtenue à la sortie de la seconde machine n'est pas identique à la lettre $P(\alpha_1)$ en entrée. L'oracle pour distinguer si les positions des rotors sont correctes et la condition pour arrêter la Bombe restent encore à déterminer. Le schéma de la figure 16 représente les connexions à réaliser entre deux répliques d'Enigma telles que proposées par Alan Turing. Sur ce schéma, $P(\alpha_1) =$ `C` en entrée de la Bombe.

7.3.1 Une réaction en chaîne

Connecter les répliques d'Enigma comme représenté sur le schéma de la figure 16 crée une réaction en chaîne. En effet, si la lettre $P(\alpha_1)$ obtenue à la sortie de la Bombe n'est pas identique à la lettre $P(\alpha_1)$ en entrée, alors celle-ci est elle aussi injectée dans la Bombe. Le courant qui traverse la Bombe peut donc faire plusieurs tours et traverser la Bombe plusieurs fois jusqu'à atteindre une lettre en entrée où du courant est déjà présent.

Question 122 Pour la position des rotors $n = 3403$ et la lettre choisie en entrée $P(\alpha_1) =$ `N`, connecter deux répliques d'Enigma aux positions n et $n+3$ et observer la lettre $P(\alpha_1)$ obtenue en sortie.

```
theMachine = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B")
n = 3403
P_alpha_1 = "N"
#
n3 = int(n % 26)
n2 = int((n / 26) % 26)
```

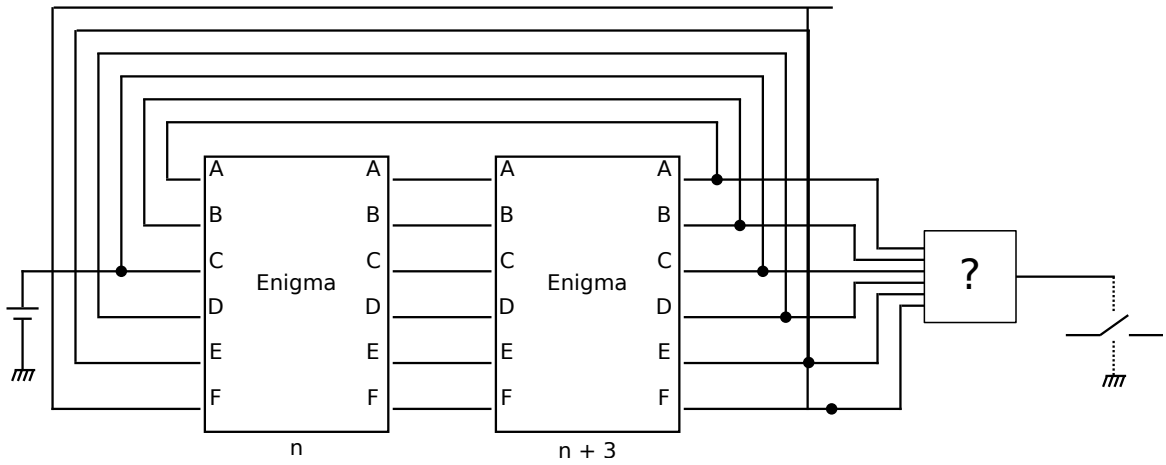

FIGURE 16 – Schéma électrique permettant de détecter les clefs probables

```
n1 = int((n / (26*26)) % 26)
#
theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[n3])
alpha_0 = theMachine.process_text(P_alpha_1)
print("n = %5d,      " % n + P_alpha_1 + " -> " + alpha_0)
#
theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 3) % 26])
P_alpha_1 = theMachine.process_text(alpha_0)
print("n = %5d + 3, " % n + alpha_0 + " -> " + P_alpha_1)
```

Question 123 La lettre $P(\alpha_1)$ obtenue en sortie est différente de `N`. Injecter de nouveau ce résultat dans les deux répliques d'Enigma. Quelle nouvelle lettre $P(\alpha_1)$ obtenons nous en sortie ?

```
theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[n3])
alpha_0 = theMachine.process_text(P_alpha_1)
print("n = %5d,      " % n + P_alpha_1 + " -> " + alpha_0)
#
theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 3) % 26])
P_alpha_1 = theMachine.process_text(alpha_0)
print("n = %5d + 3, " % n + alpha_0 + " -> " + P_alpha_1)
```

Question 124 Réitérer l'expérience avec le nouveau résultat : jusqu'à obtenir une lettre $P(\alpha_1)$ que nous avons déjà injectée en entrée de la Bombe. Combien de tours de Bombe devons nous réaliser pour obtenir une telle lettre ?

Question 125 Nous pouvons réaliser une boucle pour parcourir les différents tours de Bombe et stocker dans une liste les lettres $P(\alpha_1)$ que nous injectons. Toujours pour la position $n = 3403$, généraliser ce comportement à l'aide d'une boucle `while` en Python. Une fois terminée, afficher la liste des lettres $P(\alpha_1)$ injectées.

```
theMachine = EnigmaMachine.from_key_sheet(rotors="I III II", reflector="B")
n = 3403
n3 = int(n % 26)
n2 = int((n / 26) % 26)
n1 = int((n / (26*26)) % 26)
theList  = list()
P_alpha_1 = "N"
while ( P_alpha_1 not in theList ):
  theList.append(P_alpha_1)
  theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[n3])
  alpha_0 = theMachine.process_text(P_alpha_1)
  #
  theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 3) %
  P_alpha_1 = theMachine.process_text(alpha_0)
```

```
theList.sort()
print(theList)
print("")
```

Le comportement que nous modélisons ici représente le fondement de l'attaque du tableau de connexions par Alan Turing. L'idée est d'émettre une hypothèse sur les permutations, par exemple $P(\alpha_1) = \mathtt{N}$, puis d'injecter du courant dans cette lettre à l'entrée de la Bombe :

— si notre hypothèse est correcte, alors nous obtenons $P(\alpha_1) = \mathtt{N}$ à la sortie de la Bombe (exactement comme Marian Rejewski souhaitait le tester).

— si notre hypothèse est incorrecte, alors une réaction en chaîne se crée et toutes les valeurs obtenues pour $P(\alpha_1)$ sont de nouveau injectées en entrée de la Bombe : comme si nous testions ces autres hypothèses.

Dans notre exemple, notre hypothèse $P(\alpha_1) = \mathtt{N}$ était fausse, en n'injectant du courant qu'une seule fois, nous avons testé et pouvons maintenant considérer comme fausses les hypothèses suivantes :

— $P(\alpha_1) = \mathtt{N}$

— $P(\alpha_1) = \mathtt{B}$

— $P(\alpha_1) = \mathtt{G}$

Nous venons donc de répondre à la première problématique : nous sommes capables de **connecter les répliques d'Enigma de manière à ce que tester une lettre possible P(α_1) crée une réaction en chaîne pour tester plusieurs les possibilités**.

7.3.2 Cas d'une hypothèse incorrecte

Cette stratégie de réaction en chaîne permet d'éliminer plusieurs hypothèses en un seul test. Si, pour un programme Python, cela requiert plusieurs itérations ; pour un circuit électrique comme la Bombe de Turing, obtenir un résultat est instantané.

Malheureusement, dans notre exemple précédent, nous n'avons pu éliminer que trois hypothèses fausses d'un coup. Il reste encore 23 hypothèses à tester. Pour palier ce problème, Alan Turing propose d'injecter toutes les hypothèses dans les autres paires de répliques d'Enigma, qui attaquent les deux autres messages *okulary* **sur la même lettre**. En effet, nous avons choisi ces messages tels que nous obtenons un *okulary* sur la même lettre : la lettre N. Nous pouvons donc connecter ensemble les 26 sorties et 26 entrées des trois paires de répliques d'Enigma.

Le schéma représenté sur la figure 17 montre les connexions à appliquer sur 3 paires de répliques d'Enigma, à des positions différentes, pour injecter toutes nos hypothèses dans les 3 paires. Dans notre exemple, les valeurs n_0, n_1 et n_2 sont à choisir telles que :

— $n_0 = 3403$,

— $n_1 = 9885$,

— $n_2 = 14287$

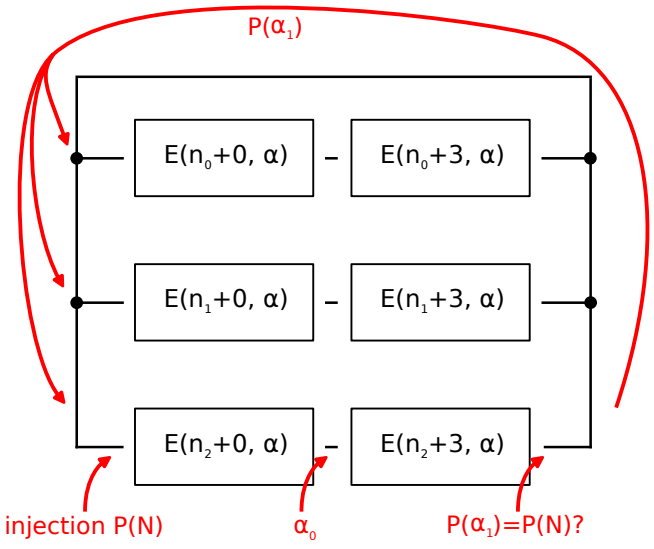

FIGURE 17 – Connexions de machines Enigma pour 3 *okulary*

Question 126 En vous inspirant de la boucle `while` écrite en Python, écrire une fonction qui injecte du courant dans les 3 paires de répliques d'Enigma.
Cette fonction doit être récursive car de nouvelles hypothèses peuvent être découvertes et doivent être ré-injectées dans les 3 paires. Elle prend trois paramètres :

— la lettre $P(\alpha_1)$ injectée,
— un pointeur vers la liste des hypothèses déjà testée,
— la liste des valeurs de n pour les positions des paires de répliques d'Enigma dans chaque boucle.

Egalement, pour plus de compréhension, afficher le résultat de chaque injection.

```
def populate(
  P_alpha_1,     #!< lettre injectee
  thePlugList,   #!< pointeur vers la liste des hypotheses deja testee
  theEngimaList #!< liste des valeurs de n
):
  if ( P_alpha_1 in thePlugList ):
    return  # condition to break the chain of recursion
  else:
    thePlugList.append(P_alpha_1)
    for n in theEngimaList:
      n3 = int(n % 26)
      n2 = int((n / 26) % 26)
      n1 = int((n / (26*26)) % 26)
      theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[n3])
      alpha_0 = theMachine.process_text(P_alpha_1)
      print("n = %5d: E(n, " % n + P_alpha_1 + ") -> " + alpha_0, end="")
      theMachine.set_display(theAlphabet[n1] + theAlphabet[n2] + theAlphabet[(n3 + 3) % 26])
      P_alpha_1 = theMachine.process_text(alpha_0)
      print(" ; E(n+3, " + alpha_0 + ") -> " + P_alpha_1)
      populate(P_alpha_1, thePlugList, theEngimaList)  # recursion
    return
```

Question 127 Testons cette fonction, de nouveau en parcourant la première boucle. Initialiser une liste vide et appeler la fonction avec `N` pour la valeur initiale de $P(\alpha_1)$; dans la liste des valeurs de n, ne choisir que 3403 et vérifier que nous obtenons le même résultat que précédemment.

```
theList = list()
populate("N", theList, [3403])
theList.sort()
print(theList)
```

Question 128 Tester de nouveau, cette fois-ci en ne parcourant que la seconde boucle : pour $n = 9885$.
Quelles sont les nouvelles hypothèses pour $P(\mathtt{N})$ que nous pouvons désormais considérer comme fausses ?

```
theList = list()
populate("N", theList, [9885])
theList.sort()
print(theList)
```

Question 129 Toutes les hypothèses pour $P(\mathtt{N})$ que nous pouvons écarter en s'injectant dans la première boucle peuvent être injectées dans la seconde boucle pour écarter de nouvelles hypothèses. Réciproquement, les hypothèses que nous pouvons écarter en s'injectant dans la seconde boucle peuvent être injectées dans la première.
Appeler de nouveau la fonction, cette fois-ci en parcourant deux boucles en même temps : la première ($n = 3403$) et la seconde ($n = 9885$).
Combien d'hypothèse pouvons nous désormais écarter pour $P(\mathtt{N})$?

```
theList = list()
populate("N", theList, [3403, 9885])
theList.sort()
print(theList)
```

Question 130 Connecter les trois boucles comme représenté sur la figure 17 et observer la liste des hypothèses pour $P(\mathtt{N})$ que nous pouvons écarter en une seule injection.

```
theList = list()
populate("N", theList, [3403, 9885, 14287])
theList.sort()
print(theList)
```

Question 131 Quelle hypothèse semble possible pour $P(\mathtt{N})$? Vérifiez la liste des permutations que nous avons choisies en début de section pour confirmer votre intuition.

Avec cette expérience, nous observons que connecter plusieurs boucles (vers la même lettre) permet de grandement augmenter le nombre d'hypothèses que nous pouvons rejeter. Dans notre exemple, connecter seulement deux boucles suffit à écarter 25 hypothèses pour $P(\mathtt{N})$. Dans certains cas, avoir deux boucles n'est pas suffisant et trois boucles sont nécessaires. Alan Turing a réalisé un travail statistique pour aider les utilisateurs de la Bombe à connecter leurs boucles et ainsi obtenir de meilleurs résultats.

Notons que dans notre exemple, avec trois boucles, nous n'écartons pas la 26^{ime} hypothèse pour $P(\mathtt{N})$. Donc faire apparaître un nombre de boucles supérieur est toujours bénéfique. Egalement, si la Bombe est positionnée à la bonne position (valeur de n pour chaque réplique d'Engima), alors ne pas écarter la 26^{ime} hypothèse suggère que cette hypothèse est correcte.

7.3.3 Cas d'une hypothèse correcte

En injectant du courant dans plusieurs boucles pour émettre une hypothèse incorrecte permet d'écarter de nombreuses nouvelles hypothèses incorrectes. Dans cette section, nous expérimentons l'injection du courant pour émettre une hypothèse correcte.

Question 132 Nous savons que, dans notre exemple, $P(\mathtt{N}) = \mathtt{O}$. Connecter les trois boucles comme représenté sur la figure 17 et injecter la valeur O dans la Bombe en rappelant la fonction précédente.
Combien de tours de boucles sont réalisés et qu'obtenons nous dans la liste d'hypothèses ?

```
theList = list()
populate("O", theList, [3403, 9885, 14287])
theList.sort()
print(theList)
```

Question 133 Compte tenu de nos expériences précédentes, proposer deux conditions pour déterminer la valeur de la lettre $P(\alpha_1)$.

En admettant que la Bombe soit dans la position n correcte, c'est-à-dire que les trois répliques d'Enigma soient aux trois positions correctes, alors nous avons deux possibilités dans notre exemple :

— soit la Bombe nous fournit 25 hypothèses dans la liste,
— soit la Bombe nous fournit 1 seule hypothèse dans la liste.

Ces possibilités définissent des conditions de détermination de la valeur de la lettre $P(\alpha_1)$. La Bombe nous fournira donc toujours la réponse, peu importe que nous ayons émis une hypothèse correcte ou non :

— si la Bombe fournit 25 hypothèses, alors α_1 est permutée avec l'hypothèse restante,
— si la Bombe fournit 1 seule hypothèse, alors α_1 est permutée avec la lettre que nous avons choisie pour notre hypothèse.

Nous venons donc de répondre à la deuxième problématique : nous sommes capables de **trouver une condition pour identifier la valeur de la lettre $P(\alpha_1)$ après une réaction en chaîne**.

7.3.4 Oracle : cas d'une position incorrecte

Nous avons répondu à deux des trois problématiques soulevées par la propositions d'Alan Turing. La dernière problématique consiste à trouver un oracle qui nous permet de distinguer si les positions des rotors sont correctes (valeur de n) et arrêter la Bombe.

Question 134 Toujours à partir de la fonction précédente, connecter une boucle (seulement la première) en émettant l'hypothèse que $P(\mathtt{N}) = \mathtt{O}$. Cette fois-ci, modifier la valeur de $n = 3403$ en $n = 3404$.
Selon la Bombe, l'hypothèse que $P(\mathtt{N}) = \mathtt{O}$ est elle correcte ?

```
theList = list()
populate("O", theList, [3404])
theList.sort()
print(theList)
```

Question 135 Toujours à partir de la fonction précédente et toujours en émettant l'hypothèse que $P(\mathtt{N}) = \mathtt{O}$, connecter deux boucles en même temps où nous avons incrémenté la position n : la première ($n = 3404$) et la seconde ($n = 9886$).
Combien d'hypothèse pouvons nous désormais écarter pour $P(\mathtt{N})$?

```
theList = list()
populate("O", theList, [3404, 9886])
theList.sort()
print(theList)
```

Question 136 Ajouter la troisième boucle où nous avons incrémenté la position n et observer le résultat ($n = 14288$).
De ce comportement, déduire un oracle qui nous permet de distinguer si les positions des rotors sont correctes.

```
theList = list()
populate("O", theList, [3404, 9886, 14288])
theList.sort()
print(theList)
```

Lorsque la Bombe est dans une position n incorrecte, c'est-à-dire que les trois répliques d'Enigma sont à trois positions incorrectes (mais toujours avec le même écart), alors la Bombe nous fournit 26 hypothèses dans la liste. Cela signifie que les 26 hypothèses peuvent être écartées. En d'autres termes, si pour une position n, les 26 hypothèses pour $P(\alpha_1)$ peuvent être écartées, alors aucun réglage n'est possible pour $P(\alpha_1)$ avec cette position n. De ce fait, la position n est forcément incorrecte.

Ici aussi, dans notre exemple, connecter seulement deux boucles suffit à écarter les 26 hypothèses pour $P(\mathtt{N})$. Dans certains cas, avoir deux boucles n'est pas suffisant et trois boucles sont nécessaires. L'ajout d'une troisième boucle ne change pas le résultat : ici aussi, faire apparaître un nombre de boucles supérieur est toujours bénéfique.

Nous avons donc trouvé notre oracle pour arrêter la Bombe : **si, pour chaque lettre, le nombre de permutations écartées est inférieur à 26, alors la Bombe s'arrête** et nous sommes face à une clef probable. Si, pour au moins une lettre, le nombre de permutations écartées est égal à 26, alors nous pouvons écarter la clef testée et la Bombe passe à la position suivante.

Notons que, si le nombre de permutations écartées est compris entre 2 et 24 (inclus), alors nous devons tester une nouvelle hypothèse en injectant de nouveau du courant avant de valider ou invalider la clef probable. Il peut donc arriver que la Bombe s'arrête et que nous devions la re-démarrer après avoir testé et invalidé une clef. C'est le cas où parcourir les boucles avec notre hypothèse ne suffit pas pour conclure.

Ceci met un terme à l'étude du premier objectif d'Alan Turing : trouver une attaque qui fonctionne quelque-soit les permutations sur le tableau de connexion. Pour résumer :

— dans le cas où la sortie d'une réplique d'Enigma doit donner une valeur $P(\alpha_1)$ et que l'entrée d'une autre réplique d'Enigma est $P(\alpha_1)$, alors nous pouvons connecter les 26 sorties de la première réplique aux 26 entrées de la seconde. L'idée est de faire apparaitre une boucle de répliques d'Enigma pour créer une réaction en chaîne et tester plusieurs permutations $P(\alpha_1)$. Nous émettons une hypothèse sur la permutation $P(\alpha_1)$ et injectons du courant dans une seule des 26 connexions ; ceci définit le point de départ de la réaction en chaîne.

— si d'autres répliques d'Enigma doivent avoir $P(\alpha_1)$ en entrée ou en sortie, alors plusieurs boucles peuvent être connectées ensemble pour augmenter le nombre de tests pour $P(\alpha_1)$. Avec trois boucles, les 26 permutations possibles sont souvent testées en une seule injection de courant.

Une fois que nous parvenons à tester les 26 permutations possibles à chaque itération, trois cas se présentent :

1. les 26 permutations sont listées par la Bombe. Dans ce cas, elles peuvent toutes être écartées : la position de la Bombe (valeur de n) est incorrecte.

2. une seule permutation est listée par la Bombe. Dans ce cas, nous avons émis une hypothèse correcte : la position de la Bombe (valeur de n) constitue une clef probable et la valeur de $P(\alpha_1)$ est donnée par notre hypothèse.
3. 25 permutations sont listées par la Bombe. Dans ce cas, elles peuvent être écartées : la position de la Bombe (valeur de n) constitue une clef probable et la valeur de $P(\alpha_1)$ est donnée par la 26^{ime} permutation qui n'est pas listée.

7.4 Attaque à clair connu

Nous avons étudié la problématique issue du premier objectif d'Alan Turing : trouver une attaque qui fonctionne quelque-soit les permutations sur le tableau de connexions. L'objectif de cette section est de répondre à la problématique issue du second objectif : **trouver une attaque qui fonctionne même si la répétition d'un trigramme est supprimée des procédures de communications**.

Pour répondre à cette problématique, une vulnérabilité qui a permis de décrypter les messages chiffrés avec Enigma est la transmission messages stéréotypés, où l'attaquant pouvait "deviner" le contenu. Parmi ces messages stéréotypés étaient les bulletins météo : souvent, ces messages contenaient en clair des textes comme `WETTER FUER DIE NACHT` (*météo pour la nuit*) ou `ZUSTAND OST WAERTIGER KANAL` (*situation dans le canal Est*).

En interceptant ces messages et en devinant des mots présents dans ceux-ci, les cryptanalystes de Bletchley Park ont été en mesure de réaliser des attaques à clair connu. La Bombe de Turing permet de réaliser de telles attaques. Les mots probables, utilisés pour ces attaques, sont baptisés *cribs*, qui signifie "antisèche", en anglais.

Bien évidemment, exactement comme dans la Bombe de Rejewski, l'attaque à clair connu ne fonctionne pas si le rotor 2 a tourné durant la frappe de la *crib* que nous testons. Il est donc parfois nécessaire de réaliser plusieurs attaques à clair connu pour obtenir un résultat satisfaisant.

7.4.1 Position de la *crib*

Après mai 1940, les machines Enigma des opérateurs devaient être configurées avec un réglage global et ne comportaient plus d'identifiant qui constituait la répétition d'un trigramme. Ce réglage stipulait le placement de 3 rotors parmi 5, les positions des anneaux des rotors, l'affichage pour les rotors, ainsi que 10 permutations à appliquer sur le tableau de connexion.

Dans cette section, nous allons déterminer comment une attaque à clair connu était possible sur un message contenant un rapport météo.

Question 137 Le script `tp-enigma/09/crib.py` contient un squelette prêt pour déterminer comment nous allons réaliser notre attaque à clair connu. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/09
debian@myhostname:~/tp-enigma/09$ ./crib.py
```

Nous allons réaliser une attaque à clair connu sur le message chiffré suivant :

```
ZVBLBGHOMSNDRSSGDXPOIKTSWZTQOGXWXGOUGZGXXWIUTWJCXKVTDRFZVYJWIIJFPZODFWAGGHECDGHHHHAAHNWEZRKOODFBFFWU
```

Imaginons, par exemple, que nous possédons une *crib* et que celle-ci indique que le message clair contient, en début de message, le texte `BULLETINXMETEOX` (où la lettre `X` était utilisée pour symboliser un espace). Afin d'illustrer le fait que l'emplacement de la *crib* n'était pas toujours connu, pour le bien de ce TP, nous avons ajouté des caractères aléatoires en début de message : entre 0 et 6. La difficulté pour effectuer notre attaque est donc que nous ne savons pas avec quel texte chiffré nous pouvons faire correspondre le texte clair. Nous devons donc placer la *crib* sur le message chiffré.

L'objectif de cette section est donc de déterminer l'emplacement de notre *crib* dans le message chiffré. Pour cela, nous allons exploiter une vulnérabilité d'Enigma dont nous n'avons pas encore parlé.

Question 138 Instancier une machine Enigma avec les réglages de votre choix. Chiffrer un message composé de 50 fois la lettre `A`.
Que peut on remarquer dans le contenu du texte chiffré ?

```
theMachine = EnigmaMachine.from_key_sheet(rotors="I II III", reflector="B")
theMachine.set_display("AAA")
print(theMachine.process_text("A" * 50))
```

Question 139 Confirmez votre intuition en chiffrant un message composé de $26 \times 25 \times 26$ fois la lettre `A` et en comptant le nombre d'occurrences.

```
theMachine.set_display("AAA")
theResults = theMachine.process_text("A" * 26 * 25 * 26)
thCount    = theResults.count("A")
print("Il y a " + str(thCount) + " fois la lettre: A")
```

Question 140 Vérifier qu'il en est de même pour toutes les lettres de l'alphabet.

```
for theLetter in theAlphabet:
  theMachine.set_display("AAA")
  theResults = theMachine.process_text(theLetter * 26 * 25 * 26)
  theCount   = theResults.count(theLetter)
  print("Il y a " + str(theCount) + " fois la lettre: " + theLetter)
```

Nous pouvons observer que, pour toutes les lettres de l'alphabet, le résultat du chiffrement par la machine Enigma est toujours **une autre lettre** de l'alphabet. Ce comportement peut être observé avec tous les réglages possibles (choix, emplacements et positions des rotors, choix du réflecteur, permutations sur le tableau de connexions). Il est dû à la présence du réflecteur : celui-ci ne peut pas réfléchir une lettre sur elle-même et le chemin pour traverser les 3 rotors dans un sens ne peut pas être utilisé dans l'autre sens. Cela constitue une vulnérabilité de la machine Enigma qui a simplifié le positionnement des *cribs* sur les textes chiffrés lors des attaques à clair connu.

Question 141 Connaissant cette vulnérabilité, nous pouvons déterminer les emplacements à exclure pour notre *crib* dans le message chiffré.
Itérer sur les tailles possibles pour le préfixe aléatoire (entre 0 et 6 caractères) que nous avons volontairement ajouté pour ce TP et afficher le début du message chiffré aligné avec la *crib*. Déduire les emplacements possibles pour celle-ci.

```
theCipher = theCipherText[0:len(theCrib) + 6]
for i in range(0, 7): # the 0 to 6 first characters can be random letters
  print("")
  thePlain = " " * i + theCrib # shift the crib
  theIndex = -1
  for j in range(0, len(thePlain)):
    if ( thePlain[j] == theCipher[j] ):
      theIndex = j
      break

  if ( theIndex != -1 ):
    print(theCipher[:theIndex] + "(" + theCipher[theIndex] + ")" + theCipher[theIndex+1:])
    print(thePlain[:theIndex]  + "(" + thePlain[theIndex]  + ")" + thePlain[theIndex+1:])
  else:
    print(theCipher)
    print(thePlain)
    print("-> Cas possible: i = " + str(i))
```

Cette attaque sur le message chiffré permet d'éliminer les emplacements impossibles pour la *crib* et ainsi déterminer l'emplacement du clair connu. Dans cet exemple, le nombre de caractères aléatoires qui préfixent le message est égal à 5. Nous savons donc que le texte chiffré suivant, avec les réglages d'Engima que nous recherchons, permet d'obtenir `BULLETINXMETEOX` :

```
GHOMSNDRSSGDXPO
```

7.4.2 Faire apparaitre des boucles

Une fois notre *crib* superposée avec le message chiffré, ceci nous permet d'obtenir des correspondances entre les lettres dans le message clair et dans le message chiffré. Ces correspondances sont obtenues en fonction des indexes des lettres dans les deux messages. Nous sommes en possession des correspondances suivantes :

```
BULLETINXMETEOX
GHOMSNDRSSGDXPO
```

Dans le message complet, les 5 premiers caractères sont des caractères aléatoires. Nous allons donc tenter de déterminer les réglages de la machine Enigma à l'index 6 (c'est-à-dire à l'index où débute notre *crib*). Plusieurs Bombes de Turing fonctionnent en parallèle, chacune pour une sélection et des ordres différents pour les rotors. Sur une de ces Bombes, l'ordre est correct.

Appelons n, compris entre 0 et $26 \times 26 \times 26$, la position que nous recherchons lors de la frappe du caractère à l'index 6. Notre attaque à clair connu nous permet de déduire que :

— une frappe de la lettre B à la position n donne un G
— une frappe de la lettre U à la position $n+1$ donne un H
— une frappe de la lettre L à la position $n+2$ donne un O
— etc.

Réciproquement, comme la machine Enigma est symétrique :

— une frappe de la lettre G à la position n donne un B
— une frappe de la lettre H à la position $n+1$ donne un U
— une frappe de la lettre O à la position $n+2$ donne un L
— etc.

Comme nous l'avons vu dans la section précédente, l'attaque du tableau de connexion fonctionne dans le cas où nous faisons apparaitre des boucles. Nous devons donc connecter les 26 sorties de nos répliques d'Enigma dans le but de créer une boucle.

Question 142 Dans l'exemple que nous étudions, une boucle est présente sur la lettre E. En quelles lettres est chiffrée la lettre E de notre *crib* dans le message chiffré que nous attaquons ?

Question 143 Soit n la position d'Enigma lors de la frappe de la lettre B, quels sont les indexes de chiffrement de la lettre E dans notre *crib* ?

Question 144 Dans notre *crib*, quel chiffrement par la machine Enigma permet de fermer une boucle ?

Question 145 Comment connecter les 26 entrées/sorties de plusieurs répliques d'Engima pour créer cette boucle et créer une réaction en chaîne ?

Dans notre exemple, à l'index $n+4$, la lettre E est chiffrée par un S. De même, à $n+12$, la lettre E est chiffrée par un X. Nous pouvons fermer la boucle en considérant le chiffrement de la lettre X par un S à l'index $n+8$.
La lettre E est également chiffrée par un G à l'index $n+10$ mais ceci ne nous permet pas de faire apparaitre une boucle.

```
BULL(E)TIN(X)MET(E)OX
GHOM(S)NDR(S)SGD(X)PO
     4     8    12
```

Le schéma représenté sur la figure 18 montre les connexions des entrées/sorties de plusieurs répliques d'Engima pour construire une boucle qui crée une réaction en chaîne.

Bien évidemment, les répliques d'Engima possèdent 26 entrées/sorties et celles-ci sont toutes connectées. Le schéma représenté sur la figure 19 donne une représentation plus précise des connexions montrées sur la figure 18. Afin de rendre ce schéma lisible, nous avons réduit le nombre d'entrées/sorties des répliques d'Engima : il faut imaginer qu'il y en a 26 à chaque fois.

Cette représentation fournit un complément d'information sur l'architecture de la Bombe. En plus de multiples répliques d'Engima, la Bombe possède 26 bus de connexion : un pour chaque lettre $P(\mathtt{A})$, $P(\mathtt{B})$, $P(\mathtt{C})$, etc. Chaque bus de connexion possède 26 fils : un pour chaque lettre A, B, C, etc. Etablir une connexion entre deux répliques d'Engima, c'est donc connecter 1 bus de 26 fils à chaque fois.

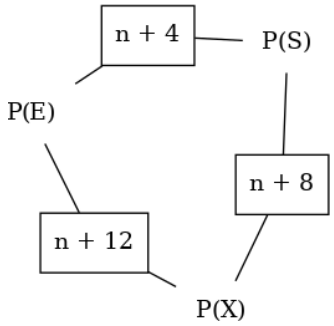

FIGURE 18 – Construction d'une boucle qui crée une réaction en chaîne

Lors de l'émission d'une hypothèse, l'injection du courant se fait dans un seul fil d'un seul bus de connexion. Par exemple, si nous émettons l'hypothèse que $P(\texttt{E}) = \texttt{A}$, alors nous injectons du courant dans le premier fil du bus $P(\texttt{E})$.

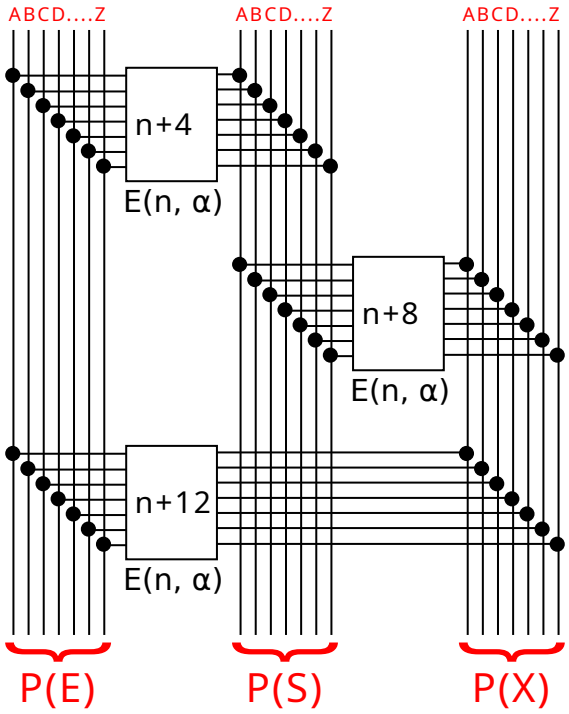

FIGURE 19 – Construction d'une boucle qui crée une réaction en chaîne : schéma plus précis

Exactement comme dans la section précédente, à partir de cette boucle, nous devons créer une réaction en chaîne en faisant traverser du courant plusieurs fois. Pour cela nous pouvons de nouveau créer une fonction récursive. Afin de gagner du temps, pour la suite de ce TP, nous utiliserons une bibliothèque *libbombe* dédiée à la manipulation de la Bombe de Turing. Cette bibliothèque a été conçue spécialement pour ce TP.

Question 146 Le script `tp-enigma/10/loop.py` contient la définition de constantes utiles à l'utilisation de la bibliothèque *libbombe*. Nous allons utiliser ces constantes et les fonctions apportées par la bibliothèque.
Compiler la bibliothèque *libbombe* avec la commande `make`.

```
debian@myhostname:~$ cd ~/tp-enigma/10
debian@myhostname:~/tp-enigma/10$ make
```

Parmi les fonctions fournies par la bibliothèque, la fonction `_bombe.add()` réalise une addition sur un nombre n d'un incrément, mais seulement sur la position du troisième rotor. Cette fonction prend deux paramètres :

— la valeur du nombre n sur lequel nous souhaitons réaliser une addition,
— la valeur de l'incrément à ajouter au rotor 3.

Elle retourne le résultat de l'addition.

Question 147 Appeler la fonction `_bombe.add()` sur une valeur quelconque et observer son comportement. Vérifier que celle-ci réalise bien une addition.

```
print(_bombe.add(42, 3))
```

Question 148 Appeler de nouveau la fonction `_bombe.add()` sur une valeur quelconque, cette fois-ci pour ajouter 26 (la taille de l'alphabet). Vérifier que nous obtenons bien le même nombre car le rotor 3 fait alors un tour complet.

```
print(_bombe.add(42, 26))
```

Une autre fonction fournie par la bibliothèque est la fonction `_bombe.init()`. Cette fonction permet de sélectionner les rotors et réflecteur utilisés dans toutes les répliques d'Engima présentes dans la Bombe. Egalement, cette fonction retire les valeurs de tout courant qui ait pu traverser la Bombe lors d'une réaction en chaîne. Elle prend quatre paramètres :

— un nombre entier faisant référence au premier rotor de la réplique d'Enigma (voir les constantes dans le script pour connaitre les correspondances),
— un nombre entier faisant référence au second rotor,
— un nombre entier faisant référence au troisième rotor,
— un nombre entier faisant référence au réflecteur.

Cette fonction ne retourne rien.

Question 149 Utiliser la fonction `_bombe.init()` pour initialiser la Bombe avec les rotors `I`, `II` et `III` ainsi que le réflecteur `B`. Utiliser les constantes définies dans le script.

```
_bombe.init(ROTOR_I, ROTOR_II, ROTOR_III, REFLECTOR_B)
```

La fonction `_bombe.results()` affiche dans la sortie standard les résultats de l'exécution de la Bombe. Pour chaque bus de connexion $P(\texttt{A})$, $P(\texttt{B})$, etc., elle affiche un vecteur de 26 booléens. Chaque élément du vecteur vaut 1 si du courant traverse le fil, ou 0 si aucun courant ne le traverse. Cette fonction ne prend pas de paramètre et ne retourne rien.

Question 150 Après une initialisation de la Bombe, aucun courant ne traverse aucun fil.
Appeler la fonction `_bombe.results()` et vérifier que nous observons 26 vecteurs composés de 26 fois zéro.

```
_bombe.results()
```

La fonction `_bombe.populate()` permet de choisir l'emplacement d'injection du courant et traverse toutes les répliques d'Enigma qui sont connectées dans la Bombe. C'est une fonction récursive qui possède un comportement similaire à celui que nous avons implémenté dans la section précédente, mais de manière générique. Elle prend deux paramètres :

— un nombre entier faisant référence à la lettre où se trouve le point d'injection ; par exemple, 0 fait référence à $P(\texttt{A})$, 1 fait référence à $P(\texttt{B})$, etc. Cela permet de sélectionner le bus de 26 fils.
— un nombre entier faisant référence à la lettre à laquelle est connecté le point d'injection dans notre hypothèse ; par exemple, 0 fait référence à `A`, 1 fait référence à `B`, etc. Cela permet de sélectionner le fil dans le bus.

Cette fonction ne retourne rien.

Question 151 Appeler la fonction `_bombe.populate()` pour émettre l'hypothèse que $P(\texttt{E}) = \texttt{E}$. Utiliser les constantes définies dans le script. Appeler de nouveau `_bombe.results()` et vérifier que le cinquième booléen du vecteur $P(\texttt{E})$ vaut 1.

```
_bombe.populate(E, E)
_bombe.results()
```

La dernière fonction de la bibliothèque qui nous intéresse est la fonction `_bombe.connexion()`. Cette fonction permet d'établir une connexion d'une réplique d'Enigma entre deux bus de 26 fils, ainsi que de choisir la position n de cette réplique. Elle prend trois paramètres :

- un nombre entier faisant référence à la lettre sur lequel se trouve le bus pour une connexion en entrée de la réplique d'Enigma ; par exemple, 0 fait référence à $P(\texttt{A})$, 1 fait référence à $P(\texttt{B})$, etc.
- un nombre entier faisant référence à la lettre sur lequel se trouve le bus pour la connexion à la sortie de la réplique d'Enigma ; ici aussi, 0 fait référence à $P(\texttt{A})$, 1 fait référence à $P(\texttt{B})$, etc.
- un nombre entier compris entre 0 et $26 \times 26 \times 26$, image de la position n de la réplique d'Enigma.

Cette fonction est susceptible de rapporter une erreur et stopper le programme si la connexion que nous demandons est physiquement impossible. Elle retourne un code d'erreur : 0 si la connexion est correcte, 1 en cas d'erreur.

Question 152 Choisir une position $n = 0$ et réinitialiser la Bombe. A l'aide de la fonction `_bombe.connexion()`, connecter 3 répliques d'Enigma exactement comme sur la figure 19 pour réaliser notre boucle. Injecter du courant en émettant l'hypothèse que $P(\texttt{E}) = \texttt{E}$ et observer le résultat.
Après la réaction en chaîne, quelles sont les hypothèse pour $P(\texttt{E})$ que nous pouvons écarter dans le cas où $n = 0$?

```
_bombe.init(ROTOR_I, ROTOR_II, ROTOR_III, REFLECTOR_B)
n = 0
_bombe.connexion(E, S, _bombe.add(n, 4))
_bombe.connexion(E, X, _bombe.add(n, 12))
_bombe.connexion(S, X, _bombe.add(n, 8))
_bombe.populate(E, E)
_bombe.results()
```

Si tout s'est bien passé, pour $n = 0$, nous devrions pouvoir écarter les hypothèses suivantes pour $P(\texttt{E})$: A, E, G, H, I, R, T, U et V.

Question 153 Contrairement à une attaque sur un identifiant, une attaque à clair connu n'apporte-t-elle pas plus de précisions sur les permutations ? Justifier votre réponse.

Question 154 Ré-itérer l'expérience, cette fois-ci pour $n = 1$ et $n = 2$. Conclure sur la suffisance des connexions que nous avons réalisées dans la Bombe.

```
for n in range(1, 3):
  _bombe.init(ROTOR_I, ROTOR_II, ROTOR_III, REFLECTOR_B)
  _bombe.connexion(E, S, _bombe.add(n, 4))
  _bombe.connexion(E, X, _bombe.add(n, 12))
  _bombe.connexion(S, X, _bombe.add(n, 8))
  _bombe.populate(E, E)
  _bombe.results()
  print("")
```

Contrairement à une attaque sur un identifiant, une attaque à clair connu permet d'écarter des hypothèses sur les permutations d'autres lettres. Comme dans notre boucle nous avons les lettres $P(\texttt{S})$ et $P(\texttt{X})$, alors injecter du courant dans un fil de $P(\texttt{E})$ permet de tester de nouvelles hypothèses. Par exemple, pour $n = 0$, nous pouvons écarter :

- pour $P(\texttt{S})$: A, F, G, K, Q, R, U, W et Y.
- pour $P(\texttt{X})$: D, E, L, O, P, S, T, X et Z.

En faisant varier la valeur de n, nous réalisons des tests exactement comme la Bombe de Turing le faisait. Parfois, comme pour $n = 2$, nous avons de la chance et notre test nous permet d'écarter les 26 hypothèses pour au moins une permutation : ceci est donc notre oracle et nous pouvons conclure que $n = 2$ n'est pas une clef probable.

D'autres fois, comme pour $n = 0$ ou $n = 1$, nous écartons un nombre de permutations supérieur à 1 et inférieur à 26. Nous ne pouvons donc rien conclure sur la valeur de n dans ces cas là. En effet, il est nécessaire d'ajouter d'autres boucles pour augmenter l'effet de la réaction en chaîne.

La fonction `_bombe.test()` réalise un test sur les 26 vecteurs de 26 booléens afin de déterminer si nous devons arrêter la Bombe. Cette fonction ne prend pas de paramètre. Elle retourne 1 dans le cas où aucun vecteur contient 26 fois 1 : c'est le cas où la Bombe doit s'arrêter. Elle retourne 0 dans le cas où au moins un vecteur contient 26 fois 1 : c'est le cas où nous pouvons écarter la valeur n testée.

Question 155 Pour les valeurs $n \in [0:2]$, émettre l'hypothèse que $P(\texttt{E}) = \texttt{E}$ et afficher le résultat de la fonction `_bombe.test()`. Vérifier que pour $n = 2$, la fonction retourne bien 0 et 1 le reste du temps.
Penser à réinitialiser la Bombe et modifier les connexions avec les nouvelles valeurs de n à chaque fois.

```
for n in range(0, 3):
  _bombe.init(ROTOR_I, ROTOR_II, ROTOR_III, REFLECTOR_B)
  _bombe.connexion(E, S, _bombe.add(n, 4))
  _bombe.connexion(E, X, _bombe.add(n, 12))
  _bombe.connexion(S, X, _bombe.add(n, 8))
  _bombe.populate(E, E)
  print("n = " + str(n) + " -> _bombe.test() = " + str(_bombe.test()))
```

7.4.3 Qu'est-ce qu'il y a au menu ?

Une fois la *crib* placée sur le message chiffrée, nous pouvons réaliser une attaque à clair connu pour faire apparaître des boucles. Nous avons trouvé une boucle dans notre attaque mais ceci n'est pas suffisant pour tester les 26 hypothèses possibles pour les permutations des lettres présentes dans la boucle.

Or nous n'avons pas utilisé toutes les lettres présentes dans la *crib* et son chiffré correspondant. D'autres connexions peuvent être établies en fonction de ce contenu. Lors de la définition de l'ensemble des connexions que nous pouvons établir sur la Bombe de Turing, les cryptanalystes de Bletchley Park appelaient ceci "donner quelque chose à manger à la Bombe". Par extension, répondre à cette question était appelé "définir le menu".

Dans cette section, nous allons donc définir un menu qui nous permet de réaliser une attaque à clair connu sur le message chiffré précédent. L'objectif est de déterminer si notre *crib* et son chiffré correspondant sont suffisants pour faire apparaître plusieurs boucles.

Question 156 Prendre du papier et un crayon, schématiser une machine Enigma à la position n qui transforme la lettre `B` en `G`. Faire de même avec une machine Enigma à la position $n+1$ qui transforme la lettre `U` en `H`. Continuer avec toutes les correspondances obtenues à l'aide de notre *crib* pour obtenir un graphe.
Dans le cas où une lettre apparaît plusieurs fois (comme la lettre `L` à $n+2$ et à $n+3$), joindre les apparitions comme montré sur la figure 20.

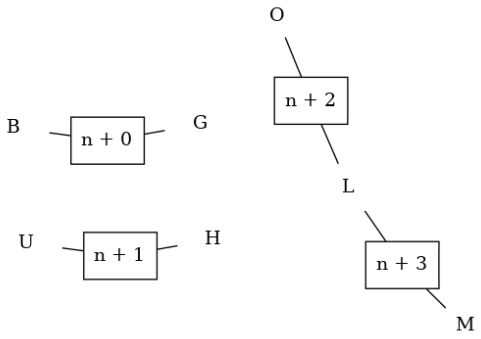

FIGURE 20 – Connexions de répliques d'Enigma, début du menu

Question 157 Le script `tp-enigma/11/menu.py` contient une fonction `drawMenu()` qui permet de dessiner le menu en fonction d'un texte chiffré et de sa *crib* correspondante. Egalement, ce script contient la définition de constantes utiles à l'utilisation de la bibliothèque *libbombe*.
Compiler la bibliothèque *libbombe* avec la commande `make`.

```
debian@myhostname:~$ cd ~/tp-enigma/11
debian@myhostname:~/tp-enigma/11$ make
```

Question 158 La fonction `drawMenu()` crée un fichier image de type *png* où le menu est représenté, elle prend trois paramètres :

— une chaîne de caractère qui représente le texte chiffré,
— une chaîne de caractère qui représente la *crib*,
— une chaîne de caractère qui détermine le nom du fichier *png* à créer.

Pour que celle-ci fonctionne, le texte chiffré et la *crib* doivent comporter le même nombre de caractères.

Appeler cette fonction pour construire le menu à partir de notre texte chiffré et de sa *crib* correspondante. Vérifier que le menu obtenu correspond à celui dessiné à la question précédente.

```
drawMenu("GHOMSNDRSSGDXPO", "BULLETINXMETEOX", "menu.png")
```

La figure 21 montre le résultat final, qui doit être obtenu avec toutes les connexions décrites par notre texte chiffré et sa *crib* correspondante. Notre attaque à clair connu revient donc à connecter les répliques d'Enigma de cette manière.

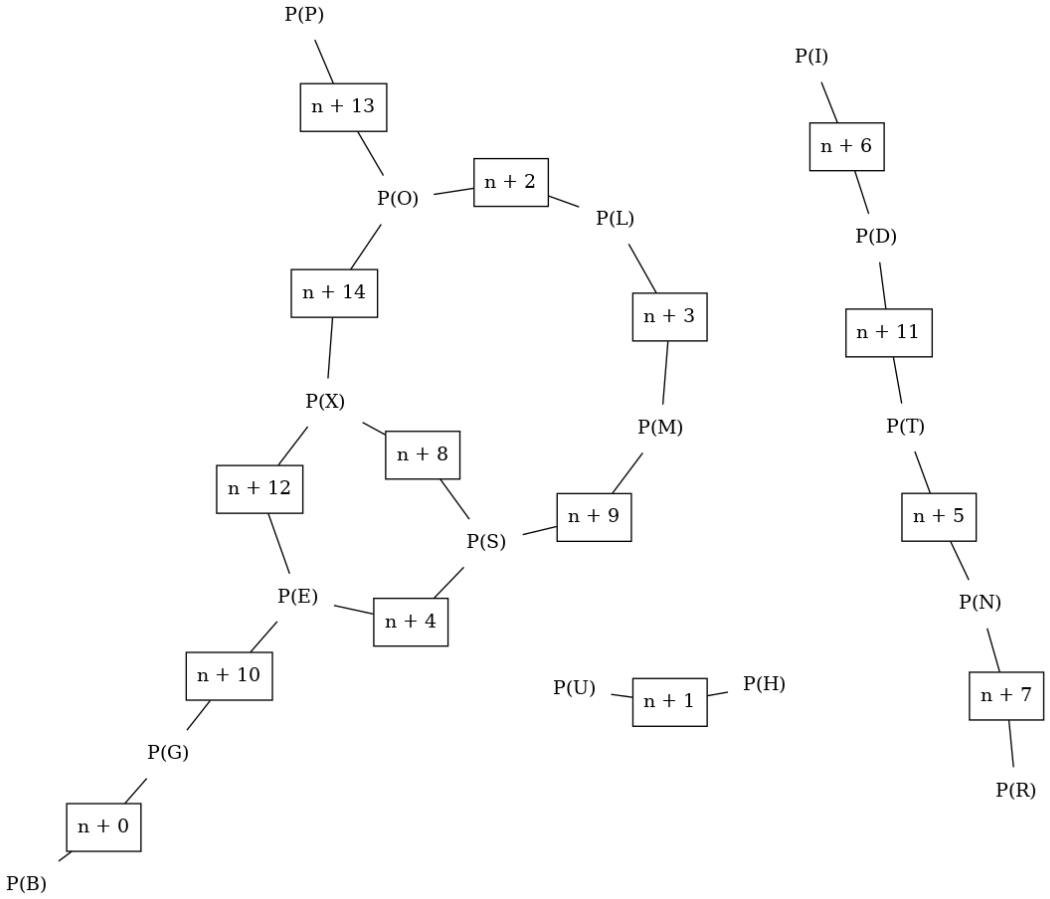

FIGURE 21 – Connexions de répliques d'Enigma, menu complet

Question 159 Comme vu dans la section précédente, en injectant du courant dans un des fils du bus $P(\mathtt{E})$, nous pouvons traverser une boucle en passant par les bus $P(\mathtt{X})$ et $P(\mathtt{S})$.
Toujours en injectant du courant dans un des fils du bus $P(\mathtt{E})$, combien de boucles traversons nous si nous connectons la Bombe comme représenté par le menu de la figure 21 ?

Question 160 Si nous injectons du courant dans un des fils du bus $P(\texttt{X})$ ou $P(\texttt{S})$, combien de boucles traversons nous si nous connectons la Bombe comme représenté par le menu de la figure 21 ?
En déduire si injecter du courant dans un des fils du bus $P(\texttt{E})$ est une bonne stratégie pour augmenter nos chances de réussite.

Si nous injectons du courant dans un des fils du bus $P(\texttt{E})$, alors nous traversons deux boucles (que le courant traverse dans les deux sens) :
- $P(\texttt{E}) \rightarrow P(\texttt{S}) \rightarrow P(\texttt{X}) \rightarrow P(\texttt{E})$
- $P(\texttt{E}) \rightarrow P(\texttt{X}) \rightarrow P(\texttt{O}) \rightarrow P(\texttt{L}) \rightarrow P(\texttt{M}) \rightarrow P(\texttt{S}) \rightarrow P(\texttt{E})$

En revanche, si nous injectons du courant dans un des fils du bus $P(\texttt{X})$ ou $P(\texttt{S})$, alors nous traversons trois boucles. Par exemple, pour $P(\texttt{X})$:
- $P(\texttt{X}) \rightarrow P(\texttt{E}) \rightarrow P(\texttt{S}) \rightarrow P(\texttt{X})$
- $P(\texttt{X}) \rightarrow P(\texttt{O}) \rightarrow P(\texttt{L}) \rightarrow P(\texttt{M}) \rightarrow P(\texttt{S}) \rightarrow P(\texttt{X})$
- $P(\texttt{X}) \rightarrow P(\texttt{O}) \rightarrow P(\texttt{L}) \rightarrow P(\texttt{M}) \rightarrow P(\texttt{S}) \rightarrow P(\texttt{E}) \rightarrow P(\texttt{X})$

De ce fait, pour maximiser nos chances de réussite, injecter du courant dans un des fils du bus $P(\texttt{E})$ n'est pas une une bonne stratégie. Il est préférable d'injecter du courant dans un des fils du bus $P(\texttt{X})$ ou $P(\texttt{S})$.

D'après notre menu, nous avons identifié trois boucles. En injectant astucieusement du courant, nous pouvons provoquer un nombre limité d'arrêts de la Bombe et ainsi détecter des clefs probables. L'objectif est de réduire au maximum ce nombre d'arrêts pour éliminer le plus de clefs possibles.

Nous allons donc utiliser la bibliothèque *libbombe* pour connecter nos répliques d'Enigma tel que défini par notre menu. Plusieurs ordres sont possibles pour les rotors donc plusieurs Bombes doivent fonctionner en parallèle. Afin de gagner du temps dans ce TP, nous nous plaçons sur la Bombe où l'ordre des rotors est celui qui a été utilisé pour chiffrer le message : `II`, `III`, `IV` et le réflecteur `B`.

Question 161 Itérer sur les valeurs de n comprises entre zéro et $26 \times 26 \times 26$, initialiser la Bombe avec l'ordre des rotors correct et connecter les répliques d'Enigma pour former nos trois boucles. Emettre l'hypothèse que $P(\texttt{X}) = \texttt{A}$; compter le nombre d'arrêts de la Bombe (lorsque `_bombe.test()` retourne 1) et afficher les valeurs de n correspondantes.

```
count = 0
for n in range(0, 26*26*26):
  _bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
  _bombe.connexion(X, E, _bombe.add(n, 12))
  _bombe.connexion(E, S, _bombe.add(n, 4))
  _bombe.connexion(S, X, _bombe.add(n, 8))
  _bombe.connexion(X, O, _bombe.add(n, 14))
  _bombe.connexion(O, L, _bombe.add(n, 2))
  _bombe.connexion(L, M, _bombe.add(n, 3))
  _bombe.connexion(M, S, _bombe.add(n, 9))
  _bombe.populate(X, A)
  if ( _bombe.test() ):
    print("n = " + str(n))
    count = count + 1
print("La Bombe stoppe " + str(count) + " fois.")
```

Question 162 Sauvegarder la dernière valeur de n considérée par la Bombe comme une clef probable : $n = 17567$. Réaliser de nouveau les connexions et émettre la même hypothèse. Cette fois-ci, afficher les résultats à l'aide de la fonction `_bombe.results()`.
Cette valeur de n et les permutations trouvées par la Bombe constituent-t-elles vraiment une clef probable ? Pourquoi ?

```
n = 17567
_bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
_bombe.connexion(X, E, _bombe.add(n, 12))
_bombe.connexion(E, S, _bombe.add(n, 4))
_bombe.connexion(S, X, _bombe.add(n, 8))
_bombe.connexion(X, O, _bombe.add(n, 14))
_bombe.connexion(O, L, _bombe.add(n, 2))
_bombe.connexion(L, M, _bombe.add(n, 3))
_bombe.connexion(M, S, _bombe.add(n, 9))
_bombe.populate(X, A)
_bombe.results()
```

Question 163 Selon la Bombe, la valeur $n = 138$ constitue également une clef probable. Réitérer l'expérience avec cette valeur et afficher les résultats.
Cette valeur de n et les permutations trouvées par la Bombe constituent-t-elles une clef probable ? Pourquoi ?

```
n = 138
_bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
_bombe.connexion(X, E, _bombe.add(n, 12))
_bombe.connexion(E, S, _bombe.add(n, 4))
_bombe.connexion(S, X, _bombe.add(n, 8))
_bombe.connexion(X, O, _bombe.add(n, 14))
_bombe.connexion(O, L, _bombe.add(n, 2))
_bombe.connexion(L, M, _bombe.add(n, 3))
_bombe.connexion(M, S, _bombe.add(n, 9))
_bombe.populate(X, A)
_bombe.results()
```

L'expérience que nous venons de mener constitue le fondement de l'attaque à clair connu par Alan Turing. L'idée est de connecter plusieurs boucles obtenues depuis une *crib* et son message chiffré correspondant. Ensuite, nous émettons une hypothèse sur la permutation d'une lettre présente dans un maximum de boucles et nous injectons du courant dans le fil du bus correspondant. La Bombe s'arrête lorsque, pour aucun des 26 bus, le courant parcours les 26 fils (toutes les hypothèses de permutation sont rejetées). Dans notre exemple, la Bombe s'arrête 714 fois.

Certains de ces arrêts sont ne sont pas valables et sont des effets de bord. Nous pouvons nous en apercevoir en observant les permutations suggérées par la Bombe. Par exemple, pour la valeur $n = 17567$, la Bombe nous suggère que $P(\texttt{M}) = \texttt{P}$ et que $P(\texttt{O}) = \texttt{P}$. Or ceci est impossible, les permutations sont symétriques.

D'autres arrêts sont valables et la Bombe ne permet pas d'observer des permutations impossibles. Ceci est le cas pour la valeur $n = 138$. La clef fournie par la Bombe (valeur de n et une liste de permutations) constitue une clef probable et doit être testée sur la véritable machine Enigma.

Dans son ouvrage *The essential Turing*, Jack Copeland explique que la Bombe de Turing fonctionnait pendant une durée approximative de vingt minutes pour parcourir les $26 \times 26 \times 26$ positions des rotors. Ce temps été rallongé par les différents arrêts. A chaque arrêt, l'opérateur observait si l'arrêt était valable ou non puis redémarrait la Bombe depuis sa position actuelle. Si l'arrêt fournissait une clef probable, alors l'opérateur la testait sur une véritable machine Enigma pendant que la Bombe continuait de fonctionner.

Sur notre Bombe, nous avons connecté sept répliques d'Enigma pour construire nos trois boucles. Or, notre *crib* nous fournit des informations supplémentaires et nous pouvons connecter des répliques d'Enigma supplémentaires. Nous pouvons séparer ces répliques en deux groupes :

— celles qui sont connectées aux boucles : $E(n+13,\alpha)$, $E(n+10,\alpha)$ et $E(n+0,\alpha)$;
— et celles qui ne sont pas connectées aux boucles : $E(n+1,\alpha)$, $E(n+6,\alpha)$, $E(n+11,\alpha)$, $E(n+5,\alpha)$ et $E(n+7,\alpha)$.

Question 164 Commençons par les répliques connectées aux boucles : $E(n+13,\alpha)$, $E(n+10,\alpha)$ et $E(n+0,\alpha)$.
D'intuition, ajouter ces connexions à la Bombe va-t-il nous permettre de rejeter des clefs probables supplémentaires lors d'arrêts qui ne sont pas valables ?
Toujours d'intuition, ajouter ces connexions à la Bombe va-t-il réduire le nombre d'arrêts de la Bombe ?

Question 165 Choisir la valeur de $n = 138$ et réitérer l'expérience précédente où nous affichons les résultats. Toujours émettre l'hypothèse que $P(\texttt{X}) = \texttt{A}$. Cette fois-ci, inclure les connexions supplémentaires :

— entre $P(\texttt{O})$ et $P(\texttt{P})$ avec $E(n+13,\alpha)$,
— entre $P(\texttt{E})$ et $P(\texttt{G})$ avec $E(n+10,\alpha)$,
— entre $P(\texttt{G})$ et $P(\texttt{B})$ avec $E(n+0,\alpha)$.

Cette fois-ci, pouvons nous rejeter la clef probable proposée par la Bombe ? Pourquoi ?

```
n = 138
_bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
_bombe.connexion(X, E, _bombe.add(n, 12))
_bombe.connexion(E, S, _bombe.add(n, 4))
_bombe.connexion(S, X, _bombe.add(n, 8))
_bombe.connexion(X, O, _bombe.add(n, 14))
_bombe.connexion(O, L, _bombe.add(n, 2))
_bombe.connexion(L, M, _bombe.add(n, 3))
```

```
_bombe.connexion(M, S, _bombe.add(n, 9))
# more replicas:
_bombe.connexion(O, P, _bombe.add(n, 13))
_bombe.connexion(E, G, _bombe.add(n, 10))
_bombe.connexion(G, B, _bombe.add(n, 0))
_bombe.populate(X, A)
_bombe.results()
```

Question 166 Toujours avec les mêmes connexions, itérer sur les valeurs de n comprises entre zéro et $26 \times 26 \times 26$ et émettre l'hypothèse que $P(\texttt{X}) = \texttt{A}$. Compter le nombre d'arrêts de la Bombe (lorsque `_bombe.test()` retourne 1). Ce nombre est il réduit par rapport à l'expérience précédente (avec seulement les boucles) ? Pourquoi ?

```
count = 0
for n in range(0, 26*26*26):
  _bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
  _bombe.connexion(X, E, _bombe.add(n, 12))
  _bombe.connexion(E, S, _bombe.add(n, 4))
  _bombe.connexion(S, X, _bombe.add(n, 8))
  _bombe.connexion(X, O, _bombe.add(n, 14))
  _bombe.connexion(O, L, _bombe.add(n, 2))
  _bombe.connexion(L, M, _bombe.add(n, 3))
  _bombe.connexion(M, S, _bombe.add(n, 9))
  _bombe.connexion(O, P, _bombe.add(n, 13))
  _bombe.connexion(E, G, _bombe.add(n, 10))
  _bombe.connexion(G, B, _bombe.add(n, 0))
  _bombe.populate(X, A)
  if ( _bombe.test() ):
    count = count + 1
print("La Bombe stoppe " + str(count) + " fois.")
```

Ajouter de nouvelles connexions à la Bombe, lorsque celles-ci sont connectées aux boucles, permet d'éliminer de nouvelles clefs probables lorsque la Bombe s'arrête. En effet, ceci permet d'obtenir de nouvelles permutations pour les lettres présentes dans la *crib*. Si une nouvelle permutation obtenue entre en conflit avec une permutation que nous suggère la Bombe depuis les boucles, alors nous pouvons rejeter la clef et nous n'avons pas besoin de la tester.

En revanche, ajouter de telles connexions ne change pas le nombre d'arrêts de la Bombe. En effet, pour réduire le nombre d'arrêts, il faut augmenter le nombre de boucles. Et rajouter ces connexions ne crée pas de boucle supplémentaire.

Question 167 Passons ensuite aux répliques non connectées aux boucles : $E(n+1,\alpha)$, $E(n+6,\alpha)$, $E(n+11,\alpha)$, $E(n+5,\alpha)$ et $E(n+7,\alpha)$.
Nous savons que celles-ci ne créeront pas de boucle supplémentaire, donc le nombre d'arrêts ne sera pas réduit. D'intuition, ajouter ces connexions à la Bombe va-t-il nous permettre de rejeter des clefs probables supplémentaires lors d'arrêts qui ne sont pas valables ?

Question 168 Choisir la valeur de $n = 138$ et réitérer l'expérience précédente où nous affichons les résultats. Toujours émettre l'hypothèse que $P(\texttt{X}) = \texttt{A}$. Cette fois-ci, inclure les connexions supplémentaires :

- entre $P(\texttt{U})$ et $P(\texttt{H})$ avec $E(n+1,\alpha)$,
- entre $P(\texttt{I})$ et $P(\texttt{D})$ avec $E(n+6,\alpha)$,
- entre $P(\texttt{D})$ et $P(\texttt{T})$ avec $E(n+11,\alpha)$.
- entre $P(\texttt{T})$ et $P(\texttt{N})$ avec $E(n+5,\alpha)$.
- entre $P(\texttt{N})$ et $P(\texttt{R})$ avec $E(n+7,\alpha)$.

Le nombre de permutations suggérées par la Bombe a-t-il augmenté ? Pourquoi ?

```
n = 138
_bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
_bombe.connexion(X, E, _bombe.add(n, 12))
_bombe.connexion(E, S, _bombe.add(n, 4))
_bombe.connexion(S, X, _bombe.add(n, 8))
_bombe.connexion(X, O, _bombe.add(n, 14))
_bombe.connexion(O, L, _bombe.add(n, 2))
_bombe.connexion(L, M, _bombe.add(n, 3))
_bombe.connexion(M, S, _bombe.add(n, 9))
```

```
_bombe.connexion(O, P, _bombe.add(n, 13))
_bombe.connexion(E, G, _bombe.add(n, 10))
_bombe.connexion(G, B, _bombe.add(n, 0))
# more replicas:
_bombe.connexion(U, H, _bombe.add(n, 1))
_bombe.connexion(I, D, _bombe.add(n, 6))
_bombe.connexion(D, T, _bombe.add(n, 11))
_bombe.connexion(T, N, _bombe.add(n, 5))
_bombe.connexion(N, R, _bombe.add(n, 7))
_bombe.populate(X, A)
_bombe.results()
```

Ajouter de nouvelles connexions à la Bombe, lorsque celles-ci ne sont pas connectées aux boucles, n'a aucun effet sur le fonctionnement de la Bombe de Turing. En effet, le courant traverse les fils des bus que nous avons connecté. Or connecter de nouvelles répliques entre elles, sans les connecter à la source de courant, ne change pas le circuit électrique qui a été précédemment construit.

En conséquence, dans la Bombe de Turing, nous venons de réaliser une attaque à clair connu sur les correspondances suivantes :

```
BULLETINXMETEOX
GHOMSNDRSSGDXPO
```

Or, ceci revient au même que de réaliser une attaque à clair connu sur les correspondances suivantes (où un point est juste une lettre ignorée) :

```
B.LLE...XME.EOX
G.OMS...SSG.XPO
```

Dans la Bombe de Turing, chaque étage comportait 12 répliques d'Enigma à connecter soi-même. Afin de maximiser les chances de réussite, il fallait réaliser un menu et le consulter pour sélectionner au maximum 12 des correspondances de lettres. Dans notre exemple, la *crib* possède 14 lettres et seulement 10 sont utiles pour notre attaque à clair connu.

Le code suivant permet d'implémenter notre attaque à clair connu à l'aide de la Bombe de Turing. Une interaction de la part de l'utilisateur est requise, à chaque arrêt de la Bombe, pour la faire repartir. Ce code requiert donc 714 interactions (`ctrl + C` pour tuer le processus).

```
for n in range(0, 26*26*26):
  _bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
  _bombe.connexion(X, E, _bombe.add(n, 12))
  _bombe.connexion(E, S, _bombe.add(n, 4))
  _bombe.connexion(S, X, _bombe.add(n, 8))
  _bombe.connexion(X, O, _bombe.add(n, 14))
  _bombe.connexion(O, L, _bombe.add(n, 2))
  _bombe.connexion(L, M, _bombe.add(n, 3))
  _bombe.connexion(M, S, _bombe.add(n, 9))
  _bombe.connexion(O, P, _bombe.add(n, 13))
  _bombe.connexion(E, G, _bombe.add(n, 10))
  _bombe.connexion(G, B, _bombe.add(n, 0))
  _bombe.populate(X, A)
  if ( _bombe.test() ):
    _bombe.results()
    print("n = " + str(n))
    input("Press Enter to continue...")
```

7.4.4 Version 2 : *Here comes a new challenger !*

William Gordon Welchman, né le 15 juin 1906 et mort le 8 octobre 1985, était un mathématicien britannique et un cryptographe de la Seconde Guerre mondiale, à Bletchley Park.

Welchman a conçu une amélioration de l'architecture de l'appareil électromécanique de décryptage, la "Bombe Cryptologique" de Turing. Cette amélioration, connue sous l'appellation de "*diagonal board*", a rendu l'appareil plus efficace dans l'attaque des messages chiffrés à l'aide de la machine Enigma allemande. Ces "Bombes" ont été les principaux outils du décryptage d'Enigma, pendant la guerre.

Source : `https://fr.wikipedia.org/wiki/William_Gordon_Welchman`

Le modèle de la Bombe que nous avons étudié dans la section précédente est le modèle initial, conçu par Alan Turing en 1939. Ce modèle fut construit et livré à Bletchley Park en mars 1940. La faiblesse de ce modèle est le nombre conséquent d'arrêts de la Bombe. En effet, ce nombre est proportionnel au nombre de boucles dans le câblage de la Bombe. Nous observons plusieurs centaines d'arrêts pour un câblage avec trois boucles.

En 1940, le mathématicien Gordon Welchman travaille avec Alan Turing sur une amélioration de la Bombe pour augmenter de manière drastique le nombre de boucles présentes dans câblage avec la même *crib*. Un nouveau modèle est conçu et livré à Bletchley Park en août 1940. Dans cette section, nous étudions le principe de fonctionnement de cette amélioration.

L'idée proposée par Gordon Welchman est d'exploiter l'aspect symétrique du tableau de connexion. Pour rappel, une permutation échange les lettres deux à deux :

$$\forall\, \alpha, \beta \in [A-Z],\ P(\alpha) = \beta \iff P(\beta) = \alpha$$

De ce fait, si la Bombe de Turing teste une hypothèse sur une permutation, par exemple $P(\mathtt{X}) = \mathtt{A}$, alors nous pouvons aussi tester $P(\mathtt{A}) = \mathtt{X}$. En conséquence, si notre *crib* ou son chiffré correspondant contient les lettres `X` et `A`, alors nous injectons du courant à deux emplacements. Ceci est d'autant plus efficace que cette double injection peut être répercutée à chaque parcours de réplique d'Enigma, à chaque tour de boucle, augmentant de manière exponentielle le nombre d'injections.

Avec cette idée, Gordon Welchman conçut une extension à la Bombe de Turing pour :

— connecter le fil `B` du bus P(`A`) avec le fil `A` du bus P(`B`)
— connecter le fil `C` du bus P(`A`) avec le fil `A` du bus P(`C`)
— connecter le fil `D` du bus P(`A`) avec le fil `A` du bus P(`D`)
— etc.

De la même manière, ceci est appliqué à tous les bus P(`B`), P(`C`), etc. Cette extension est baptisée *diagonal board*, en raison des connexions en diagonale qu'elle ajoutait aux 26 bus. Un schéma simplifié, pour les bus P(`A`), P(`B`), P(`C`) et P(`D`) est représenté sur la figure 22. Bien entendu, des connexions supplémentaires sont présentes pour les lettres suivantes.

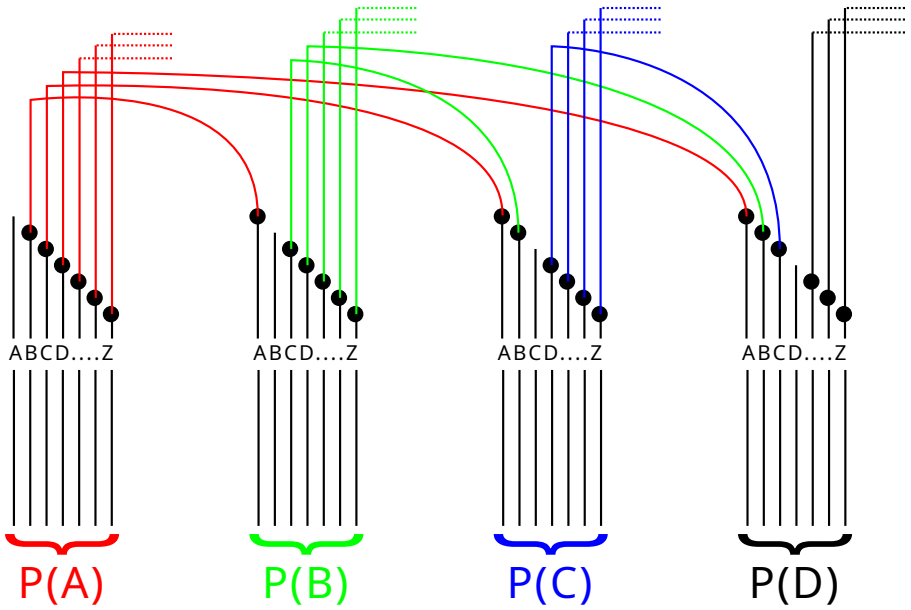

FIGURE 22 – Extension de la Bombe de Turing par Gordon Welchman : le *diagonal board*

Lorsque la nouvelle version de la Bombe, implémentant le *diagonal board*, fût livrée en août 1940, celle-ci fût baptisée *spider* (l'araignée, en anglais). En effet, le grand nombre de connexions ajoutées par le *diagonal board* était semblable à une toile d'araignée. Les cryptanalystes de Bletchley Park continuèrent néanmoins de l'appeler Bombe et c'est le nom qui a été retenu.

La bibliothèque *libbombe* implémente le *diagonal board*. Dans les sections précédentes, nous avons simplement désactivé son fonctionnement lors de la compilation.

Question 169 Le script `tp-enigma/12/diagboard.py` contient la définition de constantes utiles à l'utilisation de la bibliothèque *libbombe*. Nous allons utiliser ces constantes et les fonctions apportées par la bibliothèque.
Compiler la bibliothèque *libbombe* en activant le fonctionnement du *diagonal board* avec la commande `make`.

```
debian@myhostname:~$ cd ~/tp-enigma/12
debian@myhostname:~/tp-enigma/12$ make
```

Question 170 Comme dans la section précédente, itérer sur les valeurs de n comprises entre zéro et $26 \times 26 \times 26$, initialiser la Bombe avec l'ordre des rotors correct et connecter les répliques d'Enigma. Ne pas connecter les répliques d'Enigma non-connectées aux boucles sur le menu (voir la figure 21).
Emettre l'hypothèse que $P(\texttt{X}) = \texttt{A}$ et compter le nombre d'arrêts de la Bombe (lorsque `_bombe.test()` retourne 1). De combien ce nombre est-il réduit par rapport à la Bombe de Turing sans le *diagonal board* ?

```
count = 0
for n in range(0, 26*26*26):
  _bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
  _bombe.connexion(X, E, _bombe.add(n, 12))
  _bombe.connexion(E, S, _bombe.add(n, 4))
  _bombe.connexion(S, X, _bombe.add(n, 8))
  _bombe.connexion(X, O, _bombe.add(n, 14))
  _bombe.connexion(O, L, _bombe.add(n, 2))
  _bombe.connexion(L, M, _bombe.add(n, 3))
  _bombe.connexion(M, S, _bombe.add(n, 9))
  _bombe.connexion(O, P, _bombe.add(n, 13))
  _bombe.connexion(E, G, _bombe.add(n, 10))
  _bombe.connexion(G, B, _bombe.add(n, 0))
  _bombe.populate(X, A)
  if ( _bombe.test() ):
    count = count + 1
print("La Bombe stoppe " + str(count) + " fois.")
```

Question 171 Admettons que, pour une valeur de n donnée, pour un tour de boucle, nous obtenons $P(\texttt{X}) = \texttt{P}$. A partir du menu représenté sur la figure 21, quel nouveau tour de boucle est ajouté dans ce cas là ?
Justifier la réduction du nombre d'arrêts de la Bombe.

Question 172 Dans notre câblage de la Bombe, nous n'avons pas ajouté les répliques non connectées aux boucles : $E(n+1,\alpha)$, $E(n+6,\alpha)$, $E(n+11,\alpha)$, $E(n+5,\alpha)$ et $E(n+7,\alpha)$.
D'intuition, lors de la présence du *diagonal board*, ajouter ces connexions à la Bombe va-t-il nous permettre de réduire d'avantage le nombre d'arrêts ? Justifier votre réponse.

Avec le même câblage que dans la section précédente, le *diagonal board* nous permet de réduire grandement le nombre d'arrêts de la Bombe : de 714 à 30. Ceci est dû au fait que le *diagonal board* ajoute des boucles dans le câblage.

Cependant, ces boucles ne sont pas apparentes car elles changent à chaque itération : elles dépendent de l'hypothèse émise lors de l'injection du courant et des nouvelles hypothèses testées à chaque tour de boucle. Par exemple, si pour une valeur de n donnée, pour un tour de boucle, nous obtenons $P(\texttt{X}) = \texttt{P}$, alors nous ajoutons au menu la boucle suivante :

— $(P(\texttt{X}) = P) \rightarrow P(\texttt{O}) \rightarrow (P(\texttt{P}) = \texttt{X}) \rightarrow (P(\texttt{X}) = P)$

Le schéma de la figure 23 représente cette nouvelle boucle, ajoutée seulement dans le cas où nous testons $P(\texttt{X}) = \texttt{P}$.

A priori, avec cette stratégie, nous pouvons également parcourir les répliques d'Enigma qui ne sont pas connectées aux boucle initiales (au point d'injection du courant). De ce fait, le *diagonal board* permet de réduire le nombre d'arrêts si nous ajoutons ces autres répliques.

Question 173 Réitérer l'expérience précédente où nous faisons varier les valeurs de n comprises entre zéro et $26 \times 26 \times 26$. Cette fois-ci, inclure les connexions supplémentaires :

— entre $P(\texttt{U})$ et $P(\texttt{H})$ avec $E(n+1,\alpha)$,
— entre $P(\texttt{I})$ et $P(\texttt{D})$ avec $E(n+6,\alpha)$,
— entre $P(\texttt{D})$ et $P(\texttt{T})$ avec $E(n+11,\alpha)$.
— entre $P(\texttt{T})$ et $P(\texttt{N})$ avec $E(n+5,\alpha)$.

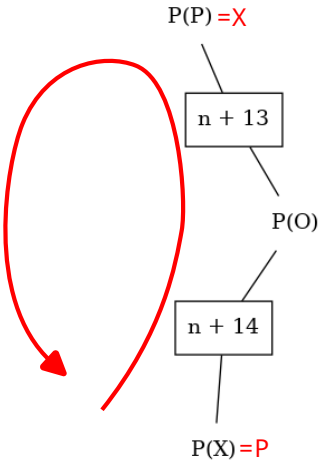

FIGURE 23 – *Diagonal board* : nouvelle boucle dans le cas où $P(\mathtt{X}) = \mathtt{P}$

— entre $P(\mathtt{N})$ et $P(\mathtt{R})$ avec $E(n+7, \alpha)$.

Emettre l'hypothèse que $P(\mathtt{X}) = \mathtt{A}$ et compter le nombre d'arrêts de la Bombe (lorsque `_bombe.test()` retourne 1). De combien ce nombre est-il réduit par rapport à la Bombe de Turing sans le *diagonal board* ?

```
count = 0
for n in range(0, 26*26*26):
  _bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
  _bombe.connexion(X, E, _bombe.add(n, 12))
  _bombe.connexion(E, S, _bombe.add(n, 4))
  _bombe.connexion(S, X, _bombe.add(n, 8))
  _bombe.connexion(X, O, _bombe.add(n, 14))
  _bombe.connexion(O, L, _bombe.add(n, 2))
  _bombe.connexion(L, M, _bombe.add(n, 3))
  _bombe.connexion(M, S, _bombe.add(n, 9))
  _bombe.connexion(O, P, _bombe.add(n, 13))
  _bombe.connexion(E, G, _bombe.add(n, 10))
  _bombe.connexion(G, B, _bombe.add(n, 0))
  # more replicas:
  _bombe.connexion(U, H, _bombe.add(n, 1))
  _bombe.connexion(I, D, _bombe.add(n, 6))
  _bombe.connexion(D, T, _bombe.add(n, 11))
  _bombe.connexion(T, N, _bombe.add(n, 5))
  _bombe.connexion(N, R, _bombe.add(n, 7))
  _bombe.populate(X, A)
  if ( _bombe.test() ):
    count = count + 1
print("La Bombe stoppe " + str(count) + " fois.")
```

Question 174 Le nombre de clefs probables obtenues est suffisamment réduit pour afficher les résultats à chaque arrêt. Afficher la valeur de n et les résultats des permutations pour chaque clef probable.
Quelle(s) clef(s) probable(s) pouvons nous rejeter sans avoir à tester ? Pourquoi ?

```
  # dans la boucle for:
  if ( _bombe.test() ):
    _bombe.results()
    print("n = " + str(n))
    print("")
```

Dans notre exemple, avec les 14 répliques d'Engima connectées selon le menu et la présence du *diagonal board*, nous n'obtenons que 2 clefs probables. L'une d'entre elle peut être rejetée car la Bombe suggère que $P(\mathtt{L}) = \mathtt{A}$ et $P(\mathtt{R}) = \mathtt{A}$.

Ce cas se présente car le *diagonal board* ne traite pas le cas où une suggestion de permutation est obtenue si 25 hypothèses sur 26 sont rejetées. En effet, le *diagonal board* rapplique ces 25 hypothèses dans mais ceci ne permet pas de tester les 26 hypothèses pour $P(\texttt{A})$. La Bombe seule ne peut donc pas rejeter la clef probable obtenue.

Question 175 Concentrons nous sur la clef probable restante. Quelles sont les permutations suggérées par la Bombe ? Proposer une solution pour obtenir facilement de nouvelles permutations.

Sur la dernière clef probable, la Bombe de Turing/Welchman nous propose quatre paires de permutations sur le tableau de connexion : `BP FG NX OY`. La Bombe indique également que les lettres `E`, `L`, `M` et `S` ne sont pas permutées.

Une solution pour obtenir facilement de nouvelles permutations et de ne pas modifier la position de la Bombe (garder la même valeur de n) mais modifier notre hypothèse sur le point d'injection du courant. De ce fait, nous devrions toujours obtenir un arrêt, cette fois-ci avec un seul fil où le courant est présent (pas 25) dans le bus de chaque lettre. Ceci permet en effet de traverser de nouvelles répliques d'Enigma dans le menu, pour les lettres présentes dans celui-ci mais absentes des permutations suggérées.

Question 176 Toujours pour la même valeur de n, émettre l'hypothèse correcte pour $P(\texttt{X})$; lister les permutations obtenues et les lettres non permutées.

```
n = 11644
_bombe.init(ROTOR_II, ROTOR_III, ROTOR_IV, REFLECTOR_B)
_bombe.connexion(X, E, _bombe.add(n, 12))
_bombe.connexion(E, S, _bombe.add(n, 4))
_bombe.connexion(S, X, _bombe.add(n, 8))
_bombe.connexion(X, O, _bombe.add(n, 14))
_bombe.connexion(O, L, _bombe.add(n, 2))
_bombe.connexion(L, M, _bombe.add(n, 3))
_bombe.connexion(M, S, _bombe.add(n, 9))
_bombe.connexion(O, P, _bombe.add(n, 13))
_bombe.connexion(E, G, _bombe.add(n, 10))
_bombe.connexion(G, B, _bombe.add(n, 0))
_bombe.connexion(U, H, _bombe.add(n, 1))
_bombe.connexion(I, D, _bombe.add(n, 6))
_bombe.connexion(D, T, _bombe.add(n, 11))
_bombe.connexion(T, N, _bombe.add(n, 5))
_bombe.connexion(N, R, _bombe.add(n, 7))
_bombe.populate(X, N)
_bombe.results()
```

Question 177 Avec ce test, nous devrions obtenir :

- permutations : `BP CI FG NX OY RW` ;
- lettres non permutées : `D`, `E`, `L`, `M`, `S` et `T`.

En admettant que le réglage du jour comporte 10 paires de lettres permutées sur le tableau de connexion, pour les permutations restantes :

- combien de lettres non permutées n'avons nous pas encore identifiées ?
- en déduire le nombre de paires de lettres permutées restantes.

7.4.5 Finalisation

Dans la section précédente, nous avons utilisé la Bombe de Turing/Welchman pour obtenir la position des rotors et certaines permutations sur le tableau de connexion. **D'autres permutations restent à définir** : pour les lettres `A`, `H`, `J`, `K`, `Q`, `U`, `V` et `Z`. Nous savons que ces lettres sont obligatoirement permutées car nous avons identifié 6 lettres non permutées : $26-6=20$, soit 10 paires de lettres permutées.

Egalement, la position que nous obtenons est le résultat de la différence entre l'affichage et le réglage des anneaux. **Nous ne savons pas quelles valeurs sont utilisées pour chiffrer le message que nous attaquons**.
Dans cette section, l'objectif est de déterminer tous ces éléments manquants afin de finaliser l'attaque.

Question 178 Le script `tp-enigma/13/final.py` contient un squelette prêt pour finaliser notre attaque à clair connu. Exécuter ce script et observer son comportement.

```
debian@myhostname:~$ cd ~/tp-enigma/13
debian@myhostname:~/tp-enigma/13$ ./final.py
```

Permutations restantes

Lorsque la Bombe fournit une liste incomplète de permutations, nous devons définir les permutations restantes. Idéalement, nous possédons une seconde *crib* (un clair connu dans le message que nous attaquons ou dans un autre message). Dans ce cas, il suffit de câbler la Bombe avec cette nouvelle *crib* et de la positionner aux réglages précédemment trouvés : pas besoin de la faire itérer sur toutes les positions. Nous obtenons alors des permutations supplémentaires.

Dans cette section, nous imaginons que nous ne soyons pas en possession d'une telle *crib*. Nous proposons des stratégies pour finaliser l'attaque à l'aide d'une simple machine Enigma.

Question 179 Instancier une machine Enigma avec les réglages fournis par la Bombe : ordre des rotors et permutations. Choisir le triplet (0, 0, 0) pour la position des anneaux.

```
theMachine = EnigmaMachine.from_key_sheet(
    rotors="II III IV",
    reflector="B",
    ring_settings=[0, 0, 0],
    plugboard_settings="BP CI FG NX OY RW"
  )
```

Voici notre *crib* accompagnée de son équivalent chiffré :

```
GHOMSNDRSSGDXPO
BULLETINXMETEOX
```

Nous ignorons les permutations pour les lettres `H` et `U`. Ces lettres sont présentes dans la *crib* à la position $n+1$.

Question 180 En fonction de la valeur n trouvée par la Bombe, déduire un affichage pour la machine Enigma afin de nous positionner à $n+1$.
Attention ! Enigma réalise une rotation avant de chiffrer.

```
n = 11644 + 1 - 1 # we type H, after 1 rotation ; enigma rotates before cipher
theDisplay = theAlphabet[int(n / 26 / 26)]   + \
             theAlphabet[int((n / 26) % 26)] + \
             theAlphabet[int(n % 26)]
print(theDisplay)
```

A la position $n+1$, selon notre *crib*, la lettre `H` est substituée par un `U`. Nous pouvons donc taper une lettre au clavier de la machine Enigma pour choisir $P(\texttt{H})$. Le résultat fourni par la machine Enigma est alors $P(\texttt{U})$.

La Bombe nous a permis d'identifier les 6 lettres qui ne sont pas permutées. Les lettres `H` et `U` n'en font pas partie. Nous savons donc que $P(\texttt{H}) \neq \texttt{H}$ et $P(\texttt{U}) \neq \texttt{U}$.

Question 181 Parmi les lettres dont nous devons définir les permutations, réaliser une attaque par force brute sur $P(\texttt{H})$ et observer le résultat avec la machine Enigma à la position $n+1$.
Ce résultat est donc $P(\texttt{U})$. Selon nos hypothèses, quelles sont les trois valeurs obtenues qui sont impossibles ? Pourquoi ?

```
for i in ["A", "J", "K", "Q", "U", "V", "Z"]:
  theMachine.set_display(theDisplay)
  theResults = theMachine.process_text(i)
  print("P(H)=" + i + " -> P(U)=" + theResults)
```

Avec ce test, nous émettons une hypothèse sur $P(\texttt{H})$ et la machine Enigma nous permet d'obtenir une conséquence sur $P(\texttt{U})$. Avec notre attaque par force brute, trois cas sont impossibles :

— $P(\texttt{H}) = \texttt{A}$ et $P(\texttt{U}) = \texttt{M}$. En effet, nous savons que la lettre M n'est pas permutée.
— $P(\texttt{H}) = \texttt{U}$ et $P(\texttt{U}) = \texttt{Z}$. En effet, le tableau de connexion est symétrique. Donc si $P(\texttt{H}) = \texttt{U}$, alors $P(\texttt{U}) = \texttt{H}$, pas Z.
— $P(\texttt{H}) = \texttt{Z}$ et $P(\texttt{U}) = \texttt{U}$. En effet, nous savons que la lettre U est forcément permutée car nous avons identifié les 6 lettres non-permutées (et la lettre U n'en fait pas partie).

En conséquence, nous avons identifié quatre possibilités pour les permutations des lettres H et U :

— $P(\texttt{H}) = \texttt{J}$ et $P(\texttt{U}) = \texttt{V}$,
— $P(\texttt{H}) = \texttt{K}$ et $P(\texttt{U}) = \texttt{Q}$,
— $P(\texttt{H}) = \texttt{Q}$ et $P(\texttt{U}) = \texttt{K}$,
— $P(\texttt{H}) = \texttt{V}$ et $P(\texttt{U}) = \texttt{J}$.

Question 182 Nous recherchons les permutations pour 8 lettres. Pour chacune des quatre possibilités identifiées, une fois les deux permutations choisies, combien de lettres restantes sont à permuter ?

Question 183 Parmi les lettres restantes, émettons une hypothèse pour la première paire de lettres permutées. Combien d'hypothèses pouvons nous émettre ?

Question 184 Une fois notre hypothèse émise, combien de lettres sont restantes pour les paires suivantes ?

Question 185 Déduire, pour chacune des quatre possibilités pour les permutations des lettres H et U, le nombre de possibilités pour les permutations restantes.
En déduire, si nous devions réaliser une attaque par force brute sur les permutations, le nombre total d'itérations nécessaires.

Nous recherchons les permutations pour 8 lettres. Pour chacune des 4 possibilités pour les permutations des lettres H et U, il reste 4 lettres.

Si nous émettons une hypothèse pour la première paire de lettres permutées, alors nous avons $4-1=3$ hypothèses. Une fois notre hypothèse émise, il reste 2 lettres à permuter : nous n'avons donc pas le choix, celles-ci sont permutées entre elles.

Si nous devions réaliser une attaque par force brute sur les permutations, il y a donc $4 \times 3 \times 1 = 12$ itérations à réaliser. Voici la liste des 12 permutations possibles lors d'une attaque par force brute :

```
thePlugList = [
  "HJ UV AK QZ",
  "HJ UV AQ KZ",
  "HJ UV AZ KQ",

  "HK UQ AJ VZ",
  "HK UQ AV JZ",
  "HK UQ AZ JV",

  "HQ UK AJ VZ",
  "HQ UK AV JZ",
  "HQ UK AZ JV",

  "HV UJ AK QZ",
  "HV UJ AQ KZ",
  "HV UJ AZ KQ"
]
```

Question 186 A l'aide d'une machine Enigma et des réglages obtenus avec la Bombe, réaliser une attaque par force brute sur le texte chiffré complet (pas seulement la *crib*). En connaissant la langue du texte clair, déduire quelles sont les permutations restantes.

```
n = 11644 - 1 # enigma rotates before cipher
theDisplay = theAlphabet[int(n / 26 / 26)]   + \
             theAlphabet[int((n / 26) % 26)] + \
             theAlphabet[int(n % 26)]
```

```
for thePlug in thePlugList:
  theMachine = EnigmaMachine.from_key_sheet(
      rotors="II III IV",
      reflector="B",
      ring_settings=[0, 0, 0],
      plugboard_settings="BP CI FG NX OY RW " + thePlug
    )
  theMachine.set_display(theDisplay)
  print(thePlug + " -> " + theMachine.process_text(theCipherText).replace("X", " "))
```

Réglages des anneaux et affichages

Avec de l'intuition et une machine Enigma, nous avons pu réaliser une petite attaque par force brute (12 itérations) et nous avons déterminé la liste des permutations restantes :

```
HV UJ AQ KZ
```

En choisissant le triplet (0, 0, 0) pour la position des anneaux, lorsque nous déchiffrons le message, nous obtenons du texte écrit en français avec des erreurs de déchiffrement :

```
BULLETIN METEOTLU JOUR AUJOURDHUI LE CIEDYSERA NUAGEU  SUR LA MAJOJQPNSACSNIVEOJVLSWQBVEYMTZSTD
```

Ce résultat s'explique du fait que le triplet (0, 0, 0) est incorrect pour la position des anneaux. Nous avons trouvé la bonne position mais les rotations des rotors 2 et 1 ne sont pas obtenues aux bons indexes. Par exemple, nous voyons que le début du message devrait correspondre au texte : `BULLETIN METEO DU JOUR`. De ce fait, le rotor 2 semble avoir tourné deux lettres trop tôt.

Question 187 Avec la valeur de n que la Bombe nous a fournie, déduire l'affichage que nous obtenons si le triplet (0, 0, 0) est choisi pour la position des anneaux.

```
n = 11644 - 1 # enigma rotates before cipher
theDisplay = theAlphabet[int(n / 26 / 26)]   + \
             theAlphabet[int((n / 26) % 26)] + \
             theAlphabet[int(n % 26)]
print(theDisplay)
```

Question 188 Modifier la position des anneaux pour que le rotor 2 tourne à la bonne lettre. Appliquer l'opération -2 modulo 26 sur l'anneau et -2 modulo 26 sur l'affichage.
Ensuite, tenter de déchiffrer de nouveau le message et vérifier.

```
theMachine = EnigmaMachine.from_key_sheet(
    rotors="II III IV",
    reflector="B",
    ring_settings=[0, 0, 24],
    plugboard_settings="BP CI FG NX OY RW HV UJ AQ KZ"
  )
theMachine.set_display("RFT")
print(theMachine.process_text(theCipherText).replace("X", " "))
```

Question 189 La position de l'anneau du rotor 3 est désormais correcte mais la fin du message est toujours illisible. Réaliser une attaque par force brute sur la position de l'anneau du rotor 2 ; ajuster l'affichage en conséquence.
A chaque itération, afficher le message. Quel est le réglage de l'anneau du rotor 2 ?

```
for i in range(0, 26):
  theMachine = EnigmaMachine.from_key_sheet(
      rotors="II III IV",
      reflector="B",
      ring_settings=[0, i, 24],
      plugboard_settings="BP CI FG NX OY RW HV UJ AQ KZ"
    )
  theDisplay = "R" + theAlphabet[int((n / 26 + i) % 26)] + "T"
  theMachine.set_display(theDisplay)
```

```
    print("r2 = %2d, " % i + theDisplay + " -> " + \
          theMachine.process_text(theCipherText).replace("X", " "))
```

Cette stratégie nous permet d'obtenir les réglages des anneaux et affichages à partir de la position fournie par la Bombe. Vous venez donc de "*craquer*" Enigma à l'aide de la Bombe de Turing/Welchman et de votre intuition. Félicitations !

Durant la seconde guerre mondiale, l'attaque à clair connu que nous venons de réaliser était mise en place tous les jours : à chaque changement de réglages. Tous les matins à 6h, les cryptanalystes recevaient un bulletin météo chiffré avec Enigma. Ce type de messages était souvent utilisé pour réaliser l'attaque.

Bien évidemment, l'attaque ne fonctionne pas dans le cas où le rotor 2 tourne au milieu de la *crib* (exactement comme l'attaque de Marian Rejewski sur l'identifiant). Dans ce cas, une autre *crib* devait être utilisée à un autre emplacement dans le message ou dans un autre message. Une solution des cryptanalystes de Bletchley Park pour trouver de telles *cribs* était de se fier à la taille des messages. Par exemple, les messages interceptés les plus courts contenait le texte `RIEN A SIGNALER`.

Un dernier point sur notre attaque est que les réglages fournis par la Bombe ne sont valables que pour un ordre des rotors donné. Or, avec 3 rotors choisis parmi 5, il faut 60 attaques, soit 20 Bombes de 3 étages, pour paralléliser l'attaque que nous avons réalisée sur tous les ordres des rotors possibles.

Dans son ouvrage *The essential Turing*, Jack Copeland explique que le nombre de Bombes à Bletchley Park était de 5 en juin 1941, puis de 15 en novembre de cette même année. La production de nouvelles Bombes a été augmentée par la suite et de nouveaux sites géographiques, aux alentours de Bletchley Park les ont accueillies. A la fin de la guerre, environ 200 Bombes étaient en opération sur les divers sites. Les renseignements obtenus grâce à ces déchiffrements font partie du programme Ultra et contribuent, peut-être de manière décisive, à la défaite du Troisième Reich.