> Noms du binome: BENICHOU - BONNEFOY
>
> Prénoms du binome: Yaniv - Nicolas
>
> Note: exporter le compte rendu basé sur le notebook au format pdf

Entropie et génération de mots de passe

- L'objectif de ce tp est de continuer à se familiariser avec la notion d'entropie, mais aussi de comprendre le lien qu'il existe entre cette mesure informationelle et la sécurité d'un générateur (humain ou executable) de mots de passes
- Ainsi, nous proposons d'étudier l'entropie d'un tel générateur, et ce en fonction du modèle probabiliste considéré pour le modéliser (contruit à partir d'une lettre, de deux lettres, de 4 lettres, ...). A l'aide de tirrages aléatoires, nous estimerons également le temps moyen nécessaire pour trouver un mot de passe à partir de ce modèle.
- A la fin de ce TP, nous considérerons un générateur de mots de passe spécifique qui générera un mot de passe en prennant **4 lettres consécutives dans un texte** (sans se soucier des espaces). Ces lettres peuvent faire parti d'un ou de plusieurs mots consécutifs.
- Nous faisons l'hypothèse que le texte n'est composé que des 26 lettres de l'alphabet, sans majuscules ni accents

Nous chercherons aussi à comprendre (voir dernière question):

- les bonnes pratiques pour le défenseur, i.e. la personne cherchant à générer/construire un système de génération de mots de passe.
- les bonnes pratiques pour l'attaquant, i.e. la personne essayant de trouver le mot de passe.

Il est important de commenter vos réponses, en utilisant des cellules markdown

In [ ]:
```python
import numpy as np

from numpy import genfromtxt
from pandas import read_csv
import pandas as pd
import time
```

Modèle monogramme (une lettre) : le générateur génère des mots de passe à partir des occurences des monogrammes

- On récupére des données composées de [lettre,frequence d'apparition de la lettre] (voir fichier csv pour comma-separated-value)

In [ ]:
```python
monogramme = pd.read_csv('monogramme.csv')
freq_mono = (monogramme['frequency']).values
letters_mono = (monogramme['letters']).values
print(monogramme) # fréquences sont déja triées
```

```
   letters  frequency
0        E     0.1776
1        S     0.0823
2        A     0.0768
3        N     0.0761
4        T     0.0730
5        I     0.0723
6        R     0.0681
7        U     0.0605
8        L     0.0589
9        O     0.0534
10       D     0.0360
11       C     0.0332
12       P     0.0324
13       M     0.0272
14       Q     0.0134
15       V     0.0127
16       G     0.0110
17       F     0.0106
18       B     0.0080
19       H     0.0064
20       X     0.0054
21       Y     0.0021
22       J     0.0019
23       Z     0.0007
24       K     0.0000
25       W     0.0000
```

Q: Quelles sont les 5 lettres les plus représentées ?

In [ ]:
```python
monogramme.head()
```

Out[ ]:

	letters	frequency
0	E	0.1776
1	S	0.0823
2	A	0.0768
3	N	0.0761
4	T	0.0730

Ecrire une fonction qui calcule l'entropie à partir d'un vecteur constitué de probabilités empiriques (note, il est important de bien *gérer* le cas ou la probabilité est nulle).

In [ ]:
```python
def entropie(freq):
    # Filtrer les probabilités nulles pour éviter les erreurs de calcul
    proba_non_nulles = freq[freq > 0]

    # Calcul de l'entropie
    ent = -np.sum(proba_non_nulles * np.log2(proba_non_nulles))
    return ent
```

Q: en utilisant ce modèle probabiliste pour générer un mot de passe, quelle est l'entropie d'un mot de passe de 8 lettres ?

In [ ]:
```python
entropie_mono_8 = entropie(monogramme['frequency'])*8
print(f"L'entropie d'un mot de passe de 8 lettres pour les monogrammes est de : {entropie_mono_8:.3f} bits.")
```

```
L'entropie d'un mot de passe de 8 lettres pour les monogrammes est de : 31.676 bits.
```

Q: A l'aide de la fonction `np.random.choice()`, estimer le temps nécessaire en secondes pour tirer 100 000 mots de passes en utilisant ce générateur ? (note: ici le tirage n'est pas forcemment réaliste, car aléatoire, mais l'idée est surtout de mesurer le temps minimal nécessaire pour générer N mots de passes).

In [ ]:
```python
# Génère plusieurs mots de passe en utilisant des monogrammes de manière optimisée.

def generer_mdp_mono(n_lettres, monogrammes, frequences, nb_mdp):
    # Génération de tous les monogrammes nécessaires en une seule opération
    all_monogrammes = np.random.choice(monogrammes, n_lettres * nb_mdp, p=frequences)
    # Regroupement des monogrammes en mots de passe
    mots_passe = [''.join(all_monogrammes[i:i + n_lettres]) for i in range(0, n_lettres * nb_mdp, n_lettres)]
    return mots_passe

nb_lettres = 8
nb_mdp = 100000

t = time.time()
mots_de_passe_mono = generer_mdp_mono(8, letters_mono, freq_mono, nb_mdp)
t_mono_100000 = time.time() - t
print("Exemple de 10 mots de passes de 8 lettres : ",mots_de_passe_mono[:10])
print(f"Temps total pour générer 100 000 mots de passe de 8 lettres (monogrammes): {t_mono_100000:.3f} secondes.")
print("On serait beaucoup plus efficace et rapide si on ne faisait pas le .join mais les mots de passes seraient moins réalistes dans ce cas.")
```

```
Exemple de 10 mots de passes de 8 lettres :  ['AAESIEET', 'JVAALQSF', 'ENEORPEO', 'PNRUSACL', 'ECEAIEAU', 'DSMADDTU', 'UEROEESB', 'ESLDAHLU', 'EEIACELE', 'IAINPCEE']
Temps total pour générer 100 000 mots de passe de 8 lettres (monogrammes): 0.485 secondes.
On serait beaucoup plus efficace et rapide si on ne faisait pas le .join mais les mots de passes seraient moins réalistes dans ce cas.
```

Nous definissons l'"entropie du devin" G (guessing entropie) comme le **nombre moyen d'essais successif nécessaires pour trouver un mot de passe à partir de notre générateur**. On peut montrer que $G \geq 2^H/4+1$ où H est l'entropie de la source (voir le papier Password_Entropy_and_Password_Quality.pdf)

Q: calculer le minorant de G pour ce modèle

In [ ]:
```python
G_minorant_mono = 2**entropie_mono_8 / 4 + 1
print(f"Le minorant de G pour les monogrammes est d'environ {G_minorant_mono:.2f} essais.")
```

```
Le minorant de G pour les monogrammes est d'environ 857904864.68 essais.
```

> En moyenne, un minimum d'environ 858 millions d'essais successifs seraient nécessaires pour trouver un mot de passe généré par ce modèle, en utilisant une stratégie d'attaque optimale.
>
> Ce calcul illustre la difficulté théorique de deviner un mot de passe généré par ce système, en se basant sur son entropie.

Q: combien de temps cela prendra-t-il pour trouver un mot de passe si l'on suppose qu'il est possible de prendre le générateur codé précédemment ? (en minutes)

In [ ]:
```python
# Calcul du temps nécessaire pour trouver un mot de passe en se basant sur le minorant de G et sur le temps pris pour générer 100000 mots de passe

# Estimation du temps pour un seul mot de passe
temps_1_mdp = t_mono_100000 / nb_mdp

# Temps total estimé pour trouver un mot de passe (en secondes)
temps_total_pour_trouver_mdp = G_minorant_mono * temps_1_mdp

# Convertir le temps en minutes
temps_total_pour_trouver_mdp_minutes = temps_total_pour_trouver_mdp / 60

print(f"Temps total estimé pour trouver un mot de passe (monogrammes) : {temps_total_pour_trouver_mdp_minutes:.2f} minutes")
```

```
Temps total estimé pour trouver un mot de passe (monogrammes) : 69.40 minutes
```

On propose maintenant d'utiliser un modèle plus évolué qui est construit à partir de la probabilité conjointe de deux lettres successives (bigramme)

In [ ]:
```python
bigramme = read_csv('bigramme.csv',keep_default_na=False)
freq_bi = (bigramme['frequency']).values
letters_bi = (bigramme['letters']).values
```

Q: Quelles sont les 5 couples de lettres les plus représentés ?

In [ ]:
```python
# Contrairement aux monogrames, les fréquences ne sont pas triés
bigramme.sort_values(by=['frequency'], ascending=False).head()
```

Out[ ]:

	letters	frequency
122	ES	0.023809
117	EN	0.021248
82	DE	0.019570
290	LE	0.018845
357	NT	0.017009

Q: en utilisant ce modèle probabiliste pour générer un mot de passe, quelle est l'entropie d'un mot de passe de 8 lettres ?

In [ ]:
```python
# Calcul de l'entropie pour un mot de passe de 8 lettres contenant donc 4 bigrammes
entropie_bi_8  = entropie(freq_bi) * 4

print(f"L'entropie d'un mot de passe de 8 lettres est de : {entropie_bi_8:.3f} bits.")
```

```
L'entropie d'un mot de passe de 8 lettres est de : 30.142 bits.
```

Q: Pourquoi cette entropie est-elle inférieure à celle du modèle construit sur des monogrammes ? Quelle propriété théorique de l'entropie peut justifier ce constat ?

> Les monogrammes, sélectionnés indépendamment, maximisent la variabilité et l'imprévisibilité, conduisant à une entropie élevée. En revanche, les bigrammes, avec leur dépendance entre lettres consécutives, présentent une variabilité réduite. Bien que les bigrammes offrent plus de combinaisons et une complexité accrue, la distribution inégale des probabilités (certains bigrammes étant plus fréquents que d'autres) diminue l'entropie globale.
>
> Ainsi, malgré une plus grande complexité théorique des bigrammes, la distribution déséquilibrée des probabilités mène à une entropie inférieure, reflétant une prévisibilité accrue par rapport à un modèle de monogrammes où chaque lettre a une chance relativement égale d'être choisie.
>
> La propriété théorique de l'entropie qui justifie ce constat est donc la dépendance entre les événements dans le modèle de bigrammes. L'entropie, en tant que mesure de l'incertitude ou de l'imprévisibilité, est influencée par la manière dont les événements (ou les lettres, dans ce cas) sont reliés les uns aux autres.

Q: A l'aide de la fonction `np.random.choice()`, calculer le temps nécessaire en secondes pour tirer 100 000 mots de passes en utilisant ce générateur ?

In [ ]:
```python
# Génère plusieurs mots de passe en utilisant des monogrammes de manière optimisée.

def generer_mdp_bi(n_lettres, monogrammes, frequences, nb_mdp):
    # Génération de tous les monogrammes nécessaires en une seule opération
    all_monogrammes = np.random.choice(monogrammes, n_lettres * nb_mdp, p=frequences)

    # Regroupement des monogrammes en mots de passe
    mots_passe = [''.join(all_monogrammes[i:i + n_lettres]) for i in range(0, n_lettres * nb_mdp, n_lettres)]
    return mots_passe

nb_lettres = 4
nb_mdp = 100000

t = time.time()
mots_de_passe_bi = generer_mdp_bi(4, letters_bi, freq_bi, nb_mdp)
t_bi_100000 = time.time() - t
#print(mots_de_passe_bi)
print(f"Temps total pour générer 100 000 mots de passe de 8 lettres (bigrammes): {t_bi_100000:.3f} secondes.")
```

```
Temps total pour générer 100 000 mots de passe de 8 lettres (bigrammes): 0.312 secondes.
```

Q: calculer le minorant de G pour ce modèle

In [ ]:
```python
H_bigramme = entropie_bi_8
G_minorant_bi = (2**H_bigramme)/4 + 1
print(f"Le minorant de G pour les bigrammes est d'environ {G_minorant_bi:.2f} essais.")
```

```
Le minorant de G pour les bigrammes est d'environ 296254956.70 essais.
```

> En moyenne, un minimum d'environ 296 millions d'essais successifs seraient nécessaires pour trouver un mot de passe généré par ce modèle, en utilisant une stratégie d'attaque optimale.
>
> Ce calcul illustre la difficulté théorique de deviner un mot de passe généré par ce système, en se basant sur son entropie.

Q: combien de temps cela prendra-t-il pour trouver un mot de passe si l'on suppose qu'il est possible de prendre le générateur codé précédemment ? (en minutes)

In [ ]:
```python
# Calcul du temps nécessaire pour trouver un mot de passe en se basant sur le minorant de G et sur le temps pris pour générer 100000 mots de passe

# Estimation du temps pour un seul mot de passe
temps_1_mdp = t_bi_100000 / nb_mdp

# Temps total estimé pour trouver un mot de passe (en secondes)
temps_total_pour_trouver_mdp = G_minorant_bi * temps_1_mdp

# Convertir le temps en minutes
temps_total_pour_trouver_mdp_minutes = temps_total_pour_trouver_mdp / 60

print(f"Temps total estimé pour trouver un mot de passe (bigrammes) : {temps_total_pour_trouver_mdp_minutes:.2f} minutes")
```

```
Temps total estimé pour trouver un mot de passe (bigrammes) : 15.40 minutes
```

Q: **Modèle Uniforme:** si maintenant on change de stratégie et on tire aléatoirement chaque lettre de l'alphabet de façon uniforme, quelle est l'entropie de ce nouveau générateur ?

In [ ]:
```python
import math
n_letters = 26
entropy_uniform_model = round(math.log2(n_letters),3)*8


print(f"L'entropie d'un générateur uniforme de mots de passe est de {entropy_uniform_model} bits.")
```

```
L'entropie d'un générateur uniforme de mots de passe est de 37.6 bits.
```

Q: A l'aide de la fonction `np.random.choice()`, calculer le temps nécessaire en secondes pour tirer 100 000 mots de passes en utilisant ce générateur ?

In [ ]:
```python
def generate_passwords(n_passwords, password_length):
    alphabet = list('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
    passwords = []
    for _ in range(n_passwords):
            passwords.append(''.join(np.random.choice(alphabet, size=password_length)))    # Taille arbitraire de 8 caractères
    return  passwords

# Example usage
n_passwords = 100000
password_length = 8
start_time = time.time()  # Start timing
passwords = generate_passwords(n_passwords, password_length)
t_uni_100000 = time.time()  - start_time
print(passwords[:10])
print(f"Temps total pour générer {n_passwords} mots de passe de {password_length} lettres (uniforme): {t_uni_100000:.3f} secondes.")
```

```
['ANIUITEE', 'CCQBRKXG', 'XCTSGBGZ', 'JFIISLVL', 'CCJRPZNN', 'OYSTKYJR', 'JUYWYMTS', 'YWKTRQMJ', 'CCQVXNBX', 'KWLADBMR']
Temps total pour générer 100000 mots de passe de 8 lettres (uniforme): 5.002 secondes.
```

Q: calculer le minorant de G pour ce modèle

In [ ]:
```python
# Calcul du minorant de G à partir de l'entropie
G_minorant_uniform = 2 ** (entropy_uniform_model)/4+1

print(f"Minorant de G calculé à partir de l'entropie pour un mot de passe de 8 lettres : {G_minorant_uniform} essais.")
```

```
Minorant de G calculé à partir de l'entropie pour un mot de passe de 8 lettres : 52079624666.3406 essais.
```

Q: dans ce cas précis, quelle est la valeur exacte de G?

> Il y a deux cas à différencier ici, celui où l'on génère les mots de passes sans remise (avec l'utilisation d'un dictionnaire par exemple) et celui où l'on génère les mots de passes avec remise (qui serait une approche bien moins efficace mais envisageable).
>
> Pour les calculs suivants, on pose: $C = 26^8 =$ le nombre de mots de passes possibles avec 8 lettres.
>
> - Dans le cas de la génération avec remise, on calculer G comme suit:
>
> $$\begin{aligned} G &= E(\text{'on trouve le mot de passe au n-ième essai'}) \\ &= \sum_{k=1}^{\infty} k * P(\text{'on trouve le mot de passe au n-ième essai'}) \\ &= \sum_{k=0}^{\infty} k * (\frac{C-1}{C})^{k-1} * \frac{1}{C} \\ &= \frac{1}{C} * \frac{d}{dx}(\sum_{k=0}^{\infty} x^k)_{(x=\frac{C-1}{C})} \\ &= \frac{1}{C} * \frac{d}{dx}(\frac{1}{1-x})_{(x=\frac{C-1}{C})} \\ &= \frac{1}{C} * \frac{1}{(1-\frac{C-1}{C})^2} \\ &= \frac{1}{C} * \frac{1}{(\frac{1}{C})^2} \\ &= C \\ &= 26^8 \end{aligned}$$
>
> - Dans le cas de la génération sans remise, on calculer G comme suit:
>
> $$\begin{aligned} G &= E(\text{'on trouve le mot de passe au n-ième essai'}) \\ &= \sum_{k=1}^{C} k * P(\text{'on trouve le mot de passe au n-ième essai'}) \\ &= \sum_{k=0}^{C} k * \frac{C-1}{C} * \frac{C-2}{C-1} * \ldots * \frac{C+1-k}{C-k+2} * \frac{1}{C+1-k} \\ &= \sum_{k=0}^{C} k * \frac{1}{C} \\ &= \frac{1}{C} * \frac{C(C+1)}{2} \\ &= \frac{C+1}{2} \\ &= (26^8+1)/2 \end{aligned}$$

In [ ]:
```python
G_exact_avec_remise = 26 ** password_length
print(f"La valeur exacte de G pour un mot de passe de 8 lettres (modèle uniforme et génération avec remise) : {G_exact_avec_remise} combinaisons possibles.")

G_exact_sans_remise = (26**password_length+1)/2
print(f"La valeur exacte de G pour un mot de passe de 8 lettres (modèle uniforme et génération sans remise) : {G_exact_sans_remise} combinaisons possibles.")
```

```
La valeur exacte de G pour un mot de passe de 8 lettres (modèle uniforme et génération avec remise) : 208827064576 combinaisons possibles.
La valeur exacte de G pour un mot de passe de 8 lettres (modèle uniforme et génération sans remise) : 104413532288.5 combinaisons possibles.
```

Q: combien de temps cela prendra-t-il pour trouver un mot de passe en utilisant le générateur codé précédemment ? (en minutes)

In [ ]:
```python
# Calcul du temps nécessaire pour trouver un mot de passe en se basant sur le minorant de G et sur le temps pris pour générer 100000 mots de passe

# Estimation du temps pour un seul mot de passe
temps_1_mdp = t_uni_100000 / nb_mdp

# Temps total estimé pour trouver un mot de passe (en secondes)
temps_total_pour_trouver_mdp = G_minorant_uniform * temps_1_mdp

# Convertir le temps en minutes
temps_total_pour_trouver_mdp_minutes = temps_total_pour_trouver_mdp / 60

print(f"Temps total estimé pour trouver un mot de passe (uniforme) : {temps_total_pour_trouver_mdp_minutes:.2f} minutes")
```

```
Temps total estimé pour trouver un mot de passe (uniforme) : 43417.41 minutes
```

Q: implémenter une attaque pratique qui consiste à:

1. **pour le défenseur: (la personne qui génère le mot de passe)** tirer un mot de passe de 4 lettres consécutives à partir de ce texte de Victor Hugo (texteFrancais.txt) tiré des Misérables.
2. **pour l'attaquant:** utiliser le modèle bigramme pour générer des mots de passe et minimiser le nombre d'essais. Pour cela on pourra :
   - dans un premier temps pré-calculer un **dictionnaire**, qui contriendra un nombre de MdP générés classés dans l'ordre du plus probable au moins probable et qui ne contient pas de doublons
   - dans un deuxième temps appeler ce dictionnaire pour comparer chacune de ses entrées au mot de passe généré.
3. Il faudra faire ses tests plusieurs fois afin de d'obtenir un nombre moyens d'appel au dictionnaire nécessaire
4. Il sera intéressant de comparer le nombre trouvé à la valeur de G (qui est une borne inférieure)
5. Question annexe: Par un simple calcul, si le générateur utilisé n'est plus ce générateur mais un générateur qui tire chaque lettre de façon équiprobable, rappeler la valeur de G. Comparer cette valeur avec la valeur trouvée en utilisant la stratégie "des 4 lettres consécutives".

In [ ]:
```python
## Fonction générant un mot de passe
def get_passwd():
    text_hugo = open("texteFrancais.txt","r")
    str_hugo = str(text_hugo.read())

    # On remplace des lettres avec accent avec des lettres sans accent
    str_hugo = str_hugo.replace("Â", "A")
    str_hugo = str_hugo.replace("Ù", "U")
    str_hugo = str_hugo.replace("Ô", "O")
    size_txt = len(str_hugo)

    idx_rand = np.random.randint(size_txt-4)
    #print(idx_rand)

    psswd = str_hugo[idx_rand:idx_rand+4]
    return(psswd)
```

In [ ]:
```python
import itertools
import pandas as pd

# Génération du dictionnaire
def generate_mdp_dictionary(password_length=4):
    # Produit cartésien des bigrammes pour tenir compte de toutes les combinaisons possibles
    couples = list(itertools.product(bigramme.letters.tolist(), repeat=2))
    probs = list(itertools.product(bigramme.frequency.tolist(), repeat=2))

    # Dictionnaire pour stocker les mots de passe et leurs probabilités
    mdp_dict = {}

    for c, p in zip(couples, probs):
        # Générer des mots de passe de 4 lettres et calculer leurs probabilités
        password = c[0][:2] + c[1][:2]
        proba = p[0] * p[1]

        if len(password) == password_length:
            if password not in mdp_dict or mdp_dict[password] < proba:
                mdp_dict[password] = proba

    # Création d'un DataFrame plutot qu'une liste pour trier les mots de passe
    df_dictionary = pd.DataFrame(list(mdp_dict.items()), columns=['mdp', 'frequency'])
    df_dictionary = df_dictionary.sort_values(by='frequency', ascending=False, ignore_index=True)

    return df_dictionary

df_dictionary = generate_mdp_dictionary()
df_dictionary
```

Out[ ]:

	mdp	frequency
0	ESES	0.000567
1	ENES	0.000506
2	ESEN	0.000506
3	DEES	0.000466
4	ESDE	0.000466
...	...	...
456971	SIQD	0.000000
456972	SIQE	0.000000
456973	HFCF	0.000000
456974	SIQG	0.000000
456975	ZZZZ	0.000000

456976 rows × 2 columns

In [ ]:
```python
# Attaques sur 1000 mots de passes

nb_trial = 1000
vec_nb_trials = np.full(nb_trial, -1)
for i in range(nb_trial):
    password = get_passwd() #génération du mdp
    try:
        index = df_dictionary[df_dictionary['mdp'] ==password].index.tolist()[0]
        vec_nb_trials[i] = index
    except:
        print(password)
```

In [ ]:
```python
# Affichage des valeurs et des comparaisons
print(f"En moyenne, il y a {np.mean(vec_nb_trials):.2f} appels.")

print(f"La valeur du minorant calculé pour un mot de passe avec 4 lettres et générateur monogramme est: {round(2**(entropie(monogramme['frequency']) * 4)/4+1, 3)}")
print(f"La valeur du minorant calculé pour un mot de passe avec 4 lettres et générateur bigramme est: {round(2**(entropie(freq_bi) * 2)/4+1, 3)}")
print(f"Cette valeur est bien une borne inférieure puisqu'elle est inférieure au nombre moyen d'appels = {np.mean(vec_nb_trials):.2f}")

print(f"La valeur du minorant calculé pour un mot de passe avec 4 lettres et générateur uniforme est: {26**4/4+1}")
print(f"Cela montre que si nous avions utilisé le générateur uniforme, la valeur G serait {round((26**4/4+1)/np.mean(vec_nb_trials), 2)} fois plus grande que le nombre
```

```
En moyenne, il y a 13523.85 appels.
La valeur du minorant calculé pour un mot de passe avec 4 lettres et générateur monogramme est: 14646.007
La valeur du minorant calculé pour un mot de passe avec 4 lettres et générateur bigramme est: 8607.029
Cette valeur est bien une borne inférieure puisqu'elle est inférieure au nombre moyen d'appels = 13523.85
La valeur du minorant calculé pour un mot de passe avec 4 lettres et générateur uniforme est: 114245.0
Cela montre que si nous avions utilisé le générateur uniforme, la valeur G serait 8.45 fois plus grande que le nombre moyen d'appels.
```

Conclusions

- ##### Définir des bonnes pratiques pour le défenseur, i.e. la personne cherchant à concevoir un système de génération de mots de passe ?

  > L'entropie étant maximale dans le cas d'un tirage uniforme, c'est le meilleur moyen de générer un mot de passe robuste. Il faut ensuite travailler avec la longueur du mot de passe. Une longueur de 8 lettres n'est pas suffisant car le mot de passe est crackable en environ 13h. En passant à 12 lettres nous arrivons à 1000 ans ce qui rend le mot de passe non crackable. En rajoutant les lettres majuscules, les chiffres et les caractères spéciaux, on obtient un mot de passe bien plus sécurisé.

- ##### Définir des bonnes pratiques pour l'attaquant, i.e. la personne essayant de trouver le mot de passe ?

  > Pour réduire le nombre d'appels avant de décoder un mot de passe, l'attanquant doit chercher à exploiter les aprioris que l'on peut faire sur un mot de passe à savoir : La plupart des gens vont utiliser des mots de passe facile à retenir donc des mots de leur langue ou présent dans des dictionnaires de mots de passe. Construire sa recherche de mot de passe autour de cet apriori probabiliste permet de tester en priorité les combinaisons les plus probables. Dans le cadre de ce TP nous nous limitons à l'association de lettres.

Un peu de lecture

Cet article montre comment des hackers, à partir de leaks de bases de mots de passes, peuvent rapidement arriver à trouver le votre: https://arstechnica.com/information-technology/2013/05/how-crackers-make-minced-meat-out-of-your-passwords/