# Noms du binôme : SALEM - SALEM

# Prénoms du binôme : Emilie - Hadrien

# 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 tirages 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 qui générera un mot de passe en prennant **8 lettres consécutives dans un texte** (sans se soucier des espaces). Ces lettres peuvent faire partir 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 [ ]:
```
import numpy as np
from numpy import genfromtxt
from pandas import read_csv
import pandas as pd
import time
```

## Modèle monogramme (une lettre)

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

In [ ]:
```
monogramme = read_csv('monogramme.csv')
freq_mono = (monogramme['frequency']).values
letters_mono = (monogramme['letters']).values
```

Q: Quelles sont les 5 lettres les plus représentées ?

In [ ]:
```
sorted_monogramme = monogramme.sort_values(by=['frequency'], ascending=False)
sorted_monogramme[:5]
```

Out[ ]:

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

R : Les 5 lettres les plus représentées sont ainsi E, S, A, N et T.

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 [ ]:
```
def entropie(freq):
    ent = 0
    for p in freq :
        if p != 0 : ent -= p*np.log2(p)
    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 ?

Un mot de passe est généré en tirant chaque lettre indépendamment les unes de autres : l'entropie totale est donc la somme des entropies de chacune des lettres (variables aléatoires indépendantes).

In [ ]:
```
entropie_8 = 8 * entropie(freq_mono)
print(f"L'entropie d'un mot de passe de 8 lettres est : {entropie_8} bits.")
```

```
L'entropie d'un mot de passe de 8 lettres est : 31.676242429778338 bits.
```

Q: A l'aide de la fonction `np.random.choice()` , estimer le temps nécessaire en secondes pour trer 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 [ ]:
```
t = time.time()

for _ in range(100000):
    np.random.choice(letters_mono, size=8, p=freq_mono)

t_100000_mono = time.time() - t
print(t_100000_mono, 's')
```

```
4.1318531036637695 s
```

Nous définissons l'"entropie du devin" G (guessing entropie) comme le **nombre moyen d'essais successifs 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 [ ]:
```
min_G_mono = 2**(entropie_8) / 4 + 1
print(f"Le minorant du nombre moyen d'attaques successives nécessaires est : {min_G_mono}")
```

```
Le minorant du nombre moyen d'attaques successives nécessaires est : 857904864.6814479
```

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

In [ ]:
```
temps_1_mdp_mono = t_100000_mono / 100000
guessing_time_mono = temps_1_mdp_mono * min_G_mono / 60
print(f"On peut espérer trouver un mot de passe généré à partir du monogramme en : {guessing_time_mono} minutes.")
```

```
On peut espérer trouver un mot de passe généré à partir du monogramme en : 590.9708388334925 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 [ ]:
```
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 [ ]:
```
sorted_bigramme= bigramme.sort_values(by=['frequency'], ascending=False)
sorted_bigramme[:5]
```

Out[ ]:

| | letters | frequency |
|---|---|---|
| 122 | ES | 0.023009 |
| 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 [ ]:
```
entropie_4 = 4 * entropie(freq_bi)
print(f"L'entropie d'un mot de passe de 8 lettres avec le bigramme est : {entropie_4} bits.")
```

```
L'entropie d'un mot de passe de 8 lettres avec le bigramme est : 30.142268404464188 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 ?

L'entropie est inférieure à celle du modèle construit sur des monogrammes d'après la propriété théorique : $H(X,Y) \leq H(X) + H(Y)$. En effet, lorsque le générateur est un bigramme, on sélectionne un couple de lettres et non plus une lettre unique. Ainsi l'entropie associée au choix de 2 lettres avec le modèle bigramme est inférieure à l'entropie associée au choix de 2 fois 1 lettre avec le modèle monogramme.

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

In [ ]:
```
t = time.time()

for _ in range(100000):
    np.random.choice(letters_bi, size=4, p=freq_bi)

t_100000_bi = time.time() - t

print(t_100000_bi, 's')
```

```
5.006297588348389 s
```

Q: calculer le minorant de $G$ pour ce modèle

In [ ]:
```
min_G_bi = 2**(entropie_4) / 4 + 1
print(f"Le minorant du nombre moyen d'attaques successives nécessaires est : {min_G_bi}")
```

```
Le minorant du nombre moyen d'attaques successives nécessaires est : 296254956.70154107
```

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

In [ ]:
```
temps_1_mdp_bi = t_100000_bi / 100000
guessing_time_bi = temps_1_mdp_bi * min_G_bi / 60
print(f"On peut espérer trouver un mot de passe généré à partir du monogramme en : {guessing_time_bi} minutes.")
```

```
On peut espérer trouver un mot de passe généré à partir du monogramme en : 243.13463292233004 minutes.
```

Q: 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 [ ]:
```
freq_unif = np.ones(26)/26
letters_unif = (monogramme['letters']).values

entropie_unif = 8 * entropie(freq_unif)
print(f"L'entropie d'un mot de passe de 8 lettres généré uniformément est : {entropie_unif} bits.")
```

```
L'entropie d'un mot de passe de 8 lettres généré uniformément est : 37.60351774512872 bits.
```

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

In [ ]:
```
t = time.time()

for _ in range(100000):
    np.random.choice(letters_unif, size=8, p=freq_unif)

t_100000_unif = time.time() - t

print(t_100000_unif, 's')
```

```
4.201736459195325 s
```

Q: calculer le minorant de $G$ pour ce modèle

In [ ]:
```
min_G_unif = 2**(entropie_unif) / 4 + 1
print(f"Le minorant du nombre moyen d'attaques successives nécessaires est : {min_G_unif}")
```

```
Le minorant du nombre moyen d'attaques successives nécessaires est : 52206766144.99937
```

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

Dans ce cas particulier, puisque les mots de passe sont tirés uniformément, le nombre moyen d'essais nécessaires est la moitié du nombre de mots de passes possibles (8 lettres avec chacune 26 possibilités, soit $26^8$ mots de passes possibles).

In [ ]:
```
G_exact = 26**8 / 2
print(G_exact)
```

```
104413532288.0
```

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

In [ ]:
```
temps_1_mdp_unif = t_100000_unif / 100000
guessing_time_unif = temps_1_mdp_unif * min_G_unif / 60
print(f"On peut espérer trouver un mot de passe généré à partir du monogramme en : {guessing_time_unif} minutes.")
```

```
On peut espérer trouver un mot de passe généré à partir du monogramme en : 36559.845376377736 minutes.
```

Q: Implémenter une attaque pratique qui consiste à:

1. **pour le défenseur:** 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 contiendra 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 ces tests plusieurs fois afin de d'obtenir un nombre moyen 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 [ ]:
```
## 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("À", "A")
    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)

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

In [ ]:
```
# Génération du dictionnaire et calcul des fréquences d'apparition de  chaque mot du dictionnaire
tab_passwd = []
tab_proba = []
for i in range(len(sorted_bigramme)) :
    for j in range(len(sorted_bigramme)) :
        mdp = sorted_bigramme['letters'][i] + sorted_bigramme['letters'][j]
        proba = sorted_bigramme['frequency'][i] * sorted_bigramme['frequency'][j]
        tab_passwd.append(mdp)
        tab_proba.append(proba)
```

In [ ]:
```
# Tri des fréquences d'apparition
N = len(tab_proba)
tab_proba_arr = np.array(tab_proba)
tab_passwd_arr = np.array(tab_passwd)

ind = np.argsort(tab_proba_arr)
tab_passwd_arr = np.flip(tab_passwd_arr[ind])
```

In [ ]:
```
# Attaques sur 1000 mots de passes
nb_trials = 1000
vec_nb_trials = np.zeros(nb_trials)

for trial in range(nb_trials):
    mdp_to_guess = get_passwd()
    N_it = 0
    for mdp in tab_passwd_arr :
        N_it += 1
        if(mdp == mdp_to_guess) : break
    vec_nb_trials[trial] = N_it
```

In [ ]:
```
mean_trials = np.mean(vec_nb_trials)
print(f"Le nombre moyen d'attaques nécessaires pour trouver le mot de passe est  : {mean_trials}")
```

```
Le nombre moyen d'attaques nécessaires pour trouver le mot de passe est  : 14879.526
```

In [ ]:
```
# Comparaison avec la valeur de G
print(f"La borne inférieure de G que nous avions calculée était : {2**(2*entropie(freq_bi)) / 4 + 1}")
```

```
La borne inférieure de G que nous avions calculée était : 8607.029219412707
```

## Conclusions

- Définir des bonnes pratiques pour le défenseur, i.e. la personne cherchant à concevoir un système de génération de mots de passe ?
- Définir des bonnes pratiques pour l'attaquant, i.e. la personne essayant de trouver le mot de passe ?

### Bonnes pratiques pour le défenseur

- Tirer les lettres du mot de passe uniformément afin de maximiser son entropie
- A distribution identique, les mots de passe plus longs ont une entropie plus importante et sont donc plus difficiles à "deviner"

### Bonnes pratiques pour l'attaquant

- Essayer les mots de passe les plus probables en premier permet de réduire le nombre moyen d'essais, dans le cas où le mot de passe à attaquer n'a pas été généré uniformément. En particulier, si le mot de passe correspond à des mots d'une langue, on peut se baser sur la fréquence d'apparition des lettres dans cette langue.

## 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/