La génération d'aléatoire à partir d'un ordinateur est une tâche complexe, il est important de faire une distinction entre les deux types principaux de génération aléatoire : le RNG (Random Number Generator) et le PRNG (Pseudo-Random Number Generator).

- Le RNG est un générateur de nombres aléatoires qui repose sur des sources de véritable aléatoire. Ces sources incluent des phénomènes physiques imprévisibles, tels que le bruit électronique, les mouvements browniens, les perturbations atmosphériques, etc. Les véritables RNG sont considérés comme générant des nombres vraiment aléatoires, car ils ne suivent aucun modèle déterministe.

- Le PRNG est un générateur de nombres pseudo-aléatoires qui utilise des algorithmes mathématiques pour produire des séquences de nombres qui ressemblent à de l'aléatoire. Cependant, ces séquences sont déterministes et peuvent être reproduites si la même graine initiale est utilisée.

Dans notre cas nous avons essayé de coder un système de RNG sans utiliser (random, secret hash etc… qui sont des PNRG pour la plupart) ou de générateur de chiffre aléatoire existant car sinon l'exercice aurait aucune utilité.

Pour réalisé un RNG il y a différentes étapes :

1 : Collecte d'entropie : collecter données à partir de sources aléatoire, telles que des mouvements de souris, frappes au clavier, des variations temporelles, données venant de capteur (plus l'entropie est élevé plus les nombres générés seront aléatoires)

2 : Mixage de l'entropie : les données d'entropie doivent être mélangées de façon à éliminer tout modèle (hachage ou technique de mélange)

3 : Générer la seed à partir de l'entropie mélangée

4 : Concevoir un algorithme de génération qui utilise la graine (doit être basée sur des principes de cryptographie si possible)

- Voici le main exécuté par notre programme :

```python
if __name__ == "__main__":
    seedInitial = generate_initial_seed()
    dataMixe = collect_entropy()

    data_to_mix = json.dumps(dataMixe)

    mix = custom_mix(data_to_mix.encode(), seedInitial)
    key = str(time.time() / 5)
    doublemix = double_mix(mix, key)

    with open("mix_data.txt", "w") as mix_file:
        mix_file.write(seed_to_binary(mix))

    with open("doublemix_data.txt", "w") as doublemix_file:
        doublemix_file.write(seed_to_binary(doublemix))
```

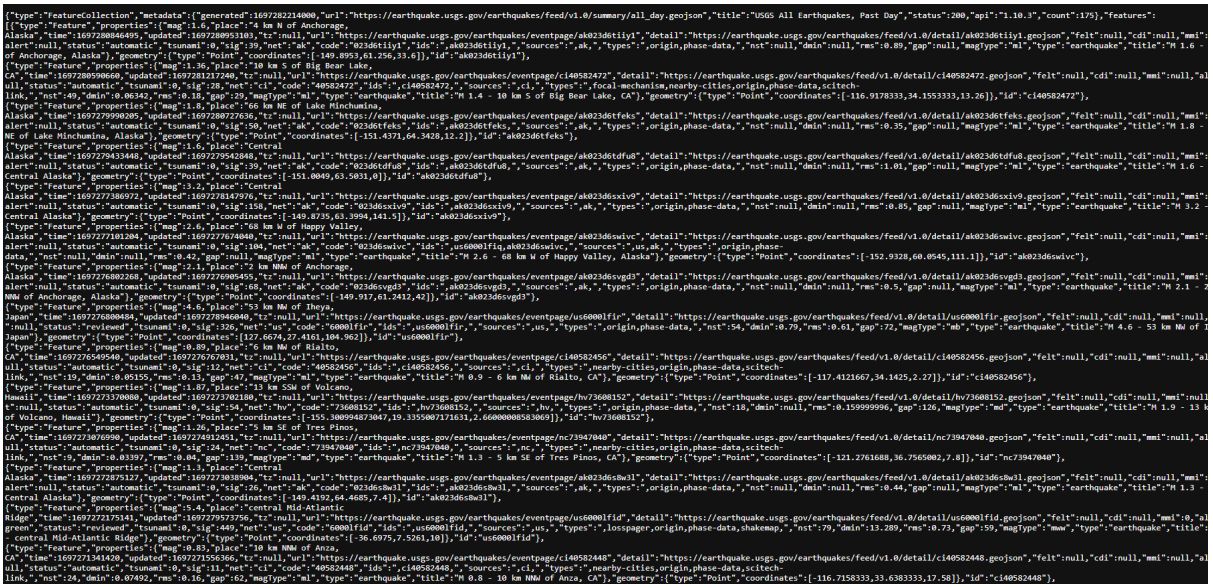
- la variable **seedInitial** est généré par la fonction **generate_initial_seed**:

```python
def generate_initial_seed():
    current_time = time.time()
    time_bytes = struct.pack("<d", current_time)
    seed = bytearray(time_bytes)
    return seed
```

qui va générer une seed initiale (qui n'est pas la seed finale) en fonction du temps et la renvoie en octet.

- **Pour collecter de l'entropie** nous avons utilisé une **api** publique de **l'institut d'études géologiques des Etats Unis** qui permet de retourner un fichier json avec tous **les séismes en cours sur terre** (le fichier est souvent actualisé environ toutes les 1 ou 2

minutes. (lien : https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_day.geojson)
les données sur les séismes en cours peuvent être utilisées comme **source d'entropie** pour générer des nombres aléatoires en raison de **l'imprévisibilité** des séismes, de leur **variabilité temporelle**, de la **quantité** de **données** disponibles en **temps réel** et de leur disponibilité



la variable **dataMixe** est générée par la fonction **collect_entropy :**
Voici notre fonction qui collecte **l'entropie** :

```
def collect_entropy():
    timestamp = int(time.time())
    url = "https://earthquake.usgs.gov/earthquakes/feed/v1.0/summary/all_day.geojson"
    response = requests.get(url)

    if response.status_code == 200:
        data = response.content

        earthquake_data = json.loads(data)

        extracted_data = {}
        extracted_data['timestamp']=timestamp/2;
        if 'features' in earthquake_data and len(earthquake_data['features']) > 0:
            first_earthquake = earthquake_data['features'][0]
            if 'properties' in first_earthquake:
                properties_data = first_earthquake['properties']
                extracted_data['mag'] = properties_data.get('mag', 'N/A')
                extracted_data['time'] = properties_data.get('time', 'N/A')
                extracted_data['updated'] = properties_data.get('updated', 'N/A')
            if 'geometry' in first_earthquake:
                geometry_data = first_earthquake['geometry']
                if 'coordinates' in geometry_data:
                    extracted_data['coordinates'] = geometry_data['coordinates']

        return extracted_data
```

Elle récupère **différentes données** tel que la **magnitude**, le **temps** a quel seconde ça c'est **produit**, le **temps** de la **dernière** mise à jour, les **coordonnées**, et le **nombre** de **seconde** depuis 1970 **divisé par 2**. Elle met tout dans une **liste** et la retourne.

```
data_to_mix = json.dumps(dataMixe)
```

Cette ligne du **main** permet de convertir les données récupérées en fichier json qui permet la facilité d'utiliser ces données

```
mix = custom_mix(data_to_mix.encode(), seedInitial)
```

Cette ligne du main permet de définir la variable **mix** qui utilise la fonction **custom_mix** qui utilise les paramètres des données en json .(ici convertit en une séquence d'octet) et la seed Initial créer au début

Voici la fonction **custom_mix** :

```
def custom_mix(data, key):
    data_to_mix = ''.join([str(byte) for byte in data])
    data_bytes = data_to_mix.encode()

    key_len = len(key)
    mixed_data = bytearray(len(data_bytes))

    for i in range(len(data_bytes)):
        char = data_bytes[i]
        key_char = key[i % key_len]
        key_value = int(key_char)
        mixed_byte = char ^ int(key_char)
        mixed_data[i] = mixed_byte

    return mixed_data
```

Cette fonction sans rentrer dans le code permet de faire un XOR (XOR cipher - Wikipedia) entre les données récupérer du dernier séisme avec

la seed récolter en fonction du nombre de seconde (il s'agit de la partie de mixage/hachage)

Voici les 2 lignes suivantes du main :

```
key = str(time.time() / 5)
doublemix = double_mix(mix, key)
```

La première instancie une nouvelle clé en divisant le nombre de secondes depuis 1970 par 5 (on va l'utiliser pour hacher encore une fois les données)

La deuxième utilise la fonction **double mix** entre les données mixer et la nouvelle clé (seed).

Voici la fonction **doublemix :**

```
def double_mix(data, key):
    data_bytes = data
    key_bytes = key.encode()

    mixed_bytes = bytes((x + y) % 256 for x, y in zip(data_bytes, key_bytes))

    current_time = struct.pack("<d", time.time())
    seed = bytearray((x + y) % 256 for x, y in zip(mixed_bytes, current_time))

    return bytes(seed)
```

La fonction **double_mix** prend des données d'entrée data et une clé key, puis les mélange en ajoutant les octets des deux entrées de manière modulaire, en garantissant que les valeurs restent dans la plage des octets valides (0 à 255). Ensuite, elle ajoute également l'heure actuelle, et le résultat final est renvoyé sous forme de séquence d'octets mélangés.

Ces dernières lignes du main permet d'écrire la première seed dans un le fichier mix_file.txt et la véritable seed dans doublemix_data

La seed ressemble à ca :

11010011101110111111110001111000011010111111011111110001110101
010

```
with open("mix_data.txt",."w") as mix_file:
    mix_file.write(seed_to_binary(mix))

with open("doublemix_data.txt", "w") as doublemix_file:
    doublemix_file.write(seed_to_binary(doublemix))
```

Un autre fichier exécute le premier fichier et test la seed créer aléatoire :

```
import random
import subprocess



subprocess.run(["python", "monRandom.py"])
with open('doublemix_data.txt', 'r') as file:
    seed = file.read().strip()


ma_liste = [1, 2, 3, 4, 5, 6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]


seed_int = int(seed, 2)

random.seed(seed_int)
index_choisi = random.randint(0, len(ma_liste) - 1)

chiffre_aleatoire = ma_liste[index_choisi]

with open('resultat.txt', 'w') as fichier:
    fichier.write(str(chiffre_aleatoire))
```

ici la classe random est utilisé pour interpréter la seed généré au préalable, ce qui n'est pas réellement aléatoire dans la classe random c'est la création de la seed (ici la seed est déjà généré par nous même)

Le résultat est écrit dans le fichier resultat.txt qui pourra être interprété par le jeux (ces fichier seront en .htacces all access denied bien évidemment mais il serait peut être judicieux de les crypter pour un maximum de sécurité)

```python
import subprocess

listeTout = []
listes = [[] for _ in range(21)]

for loop in range(30):
    subprocess.run(["python", "random_entropy.py"])
    with open('resultat.txt', 'r') as file:
        resultat = file.read().strip()
    try:
        nombre = int(resultat)
    except ValueError:
        nombre = 0
    if 1 <= nombre <= 20:
        listes[nombre].append(resultat)
        listeTout.append(resultat)

print(listeTout)
print(listes[1:])
```

Ici on teste avec 30 répétitions notre programme et voici le résultat :

La liste des résultat sortit dans l'ordre de sortie : ['20', '15', '19', '18', '8', '4', '3', '17', '8', '6', '8', '9', '12', '15', '14', '17', '12', '13', '18', '14', '8', '1', '7', '3', '12', '18', '15', '9', '12', '19']

et voici les résultats regroupé par nombre : [['1'], [], ['3', '3'], ['4'], [], ['6'], ['7'], ['8', '8', '8', '8'], ['9', '9'], [], [], ['12', '12', '12', '12'], ['13'], ['14', '14'], ['15', '15', '15'], [], ['17', '17'], ['18', '18', '18'], ['19', '19'], ['20']]

30 occurrences ne suffit pas pour prouver que c’est aléatoire on pourrait les tester avec différents test comme par exemple le test de Diehard.