Fonctions de hachage

Rappel Soit E un ensemble de messages de taille quelconque. On souhaite *indexer* ces messages à l'aide d'une fonction d'encodage H qui va de $\mathcal{M}$ (l'ensemble des messages possibles) vers l'intervalle $[0,...,n[$.

La *fonction de hachage* :

- "calcule" la valeur de l'identifiant à partir du message de départ.
- doit être telle que la probabilité de choisir le même identifiant pour deux messages différents soit extrêmement faible.

Etape 1 : transcodage binaire des données

L'ensemble des messages possibles peut être réduit à l'ensemble des entiers naturels. En effet, chaque caractère d'un texte peut être traduit en entier en interprétant le code binaire correspondant comme un entier.

Il existe différents encodages binaires possibles pour les caractères :

- le code ASCII code les caractères du clavier anglais sur 7 bits, ce qui permet d'interpréter chaque caractère comme un entier entre 0 et 127
  - ainsi :

```
code = ord('/')
print(code)
```

  - affiche la valeur 47 (le code ASCII du caractère `'/'`)
- La norme UTF-8 encode les caractères sur un nombre d'octets variant entre 1 et 4. Il permet ainsi de coder un nombre de caractères considérablement plus élevé.
  - Exemple : le smiley `'□'` appartient à la norme utf-8. Pour obtenir la valeur entière correspondante :

```
code = ord('□')
print(code)
```

- On peut inversement afficher le caractère à partir de son code entier :

```
print(chr(233))
print(chr(119070))
```

Pour traduire une *chaîne de caractères* en entier, il faut "construire un nombre" à partir de chaque caractère de la chaîne en prenant en compte sa position.

- ainsi, dans le système décimal, la position du chiffre dans le nombre définit à quelle puissance de 10 il appartient (unité, dizaines, centaines, etc...) Le chiffre le plus à gauche a la puissance la plus élevée et celui le plus à droite la puissance la plus faible.

- Si on suppose, pour simplifier, que chaque caractère est codé par un entier entre 0 et 255 (soit le code ASCII "étendu"), alors toute séquence de caractères (de claviers européens) exprime un nombre en base 256.
    - Un tel nombre s'appelle un "bytestring" en python.
    - Il existe une fonction `encode` qui effectue une telle traduction
        - Exemple :

```
s = 'paul.poitevin@centrale-marseille.fr'
b = s.encode()
```

Un nombre en base 256 est difficile à lire et interpréter. On le traduit en base 10 :

```
i = int.from_bytes(b, byteorder='big')
print("i =", i)
```

Ce qui donne :

```
i =
852806586114976881510056778471769180697471859150728801241505293627173168229624211058
```

Ce code est difficilement exploitable. Rappelons que l'on souhaite un code relativement compact pour indexer nos données (ici des adresses e-mail)

Etape 2 : Réduction du code

L'opérateur modulo (`%` en python) donne le reste de la division entière par le deuxième opérande.

Soit `H_code` notre fonction de hachage :

```
def H_code(s, n):
     b = s.encode()
     i = int.from_bytes(b, byteorder='big')
     return i % n
```

- Avantage :
    - le code retourné est plus compact
- Inconvénient:
    - si n est une puissance de 2, ce codage revient à sélectionner les bits de poids faible. En effet, pour un nombre exprimé en base 2, la division entière par 2^h a pour résultat les bits de poids fort $>h$ et pour reste les h bits de poids faible.
        - deux données différentes peuvent alors avoir le même code si elles se terminent de la même façon
            - Exemple : $n = 2^{32} = 4294967296$:
                - H_code(`paul.poitevin@centrale-marseille.fr`, n) = 1697539698
                - H_code(`martin.mollo@centrale-marseille.fr`, n) = 1697539698
        - deux données proches ou très similaires auront un index proche ou similaire : si `j = i + 1`, `H(j) = H(i) + 1` (presque toujours)

- Exemple : $n = 2^{32} = 4294967296$:
  - H_code(`paul.poitevin@centrale-marseille.fr`, n) = `1697539698`
  - H_code(`paul.poitevin@centrale-marseille.fs`, n) = `1697539699`

—> il faut prendre n *premier* : en effet, si n est premier > 2, ses multiples ne sont pas des puissances de 2, et la division entière par n présente un risque faible de ne sélectionner que les bits de poids fort. **Exemple** :

- $n = 67280421310721$ (premier):
  - H_code(`paul.poitevin@centrale-marseille.fr`, n) = `1804706371`
  - H_code(`martin.mollo@centrale-marseille.fr`, n) = `3579559911`

Néanmoins, de par les propriétés de la division entière, le reste de `(m + 1) / n` vaut `1 + m % n` presque toujours.

Exemple :

- $n = 67280421310721$ (premier):
  - H_code(`paul.poitevin@centrale-marseille.fr`, n) = `1804706371`
  - H_code(`paul.poitevin@centrale-marseille.fs`, n) = `1804706372`

Etape 3 : combiner produit et modulo

Soient m et n deux nombres premiers entre eux :

```
def H_code(s, m, n):
     b = s.encode()
     i = int.from_bytes(b, byteorder='big')
     return (i * m) % n
```

- Avantage :
- deux entiers proches donneront auront des codes très différents : si `j = i + 1`, `j * m - i * m = m`

–> il est préférable de prendre $m > n$, pour que les valeurs entières i et $i+1$ produisent des codes arbitrairement éloignés sur l'intervalle $[0, ..., n-1]$.

- Inconvénient :
  - deux données différentes peuvent toujours avoir le même code
  - le produit `i * m` peut etre coûteux à calculer

Collision de codes

On appelle *collision* le fait que deux messages différents s1 et s2 produisent un code identique, i.e. $$H(s_1)=H(s_2)\text{ avec }s_1 \neq s_2$$

On vous donne les fonctions d'encodage et de décodage suivante :

```
def encode(s):
    # chaine de caractères --> entiers naturels
    if type(s) == bytes:
        b = s
    else:
        b = s.encode()
    i = int.from_bytes(b, byteorder='big')
    return i
```

```
def decode(i):
    # entiers naturels --> chaîne de caractères
    h = hex(i)
    try:
        b = bytes.fromhex(h[2:])
    except:
        b =  bytes.fromhex('0' + h[2:])
    try:
        s = b.decode() #"replace")
    except:
        s = b
    return s
```

ainsi que la fonction de hachage :

```
P1 = 67280421310721
P2 = 32416188517
def H_code(s, p = P1, m = P2):
    i = encode(s)
    return (i * p) % m
```

Il est assez simple de trouver deux messages de même code en modifiant les derniers caractères.

Par exemple, si on part de l'adresse mail:

```
print(H_code("paul.poitevin@centrale-marseille.fr"))
```

14361131238

```
print(H_code("paul.poitevin@centrale-marseiy0txYG"))
```

14361131238

QUESTION

Sans rien changer aux fonctions `encode`, `decode` et `H_code`, donner une chaîne de caractères qui a le même H-code que votre adresse de centrale *en modifiant les premiers caractères de l'adresse mail uniquement*.

Exemple :

- Votre adresse mail : `paul.poitevin@centrale-marseille.fr`
- Réponse attendue : `w)a*9.oitevin@centrale-marseille.fr`

```
<!-- * Pour répondre, vous devez remplir le formulaire à l'adresse suivante : {{https://goo.gl/forms/0TzUG5eQZEmZDb4L2|https://goo.gl/forms/0TzUG5eQZEmZDb4L2}} -->
```

From:
https://wiki.centrale-med.fr/informatique/ - **WiKi informatique**

Permanent link:
https://wiki.centrale-med.fr/informatique/tc_info:private_s5-tpa2

Last update: **2020/12/11 11:01**