TP nº 9 - Tables de hachage

Exercice 1. *En* python

En *python* les tables de hachage sont appelées *dictionnaires*. Dans un dictionnaire, on associe une *valeur* à une *clé*.
Les clés peuvent être de presque n'importe quel type : entiers, chaînes de caractères, fonctions, éléments d'une classe, etc. (mais pas une liste python).
On peut créer un dictionnaire de plusieurs manières :

```
- d = {"a" : 12, "blop" : "blip", 42 : [1, 2, 3]}
- d = {}
  d["a"] = 12
  d["blop"] = "blip"
  d[42] = [1, 2, 3]
- l = [("a", 12), ("blop", "blip"), (42, [1, 2, 3])]
  d = dict(l)
```

Pour accéder à la valeur d'un dictionnaire `d` correspondant à la clé `k` on utilise la syntaxe `d[k]`. Par exemple, si `d` est le dictionnaire défini prédemment, `d["blop"]` vaut la chaîne de caractères `"blip"`.

1. Choisissez 5 mots de la langue française et créez un dictionnaire qui associe à chacun de ces mots sa traduction en anglais.

R.

```
dic = {"chat": "cat", "chien": "dog", "vache": "cow", "tigre": "tiger",
    "licorne": "unicorn"}
```

2. Ajoutez une entrée au dictionnaire de la question précédente (un nouveau mot et sa définition).

R.

```
dic["souris"] = "mouse"
```

Pour savoir si une clé `k` est présente dans un dictionnaire `d` on peut utiliser la syntaxe `d.has_key(k)` (qui renvoie `True` ou `False`).

3. Écrivez une fonction `ajoute(mot1, mot2, d)` qui prend en argument un mot en français, sa traduction en anglais et ajoute ces deux mots dans le dictionnaire `d` uniquement si `mot1` n'est pas une clé du dictionnaire (si `mot1` apparaît dans `d` la fonction ne fait rien).

R.

```
def ajoute(mot1, mot2, d):
  if not d.has_key(mot1):
    d[mot1] = mot2
```

Pour obtenir la liste de toutes les clés du dictionnaire, on utilise la syntaxe `d.keys()` qui renvoie une liste python contenant les clés.

4. Écrivez une fonction qui affiche à l'écran toutes les valeurs correspondant aux clés qui sont dans votre dictionnaire (ici, tous les mots en anglais qui apparaissent dans votre dictionnaire).
Indication : on exécute une boucle `for` sur tous les éléments de `d.keys()` et l'on renvoie pour chacun la valeur qui lui est associée.

R.

```
def valeurs(d):
  for k in d.keys():
    print d[k]
```

On aurait aussi pu utiliser la fonction `d.values()` qui renvoie une liste contenant les valeurs du dictionnaire :

```
def valeurs(d):
  for v in d.values():
    print v
```

Pour supprimer une entrée du dictionnaire, on peut utiliser la fonction `del` (qui permet de supprimer une variable quelconque de manière générale). Ainsi, dans l'exemple initial, pour supprimer l'entrée correspondant à la clé `"blop"` on peut utiliser l'instruction `del(d["blop"])`.

5. Écrivez une fonction `delete(char, dict)` qui prend en argument un caractère `char` et un dictionnaire `dict` et supprime du dictionnaire toutes les entrées correspondant à des clés qui commencent par la lettre `char`.

R.

```
def delete(char, dict):
  for k in dict.keys():
    if k[0] == char:
      del(d[k])
```

Exercice 2. *Permutations*

On considère des clés sur un ensemble de 256 caractères (l'alphabet ASCII 8 bits par exemple) et l'on associe à chaque clé l'entier qu'elle représente en base 256.
Ainsi, par exemple, puisque les caractères `B`, `l`, `o` et `p` correspondent aux valeurs 66, 108, 111 et 112 respectivement, la clé « Blop » est associée à l'entier

$$66.256^3 + 108.256^2 + 111.256 + 112 = 1114402672$$

1. Écrivez une fonction *python* qui prend en entrée une chaîne de caractères en ASCII 8 bits et renvoie l'entier associé.
Indication : On pourra utiliser la fonction `ord(c)` qui renvoie la valeur ASCII du caractère `c`.

R.

```
def str_to_int(s):
  n = 0
  for c in s:
    n = n * 256 + ord(c)
  return n
```

Remarque : Cette fonction utilise le principe de la méthode de Horner pour évaluer le polynôme

$$a_0X^d + a_1X^{d-1} + \ldots + a_{d-1}X + a_d$$

en $X = 256$, où $a_0, a_1, \ldots a_d$ sont les entiers correspondant aux lettres de la chaîne s passée en argument.
On peut aussi écrire une fonction au fonctionnement plus simple mais moins efficace :

```
def str_to_int(s):
  n = 0
  for i in range(len(s)):
    n = n + ord(s[i]) * 256 ** (len(s) - i - 1)
  return n
```

Si l'on utilise la fonction de hachage

$$h : \quad x \mapsto (x \mod 255)$$

pour tout mot x, si un mot y est obtenu à partir de x par permutation de ses lettres (mêmes lettres, même nombre d'occurrences, mais l'ordre est quelconque) alors $h(x) = h(y)$.

2. Écrivez la fonction h en *python* qui prend en argument une chaîne de caractères, la convertit en entier puis le hache et vérifiez la propriété annoncée sur quelques exemples.

R.

```
def hachage(s):
  return str_to_int(s) % 255
```

On peut ensuite remarquer que `hachage("chien")` et `hachage("niche")` renvoient la même valeur (en l'occurrence 9).

Remarque : Dès que les mots sont un peu longs (c'est déjà le cas pour « chien » et « niche »), le résultat de la fonction `hachage` est un *entier long* (ce qui est indiqué en *Python* par la lettre « L » après la valeur numérique), bien que sa valeur soit petite (comprise entre 0 et 255). C'est dû au fait que le calcul intermédiaire (la fonction `str_to_int`) passe par un entier très grand avant de calculer le résultat modulo 255. On pourrait alléger le calcul en effectuant le modulo après chaque étape de calcul dans le calcul de la fonction `str_to_int` :

```
def hachage(s):
  n = 0
  for c in s:
    n = (n * 256 + ord(c)) % 255
  return n
```

Or, on remarque en faisant ceci que puisque $256 \equiv 1\ [255]$ la ligne

```
    n = (n * 256 + ord(c)) % 255
```

peut être remplacée par

```
    n = (n + ord(c)) % 255
```

ce qui simplifie encore le calcul, et permet de comprendre pourquoi l'ordre des lettres dans le mot de départ n'a pas d'importance...

3. Expliquez.

R. Comme il a été dit dans la réponse à la question précédente, puisque $256 \equiv 1\ [255]$, il est également vrai que pour tout k, $256^k \equiv 1\ [255]$. Ainsi, la fonction de hachage associe à une suite d'entiers $a_0, a_1, \ldots, a_d$ correspondant à une chaîne de caractères s la valeur

$$\begin{array}{rl} & (a_0.256^d + a_1.256^{d-1} + \ldots + a_{d-1}.256 + a_d) \mod 256 \\ = & (a_0 + a_1 + \ldots + a_{d-1} + a_d) \mod 256 \end{array}$$

Le résultat est donc simplement la somme des valeurs de chaque lettre, modulo 256, et ne dépend donc pas de l'ordre dans lequel les lettres apparaissent dans le mot de départ.

Exercice 3. *Le paradoxe des anniversaires*

1. Si l'on considère un groupe de N personnes, quelle est la probabilité que deux d'entre elles soient nées le même jour de l'année ? (donnez simplement une expression de la probabilité)

R. On calcule d'abord la probabilité que toutes les personnes soient nées un jour différent (on néglige le 29 février et on compte donc une année de 365 jours). La première personne ne pose pas de problème, la seconde a une probabilité 364/365 d'être née un jour différent de la première, la troisième a une probabilité 363/365 d'être née un jour différent des deux précédentes, et ainsi de suite. Finalement la probabilité que tous soient nés un jour différent est

$$\prod_{i=1}^{N-1} \frac{365-i}{365}$$

et donc la probabilité que deux personnes aient un anniversaire en commun est

$$1 - \prod_{i=1}^{N-1} \frac{365-i}{365}$$

2. Écrivez la fonction python `anniversaires(n)` qui calcule la valeur numérique de la probabilité qu'il existe 2 personnes parmi un groupe de `n` personnes ayant leur anniversaire le même jour.
En particulier, combien vaut cette probabilité pour un groupe de 23 personnes ?

R.

```
def anniversaire(n):
  p = 1.
  for i in range(1, n):
    p = p * (365-i)/365
  return 1 - p
```

Remarque : Pour des raisons d'arrondis, cette fonction déclare que la probabilité pour plus de 153 personnes est 1. On pourrait calculer la valeur exacte sous la forme d'un rapport de deux entiers en utilisant la fonction

```
def anniversaire(n):
  numerateur = 1
  for i in range(1, n):
    numerateur = numerateur * (365-i)
  denominateur = 365**(n-1)
  return (denominateur - numerateur, denominateur)
```

Cette fonction renvoie un couple qui représente le numérateur et le dénominateur de la probabilité recherchée. Comme *Python* est capable de gérer des entiers arbitrairement grands il n'y a aucune erreur d'arrondi.

3. Quel est le rapport avec les tables de hachage ?

R. La probabilité d'avoir une collision (deux clés hachées sur la même valeur) en insérant n clés dans une table de hachage de taille 365 est la même que la probabilité d'avoir un conflits d'anniversaires. De manière générale, en remplaçant 365 par la taille de la table de hachage, on a la probabilité d'avoir une collision en insérant n valeurs.

4. Généralisez la fonction `anniversaire` pour qu'elle calcule la probabilité que l'on obtienne une collision en ajoutant `n` valeurs dans une table de hachage de taille `m`.

R.

```
def collisions(n, m):
  p = 1.
  for i in range(1, n):
    p = p * (m-i)/m
  return 1 - p
```

Remarque : On peut vérifier expérimentalement à l'aide de cette fonction que le nombre de clés à insérer pour avoir une probabilité fixée d'avoir une collision est de l'ordre de $\sqrt{m}$ (le plus simple est de calculer les valeurs `collisions`(n, n^2) pour des valeurs de n de plus en plus grandes et de remarquer que la probabilité converge).