Introduction aux tables de hachages (pa.dilla.fr/14)

Dans le programme 2 du chapitre *Modularité*, `s` est un tableau définit par `s=[]` :

- le tableau `s` est petit et prend peu de place en mémoire
- **mais** la recherche `if x in s:` n'est pas immédiate. Dans *le pire des cas*, il faut parcourir tout le tableau pour être certain que `x` n'y est pas.

Au contraire, dans le programme 3, `s` est définit par `s = [False] * 366` :

- la recherche `if s[x]:` est immédiate
- **mais** le tableau `s` prend beaucoup de place en mémoire.

Le programme 4 ci-dessous te propose une solution qui prend le meilleur des deux tentatives précédentes :

- peu de place en mémoire (comme le programme 2)
- quasi immédiateté de la recherche (comme le programme 3).

```
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = [[] for _ in range(23)]
    for x in t:
        if x in s[x % 23]:
            return True
        s[x % 23].append(x)
    return False
```

REMARQUE

On crée un tableau `s` de 23 cases car on sait qu'il n'y aura 23 dates à y enregistrer. L'occupation en mémoire est donc faible.

Ensuite, on attribut à chacune des 365 dates possibles une case *fixe et bien définie*. Par exemple, la date 42 sera toujours rangée dans `s[19]`.

Comment sait-on que la date `42` est enregistrée dans l'emplacement `19` de `s` ? Pour obtenir ce rang (`19`) associé à la date `42`, on utilise l'opération **modulo 23** (notée `% 23`) qui renvoie le *reste de la division euclidienne par 23*. Cette opération renvoie un nombre compris entre 0..22. Ce qui est parfait car le tableau `s` contient 23 emplacements. Le calcul du rang `42 % 23` donne pour résultat `19` et est immédiat.

REMARQUE

Mais il est **possible** que plusieurs dates soient dans la même case. Par exemple les dates 65 ou 88 se rangeront aussi dans `s[19]`.

C'est pourquoi chaque case de `s` ne contient pas une date, mais un tableau de date que nous appellerons **paquet**. C'est donc pourquoi `s = [[] for _ in range(23)]`.

Pour rendre la recherche *quasi* immédiate, il faut que chaque paquet soit quasiment vide. Puisque le tirage des 23 dates est aléatoire, il faudrait beaucoup beaucoup de malchance pour qu'un grand nombre d'entre elles se trouvent dans le même paquet.

Par exemple, imaginons que la date tirée soit 42. On sait *immédiatement* qu'il faut chercher dans le paquet `s[19]`. Maintenant si ce paquet contient beaucoup de dates (pas de chance !) la recherche prend

du temps. Sinon, le paquet est quasiment vide et la recherche est *quasi immédiate* !

On peut donc conclure que, *en moyenne*, la recherche `x in s[x % 23]` est *quasi immédiate*.

La méthode exposée ci-dessus est une ébauche de la structure de données fondamentale **table de hachage**.
Cette structure de données est sous-jacente aux **ensembles** et aux **dictionnaires** de Python. Elle est très polyvalente, permet de représenter des ensembles de taille arbitraire avec des opérations d'accès aux éléments extrêmement rapides. Cette structure de données est considérée comme la plus efficace dans la plus grande variété des cas courants !

Amélioration du modèle de tables de hachages

Dans la suite, nous allons améliorer notre modèle de *tables de hachage* afin de :

Représenter des ensembles de taille *arbitraire* de façon efficace. Par exemple, ne plus limiter à 23 paquets !

Rendre vraiment aléatoire la répartition dans les paquets. Par exemple, faire en sorte que les multiples, les nombres proches ou liés les uns aux autres ne soient pas systématiquement associés au même paquet.

Rendre l'ensemble plus polyvalent en y associant autre chose que des nombres entiers. Par exemple on aimerait aussi associer des chaînes de caractères comme `"alice"` ou `"bob"` à différents paquets.

Voici ce que donne le programme final :

```
def cree():
    return { 'taille':0 ,
             'paquets': [[] for _ in range(32)] }

def contient(s, x):
    p = hash(x) % len(s['paquets'])
    return x in s['paquets'][p]

def ajoute(s,x):
    if contient(s, x):
        return
    s['taille'] += 1
    if s['taille'] > len(s['paquets']):
        _etend(s)
    _ajoute_aux(s['paquets'], x)

def _ajoute_aux(t, x):
    p = hash(x) % len(t)
    t[p].append(x)

def _etend(s):
    tmp = [[] for _ in range( 2 * len(s['paquets']) )]
    for x in enumere(s):
        _ajoute_aux(tmp, x)
    s['paquets'] = tmp

def enumere(s):
    tab = []
    for paquet in s['paquets']:
        tab.extend(paquet)
    return tab
```

Fonction de hachage

Commençons par les deux derniers points. Pour obtenir une vraie table de hachage, on va utiliser une fonction appelée *fonction de hachage*, qui prend en paramètre un élément à stocker (nombre entier, chaîne de caractère, flottant, objet) et renvoie un nombre entier définissant le numéro dans lequel insérer l'élément.

En utilisant cette fonction *modulo le nombre de paquets*, on s'assure que l'association entre un objet et son paquet est aléatoire et indépendant.

En Python, l'appel à une fonction de hachage se fait par la fonction `hash(obj)` où `obj` est l'objet à associer.

Pour remplir un tableau de 23 cases avec cette fonction de hachage, on remplace tout simplement `x % 23` par `hash(x) % 23`. La fonction `hash(x)` s'occupe de la répartition aléatoire et `% 23` se charge de répartir le nombre entier obtenu en une nombre appartenant à 0..22.

C'est pourquoi dans le programme on a `hash(x) % len(s['paquets'])` ou encore `hash(x) % len(t)` : on obtient le rang `p` de la date `x` en effectuant un *modulo le nombre de paquets* de la table de hachage.

Tableau de taille variable

Afin de rendre la table de hachage efficace, il faut s'assurer que chaque paquet reste le plus petit possible. En effet, si un paquet contient beaucoup d'éléments, la recherche dans ce paquet n'est plus efficace. En revanche, si le paquet est vide ou contient un ou deux éléments, la recherche est *immédiate*.

On va alors se fixer la règle suivante :

> *la taille de la table de hachage (le nombre de paquets) ne doit jamais être inférieure au nombre d'éléments stockés qui y sont stockés.*

Ainsi, c'est la procédure `ajoute(s,x)` qui s'occupe d'ajouter l'élément `x` à la

table de hachage `s` et qui a en charge d'augmenter la taille de la table.

Si `x` est déjà présent `if contient(s, x)`, on ignore toutes les instructions suivante et on appelle un `return` tout seul qui termine l'appel à la fonction `ajoute()`.

Si `x` n'est pas déjà dans `s`, alors les lignes après le `return` s'exécutent. Puisque `x` va être ajouté, on incrémente la taille de la table de hachage `s['taille'] += 1`.

Ensuite la procédure vérifie si le nombre d'éléments dépasse le nombre de paquets `if s['taille'] > len(s['paquets']):`. Si c'est le cas, la procédure `_etend(s)` va :

- créer un tableau deux fois plus grand que la table de hachage `tmp = [[] for _ in range( 2 * len(s['paquets']) )]`
- répartir tous les éléments de la table de hachage dans ce tableau temporaire `for x in enumere(s): _ajoute_aux(tmp, x)`
- faire pointer la table de hachage vers le tableau temporaire `s['paquets'] = tmp`.

Pour finir, un appel à la procédure `_ajoute_aux()` s'occupe (enfin) d'ajouter l'élément `x` dans le paquet auquel il est associé `_ajoute_aux(s['paquets'], x)`.

Test du module

[11]:
```
# créer doublon
def contient_doublon(t):
    """la structure contient-elle un doublon?"""
    s = cree()
    for x in t:
        if contient(s, x):
            return True
        ajoute(s, x)
    return False

# Création du tableau de dates aléatoires
from random import randint
```

```
n = 0
n_doublons = 0

while n < 1000 :
    t = [None] * 23
    for j in range(23):
        t[j] = randint(1,365)
    if contient_doublon(t):
        n_doublons += 1
    n += 1

print (n_doublons,"doublons sur",n,"tirages")
print ("fréquence : ", n_doublons/n)
```

```
483 doublons sur 1000 tirages
fréquence :  0.483
```