Dictionnaires et mémoïsation

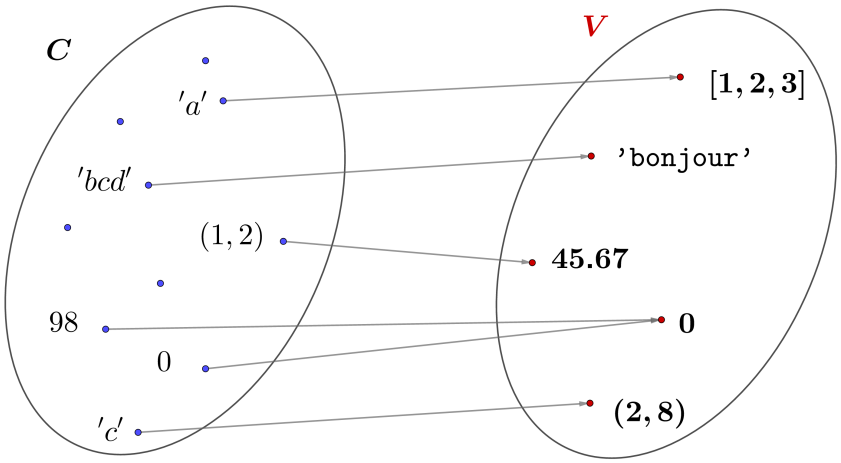
1. Savoir créer un dictionnaire, accéder à une valeur, insérer une clé et une valeur. Recopie.
2. Savoir parcourir un dictionnaire : soit par ses clés, soit par ses valeurs, soit par ses couples (clés,valeurs).
3. Principe de fonctionnement d'un dictionnaire (présentation du hachage).
4. Utilisation d'un dictionnaire pour la mémoïsation.

---

1 Dictionnaires

1.1 création, manipulation des dictionnaires

Un dictionnaire est un ensemble non ordonné de couples (clé, valeur). Les clés d'un dictionnaire doivent être distinctes et de type immuable. Il n'y a pas de contraintes pour les valeurs.




```
D = dict() # ou D = {}
D['a'] = [1,2,3]
D['bcd'] = 'bonjour'
D[(1,2)] = 45.67
D[98] = 0
D[0] =0
D['c'] = (2,8)
# ou encore :
D = {'a': [1, 2, 3], 'bcd': 'bonjour', (1, 2): 45.67, 98: 0, 0: 0, 'c': (2, 8)}
```

On peut aussi créer un dictionnaire par compréhension :

```
1 D = {k: 0 for k in range(10)}
```

Syntaxe sur les dictionnaires

• à connaître

Pour vérifier si une clé est dans le dictionnaire, on utilise `cle in D`, qui renvoie `True` ou `False`.
L'itération sur les clés `for k in D.keys():` peut s'écrire plus simplement `for k in D:` et parcourt toutes les clés du dictionnaire.

• éventuellement, dans une aide fournie

On peut vouloir parcourir l'ensemble des clés d'un dictionnaire, l'ensemble des valeurs d'un dictionnaire,ou parcourir tous les couples (clés,valeurs). Trois méthodes sont associées :

— `D.keys()` renvoie la liste des clés du dictionnaire `D`,

— `D.values()` renvoie la liste des valeurs présentes dans le dictionnaire `D`,

— `D.items()` renvoie la liste des couples (clés, valeurs) du dictionnaire `D`.

ce qui donne les parcours possibles d'un dictionnaire suivants :

```
for k in D.keys():    for v in D.values():    for (k, v) in D.items():
```

Enfin, la fonction `len` et la méthode `copy`, rencontrées sur les listes, existent pour un dictionnaire.
`len(D)` retourne le nombre de couples (clé,valeur) du dictionnaire.
`D.copy()` retourne une copie du dictionnaire `D`.

1.2 le principe du hachage

Pour que la manipulation d'un dictionnaire soit efficace, il faut pouvoir accéder rapidement à la valeur associée à une clé. Lors du test `cle in D`, `Python` ne fait pas un parcours systématique de toutes les clés du dictionnaire jusqu'à trouver celle correspondant à `cle`. Une telle méthode serait de complexité $\Theta(n)$ où n est le nombre de clés du dictionnaire, et serait trop lente pour des dictionnaires de millions d'éléments.
La recherche d'une clé se fait en réalité en temps constant (c'est-à-dire avec une complexité $\Theta(1)$, indépendante de la taille du dictionnaire) grâce aux fonctions de hachage.

Pour retrouver rapidement une clé dans un dictionnaire, `Python` utilise une fonction de hachage (par exemple `hash`, mais il y en a d'autres) qui à un objet de type *hachable* associe un entier.

`hash(87654), hash(3.14), hash((1,2)), hash('bonjour')` renvoie
`(87654, 322818021289917443, 3713081631934410656, 5776615754334337163)`.
`hash([1,2,3])` renvoie une erreur, car une liste n'est pas de type hachable.
Une condition nécessaire pour être de type hachable est d'être de type immuable, ou composé (par exemple pour un tuple) d'objets de types immuables.

Explication du principe général de l'implémentation d'un dictionnaire et de recherche d'une clé

Il est défini un nombre entier m (largeur de hachage) et, quitte à travailler modulo m, une fonction de hachage h à valeurs dans $[\![0, m-1]\!]$. Le nombre de clefs possibles étant très important, le cardinal de l'ensemble des clés est beaucoup plus grand que m et une telle fonction ne sera pas injective. Imaginons une liste `tiroirs` de casiers (`tiroirs`

est de taille m).

Quand on veut stocker (c, v) dans le dictionnaire, la valeur $i = h(c)$ est calculée. Dans le casier `tiroirs[i]` est stocké le couple (c, v).
Et quand on veut chercher si la clé c est dans le dictionnaire, on calcule $i = h(c)$ et on n'a qu'à regarder dans le casier `tiroirs[i]` si c y est.
Même chose pour modifier une valeur du dictionnaire. Schématiquement :

`tiroirs[0]`	
`tiroirs[1]`	(c_3, v_3), (c_5, v_5), (c_6, v_6)
`tiroirs[2]`	
`tiroirs[3]`	(c_2, v_2)
`tiroirs[4]`	
`tiroirs[5]`	
`tiroirs[6]`	(c_1, v_1) et (c_4, v_4)

Il y a des *collisions* : des clés différentes conduisent à une même valeur de hachage. Mais si la répartition entre les différents paquets est équilibrée, chaque paquet ne contiendra qu'un petit nombre de valeurs, et on retrouvera rapidement la valeur associée à une clé en la cherchant dans son paquet. Si on considère que le calcul de $h(c)$ se réalise en temps constant et que chaque paquet est de petite taille, on comprend que les opérations sur les dictionnaires requièrent une complexité temporelle constante.

Une fonction de hachage a d'autres usages. Par exemple, lorsque vous choisissez un mot de passe sur un site sécurisé, ce dernier ne va pas stocker votre mot de passe en clair, mais va stocker le résultat de l'application d'une fonction de hachage h à votre mot de passe. Sachant qu'il est très difficile de trouver les antécédents d'un entier par la fonction h, pirater le site ne mettra pas en cause la sécurité de votre mot de passe.
Une autre application des fonctions de hachage est le contrôle d'intégrité d'un fichier informatique : à chaque fichier est associée une valeur par une fonction de hachage qui permet de vérifier si un fichier téléchargé a été transmis correctement.

2 Utilisation d'un dictionnaire pour la mémoïsation

Définition 1

La *mémoïsation* d'une fonction consiste à mettre en place, avec un dictionnaire, une mémorisation des calculs effectués pour éviter de refaire plusieurs fois un même calcul.
La clé sera égale aux arguments d'entrée de la fonction et la valeur sera le résultat de la fonction.

2.1 l'exemple-type de la suite de Fibonacci

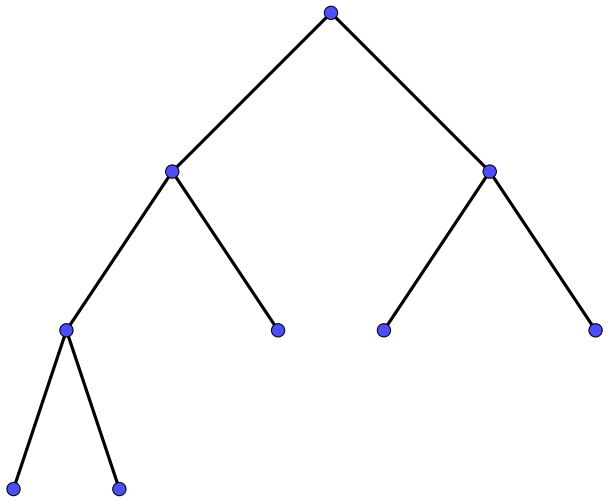
Exercice 1 : On considère la suite de Fibonacci donnée par $\begin{cases} F_0 & = 0 \\ F_1 & = 1 \\ F_{n+2} & = F_{n+1} + F_n \text{ pour } n \in \mathbb{N} \end{cases}$

1. **Méthode ascendante, de « bas en haut »** (*bottom-up* en anglais).
   Dans cette méthode, on commence par résoudre les sous-problèmes les plus petits, et, de proche en proche, on résout de manière itérative les problèmes plus gros.

« Calcul de bas en haut » : des petits problèmes vers les gros problèmes

Écrire une fonction `Fibo_iteratif` qui calcule F_n à l'aide d'une boucle `for`, en stockant les résultats successifs dans une liste. Donner la complexité temporelle de cet algorithme.

2. **Approche « de haut en bas »** (*top-down* en anglais)
   (a) Écrire une fonction `Fibo_recursif` pour le calcul de F_n utilisant un algorithme récursif : des gros problèmes vers les petits problèmes.
   (b) Compléter l'arbre suivant pour visualiser, lors de l'exécution `Fibo_recursif(4)`, le nombre d'appels à F_0, F_1, F_2, F_3.



Il s'avère que le calcul récursif de F_{47} par ce programme nécessite plus de 12 milliards d'appels à F_1 !
Nous admettons que la complexité temporelle de ce programme est $\Theta(\varphi^n)$ où $\varphi = \frac{1+\sqrt{5}}{2}$.

3. **Approche « de haut en bas » avec mémoïsation**

```python
Memo = dict()

def fibonacci(n:int) -> int:
    if n == 0:
        return 0
    if n == 1:
        return 1
    if n not in Memo:
        # si on arrive ici c'est que Fn n'a pas encore été calculé
        Memo[n] = fibonacci(n-1) + fibonacci(n-2)
    return Memo[n]
```

Donner la complexité temporelle de cette approche récursive avec mémoïsation.
La complexité en mémoire est $\Theta(n)$ puisque le calcul de `fibonacci(n)` provoque le stockage de toutes les valeurs de `fibonacci(k)` pour $2 \leqslant k \leqslant n$.