Dictionnaires et programmation dynamique

Plan

Éléments de cours

- Dictionnaire ☐
- Fonction de hashage ☐
- Collision ☐
- Programmation dynamique ☐
- Chevauchement ☐
- Mémoïsation ☐
- Propriété de sous-structure optimale ☐

Capacités exigibles

- Dictionnaires, clés et valeurs.
- On présente les principes du hachage, et les limitations qui en découlent sur le domaine des clés utilisables.
- Syntaxe pour l'écriture des dictionnaires. Parcours d'un dictionnaire.
- Programmation dynamique. Propriété de sous-structure optimale. Chevauchement de sous-problèmes.
- Calcul de bas en haut ou par mémoïsation. Reconstruction d'une solution optimale à partir de l'information calculée.
- La mémoïsation peut être implémentée à l'aide d'un dictionnaire. On souligne les enjeux de complexité en mémoire.

I Dictionnaires

A Rappels et introduction

Définition : Dictionnaire

Un **dictionnaire** est un objet numérique stockant des **valeurs** (ensemble V), en les associant chacune à une **clé** unique (ensemble C).

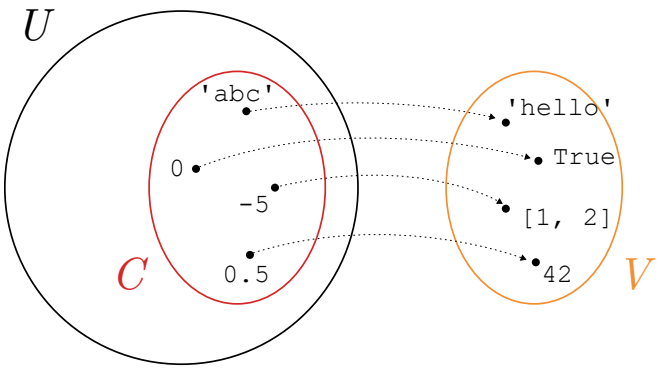
On peut le voir mathématiquement comme une application de C dans V.

Remarque

- En PYTHON, une clé peut être n'importe quel objet immutable. Les plus courants sont : `str`, `int`, `tuple`, `bool`. On notera U l'ensemble de toutes les clés possibles.
- On peut voir un dictionnaire comme une liste dont les indices peuvent être n'importe quel objet immutable.

Exemple

On peut représenter un dictionnaire ainsi :



En PYTHON, on pourrait créer ce dictionnaire ainsi :

```
>>> dic = { 'abc': 'hello', 0: True, -5: [1, 2], 'coucou': 42 }
```

Et on accèderait à ses éléments de la manière suivante :

```
>>> dic['abc']
'hello'
>>> dic[0]
True
>>> dic[-5]
[1, 2]
>>> dic[0.5]
42
```

B Fonctionnement interne

Rappel

Pour un langage donné (comme Python), il est difficile d'envisager U (l'espace de toutes les clés possibles). Sa taille étant tellement grande, retrouver un élément spécifique est une opération très coûteuse.

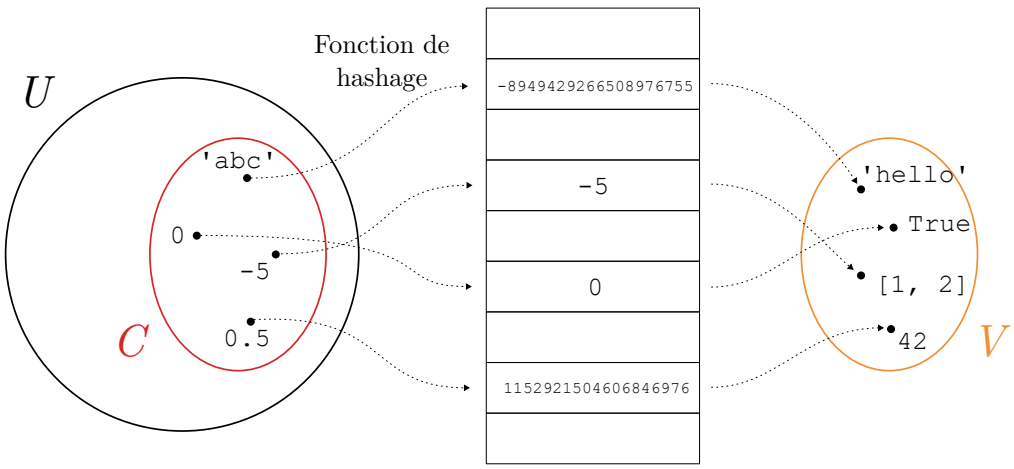
Définition : Fonction de hashage

Pour traiter plus efficacement les clés, Python doit tout d'abord chacune les convertir en un entier relatif. À partir de cet entier, il peut alors facilement retrouver la valeur associée, comme il le ferait pour une liste quelconque.

La fonction permettant la conversion des clés en entiers est appelée **fonction de hashage**.

Exemple

Dans l'exemple précédent, on peut ajouter un tableau fictif faisant l'intermédiaire entre les clés et les valeurs.



Remarque

En informatique, le terme **fonction de hashage** désigne de manière générale une fonction associant à n'importe quel objet immutable, un entier relatif. On utilise ce concept ici dans le cas des clés d'un dictionnaire.

Exemple

Il existe une fonction de base de Python, permettant d'effectuer un tel hashage. Il s'agit de la fonction `hash`, qui renvoie un nombre entre -2^{64} et $2^{64}-1$:

```
>>> hash(0)
0
>>> hash(-5)
-5
>>> hash(True)
1
>>> hash('abc')
-8949429266508976755
>>> hash(0.5)
1152921504606846976
```

Définition : Collision

La fonction de hashage n'est pas nécessairement injective (deux clés peuvent être hashées en un même entier). On parle alors de **collision**.

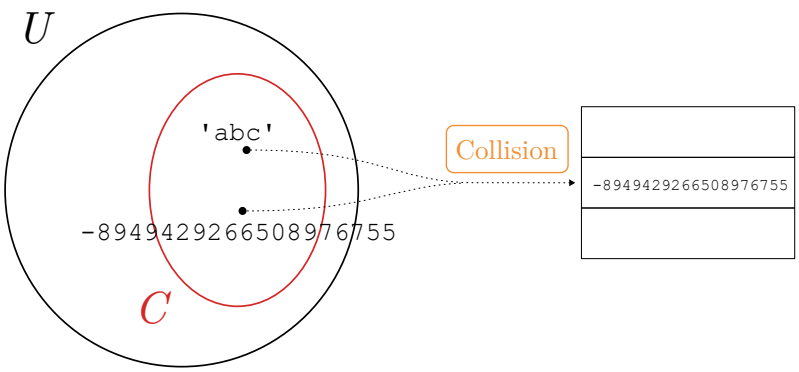
Remarque

On pourrait empêcher ce genre de problème en limitant l'espace possible pour les clés, afin de supprimer celles étant hashées de le même manière qu'une autre.

Exemple

Selon les exemples précédents, on peut facilement fabriquer des collisions :

```
>>> hash('abc') == hash(-8949429266508976755)
True
```



Et pourtant PYTHON est capable de créer un dictionnaire avec ces deux éléments comme clés :

```
>>> dic = {'abc': 'hello', -8949429266508976755: 'world'}
>>> dic['abc']
'hello'
>>> dic[-8949429266508976755]
'world'
```

C'est donc

- soit que PYTHON utilise une autre fonction de hashage ;
- soit qu'il existe des moyens de gérer les collisions.

Remarque

En pratique, il existe effectivement plusieurs solutions pour s'affranchir du problème de collisions, mais cela dépasse le cadre du programme.

II Programmation dynamique

A Exemple de la suite de Fibonacci

✔ Exemple

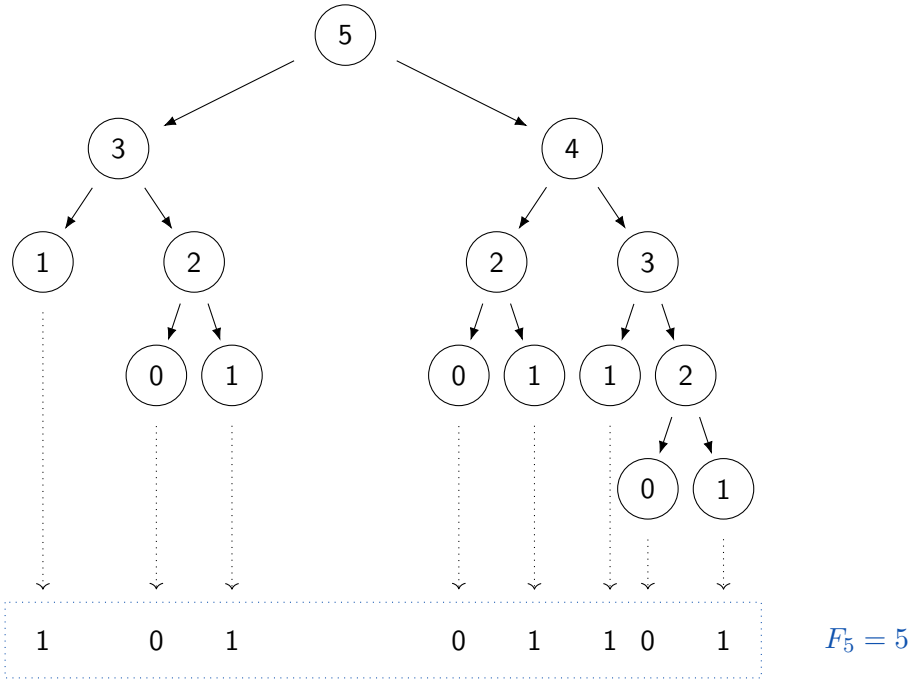
On traitera cette partie par l'étude d'un exemple simple : celui de la suite de FIBONACCI définie telle que

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ \forall n \in \mathbb{N}, \quad F_{n+2} = F_n + F_{n+1} \end{cases}$$

On peut imaginer plusieurs versions très simples d'algorithmes pour obtenir calculer F_n, par exemple par méthode récursive :

```
1  def Fibo(n):
2      if n <= 1:
3          return n     # Fibo(0) = 0 ; Fibo(1) = 1
4      else:
5          return Fibo(n-1) + Fibo(n-2)
6
```

Mais alors pour calculer `Fibo(5)`, on sera amenés à appeler plusieurs fois `Fibo(3)` et de même pour `Fibo(2)` :



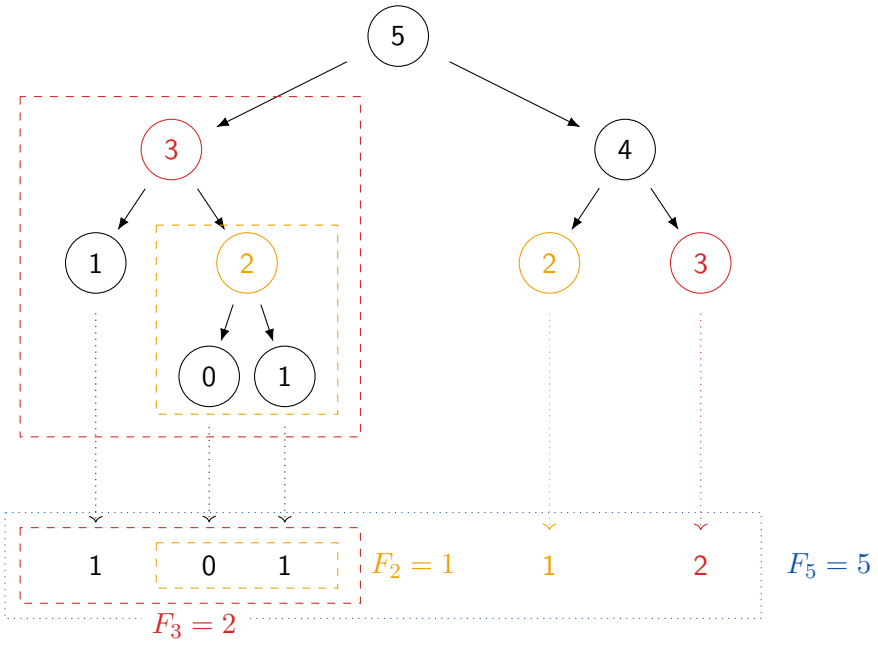
L'idée est donc de garder en mémoire les résultats en fur et à mesure, afin de les réutiliser le moment venu. Les dictionnaires sont particulièrement adaptés à cette fin :

```
stock_fibo = { 0:0, 1:1 }   # Premières valeurs de la suite

def Fibo_dynamique(n):
    if not n in stock_fibo: # Si le terme n'a pas encore été calculé
        stock_fibo[n] = Fibo_dynamique(n-2) + Fibo_dynamique(n-1)
    return stock_fibo[n]

```

Ainsi on lorsque `Fibo_dynamique(3)` est calculé une fois, sa valeur sera réutilisée plus tard, sans avoir à refaire tout son calcul (et de même pour `Fibo_dynamique(2)`) :



B Notions importantes

Définition : Programmation dynamique

Un problème de **programmation dynamique** est une situation dans laquelle on planifie la résolution, en la décomposant en sous-problèmes, puis en les résolvant du plus petit au plus grand.

Exemple

Dans la suite de FIBONACCI, le calcul d'un terme se fait à partir des deux précédents. Pour avoir le résultat final, on ajoute tous les termes triviaux obtenus F_0 et F_1.

Propriété : Chevauchement

La solution d'un problème de programmation dynamique est issue de calculs de sous problèmes **identiques**. On parle alors de **chevauchement**.

Exemple

Dans le calcul de F_5, le terme F_3 apparaît deux fois, il y a chevauchement du problème F_3. De même F_2 apparaît trois fois, donc il y a chevauchement de ce problème également.

Définition : Mémoïsation

En vertu de cette dernière propriété, dans un problème de programmation dynamique, il est toujours pertinent de garder en mémoire les résultats intermédiaires, afin de reconstruire la solution finale à partir de l'information calculée. On parle de **mémoïsation**.

Remarque

Le mot **mémoïsation**, ressemble à "*mémorisation*" de manière peu surprenante. Attention à bien respecter l'utilisation de ce terme technique.

Propriété : Propriété de sous-structure optimale

La solution optimale à un problème dynamique s'obtient en combinant les solutions optimales de ses sous-problèmes. Cette propriété est parfois également appelée **principe d'optimalité de Bellman**.

Exemple

Le calcul de F_n est optimal car il repose sur les résolutions de F_{n-1} et F_{n-2}, elles-mêmes optimales puisque le calcul final (le problème le plus trivial) consiste simplement à extraire une valeur pré-enregistrée.

Remarque

Si l'on gagne en temps de calcul, on perd cependant en mémoire. Il peut donc y avoir un compromis à trouver entre temps et données utilisées (en pratique toujours en faveur de la minimisation du temps).