

Rappels Python et programmation dynamique

ITC PC

M. Charles

Introduction

Écrire une fonction Python de signature `fib(n: int) -> int` qui prend en paramètre un nombre n et renvoie F_n défini par :

$$F_0 = 0$$

$$F_1 = 1$$

$$\forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

Écrire une fonction Python de signature `fib(n: int) -> int` qui prend en paramètre un nombre n et renvoie F_n défini par :

$$F_0 = 0$$

$$F_1 = 1$$

$$\forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

Deux solutions :

```
def fib(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Écrire une fonction Python de signature `fib(n: int) -> int` qui prend en paramètre un nombre n et renvoie F_n défini par :

$$F_0 = 0$$

$$F_1 = 1$$

$$\forall n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

Deux solutions :

```
def fib(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

```
def fib(n: int) -> int:
    if n == 0:
        return 0
    u = 0
    v = 1
    for i in range(n-1):
        temp = u + v
        u = v
        v = temp
    return v
```

Sur quels critères comparer ces deux solutions ? L'une est-elle meilleure que l'autre ?

Sur quels critères comparer ces deux solutions ? L'une est-elle meilleure que l'autre ?

- Complexité en temps :
 - Solution 1 : $O(F_n)$!
 - Solution 2 : $O(n)$

Sur quels critères comparer ces deux solutions ? L'une est-elle meilleure que l'autre ?

- Complexité en temps :
 - Solution 1 : $O(F_n)$!
 - Solution 2 : $O(n)$
- Complexité en espace :
 - Solution 1 : $O(F_n)$ également...
 - Solution 2 : $O(1)$

Sur quels critères comparer ces deux solutions ? L'une est-elle meilleure que l'autre ?

- Complexité en temps :
 - Solution 1 : $O(F_n)$!
 - Solution 2 : $O(n)$
- Complexité en espace :
 - Solution 1 : $O(F_n)$ également...
 - Solution 2 : $O(1)$

Quelle différence de fonctionnement explique une telle différence ?

Programmation dynamique

On considère des problèmes qui vérifient les conditions suivantes :

- le problème peut être décomposé en sous-problèmes plus simples ;
- une solution (optimale) au problème peut s'obtenir à partir de solutions (optimales) aux sous-problèmes ;
- les sous-problèmes ne sont pas indépendants entre eux (on parle de *chevauchement* ou de *recouvrement*).

On considère des problèmes qui vérifient les conditions suivantes :

- le problème peut être décomposé en sous-problèmes plus simples ;
- une solution (optimale) au problème peut s'obtenir à partir de solutions (optimales) aux sous-problèmes ;
- les sous-problèmes ne sont pas indépendants entre eux (on parle de *chevauchement* ou de *recouvrement*).

Dans ce cas, on peut faire de la programmation dynamique :

La **programmation dynamique** est une méthode algorithmique dans laquelle on décompose le problème en sous-problèmes plus simples et on stocke les résultats intermédiaires de calculs (on parle de *mémoïsation*).

Le calcul des termes de la suite de Fibonacci est un exemple de problème propice à la résolution par programmation dynamique :

- pour calculer F_n , on peut d'abord calculer F_{n-1} et F_{n-2} qui sont tout deux plus simples ;
- connaître F_{n-1} et F_{n-2} permet de retrouver F_n ;
- les calculs de F_{n-1} et F_{n-2} se recouvrent.

Un exemple plus compliqué

Écrire une fonction python d'en-tête `C(n: int, k: int) -> int` qui calcule le coefficient binomiale $\binom{n}{k}$ en utilisant la formule de Pascal.

```
def C(n: int, k: int) -> int:
    if k == 0 or k == n:
        return 1
    else:
        return C(n-1,k) + C(n-1,k-1)
```

```
def C(n: int, k: int) -> int:  
    if k == 0 or k == n:  
        return 1  
    else:  
        return C(n-1,k) + C(n-1,k-1)
```

Problème(s) ?

Comment peut-on stocker les résultats intermédiaires ?

Comment peut-on stocker les résultats intermédiaires ?

Solution : dictionnaires !

`M = {}`

```
def C(n: int, k: int) -> int:
    if k == 0 or k == n:
        return 1
    elif (n,k) in M:
        return M[n,k]
    else:
        M[n,k] = C(n-1,k) + C(n-1,k-1)
        return M[n,k]
```

Un code plus propre

```
def C(n: int, k: int) -> int:
    M = {}

    def C_rec(n: int, k: int) -> int:
        if k == 0 or k == n:
            return 1
        elif (n,k) in M:
            return M[n,k]
        else:
            M[n,k] = C_rec(n-1,k) +
C_rec(n-1,k-1)
            return M[n,k]

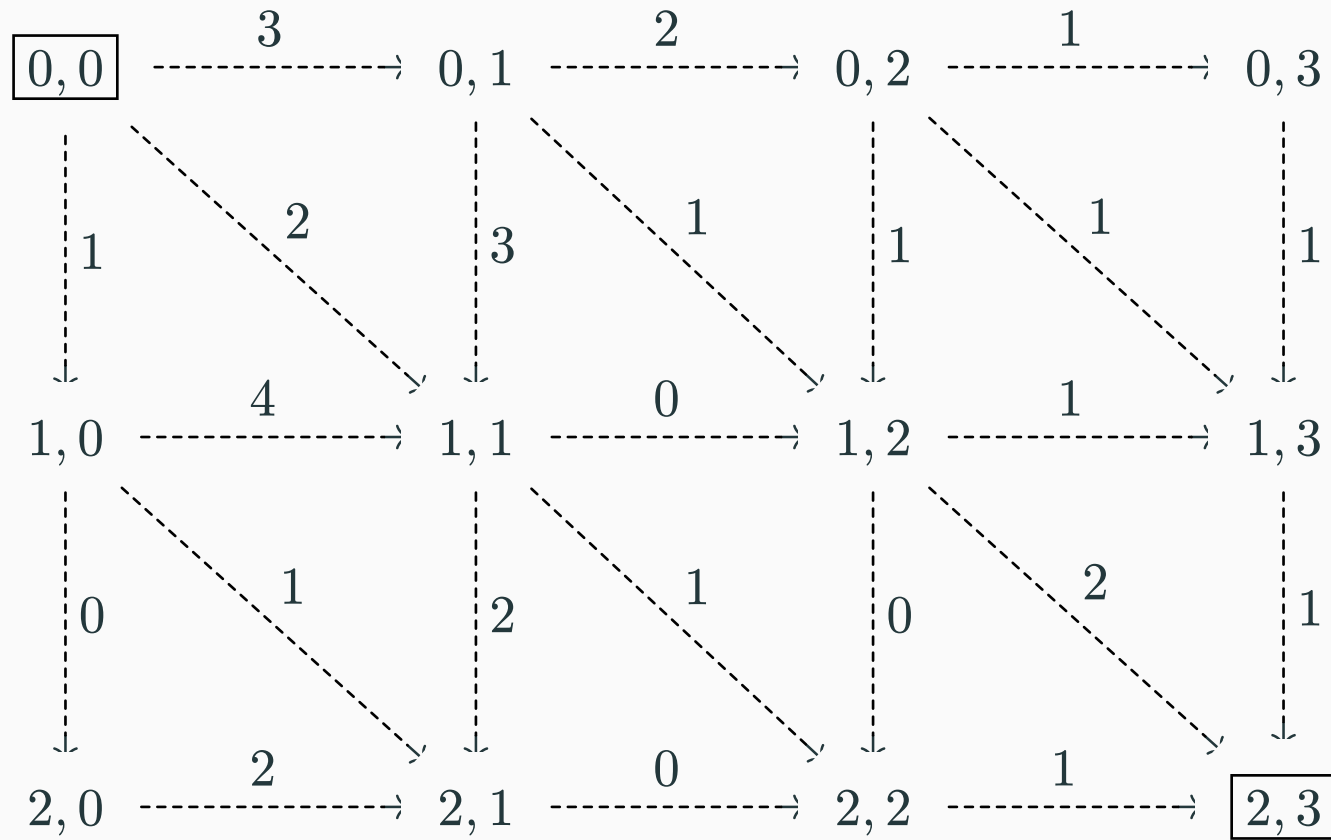
    return C_rec(n,k)
```

```
def C(n: int, k: int, M = {}) -> int:
    if k == 0 or k == n:
        return 1
    elif (n,k) in M:
        return M[n,k]
    else:
        M[n,k] = C(n-1,k, M) + C_rec(n-1,k-1,
M)
        return M[n,k]
```

On dispose d'une grille rectangulaire de $(n + 1) \times (m + 1)$ cellules, reliées par des arcs pondérés positivement dans seulement trois directions : S, E, SE.

Le but est de trouver la longueur d'un plus court chemin de $(0, 0)$ à (n, m) .

Exercice : diagramme



Exercice : solution

```
def D(n, m, poids: dict) -> int:
    dist = {}

    def D_rec(n, m) -> int:
        if (n,m) in dist:
            return dist[(n,m)]

        if m == 0 and n == 0:
            dist[(0,0)] = 0
        elif m == 0:
            dist[(n,0)] = D_rec(n-1, 0) + poids[(n-1,0), (n, 0)]
        else if n == 0:
            dist[(0,m)] = D_rec(0,m-1) + poids[(0,m-1), (0,m)]
        else:
            dist[(n,m)] = min(
                D_rec(n-1, m-1) + poids[(n-1, m-1), (n, m)],
                D_rec(n-1, m) + poids[(n-1, m), (n, m)],
                D_rec(n, m-1) + poids[(n, m-1), (n, m)]
            )

        return dist[(n,m)]

    return D_rec(n,m,poids)
```

Approche dite **top-bottom**.

Exercice : une autre solution

```
def D(n, m, poids: dict) -> int:
    dist = np.zeros((n+1, m+1)) # valeur initiale arbitraire

    dist[0, 0] = 0
    for i in range(1, n+1):
        dist[i, 0] = dist[i-1, 0] + poids[(i-1, 0), (i, 0)]
    for j in range(1, m+1):
        dist[0, j] = dist[0, j-1] + poids[(0, j-1), (0, j)]

    for i in range(1, n+1):
        for j in range(1, m+1):
            dist[i, j] = min(
                dist[i-1, j-1] + poids[(i-1, j-1), (i, j)],
                dist[i-1, j] + poids[(i-1, j), (i, j)],
                dist[i, j-1] + poids[(i, j-1), (i, j)]
            )

    return dist[n, m]
```

Approche dite **bottom-up**.

Remarque. La méthode précédente est centrale dans l'algorithme de calcul de la **distance d'édition de Levenshtein**, qui a par exemple pour applications :

- comparaison de deux fichiers texte ($\approx$ Unix : `diff`)
 - gestion de versions
 - détection de plagiat
- correction orthographique

La paternité de la méthode de programmation dynamique est attribuée à Richard Bellman (1920-1984) qui l'a conceptualisée vers 1952.

- Algorithme de Bellman-Ford pour les plus courts chemins dans un graphe.
- La relation de récurrence liant solution au problème principal et solutions aux sous-problèmes intermédiaires est parfois appelée *équation de Bellman* (revoir par exemple F_n et $\binom{n}{k}$).

- Problèmes du rendu de monnaie, du sac à dos
- Comparaison de chaînes de caractères
 - plus longue sous-suite commune
 - distance d'édition (de Levenshtein)
- Plus courts chemins dans un graphe
 - Bellman-Ford ($\approx$ Dijkstra mais avec poids de signe quelconque)
 - Floyd-Warshall ($\approx$ Bellman-Ford mais plus efficace dans un graphe dense)

- Calcul de haut en bas (*top-down*)
Typiquement récursif, avec mémoire passée en argument. On part du problème principal et on descend.
- Calcul de bas en haut (*bottom-up*)
Typiquement itératif. On part des sous-problèmes les plus élémentaires et on remonte.

Version avec fonction interne :

```
def f(objet):  
    memoire = {...}  
    def f_rec(objet intermédiaire):  
        if objet intermédiaire not in memoire:  
            calculer f_rec sur les sous-objets nécessaires  
            combiner les valeurs pour obtenir le résultat  
            stocker le résultat dans memoire  
        return memoire[objet intermédiaire]  
    return f_rec(objet)
```

Version sans fonction interne mais avec argument optionnel :

```
def f(objet, memoire={...}):  
    if objet not in memoire:  
        calculer f sur les sous-objets nécessaires  
        combiner les valeurs pour obtenir le résultat  
        stocker le résultat dans memoire  
    return memoire[objet]
```

On aborde les objets intermédiaires par ordre croissant de complexité. Cela suppose de fixer un ordre de parcours pertinent sur les sous-problèmes.

```
def f(objet):  
    memoire = {}  
    for o in sous_objets_par_ordre_croissant(objet):  
        calculer f(o) grâce aux valeurs précédentes  
        stocker le résultat dans memoire  
    return memoire[objet]
```

- En top-down, on ne résout que les sous-problèmes nécessaires ; en bottom-up, on résout tous les sous-problèmes, mais on ne risque en contrepartie pas de saturer la pile de récursion (limitée à 1000 en Python).
- En bottom-up, on doit concevoir un ordre de parcours sur les sous-problèmes. C'est parfois facile, parfois moins.

- **Dictionnaires** : clés plus générales que des entiers ou des uplets, ne prendront que la taille mémoire nécessaire.

Exemple : coefficients binomiaux, distance de Levenshtein.

- **Tableaux** : nécessitent en principe de savoir à l'avance de quel espace on a besoin en mémoire.

Exemple : F_n .

Dictionnaires en Python

Un dictionnaire est :

- une collection non ordonnée d'objets : les **valeurs**,
- chacune étant associée à une **clé**, les clés n'ayant pas de type particulier (ex. : couples de couples).

Exemple :

```
>>> d = {'a': 'lettre', 2: 'nombre', (1, 2): 'couple'}  
>>> type(d)  
<class 'dict'>
```

- On veut pouvoir **insérer, modifier ou supprimer** des valeurs arbitraires.

```
>>> d[2] = ...
```

```
>>> del d[2]
```

- On veut pouvoir **tester l'appartenance** d'une clé dans le dictionnaire. (Autrement dit, savoir s'il y a ou non une valeur associée.)

```
>>> ... in d
```

- On veut pouvoir **compter les clés** d'un dictionnaire.
- On veut pouvoir effectuer une **copie indépendante** d'un dictionnaire.

```
>>> len(d)
```

```
>>> e = d.copy()
```

```
>>> # modifier e, observer d
```

- Enfin, toutes ces opérations doivent pouvoir être faites **en temps constant** (en $O(1)$).

Listes ou tableaux :

- accès et modification par position, donc clés doublement pas de type arbitraire ;
- insertion, suppression en $O(n)$ et pas en $O(1)$.

Remarques : techniquement, l'insertion dans un dictionnaire a une complexité au pire de $O(n)$. Mais dans la plupart des cas, la complexité sera en fait $O(1)$.

En Python, on dispose de méthodes permettant de convertir un dictionnaire en un *itérable* sur lequel on pourra faire tourner une boucle.

- Méthode **keys** : écrire

```
for k in d.keys()
```

permet de parcourir les clés de d.

- Méthode **items** : écrire

```
for (k, v) in d.items()
```

permet de parcourir les couples (clé, valeur) de d.

On ne connaît a priori pas l'ordre dans lequel Python va effectuer ces parcours.

On donne un dictionnaire prérempli `d`. Les clefs sont des noms de villes et les valeurs le nombre d'habitants dans chaque ville.

Écrire une fonction qui prend en paramètre le dictionnaire `d` et qui renvoie le nom de la ville ayant le plus d'habitants.

```
def max_habitant(d: dict) -> str:
    nom_min = ""
    nb_habitants = 0
    for (ville,nb) in d.items():
        if nb >= nb_habitants:
            nom_min = ville
            nb_habitants = nb
    return nom_min
```

Fonctionnement interne par tables de hachage

Un cas d'utilisation des fonctions de hachage : mots de passe et authentification en ligne.

Un cas d'utilisation des fonctions de hachage : mots de passe et authentification en ligne.

Fonction de hachage : fonction

$$h : K \rightarrow E$$

avec

- K : espace des clés (très grand)
- E : espace des empreintes (pas trop grand)
- h « presque injective » (en quel sens ?)

- Protocoles notables : MD5, puis SHA-1, et tous les SHA-*
- En Python :
 - fonction native hash (mais plusieurs inconvénients)
 - `from hashlib import md5, sha1, blake2s`

```
def h(k):  
    return blake2s(k, digest_size=8).hexdigest()
```

```
h(b"Ceci est une clé secrète")
```

Idée :

- on choisit une fonction de hachage

$$h : K \rightarrow \llbracket 0, m - 1 \rrbracket ;$$

- on initialise un tableau T de taille m ;
- la valeur associée à la clé k sera stockée dans le tableau T dans la case numéro $h(k)$.

Ainsi toutes les opérations se font essentiellement en temps constant.

Les collisions :

- sont par construction difficiles à générer volontairement (ce serait une faille de sécurité de la fonction de hachage) ;
- mais se produisent avec une probabilité très élevée (paradoxe des anniversaires).

On peut par exemple s'en sortir en stockant dans la case numéro $h(k)$ la liste de tous les couples (k, v) .

Il n'existe pas de fonctions de hachage toute faite dans Python pour tous les objets. En pratique, un critère simple est que les objets soient de tailles fixes.

Donc on ne peut pas utiliser de liste comme clefs :

```
>>> d = {[0]: 0, [1, 2, 3]: 2, [0, 1]: None}
```