X - TP 3 Programmation dynamique

1 Exemple introductif : la suite de Fibonacci

"Tout le monde" connaît la suite de Fibonacci : il s'agit d'une suite de nombres entiers qui commence par 0, 1, et dans laquelle tout nombre est la somme des deux nombres précédents. Plus formellement, on définit la suite de Fibonacci $(F_n)_{n\geq 0}$ par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geq 0 \end{cases}$$

Code

```
def fibo(n):
    """ int -> int
    Renvoie le terme d'indice n de la suite de Fibonacci """
    pass
```

Question 1.

1. Écrire une fonction récursive `fibo` qui prend en entrée un entier `n` supposé positif, et qui renvoie le nombre F_n correspondant.

2. On note c_n le nombre d'opérations élémentaires (addition, soustraction, multiplication, division) effectuées lorsque l'on exécute l'instruction `fibo(n)`.

a. Déterminer c_0, c_1.

b. Pour tout $n \in \mathbb{N}$, exprimer c_n en fonction de c_{n-1} et de c_{n-2}.

c. À la calculatrice, représenter graphiquement $v_n = \ln(c_n)$, pour $0 \leq n \leq 100$. Que constate-t-on ?

d. Quelle est la complexité de la fonction `fibo` ?

3. Dresser l'arbre d'appel de l'instruction `fibo(4)`. Quels calculs sont réalisés "en trop" ?

1.1 Approche top-down et mémoïsation

La technique dite de mémoïsation permet d'éviter de répéter des calculs déjà effectués plusieurs fois. On décrit le fonctionnement d'une fonction récursive `fibomem` qui utilise pour cela une variable supplémentaire `memo` de type dictionnaire, qui va jouer le rôle de "mémoire" partagée entre les appels récursifs :

- les clés de `memo` correspondront aux valeurs de `n` pour lesquelles on a déjà calculé F_n ;
- la valeur associée à `n` dans `memo` est F_n.

À chaque appel récursif `fibomem(n, memo)` :

- si `n` est une clé de `memo` : cela veut dire que le calcul a déjà été effectué. Il suffit donc de renvoyer `memo[n]`.
- sinon : on effectue le calcul de F_n en mémoïsant les résultats intermédiaires. On met à jour `memo` avec la valeur de F_n et on renvoie F_n.

Remarque. L'approche "top-down" veut dire que l'on part du problème du calcul de F_n (le plus complexe) que l'on résout en se ramenant à des problèmes plus simples "vers le bas" (F_{n-1}, F_{n-2}).

Code

```
def fibomem(n, memo):
    """ int, {int:int} -> int
    Calcule le terme d'indice n de la suite de Fibonacci
    memo est un dictionnaire où les calculs intermédaires ont été stockés
    """
    if n in memo:
        return ...
    if n <= 1:
        memo[n] = n
        return memo[n]
    else:
        inter1 = ...
        inter2 = ...
        memo[n] = ...
        return ...
```

Code

```
print(fibomem(50, dict()))
```

Résultat

```
12586269025
```

1.2 Approche bottom-up et programmation dynamique

L'approche dite "par programmation dynamique" repose elle-aussi sur l'utilisation d'une structure de donnée auxiliaire afin de mémoriser les résultats intermédaires. Pour calculer le terme d'indice `n` de la suite de Fibonacci, on utilise un tableau `T` de taille `n + 1` définit de la manière suivante : on stocke dans `T[n]` le terme d'indice `n` de la suite de Fibonacci, si celui-ci est connu.

On remarque que l'on connaît immédiatement les valeurs à stocker dans `T[0]`, et `T[1]`. Puis, on raisonne de proche en proche : on peut ensuite calculer `T[2]`, etc. à l'aide d'une boucle. Pour finir, on renvoie `T[n]` (qui correspond à F_n, par définition !).

Remarque. L'approche "bottom-up" veut dire que l'on part du problème le plus simple (calcul de F_0, F_1) pour trouver les solutions aux problèmes plus complexes "vers le haut" (F_2, F_3,..., F_n).

Code

```
def fibodyn(n):
    """ int -> int
    Calcule le terme d'indice n de la suite de Fibonacci """
    pass
    T = [None]*(n + 1)
    T[0], T[1] = ..., ...
    for i in range(..., ...):
        T[i] = ...
    return ...
```

Code

```
print(fibodyn(50))
```

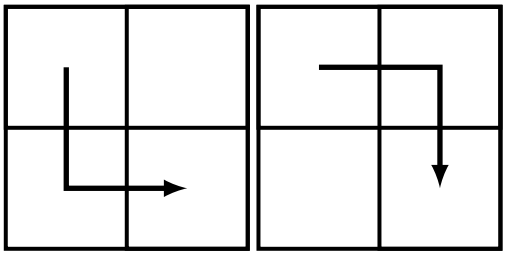
Résultat

```
12586269025
```

2 Nombre de chemins

On se demande de combien de manière possible on peut se rendre de la case supérieure gauche d'une grille $n \times m$ à la case inférieure droite. On n'autorise que des déplacements d'une case, vers la droite ou vers le bas.

Par exemple, si la grille est de taille 2×2, il y a deux chemins possibles.
Si la grille est de taille 1×1, on considère qu'il n'y a qu'un seul chemin possible.



Question 2. Donner le nombre de chemins possibles lorsque la grille est de taille :

1. 3×1 **2.** 3×2 **3.** 3×3 **4.** 3×4 **5.** 4×4.

2.1 Algorithme récursif

Écrire une fonction récursive `nombre_chemins` qui étant donné deux entiers `n` et `m` renvoie le nombre de chemins possibles entre la case supérieure gauche et la case inférieure droite d'une grille constituée de `n` lignes et de `m` colonnes.

On remarquera que :

- il n'y a qu'une seule solution lorsque `n` ou `m` vaut `1` ;
- pour qu'un chemin atteigne la case d'indice `(n - 1, m - 1)` dans la grille, il est nécessaire qu'il atteigne soit la case d'indice `(n - 2, m - 1)` (au dessus) ou `(n - 1, m - 2)` (à gauche).

Code

```
def nombre_chemins(n, m):
    """ int, int -> int
    Renvoie le nombre de chemins de (haut, gauche) à (bas, droite) dans
    ↪ une grille n×m """
    pass
```

Code

```
print(nombre_chemins(3, 4))
```

Résultat

```
10
```

Question 3. **1.** Mémoïser la fonction `nombre_chemins` afin de pouvoir déterminer le nombre de chemins reliant la case supérieure gauche à la case inférieure droite d'une grille 20×20.

2. **(Optionnel)** Écrire une fonction `liste_chemins` qui étant donné deux entiers `n` et `m` renvoie la liste de tous les chemins reliant la case supérieure gauche à la case inférieure droite d'une grille de taille `(n, m)`. On indiquera avec `"D"` (resp. `"B"`) un déplacement vers la droite (resp. vers le bas). Lorsque `n` et `m` sont tous deux égaux à `1`, on renverra `[[]]` (un seul chemin, aucun déplacement).

Code

```
print(liste_chemins(3, 3))
```

Résultat

```
[['D', 'D', 'B', 'B'], ['D', 'B', 'D', 'B'], ['B', 'D', 'D', 'B'],
↪ ['D', 'B', 'B', 'D'], ['B', 'D', 'B', 'D'], ['B', 'B', 'D',
↪ 'D']]
```

2.2 Programmation dynamique

On peut également résoudre ce problème de manière itérative en résolvant les problèmes "les plus simples d'abord" :

1. **Définir les sous problèmes.** Pour i, j deux entiers tels que $0 \leq i < n$ et $0 \leq j < m$ on définit $T(i,j)$ comme le nombre le nombre de chemins de la case d'indice $(0,0)$ jusqu'à la case d'indice (i,j).
2. **Identifier la relation de récurrence entre les sous-problèmes.**

$$T(i,j) = \begin{cases} 1 & \text{si } i = 0 \\ 1 & \text{si } j = 0 \\ T(i-1,j) + T(i,j-1) & \text{sinon} \end{cases}$$

3. **En déduire un algorithme qui résolve les sous-problèmes.** On initialise un tableau de taille (n,m), que l'on parcourt en remplissant successivement les cases à l'aide de la relation de récurrence.
4. **Résoudre le problème général.** On renvoit $T(n-1, m-1)$.

Code

```python
def nombre_chemins_dyn(n, m):
    """ int, int -> int
    Renvoie le nombre de chemins de (haut, gauche) à (bas, droite) dans une grille n×m """
    T = [ [None for j in range(m)] for i in range(n)]
    for i in range(n):
        T[...][...] = 1
    for j in range(m):
        ...
    for i in range(1, n):
        for j in range(1, m):
            T[i][j] = ...
    return ...
```

Code

```python
print(nombre_chemins_dyn(3, 4))
```

Résultat

```
10
```

3 Rendu de monnaie

Un système monétaire est la donnée d'une liste de "pièces" de différentes valeurs. Par exemple, le système monétaire utilisé en France en 2024 est `[0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100, 200, 500]`. On se demande, étant donné un `systeme` monétaire et un `montant` à rendre, combien de pièces au minimum doit-on utiliser pour rendre le `montant` dans en utilisant uniquement les pièces du `systeme`. Pour des raisons de simplicité on ne s'intéresse qu'aux montants entiers (positifs) et aux systèmes de monnaie dont les pièces sont des valeurs entières et dont la plus petite valeur est 1 : il est toujours possible de se ramener à ce cas.

Par exemple, pour rendre `13€` dans notre système monétaire, on peut utiliser 5 pièces en rendant 5€ + 5€ + 1€ + 1€ + 1€, ou utiliser 3 pièces en rendant 10€ + 2€ + 1€, cette dernière solution étant optimale.

Question 4. On considère le système monétaire `[1, 4, 8, 10]`.
Déterminer la solution au problème du rendu de monnaie pour les montants :

1. 12 **2.** 14 **3.** 21 **4.** 22 **5.** 36 **6.** 47

3.1 Algorithme glouton

Un algorithme glouton est un algorithme qui suit le principe de faire à chaque étape un choix optimum dans l'espoir d'aboutir à la solution optimale (ça ne marche pas toujours). Par exemple dans le cas du rendu de monnaie, l'algorithme glouton consiste à rendre systématiquement la pièce ayant la plus grande valeur tant que cela est possible :

- à chaque étape on cherche la `plus_grande` pièce dans le `systeme` monétaire qui soit inférieure au montant à rendre ;
- lorsque la `plus_grande` pièce est trouvée, on la rend : il reste alors `montant - plus_grande` à rendre.

Code

```
def rendu_monnaie_glouton(systeme, montant):
    """ [int], int -> int
    Renvoie le nombre de pièces minimales à utiliser pour rendre
    le montant avec les pièces de systeme """
    nb_pieces = 0
    while montant > 0:
        # On cherche la plus grande pièce à rendre
        plus_grande = systeme[0]
        for p in systeme:
            if ...:
                ...
        # on rend la pièce trouvée
        montant = ...
        nb_pieces = ...
    return nb_pieces
```

Code

```
stm = [1, 4, 8, 10]
for m in [12, 14, 21, 22, 36, 47]:
    print(rendu_monnaie_glouton(stm, m))
```

Résultat

```
3
2
3
4
6
8
```

3.2 Algorithme récursif

Il est possible de résoudre ce problème à l'aide d'un algorithme récursif. L'idée est la suivante :

- si le `montant` à rendre est nul, alors on n'utilise aucune pièce.
- sinon, le `montant` à rendre est strictement positif. Pour chaque pièce `p` du `systeme`, on calcule récursivement le nombre minimal de pièce nécessaires pour rendre `montant - p`. Attention, à cette étape il est nécessaire de s'assurer que `montant - p` est bien un nombre positif ou nul. Enfin, on termine en renvoyant le plus petit nombre de cette liste, plus un.

Code

```
def rendu_monnaie_rec(systeme, montant):
    """ [int], int -> int
    montant >= 0
    Renvoie le nombre de pièces minimales à utiliser pour rendre
le montant avec les pièces de systeme """
    if ...:
        ...
    else:
        # on détermine de manière récursive le nombre minimum
        # de pièce à rendre pour rendre montant - p, ou p est
        # une des pièces du systeme.
        liste = []
        for p in systeme:
            ... # à compléter
        # On renvoie le minimum de la liste, plus 1.
        ...
```

Code

```
stm = [1, 4, 8, 10]
for m in [12, 14, 21, 22, 36]:
    print(rendu_monnaie_rec(stm, m))
```

Résultat

```
2
2
3
3
4
```

Question 5. **1.** **a.** Comparer les résultats obtenus à l'aide de la fonction `rendue_monnaie_rec` avec ceux de `rendu_monnaie_glouton`.

Que constate-t-on ?

b. Que se passe-t-il lorsque l'on exécute l'instruction `rendu_monnaie_rec(stm, 47)` ?

c. Donner les avantages et les inconvénients de l'algorithme glouton du rendu de monnaie, ainsi que de l'algorithme récursif de rendu de monnaie.

2. **a.** Mémoïser la fonction `rendu_monnaie_rec` : on ajoutera un argument supplémentaire `memo` de type dictionnaire définit de la manière suivante :

- les clés de `memo` sont les `montant` pour lesquels on a déjà résolu de problème du rendu de monnaie ;
- la valeur associée à la clé `k` dans `memo` est le nombre de pièce minimal à utiliser pour rendre `k` dans le `systeme`.

On testera dès le début de la fonction `rendu_monnaie_rec` si `montant` se trouve dans `memo`. Si cela est le cas on renverra immédiatement le nombre de pièces minimal à utiliser pour rendre `montant`. Sinon, on applique le même algorithme que précédement, en prenant soin de mettre à jour `memo` avant de renvoyer le nombre de pièce minimal à utiliser pour rendre `montant`.

b. En déduire le nombre de pièce minimal à utiliser pour rendre le montant `47` dans le système `[1, 4, 8, 10]`.

3.3 Programmation dynamique

On peut utiliser la programmation dynamique pour résoudre le problème du rendu de monnaie : on cherche à trouver le nombre minimum de pièces qui doivent être choisies dans un système monétaire afin de rendre un montant n.

1. **Définir les sous-problèmes.** Il s'agit de trouver le nombre $T(i)$ qui étant donné un système monétaire $S \subset \mathbb{N}^*$ renvoie le minimum de pièces que l'on peut utiliser pour rendre la somme $i \in \mathbb{N}$. On note que comme $1 \in S$ par hypothèse, on rend dans le pire des cas la somme i avec i pièces de 1.

2. **Identifier une relation de récurrence entre les sous-problèmes.**

$$T(i) = \begin{cases} 0 & \text{si } i = 0 \\ 1 + \min(\{T(i-p), \text{ avec } p \in S \text{ et } i - p \geq 0\}) & \text{sinon.} \end{cases}$$

   Explication. Lorsque le montant à rendre i est nul, alors on ne rend aucune pièce de monnaie. Sinon, pour toutes les valeurs p des pièces du système qui sont plus petites que i, on doit calculer $T(i-p)$ correspondant au nombre de pièces nécessaires au minimum pour rendre le montant $i - p$.

   $T(i)$ vaut alors la plus petite de ces valeurs, à laquelle on ajoute 1 (on rend la pièce p).

3. **En déduire un algorithme qui résolve les sous-problèmes.** On initialise un tableau `T` de taille `montant + 1`. Initialement, la case d'indice `i` du tableau `T` contient la valeur `i` (pire rendu de monnaie possible). Puis, pour chaque montant `i` compris entre `0` et `montant + 1` (exclu), et pour chaque pièce `p` du `systeme` :

   - si il est plus avantageux de rendre `i + p` avec `T[i] + 1` pièces de monnaie que la valeur actuelle de `T[i + p]`, alors on met à jour `T[i + p]`.
   - sinon on ne fait rien

4. **Résoudre le problème général.** On renvoit $T(n)$.

Code

```
def rendu_monnaie_dyn(systeme, montant):
    """ [int], int -> int
    Renvoie le nombre minimum de pièces de systeme à utiliser pour rendre
    montant """
    # dans le pire des cas on rend i avec i pièces de 1
    T = [... for i in range(montant + 1)]
    for i in range(montant + 1):
        for p in systeme:
            if i + p <= montant and ...:
                ...
    return ...
```

Code

```
systeme = [1, 4, 8, 12]
for m in [1, 4, 8, 12, 14, 21, 22, 36]:
    print(rendu_monnaie_dyn(systeme, m), end=' ')
print(rendu_monnaie_dyn(systeme, 47))
```

Résultat

```
1 1 1 1 3 3 4 3 7
```

Question 6. Dans toutes les questions ci-dessous, $c \in \mathbb{N}^*$ désigne le nombre de pièces de `systeme` et $n \in \mathbb{N}$ désigne le `montant` à rendre.

1. **a.** Compléter et tester le code de la fonction `rendu_monnaie_dyn`.

   b. Quelle est la complexité en temps de `rendu_monnaie_dyn` en fonction de c et de n ?

   c. Quelle est la complexité en mémoire de `rendu_monnaie_dyn` en fonction de c et de n ?

2. On souhaite dans cette question obtenir également la liste des pièces utilisées pour rendre `montant` à l'aide de `systeme`. Pour cela, on stocke dans `T[i]` la **liste** des pièces utilisées pour rendre `i` à l'aide de `systeme` en minimisant le nombre de pièces utilisées.

   a. Initialement, quelle liste doit-on stocker dans `T[i]` ?

   b. Pour chaque montant `i` compris entre `0` et `montant + 1` (exclu) et pour chaque pièce `p` du `systeme` :

   i. À quelle condition portant sur les listes `T[i + p]` et `T[i]` est-il plus avantageux de rendre la monnaie à l'aide de la solution stockée en `T[i]` ?

   ii. Dans le cas où il est plus avantageux d'utiliser la solution stockée en `T[i]`, comment doit-on mettre à jour `T[i + p]` ?

   c. En déduire le code de la fonction `rendu_monnaie` qui détermine la liste des pièces à utiliser pour rendre `montant` à l'aide des pièces de `systeme` en minimisant le nombre de pièces utilisées.

   Attention. Lorque l'on écrira le code concernant la mise à jour de `T[i + p]` on prendra soin **de ne pas modifier** la liste stockée en `T[i]`.

3. **a.** Quelle est la complexité en temps dans le pire des cas de la fonction `rendu_monnaie` ?

   b. Proposer un algorithme ayant une complexité en temps dans le pire des cas de l'ordre de cn permettant d'obtenir la listes des pièces à utiliser pour rendre `montant` dans `systeme` en utilisant le minimum de pièces possibles.

Code

```
def rendu_monnaie(systeme, montant):
    """ [int], int -> [int]
    Trouve le meilleur rendu de monnaie dans systeme """
    pass
```

Code

```
systeme = [1, 4, 8, 12]
for m in [1, 4, 8, 12, 14, 21, 22, 36]:
    print(rendu_monnaie(systeme, m))
print(rendu_monnaie(systeme, 47))
```

Résultat

```
[1]
[4]
[8]
[12]
[1, 1, 12]
[1, 8, 12]
[1, 1, 8, 12]
[12, 12, 12]
[1, 1, 1, 8, 12, 12, 12]
```