Programmation dynamique
TP découverte

I. Exemple introductif

La fameuse suite de Fibonacci est définie par $F_0 = 0, F_1 = 1$ et $\forall n \geqslant 2, F_n = F_{n-1} + F_{n-2}$
On souhaite calculer F_n en Python.

A. Première approche.

On peut calculer récursivement le terme F_n comme suit

```python
def fib(n : int) -> int:
    if n == 0:
        res = 0
    elif n == 1:
        res = 1
    else :
        # appel récursif
        res = fib(n-2) + fib(n-1)

    return res
```

a. Recopier cette fonction et la tester pour $n = 20$ puis 30.

b. Retester avec $n = 35$ puis 40.

Là ça devrait ralentir !

> N'essayez pas plus loin ! La réponse pour $n = 42$ prend déjà une dizaine de minutes et pour $n = 45$ il faut compter une heure...

c. Pour cerner le problème, insérer une ligne `print(n)` juste après l'appel récursif (la ligne `res =`) et tester ce qui se passe pour $n = 10$.

> En fait cette fonction calcule plusieurs fois les mêmes termes de la suite, ce qui est bien sûr une grande perte de temps.
> La complexité est en fait exponentielle !

Dans la suite on va considérer deux approches pour éviter ce problème.

B. Solution récursive avec mémoïsation (dite *top-down*)

Une solution consiste à stocker les valeurs déjà calculées. On parle de **mémoïsation**.
(il n'y a pas de faute de frappe. Le créateur du terme voulait faire la différence avec la simple "mémorisation" de valeurs.)

On va pour cela définir un dictionnaire globalement avant la fonction qui va servir à stocker les valeurs déjà calculées.

> On parle souvent de « mémoire cache » ; on appellera donc le dictionnaire `cache`. Par exemple `fib(10)` qui vaut 55 correspondra à `cache[10] = 55`

> PRINCIPE DE LA MÉMOÏSATION
> On crée tout d'abord un dictionnaire vide.
> Ensuite dans la fonction,
> - si n n'est **pas** une des clés du dictionnaire (fib(n) non connnu) on calcule, en utilisant la fonction récursive, la valeur correspondante qu'on stocke dans `cache[n]`
> - **après** cette étape, on retourne la valeur `cache[n]`

a. Recopier et compléter les pointillés de la fonction suivante en utilisant le principe donné précédemment.

```
1  cache = {} # dictionnaire vide
2
3  def fib_mem(n: int) -> int:
4      if n not in ........ :
5          # on complète cache[n]
6          if n == 0:
7              cache[n] = ...
8          elif n ==1:
9              ..............
10         else :
11             # appel récursif
12             cache[n] = ....................
13
14     # Dans tous les cas, cache[n] est connu à ce stade
15     return cache[n]
```

b. Afin de voir l'évolution du dictionnaire `cache`, ajouter une ligne **print**`(cache)` en ligne 13 juste après l'appel récursif .

Testez avec `fib_mem(10)` *(pas beaucoup plus, sinon c'est le chantier à l'écran...)*

c. Enlever la ligne du print (sinon l'affichage est vite illisible).

Tester la fonction avec $n = 30, 40$ puis 45,50 ou davantage.

(les réponses devraient être cette-fois ci "instantanées")

C. Solution itérative (dite *bottom-up*)

En regardant l'évolution du dictionnaire `cache`, vous pouvez constater que les valeurs sont calculées « dans l'ordre ».

Cette remarque mène à l'évolution suivante

On peut directement calculer les valeurs de `fib(k)` pour k de plus en plus grand.
On stoppe lorsque $k = n$.

> En effet connaître `fib(0)` et `fib(1)` donne `fib(2)`.
> Puis `fib(1)` et `fib(2)` permettent de calculer `fib(3)`.
> Et ainsi de suite.

PRINCIPE DE LA MÉTHODE ITÉRATIVE BOTTOM-UP

Dans la fonction,

- On crée une liste de taille suffisante pour stocker toutes les valeurs qu'on va calculer (on peut la remplir avec des `None`)
- On initialise avec les premières valeurs
- On complète les suivantes avec une boucle
- En fin de boucle, on retourne la valeur souhaitée du tableau.

a. Recopier et compléter les pointillés dans la fonction suivante en appliquant le principe précédent.

```
def fib_bottom_up(n : int) -> int:
    # initialisation de la liste
    fib = [None] * (...)

    # remplissage des premières valeurs
    fib[0] = ...
    ............
```

```
# boucle principale pour calculer les valeurs suivantes
for k in range(...):
    fib[k] = ...........

# fin de boucle : on retourne la valeur voulue de la liste
return ......
```

b. Tester la fonction avec quelques valeurs de n. *(vérifiez que vous avez les mêmes résultats qu'avant...)*

Dans la boucle, insérer une ligne **print** (`fib`) pour voir l'évolution du tableau (avec $n = 10$ par ex).

II. Exemple plus poussé

En pratique, cette technique de programmation s'utilise souvent avec des exemples « en deux dimensions ». (notre suite de valeurs dépend de "deux" indices, ce qui aura des conséquences sur les dictionnaires et tableaux à utiliser).

Un exemple est le problème de la recherche d'une sous-séquence commune à deux chaîne de caractères :

- On se donne deux chaînes de caractères x et y fixées.
- Une sous-séquence commune est une chaîne de caractères communs à x et y *(les caractères doivent être dans l'ordre mais pas obligatoirement consécutifs)*.
- On cherche la plus longue possible de ces sous-séquences.

> Par exemple avec les chaînes `x = "ratatouille"` et `y = "rapido"`, une sous-chaîne commune possible est `"rao"` (même si le 'o' est séparé des autres lettres).

A. Les formules

Pour les formules on considère plus généralement des "tranches" de x et y : les i premiers caractères de x et les j premiers de y. *(les tranches sont donc de la forme $x_0 \dots x_{i-1}$ et $y_0 \dots y_{j-1}$)*

On note $L(i,j)$ la longueur d'une plus longue sous-séquence commune à ces deux tranches

On ADMET, pour l'instant, les relations suivantes :

Valeurs initiales. Si $i = 0$ ou si $j = 0$ alors $L(i,j) = 0$.

Récurrence. Pour $i \geqslant 1$ et $j \geqslant 1$, $L(i,j) = \begin{cases} 1 + L(i-1, j-1) & \text{si } x_{i-1} = y_{j-1} \\ \max\Big(L(i-1,j)\,;\,L(i,j-1)\Big) & \text{sinon} \end{cases}$

B. Programmation avec mémoïsation

En utilisant le principe de mémoïsation vu sur l'exemple de Fibonacci, recopier et compléter la fonction ci-dessous qui calcule $L(i,j)$.

Quelques remarques :

- Vu qu'il y a deux indices, les clés du dictionnaire cache seront des couples (i,j). La valeur correspondante est donc `cache[(i,j)]`
- Vu la relation de récurrence, l'étape d'appel récursif comportera un test.

```
# vous pouvez changer les mots pour tester
x = "ratatouille"
y = "rapido"
```

```
cache = {}
def lplssc_mem(i: int, j: int) -> int:
    """ Longueur d'une plus longue sous—séquence commune aux tranches x[0:i] et y[0:j]

    On suppose i <= len(x) et j <= len(y)
    """

    if (i, j) not in ........ :
        if ......... :
            cache[(i,j)] = 0
        else :
            # appel r écursif
            # plusieurs lignes à écrire


    return .....
```

C. Programmation bottom-up

Recommencer cette fois-ci avec la méthode "bottom-up".
Quelques remarques encore :

- Au lieu d'un simple tableau, vous aurez une "matrice" modélisée par une liste de listes.

  Vous rappelez-vous comment on accède à l'élément d'indices (i, j) ?
- Pour le "remplissage des premières valeurs", vous aurez déjà deux petites boucles à écrire
- Pour les "valeurs suivantes", vous aurez deux boucles imbriquées à gérer

```
def lplssc_bottom_up(x:str, y:str) -> int:
    """ Détermine la longueur d'une plus longue sous — chaine commune aux mots x et y
    """

    n = len(x)
    p = len(y)

    # initialisation de la matrice
    matrice = [ [None] * (p+1) for i in range(n+1)] # initialisation matrice
        remplie avec None

    # remplissage des premi ères valeurs
    for i in range(n+1) :
        matrice[i][0] = 0

    for ..... :
        ........

    # calcul des valeurs suivantes ( plusieurs lignes à écrire )


    # fin de boucle
    return ...
```

D. Pour aller plus loin

a. Justifier les relations admises à la section A..

b. Comment retrouver une plus longue sous-séquence commune à partir de la matrice des $L(i, j)$?

c. Ecrire une fonction `plssc` qui calcule une plus longue sous-séquence commune.