La récursivité

Exercices

Ces exercices sont à faire sur le NoteBook Jupyter que vous trouverez sur le serveur NSI.

Exercice 1

1. Écrire cette fonction `somme()` en Python, et la tester sur plusieurs exemples.

2. Écrire une nouvelle fonction `somme_carres()` calculant la somme des carrés des éléments d'une liste

```
### Quelques exemples pour tester les fonctions
assert somme([2, 12, 1, 8]) == ...
assert somme_carres([2]) == ...
assert somme_carres([2, 12, 1, 8]) == 213
```

Exercice 2 — *Implémentation récursive de la fonction factorielle*

Rappelons tout d'abord que $n! = 1 \times 2 \times 3 \times ... \times n$.
On montre alors facilement la relation de récurrence $n! = n \times (n-1)!$
Si l'on sait calculer $(n-1)!$ on connaîtra donc la valeur de $n!$
Or, toujours d'après la formule de récurrence, $(n-1)! = (n-1) \times (n-2)!$
On est donc ramené au calcul de $(n-2)!$ et ainsi de suite jusqu'à $1!$ dont on connaît la valeur : 1.

1. Écrire l'implémentation Python de cette fonction :

```
### Définition de la fonction
def factoriel(n : int):
    """
    Par définition n!=n*(n-1)! et 1!=1
    """
    # À compléter

### Quelques tests
assert factoriel(1) == 1
assert factoriel(5) == 120
```

2. Écrire sur une feuille la pile des appels récursifs de cette fonction pour `n = 4`

Exercice 3 — *Déterminer le minimum d'une liste d'entiers*

Supposons que nous ayons une fonction 'min(a,b)' qui renvoie le plus petit des entiers `a` et `b` et une liste $L = [a_0, a_1, ..., a_{n-1}]$ d'entiers dont il faut déterminer le minimum. **Version itérative**

- On initialise le minimum à mini $= a_0$
- On parcours la liste en appelant à chaque étape : $min(mini, a_i)$

Version récursive

- Le minimum de la liste L est le minimum entre a_0 et le minimum de la liste $L' = [a_1, a_2, ..., a_{n-1}]$ qui est lui même le minimum entre et le minimum de la liste et ainsi de suite...
- La condition d'arrêt étant : s'il n'y a qu'un seul élément dans la liste alors le minimum de la liste est cet élément.

1. Écrire ces deux fonctions :

```
### Définition de la fonction miniit - Version itérative
def miniit(L : list) -> int:
    # À compléter

### Définition de la fonction minrec - Version récursive
def minirec(L : list) -> int:
    # À compléter

### Quelques tests
assert miniit([2, 1, 3, 5]) == 1
assert minirec([2, 1, 3, 5]) == 1

from random import randint
L = [randint(1,100) for i in range(100)]
print(L, miniit(L), minirec(L))
```

2. Écrire sur une feuille la pile des appels récursifs la fonction `minirec`, pour la liste `[2, 1, 3, 5]`, sur le modèle de l'exemple donné pour la fonction `somme`.

Exercice 4 — *La suite de Fibonacci*

La suite de Fibonacci est la suite (u_n) définie sur $\mathbb{N}$ de la manière suivante :

$$\begin{cases} u_0 = 0 \text{ et } u_1 = 1 \\ u_n = u_{n-1} + u_{n-2} \text{ si } n \geqslant 2 \end{cases}$$

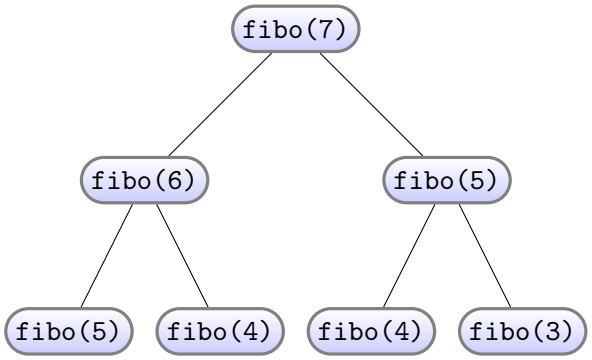
Ses premiers termes sont 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

1. Écrire l'implémentation récursive d'une fonction donnant le terme de rang n de la suite de Fibonacci :

```
### Définition de la fonction
def fibo_rec(n : int) ->int:
    """    Implémentation récursive de la suite de Fibonacci
    """
        # À compléter

### Quelques tests :
assert fibo_rec(0) == 0
assert fibo_rec(1) == 1
assert fibo_rec(9) == 34
```

Si l'on veut savoir quels sont les appels récursifs effectués lors de l'appel de cette fonction pour `n`=7, nous pouvons représenter ces appels sous forme d'un arbre, où chaque sommet correspondra à un appel et contiendra la valeur du paramètre à cet instant.



2. Le début de cet arbre est donné ci-dessus, le recopier puis le compléter pour obtenir tous les appels récursifs de cette fonction lorsque `n = 7`.

3. Écrire l'implémentation itérative d'une fonction donnant le terme de rang `n` de la suite de Fibonacci, puis observer la différence de comportement de ces deux fonctions lors de l'appel pour `n = 30`.

Remarque : Vous constaterez que le temps de calcul par la méthode récursive est extrêmement long, nous verrons comment éviter de faire autant de calcul inutiles car déjà réalisée lorsque nous étudierons la programmation dynamique.

Commencez par regarder la vidéo suivante :

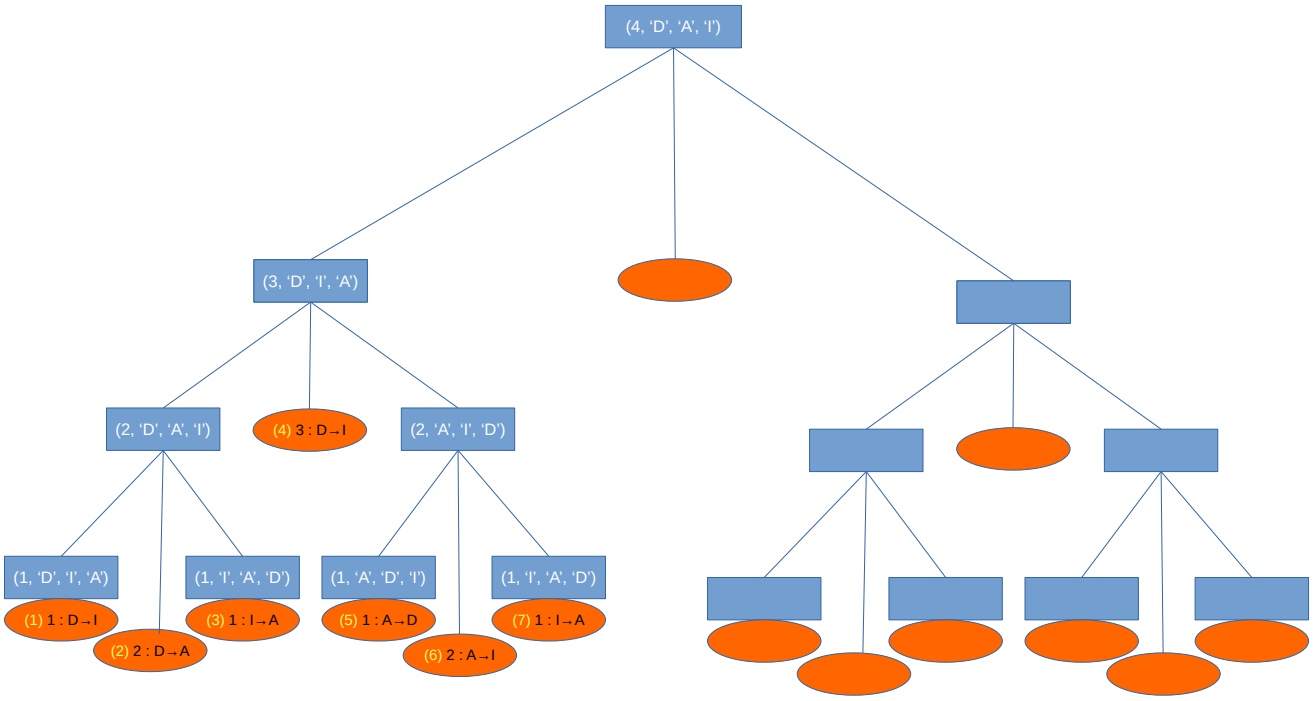
https://www.youtube.com/watch?v=U3nGNJTxYc4

Voici l'implémentation en Python de cet algorithme récursif pour résoudre le problème des tours de Hanoï.

1. Exécuter le programme suivant et observer le résultat obtenu :

```
def hanoi(n, X, Y, Z):
    """
    Cette fonction affiche les déplacements à effectuer
    pour résoudre le problème des tours de Hanoï comportant
    n cylindres, les variables X, Y et Z représentent le
    nom (chaine de caractères str)
    que l'on donne à chacun des piquets. X étant le
    piquet de départ, Y celui d'arrivé et Z le piquet intermédiaire
    """
    if n == 1:
        print(f"Déplacer le disque 1 de {X} à {Y}")
    else:
        hanoi(n-1, X, Z, Y)
        print(f"Déplacer le disque {n} de {X} à {Y}")
        hanoi(n-1, Z, Y, X)
```

2. Voici l'arbre, incomplet, décrivant l'exécution de cette fonction pour les paramètres `(4, 'D', 'A', 'I')`. Le compléter.



Les rectangles bleus contiennent les différentes valeurs des paramètres lors des appels récursifs, et les ellipses contiennent l'affichage dans la console ainsi que l'ordre d'affichage.