Programmation 2 Spring, 2026

TD 8: Recursion

1. Écrivez une fonction récursive `mypower` qui calcule a^b pour a et b donnés, où b est un entier positif.

   N'utilisez pas l'opérateur `**`. Vous n'avez pas le droit d'utiliser des boucles ou toute autre fonction.

Solution.

```
def mypower(a, b):
    if b == 1:
        return a
    return a*mypower(a,b-1)

print(mypower(2, 8))
```

2. Écrivez une fonction récursive `mymax` qui renvoie le maximum d'une liste.

   Vous n'avez pas le droit d'utiliser des boucles ou toute autre fonction.

Solution.

```
def mymax(A):
    if len(A) == 1:
        return A[0]
    else:
        m2 = mymax(A[1:])
        if A[0] > m2:
            return A[0]
        else:
            return m2

print(mymax([3, 7, 12, 346, 13, 74]))
```

3. Écrivez une fonction récursive `evenbeforeodd1` qui prend en paramètre une liste d'entiers `A` et retourne une nouvelle liste contenant les mêmes éléments que `A`, réorganisés de telle sorte que :

   - tous les nombres pairs apparaissent avant tous les nombres impairs,

   La fonction ne doit pas modifier la liste originale `A` et doit être implémentée de manière récursive.

   Vous n'avez pas le droit d'utiliser des boucles.

Solution.

```
def evenbeforeodd1(A):
    if len(A) == 1:
        return [ A[0] ]

    B = evenbeforeodd1(A[1:])

    if A[0]%2 == 0:
        B.insert(0, A[0])
    else: B.append(A[0])

    return B

print( evenbeforeodd1([4,1,7,5,9,3,1,2,4,2,7]) )
```

4. Écrivez une fonction récursive `evenbeforeodd2` qui prend en paramètre une liste d'entiers `A` et retourne deux listes `B` et `C` contenant respectivement les nombres pairs et impairs de `A`, réorganisés de telle sorte que :

   - l'ordre relatif des nombres pairs entre eux soit préservé dans `B`, et
   - l'ordre relatif des nombres impairs entre eux soit également préservé dans `C`.

   La fonction ne doit pas modifier la liste originale `A` et doit être implémentée de manière récursive.

   Vous n'avez pas le droit d'utiliser des boucles.

Solution.

```
def evenbeforeodd2(A):
    if len(A) == 1:
        if A[0]%2 == 0:
            return [ A[0] ], [ ]
        else:
            return [ ], [ A[0] ]

    B, C = evenbeforeodd2(A[1:])

    if A[0]%2 == 0:
        B.insert(0, A[0])
    else:
        C.insert(0, A[0])

    return B, C

print( evenbeforeodd2([4,1,7,5,9,3,1,2,4,2,7]) )
```

5. Écrivez une fonction Python récursive `reverse` qui prend une chaîne de caractères `s` et retourne son inverse.

   Vous n'avez pas le droit d'utiliser des boucles ou toute autre fonction.

   Exemple: Pour `s = "hello"`, il renvoie `"olleh"`.

Solution.

```
def reverse(s):
    if len(s) == 1: return s
    else: return reverse(s[1:])+s[0]

print(reverse("hello"))
```

6. Écrivez deux fonctions :

   - l'une appelée `findeven_loops` qui utilise l'itération, et
   - l'autre appelée `findeven_recursion` qui utilise la récursivité.

   Ces fonctions réalisent l'opération suivante :

   > L'entrée de la fonction est une liste d'entiers `A`. La fonction renvoie une (nouvelle) liste contenant uniquement les entiers pairs.

   Exemple: Pour `A = [3,1,5,4,4,2,1,22,7]`, il renvoie `[4, 4, 2, 22]`.

   Solution.

```python
def findeven_recursion(A):
    if len(A) == 0:
        return []

    B = findeven_recursion(A[1:])
    if A[0]%2 == 0:
        B.append(A[0])
    return B

print(findeven_recursion([3,1,5,4,4,2,1,22,7]))
```

```python
def findeven_loops(A):
    B = []

    for x in A:
        if x%2 == 0:
            B.append(x)
    return B

print(findeven_loops([3,1,5,4,4,2,1,22,7]))
```

7. Écrivez une fonction récursive `flattenAll` qui prend une liste `A` qui contient des entiers (qui pourrait être à l'intérieur d'autres listes contenues dans `A`), et renvoie une **liste aplatie** contenant tous les entiers de `A`.

   Notez que `A` peut contenir une liste de listes de listes, et ainsi de suite ...

   Exemple: Pour `A = [2, [ [ [3,4], [4] ], 1 ], [6] ]`, il renvoie `[2,3,4,4,1,6]`.

   Solution.

```
def flattenAll(A):
    if len(A) == 0: return []

    B = flattenAll(A[1:])

    if type(A[0]) != int:
        return flattenAll(A[0]) + B
    else:
        return [A[0]] + B


A = [2, [ [ [3,4], [4] ], 1 ], [6] ]
print(flattenAll(A))

A = [ [ [ [3,4], [4] ] ] ]
print(flattenAll(A))
```

8. Écrivez une fonction récursive `allSubsets` qui renvoie la liste de tous les sous-ensembles d'une liste d'entrée `A` (sans répéter aucun sous-ensemble).

   Rappelons que le nombre total de sous-ensembles de `A` est $2^{\texttt{len}(A)}$.

   Exemple: Pour `A = [2,5,8]`, il renvoie `[[], [2], [5], [8], [2,5], [2,8], [5,8], [2,5,8]]`.

Solution.

```
def allSubsets(A):
    if len(A) == 1:
        return [ [], [A[0]] ]
    B = allSubsets(A[1:])
    n = len(B)
    for i in range(0, n):
        B.append( B[i] + [A[0]] )
    return B

print(allSubsets([1,2,3]))
```

9. Écrivez une fonction récursive `allPermutations` qui renverra une liste de toutes les permutations d'une liste d'entrée `A`.

   Rappelons que le nombre total des permutations de `A` est `len(A)!`.

   Exemple: Pour `A = [2,5,8]`, il renvoie `[[2,5,8], [2,8,5], [5,2,8], [5,8,2], [8,2,5], [8,5,2]]`.

Solution 1.

```
def allPermutations(A):
    if len(A) == 1:
        return [ [A[0]] ]
    B = allPermutations(A[1:])

    C = []
    for i in range(0, len(B)):
        for j in range(0, len(B[i])+1):
            w = list(B[i])
            w.insert(j, A[0])
            C.append(w)
    return C

print(allPermutations([1,2,3,4]))
```

Solution 2.

```
def allPermutations(A):
    if len(A) == 1:
        return [ A ]

    B = []
    for i in range(0, len(A)):
        p = A[i]
        C = A[:i] + A[i+1:]
        for x in allPermutations(C):
            B.append( [p] + x )
    return B

print(allPermutations([1,2,3,4]))
```

10. (**TP question.**) On considère une piste circulaire comportant n stations-service disposées tout autour.

Chaque station i, où $i \in [0, n-1]$, dispose d'une certaine quantité d'essence, notée `gas[i]`.

Cette quantité peut varier d'une station à l'autre—certaines peuvent avoir très peu, voire pas d'essence du tout, tandis que d'autres peuvent en avoir un large excédent.

Soit `cost[i]` la quantité d'essence nécessaire pour se rendre de la station i à la station $(i+1) \bmod n$.

La quantité totale d'essence disponible, toutes stations confondues, est exactement suffisante pour effectuer un tour complet de la piste. Autrement dit,

$$\sum_{i=0}^{n-1} \texttt{gas}[i] = \sum_{i=0}^{n-1} \texttt{cost}[i].$$

Vous commencez votre trajet avec le réservoir vide. À chaque fois que vous atteignez une station, vous pouvez récupérer toute l'essence qui s'y trouve et l'ajouter à votre réservoir.

Votre objectif est de choisir une station de départ telle qu'en suivant la piste dans l'ordre, vous ne tombiez jamais en panne d'essence et puissiez ainsi effectuer un tour complet (c'est-à-dire revenir à votre point de départ).

Montrer qu'il existe toujours au moins une station de départ permettant d'effectuer le circuit complet sans jamais tomber en panne d'essence. Cette fois, faites une preuve par induction.

Donnez aussi l'algorithme récursif correspondant, qui suit la preuve.

Solution. Procédons par récurrence sur n.

Cas de base $n = 1$. S'il n'y a qu'une seule station, on a $\texttt{gas}[0] = \texttt{cost}[0]$. Donc c'est possible de faire un tour.

Étape d'induction. Supposons que la propriété est vraie pour tout circuit de $k < n$ stations. Montrons-la pour un circuit de n stations ($n \geq 2$).

Cas 1 : $\texttt{gas}[i] \geq \texttt{cost}[i]$ pour tout i. Alors en partant de n'importe quelle station, on ne consomme jamais plus que ce qu'on a à chaque étape. Le réservoir ne peut pas devenir négatif, donc on peut partir de **n'importe quelle station**.

Cas 2 : Il existe un indice j tel que $\texttt{gas}[j] < \texttt{cost}[j]$. Dans ce cas, la station j est un « puits » : on ne peut pas en repartir si on y arrive avec trop peu d'essence. On va « fusionner » la station j avec la précédente pour réduire le problème.

On supprime la station j du circuit, et on modifie la station $j-1$ (indice modulo n) pour tenir compte du fait qu'on doit maintenant aller directement de $j-1$ à $j+1$.

La nouvelle quantité d'essence, `gas_p`, à la station $j-1$ reste inchangée :

$$\texttt{gas\_p}[j-1] = \texttt{gas}[j-1].$$

Mais le nouveau coût, `cost_p`, pour aller de $j-1$ à $j+1$ devient :

$$\texttt{cost\_p}[j-1] = \texttt{cost}[j-1] + \Big(\texttt{cost}[j] - \texttt{gas}[j]\Big).$$

En effet :

- On doit d'abord aller de $j-1$ à j : coût $\texttt{cost}[j-1]$.

- Puis de j à $j+1$: coût $\texttt{cost}[j]$.
- Mais en passant par j, on aurait pris $\texttt{gas}[j]$ qu'on n'aura pas si on saute la station.

Vérifions la conservation de l'énergie totale. Avant modification :

$$S = \sum_{i=0}^{n-1} \texttt{gas}[i] - \sum_{i=0}^{n-1} \texttt{cost}[i] = 0.$$

Après modification, on a enlevé la station j et modifié le coût de $j-1$. Calculons la nouvelle somme des différences :

$$\begin{aligned} S' &= \left(\sum_{i\neq j} \texttt{gas\_p}[i]\right) - \left(\sum_{i\neq j} \texttt{cost\_p}[i]\right) \\ &= \left(\sum_{i\neq j} \texttt{gas}[i]\right) - \left(\sum_{\substack{i\neq j \\ i\neq j-1}} \texttt{cost}[i]\right) - \Big(\underbrace{\texttt{cost}[j-1] + \texttt{cost}[j] - \texttt{gas}[j]}_{\texttt{cost\_p}[j-1]}\Big) \\ &= \sum_{i=0}^{n-1} \texttt{gas}[i] - \sum_{i=0}^{n-1} \texttt{cost}[i] = 0. \end{aligned}$$

La condition initiale est donc conservée.

Maintenant, par hypothèse d'induction, il existe une station de départ s dans le circuit modifié (de taille $n-1$) qui permet d'effectuer le tour sans panne.

Dans ce circuit modifié, la station $j-1$ a été modifiée.

- Si $s \neq j-1$, alors dans le circuit original, en partant de la même station, on peut faire le tour en passant par j normalement.
- Si $s = j-1$ dans le circuit modifié, alors dans le circuit original, en partant de $j-1$, on peut aller jusqu'à j avec le réservoir suffisant car le nouveau coût $\texttt{cost}[j-1] + \texttt{cost}[j] - \texttt{gas}[j]$ correspond exactement à ce qu'on consomme pour aller de $j-1$ à $j+1$ en passant par j, et en passant par j, on récupère $\texttt{gas}[j]$ qui compense le manque.

Par principe de récurrence, la propriété est vraie pour tout $n \geq 1$.

```python
def find_valid_start(gas, cost):
    n = len(gas)

    if n == 1:
        return 0

    for j in range(0, n):
        if cost[j] > gas[j]:
            cost_p = cost[:j] + cost[j+1:]
            cost_p[j-1] = cost[j-1] + cost[j] - gas[j]

            gas_p = gas[:j] + gas[j+1:]

            sol = find_valid_start(gas_p, cost_p)
            if sol >= j: sol += 1
            return sol
    return 0
```

11. **(difficile)** Étant donnée une chaîne `s`, écrivez une fonction récursive `findPalindrome` qui renvoie le plus grand palindrome de `s`, où nous sommes autorisés à supprimer des caractères de `s`.

    Exemple: Pour `s = "abcdba"`, il renvoie `"abcba"` ou `"abdba"`.

    Solution.

```
def findPalindrome(s):
    if len(s) == 1 or len(s) == 0:
        return s

    if s[0] == s[-1]:
        return s[0] + findPalindrome(s[1:-1]) + s[-1]
    else:
        t1 = findPalindrome(s[:-1])
        t2 = findPalindrome(s[1:])
        if len(t1) > len(t2):
            return t1
        else: return t2

print(findPalindrome("abcdfadfasdfpajsasdfkjasasdfasdfba"))
```

12. **(difficile)** On vous donne une liste `A` de entiers non négatifs, pour un jeu suivante.

    Joueur 1 choisit un entier parmi les extrémités de la liste, suivi de Joueur 2, puis de Joueur 1, et ainsi de suite.

    Chaque fois qu'un joueur choisit un parmi, celui-ci ne sera plus disponible pour le joueur suivant et est supprimé de `A`.

    Le jeux continue jusqu'à ce que tous les entiers aient été choisis.

Le joueur avec le score maximum gagne.

Soit une liste d'entiers `A`, écrivez une fonction récursive `listGame` qui calcule la valeur du jeu pour Joueur 1. Autrement dit, elle renvoie le score de `Joueur 1` si `Joueur 1` joue parfaitement.

Solution.

Explications

La fonction `listGame(A, i, j)` calcule le score maximal que le joueur courant peut garantir pour `A[i..j]`.

Cas de Base. Si $i == j$: retourner $A[i]$ car il ne reste qu'un seul élément.

Cas Récursif.

(a) Calculer la somme totale du sous-tableau courant : mysum $= \sum_{k=i}^{j} A[k]$.

(b) Calculer récursivement le meilleur score de l'adversaire si :

- Le joueur prend l'élément gauche : gauche $=$ `listGame`$(A, i+1, j)$.
- Le joueur prend l'élément droit : droit $=$ `listGame`$(A, i, j-1)$.

(c) Le choix optimal du joueur courant : `max(mysum - gauche, mysum - droit)`.

```
def listGame(A, i = 0, j = None):
    if j == None: j = len(A)-1

    if i == j: return A[i]

    mysum = 0
    for k in range(i, j+1): mysum += A[k]

    left = listGame(A, i+1, j)
    right = listGame(A, i, j-1)

    return max (mysum-left,  mysum-right)


import random
A = [ random.randint(0,100) for i in range(10) ]
print(A)
print(listGame(A))
```