# Exercices de bac sur la récursivité

**Exercice 1 :** *Cet exercice est consacré à l'analyse et à l'écriture de programmes récursifs.*

1) a) Expliquer en quelques mots ce qu'est une fonction récursive.
**Solution :** Une fonction récursive est une fonction qui s'appelle elle-même dans son code.

b) On considère la fonction Python suivante :

```
def compte_rebours(n):
    """ n est un entier positif ou nul """
    if n >= 0:
        print(n)
        compte_rebours(n-1)
```

L'appel `compte_rebours(3)` affiche successivement les nombres 3, 2, 1 et 0.
Expliquer pourquoi le programme s'arrête après l'affichage du nombre 0.
**Solution :** Après l'affichage de 0, `n` vaut -1. La condition du `if` n'est pas vérifiée et l'exécution de la fonction se termine.

2) En mathématiques, la **factorielle** d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à $n$. Par convention, la factorielle de 0 est 1. Par exemple :

- la factoriel de 1 est 1
- la factoriel de 2 est $2 \times 1 = 2$
- la factoriel de 3 est $3 \times 2 \times 1 = 6$
- la factoriel de 4 est $4 \times 3 \times 2 \times 1 = 24$

Compléter le programme ci-dessous afin que la fonction récursive `fact` renvoie la factorielle de l'entier passé en paramètre de cette fonction.
Exemple : `fact(4)` renvoie 24.

```
def fact(n):
    """ Renvoie le produit des nombres entiers
        strictement positifs inférieurs à n """
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

3) La fonction `somme_entiers_rec` ci-dessous permet de calculer la somme des entiers, de 0 à l'entier naturel `n` passé en paramètre. Par exemple :

- Pour `n` = 0, la fonction renvoie la valeur 0.
- Pour `n` = 1, la fonction renvoie la valeur $0 + 1 = 1$.

...

- Pour `n` = 4, la fonction renvoie la valeur $0 + 1 + 2 + 3 + 4 = 10$.

```
1 def somme_entiers_rec(n):
2     """ Permet de calculer la somme des entiers,
3     de 0 à l'entier naturel n """
4     if n == 0:
5         return 0
```

```
6       else:
7           print(n)   # pour vérifier
8           return n + somme_entiers_rec(n-1)
```

L'instruction `print(n)` de la ligne 7 dans le code précédent a été insérée afin de mettre en évidence le mécanisme en œuvre au niveau des appels récursifs.

a) Écrire ce qui sera affiché dans la console après l'exécution de la ligne suivante :
`res = somme_entiers_rec(3)`
**Solution :** Il va s'afficher 3, 2 et 1.

b) Quelle valeur sera alors affectée à la variable `res` ?
**Solution :** La valeur renvoyée est $3+2+1+0=6$.

4) Écrire en Python une fonction `somme_entiers` non récursive : cette fonction devra prendre en argument un entier naturel `n` et renvoyer la somme des entiers de 0 à `n` compris. Elle devra donc renvoyer le même résultat que la fonction `somme_entiers_rec` définie à la question 3. Exemple : `somme_entiers(4)` renvoie 10.

```
def somme_entier(n):
    s = n
    for i in range(n):
        s = s + i
    return s
```

**Exercice 2 :** *Cet exercice porte sur les langages et la programmation (récursivité).*

```
def f(n):
    if n == 0:
        print("Partez !")
    else:
        print(n)
        f(n-1)
```

1) Voici une fonction codée en Python :

a) Qu'affiche la commande `f(5)` ?
**Solution :** On va obtenir 5, 4, 3, 2, 1 et `Partez !`.

b) Pourquoi dit-on de cette fonction qu'elle est récursive ?
**Solution :** La fonction s'appelle elle-même. Elle est donc récursive.

2) On rappelle qu'en Python l'opérateur + a le comportement suivant sur les chaines de caractères et les listes :

```
>>> S = 'a' + 'bc'
>>> S
'abc'
```

```
>>> L = ['a'] + ['b', 'c']
>>> L
['a', 'b', 'c']
```

On a besoin pour les questions suivantes de pouvoir ajouter une chaine de caractères `s` en préfixe à chaque chaine de caractères de la liste `chaines`.
On appellera cette fonction `ajouter`.
Par exemple, `ajouter("a", ["b", "c"])` doit renvoyer `["ab", "ac"]`.

a) Compléter le code suivant :

```
def ajouter(s, chaines):
    resultat = []
    for mot in chaines:
        resultat.append(s + mot)
    return resultat
```

b) Que renvoie la commande `ajouter("b", ["a", "b", "c"])` ?
**Solution :** `["ba", "bb", "bc"]`.

c) Que renvoie la commande `ajouter("a", [""])` ?
**Solution :** `["a"]`.

3) On s'intéresse ici à la fonction suivante écrite en Python où `s` est une chaine de caractères et `n` un entier naturel.

```
def produit(s, n):
    if n == 0:
        return [""]
    else:
        resultat = []
        for c in s:
            resultat = resultat + ajouter(c, produit(s, n - 1))
        return resultat
```

a) Que renvoie la commande `produit("ab", 0)` ? Le résultat est-il une liste vide ?
**Solution :** On obtient `[""]`. Ce n'est pas une liste vide, puisqu'elle contient le texte vide.

b) Que renvoie la commande `produit("ab", 1)` ?
**Solution :** `["a", "b"]`.

c) Que renvoie la commande `produit("ab", 2)` ?
**Solution :** `["aa", "ab", "ba", "bb"]`.

**Exercice 3 :** *Cet exercice porte sur l'algorithmique et la programmation.*
Un **palindrome** est un mot qui se lit de la même manière de la gauche vers la droite que de la droite vers la gauche. Par exemple, *kayak* est un palindrome.
On propose ci-dessous une fonction pour tester si un mot est un palindrome.
On précise que, pour une chaîne de caractères `chaine` :

- l'instruction `len(chaine)` renvoie sa longueur ;
- l'instruction `chaine[-1]` renvoie son dernier caractère ;
- l'instruction `chaine[1:-1]` renvoie la chaîne privée de son premier caractère et de son dernier caractère.

```
def tester_palindrome(chaine):
    if len(chaine) < 2:
        return True
    elif chaine[0] != chaine[-1]:
        return False
    else:
        chaine = chaine[1:-1]
        return tester_palindrome(chaine)
```

1) On saisit, dans la console, l'instruction suivante :

```
>>> tester_palindrome('kayak')
```

Combien de fois est appelée la fonction `tester_palindrome` lors de l'exécution de cette instruction ? On veillera à compter l'appel initial.
**Solution :** L'appel `tester_palindrome('kayak')` va provoquer l'appel `tester_palindrome('aya')` qui va provoquer l'appel `tester_palindrome('y')`. Il y aura donc 3 appels.

2) a) Justifier que la fonction `tester_palindrome` est récursive.
**Solution :** Elle est récursive parce qu'elle s'appelle elle-même.

b) Expliquer pourquoi l'appel à la fonction `tester_palindrome` se terminera quelle que soit la chaîne de caractères sur laquelle elle s'applique.
**Solution :** Chaque nouvel appel récursif se fait avec une chaîne de taille diminuée de 2. Dans le "pire" des cas, on arrivera à un appel récursif avec une chaîne vide ou de taille 1, ce qui arrête les appels récursifs.

3) La saisie, dans la console, de l'instruction `tester_palindrome(53235)` génère une erreur.
a) Parmi les quatre propositions suivantes, indiquer le type d'erreur affiché :
- `ZeroDivisionError`
- `ValueError`
- `TypeError` C'est parce que la fonction va vouloir utiliser `len(53235)`, ce qui provoque l'erreur puisque la fonction `len` ne peut pas s'appliquer sur un entier.
- `IndexError`

b) Proposer sur la copie une ou plusieurs instructions qu'on pourrait écrire en première ligne de la fonction `tester_palindrome` et permettant d'afficher clairement cette erreur à l'utilisateur.
**Solution :** On peut rajouter `assert type(chaine) == str, "Il faut donner un texte"`.

4) Écrire le code d'une fonction itérative (non récursive) `est_palindrome` qui prend en paramètre une chaîne de caractères et renvoie un booléen égal à `True` si la chaîne de caractères est un palindrome, `False` sinon.
**Solution :**

```
def est_palindrome(chaine):
    n = len(chaine)
    for i in range(n//2):
        if chaine[i] != chaine[n-i-1]:
            return False
    return True
```

**Exercice 4 :** *Cet exercice traite du thème « programmation », et principalement de la récursivité.*
On rappelle qu'une chaine de caractères peut être représentée en Python par un texte entre guillemets "" et que :
- la fonction `len` renvoie la longueur de la chaine de caractères passée en paramètre ;
- si une variable `ch` désigne une chaine de caractères, alors `ch[0]` renvoie son premier caractère, `ch[1]` le deuxième, etc ;
- l'opérateur + permet de concaténer deux chaines de caractères.

```
>>> texte = "bricot"
>>> len(texte)
6
>>> texte[0]
"b"
>>> texte[1]
"r"
>>> "a" + texte
"abricot"
```

On s'intéresse dans cet exercice à la construction de chaines de caractères suivant certaines règles de construction.
**Règle A :** Une chaine est construite suivant la règle A dans les deux cas suivants :
- soit elle est égale à `"a"` ;
- soit elle est de la forme `"a" + chaine + "a"`, où `chaine` est une chaine de caractères construite suivant la règle A.

**Règle B :** Une chaine est construite suivant la règle B dans les deux cas suivants :
- soit elle est de la forme `"b" + chaine + "b"`, où `chaine` est une chaine de caractères construite suivant la règle A ;

- soit elle est de la forme `"b" + chaine + "b"`, où `chaine` est une chaine de caractères construite suivant la règle B.

On a reproduit ci-dessous l'aide de la fonction `choice` du module `random`.

```
>>> from random import choice
>>> help(choice)
Help on method choice in module random:
choice(seq) method of random.Random instance
Choose a random element from a non-empty sequence.
```

La fonction `A` ci-dessous renvoie une chaine de caractères construite suivant la règle A, en choisissant aléatoirement entre les deux cas de figure de cette règle.

```
def A():
    if choice([True, False]):
        return "a"
    else:
        return "a" + A() + "a"
```

1) a) Cette fonction est-elle récursive ? Justifier.

**Solution :** La fonction `A` s'appelle elle-même, donc `A` est une fonction récursive.

b) La fonction `choice([True, False])` peut renvoyer `False` un très grand nombre de fois consécutives. Expliquer pourquoi ce cas de figure amènerait à une erreur d'exécution.

**Solution :** Si `choice([True, False])` renvoie `False` consécutivement un nombre de fois supérieur à la limite de profondeur de récursion autorisée (1000 par défaut avec Python), dans ce cas une erreur d'exécution se produit.

Dans la suite, on considère une deuxième version de la fonction `A`. À présent, la fonction prend en paramètre un entier `n` tel que,

- si la valeur de `n` est négative ou nulle, la fonction renvoie `"a"` ;
- si la valeur de `n` est strictement positive, elle renvoie une chaine de caractères construite suivant la règle A avec un `n` décrémenté de 1, en choisissant aléatoirement entre les deux cas de figure de cette règle.

```
def A(n):
    if n <= 0 or choice([True, False]):
        return "a"
    else:
        return "a" + A(n-1) + "a"
```

2) a) Recopier sur la copie et compléter aux emplacements des points de suspension `...` le code de cette nouvelle fonction `A`.

b) Justifier le fait qu'un appel de la forme `A(n)` avec `n` un nombre entier positif inférieur à 50, termine toujours.

**Solution :** Pour $n > 0$, l'appel à `A(n)` provoque ou bien un arrêt de la fonction, ou bien un appel récursif avec le paramètre `n - 1`.
Un appel à `A(50)` pourrait provoquer dans le pire des cas 50 appels récursifs pour arriver à `A(0)` qui termine, ou alors terminer avant !

On donne ci-après le code de la fonction récursive B qui prend en paramètre un entier `n` et qui renvoie une chaine de caractères construite suivant la règle B.

```
def B(n):
    if n <= 0 or choice([True, False]):
        return "b" + A(n - 1) + "b"
    else:
        return "b" + B(n - 1) + "b"
```

On admet que :

- les appels `A(-1)` et `A(0)` renvoient la chaine `"a"` ;
- l'appel `A(1)` renvoie la chaine `"a"` ou la chaine `"aaa"` ;
- l'appel `A(2)` renvoie la chaine `"a"`, la chaine `"aaa"` ou la chaine `"aaaaa"`.

3) Donner toutes les chaines possibles renvoyées par les appels `B(0)`, `B(1)` et `B(2)`.

**Solution :**

- `B(0)` renvoie `"bab"`
- `B(1)` renvoie `"bab"` ou `"bbabb"`.
- `B(2)` renvoie `"bab"`, `"baaab"`, `"bbabb"` ou `"bbbabbb"`.

On suppose maintenant qu'on dispose d'une fonction `raccourcir` qui prend comme paramètre une chaine de caractères de longueur supérieure ou égale à 2, et renvoie la chaine de caractères obtenue à partir de la chaine initiale en lui ôtant le premier et le dernier caractère. Par exemple :

```
>>> raccourcir("abricot")
"brico"
```

```
>>> raccourcir("ab")
""
```

4) a) Recopier sur la copie et compléter les points de suspension `...` du code de la fonction `regle_A` ci-dessous pour qu'elle renvoie **`True`** si la chaine passée en paramètre est construite suivant la règle A, et **`False`** sinon.

```
def regle_A(chaine):
    n = len(chaine)
    if n >= 2:
        return chaine[0] == "a" and chaine[n - 1] == "a"\
            and regle_A(raccoourcir(chaine))
    else:
        return chaine == "a"
```

b) Écrire le code d'une fonction `regle_B`, prenant en paramètre une chaine de caractères et renvoyant **`True`** si la chaine est construite suivant la règle B, et **`False`** sinon.

**Solution :**

```
n = len(chaine)
if n >= 2:
    return chaine[0] == "b" and chaine[n - 1] == "b" and (
        regle_A(raccourcir(chaine)) or regle_B(raccourcir(chaine))
    )
else:
    return False
```

**Exercice 5 :** On s'intéresse dans cet exercice à un algorithme de mélange des éléments d'une liste.

1) Pour la suite, il sera utile de disposer d'une fonction `echange` qui permet d'échanger dans une liste `lst` les éléments d'indice `i1` et `i2`.

Expliquer pourquoi le code Python ci-contre ne réalise pas cet échange et en proposer une modification.

```
def echange(lst, i1, i2):
    lst[i2] = lst[i1]
    lst[i1] = lst[i2]
```

**Solution :** Lors de la première affectation, la valeur de `lst[i2]` est remplacée par celle de `lst[i1]`. Les deux cases sont égales et la valeur de `lst[i2]` est perdue. La deuxième affectation ne change donc rien.
Pour faire l'échange, il faut utiliser une variable temporaire qui servira à stocker `lst[i2]`.

2) La documentation du module `random` de Python fournit les informations ci-dessous concernant la fonction `randint(a,b)` :
`Renvoie un entier aléatoire N tel que a <= N <= b.`
Parmi les valeurs ci-dessous, quelles sont celles qui peuvent être renvoyées par l'appel `randint(0, 10)` ?

`0` `1` `3.5` `9` `10` `11`

**Solution :** Les valeurs possibles sont `0`, `1`, `9` et `10`.

3) Le mélange de Fischer Yates est un algorithme permettant de permuter aléatoirement les éléments d'une liste. On donne ci-dessous une mise en œuvre récursive de cet algorithme en Python.

```
from random import randint

def melange(lst, ind):
    print(lst)
    if ind > 0:
        j = randint(0, ind)
        echange(lst, ind, j)
        melange(lst, ind-1)
```

a) Expliquer pourquoi la fonction `melange` se termine toujours.
**Solution :** La valeur de `ind` diminue de 1 à chaque appel et il n'y a pas d'appel récursif si `ind` est inférieure ou égale à 0. La suite d'appels va donc forcément s'arrêter.

b) Lors de l'appel de la fonction `melange`, la valeur du paramètre `ind` doit être égal au plus grand indice possible de la liste `lst`.

Pour une liste de longueur $n$, quel est le nombre d'appels récursifs de la fonction `melange` effectués, sans compter l'appel initial ?
**Solution :** Si la liste n'a qu'un seul élément, `ind` vaut 0 et il n'y a pas d'appel récursif. Si `ind` vaut 1, il y a un appel. Donc si la liste est de taille `n`, il y a `n`$-1$ appels récursifs.

c) On considère le script ci-dessous :

```
lst = [v for v in range(5)]
melange(lst, 4)
```

On suppose que les valeurs successivement renvoyées par la fonction `randint` sont `2`, `1`, `2` et `0`.

Les deux premiers affichages produits par l'instruction `print(lst)` de la fonction melange sont ci-contre

```
[0, 1, 2, 3, 4]
[0, 1, 4, 3, 2]
```

Donner les affichages suivants produits par la fonction `melange`.
**Solution :** On obtient :

```
[0, 1, 2, 3, 4]
[0, 1, 4, 3, 2]   # on echange en 4 et 2
[0, 3, 4, 1, 2]   # on echange en 3 et 1
[0, 3, 4, 1, 2]   # on echange en 2 et 2
[3, 0, 4, 1, 2]   # on echange en 1 et 0
```

d) Proposer une version itérative du mélange de Fischer Yates.

**Solution :** Version avec une boucle **while** :

```
def melange(lst):
    ind = len(lst)-1
    while ind > 0:
        j = randint(0, ind)
        echange(lst, ind, j)
        ind = ind - 1
```

Version avec une boucle et compte à rebours :

```
def melange(lst):
    for ind in range(len(lst)-1, 0, -1):
        j = randint(0, ind)
        echange(lst, ind, j)
```

Version avec une boucle en allant dans l'autre sens :

```
def melange(lst):
    for ind in range(1, len(lst)):
        j = randint(0, ind)
        echange(lst, ind, j)
```