Chapitre : Fonctions récursives – Exercices

Exercice 1 – Récursion comme remplacement d'une boucle

On souhaite demander à l'utilisateur un entier strictement positif.

1.a Version itérative (rappel)

```
n = int(input("Donner un entier positif : "))
while n <= 0:
    print("Erreur")
    n = int(input("Donner un entier positif : "))
```

Questions :

1. Développer la version récursive de ce code.
2. Quel est le cas de base ?
3. Pourquoi la fonction termine-t-elle ?
4. En quoi cette fonction remplace-t-elle la boucle while ?

Exercice 2 – Compter récursivement (récursion simple)

```
def compte(n):
    if n == 0:
        return 0
    return 1 + compte(n-1)
```

Questions :

1. Calculer compte(3)
2. Que calcule cette fonction ?
3. Écrire une version itérative
4. Identifier le cas de base et le cas récursif
5. Développer la version itérative de cette fonction

Exercice 3 – Somme des entiers (classique)

```
def somme(n):
    if n == 0:
        return 0
    return n + somme(n-1)
```

Questions :

1. Calculer somme(4)
2. Donner l'expression mathématique calculée
3. Pourquoi la fonction s'arrête-t-elle ?
4. Que se passe-t-il si n est négatif ?

Exercice 4 – Récursion sur une liste (simple et concrète)

```
def somme_liste(L):
    if L == []:
        return 0
    return L[0] + somme_liste(L[1:])
```

Questions :

1. Quel est le cas de base ?
2. Tracer l'exécution pour L = [1, 3, 5]
3. Pourquoi la taille de la liste diminue-t-elle à chaque appel ?
4. Donner une version avec une boucle for

Exercice 5 – Recherche d'un élément dans une liste (récursive)

```
def contient(x, L):
    if L == []:
        return False
    if L[0] == x:
        return True
    return contient(x, L[1:])
```

Questions :

1. Tester la fonction avec x = 3, L = [1, 2, 3, 4]
2. Identifier le cas de base
3. Pourquoi cette fonction termine-t-elle toujours ?
4. Comparer avec une version itérative

Récursion multiple (consolidation)

Exercice 6 – Fibonacci (rappel fondamental)

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Questions :

1. Calculer fib(4)
2. Combien y a-t-il d'appels récursifs ?
3. Pourquoi parle-t-on de récursion multiple ?
4. Quel est le problème principal de cette fonction ?

Exercice 7 – Nombre de façons de monter un escalier

Un escalier a n marches.
On peut monter 1 ou 2 marches à la fois.

```
def nb_facons(n):
  if n == 0:
    return 1
  if n < 0:
    return 0
  return nb_facons(n-1) + nb_facons(n-2)
```

Questions :

1. Calculer nb_facons(3)
2. Pourquoi y a-t-il deux appels récursifs ?
3. Faire le lien avec la suite de Fibonacci
4. Identifier les cas de base

Exercice 8 – Récursion multiple contrôlée

```
def f(n):
  if n == 0:
    return 1
  return f(n-1) + f(n-1)
```

Questions :

1. Calculer f(1), f(2), f(3)
2. Que vaut f(n) en fonction de n ?
3. Combien d'appels récursifs pour f(3) ?
4. Pourquoi cette fonction devient rapidement lente ?

Exercice 9 – Erreur classique (à corriger)

```
def puissance(n):
    if n == 0:
        return 1
    puissance(n-1)
```

Questions :

1. Quel est le problème dans cette fonction ?
2. La fonction termine-t-elle ?
3. Corriger la fonction pour calculer 2^n

Exercice 10 – Comparer récursif / itératif (objectif bac)

```
def factorielle_rec(n):
    if n == 0:
        return 1
    return n * factorielle_rec(n-1)
```

Questions :

1. Écrire la version itérative
2. Identifier le cas de base
3. Dans quel cas la récursion est-elle plus lisible ?
4. Dans quel cas la boucle est-elle préférable ?

Chapitre : Fonctions récursives – Corrigé

Exercice 1 – Récursion comme remplacement d'une boucle

1. Version récursive

```
def saisir_entier_positif():
  n = int(input("Donner un entier positif : "))
  if n > 0:
    return n
  else:
    print("Erreur")
    return saisir_entier_positif()
```

---

2. Cas de base

Le cas de base est atteint lorsque :

$n > 0$

Dans ce cas, la fonction **ne s'appelle plus elle-même** et retourne une valeur.

---

3. Pourquoi la fonction termine-t-elle ?

À chaque appel récursif :

- l'utilisateur est invité à entrer une nouvelle valeur,
- dès qu'une valeur strictement positive est saisie, la fonction s'arrête.

👉 Il existe donc toujours une situation qui permet d'atteindre le cas de base.

---

4. En quoi cette fonction remplace-t-elle la boucle while ?

- La boucle while répète le code tant que la condition n'est pas satisfaite.
- La fonction récursive répète le code en **s'appelant elle-même** tant que la condition n'est pas satisfaite.

👉 La récursion joue ici exactement le même rôle qu'une boucle.

Exercice 2 – Compter récursivement

```
def compte(n):
    if n == 0:
        return 0
    return 1 + compte(n-1)
```

---

1. Calcul de compte(3)

compte(3)

= 1 + compte(2)

= 1 + (1 + compte(1))

= 1 + (1 + (1 + compte(0)))

= 1 + 1 + 1 + 0

= 3

---

2. Que calcule cette fonction ?

La fonction calcule **le nombre d'entiers entre 1 et n**.

👉 Autrement dit :

$$compte(n) = n$$

---

3. Version itérative

```
def compte_iteratif(n):
    c = 0
    while n > 0:
        c = c + 1
        n = n - 1
    return c
```

---

4. Cas de base et cas récursif

- **Cas de base** : n == 0
- **Cas récursif** : 1 + compte(n-1)

Exercice 3 – Somme des entiers

```
def somme(n):
    if n == 0:
        return 0
    return n + somme(n-1)
```

---

1. Calcul de somme(4)

somme(4)

= 4 + somme(3)

= 4 + 3 + somme(2)

= 4 + 3 + 2 + somme(1)

= 4 + 3 + 2 + 1 + somme(0)

= 10

---

2. Expression mathématique calculée

$$1 + 2 + 3 + \cdots + n$$

---

3. Pourquoi la fonction s'arrête-t-elle ?

À chaque appel :

- n diminue de 1
- on finit par atteindre n = 0, qui est le cas de base

---

4. Que se passe-t-il si n est négatif ?

La fonction **ne termine pas**, car :

- n devient de plus en plus négatif
- le cas n == 0 n'est jamais atteint

👉 Il faut toujours s'assurer que les paramètres permettent d'atteindre le cas de base.

Exercice 4 – Récursion sur une liste

```
def somme_liste(L):
    if L == []:
        return 0
    return L[0] + somme_liste(L[1:])
```

---

1. Cas de base

Le cas de base est :

L == []

---

2. Tracé pour L = [1, 3, 5]

somme_liste([1,3,5])

= 1 + somme_liste([3,5])

= 1 + 3 + somme_liste([5])

= 1 + 3 + 5 + somme_liste([])

= 9

---

3. Pourquoi la taille de la liste diminue-t-elle ?

À chaque appel récursif, on utilise :

L[1:]

👉 On enlève le premier élément, donc la liste devient plus courte.

---

4. Version itérative

```
def somme_liste_iteratif(L):
    s = 0
    for x in L:
        s = s + x
    return s
```

Exercice 5 – Recherche d'un élément

```
def contient(x, L):
    if L == []:
        return False
    if L[0] == x:
        return True
    return contient(x, L[1:])
```

---

1. Test avec x = 3, L = [1,2,3,4]

Résultat : True

---

2. Cas de base

L == []

---

3. Pourquoi la fonction termine-t-elle toujours ?

- À chaque appel, la liste diminue
- On finit soit par trouver l'élément, soit par atteindre la liste vide

---

4. Version itérative

```
def contient_iteratif(x, L):
    for e in L:
        if e == x:
            return True
    return False
```

Exercice 6 – Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

---

1. Calcul de fib(4)

fib(4) = 3

---

2. Nombre d'appels récursifs

Il y a **beaucoup d'appels**, car les mêmes valeurs sont recalculées plusieurs fois.

---

3. Pourquoi récursion multiple ?

Parce que la fonction s'appelle **deux fois** dans le cas récursif.

---

4. Problème principal

La fonction est **très lente** pour les grandes valeurs de n.

Exercice 7 – Escalier

1. Calcul de nb_facons(3)

Résultat : **3 façons**

---

2. Deux appels récursifs

- 1 marche : nb_facons(n-1)
- 2 marches : nb_facons(n-2)

---

3. Lien avec Fibonacci

Même relation :

$$f(n) = f(n-1) + f(n-2)$$

---

4. Cas de base

n == 0

n < 0

Exercice 8 – Récursion multiple contrôlée

```
def f(n):
  if n == 0:
    return 1
  return f(n-1) + f(n-1)
```

---

1. Calculs

- f(1) = 2

- f(2) = 4
- f(3) = 8

---

2. Expression de f(n)

$$f(n) = 2^n$$

---

3. Appels pour f(3)

Il y a **7 appels récursifs**.

---

4. Pourquoi lente ?

Le nombre d'appels **double à chaque niveau**.

Exercice 9 – Erreur classique

```
def puissance(n):
  if n == 0:
    return 1
  puissance(n-1)
```

---

1. Problème

La fonction ne retourne rien dans le cas récursif.

---

2. Termine-t-elle ?

Oui, mais elle retourne None.

---

3. Correction

```
def puissance(n):
  if n == 0:
```

```
    return 1
  return 2 * puissance(n-1)
```

Exercice 10 – Factorielle

```
def factorielle_rec(n):
  if n == 0:
    return 1
  return n * factorielle_rec(n-1)
```

---

1. Version itérative

```
def factorielle_iter(n):
  f = 1
  for i in range(1, n+1):
    f = f * i
  return f
```

---

2. Cas de base

n == 0

---

3. Quand la récursion est plus lisible ?

Quand le problème se définit naturellement par rapport à un plus petit problème.

---

4. Quand la boucle est préférable ?

Quand on cherche l'efficacité et la simplicité d'exécution.