Récursivité

Langages et programmation

Exercice 1 *QCM*

Pour chaque question, une seule réponse est correcte.

1. Quelle est la condition nécessaire pour qu'une fonction récursive se termine ?

   A. Elle doit utiliser une boucle `while`

   B. Elle doit posséder au moins un cas de base qui ne fait pas d'appel récursif

   C. Elle doit prendre au moins deux paramètres

   D. Elle doit renvoyer une liste

2. Que produit l'appel `f(4)` avec la fonction suivante ?

```
def f(n):
    if n == 0:
        return 1
    return 2 * f(n - 1)
```

   A. 4

   B. 8

   C. 16

   D. 32

3. On considère la fonction `factorielle` du cours :

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

   Combien d'appels au total (y compris l'appel initial) `factorielle(5)` génère-t-il ?

   A. 5

   B. 6

   C. 10

   D. 25

4. Quelle est la complexité en temps de `fibonacci(n)` en version récursive naïve (sans mémoïsation) ?

   A. $O(n)$

   B. $O(n^2)$

C. $O(n \log n)$

D. $O(2^n)$

Corrigé de l'exercice 1.

1. **B.** Sans cas de base, la fonction s'appelle indéfiniment et provoque un `RecursionError` (dépassement de la taille de la pile d'appels). A est faux (la récursivité remplace justement les boucles). C et D ne sont pas des conditions nécessaires.
2. **C.** $f(4) = 2 \times f(3) = 2 \times 2 \times f(2) = 2^3 \times f(1) = 2^4 \times f(0) = 2^4 \times 1 = 16$. Cette fonction calcule 2^n. A confond $f(n)$ et n. B confond avec 2^3. D confond avec 2^5.
3. **B.** Les appels sont : `factorielle(5)`, `factorielle(4)`, `factorielle(3)`, `factorielle(2)`, `factorielle(1)` et `factorielle(0)`. Le cas de base est `n == 0` : on a donc six appels. A oublie l'appel `factorielle(0)`, qui est le cas de base. C et D surestiment le nombre d'appels : chaque appel, sauf le cas de base, n'en déclenche qu'un seul autre.
4. **D.** Chaque appel génère deux nouveaux appels, ce qui produit un arbre d'appels dont le nombre de nœuds croît exponentiellement. A est la complexité de la version mémoïsée. B et C sont des complexités polynomiales qui ne correspondent pas à la structure de l'arbre d'appels.

Exercice 2 *Identifier cas de base et appel récursif (exercice guidé)*

Pour chacune des fonctions suivantes, **sans exécuter le code**, identifiez le cas de base (quelle condition arrête la récursion ?) et l'appel récursif (comment le problème se réduit-il ?).

a)

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

b) On utilise le module `os` : `os.listdir(dossier)` renvoie la liste des noms contenus dans un dossier, `os.path.join(dossier, nom)` construit le chemin complet d'un élément, et `os.path.isdir(chemin)` teste si ce chemin est un dossier.

```
import os

def nb_fichiers(dossier):
    total = 0
    for element in os.listdir(dossier):
        chemin = os.path.join(dossier, element)
        if os.path.isdir(chemin):
            total += nb_fichiers(chemin)
        else:
            total += 1
    return total
```

c)

```
def pgcd(a, b):
    if b == 0:
        return a
```

```
    return pgcd(b, a % b)
```

Corrigé de l'exercice 2. **a)** Cas de base : `n == 0` (on ne fait rien, la récursion s'arrête). Appel récursif : `compte_a_rebours(n - 1)`, le problème se réduit car `n` diminue de 1 à chaque appel.

b) Cas de base implicite : un dossier qui ne contient aucun sous-dossier (la boucle ne fait aucun appel récursif, on ne fait que compter les fichiers). Appel récursif : `nb_fichiers(chemin)` quand l'élément est un sous-dossier. Le problème se réduit car on descend dans l'arborescence. C'est un exemple de récursivité appliquée au domaine de l'informatique (parcours de système de fichiers).

c) Cas de base : `b == 0`, on renvoie `a`. Appel récursif : `pgcd(b, a % b)`. Le reste `a % b` est strictement inférieur à `b`, donc `b` diminue à chaque appel. C'est l'algorithme d'Euclide.

Exercice 3 *Dérouler la pile d'appels*

Pour chaque fonction, déroulez l'exécution en montrant la pile d'appels (les appels empilés, le cas de base, puis le dépilage).

a) `somme([3, 5, 2])` :

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

b) `nb_voyelles("bonjour")` :

```
def nb_voyelles(mot):
    if mot == "":
        return 0
    if mot[0] in "aeiouy":
        return 1 + nb_voyelles(mot[1:])
    return nb_voyelles(mot[1:])
```

c) `nb_chiffres(4073)` :

```
def nb_chiffres(n):
    if n < 10:
        return 1
    return 1 + nb_chiffres(n // 10)
```

Corrigé de l'exercice 3. **a)**

```
somme([3, 5, 2])
  = 3 + somme([5, 2])
  = 3 + (5 + somme([2]))
  = 3 + (5 + (2 + somme([])))
  = 3 + (5 + (2 + 0))          ← cas de base (liste vide)
  = 3 + (5 + 2) = 3 + 7 = 10
```

b)

```
nb_voyelles("bonjour")
```

```
  "b" non → nb_voyelles("onjour")
  "o" oui → 1 + nb_voyelles("njour")
  "n" non → nb_voyelles("jour")
  "j" non → nb_voyelles("our")
  "o" oui → 1 + nb_voyelles("ur")
  "u" oui → 1 + nb_voyelles("r")
  "r" non → nb_voyelles("")
  "" → 0                          ← cas de base
Dépilage : 0 → 0 → 1+0=1 → 1+1=2 → 2 → 2 → 1+2=3 → 3
Résultat : 3 voyelles (o, o, u).
```

c)

```
nb_chiffres(4073)
  4073 >= 10 → 1 + nb_chiffres(407)
  407 >= 10  → 1 + nb_chiffres(40)
  40 >= 10   → 1 + nb_chiffres(4)
  4 < 10     → 1                ← cas de base
Dépilage : 1 → 1+1=2 → 1+2=3 → 1+3=4
Résultat : 4 chiffres.
```

Exercice 4 *Compléter des fonctions récursives (exercice guidé)*

Complétez les parties manquantes (notées `...`).

a) Somme des chiffres d'un entier :

```python
def somme_chiffres(n):
    if n < 10:        # cas de base
        return ...    # (1)
    return ... + somme_chiffres(...)  # (2) et (3)
```

b) Recherche d'un élément dans une liste :

```python
def contient(L, x):
    if len(L) == 0:
        return ...    # (4)
    if L[0] == x:
        return ...    # (5)
    return contient(...)  # (6)
```

c) Compter le nombre d'occurrences d'un caractère dans une chaîne :

```python
def compter(chaine, c):
    if chaine == "":
        return ...    # (7)
    if chaine[0] == c:
        return ... + compter(...)  # (8) et (9)
    return compter(...)            # (10)
```

Corrigé de l'exercice 4. **a)** (1) : `n` (un nombre à un chiffre est sa propre somme). (2) : `n % 10` (le dernier chiffre). (3) : `n // 10` (le nombre sans son dernier chiffre).

```python
def somme_chiffres(n):
    if n < 10:
        return n
    return n % 10 + somme_chiffres(n // 10)
```

Vérification : `somme_chiffres(347)` = 7 + 4 + 3 = 14.

b) (4) : `False`. (5) : `True`. (6) : `L[1:], x`.

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

c) (7) : `0`. (8) : `1`. (9) : `chaine[1:], c`. (10) : `chaine[1:], c`.

```python
def compter(chaine, c):
    if chaine == "":
        return 0
    if chaine[0] == c:
        return 1 + compter(chaine[1:], c)
    return compter(chaine[1:], c)
```

Vérification : `compter("ananas", "a")` = 1 + 1 + 1 = 3.

Exercice 5 *Fonctions récursives (domaines variés)*

a) Un biologiste modélise une population de bactéries : chaque bactérie se divise en deux toutes les heures. Écrire une fonction récursive `population(n)` qui renvoie le nombre de bactéries après `n` heures, en partant d'une seule bactérie.

b) Un informaticien souhaite calculer la taille totale (en octets) de tous les fichiers d'un dossier et de ses sous-dossiers. Écrire une fonction récursive `taille_dossier(chemin)` qui parcourt récursivement l'arborescence.

Indication : `os.listdir(chemin)` renvoie la liste des éléments du dossier ; `os.path.join(chemin, nom)` construit le chemin complet ; `os.path.isdir(c)` teste si c'est un dossier ; `os.path.getsize(c)` renvoie la taille d'un fichier.

c) Un géographe mesure la longueur d'une côte à différentes échelles. Écrire une fonction récursive `longueur_cote(points)` qui calcule la longueur totale d'une ligne brisée définie par une liste de points (x, y).

Corrigé de l'exercice 5. **a)**

```python
def population(n):
    if n == 0:
        return 1
    return 2 * population(n - 1)
```

Cas de base : au temps 0, une seule bactérie. Appel récursif : à chaque heure, la population double. `population(n)` renvoie 2^n.

b)

```python
import os

def taille_dossier(chemin):
    if not os.path.isdir(chemin):
        return os.path.getsize(chemin)
    total = 0
    for element in os.listdir(chemin):
        sous_chemin = os.path.join(chemin, element)
        total += taille_dossier(sous_chemin)
    return total
```

Cas de base : si le chemin est un fichier, on renvoie sa taille. Appel récursif : pour un dossier, on additionne les tailles de tous ses éléments (fichiers et sous-dossiers).

c)

```python
import math

def longueur_cote(points):
    if len(points) <= 1:
        return 0
    x1, y1 = points[0]
    x2, y2 = points[1]
    d = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
    return d + longueur_cote(points[1:])
```

Cas de base : une liste de zéro ou un point a une longueur nulle. Appel récursif : on ajoute la distance entre les deux premiers points et la longueur de la ligne brisée restante.

Exercice 6 *Récursivité sur les listes*

a) Écrire une fonction récursive `longueur(L)` qui renvoie la longueur d'une liste sans utiliser `len`.

b) Écrire une fonction récursive `maximum(L)` qui renvoie le plus grand élément d'une liste non vide, sans utiliser `max`.

c) Écrire une fonction récursive `miroir(L)` qui renvoie la liste inversée, sans utiliser `reverse` ni `[::-1]`.

d) (**Bonus**) Écrire une fonction récursive `aplatir(L)` qui transforme une liste imbriquée en liste plate. Exemple : `aplatir([1, [2, 3], [4, [5, 6]]])` → `[1, 2, 3, 4, 5, 6]`.

Corrigé de l'exercice 6. **a)**

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

Cas de base : une liste vide a une longueur 0. Appel récursif : on ajoute 1 et on calcule la longueur du reste.

b)

```python
def maximum(L):
    if len(L) == 1:
        return L[0]
    m = maximum(L[1:])
    if L[0] > m:
        return L[0]
    return m
```

Cas de base : une liste d'un élément, dont le maximum est cet élément. Appel récursif : on compare le premier élément au maximum du reste.

c)

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

Cas de base : le miroir d'une liste vide est une liste vide. Appel récursif : on inverse le reste et on ajoute le premier élément à la fin.

d) (Bonus)

```python
def aplatir(L):
    if L == []:
        return []
    if isinstance(L[0], list):
        return aplatir(L[0]) + aplatir(L[1:])
    return [L[0]] + aplatir(L[1:])
```

Si le premier élément est une liste, on l'aplatit récursivement avant de le concaténer avec le résultat de l'aplatissement du reste.

Exercice 7 *Mémoïsation*

a) Voici la version récursive naïve de Fibonacci :

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

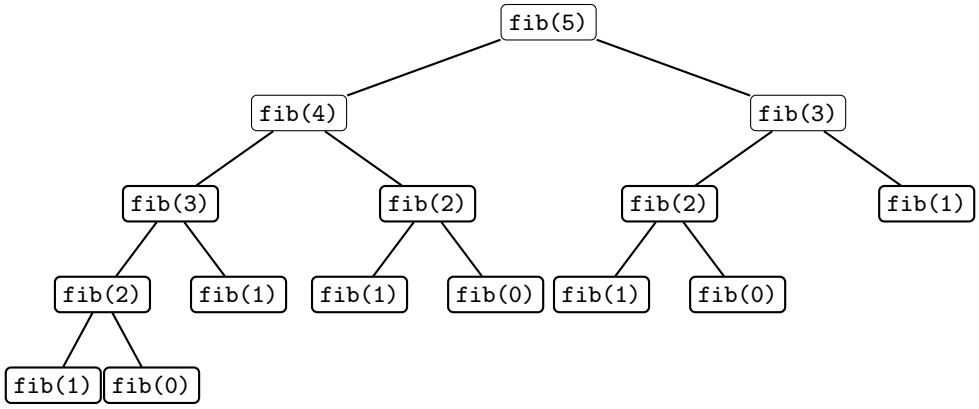
Dessinez l'arbre d'appels pour `fibonacci(5)`. Quelles valeurs sont calculées plusieurs fois ?

b) Expliquez pourquoi `fibonacci(35)` est très lent avec cette version.

c) Réécrivez la fonction en utilisant la **mémoïsation** : un dictionnaire `memo` qui stocke les résultats déjà calculés.

d) Quelle est la complexité de la version mémoïsée ?

Corrigé de l'exercice 7. **a)** Arbre d'appels de `fibonacci(5)` :



`fib(2)` est calculé trois fois, `fib(3)` est calculé deux fois, `fib(1)` est calculé cinq fois.

b) L'arbre d'appels de `fibonacci(35)` contient $2 \cdot F(36) - 1 = 29\,860\,703$ appels : les mêmes valeurs sont recalculées un nombre exponentiel de fois (la croissance est en φ^n, avec $\varphi \approx 1{,}618$). Par exemple, `fibonacci(2)` est calculé plus de 5 millions de fois.

c)

```python
def fibonacci_memo(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]
```

Avant de calculer, on vérifie si le résultat est déjà dans le dictionnaire. Sinon, on le calcule, on le stocke, puis on le renvoie.

d) La complexité est $O(n)$ en temps et $O(n)$ en mémoire. Chaque valeur de `fibonacci(k)` est calculée au plus une fois. On passe d'une complexité exponentielle $O(2^n)$ à une complexité linéaire.

Exercice 8 *Synthèse – les tours de Hanoï*

Le problème des tours de Hanoï consiste à déplacer une pile de n disques d'un piquet source vers un piquet destination, en utilisant un piquet auxiliaire. Deux règles : on ne déplace qu'un seul disque à la fois, et on ne pose jamais un disque sur un disque plus petit.

a) Résolvez le problème à la main pour $n = 2$ disques (piquets A, B, C). Combien de déplacements faut-il ?

b) Expliquez l'idée récursive en trois étapes : que fait-on pour déplacer n disques de A vers C en utilisant B ?

c) Écrire la fonction récursive `hanoi(n, source, destination, auxiliaire)` qui affiche les déplacements.

d) Combien de déplacements sont nécessaires pour n disques ? On note $T(n)$ ce nombre. Montrez que $T(n) = 2 \times T(n-1) + 1$, puis que $T(n) = 2^n - 1$.

e) Vérifiez en ajoutant un compteur à la fonction.

Corrigé de l'exercice 8. **a)** Pour deux disques :

1. déplacer le petit disque de A vers B ;
2. déplacer le grand disque de A vers C ;
3. déplacer le petit disque de B vers C.

Il faut trois déplacements.

b) L'idée récursive pour déplacer n disques de A vers C :

1. déplacer les $n-1$ disques du dessus de A vers B (en utilisant C comme auxiliaire) ;
2. déplacer le grand disque de A vers C ;
3. déplacer les $n-1$ disques de B vers C (en utilisant A comme auxiliaire).

Les étapes 1 et 3 sont des sous-problèmes de même nature avec $n-1$ disques. Le cas de base est $n = 1$: on déplace directement le disque.

c)

```python
def hanoi(n, source, destination, auxiliaire):
    if n == 1:
        print(f"Déplacer disque de {source} vers {destination}")
        return
    hanoi(n - 1, source, auxiliaire, destination)
    print(f"Déplacer disque de {source} vers {destination}")
    hanoi(n - 1, auxiliaire, destination, source)
```

d) $T(1) = 1$ (un seul déplacement). Pour $n > 1$: $T(n) = T(n-1) + 1 + T(n-1) = 2 \times T(n-1) + 1$.

Montrons que $T(n) = 2^n - 1$ par récurrence : $T(1) = 2^1 - 1 = 1$ (vrai). Supposons $T(k) = 2^k - 1$ pour un certain $k \geq 1$. Alors $T(k+1) = 2 \times (2^k - 1) + 1 = 2^{k+1} - 2 + 1 = 2^{k+1} - 1$.

e)

```python
def hanoi_compteur(n, source, destination, auxiliaire):
    if n == 1:
        return 1
    return (hanoi_compteur(n - 1, source, auxiliaire, destination)
            + 1
            + hanoi_compteur(n - 1, auxiliaire, destination, source))

for k in range(1, 8):
    print(f"n={k} : {hanoi_compteur(k, 'A', 'C', 'B')}"
          f" déplacements (2^{k}-1 = {2**k - 1})")
```

On vérifie bien $T(n) = 2^n - 1$ pour chaque valeur.