TP 3.2 – Récursivité et listes

Objectifs –

- ◇ Formuler récursivement un parcours de liste.
- ◇ Comparer des formulations récursive et itérative.
- ◇ Utiliser la division euclidienne dans un algorithme récursif.
- ◇ Reconnaître une décomposition récursive d'un problème.

Parcours principal

Exercice 1 – Tester la croissance d'une liste

Une liste d'entiers est croissante lorsque chacun de ses éléments, à partir du deuxième, est supérieur ou égal à l'élément qui le précède. Par convention, la liste vide et toute liste ne contenant qu'un élément sont croissantes.

Question 1. Écrire une fonction récursive `est_croissante_rec(lst: list[int]) -> bool`. On pourra comparer les deux premiers éléments, puis effectuer un appel récursif sur `lst[1:]`.

Question 2. Valider la fonction par plusieurs tests, notamment sur les cas limites.

Question 3. Écrire une fonction itérative `est_croissante_iter(lst: list[int]) -> bool` ayant la même spécification, en utilisant une boucle `while`.

Exercice 2 – Calculer le maximum d'une liste

Dans cet exercice, la liste passée en argument est une liste non vide d'entiers.

Question 1. Écrire une fonction itérative `maximum_iter(lst: list[int]) -> int` qui renvoie le plus grand élément de `lst`.

Question 2. Pour une liste contenant au moins deux éléments, expliquer comment obtenir son maximum à partir de son premier élément et du maximum de `lst[1:]`.

Question 3. En déduire une fonction récursive `maximum_rec(lst: list[int]) -> int` ayant la même spécification.

Question 4. Tester les deux fonctions sur plusieurs listes bien choisies.

Exercice 3 – Écrire un entier en base b

Soient n un entier naturel et b un entier compris entre 2 et 10. On souhaite représenter l'écriture de n en base b par une liste de chiffres, du chiffre de poids fort au chiffre des unités.
Par exemple, l'écriture de 19 en base 2 est représentée par `[1, 0, 0, 1, 1]`, car :

$$1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1 = 19$$

Le chiffre des unités est le reste `n % b`. Le quotient `n // b` correspond au nombre obtenu en supprimant ce dernier chiffre.

Question 1. Dérouler les appels récursifs de `en_base(2, 19)`, puis indiquer les listes construites lors des retours successifs.

Question 2. Écrire une fonction récursive `en_base(b: int, n: int) -> list[int]`. On conviendra que l'écriture de 0 est représentée par `[0]`.

Question 3. Tester abondamment la fonction, notamment avec des valeurs inférieures à la base et avec différentes bases.

Si le temps le permet

Exercice 4 – Version itérative de l'écriture en base b

Écrire une fonction itérative `en_base_iter(b: int, n: int) -> list[int]` ayant la même spécification que la fonction `en_base`. On pourra construire les chiffres en partant des unités, puis renverser la liste obtenue avec la méthode `reverse()`.

Exercice 5 – Recherche séquentielle

La recherche séquentielle d'une valeur v dans une liste consiste à parcourir les éléments jusqu'à trouver une occurrence de v. Si la fin de la liste est atteinte, c'est que la valeur n'est pas présente.

Question 1. Écrire une fonction itérative `recherche_sequentielle(v: int, lst: list[int]) -> bool` qui indique si la valeur recherchée est présente dans la liste

Question 2. Écrire une fonction itérative `recherche_sequentielle2(v: int, lst: list[int]) -> int` qui renvoie cette fois un indice de la liste correspondant à la valeur recherchée si la valeur est effectivement présente, et -1 si la valeur n'est pas présente.

Question 3. Écrire la version récursive de la fonction précédente `recherche_sequentielle_rec(v: int, lst: list[int]) -> int`. On pourra utiliser `lst[1:]`.

Question 4. Tester les deux fonctions sur les mêmes exemples, notamment avec une liste vide.

Travail à la maison

Exercice 6 – Tours de Hanoï

On dispose de trois tiges et de n disques de diamètres différents, placés sur une tige du plus grand au plus petit. Il faut déplacer la pile vers une autre tige en respectant les règles suivantes :

- ◇ on ne déplace qu'un disque à la fois ;
- ◇ on ne pose jamais un disque sur un disque plus petit.

Les tiges sont numérotées de 1 à 3. Un déplacement de la tige i vers la tige j sera affiché sous la forme `"i --> j"`.

Question 1. Pour déplacer $n > 1$ disques de la tige de départ vers la tige d'arrivée, expliquer comment décomposer le problème en trois étapes faisant intervenir la tige intermédiaire.

Question 2. Écrire une fonction récursive `hanoi(n: int, depart: int, intermediaire: int, arrivee: int) -> None` qui affiche les déplacements nécessaires. On supposera $n \geqslant 1$ et les trois numéros de tiges distincts.

Question 3. Donner les déplacements affichés par `hanoi(3, 1, 2, 3)`.

Question 4. On note $T(n)$ le nombre de déplacements effectués avec n disques. Donner une relation de récurrence vérifiée par T, puis exprimer $T(n)$ en fonction de n.

Question 5. Quelle est la profondeur maximale des appels récursifs ? Dans la pratique, quelle difficulté apparaît bien avant un éventuel dépassement de la limite de récursion ?

Approfondissement facultatif

Exercice 7 – Recherche dichotomique

La recherche dichotomique d'un entier v s'applique à une liste d'entiers triée dans l'ordre croissant. On compare v à l'élément situé au milieu de la fenêtre de recherche. Selon le résultat, on poursuit dans la moitié gauche ou dans la moitié droite.

Question 1. On note `g` et `d` les bornes gauche incluse et droite exclue de la fenêtre de recherche. Dérouler la recherche de 8 dans :

$$[-5, 1, 2, 3, 4, 7, 8, 10]$$

Question 2. Écrire une fonction itérative `recherche_dicho(v: int, lst: list[int]) -> bool` utilisant les indices `g` et `d`, sans extraire de sous-liste.

Question 3. Écrire une version récursive `recherche_dicho_rec(v: int, lst: list[int]) -> bool`. Pour cette question, l'utilisation de tranches est autorisée.

Question 4. Comparer le nombre maximal d'étapes d'une recherche séquentielle et d'une recherche dichotomique dans une liste de taille n.

Exercice 8 – Sous-listes et permutations

Une *sous-liste* de `lst` est obtenue en supprimant un nombre quelconque d'éléments tout en conservant l'ordre des éléments restants. Par exemple, `[]`, `[1, 3]` et `[1, 2, 3]` sont des sous-listes de `[1, 2, 3]`.

Question 1. Écrire une fonction récursive `sous_listes(lst: list) -> list[list]` qui renvoie toutes les sous-listes de `lst`. On pourra d'abord construire les sous-listes ne contenant pas le dernier élément, puis celles qui le contiennent.

Question 2. Une permutation de `lst` est une liste obtenue en modifiant l'ordre de ses éléments. Écrire une fonction récursive `permutations(lst: list) -> list[list]` qui renvoie toutes les permutations de `lst`. On pourra insérer le premier élément à toutes les positions dans chacune des permutations des éléments restants.

Exercice 9 – Évaluation d'un polynôme par Horner

Un polynôme :

$$P(x) = a_n x^n + \cdots + a_1 x + a_0$$

est représenté par la liste de ses coefficients rangés par degrés décroissants : `[a_n, ..., a_1, a_0]`. Par exemple, $2x^2 + x - 1$ est représenté par `[2, 1, -1]`.
La méthode de Horner utilise l'écriture suivante :

$$P(x) = (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots)x + a_0$$

Question 1. Écrire une fonction récursive `eval_horner_rec(p: list[float], x: float) -> float`. La liste `p` est supposée non vide.

Question 2. Combien de multiplications sont effectuées pour un polynôme de degré n ?

Question 3. Écrire une version itérative `eval_horner_iter`.

Exercice 10 – Courbe de Koch avec `turtle`

À la profondeur 0, la courbe de Koch est un segment de longueur l. Pour passer à la profondeur suivante, chaque segment est remplacé par quatre segments de longueur $l/3$, séparés par des rotations successives de $60°$, $-120°$ et $60°$.
On pourra utiliser les instructions suivantes du module `turtle` :

```
forward(longueur)
left(angle)
right(angle)
```

Écrire une fonction récursive `koch(longueur: float, profondeur: int) -> None` qui trace cette courbe.

Corrigés des exercices

Corrigé – Exercice 1 – Tester la croissance d'une liste

Question 1. Si la liste contient au moins deux éléments, elle est croissante si ses deux premiers éléments sont dans le bon ordre et si la liste privée de son premier élément est elle-même croissante :

```
def est_croissante_rec(lst: list[int]) -> bool:
    """Indique si lst est croissante."""
    if len(lst) <= 1:
        return True
    return (lst[0] <= lst[1]
            and est_croissante_rec(lst[1:]))
```

Grâce au fonctionnement de **and**, l'appel récursif n'est pas effectué si `lst[0] <= lst[1]` est faux.

Question 2.

```
assert est_croissante_rec([])
assert est_croissante_rec([3])
assert est_croissante_rec([1, 2, 2, 5])
assert not est_croissante_rec([1, 4, 3, 5])
assert not est_croissante_rec([2, 1])
```

Question 3.

```
def est_croissante_iter(lst: list[int]) -> bool:
    """Indique si lst est croissante."""
    i = 1
    while i < len(lst):
        if lst[i] < lst[i - 1]:
            return False
        i = i + 1
    return True
```

Corrigé – Exercice 2 – Calculer le maximum d'une liste

Question 1.

```
def maximum_iter(lst: list[int]) -> int:
    """Renvoie le maximum de la liste non vide lst."""
    assert len(lst) > 0
    maximum = lst[0]
    for valeur in lst[1:]:
        if valeur > maximum:
            maximum = valeur
    return maximum
```

Question 2. Le maximum de `lst` est le plus grand des deux nombres suivants : son premier élément et le maximum de la liste privée de cet élément. Le cas de base est celui d'une liste ne contenant qu'un élément.

Question 3.

```
def maximum_rec(lst: list[int]) -> int:
    """Renvoie le maximum de la liste non vide lst."""
    assert len(lst) > 0
    if len(lst) == 1:
        return lst[0]
    maximum_reste = maximum_rec(lst[1:])
    if lst[0] > maximum_reste:
        return lst[0]
    return maximum_reste
```

Question 4.

```
tests = [[4], [1, 5, 2, 5], [-8, -3, -12]]
for lst in tests:
    assert maximum_rec(lst) == maximum_iter(lst)
```

Corrigé – Exercice 3 – Écrire un entier en base b

Question 1. Les appels successifs sont :

```
en_base(2, 19)
```

```
en_base(2, 9)
en_base(2, 4)
en_base(2, 2)
en_base(2, 1)
```

Les listes construites lors des retours sont successivement :

```
[1]
[1, 0]
[1, 0, 0]
[1, 0, 0, 1]
[1, 0, 0, 1, 1]
```

Question 2.

```
def en_base(b: int, n: int) -> list[int]:
    """Renvoie la liste des chiffres de n en base b.

    Préconditions: 2 <= b <= 10 et n >= 0.
    """
    assert 2 <= b <= 10 and n >= 0
    if n < b:
        return [n]
    return en_base(b, n // b) + [n % b]
```

Question 3.

```
assert en_base(2, 0) == [0]
assert en_base(2, 1) == [1]
assert en_base(2, 19) == [1, 0, 0, 1, 1]
assert en_base(10, 1789) == [1, 7, 8, 9]
assert en_base(8, 64) == [1, 0, 0]
```

Corrigé – Exercice 4 – Version itérative de l'écriture en base b

Il faut traiter séparément le cas $n = 0$:

```
def en_base_iter(b: int, n: int) -> list[int]:
    """Renvoie la liste des chiffres de n en base b.

    Préconditions: 2 <= b <= 10 et n >= 0.
    """
    assert 2 <= b <= 10 and n >= 0
    if n == 0:
        return [0]

    chiffres = []
    while n > 0:
        chiffres.append(n % b)
        n = n // b
    chiffres.reverse()
    return chiffres
```

Corrigé – Exercice 5 – Recherche séquentielle

Question 1.

```
def recherche_sequentielle(v: int, lst: list[int]) -> bool:
    """Indique si v appartient à lst."""
    for element in lst:
        if element == v:
            return True # on sort tout de suite, pas besoin de
    continuer
    return False
```

Question 2.

```
def recherche_sequentielle2(v: int, lst: list[int]) -> int:
    """Indique si v appartient à lst."""
    n = len(lst)
    for k in range(n):
        if element[k] == v:
            return k
    return -1
```

Question 3. Le cas de base correspond à la liste vide :

```
def recherche_sequentielle_rec(
        v: int, lst: list[int]) -> bool:
```

```
    """Indique récursivement si v appartient à lst."""
    if len(lst) == 0:
        return False
    if lst[0] == v:
        return True
    return recherche_sequentielle_rec(v, lst[1:])
```

Question 4.

```
tests = [([], 1), ([1], 1), ([1, 3, 5], 3),
         ([1, 3, 5], 4)]
for lst, v in tests:
    assert recherche_sequentielle(v, lst) \
           == recherche_sequentielle_rec(v, lst)
```

Corrigé – Exercice 6 – Tours de Hanoï

Question 1. Pour déplacer n disques de la tige de départ vers la tige d'arrivée :

1. on déplace les $n-1$ plus petits disques de la tige de départ vers la tige intermédiaire ;
2. on déplace le plus grand disque vers la tige d'arrivée ;
3. on déplace les $n-1$ disques de la tige intermédiaire vers la tige d'arrivée.

Question 2.

```
def hanoi(n: int, depart: int,
          intermediaire: int, arrivee: int) -> None:
    """Affiche les déplacements résolvant les tours de Hanoï.

    Préconditions: n >= 1 et les trois tiges sont distinctes.
    """
    assert n >= 1
    assert depart != intermediaire
    assert depart != arrivee
    assert intermediaire != arrivee
    if n == 1:
        print(depart, "-->", arrivee)
    else:
        hanoi(n - 1, depart, arrivee, intermediaire)
        print(depart, "-->", arrivee)
        hanoi(n - 1, intermediaire, depart, arrivee)
```

Question 3. L'appel produit :

```
1 --> 3
1 --> 2
3 --> 2
1 --> 3
2 --> 1
2 --> 3
1 --> 3
```

Question 4. Le déplacement de n disques nécessite deux déplacements de $n-1$ disques et un déplacement supplémentaire. On a donc :

$$T(1) = 1 \qquad \text{et} \qquad T(n) = 2T(n-1) + 1$$

On en déduit :

$$T(n) = 2^n - 1$$

Question 5. La profondeur maximale de récursion est n. Cependant, le nombre de déplacements croît exponentiellement : le temps nécessaire à l'affichage devient prohibitif bien avant que la limite de récursion soit atteinte.

Corrigé – Exercice 7 – Recherche dichotomique

Question 1. Au départ, `g = 0` et `d = 8`. Le milieu est `m = 4` et `lst[m] = 4`. Comme $8 > 4$, on pose `g = 5`. Le nouveau milieu est `m = 6` et `lst[m] = 8` : la valeur est trouvée.

Question 2.

```
def recherche_dicho(v: int, lst: list[int]) -> bool:
    """Recherche v dans la liste croissante lst."""
    g = 0
    d = len(lst)
    while g < d:
        m = (g + d) // 2
        if lst[m] == v:
            return True
```

```
        if v < lst[m]:
            d = m
        else:
            g = m + 1
    return False
```

Question 3.

```
def recherche_dicho_rec(
        v: int, lst: list[int]) -> bool:
    """Recherche récursivement v dans la liste croissante lst."""
    if len(lst) == 0:
        return False
    m = len(lst) // 2
    if lst[m] == v:
        return True
    if v < lst[m]:
        return recherche_dicho_rec(v, lst[:m])
    return recherche_dicho_rec(v, lst[m + 1:])
```

Question 4. Dans le pire des cas, la recherche séquentielle examine n éléments. La recherche dichotomique divise par deux la taille de la fenêtre à chaque étape : elle effectue un nombre d'étapes de l'ordre de $\log_2(n)$. Cette comparaison suppose toutefois que la liste est déjà triée. De plus, la version récursive proposée crée des sous-listes ; une version récursive fondée sur des indices éviterait ces copies.

Corrigé – Exercice 8 – Sous-listes et permutations

Question 1. La liste vide possède une unique sous-liste, elle-même :

```
def sous_listes(lst: list) -> list[list]:
    """Renvoie toutes les sous-listes de lst."""
    if len(lst) == 0:
        return [[]]
    dernier = lst[-1]
    sans_dernier = sous_listes(lst[:-1])
    avec_dernier = []
    for sous_liste in sans_dernier:
        avec_dernier.append(sous_liste + [dernier])
    return sans_dernier + avec_dernier
```

Question 2.

```
def permutations(lst: list) -> list[list]:
    """Renvoie toutes les permutations de lst."""
    if len(lst) == 0:
        return [[]]
    premier = lst[0]
    permutations_reste = permutations(lst[1:])
    resultat = []
    for permutation in permutations_reste:
        for i in range(len(permutation) + 1):
            nouvelle = (permutation[:i] + [premier]
                        + permutation[i:])
            resultat.append(nouvelle)
    return resultat


assert len(sous_listes([1, 2, 3])) == 8
assert len(permutations([1, 2, 3])) == 6
```

Corrigé – Exercice 9 – Évaluation d'un polynôme par Horner

Question 1. Si $P(x) = Q(x)x + a_0$, la liste représentant Q est `p[:-1]` :

```
def eval_horner_rec(
        p: list[float], x: float) -> float:
    """Évalue en x le polynôme représenté par la liste non vide
    p."""
    assert len(p) > 0
    if len(p) == 1:
        return p[0]
    return eval_horner_rec(p[:-1], x) * x + p[-1]
```

Question 2. Pour un polynôme de degré n, la fonction effectue n multiplications.

Question 3.

```
def eval_horner_iter(
        p: list[float], x: float) -> float:
    """Évalue en x le polynôme représenté par la liste non vide
```

```
    p."""
    assert len(p) > 0
    resultat = p[0]
    for coefficient in p[1:]:
        resultat = resultat * x + coefficient
    return resultat


p = [2, 1, -1]
assert eval_horner_rec(p, 3) == 20
assert eval_horner_iter(p, 3) == 20
```

Corrigé – Exercice 10 – Courbe de Koch avec `turtle`

À la profondeur 0, on trace un segment. Sinon, chacun des quatre segments est lui-même remplacé par une courbe de profondeur inférieure d'une unité :

```
from turtle import forward, left, right


def koch(longueur: float, profondeur: int) -> None:
    """Trace une courbe de Koch."""
    assert longueur >= 0 and profondeur >= 0
    if profondeur == 0:
        forward(longueur)
    else:
        koch(longueur / 3, profondeur - 1)
        left(60)
        koch(longueur / 3, profondeur - 1)
        right(120)
        koch(longueur / 3, profondeur - 1)
        left(60)
        koch(longueur / 3, profondeur - 1)
```