Notes de cours
Semaine 6

Cours Turing

Introduction

Pour cette sixième semaine, nous allons aborder le concept de *récursivité* en programmation. La récursivité est un phénomène où une entité se définit en termes d'elle-même. Le concept peut sembler abstrait au premier abord, mais nous allons l'aborder de manière progressive avec des exemples concrets.

En programmation, la récursivité se manifeste principalement à travers les fonctions récursives et les structures de données récursives. Nous verrons également comment la récursivité est utilisée dans des techniques algorithmiques telles que *diviser pour régner*.

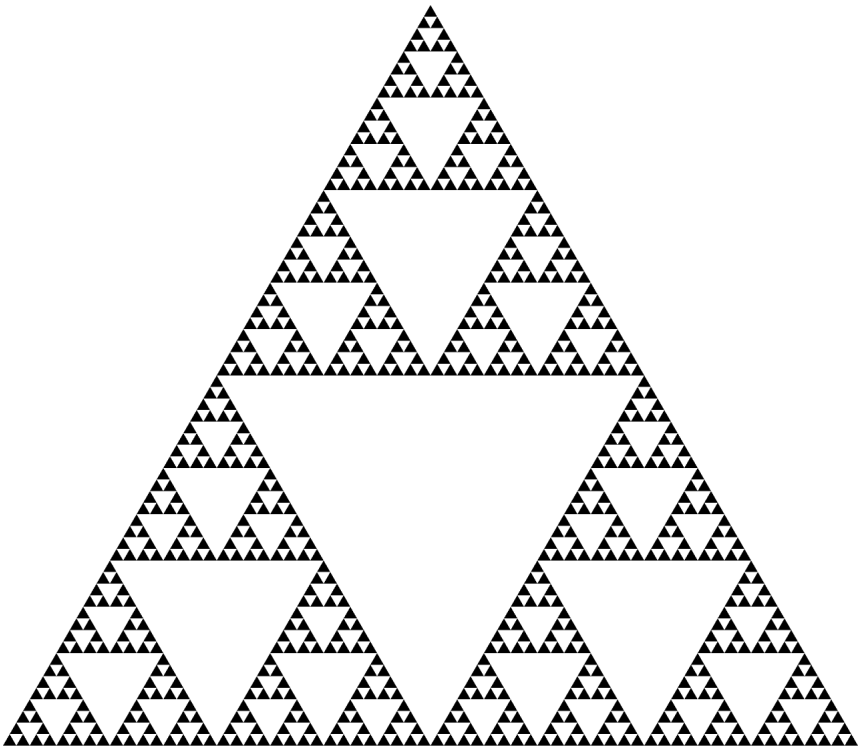

FIGURE 1 – Exemple de structure récursive : le triangle de Sierpiński

1 Fonctions récursives

On appelle une fonction *récursive* une fonction qui contient des appels à elle-même dans son corps. La récursivité est une technique de programmation puissante qui permet parfois d'implémenter des algorithmes de manière plus simple et plus élégante que les approches itératives (utilisant des boucles).

1.1 Exemple : compte à rebours

Par exemple, ci-dessous un exemple simple de fonction récursive en Python qui affiche un compte à rebours :

```python
def compte_a_rebours(n: int):
    if n == 0:
        print("Décollage !")
    else:
        print(n)
        compte_a_rebours(n - 1)
```

Dans cet exemple, la fonction `compte_a_rebours` s'appelle elle-même avec un argument décrémenté jusqu'à atteindre le cas où n est égal à 0.

On appelle **cas de base** les cas où la fonction ne s'appelle pas elle-même, ce qui permet d'éviter de faire des appels récursifs infinis. Les cas où la fonction s'appelle elle-même sont appelés **cas récursifs**.

Notez que nous aurions aussi pu, dans cet exemple, utiliser une boucle pour réaliser le compte à rebours.

```python
def compte_a_rebours_boucle(n: int):
    while n != 0:
        print(n)
        n -= 1
    print("Décollage !")
```

1.2 Exemple : factorielle

Un autre exemple classique de fonction récursive est la définition de la fonction factorielle. Pour un entier naturel n, la factorielle de n, notée $n!$, est définie comme le produit de tous les entiers positifs inférieurs ou égaux à n.

L'opération peut être définie de manière *récursive* avec les deux cas suivants :

— **Cas de base :** La factorielle de 0 est 1, c'est-à-dire :

$$0! = 1$$

.

— **Cas récursif :** Pour tout entier naturel $n > 0$, la factorielle de n est donnée par :

$$n! = n \cdot (n-1)!$$

En Python, la fonction factorielle peut être définie de manière récursive comme suit :

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

Dans l'exemple ci-dessus, la fonction `factorielle` s'appelle elle-même pour calculer la factorielle de `n - 1`. Contrairement à l'exemple du compte à rebours, le retour de la fonction est utilisé dans le calcul du résultat final.

Tout comme l'exemple précédent, il est possible de définir la fonction factorielle sans utiliser la récursivité, en utilisant une boucle à la place :

```
def factorielle_boucle(n: int) -> int:
    resultat = 1
    for i in range(1, n + 1):
        resultat *= i
    return resultat
```

1.3 Exemple : suite de Fibonacci

La suite de Fibonacci est une séquence de nombres où chaque nombre est la somme des deux précédents. La suite commence généralement par les nombres 0 et 1. Ainsi, les premiers termes de la suite de Fibonacci sont :

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

La suite de Fibonacci peut être définie de manière récursive avec les deux cas suivants :

— **Cas de base :**

$$F(0) = 0$$

$$F(1) = 1$$

— **Cas récursif :** Pour tout entier naturel $n \geq 2$,

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

En Python, la fonction pour calculer le n-ième terme de la suite de Fibonacci de manière récursive peut être définie comme suit :

```
def fibonacci(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Nous avons ici deux cas de base (pour $n = 0$ et $n = 1$) et un cas récursif pour les autres valeurs de n.

Contenu avancé

Il est important de noter que cette implémentation récursive de la suite de Fibonacci n'est pas très efficace pour de grandes valeurs de n, car elle entraîne de nombreux appels redondants. Il existe cependant des méthodes afin de regagner en efficacité. Nous en parlerons plus en détail plus bas dans le document.

1.4 Structure d'une fonction récursive

Une fonction récursive traite généralement deux types de cas en fonction de son ou ses arguments :

- **Cas de base :** On appelle *cas de base* les cas où la fonction ne s'appelle pas elle-même, où elle retourne une valeur simple ou effectue une action simple. Ces cas permettent de terminer la récursion.
- **Cas récursifs :** On appelle *cas récursifs* les cas où la fonction s'appelle elle-même avec des arguments modifiés (généralement plus simples ou plus petits) pour progresser vers un cas de base. Le nombre d'appels récursifs dépend de la nature du problème et de la manière dont les arguments sont modifiés.

Pour que les appels récursifs se terminent correctement, il est crucial que chaque suite d'appels récursifs se termine par un cas de base. Très souvent, cela implique qu'un ou plusieurs arguments de la fonction soient modifiés à chaque appel récursif pour se rapprocher du cas de base.

1.5 Comparaison entre récursivité et itération

La récursivité (utilisation de fonctions récursives) et l'itération (utilisation de boucles) sont deux approches différentes qui peuvent être utilisées de manière interchangeable pour résoudre certains problèmes. En effet, les deux techniques ont la même puissance expressive, ce qui signifie que tout algorithme pouvant être implémenté de manière itérative peut également être implémenté de manière récursive, et vice versa. Cependant, certains problèmes se prêtent mieux à l'une ou l'autre approche en fonction de leur nature.

Au niveau des performances, les fonctions récursives peuvent parfois être moins efficaces que les boucles en raison de la surcharge liée aux appels de fonction et à la gestion de la pile d'appels.

La récursivité reste néanmoins une technique puissante et élégante pour résoudre des problèmes qui ont une structure naturellement récursive, comme nous allons le voir dans les prochaines sections.

Contenu avancé

Parfois, la structure récursive d'un problème amène à des appels répétés aux mêmes sous-problèmes, ce qui peut entraîner une inefficacité. Dans de tels cas, des techniques comme la *mémoïsation* ou la *programmation dynamique* peuvent être utilisées pour optimiser les performances en stockant les résultats des sous-problèmes déjà résolus.
Un exemple classique est le calcul des nombres de Fibonacci, où une approche naïve récursive peut entraîner une explosion du nombre d'appels.

```
def fibonacci(n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Un appel à `fibonacci(5)` entraînera les appels suivants :
- `fibonacci(5)` appelle `fibonacci(4)` et `fibonacci(3)`
- `fibonacci(4)` appelle `fibonacci(3)` et `fibonacci(2)`
- `fibonacci(3)` appelle `fibonacci(2)` et `fibonacci(1)`
- `fibonacci(2)` appelle `fibonacci(1)` et `fibonacci(0)`

En comptabilisant les appels, on obtient :
- `fibonacci(5)` : 1 appel
- `fibonacci(4)` : 1 appel
- `fibonacci(3)` : 2 appels
- `fibonacci(2)` : 3 appels
- `fibonacci(1)` : 5 appels
- `fibonacci(0)` : 3 appels

Le nombre total d'appels est donc de 15 pour calculer `fibonacci(5)`.
Grâce à la mémoïsation, on peut améliorer considérablement l'efficacité de cette fonction.

```
memo = {}  # Dictionnaire pour stocker les résultats déjà calculés.
def fibonacci_memo(n: int) -> int:
    if n in memo:  # On vérifie si le résultat est déjà calculé.
        return memo[n]
    if n <= 1:
        result = n
    else:
        result = fibonacci_memo(n - 1) + fibonacci_memo(n - 2)
    memo[n] = result
    return result
```

Avec cette version *mémoïsée*, chaque nombre de Fibonacci n'est calculé qu'une seule fois, réduisant ainsi le nombre total d'appels à `fibonacci_memo(n)` à `n + 1`.

2 Diviser pour régner

La récursivité est particulièrement utile pour résoudre des problèmes qui peuvent être décomposés en sous-problèmes plus petits et similaires. On appelle cette technique d'algorithmique *diviser pour régner*. L'idée principale de cette approche est de diviser un problème en plusieurs sous-problèmes plus petits, de résoudre ces sous-problèmes de manière récursive, puis de combiner les résultats pour obtenir la solution au problème initial.

2.1 Exemple : recherche dichotomique

Un exemple classique de l'approche *diviser pour régner* est la recherche dichotomique. La recherche dichotomique permet de trouver un élément dans une liste triée en divisant continuellement la liste en deux moitiés et en se concentrant sur la moitié qui pourrait contenir l'élément recherché.

Voici une implémentation récursive de la recherche dichotomique en Python :

```python
def recherche_dicho(liste: list[int], cible: int, debut: int, fin: int) -> int:
    if debut > fin:
        return None
    milieu = (debut + fin) // 2
    if liste[milieu] == cible:
        return milieu
    elif liste[milieu] < cible:
        return recherche_dicho(liste, cible, milieu + 1, fin)
    else:
        return recherche_dicho(liste, cible, debut, milieu - 1)
```

Dans cet exemple, la fonction `recherche_dichotomique` divise la liste en deux moitiés à chaque appel récursif, et se concentre sur la moitié qui pourrait contenir l'élément recherché.

2.2 Exemple : rendu de monnaie

Un autre exemple d'algorithme *diviser pour régner* est le problème du rendu de monnaie. Le but est de calculer combien de façons différentes on peut rendre une somme donnée avec un ensemble de pièces de monnaie de différentes valeurs. Par exemple, pour rendre 5 francs avec des pièces de 1 franc et 2 francs, on peut le faire de plusieurs façons :

— 5 pièces de 1 franc
— 3 pièces de 1 franc et 1 pièce de 2 francs
— 1 pièce de 1 franc et 2 pièces de 2 francs

Soit un total de 3 façons de rendre 5 francs avec des pièces de 1 franc et 2 francs.

L'idée pour résoudre ce problème est de réfléchir de manière récursive. Commençons par identifier les cas de base : les cas où la réponse est immédiate.

— Si la somme à rendre est 0, il y a exactement une façon de le faire : ne rien rendre.
— Si par contre la somme à rendre est négative ou si nous n'avons plus de pièces disponibles, il n'y a aucune façon de rendre cette somme.

Vient ensuite le cas récursif. Dans ce cas, nous avons une somme strictement positive à rendre et au moins une pièce disponible. Nous avons alors deux options :

— Utiliser la première pièce disponible pour rendre une partie de la somme, puis résoudre le problème pour la somme restante avec toutes les pièces disponibles.

— Ne pas utiliser la première pièce, et résoudre le problème pour la même somme mais avec les pièces restantes (sans la première pièce).

En combinant les résultats de ces deux options, nous obtenons le nombre total de façons de rendre la somme.

Voici une implémentation récursive de cette approche en Python :

```python
def rendu_de_monnaie(somme: int, pieces: list[int]) -> int:
    # Cas de base.
    if somme == 0:
        return 1  # Une façon de rendre la somme 0 (ne rien faire).
    if somme < 0 or not pieces:
        return 0  # Pas de façon de rendre une somme négative ou sans pièces.

    # Cas récursif.
    # Soit on utilise la première pièce.
    avec_piece = rendu_de_monnaie(somme - pieces[0], pieces)
    # Soit on n'utilise pas la première pièce.
    sans_piece = rendu_de_monnaie(somme, pieces[1:])
    return avec_piece + sans_piece
```

3 Structures de données récursives

Dans les deux sections précédentes, nous avons vu la notion de fonctions récursives et l'approche algorithmique *diviser pour régner*. Il s'agit en quelque sorte du pendant procédural de la récursivité. La récursivité peut se retrouver également dans le monde des données, et non plus seulement des calculs, à travers les *structures de données récursives*.

Une structure de données est dite *récursive* si elle est définie en termes d'elle-même. Cela signifie qu'une instance de la structure peut contenir des références à d'autres instances de la même structure.

Les structures de données récursives sont particulièrement utilisées pour représenter des données hiérarchiques ou imbriquées.

Exemple

Si vous avez déjà travaillé avec des documents HTML (pour les pages web), vous avez déjà manipulé une structure de données récursive. En effet, un document HTML est constitué d'éléments qui peuvent contenir d'autres éléments, formant ainsi une hiérarchie d'éléments imbriqués.

3.1 Exemple : listes chaînées

La liste chaînée est l'exemple classique de structure de données récursive, et peut-être l'exemple le plus simple. Une liste chaînée est une collection d'éléments, où chaque élément (appelé *nœud*) contient une valeur et une référence vers le nœud suivant dans la liste. Une liste vide est représentée par une valeur nulle, comme par exemple `None` en Python.

En Python, on pourrait utiliser un tuple de deux éléments pour représenter un nœud d'une liste chaînée non-vide et None pour représenter une liste vide. Voici une définition simple d'une liste chaînée en Python, encodée de cette manière :

```
ma_liste_chaînée = (1, (2, (3, None)))
```

Dans cet exemple, la liste chaînée contient les éléments 1, 2 et 3. Le premier élément est un tuple contenant la valeur 1 et une référence vers le nœud suivant, qui est lui-même un tuple contenant la valeur 2 et une référence vers le nœud suivant, et ainsi de suite jusqu'à ce que le dernier nœud référence `None`.

Pour manipuler une liste chaînée, on utilise souvent des fonctions récursives. Par exemple, voici une fonction récursive pour calculer la somme des éléments d'une liste chaînée :

```
def somme_liste_chaînée(liste: tuple | None) -> int:
    if liste is None:
        return 0
    else:
        valeur, reste = liste
        return valeur + somme_liste_chaînée(reste)
```

Dans cet exemple, la fonction `somme_liste_chaînée` s'appelle elle-même pour calculer la somme des éléments restants de la liste chaînée.

Contenu avancé

Le type qui correspond à une liste chaînée d'entiers peut être défini en Python (à partir de la version 3.12) comme suit :

```
type LinkedList[T] = tuple[T, LinkedList[T]] | None
```

Ici, `LinkedList[T]` est un type générique qui peut contenir des éléments de type `T`. Une liste chaînée est soit un tuple contenant une valeur de type `T` et une référence vers une autre liste chaînée de type `LinkedList[T]`, soit `None` pour représenter une liste vide.

3.2 Exemple : arbres

Une autre structure de données récursive couramment utilisée est l'*arbre*. Alors qu'un nœud d'une liste chaînée ne peut avoir qu'un seul nœud enfant (le nœud suivant), un nœud d'un arbre peut avoir plusieurs nœuds enfants. Un arbre est constitué de nœuds, où chaque nœud contient une valeur et une *liste* de références vers ses nœuds enfants. Un arbre vide est souvent représenté par une valeur nulle, comme `None` en Python.

Voici un exemple simple d'un arbre en Python :

```
mon_arbre = (1, [(2, []), (3, [(4, [])])])
```

Dans cet exemple, l'arbre a pour *racine* le nœud avec la valeur 1. Ce nœud a deux enfants : le nœud avec la valeur 2 (qui n'a pas d'enfants) et le nœud avec la valeur 3 (qui a un enfant, le nœud avec la valeur 4).

Comme pour les listes chaînées, on utilise souvent des fonctions récursives pour manipuler des arbres. Par exemple, voici une fonction récursive pour calculer la somme des valeurs de tous les nœuds d'un arbre :

```
def somme_arbre(arbre: tuple | None) ->
    if arbre is None:
        return 0
    else:
        valeur, enfants = arbre
        sommes_enfants = [somme_arbre(enfant) for enfant in enfants]
        return valeur + sum(sommes_enfants)
```

Les arbres sont très utiles pour représenter des données hiérarchiques, comme par exemple les systèmes de fichiers et dossiers, ou les structures de documents XML ou JSON. On retrouve également des structures arborescentes en dehors de l'informatique, comme par exemple les arbres phylogénétiques en biologie ou les organisations hiérarchiques dans les entreprises. Les arbres permettent de modéliser efficacement ces relations hiérarchiques.