Récursivité

Activité – Terminale NSI

① Une fonction qui s'appelle elle-même

Exercice 1 *La factorielle, deux écritures*

On rappelle que la **factorielle** d'un entier $n \geqslant 0$ est $n! = 1 \times 2 \times 3 \times \cdots \times n$, avec $0! = 1$ par convention. Voici deux versions Python qui calculent $n!$:

Version itérative.

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

Version récursive.

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

1. Calculer à la main `fact_iter(4)` et `fact_rec(4)` en détaillant les étapes pour chaque version.
2. En observant la version récursive, repérer :
   — le **cas de base** : pour quelle valeur de n la fonction renvoie-t-elle un résultat *sans* se rappeler ?
   — l'**appel récursif** : où se trouve l'appel à `fact_rec` ?
3. Que se passerait-il si l'on oubliait le cas de base `if n == 0: return 1` ?

Remarque 1. Une fonction **récursive** est une fonction qui *s'appelle elle-même* sur un cas « plus petit ». Pour qu'elle se termine, deux ingrédients sont indispensables :

— un **cas de base**, traité directement ;
— un **appel récursif** qui se rapproche du cas de base à chaque étape.

② Tracer une fonction récursive

Exercice 2 *Trace de `somme(4)`*

Considérer la fonction :

```python
def somme(n):
    """Renvoie 1 + 2 + ... + n."""
    if n == 0:
        return 0
    return n + somme(n - 1)
```

1. Compléter la « descente » d'appels :

`somme(4) = 4 + somme(3)`
`somme(3) = ..... + somme(.....)`
`somme(2) = ..... + somme(.....)`
`somme(1) = ..... + somme(.....)`
`somme(0) = .....` (*cas de base*)

2. Compléter la « remontée » qui calcule les valeurs en reprenant chaque ligne dans l'ordre inverse :

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

3. Combien d'appels récursifs sont effectués au total pour calculer `somme(n)` ?

③ Récursivité sur les listes

Propriété 1 **(Décomposition d'une liste)** Pour traiter une liste `L` récursivement, on la décompose en deux parties :

— le **premier élément** `L[0]` ;

— le **reste** `L[1:]`.

Le **cas de base** est la liste vide `[]`.

Exercice 3 *Trois fonctions récursives sur les listes*

1. Compléter la fonction `longueur(L)` qui renvoie la longueur d'une liste, sans utiliser `len` :

```python
def longueur(L):
    if L == []:
```

```
        return ............
    return ............ + longueur(............)
```

2. Compléter la fonction `somme_liste(L)` qui renvoie la somme des éléments d'une liste de nombres :

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

3. Compléter la fonction `maximum(L)` qui renvoie le maximum d'une liste non vide :

```
def maximum(L):
    assert len(L) >= 1, "Liste vide"
    if len(L) == 1:
        return ............
    reste = maximum(............)
    if L[0] > reste:
        return ............
    else:
        return ............
```

④ Quand la récursivité explose : Fibonacci

Exercice 4 *`fibo(5)` et son arbre d'appels*

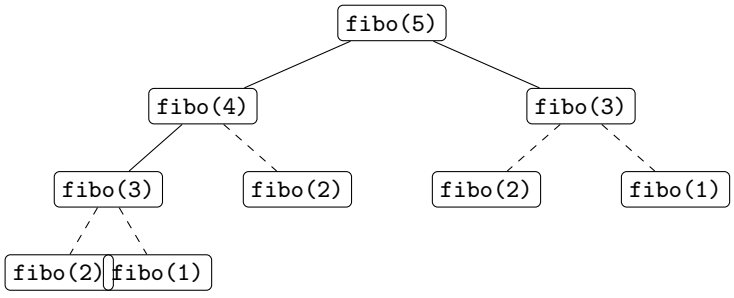
On définit la suite de Fibonacci par $F_0 = 0$, $F_1 = 1$ et, pour $n \geqslant 2$:

$$F_n = F_{n-1} + F_{n-2}.$$

Une traduction directe en Python :

```
def fibo(n):
    if n == 0 or n == 1:
        return n
    return fibo(n - 1) + fibo(n - 2)
```

1. Compléter l'arbre d'appels de `fibo(5)` ci-dessous (chaque nœud représente un appel) :



Les pointillés indiquent les sous-arbres à compléter sur le cahier (jusqu'aux feuilles `fibo(0)` ou `fibo(1)`).

2. Combien de feuilles l'arbre contient-il en tout ? Combien sont étiquetées `fibo(0)` ? `fibo(1)` ?

3. Combien d'appels récursifs sont effectués au total pour calculer `fibo(5)` ?
4. Pour `fibo(50)`, le nombre d'appels devient gigantesque (de l'ordre de 10^{10}). Pourquoi cet algorithme est-il alors inutilisable ? Quelle est la cause profonde de ce gaspillage ?

5 Récursivité ou itération ?

Exercice 5 *Choisir le bon outil*

Pour chaque problème, indiquer si une approche **récursive** ou **itérative** semble plus naturelle, et expliquer en une phrase.

1. Calculer la somme des entiers de 1 à n.
2. Calculer le PGCD de deux entiers via l'algorithme d'Euclide ($\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b)$, cas de base $b = 0$).
3. Inverser une chaîne de caractères.
4. Parcourir tous les fichiers d'un dossier et de ses sous-dossiers.
5. Trouver le maximum d'une liste.

6 Ce qu'il faut retenir

Propriété 2 **(Définition d'une fonction récursive)** Une fonction récursive doit comporter :

— au moins un **cas de base** (qui ne déclenche pas d'appel récursif) ;

— au moins un **appel récursif** qui se rapproche du cas de base.

Sans cas de base, la récursion est infinie et Python lève une `RecursionError`.

Propriété 3 **(Schéma sur les listes)** Pour traiter une liste `L` récursivement, on raisonne par :

— **cas de base** : la liste vide `[]` (ou la liste à un élément) ;

— **cas récursif** : combiner `L[0]` avec le résultat sur `L[1:]`.

Remarque 2. La récursivité *naïve* (comme `fibo`) peut conduire à un nombre d'appels exponentiel à cause des sous-problèmes recalculés. La **programmation dynamique** (mémoïsation) corrige ce défaut. La récursivité est aussi à la base d'algorithmes très puissants : *diviser pour régner*, parcours d'arbres et de graphes, exploration de structures hiérarchiques.

Solutions des exercices

Corrigé de l'exercice 1.

1. **Itératif** : `resultat` prend successivement les valeurs 1, 1, 2, 6, 24. Renvoie 24.

   Récursif : `fact_rec(4) = 4 * fact_rec(3) = 4 * 3 * fact_rec(2) = 4 * 3 * 2 * fact_rec(1) = 4 * 3 * 2 * 1 * fact_rec(0) = 4 * 3 * 2 * 1 * 1 = 24`.

2. Cas de base : $n = 0$, qui renvoie 1 directement. Appel récursif : `fact_rec(n - 1)` dans la dernière ligne.

3. Sans cas de base, la fonction s'appellerait indéfiniment avec des valeurs négatives (-1, -2, ...). Python finirait par lever une `RecursionError` (limite par défaut : 1 000 appels imbriqués).

Corrigé de l'exercice 2.

1. Descente :

`somme(4) = 4 + somme(3)`
`somme(3) = 3 + somme(2)`
`somme(2) = 2 + somme(1)`
`somme(1) = 1 + somme(0)`
`somme(0) = 0` (*cas de base*)

2. Remontée :

`somme(0) = 0`
`somme(1) = 1 + 0 = 1`
`somme(2) = 2 + 1 = 3`
`somme(3) = 3 + 3 = 6`
`somme(4) = 4 + 6 = 10`

3. $n + 1$ appels au total (`somme(n)`, `somme(n - 1)`, ..., `somme(0)`).

Corrigé de l'exercice 3.

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

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

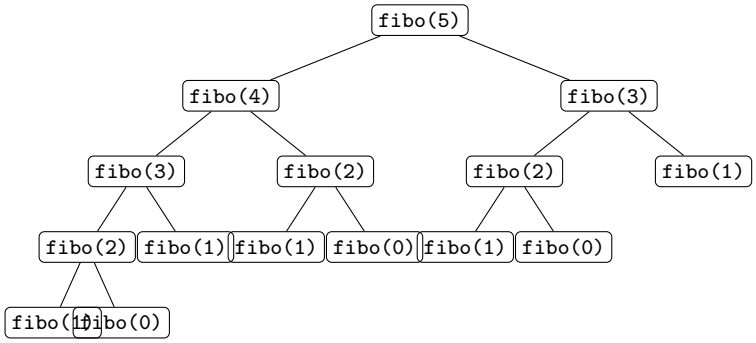
def maximum(L):
    assert len(L) >= 1, "Liste vide"
    if len(L) == 1:
```

```
        return L[0]
    reste = maximum(L[1:])
    if L[0] > reste:
        return L[0]
    else:
        return reste
```

Schéma général : cas de base sur la liste vide (ou la liste à un élément), et appel récursif sur `L[1:]`.

Corrigé de l'exercice 4.

1. Arbre complet :



2. L'arbre contient **8 feuilles** : 5 étiquetées `fibo(1)` et 3 étiquetées `fibo(0)` (on remarque que $5+3 = 8 = F_6$, ce n'est pas un hasard).
3. L'arbre contient 15 nœuds en tout, donc 15 appels.
4. De nombreux sous-problèmes sont calculés **plusieurs fois** (par exemple `fibo(2)` apparaît trois fois dans l'arbre de `fibo(5)`). Pour `fibo(50)`, ce gaspillage devient catastrophique : le nombre d'appels croît comme φ^n (où φ est le nombre d'or). C'est précisément le défaut que la **programmation dynamique** corrige (en mémorisant les résultats déjà calculés).

Corrigé de l'exercice 5.

1. **Itératif** (boucle `for` avec accumulateur, plus simple).
2. **Récursif** (la définition mathématique elle-même est récursive).
3. Les deux conviennent ; la version itérative est plus simple en Python (slicing inverse).
4. **Récursif** (la structure des dossiers est elle-même récursive : un dossier contient des sous-dossiers).
5. **Itératif** (boucle `for` avec recherche d'extremum, plus efficace).

Règle générale : la récursivité est élégante quand le problème a une *structure récursive naturelle* (arbres, dossiers, division en sous-problèmes). Sinon, une boucle est souvent préférable car elle évite la création d'une pile d'appels.