RÉCURSIVITÉ

À la fin de ce chapitre, je sais :

- ☞ expliquer le principe d'un algorithme récursif
- ☞ imaginer une version récursive d'un algorithme
- ☞ trouver et coder une condition d'arrêt à la récursivité
- ☞ coder des algorithmes à récursivité simple et multiple en Python
- ☞ identifier le type de récursivité d'un algorithme

A Principes

La récurrence est une méthode à la fois simple et puissante pour définir un objet ou résoudre un problème mathématique. La récursivité, c'est la projection de cette méthode en informatique. La plupart des langages contemporains sont récursifs, c'est à dire qu'ils permettent de programmer récursivement.

■ **Définition 1 — Algorithme récursif.** Un algorithme $\mathcal{A}$ qui résout un problème $\mathcal{P}$ est dit récursif si, où au cours de son exécution, il s'utilise lui-même pour résoudre le problème $\mathcal{P}$ avec des données d'entrées différentes.

■ **Définition 2 — Appel récursifs d'une fonction.** On désigne par le terme appel récursif l'utilisation d'une fonction dans la définition de la fonction elle-même.

■ **Définition 3 — Pile d'exécution.** La pile dexécution d'un processus est un emplacement mémoire destiné à mémoriser les appels de fonction, les paramètres, les variables locales ainsi que ladresse de retour[a] de chaque fonction en cours dexécution.

a. c'est à dire la case mémoire qui va contenir la valeur renvoyée par la fonction.

Vocabulary 1 — Call stack ⇜⇝ Pile d'exécution

L'algorithme 1 est un algorithme récursif qui permet de calculer la fonction factorielle d'un entier naturel. La ligne 5 contient l'appel récursif. À la lecture de cet algorithme, on peut être pris d'un vertige : cela va-t-il se terminer? Est-possible de s'appeler soi-même pour résoudre

un problème ? Comment gérer en mémoire les appels successifs ?

Algorithme 1 Factoriel récursif

```
1 : Fonction FACT(n)
2 :     si n <= 1 alors                          ▷ Condition d'arrêt
3 :         renvoyer 1
4 :     sinon
5 :         renvoyer n × FACT(n-1)               ▷ Appel récursif
```

Pour calculer, cet algorithme procède comme suit : il suspend momentanément le calcul du résultat de la fonction à la réception du résultat de l'appel récursif en empilant l'appel récursif sur la pile d'exécution. La pile d'exécution est donc une zone mémoire essentielle qui garantit le bon déroulement des appels successifs des fonctions récursives ou non.

■ **Exemple 1 — Calculer 3! récursivement.** Dans le cas de 3!, l'exécution se présente ainsi :

1. appel de FACT(3),
2. attente du résultat de FACT(2) qui est empilé sur la pile d'exécution,
3. attente du résultat de FACT(1) qui est empilé sur la pile d'exécution,
4. FACT(1) renvoie 1, on dépile,
5. 1 est multiplié par 2, 2 est renvoyé par FACT(2), on dépile,
6. 2 est multiplié par 3 et 6 est renvoyé par FACT(3).

La figure 1 illustre ce fonctionnement.

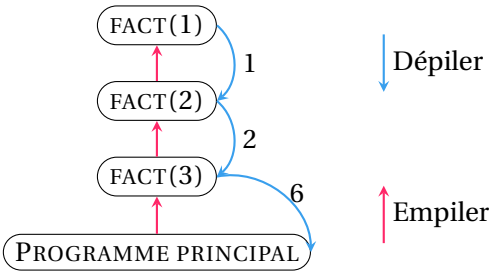

FIGURE 1 – Vision de la pile d'exécution de l'algorithme factoriel récursif 1 pour le calcul de 3!

R Comme toute zone mémoire, la pile d'exécution est limitée, ce qui a des conséquences concrètes en programmation : on ne peut pas effectuer une infinité d'appels récursif. Une limite existe à la profondeur de la récursivité : la nature est têtue, notre monde physique demeure fini.

R La définition 1 ne garantit pas le bon fonctionnement d'un algorithme récursif. Pour qu'un algorithme récursif aboutisse correctement, il est nécessaire que le nombre d'appels récursifs soit fini. Dans le cas contraire, la pile d'exécution explose en mémoire : il n'y a plus assez de place pour la stocker et le système s'effondre.

M **Méthode 1 — Formuler correctement un algorithme récursif** Afin d'éviter les appels récursifs infinis, il est nécessaire d'être vigilant lors de l'écriture des algorithmes récursifs. Il faut :

1. distinguer au moins un cas sans appel récursif qui permet de terminer l'algorithme, c'est à dire **une condition d'arrêt**,
2. appeler récursivement avec des données **plus proches** des données qui satisfont la condition d'arrêt.

Dans le cas de l'algorithme 1, la condition d'arrêt est la ligne 2. Par ailleurs, on voit bien que l'appel récursif stipule paramètre $n-1$ qui se rapproche de cette condition d'arrêt. Une formulation plus correcte et plus théorique repose sur l'induction et la notion d'ordre bien fondé.

B Types de récursivité

■ **Définition 4 — Récursivité simple.** La récursivité est dite simple si une exécution de l'algorithme aboutit à un seul appel récursif.

■ **Exemple 2 — Algorithmes simplement récursifs.** L'algorithme factoriel récursif 1 est une récursivité simple. Il est possible de formuler l'algorithme de recherche dichotomique récursivement (cf. algorithme 2). Cet algorithme est également de récursivité simple : chaque chemin d'exécution n'opère qu'un seul appel récursif.

■ **Définition 5 — Récursivité multiple.** La récursivité est dite multiple si une exécution de l'algorithme aboutit à plusieurs appels récursifs.

■ **Exemple 3 — Algorithmes à récursivité multiple.** Les algorithmes calculant la suite de Fibonacci, la solution au jeu des tours d'Hanoi ou l'exemple 3 sont à récursivité multiples. L'algorithme 3 est à récursivité multiple car la fonction est appelée deux fois à la ligne sept.

Algorithme 2 Recherche récursive d'un élément par dichotomie dans un tableau trié

```
1 : Fonction REC_DICH(t, g, d, elem)
2 :     si g = d alors                                   ▷ Condition d'arrêt
3 :         si t[g] = elem alors
4 :             renvoyer g
5 :         sinon
6 :             renvoyer l'élément n'a pas été trouvé
7 :     sinon
8 :         m ← (g+d)//2
9 :         si t[m] < elem alors
10 :            REC_DICH(t, m+1, d, elem)                ▷ Appel récursif
11 :        sinon
12 :            REC_DICH(t, g, m, elem)                  ▷ Appel récursif
```

Algorithme 3 Est-il mon aïeul?

```
1 : Fonction EST_AÏEUL(p1, p2)
2 :     si p1 est un des parents de p2 alors
3 :         renvoyer Vrai
4 :     sinon si p1 est plus jeune que p2 alors
5 :         renvoyer Faux
6 :     sinon
7 :         renvoyer EST_AÏEUL(p1, mère(p2)) ou EST_AÏEULÀ(p1, père(p2))
```

■ **Définition 6 — Récursivité terminale.** Un algorithme est dit à récursivité terminale s'il renvoie directement le résultat de l'appel récursif sans opération supplémentaire. Cette propriété rend l'opération de dépilement de la pile d'exécution très efficace puisqu'on n'a pas à faire de calculs supplémentaires.

■ **Exemple 4 — Factoriel récursif terminal.** On peut transformer l'algorithme 1 pour qu'il présente une récursivité terminale (cf. 4)

Algorithme 4 Factoriel récursif terminal

```
1 : Fonction FACT(n, acc)                    ▷ on ajoute un accumulateur
2 :     si n <= 1 alors                      ▷ Condition d'arrêt
3 :         renvoyer acc
4 :     sinon
5 :         renvoyer FACT(n-1, n × acc )     ▷ Appel récursif avec accumulateur
```

C Du récursif à l'itératif

Dans la pratique, un algorithme est souvent énoncé récursivement car l'expression est plus puissante, plus intelligible et plus facile à trouver [1]. Cependant, on convertit la plupart du temps l'algorithme récursif en itératif afin d'être plus performant : il s'agit d'éviter d'avoir à effectuer les appels récursifs successifs qui ralentissent souvent l'exécution sur les machines concrètes.

M **Méthode 2 — Du récursif à l'itératif** La formulation d'un algorithme sous la forme d'un algorithme récursif **terminal** permet de facilement en déduire une version itérative. Le principe est expliqué par les algorithmes 5 et 6.

En appliquant la méthode décrite par ces algorithmes, on peut retrouver facilement la version itérative de factorielle (cf. algorithme 7).

Algorithme 5 Algorithme récursif terminal

```
1 : Fonction REC_TERM(n)
2 :     si ARRÊTER(n) alors
3 :         renvoyer SOLUTION(n)
4 :     sinon
5 :         CALCULER_AVEC(n)
6 :         renvoyer REC_TERM(MODIFIER(n))
```

Algorithme 6 Version itérative d'un algorithme récursif terminal

```
1 : Fonction ITER(n)
2 :     tant que non ARRÊTER(n) répéter
3 :         CALCULER_AVEC(n)
4 :         n ← MODIFIER(n)
5 :     renvoyer SOLUTION(n)
```

Algorithme 7 Factoriel itératif

```
1 : Fonction FACT(n)
2 :     acc ← 1
3 :     tant que n > 1 répéter
4 :         acc ← n × acc
5 :         n ← n - 1
6 :     renvoyer acc
```

1. parfois…

R Programmer récursivement est une manière de penser très structurante. Elle repose sur le principe d'induction [a]. Les algorithmes proposés en TP permettent de développer cette compétence importante.

a. exposé en option informatique