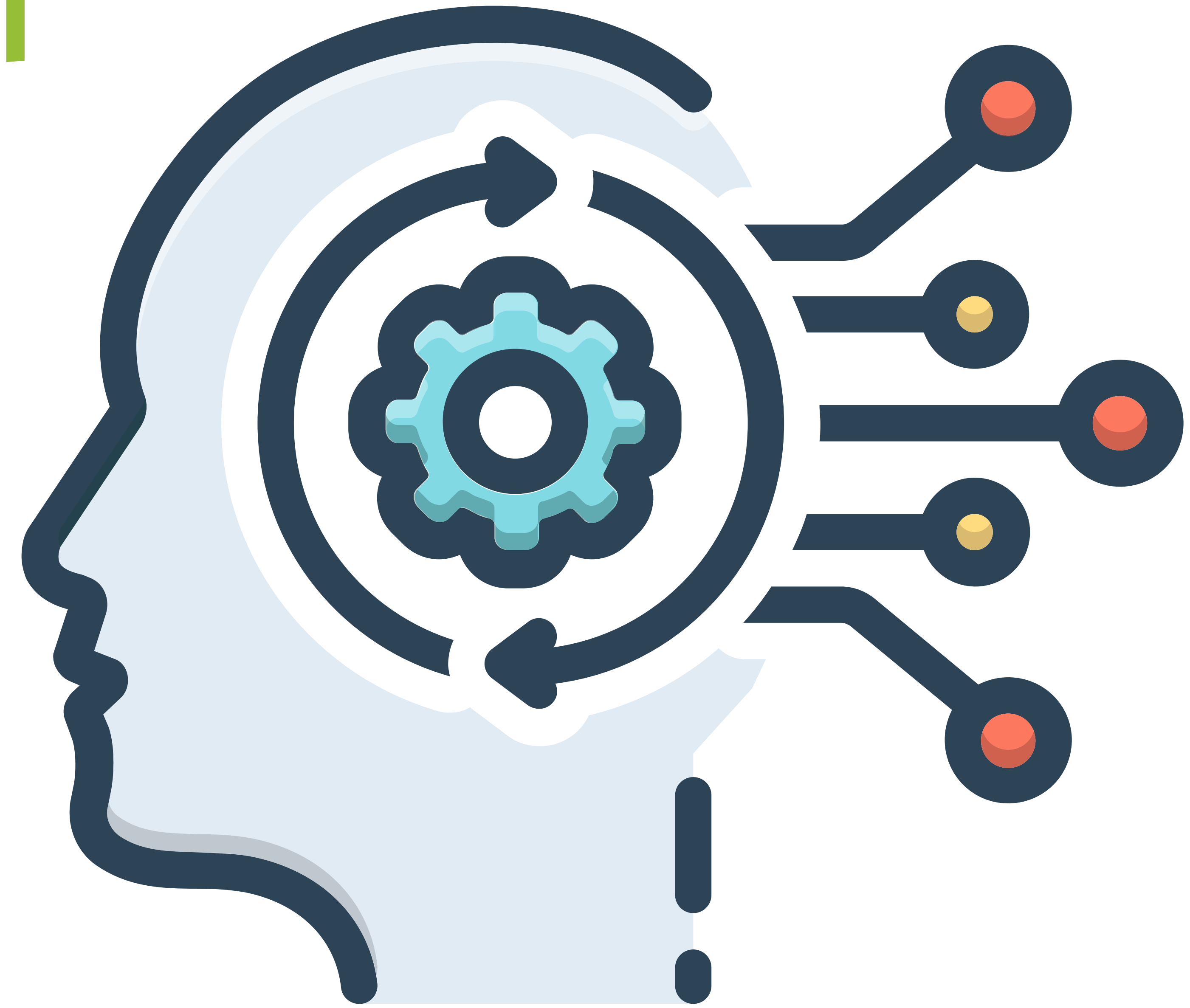
DR B. DIOP
babacar.diop@ussein.edu.sn

Algorithmes et Programmation II

Récursivité



INFO – AGROTIC

Last update: oct. 2025

C'est quoi, la récursivité ?

Définition

- En programmation, c'est une technique où une fonction s'appelle elle-même pour résoudre un problème.
- L'objectif : Décomposer un gros problème en versions plus petites et plus simples du même problème.

C'est quoi, la récursivité ?

Syntaxe d'une fonction récursive

- Voici la structure de base en C :

```c
return_type nom_fonction(parametres) {

    // 1. Cas de base (Condition d'arrêt)
    if (condition_de_base) {
        // Retourner une valeur simple (fin de la récursion)
    }

    // 2. Cas récursif (L'appel à soi-même)
    else {
        // Appel récursif avec des paramètres modifiés
        // (on se rapproche du cas de base)
        nom_fonction(parametres_modifies);
    }
}
```

C'est quoi, la récursivité ?

Syntaxe d'une fonction récursive

- Voici la structure de base en C :

```c
return_type nom_fonction(parametres) {

    // 1. Cas de base (Condition d'arrêt)
    if (condition_de_base) {
        // Retourner une valeur simple (fin de la récursion)
    }

    // 2. Cas récursif (L'appel à soi-même)
    else {
        // Appel récursif avec des paramètres modifiés
        // (on se rapproche du cas de base)
        nom_fonction(parametres_modifies);
    }
}
```

Cas de base

Cas récursif

Pourquoi utiliser la Récursivité ?

Pourquoi utiliser la Récursivité ?

- Élégance : Elle permet d'écrire des solutions très claires et concises pour des problèmes complexes (ex: parcours d'arbres, algorithmes "diviser pour régner").
- Approche naturelle : Certains problèmes sont intrinsèquement récursifs (ex: Factoriel, Fibo).

Points de vigilance ⚠️

- Performance : Chaque appel de fonction consomme de la mémoire sur la "pile" (stack). Une récursion trop profonde peut causer un "Stack Overflow".
- Complexité : Peut être plus difficile à déboguer qu'une simple boucle for ou while.
- La clé absolue : Toujours, toujours, toujours avoir un cas de base !

Les 2 Piliers de la Récursivité

Toute fonction récursive repose sur deux composants essentiels :

1. Le Cas de Base (Base Case)
   - C'est la condition d'arrêt.
   - C'est le moment où le problème est si simple que la fonction peut retourner une réponse directe sans s'appeler à nouveau.
   - Sans cas de base, la fonction s'appelle à l'infini ! (erreur "Stack Overflow" 💥).
2. Le Cas Récursif (Recursive Case)
   - C'est le moment où la fonction s'appelle elle-même. 🔄
   - L'astuce est de s'appeler avec une version réduite ou simplifiée du problème.

Chaque appel récursif doit se rapprocher un peu plus du cas de base.

Récursivité : calcul du factoriel

- Rappel mathématique : $5! = 5 \times 4 \times 3 \times 2 \times 1$

- Logique récursive :

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

...

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

```
int factorial(int n) {

    // Cas de base : 0! ou 1! valent 1
    if (n == 0 || n == 1) {
        return 1;
    }

    // Cas récursif : n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}
```

Récursivité : calcul du factoriel

Comprendre la "Pile" : Trace de factorial(3)

- factorial(3) est appelé.
- n n'est pas 1. Cas récursif.
- Doit retourner 3 * factorial(2).
- Met factorial(2) en attente...
- factorial(2) est appelé.
- n n'est pas 1. Cas récursif.
- Doit retourner 2 * factorial(1).

```
int factorial(int n) {

    // Cas de base : 0! ou 1! valent 1
    if (n == 0 || n == 1) {
        return 1;
    }

    // Cas récursif : n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}
```

Récursivité : calcul du factoriel

Comprendre la "Pile" : Trace de factorial(3)

- (... suite)
- Met factorial(1) en attente...
- factorial(1) est appelé.
- n est 1. Cas de base atteint !
- Retourne 1.
- La pile se "dépile" :
- factorial(2) reçoit le 1 et retourne 2 * 1 = 2.
- factorial(3) reçoit le 2 et retourne 3 * 2 = 6.
- Résultat final : 6

```
int factorial(int n) {

    // Cas de base : 0! ou 1! valent 1
    if (n == 0 || n == 1) {
        return 1;
    }

    // Cas récursif : n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}
```

Types de Récursivité : Directe vs Indirecte

Récursivité Directe

- La fonction s'appelle elle-même directement. (Le plus courant).

```c
// Exemple : Compte à rebours
void countdown(int n) {
    if (n == 0) {
        printf("Blastoff!\n");
    } else {
        printf("%d\n", n);
        countdown(n - 1); // Appel direct
    }
}
```

Types de Récursivité : Directe vs Indirecte

Récursivité Indirecte (ou croisée)

- Une fonction A appelle une fonction B, qui (directement ou indirectement) appelle à nouveau la fonction A.

```
void functionA(int n) {
    if (n > 0) {
        functionB(n - 1); // A appelle B
    }
}

void functionB(int n) {
    if (n > 1) {
        functionA(n / 2); // B appelle A
    }
}
```

Types de Récursivité : Terminale vs Non-Terminale

Récursivité Terminale (Tail Recursion)

- L'appel récursif est la toute dernière opération effectuée par la fonction. Il n'y a aucun calcul après le retour de l'appel.
- Avantage : Peut être optimisé par les compilateurs pour éviter le "Stack Overflow".
    -

```
// Factoriel en version terminale
int factorial(int n, int result) {
    if (n == 0) {
        return result; // Cas de base
    } else {
        // L'appel est la DERNIÈRE chose faite
        return factorial(n - 1, result * n);
    }
}
// Appel initial : factorial(5, 1)
```

Types de Récursivité : Terminale vs Non-Terminale

Récursivité Non-Terminale

- Une opération est effectuée après le retour de l'appel récursif.

```
// Fibonacci est non-terminal
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        // L'ADDITION (+) se fait APRES les retours
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Types de Récursivité : Imbriquée

Récursivité Imbriquée (Nested Recursion)

- Un appel récursif est utilisé comme paramètre d'un autre appel récursif.
- Ces fonctions peuvent devenir extrêmement complexes et croître très rapidement.
- Exemple célèbre : la fonction d'Ackermann.

```
int ackermann(int m, int n) {
    if (m == 0) {
        return n + 1;
    } else if (n == 0) {
        return ackermann(m - 1, 1);
    } else {
        // L'appel interne est un paramètre de l'appel externe
        return ackermann(m - 1, ackermann(m, n - 1));
    }
}
```

Exercices d'application

Exercice 1 : Somme des Entiers

- Problème :
  - Écrivez une fonction récursive **int sum(int n)** qui calcule la somme de tous les entiers de 1 à N.
  - Exemple : **sum(4)** doit retourner 10 (car 4 + 3 + 2 + 1 = 10).
  - Indices :
    - Comment exprimer **sum(n)** en utilisant **sum(n-1)** ?
    - Quel est le cas de base le plus simple pour **sum(n)** ? (Que vaut **sum(1)** ou **sum(0)**) ?

Exercices d’application

Exercice 2 : Fonction Puissance

- Problème :
  - Écrivez une fonction récursive **double power(double base, int exp)** qui calcule $base^{exp}$ (base à la puissance exposant).
  - Exemple : **power(2, 3)** doit retourner 8 (car 2 x 2 x 2 = 8).
- Indices :
  - Comment exprimer $base^{exp}$ en utilisant $base^{exp-1}$?
  - Quel est le cas de base ? (Que se passe-t-il si l'exposant exp est 0 ?)