Domus Studiorum

# Introduction à l'Algorithmique

Version légère

Oskhane Boya Gueï
+225 05 65 62 32 17
31/12/2025

# Introduction : Bienvenue dans le monde des algorithmes

Imaginez que vous devez expliquer à quelqu'un qui ne sait pas cuisiner comment préparer une omelette. Vous allez naturellement décomposer la tâche en étapes simples : casser les œufs, les battre, chauffer la poêle, verser le mélange, etc. Félicitations, vous venez de créer un algorithme !

L'informatique repose sur ce même principe : décomposer des problèmes complexes en instructions simples et précises qu'un ordinateur peut exécuter.

# Chapitre 1 : Comprendre les algorithmes

## 1.1 Qu'est-ce qu'un algorithme ?

Le terme "algorithme" vient du nom du mathématicien persan Al-Khwarizmi (vers 780-850). Un algorithme est une séquence finie et ordonnée d'instructions non ambiguës permettant de résoudre un problème ou d'accomplir une tâche.

**Caractéristiques essentielles d'un algorithme :**

- Il doit être **fini** : se terminer après un nombre déterminé d'étapes
- Il doit être **précis** : chaque instruction doit être claire et sans ambiguïté
- Il doit avoir des **entrées** : données initiales (parfois aucune)
- Il doit produire des **sorties** : résultats (au moins un)
- Il doit être **efficace** : résoudre le problème dans un temps raisonnable

**Exemple concret :** Trouver le plus grand nombre parmi trois nombres

Entrées : Trois nombres A, B, C

Sortie : Le nombre maximum

1. Prendre trois nombres A, B et C
2. Comparer A et B, garder le plus grand
3. Comparer ce résultat avec C
4. Le nombre restant est le maximum

## 1.2 Qu'est-ce que l'algorithmique ?

L'algorithmique est la discipline qui étudie les algorithmes. Elle englobe leur conception, leur analyse, leur optimisation et leur mise en œuvre. C'est le cœur de l'informatique, car avant d'écrire du code, il faut d'abord concevoir la logique qui le sous-tend.

## 1.3 Pourquoi étudier l'algorithmique ?

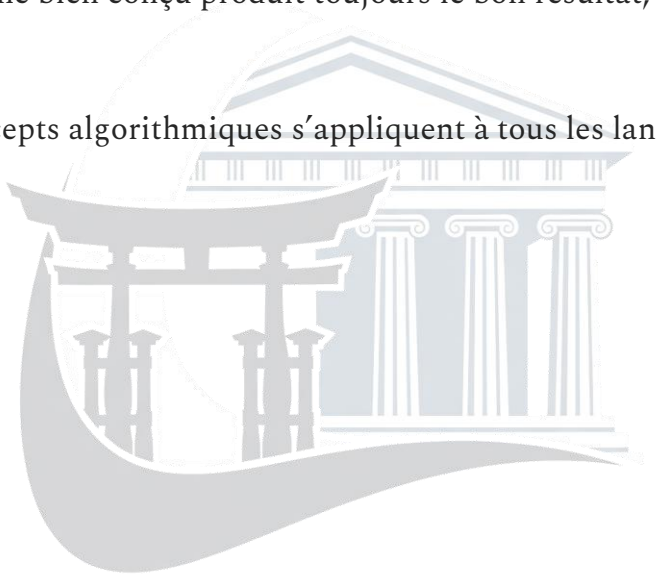
**Efficacité :** Un bon algorithme résout un problème rapidement, même avec de grandes quantités de données. Par exemple, rechercher un nom dans un annuaire de 10 000 personnes peut prendre 10 000 opérations avec un mauvais algorithme, mais seulement 14 avec un bon algorithme.

**Automatisation :** Les algorithmes permettent d'automatiser des tâches répétitives ou complexes, libérant du temps pour des activités plus créatives.

**Optimisation :** Ils aident à trouver les meilleures solutions possibles (plus rapides, moins coûteuses en mémoire).

**Fiabilité :** Un algorithme bien conçu produit toujours le bon résultat, éliminant les erreurs humaines.

**Universalité :** Les concepts algorithmiques s'appliquent à tous les langages de programmation.



# Chapitre 2 : Méthodes de raisonnement algorithmique

Avant de plonger dans la syntaxe, apprenons à penser comme un informaticien.

## 2.1 Le Rasoir d'Ockham

**Principe :** Entre plusieurs solutions, la plus simple est souvent la meilleure. Ne compliquez pas inutilement.

**Application pratique :** Votre programme plante ? Avant d'imaginer un bug complexe dans votre système, vérifiez d'abord les erreurs évidentes : fautes de frappe, fichiers manquants, syntaxe incorrecte.

**Exemple algorithmique :** Pour vérifier si un nombre est pair, au lieu de créer une boucle complexe qui divise par tous les nombres pairs possibles, utilisez simplement l'opération modulo : si nombre % 2 == 0, alors il est pair.

## 2.2 La décomposition en sous-problèmes (Diviser pour régner)

**Principe :** Face à un problème complexe, divisez-le en sous-problèmes plus simples, résolvez chacun séparément, puis combinez les solutions.

**Application pratique :** Pour créer un système de gestion de bibliothèque, décomposez en : gestion des livres, gestion des emprunts, gestion des utilisateurs, système de recherche. Traitez chaque module indépendamment.

**Exemple algorithmique :** Pour trier une liste de 1000 nombres, au lieu de tout faire d'un coup, divisez la liste en deux, triez chaque moitié, puis fusionnez-les (c'est le principe du tri fusion).

## 2.3 Penser en dehors des sentiers battus

**Principe :** N'hésitez pas à explorer des approches non conventionnelles.

**Application pratique :** Comment compter rapidement le nombre de personnes dans une salle bondée ? Au lieu de compter une par une, comptez les rangées et multipliez par le nombre de sièges par rangée.

## 2.4 L'approche pas à pas

**Principe :** Construisez votre solution progressivement, en testant chaque étape.

**Application pratique :** Ne codez pas tout d'un coup. Écrivez d'abord une version simple qui fonctionne, testez-la, puis ajoutez progressivement des fonctionnalités.

# Chapitre 3 : Méthodologie de conception d'un algorithme

## 3.1 Les six étapes de la conception

**Étape 1 : Comprendre le problème**

Avant tout, assurez-vous de bien comprendre ce qu'on vous demande. Posez-vous ces questions :

- Quelles sont les données d'entrée ?
- Quel résultat est attendu ?
- Y a-t-il des contraintes particulières ?

**Exemple :** "Calculer la moyenne d'un élève"

- Entrées : les notes de l'élève
- Sortie : un nombre représentant la moyenne
- Contraintes : les notes sont entre 0 et 20

**Étape 2 : Définir l'objectif clairement**

Formulez en une phrase précise ce que votre algorithme doit accomplir.

**Exemple :** "L'algorithme doit additionner toutes les notes de l'élève et diviser le résultat par le nombre de notes pour obtenir la moyenne."

**Étape 3 : Lister les étapes en français**

Écrivez la solution en langage naturel, étape par étape.

**Exemple :**

1. Demander le nombre de notes
2. Pour chaque note, la demander et l'ajouter à une somme
3. Diviser la somme par le nombre de notes
4. Afficher le résultat

**Étape 4 : Écrire le pseudo-code**

Traduisez votre description en pseudo-code (langage intermédiaire entre le français et le code).

**Étape 5 : Traduire en langage de programmation**

Transformez le pseudo-code en code réel (Python, Java, C, etc.).

**Étape 6 : Tester et optimiser**

Testez avec différents cas, y compris des cas limites (nombres négatifs, zéro, très grandes valeurs). Optimisez si nécessaire.

# Chapitre 4 : Structure générale d'un algorithme

Un algorithme bien écrit suit une structure standardisée en trois parties.

## 4.1 L'en-tête

Donne un nom significatif à l'algorithme et spécifie les entrées/sorties. Le nom doit refléter ce que fait l'algorithme.

```
Algorithme CalculerMoyenne
Entrées : nombre de notes, valeur de chaque note
Sortie : la moyenne des notes
```

## 4.2 La partie déclarative

Déclare tous les objets (variables, constantes) utilisés par l'algorithme. C'est comme préparer tous les ingrédients avant de cuisiner.

```
Variables :
    nombreNotes : entier
    somme : réel
    moyenne : réel
    note : réel
    i : entier

Constantes :
    NOTE_MAX : réel = 20.0
```

## 4.3 Le corps de l'algorithme

Contient les instructions à exécuter, délimitées par Début et Fin.

```
Début
    somme <- 0
    Écrire("Combien de notes ?")
    Lire(nombreNotes)

    Pour i <- 1 jusqu'à nombreNotes Faire
        Écrire("Note ", i, " : ")
        Lire(note)
        somme <- somme + note
    FinPour

    moyenne <- somme / nombreNotes
    Écrire("La moyenne est : ", moyenne)
Fin
```

## 4.4 Algorithme complet

```
Algorithme CalculerMoyenne
Entrées : nombre de notes, valeur de chaque note
Sortie : la moyenne des notes

Variables :
    nombreNotes : entier
    somme : réel
    moyenne : réel
    note : réel
    i : entier
```

```
Constantes :
    NOTE_MAX : réel = 20.0

Début
    somme <- 0
    Écrire("Combien de notes ?")
    Lire(nombreNotes)

    Pour i <- 1 jusqu'à nombreNotes Faire
        Écrire("Entrez la note ", i, " : ")
        Lire(note)
        somme <- somme + note
    FinPour

    moyenne <- somme / nombreNotes
    Écrire("La moyenne est : ", moyenne)
Fin
```

# Chapitre 5 : Les variables et les constantes

## 5.1 Comprendre les variables

Une variable est comme une boîte dans la mémoire de l'ordinateur où vous pouvez stocker une information. Cette information peut changer pendant l'exécution du programme.

**Analogie :** Imaginez un casier avec une étiquette (nom), un type de contenu autorisé (type), et ce qui s'y trouve actuellement (valeur).

**Caractéristiques d'une variable :**

- **Nom** : identificateur unique (ex : age, nomUtilisateur, compteur)
- **Type** : nature des données qu'elle peut contenir
- **Valeur** : contenu actuel (peut changer)

**Règles de nommage :**

- Commencer par une lettre
- Pas d'espaces (utilisez _ ou la casse camel : monAge)
- Éviter les caractères spéciaux
- Utiliser des noms descriptifs (prix plutôt que x)

**Syntaxe de déclaration :**

```
Variables :
    age : entier
    prix : réel
    nom : chaîne
    estValide : booléen
```

## 5.2 Comprendre les constantes

Une constante est une valeur qui ne change jamais pendant l'exécution du programme. On l'utilise pour des valeurs fixes comme PI, le nombre de jours dans une semaine, ou des limites définies.

**Avantages des constantes :**

- Rendent le code plus lisible
- Facilitent les modifications (changez la valeur à un seul endroit)
- Évitent les erreurs de modification accidentelle

**Syntaxe de déclaration :**

```
Constantes :
    PI : réel = 3.14159
    JOURS_SEMAINE : entier = 7
    TAUX_TVA : réel = 0.20
    MESSAGE_BIENVENUE : chaîne = "Bienvenue !"
```

**Convention :** On écrit généralement les noms de constantes en MAJUSCULES.

# Chapitre 6 : Les types de données

Les types de données définissent quelle sorte d'information peut être stockée dans une variable.

## 6.1 Les types de base

**Entier** (**integer**)

- Nombres sans virgule, positifs ou négatifs
- Exemples : -5, 0, 42, 1000
- Utilisations : compteurs, âges, quantités

**Réel** (**float/double**)

- Nombres avec virgule
- Exemples : 3.14, -0.5, 2.0
- Utilisations : prix, mesures, calculs scientifiques

**Booléen** (**boolean**)

- Deux valeurs possibles : Vrai ou Faux
- Utilisations : tests de conditions, drapeaux (flags)
- Exemples : estMajeur, aReussi, estConnecté

**Caractère** (**char**)

- Un seul caractère entre apostrophes
- Exemples : 'A', 'z', '5', '?'
- Utilisations : initiales, symboles

**Chaîne de caractères** (**string**)

- Suite de caractères entre guillemets
- Exemples : "Bonjour", "Paris", "2024"
- Utilisations : noms, phrases, textes

## 6.2 Opérations sur les types

**Opérateurs arithmétiques (pour entiers et réels) :**

| SYMBOLE | EXEMPLE |
|---|---|
| **+ : ADDITION** | 5 + 3 = 8 |
| **- : SOUSTRACTION** | 5 - 3 = 2 |
| *** : MULTIPLICATION** | 5 * 3 = 15 |
| **/ : DIVISION** | 5 / 2 = 2.5 |
| **% : MODULO (RESTE)** | 5 % 2 = 1 |
| **^ : PUISSANCE** | 5 ^ 2 = 25 |

**Opérateurs de comparaison (résultat booléen) :**

| OPÉRATEUR | EXEMPLE |
|---|---|
| **== : EGAL A** | 5 == 5 → Vrai |
| **!= : DIFFERENT DE** | 5 != 3 → Vrai |
| **< : INFERIEUR A** | 5 < 3 → Faux |
| **> : SUPERIEUR A** | 5 > 3 → Vrai |
| **<= : INFERIEUR OU EGAL** | 5 <= 5 → Vrai |
| **>= : SUPERIEUR OU EGAL** | 5 >= 3 → Vrai |

**Opérateurs logiques (pour booléens) :**

ET (AND) : Vrai si les deux sont vrais

OU (OR) : Vrai si au moins un est vrai

NON (NOT) : Inverse la valeur

**Exemples :**

$(5 > 3)$ ET $(2 < 4)$ → Vrai ET Vrai → Vrai

$(5 > 3)$ OU $(2 > 4)$ → Vrai OU Faux → Vrai

NON $(5 > 3)$ → NON Vrai → Faux

**Opérations sur les chaînes :**

Concaténation : "Bon" + "jour" = "Bonjour"

Longueur : longueur("Hello") = 5

## 6.3 Conversions de types

Parfois, vous devez convertir un type en un autre :

entierVersRéel(5) → 5.0

réelVersEntier(5.7) → 5

chaîneVersEntier("42") → 42

entierVersChaîne(42) → "42"

# Chapitre 7 : Les instructions de base

## 7.1 L'affectation

L'affectation stocke une valeur dans une variable. On utilise le symbole ← (ou = dans certains langages).

age ← 25

prix ← 19.99

nom ← "Alice"

estActif ← Vrai

**Attention :** L'affectation n'est pas une égalité mathématique !

x ← 5

x ← x + 1  // x vaut maintenant 6

Ici, on lit d'abord la valeur actuelle de x (5), on ajoute 1, puis on stocke le résultat (6) dans x.

## 7.2 L'entrée (lecture)

Pour récupérer une information fournie par l'utilisateur :

Lire(variable)

**Exemple :**

```
Écrire("Quel est votre âge ?")
Lire(age)
```

## 7.3 La sortie (écriture/affichage)

Pour afficher une information à l'utilisateur :

Écrire(expression)

**Exemples :**

```
Écrire("Bonjour !")
Écrire("Vous avez ", age, " ans")
Écrire(5 + 3)    // Affiche 8
```

## 7.4 Les commentaires

Les commentaires expliquent le code. Ils sont ignorés par l'ordinateur mais essentiels pour la lisibilité.

```
// Commentaire sur une ligne

/* Commentaire
   sur plusieurs
   lignes */
```

**Bonnes pratiques :**

- Expliquez-le "pourquoi", pas le "quoi" évident
- Commentez les parties complexes
- Mettez à jour les commentaires quand vous modifiez le code

**Exemple :**

```
// Mauvais commentaire
x ← x + 1  // Ajoute 1 à x

// Bon commentaire
x ← x + 1  // Passe au client suivant
```

# Chapitre 8 : Les structures alternatives (conditions)

Les structures alternatives permettent à l'algorithme de prendre des décisions en fonction de conditions.

## 8.1 La structure Si... FinSi

Exécute un bloc d'instructions seulement si une condition est vraie.

**Syntaxe :**

```
Si condition Alors
    instructions
FinSi
```

**Exemple :**

```
Lire(age)
Si age >= 18 Alors
    Écrire("Vous êtes majeur")
FinSi
```

**Diagramme de flux :**

Condition vraie ? → OUI → Exécuter instructions
↓ NON
Continuer

## 8.2 La structure Si... Sinon... FinSi

Propose deux chemins : un si la condition est vraie, un autre si elle est fausse.

**Syntaxe :**

```
Si condition Alors
    instructions1
Sinon
    instructions2
FinSi
```

**Exemple :**

```
Lire(note)
Si note >= 10 Alors
    Écrire("Admis")
Sinon
    Écrire("Refusé")
FinSi
```

## 8.3 La structure Si... SinonSi... Sinon... FinSi

Permet de tester plusieurs conditions successives.

**Syntaxe :**

```
Si condition1 Alors
    instructions1
SinonSi condition2 Alors
    instructions2
SinonSi condition3 Alors
    instructions3
Sinon
    instructions4
FinSi
```

**Exemple (système de notation) :**

```
Lire(note)
Si note >= 16 Alors
    Écrire("Très bien")
SinonSi note >= 14 Alors
    Écrire("Bien")
SinonSi note >= 12 Alors
    Écrire("Assez bien")
SinonSi note >= 10 Alors
    Écrire("Passable")
Sinon
    Écrire("Insuffisant")
FinSi
```

**Important :** Les conditions sont testées dans l'ordre. Dès qu'une est vraie, les autres ne sont pas évaluées.

## 8.4 La structure Selon... FinSelon

Utile quand on teste plusieurs valeurs possibles d'une même variable. Plus lisible que de multiples SinonSi.

**Syntaxe :**

```
Selon variable Faire
    Valeur1 :
        instructions1
    Valeur2 :
        instructions2
    Valeur3 :
```

```
        instructions3
    Sinon :
        instructions_par_défaut
FinSelon
```

**Exemple (menu) :**

```
Écrire("Choisissez : 1-Nouveau 2-Ouvrir 3-Sauvegarder 4-Quitter")
Lire(choix)

Selon choix Faire
    1 :
        Écrire("Création d'un nouveau fichier...")
    2 :
        Écrire("Ouverture d'un fichier...")
    3 :
        Écrire("Sauvegarde en cours...")
    4 :
        Écrire("Au revoir !")
    Sinon :
        Écrire("Choix invalide")
FinSelon
```

## 8.5 Conditions composées

Vous pouvez combiner plusieurs conditions avec ET, OU, NON.

**Exemples :**

```
// Vérifier qu'un âge est dans une fourchette
Si age >= 18 ET age <= 65 Alors
```

```
    Écrire("En âge de travailler")
FinSi

// Vérifier si c'est le week-end
Si jour == "Samedi" OU jour == "Dimanche" Alors
    Écrire("C'est le week-end !")
FinSi

// Vérifier qu'un nombre n'est pas nul
Si NON (nombre == 0) Alors
    résultat ← 100 / nombre
FinSi
```

# Chapitre 9 : Les structures répétitives (boucles)

Les boucles permettent de répéter un bloc d'instructions plusieurs fois, évitant ainsi de dupliquer du code.

### 9.1 La boucle TantQue

Répète des instructions tant qu'une condition reste vraie. La condition est vérifiée **avant** chaque itération.

**Syntaxe :**

```
TantQue condition Faire
    instructions
FinTantQue
```

**Fonctionnement :**

1. Vérifier la condition
2. Si vraie → exécuter les instructions, retour à l'étape 1
3. Si fausse → sortir de la boucle

**Exemple (compte à rebours) :**

```
compteur ← 10
TantQue compteur > 0 Faire
    Écrire(compteur)
    compteur ← compteur - 1
FinTantQue
Écrire("Décollage !")
```

**Attention : Boucle infinie !** Si la condition reste toujours vraie, la boucle ne s'arrêtera jamais :

```
// ERREUR - Boucle infinie
x ← 5
```

```
TantQue x > 0 Faire
    Écrire(x)
    // Oubli : x n'est jamais modifié !
FinTantQue
```

## 9.2 La boucle Pour

Utilisée quand on connaît à l'avance le nombre d'itérations. Idéale pour parcourir une plage de valeurs.

**Syntaxe :**

```
Pour compteur ← valeurInitiale Jusqu'à valeurFinale Faire
    instructions
FinPour
```

**Exemple (afficher les nombres de 1 à 10) :**

```
Pour i ← 1 Jusqu'à 10 Faire
    Écrire(i)
FinPour
```

**Exemple (table de multiplication) :**

```
Lire(nombre)
Pour i ← 1 Jusqu'à 10 Faire
    Écrire(nombre, " x ", i, " = ", nombre * i)
FinPour
```

**Variation : pas différent de 1**

Certaines syntaxes permettent de définir le pas (incrément) :

```
Pour i ← 0 Jusqu'à 20 Pas de 2 Faire
    Écrire(i)    // Affiche 0, 2, 4, 6, ..., 20
FinPour
```

## 9.3 La boucle Répéter... Jusqu'à

Exécute les instructions **au moins une fois**, puis vérifie la condition. Continue tant que la condition est **fausse**.

**Syntaxe :**

```
Répéter
    instructions
Jusqu'à condition
```

**Différence avec TantQue :**

- TantQue : peut ne jamais s'exécuter si la condition est fausse dès le début
- Répéter : s'exécute au moins une fois

**Exemple (validation d'une saisie) :**

```
Répéter
    Écrire("Entrez un nombre entre 1 et 10 :")
    Lire(nombre)
Jusqu'à nombre >= 1 ET nombre <= 10
Écrire("Merci !")
```

## 9.4 Comparaison des trois boucles

**TantQue :**

- Utiliser quand : vous ne savez pas combien d'itérations seront nécessaires

- Condition vérifiée : avant chaque itération
- Nombre d'exécutions : 0 ou plus

**Pour :**

- Utiliser quand : vous connaissez le nombre d'itérations
- Condition vérifiée : automatiquement via le compteur
- Nombre d'exécutions : nombre fixe

**Répéter :**

- Utiliser quand : vous voulez exécuter au moins une fois
- Condition vérifiée : après chaque itération
- Nombre d'exécutions : 1 ou plus

## 9.5 Boucles imbriquées

On peut placer une boucle à l'intérieur d'une autre. C'est utile pour traiter des structures à deux dimensions (tableaux, grilles).

**Exemple (table de multiplication complète) :**

```
Pour i ← 1 Jusqu'à 10 Faire
    Pour j ← 1 Jusqu'à 10 Faire
        Écrire(i, " x ", j, " = ", i * j)
    FinPour
FinPour
```

**Exemple (dessin d'un rectangle avec des étoiles) :**

```
largeur ← 5
hauteur ← 3

Pour ligne ← 1 Jusqu'à hauteur Faire
    Pour colonne ← 1 Jusqu'à largeur Faire
```

```
        Écrire("*")
    FinPour
    Écrire("")  // Saut de ligne
FinPour

// Affichage :
// *****
// *****
// *****
```

# Chapitre 10 : Les tableaux

## 10.1 Qu'est-ce qu'un tableau ?

Un tableau est une structure qui permet de stocker plusieurs valeurs du même type sous un seul nom. C'est comme une étagère avec plusieurs cases numérotées.

**Analogie :** Un immeuble avec des appartements numérotés. L'immeuble a un nom (le nom du tableau), et chaque appartement a un numéro (l'indice).

**Déclaration :**

```
Variables :
    notes : tableau[5] de réels
    // Tableau de 5 cases pour stocker des notes
```

**Indices :** Les cases du tableau sont numérotées à partir de 0 (ou 1 selon les conventions).

notes[0], notes[1], notes[2], notes[3], notes[4]

### 10.2 Manipulation des tableaux

**Initialisation :**

```
notes[0] ← 15.5
notes[1] ← 12.0
notes[2] ← 18.0
notes[3] ← 10.5
notes[4] ← 14.0
```

**Accès aux éléments :**

```
Écrire(notes[2])    // Affiche 18.0
moyenne ← (notes[0] + notes[1] + notes[2] + notes[3] + notes[4]) / 5
```

**Parcourir un tableau avec une boucle :**

```
Pour i ← 0 Jusqu'à 4 Faire
    Écrire("Note ", i+1, " : ", notes[i])
FinPour
```

## 10.3 Exemple complet : Recherche du maximum

```
Algorithme RechercheMaximum

Entrées : 10 nombres entiers
Sortie : le nombre maximum

Variables :
    nombres : tableau[10] de entiers
    i : entier
    maximum : entier

Début
    // Remplir le tableau
    Pour i ← 0 Jusqu'à 9 Faire
        Écrire("Entrez le nombre ", i+1, " :")
        Lire(nombres[i])
    FinPour

    // Chercher le maximum
    maximum ← nombres[0]
    Pour i ← 1 Jusqu'à 9 Faire
        Si nombres[i] > maximum Alors
            maximum ← nombres[i]
```

```
        FinSi
    FinPour

    Écrire("Le maximum est : ", maximum)
Fin
```

# Chapitre 11 : Les procédures et les fonctions

## 11.1 Pourquoi utiliser des procédures et fonctions ?

Imaginez que vous devez afficher un message de bienvenue à plusieurs endroits dans votre programme. Au lieu de réécrire le code à chaque fois, vous créez un sous-programme réutilisable.

**Avantages :**

- **Réutilisabilité :** Écrivez une fois, utilisez plusieurs fois
- **Lisibilité :** Code mieux organisé et plus facile à comprendre
- **Maintenance :** Modifiez à un seul endroit
- **Décomposition :** Divisez un problème complexe en sous-tâches

## 11.2 Les procédures

Une procédure est un bloc d'instructions nommé qui effectue une tâche mais ne retourne pas de valeur (ou retourne plusieurs valeurs via des paramètres).

**Syntaxe :**

```
Procédure NomProcédure(paramètres)
Variables locales :
    // Variables utilisées uniquement dans la procédure
Début
    instructions
Fin
```

**Exemple simple :**

```
Procédure AfficherBienvenue()
Début
   Écrire("========================")
   Écrire("   Bienvenue dans le programme !")
```

```
    Écrire("========================")
Fin

// Utilisation dans l'algorithme principal
Début
    AfficherBienvenue()
    // Suite du programme...
Fin
```

**Exemple avec paramètres :**

```
Procédure AfficherMessage(nom : chaîne, age : entier)
Début
    Écrire("Bonjour ", nom)
    Écrire("Vous avez ", age, " ans")
Fin

// Utilisation
Début
    AfficherMessage("Alice", 25)
    AfficherMessage("Bob", 30)
Fin
```

## 11.3 Les fonctions

Une fonction est similaire à une procédure, mais elle retourne **une seule valeur**. On utilise une fonction quand on a besoin d'un calcul ou d'une transformation.

**Syntaxe :**

```
Fonction NomFonction(paramètres) : TypeRetour
Variables locales :
```

```
    // Variables utilisées uniquement dans la fonction
Début
    instructions
    Retourner valeur
Fin
```

**Exemple : Fonction pour calculer le carré d'un nombre**

```
Fonction Carré(nombre : réel) : réel
Début
    Retourner nombre * nombre
Fin


// Utilisation
Début
    x ← 5
    résultat ← Carré(x)
    Écrire(résultat)  // Affiche 25
Fin
```

**Exemple : Fonction pour vérifier la parité**

```
Fonction EstPair(nombre : entier) : booléen
Début
    Si nombre % 2 == 0 Alors
        Retourner Vrai
    Sinon
        Retourner Faux
    FinSi
Fin
```

```
// Utilisation
Début
    Si EstPair(8) Alors
        Écrire("8 est pair")
    FinSi
Fin
```

## 11.4 Paramètres : passage par valeur vs passage par référence

**Passage par valeur (par défaut) :**

La fonction reçoit une copie de la valeur. Modifier le paramètre dans la fonction n'affecte pas la variable d'origine.

```
Fonction Doubler(x : entier) : entier
Début
    x ← x * 2
    Retourner x
Fin

Début
    a ← 5
    b ← Doubler(a)
    Écrire(a)  // Affiche 5 (inchangé)
    Écrire(b)  // Affiche 10
Fin
```

**Passage par référence :**

La fonction reçoit l'adresse de la variable. Les modifications affectent la variable d'origine. On utilise souvent le mot-clé Var ou Ref.

```
Procédure DoublerRef(Var x : entier)
Début
    x ← x * 2
Fin


Début
    a ← 5
    DoublerRef(a)
    Écrire(a)  // Affiche 10 (modifié)
Fin
```

## 11.5 Exemple complet : Programme modulaire

```
Algorithme GestionNotes


// Fonction pour calculer la moyenne
Fonction CalculerMoyenne(notes : tableau[5]
                         de réels) : réel
Variables :
    somme : réel
    i : entier
Début
    somme ← 0
    Pour i ← 0 Jusqu'à 4 Faire
        somme ← somme + notes[i]
    FinPour
    Retourner somme / 5
Fin
```

```
// Procédure pour saisir les notes
Procédure SaisirNotes(Var notes : tableau[5] de réels)
Variables :
    i : entier
Début
    Pour i ← 0 Jusqu'à 4 Faire
        Écrire("Note ", i+1, " : ")
        Lire(notes[i])
    FinPour
Fin

// Programme principal
Variables :
    mesNotes : tableau[5] de réels
    moyenne : réel

Début
    Écrire("Saisissez vos 5 notes")
    SaisirNotes(mesNotes)
    moyenne ← CalculerMoyenne(mesNotes)
    Écrire("Votre moyenne est : ", moyenne)
Fin
```

# Chapitre 12 : Introduction à la récursivité

## 12.1 Qu'est-ce que la récursivité ?

La récursivité est une technique où une fonction s'appelle elle-même pour résoudre un problème en le décomposant en problèmes plus petits et similaires.

**Analogie :** Imaginez que vous cherchez un livre dans une pile. Vous regardez le premier livre. S'il ne correspond pas, vous recommencez la même opération avec le reste de la pile (plus petite).

## 12.2 Structure d'une fonction récursive

Toute fonction récursive doit avoir :

1. **Un cas de base** : condition d'arrêt qui évite la récursion infinie
2. **Un cas récursif** : appel de la fonction avec un problème plus petit

**Syntaxe générale :**

```
Fonction Recursive(paramètre) : type
Début
    Si cas_de_base Alors
        Retourner solution_simple
    Sinon
        Retourner Recursive(problème_plus_petit)
    FinSi
Fin
```

## 12.3 Exemple : Calcul de la factorielle

La factorielle de n (notée n!) est définie comme :

- 0! = 1 (cas de base)
- n! = n × (n-1)! pour n > 0 (cas récursif)

**Version récursive :**

```
Fonction Factorielle(n : entier) : entier

Entrée : un entier positif n
Sortie : n!

Début
    // Cas de base
    Si n == 0 OU n == 1 Alors
        Retourner 1
    Sinon
        // Cas récursif
        Retourner n * Factorielle(n - 1)
    FinSi
Fin

// Utilisation
Début
    Écrire("5! = ", Factorielle(5))  // Affiche 120
Fin
```

**Déroulement pour Factorielle(5) :**

Factorielle(5)
= 5 * Factorielle(4)
= 5 * (4 * Factorielle(3))
= 5 * (4 * (3 * Factorielle(2)))

= 5 * (4 * (3 * (2 * Factorielle(1))))

= 5 * (4 * (3 * (2 * 1)))

= 5 * (4 * (3 * 2))

= 5 * (4 * 6)

= 5 * 24

= 120

### 12.4 Exemple : Somme des n premiers entiers

Calculer 1 + 2 + 3 + ... + n

**Version récursive :**

```
Fonction SommeEntiers(n : entier) : entier
Début
    // Cas de base
    Si n == 0 Alors
        Retourner 0
    Sinon
        // Cas récursif
        Retourner n + SommeEntiers(n - 1)
    FinSi
Fin
```

## 12.5 Récursivité vs Itération

**Avantages de la récursivité :**

- Code plus élégant et concis
- Plus naturel pour certains problèmes (arbres, fractales)

**Inconvénients :**

- Consomme plus de mémoire (pile d'appels)
- Peut être moins efficace que les boucles
- Risque de débordement de pile si trop d'appels

**Version itérative de la factorielle (pour comparaison) :**

```
Fonction FactorielleIterative(n : entier) : entier
Variables :
    résultat : entier
    i : entier
Début
    résultat ← 1
    Pour i ← 2 Jusqu'à n Faire
        résultat ← résultat * i
    FinPour
    Retourner résultat
Fin
```

**Quand utiliser la récursivité ?**

- Pour des problèmes naturellement récursifs (parcours d'arbres, tri fusion)
- Quand la clarté du code est prioritaire
- Quand la profondeur de récursion reste raisonnable

# Chapitre 13 : Complexité algorithmique (introduction intuitive)

## 13.1 Pourquoi s'intéresser à la complexité ?

Deux algorithmes peuvent résoudre le même problème mais avec des performances très différentes. Comprendre la complexité vous aide à choisir le meilleur algorithme, surtout quand les données sont volumineuses.

**Analogie :** Chercher un livre dans une bibliothèque. Vous pouvez :

- Méthode 1 : Regarder tous les livres un par un (lent)

- Méthode 2 : Utiliser le catalogue classé par ordre alphabétique (rapide)

## 13.2 Qu'est-ce que la complexité ?

La complexité mesure combien d'opérations un algorithme effectue en fonction de la taille des données (notée n).

**Types de complexité :**

- **Complexité temporelle** : combien de temps l'algorithme prend
- **Complexité spatiale** : combien de mémoire il utilise

## 13.3 Notation intuitive (sans formalisme mathématique)

On utilise des termes simples pour décrire comment le temps d'exécution évolue :

**Constante** : Le temps ne dépend pas de la taille des données

```
// Accéder à un élément d'un tableau
valeur ← tableau[5]  // Toujours instantané
```

**Linéaire** : Le temps est proportionnel à la taille

```
// Parcourir tout un tableau
Pour i ← 0 Jusqu'à n-1 Faire
    Écrire(tableau[i])
FinPour
// Si n double, le temps double aussi
```

**Quadratique** : Le temps est proportionnel au carré de la taille

```
// Boucles imbriquées
Pour i ← 0 Jusqu'à n-1 Faire
    Pour j ← 0 Jusqu'à n-1 Faire
        // traitement
```

```
    FinPour
FinPour
// Si n double, le temps est multiplié par 4
```

**Logarithmique** : Le temps croît très lentement

// Recherche dichotomique (voir exemple ci-dessous)
// Même si n double, on ajoute juste une étape

## 13.4 Exemple concret : Recherche dans un tableau

**Recherche linéaire (parcourir tout le tableau) :**

```
Fonction RechercheLinéaire(tableau : tableau de entiers, valeur : entier) : entier

Entrées : un tableau et une valeur à chercher
Sortie : indice de la valeur (-1 si absent)

Variables :
    i : entier

Début
    Pour i ← 0 Jusqu'à Longueur(tableau)-1
        Faire
        Si tableau[i] == valeur Alors
            Retourner i
        FinSi
    FinPour
    Retourner -1  // Non trouvé
Fin
```

Complexité : Linéaire
- Dans le pire cas, on parcourt tout le tableau
- 1000 éléments → jusqu'à 1000 comparaisons

**Recherche dichotomique (dans un tableau trié) :**

```
Fonction RechercheDichotomique(tableau : tableau  trié de entiers, : entier): entier


Entrées : un tableau TRIÉ et une valeur
Sortie : indice de la valeur (-1 si absent)


Variables :
    gauche, droite, milieu : entier


Début
    gauche ← 0
    droite ← Longueur(tableau) - 1


    TantQue gauche <= droite Faire
        milieu ← (gauche + droite) / 2


        Si tableau[milieu] == valeur Alors
            Retourner milieu
        SinonSi tableau[milieu] < valeur Alors
            gauche ← milieu + 1
        Sinon
            droite ← milieu - 1
        FinSi
    FinTantQue
```

```
    Retourner -1  // Non trouvé
Fin
```

Complexité : Logarithmique
- À chaque étape, on divise le problème par 2
- 1000 éléments → environ 10 comparaisons
- 1 000 000 éléments → environ 20 comparaisons !

## 13.5 Comparaison visuelle

Pour chercher dans 1 million d'éléments :

Recherche linéaire : jusqu'à 1 000 000 opérations
Recherche dichotomique : environ 20 opérations

Gain : 50 000 fois plus rapide !

## 13.6 Règles pratiques

**Pour améliorer la complexité :**

1. Évitez les boucles imbriquées inutiles
2. Utilisez des structures de données adaptées
3. Triez les données si vous devez chercher souvent
4. Sortez des boucles dès que possible

**Exemple d'optimisation :**

```
// Version lente (quadratique)
Pour i ← 0 Jusqu'à n-1 Faire
    Pour j ← 0 Jusqu'à n-1 Faire
        Si tableau[i] == tableau[j] ET i != j Alors
            Écrire("Doublon trouvé")
```

```
        FinSi
    FinPour
FinPour

// Version optimisée (on peut faire mieux avec un tri d'abord)
Trier(tableau)
Pour i ← 0 Jusqu'à n-2 Faire
    Si tableau[i] == tableau[i+1] Alors
        Écrire("Doublon trouvé")
    FinSi
FinPour
```

# Chapitre 14 : Pièges classiques en algorithmique

## 14.1 Variables non initialisées

**Erreur :**

```
Variables :
    somme : entier

Début
    Pour i ← 1 Jusqu'à 10 Faire
        somme ← somme + i  // somme n'a pas de valeur initiale !
    FinPour
Fin
```

**Solution :**

```
Variables :
    somme : entier

Début
    somme ← 0  // Initialisation indispensable
    Pour i ← 1 Jusqu'à 10 Faire
        somme ← somme + i
    FinPour
Fin
```

## 14.2 Indices hors limites

**Erreur :**

```
Variables :
    tableau : tableau[10] de entiers

Début
    Pour i ← 0 Jusqu'à 10 Faire  // Erreur : va de 0 à 10 (11 valeurs)
        Lire(tableau[i])
    FinPour
Fin
```

**Solution :**

```
Début
    Pour i ← 0 Jusqu'à 9 Faire  // Correct : indices 0 à 9
        Lire(tableau[i])
    FinPour
Fin
```

## 14.3 Boucles infinies

**Erreur :**

```
i ← 0
TantQue i < 10 Faire
    Écrire(i)
    // Oubli : i n'est jamais incrémenté !
FinTantQue
```

**Solution :**

```
i ← 0
TantQue i < 10 Faire
```

```
    Écrire(i)
    i ← i + 1  // Ne pas oublier de modifier la condition
FinTantQue
```

## 14.4 Confusion affectation / comparaison

**Erreur :**

```
Si x = 5 Alors  // Erreur : = au lieu de ==
    Écrire("x vaut 5")
FinSi
```

**Solution :**

```
Si x == 5 Alors  // Correct : == pour comparer
    Écrire("x vaut 5")
FinSi
```

## 14.5 Division par zéro

**Erreur :**

```
Lire(diviseur)
résultat ← 100 / diviseur  // Si diviseur vaut 0, erreur !
```

**Solution :**

```
Lire(diviseur)
Si diviseur != 0 Alors
    résultat ← 100 / diviseur
    Écrire("Résultat : ", résultat)
Sinon
```

```
    Écrire("Erreur : division par zéro impossible")
FinSi
```

## 14.6 Mauvais ordre des conditions

**Erreur :**

```
Lire(note)
Si note >= 10 Alors
    Écrire("Passable")
SinonSi note >= 12 Alors  // Ne sera jamais atteint !
    Écrire("Assez bien")
FinSi
```

**Solution :**

```
Lire(note)
Si note >= 16 Alors
    Écrire("Très bien")
SinonSi note >= 14 Alors
    Écrire("Bien")
SinonSi note >= 12 Alors
    Écrire("Assez bien")
SinonSi note >= 10 Alors
    Écrire("Passable")
Sinon
    Écrire("Insuffisant")
FinSi
```

## 14.7 Oubli de mise à jour d'un compteur

**Erreur :**

```
compteur ← 0
TantQue compteur < 5 Faire
    Écrire("Bonjour")
    // Oubli : compteur n'est jamais incrémenté
FinTantQue
```

**Solution :**

```
compteur ← 0
TantQue compteur < 5 Faire
    Écrire("Bonjour")
    compteur ← compteur + 1
FinTantQue
```

## 14.8 Modification d'un compteur de boucle Pour

**Erreur :**

```
Pour i ← 1 Jusqu'à 10 Faire
    Écrire(i)
    i ← i + 2  // Ne modifiez jamais le compteur manuellement !
FinPour
```

**Solution :**

```
// Si vous voulez un pas différent, utilisez la syntaxe appropriée
Pour i ← 1 Jusqu'à 10 Pas de 3 Faire
    Écrire(i)
FinPour
```

# Chapitre 15 : Exercices pratiques guidés

## Exercice 1 : Calculateur d'IMC (débutant)

**Objectif :** Calculer l'Indice de Masse Corporelle et indiquer la catégorie.

**Formule :** IMC = poids / (taille)$^2$

**Solution guidée :**

```
Algorithme CalculateurIMC

Entrées : poids (kg), taille (m)
Sortie : IMC et catégorie

Variables :
    poids : réel
    taille : réel
    imc : réel

Début
    Écrire("Entrez votre poids (kg) :")
    Lire(poids)
    Écrire("Entrez votre taille (m) :")
    Lire(taille)

    imc ← poids / (taille * taille)

    Écrire("Votre IMC est : ", imc)

    Si imc < 18.5 Alors
        Écrire("Insuffisance pondérale")
```

```
    SinonSi imc < 25 Alors
        Écrire("Corpulence normale")
    SinonSi imc < 30 Alors
        Écrire("Surpoids")
    Sinon
        Écrire("Obésité")
    FinSi
Fin
```

## Exercice 2 : Jeu du nombre mystère (intermédiaire)

**Objectif :** L'ordinateur choisit un nombre entre 1 et 100, l'utilisateur doit le deviner.

```
Algorithme NombreMystere

Entrée : propositions de l'utilisateur
Sortie : nombre de tentatives

Variables :
    nombreSecret : entier
    proposition : entier
    tentatives : entier

Début
    nombreSecret ← 42  // Simulation
    tentatives ← 0

    Écrire("Devinez le nombre entre 1 et 100")
```

```
    Répéter
        Écrire("Votre proposition :")
        Lire(proposition)
        tentatives ← tentatives + 1

        Si proposition < nombreSecret Alors
            Écrire("C'est plus !")
        SinonSi proposition > nombreSecret
                Alors
            Écrire("C'est moins !")
        FinSi
    Jusqu'à proposition == nombreSecret

    Écrire("Bravo ! Trouvé en ", tentatives, " tentatives")
Fin
```

## Exercice 3 : Tri d'un tableau (avancé)

**Objectif :** Trier un tableau de nombres par ordre croissant (tri par sélection).

```
Algorithme TriSelection

Entrées : 10 nombres entiers
Sortie : tableau trié par ordre croissant

Variables :
    tableau : tableau[10] de entiers
    i, j, indexMin, temp : entiers

Début
```

```
// Remplissage du tableau
Pour i ← 0 Jusqu'à 9 Faire
    Écrire("Nombre ", i+1, " :")
    Lire(tableau[i])
FinPour

// Tri par sélection
Pour i ← 0 Jusqu'à 8 Faire
    // Trouver le minimum dans la partie
    // non triée
    indexMin ← i
    Pour j ← i+1 Jusqu'à 9 Faire
        Si tableau[j] < tableau[indexMin]
           Alors
            indexMin ← j
        FinSi
    FinPour

    // Échanger tableau[i] et
    // tableau[indexMin]
    temp ← tableau[i]
    tableau[i] ← tableau[indexMin]
    tableau[indexMin] ← temp
FinPour

// Affichage du tableau trié
Écrire("Tableau trié :")
Pour i ← 0 Jusqu'à 9 Faire
```

```
        Écrire(tableau[i])
    FinPour
Fin
```

## Exercice 4 : Palindrome (intermédiaire)

**Objectif :** Vérifier si un mot est un palindrome (se lit de la même façon dans les deux sens).

```
Algorithme VerifierPalindrome

Entrée : une chaîne de caractères
Sortie : Vrai si palindrome, Faux sinon

Variables :
    mot : chaîne
    debut, fin : entier
    estPalindrome : booléen

Début
    Écrire("Entrez un mot :")
    Lire(mot)

    debut ← 0
    fin ← Longueur(mot) - 1
    estPalindrome ← Vrai

    TantQue debut < fin ET estPalindrome Faire
        Si mot[debut] != mot[fin] Alors
            estPalindrome ← Faux
```

```
        FinSi
        debut ← debut + 1
        fin ← fin - 1
    FinTantQue

    Si estPalindrome Alors
        Écrire(mot, " est un palindrome")
    Sinon
        Écrire(mot, " n'est pas un palindrome")
    FinSi
Fin
```

# Chapitre 16 : Du pseudo-code au langage de programmation

## 16.1 Comprendre la différence

Le **pseudo-code** (ou algorithme) que vous avez appris dans ce cours est un langage **intermédiaire** entre le français et le code informatique réel. C'est un outil de conception et de communication.

**Caractéristiques du pseudo-code :**

- Langage universel compris par tous les informaticiens
- Syntaxe flexible et proche du langage naturel
- Indépendant de tout langage de programmation
- Permet de se concentrer sur la logique sans détails techniques

**Les langages de programmation** (Python, Java, C, JavaScript, etc.) sont des langages **formels** avec une syntaxe stricte que l'ordinateur peut exécuter.

**Pourquoi deux étapes ?**

- Concevoir d'abord l'algorithme vous force à réfléchir à la logique
- Un bon algorithme peut être traduit dans n'importe quel langage
- Vous évitez de vous perdre dans les détails syntaxiques au début

## 16.2 Exemple de traduction : Pseudo-code → Python

**Pseudo-code (ce que vous avez appris) :**

```
Algorithme CalculerMoyenne

Variables :
    note1, note2, note3 : réel
    moyenne : réel
```

```
Début
    Écrire("Entrez la note 1 :")
    Lire(note1)
    Écrire("Entrez la note 2 :")
    Lire(note2)
    Écrire("Entrez la note 3 :")
    Lire(note3)

    moyenne ← (note1 + note2 + note3) / 3

    Écrire("La moyenne est :", moyenne)
Fin
```

**Python (langage de programmation réel) :**

```
# Pas besoin de déclarer les variables en Python

note1 = float(input("Entrez la note 1 : "))
note2 = float(input("Entrez la note 2 : "))
note3 = float(input("Entrez la note 3 : "))

moyenne = (note1 + note2 + note3) / 3

print("La moyenne est :", moyenne)
```

**Observations :**

- Écrire() devient print() en Python
- Lire() devient input() avec conversion float()
- ← devient = pour l'affectation
- Python n'exige pas de déclaration de variables

- Python utilise l'indentation pour délimiter les blocs (pas de Début/Fin)

## 16.3 Exemple avec une structure conditionnelle

**Pseudo-code :**

```
Si age >= 18 Alors
    Écrire("Majeur")
Sinon
    Écrire("Mineur")
FinSi
```

**Python :**

```
if age >= 18:
    print("Majeur")
else:
    print("Mineur")
```

**Java :**

```
if (age >= 18) {
    System.out.println("Majeur");
} else {
    System.out.println("Mineur");
}
```

**C :**

```
if (age >= 18) {
    printf("Majeur\n");
} else {
```

```
    printf("Mineur\n");
}
```

## 16.4 Exemple avec une boucle

**Pseudo-code :**

```
Pour i ← 1 Jusqu'à 5 Faire
    Écrire(i)
FinPour
```

**Python :**

```
for i in range(1, 6):  # range(1, 6) génère 1,2,3,4,5
    print(i)
```

**Java :**

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i);
}
```

## 16.5 Tableau de correspondances

| PSEUDO-CODE | PYTHON | JAVA/C |
|---|---|---|
| Écrire(x) | print(x) | System.out.println(x) / printf() |
| Lire(x) | x = input() | Scanner.nextInt() / scanf() |
| ← | = | = |
| == | == | == |
| ET | and | && |
| OU | or | ` |

| | | |
|---|---|---|
| NON | not | ! |
| Si…Alors | if: | if (…) { |
| Pour…Faire | for … in range(): | for (…) { |
| TantQue…Faire | while: | while (…) { |
| Fonction | def nom(): | type nom() { |

## 16.6 Les étapes de votre parcours

1. APPRENDRE L'ALGORITHMIQUE (ce cours)

↓

Vous maîtrisez : la logique, les structures, la décomposition

2. CHOISIR UN LANGAGE DE PROGRAMMATION

↓

Recommandés pour débuter : Python (simple), JavaScript (web), Java (structuré)

3. APPRENDRE LA SYNTAXE DU LANGAGE

↓

Traduire vos algorithmes dans ce langage

4. PRATIQUER, PRATIQUER, PRATIQUER

↓

Résoudre des problèmes, créer des projets

**Le message clé :** Ce que vous avez appris dans ce cours (logique algorithmique, structures de contrôle, décomposition) est **universel**. Que vous choisissiez Python, Java, C, JavaScript ou tout autre langage, vous utiliserez exactement les mêmes concepts. Seule la syntaxe change !

# Chapitre 17 : Exercices de raisonnement pur

Ces exercices développent votre capacité à analyser des algorithmes sans avoir à en écrire. C'est une compétence essentielle pour les examens et pour comprendre du code existant.

## 17.1 Prédire la sortie d'un algorithme

**Exercice 1 :**

```
Variables :
    x, y : entier


Début
    x ← 5
    y ← 3
    x ← x + y
    y ← x - y
    x ← x - y
    Écrire("x = ", x, " y = ", y)
Fin
```

**Question :** Que va afficher ce programme ?

**Réponse :** x = 3 y = 5

**Explication :** C'est une technique classique pour échanger deux variables sans variable temporaire.

- Après x ← x + y : x=8, y=3
- Après y ← x - y : x=8, y=5
- Après x ← x - y : x=3, y=5

**Exercice 2 :**

```
Variables :
    i, somme : entier

Début
    somme ← 0
    Pour i ← 1 Jusqu'à 4 Faire
        Si i % 2 == 0 Alors
            somme ← somme + i
        FinSi
    FinPour
    Écrire(somme)
Fin
```

**Question :** Quelle valeur sera affichée ?

**Réponse :** 6

**Explication :** La boucle parcourt 1, 2, 3, 4. Seuls 2 et 4 sont pairs, donc somme = 2 + 4 = 6.

**Exercice 3 :**

```
Variables :
    n, résultat : entier

Début
    n ← 5
    résultat ← 1
    TantQue n > 0 Faire
        résultat ← résultat * n
        n ← n - 1
```

```
    FinTantQue
    Écrire(résultat)
Fin
```

**Question :** Que calcule cet algorithme ? Quelle sera la valeur affichée ?

**Réponse :** Il calcule la factorielle de 5. Valeur affichée : 120

**Explication :**

- résultat = 1 × 5 = 5
- résultat = 5 × 4 = 20
- résultat = 20 × 3 = 60
- résultat = 60 × 2 = 120
- résultat = 120 × 1 = 120

## 17.2 Repérer les erreurs logiques

**Exercice 4 :**

```
Variables :
    tableau : tableau[5] de entiers
    i, max : entier

Début
    // Remplissage
    Pour i ← 0 Jusqu'à 4 Faire
        Lire(tableau[i])
    FinPour

    // Recherche du maximum
    max ← 0
```

```
    Pour i ← 0 Jusqu'à 4 Faire
        Si tableau[i] > max Alors
            max ← tableau[i]
        FinSi
    FinPour

    Écrire("Maximum :", max)
Fin
```

**Question :** Quelle est l'erreur logique dans cet algorithme ?

**Réponse :** max est initialisé à 0. Si tous les nombres du tableau sont négatifs, le maximum sera incorrectement affiché comme 0.

**Correction :** Initialiser max ← tableau[0] au lieu de max ← 0.

**Exercice 5 :**

```
Variables :
    n : entier

Début
    Écrire("Entrez un nombre positif :")
    Lire(n)

    Si n >= 0 Alors
        Écrire("Nombre positif")
    Sinon
        Écrire("Nombre négatif")
    FinSi
Fin
```

**Question :** Quel problème présente cet algorithme ?

**Réponse :** Le cas où n = 0 est traité comme "positif", mais zéro n'est ni positif ni négatif. Le message est trompeur.

**Correction :** Ajouter un cas spécifique pour zéro ou reformuler les messages.

**Exercice 6 :**

```
Variables :
    i, compteur : entier


Début
    compteur ← 0
    Pour i ← 1 Jusqu'à 10 Faire
        Si i % 2 == 0 Alors
            compteur ← compteur + 1
        Sinon
            compteur ← 0
        FinSi
    FinPour
    Écrire("Nombres pairs :", compteur)
Fin
```

**Question :** Quelle erreur logique contient cet algorithme ?

**Réponse :** À chaque nombre impair, le compteur est réinitialisé à 0 au lieu de simplement ne pas l'incrémenter. Résultat final : 1 (car 10 est le dernier nombre pair).

**Correction :** Supprimer le Sinon compteur ← 0.

## 17.3 Comparer deux algorithmes

**Exercice 7 :**

**Algorithme A :**

```
Fonction RechercheA(tableau : tableau de entiers, valeur : entier) : booléen
Variables :
    i : entier
Début
    Pour i ← 0 Jusqu'à Longueur(tableau)-1 Faire
        Si tableau[i] == valeur Alors
            Retourner Vrai
        FinSi
    FinPour
    Retourner Faux
Fin
```

**Algorithme B :**

```
Fonction RechercheB(tableau : tableau de entiers, valeur : entier) : booléen
Variables :
    i : entier
    trouvé : booléen
Début
    trouvé ← Faux
    Pour i ← 0 Jusqu'à Longueur(tableau)-1 Faire
        Si tableau[i] == valeur Alors
            trouvé ← Vrai
        FinSi
    FinPour
```

```
    Retourner trouvé
Fin
```

**Question :** Les deux algorithmes produisent-ils le même résultat ? Lequel est plus efficace et pourquoi ?

**Réponse :**

- Les deux produisent le même résultat final.
- **Algorithme A est plus efficace** : il s'arrête dès qu'il trouve la valeur (retour immédiat).
- Algorithme B parcourt toujours tout le tableau, même après avoir trouvé la valeur.

**Exercice 8 :**

**Algorithme A (tri par sélection) :**

Complexité : Quadratique
Parcourt le tableau n fois, à chaque fois parcourt les éléments restants.
Pour n=1000 : environ 500 000 comparaisons

**Algorithme B (tri fusion) :**

Complexité : n × log(n)
Divise récursivement le tableau et fusionne.
Pour n=1000 : environ 10 000 comparaisons

**Question :** Pour trier 1 million d'éléments, quel algorithme choisiriez-vous ?

**Réponse : Algorithme B (tri fusion).**

- Tri par sélection : ~500 milliards de comparaisons
- Tri fusion : ~20 millions de comparaisons
- Gain : 25 000 fois plus rapide !

### 17.4 Analyses de traces d'exécution

**Exercice 9 :**

```
Variables :
    a, b, c : entier

Début
    a ← 2
    b ← 3
    c ← a
    a ← b
    b ← c
    Écrire(a, b, c)
Fin
```

**Question :** Complétez le tableau de trace d'exécution :

| Ligne | Instruction | a | b | c |
|---|---|---|---|---|
| 1 | a ← 2 | 2 | ? | ? |
| 2 | b ← 3 | 2 | 3 | ? |
| 3 | c ← a | 2 | 3 | 2 |
| 4 | a ← b | 3 | 3 | 2 |
| 5 | b ← c | 3 | 2 | 2 |

**Sortie :** 3 2 2

**Exercice 10 :**

```
Variables :
    i, n, somme : entier
```

```
Début
    n ← 4
    somme ← 0
    Pour i ← 1 Jusqu'à n Faire
        somme ← somme + i
    FinPour
    Écrire(somme)
Fin
```

**Question :** Tracez l'évolution de i et somme :

| Itération | i | somme |
|---|---|---|
| Avant boucle | - | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 3 |
| 3 | 3 | 6 |
| 4 | 4 | 10 |

**Sortie :** 10

## 17.5 Questions de compréhension conceptuelle

### 17.5.1 Questions

**Exercice 11 :** Quel est l'avantage principal d'une fonction par rapport à du code répété plusieurs fois ?

**Exercice 12 :** Pourquoi la recherche dichotomique nécessite-t-elle un tableau trié ?

**Exercice 13 :** Quand faut-il préférer une boucle TantQue à une boucle Pour ?

### 17.5.2 Réponses

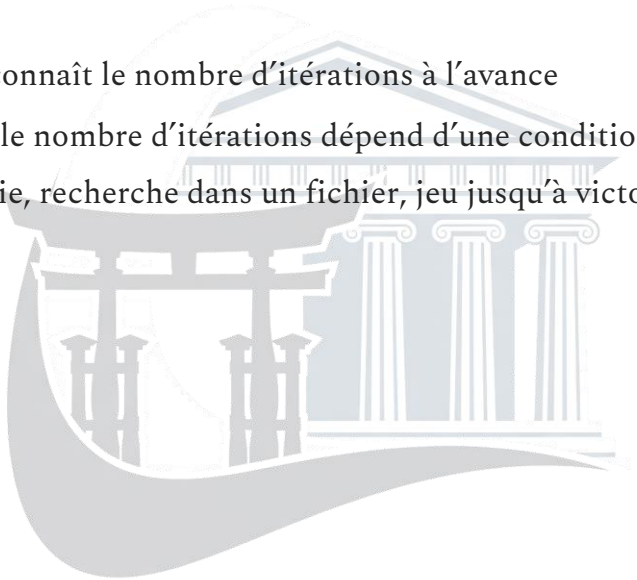
**Exercice 11 :**

- Réutilisabilité (écrit une fois, utilisé partout)
- Maintenance facilitée (modifié à un seul endroit)
- Lisibilité améliorée (nom explicite de la fonction)
- Réduction des erreurs (moins de code dupliqué)

**Exercice 12 :** La recherche dichotomique fonctionne en divisant l'espace de recherche par deux à chaque étape, en comparant avec l'élément du milieu. Si le tableau n'est pas trié, on ne peut pas savoir dans quelle moitié chercher.

**Exercice 13 :**

- Pour : quand on connaît le nombre d'itérations à l'avance
- TantQue : quand le nombre d'itérations dépend d'une condition qui peut varier (ex : validation de saisie, recherche dans un fichier, jeu jusqu'à victoire)



# Chapitre 18 : Conseils et bonnes pratiques

## 18.1 Nommage

- **Soyez descriptif :** age plutôt que a, nombreEtudiants plutôt que n
- **Conventions :** camelCase (monAge) ou snake_case (mon_age)
- **Constantes en majuscules :** PI, TAUX_TVA
- **Évitez les abréviations obscures :** nbEtud peut être ambigu

## 18.2 Indentation et lisibilité

L'indentation montre la structure du code. Chaque niveau de profondeur est décalé.

**Bon :**

```
Pour i ← 1 Jusqu'à 10 Faire
    Si i % 2 == 0 Alors
        Écrire(i)
    FinSi
FinPour
```

**Mauvais :**

```
Pour i ← 1 Jusqu'à 10 Faire
Si i % 2 == 0 Alors
Écrire(i)
FinSi
FinPour
```

## 18.3 Commentaires efficaces

- Expliquez le "pourquoi", pas le "comment" évident
- Commentez les parties complexes

- Gardez les commentaires à jour

## 18.4 Gestion des erreurs

Anticipez les problèmes : division par zéro, indices hors limites, saisies invalides.

```
Lire(diviseur)
Si diviseur != 0 Alors
    résultat ← 100 / diviseur
Sinon
    Écrire("Erreur : division par zéro impossible")
FinSi
```

## 18.5 Tests

Testez votre algorithme avec :

- **Cas normaux :** valeurs typiques
- **Cas limites :** valeurs minimales/maximales
- **Cas d'erreur :** valeurs invalids

## 18.6 Optimisation

Après avoir un algorithme qui fonctionne, cherchez à l'améliorer :

- Éliminer les calculs redondants
- Sortir des boucles dès que possible
- Choisir les bonnes structures de données

**Exemple d'optimisation :**

```
// Avant : calcule la longueur à chaque itération
Pour i ← 0 Jusqu'à Longueur(tableau) - 1 Faire
```

```
    // traitement
FinPour


// Après : calcule la longueur une seule fois
n ← Longueur(tableau)
Pour i ← 0 Jusqu'à n - 1 Faire
    // traitement
FinPour
```

# Chapitre 19 : Carte mentale de l'algorithmique

```
ALGORITHMIQUE
 |
 ├─── FONDAMENTAUX
 |    ├─── Variables (stockage modifiable)
 |    ├─── Constantes (stockage fixe)
 |    ├─── Types de données
 |    |    ├─── Entier
 |    |    ├─── Réel
 |    |    ├─── Booléen
 |    |    ├─── Caractère
 |    |    └─── Chaîne
 |    └─── Opérateurs
 |         ├─── Arithmétiques (+, -, *, /, %, ^)
 |         ├─── Comparaison (==, !=, <, >, <=, >=)
 |         └─── Logiques (ET, OU, NON)
 |
 ├─── INSTRUCTIONS DE BASE
 |    ├─── Affectation (←)
 |    ├─── Entrée (Lire)
 |    ├─── Sortie (Écrire)
 |    └─── Commentaires (//)
 |
 ├─── STRUCTURES DE CONTRÔLE
 |    ├─── Alternatives (Choix)
 |    |    ├─── Si... FinSi
 |    |    ├─── Si... Sinon... FinSi
 |    |    ├─── Si... SinonSi... FinSi
 |    |    └─── Selon... FinSelon
```

```
│   │
│   └──── Répétitives (Boucles)
│       ├──── TantQue… FinTantQue
│       ├──── Pour… FinPour
│       └──── Répéter… Jusqu'à
│
├──── STRUCTURES DE DONNÉES
│   ├──── Tableaux (collections indexées)
│   ├──── Listes (collections dynamiques)
│   ├──── Piles (LIFO)
│   └──── Files (FIFO)
│
├──── MODULARITÉ
│   ├──── Procédures (pas de retour)
│   ├──── Fonctions (retour une valeur)
│   ├──── Paramètres
│   │   ├──── Par valeur
│   │   └──── Par référence
│   └──── Variables locales/globales
│
├──── TECHNIQUES AVANCÉES
│   ├──── Récursivité
│   ├──── Tri (sélection, insertion, fusion)
│   └──── Recherche (linéaire, dichotomique)
│
└──── QUALITÉ
    ├──── Complexité (temps/espace)
    ├──── Tests (normaux, limites, erreurs)
    ├──── Optimisation
    └──── Bonnes pratiques
```

```
├── Nommage clair
├── Indentation
├── Commentaires
└── Gestion d'erreurs
```

# Glossaire algorithmique

**Affectation :** Opération qui attribue une valeur à une variable (symbole ←).

**Algorithme :** Séquence finie d'instructions précises permettant de résoudre un problème.

**Boucle** (**ou itération**) **:** Structure répétitive qui exécute un bloc d'instructions plusieurs fois (Pour, TantQue, Répéter).

**Booléen :** Type de données qui ne peut prendre que deux valeurs : Vrai ou Faux.

**Cas de base :** Dans une fonction récursive, condition d'arrêt qui empêche les appels infinis.

**Complexité :** Mesure du nombre d'opérations effectuées par un algorithme en fonction de la taille des données.

- **Complexité temporelle :** temps d'exécution
- **Complexité spatiale :** mémoire utilisée

**Condition :** Expression booléenne (Vrai ou Faux) testée dans une structure alternative ou répétitive.

**Constante :** Valeur nommée qui ne change jamais pendant l'exécution de l'algorithme.

**Déclaration :** Action de définir une variable ou constante en spécifiant son nom et son type.

**Fonction :** Sous-programme qui effectue un calcul et retourne une seule valeur.

**Incrémentation :** Action d'augmenter la valeur d'une variable (souvent de 1).

**Indice :** Numéro qui identifie la position d'un élément dans un tableau (commence généralement à 0).

**Initialisation :** Première affectation d'une valeur à une variable.

**Itération :** Une exécution du corps d'une boucle. Synonyme : passage.

**Langage de programmation :** Langage formel (Python, Java, C, etc.) permettant d'écrire des programmes exécutables par un ordinateur.

**Modulo** (%) **:** Opérateur qui renvoie le reste de la division entière (ex : 7 % 3 = 1).

**Opérateur :** Symbole représentant une opération (arithmétique, logique, de comparaison).

**Paramètre :** Variable définie dans la signature d'une fonction/procédure, qui reçoit une valeur lors de l'appel.

- **Par valeur :** copie de la valeur
- **Par référence :** adresse de la variable

**Pile d'appels :** Structure mémoire qui stocke les appels de fonctions en cours (important pour la récursivité).

**Procédure :** Sous-programme qui effectue une tâche sans retourner de valeur (ou retournant plusieurs valeurs via paramètres).

**Pseudo-code :** Représentation simplifiée d'un algorithme, à mi-chemin entre le langage naturel et le code informatique.

**Récursivité :** Technique où une fonction s'appelle elle-même pour résoudre un problème en le décomposant.

**Structure alternative** (**ou conditionnelle**) **:** Construction qui permet d'exécuter différentes instructions selon qu'une condition est vraie ou fausse (Si...Alors...Sinon).

**Structure répétitive :** Voir Boucle.

**Tableau :** Structure de données permettant de stocker plusieurs valeurs du même type, accessibles par un indice.

**Type de données :** Catégorie définissant la nature des valeurs qu'une variable peut contenir (entier, réel, chaîne, booléen, caractère).

**Variable :** Espace mémoire nommé qui peut stocker une valeur modifiable pendant l'exécution du programme.

## Conclusion

Vous avez maintenant les fondations solides de l'algorithmique ! Vous maîtrisez les concepts essentiels : variables, structures de contrôle, boucles, tableaux, modularité avec les fonctions et procédures, récursivité et complexité. Vous savez aussi comment passer du pseudo-code à un langage de programmation réel.

**Les prochaines étapes dans votre apprentissage :**

- **Choisir un langage de programmation** : Python (recommandé pour débuter), Java, JavaScript, C
- **Pratiquer régulièrement** : Sites comme France-IOI, CodinGame, LeetCode, HackerRank
- **Algorithmique avancée** : structures de données complexes (listes chaînées, arbres, graphes)
- **Analyse de complexité approfondie** : notations O, Ω, Θ
- **Algorithmes de tri avancés** : tri rapide (quicksort), tri par tas (heapsort)
- **Algorithmes de recherche et parcours** : parcours en profondeur, en largeur
- **Programmation orientée objet**
- **Projets concrets** : créez vos propres applications

**Rappelez-vous :**

- La pratique est essentielle : codez régulièrement
- Commencez simple, puis complexifiez progressivement
- Décomposez les problèmes en sous-problèmes
- N'ayez pas peur de l'erreur : elle fait partie de l'apprentissage
- Testez toujours vos algorithmes avec différents cas
- Cherchez à comprendre plutôt qu'à mémoriser
- Lisez le code des autres pour progresser
- Participez à des communautés de programmeurs

**Un dernier conseil :** L'algorithmique est comme un sport. Vous ne deviendrez pas expert en lisant simplement ce cours. Vous devez pratiquer, faire des exercices, résoudre des

problèmes, vous tromper, recommencer. C'est dans cette répétition que vous développerez l'intuition algorithmique qui fait la différence.

**Le véritable apprentissage commence maintenant !** Prenez un problème simple, écrivez l'algorithme en pseudo-code, traduisez-le dans votre langage de programmation préféré, testez-le, corrigez-le, optimisez-le. Répétez ce processus encore et encore, avec des problèmes de plus en plus complexes.

Bonne continuation dans votre voyage dans le monde passionnant de l'informatique !

*Fin du cours.*

## Annexe : Résumé des notations utilisées

| SYMBOLE | UTILITE |
|---|---|
| ← | Affectation |
| == | Égalité (comparaison) |
| != | Différent de |
| <, >, <=, >= | Comparaisons |
| **ET, OU, NON** | Opérateurs logiques |
| // | Commentaire |
| [ ] | Indices de tableau |
| % | Modulo (reste de division) |
| ^ | Puissance |

**Structure générale d'un algorithme :**

```
Algorithme NomAlgorithme
Entrées : ...
Sortie : ...

Variables :
    ...

Constantes :
    ...

Début
    instructions
Fin
```

**Structures de contrôle :**

```
Si condition Alors
    instructions
[SinonSi condition2 Alors
    instructions2]
[Sinon
    instructions3]
FinSi

Pour compteur ← début Jusqu'à fin Faire
    instructions
FinPour

TantQue condition Faire
    instructions
FinTantQue

Répéter
    instructions
Jusqu'à condition

Fonction NomFonction(paramètres) : type
    instructions
    Retourner valeur
Fin

Procédure NomProcédure(paramètres)
    instructions
Fin
```

EOF.