

Rappels sur les fonctions en Python

Fiche de synthèse

① DÉFINIR ET APPELER UNE FONCTION

Définition 1 (Fonction) Une **fonction** est un bloc d'instructions nommé, défini avec le mot-clé `def`, que l'on exécute en l'appelant par son nom suivi de parenthèses. Elle peut recevoir des paramètres et renvoyer une valeur avec l'instruction `return`.

```
def carre(x):
    return x * x

print(carre(5)) # affiche 25
```

Une valeur **renvoyée** par `return` peut être stockée dans une variable ou réutilisée dans un calcul; un **affichage** par `print` ne fait que l'écrire à l'écran. Une fonction qui se termine sans `return` renvoie la valeur spéciale `None`, qui signifie « aucune valeur ».

② PORTÉE DES VARIABLES

Définition 2 (Variable locale) Une variable affectée dans le corps d'une fonction est **locale** : elle n'existe que pendant l'exécution de la fonction et n'est pas visible en dehors.

```
x = 10
y = 10

def une_fonction():
    x = 5 # x locale, x global intact
    global y
    y = 5 # réaffecte le y global

une_fonction()
print(x) # affiche 10
print(y) # affiche 5
```

Seul le mot-clé `global` permet de réaffecter une variable globale; on l'évite autant que possible. En revanche, une liste ou un dictionnaire global peut être modifié en place sans `global`, par exemple avec `L.append(3)` ou `d[cle] = valeur`.

③ PARAMÈTRES

```
def presentation(prenom, age=15, ville="Lyon"):
    print(f"{prenom}, {age} ans, {ville}")

presentation("Gaudric") # Gaudric, 15 ans, Lyon
presentation("Justine", 21, "Paris")
```

```
presentation("Audrey", ville="Toulouse")
```

- Les **paramètres positionnels** reçoivent les valeurs dans l'ordre de la définition : "Justine", puis 21, puis "Paris".
- Un **argument nommé** s'écrit `nom=valeur` à l'appel : on peut alors donner les paramètres dans n'importe quel ordre.
- Un **paramètre par défaut** a une valeur fixée dans la définition (`age=15`) : on peut l'omettre à l'appel.

④ FONCTIONS EN PARAMÈTRE

On peut passer une fonction en paramètre à une autre fonction, en écrivant son nom sans parenthèses.

```
def appliquer(f, mots):
    return [f(mot) for mot in mots]

print(appliquer(len, ["Les", "mots"]))
# affiche [3, 4]
```

⑤ PRÉCONDITIONS ET DOCUMENTATION

Définition 3 (Assertion) L'instruction `assert condition, "message"` vérifie une condition; si elle est fausse, le programme s'arrête sur une erreur `AssertionError`, avec le message s'il est fourni.

```
def est_une_voyelle(lettre):
    """Renvoie True si lettre, chaîne de
    longueur 1, est une voyelle, False sinon."""
    assert isinstance(lettre, str)
    assert len(lettre) == 1, "une seule lettre"
    return lettre in "aeiouy"
```

`isinstance(v, T)` renvoie `True` si la valeur `v` est de type `T`. La **docstring**, chaîne placée en première ligne du corps, décrit le rôle de la fonction, ses paramètres et sa valeur renvoyée; `help(est_une_voyelle)` l'affiche.

📌 À retenir.

- `return` renvoie une valeur réutilisable; `print` ne fait qu'afficher.
- Une variable affectée dans une fonction est locale; `global` est à éviter.
- Paramètres positionnels, arguments nommés et paramètres par défaut se combinent à l'appel.
- `assert` vérifie les préconditions; la docstring documente la fonction.

Rappels sur les dictionnaires en Python

Fiche de synthèse

① DÉFINITION

Définition 1 (Dictionnaire) Un **dictionnaire** est un ensemble de couples (clé, valeur). On accède à une valeur par sa **clé**, et non par une position comme dans une liste.

Dans un annuaire, les noms sont les clés et les numéros de téléphone les valeurs. Chaque clé est unique ; une même valeur peut apparaître plusieurs fois. Un dictionnaire est **mutable** : on peut ajouter, modifier et supprimer des couples.

② CRÉER UN DICTIONNAIRE

```
d = {} # dictionnaire vide
films = {'Aladdin': 569439, 'Rocketman': 277017}
couples = [('a', 1), ('b', 2)]
d2 = dict(couples) # {'a': 1, 'b': 2}
doubles = {k: 2 * v for k, v in d2.items()}
```

La dernière ligne construit un dictionnaire **par compréhension** : `{'a': 2, 'b': 4}`.

③ OPÉRATIONS

Instruction	Effet
<code>films['Aladdin']</code>	valeur associée à la clé
<code>films['Up'] = 120</code>	ajoute le couple, ou remplace la valeur si la clé existe déjà
<code>del films['Up']</code>	supprime le couple
<code>'Up' in films</code>	True si la clé est présente
<code>len(films)</code>	nombre de couples

④ PARCOURIR UN DICTIONNAIRE

```
for film in films: # les clés
    print(film)
for n in films.values(): # les valeurs
    print(n)
for film, n in films.items(): # les couples
    print(f"{film} : {n} entrées")
```

Parcourir `films` ou `films.keys()` donne les clés, dans l'ordre où elles ont été ajoutées.

⑤ CLÉ ABSENTE

Lire une clé absente provoque une erreur `KeyError`. Deux façons de l'éviter :

```
try:
    n = films['Mon film']
except KeyError:
    n = 0

n = films.get('Mon film', 0)
```

Python exécute le bloc `try` ; si une erreur `KeyError` s'y produit, il exécute le bloc `except` au lieu d'arrêter le programme. La méthode `get(cle, default)` renvoie la valeur associée à la clé, ou `default` si la clé est absente (`None` si on ne donne pas de second paramètre).

⑥ TYPES DES CLÉS ET DES VALEURS

- Une **clé** doit être d'un type non mutable : entier, chaîne de caractères, tuple. Une liste ne peut pas servir de clé : `{[1, 2]: 'a'}` provoque l'erreur `TypeError: unhashable type: 'list'`.
- Une **valeur** peut être de n'importe quel type, y compris une liste ou un autre dictionnaire.

```
grille = {(0, 0): 'T', (0, 1): 'S'}
thon = {'espece': 'T', 'gestation': 3}
mer = {(0, 0): thon}
print(mer[(0, 0)]['espece']) # affiche T
```

⑦ RECHERCHE : LISTE OU DICTIONNAIRE

- Chercher un élément dans une **liste** demande, dans le pire cas, de parcourir tous ses éléments : la complexité est en $O(n)$.
- Chercher une clé dans un **dictionnaire** est en $O(1)$ en moyenne : une **fonction de hachage** transforme la clé en un indice de tableau, ce qui donne directement l'emplacement de la valeur, sans parcours.

Pour $n = 10\,000\,000$, le test `n - 1 in grande_liste` prend quelques centièmes de seconde, le même test sur un dictionnaire environ une microseconde.

À retenir.

- Un dictionnaire associe des valeurs à des clés uniques et non mutables.
- On ajoute ou remplace par `d[cle] = valeur`, on supprime par `del d[cle]`, on teste la présence par `cle in d`.
- On parcourt les clés, `.values()` ou `.items()`.
- Une clé absente se traite avec `try/except KeyError` ou avec `get`.
- La recherche d'une clé est en $O(1)$ en moyenne, contre $O(n)$ dans une liste.

Récurtivité

Fiche de synthèse

① FONCTION RÉCURSIVE

Définition 1 (Fonction récursive) Une fonction est **récursive** quand elle s'appelle elle-même. Elle comporte au moins un **cas de base**, traité sans appel récursif, qui arrête les appels, et un **appel récursif** sur un problème plus petit, qui se rapproche du cas de base.

```
def factorielle(n):
    if n == 0:                # cas de base
        return 1
    else:
        return n * factorielle(n - 1)
```

Pour $n \geq 1$, $n! = n \times (n - 1)!$, et $0! = 1$: la fonction traduit directement cette définition.

✎ Écrire une fonction récursive.

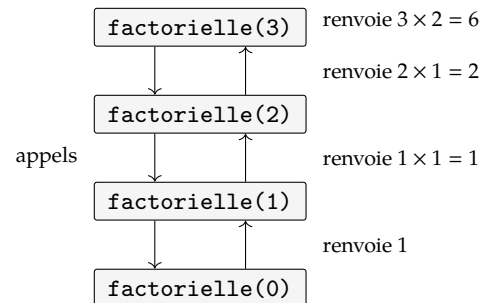
1. Trouver le cas le plus simple et sa réponse : le cas de base.
2. Exprimer le résultat à l'aide du même problème sur une donnée plus petite : $n - 1$, la liste privée de son premier élément `liste[1:]`, la chaîne privée de son premier caractère.
3. Vérifier que chaque appel se rapproche du cas de base, sinon les appels ne s'arrêtent jamais.

```
def somme(liste):
    if len(liste) == 1:
        return liste[0]
    else:
        return liste[0] + somme(liste[1:])

def inverse(s):
    if len(s) == 0:
        return s
    else:
        return inverse(s[1:]) + s[0]
```

② LA PILE D'APPELS

Définition 2 (Pile d'appels) À chaque appel de fonction, Python **empile** un contexte qui contient les paramètres et les variables locales de cet appel. Quand l'appel se termine, son contexte est **dépilé** et son résultat est transmis à l'appel qui l'a déclenché.

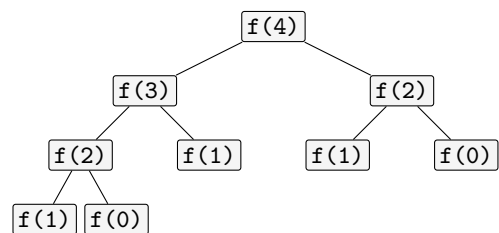


Les appels s'empilent jusqu'au cas de base, puis les résultats remontent de bas en haut. `factorielle(n)` demande $n + 1$ appels : la mémoire utilisée est proportionnelle à n .

③ APPELS MULTIPLES

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

Chaque appel en déclenche deux. L'**arbre des appels** de `fibonacci(4)` montre que `fibonacci(2)` est calculé deux fois et `fibonacci(1)` trois fois :



Le nombre d'appels croît de façon exponentielle : `fibonacci(35)` en demande près de trente millions.

Définition 3 (Mémoïsation) La **mémoïsation** consiste à ranger chaque résultat calculé, par exemple dans un dictionnaire, pour le relire au lieu de le recalculer.

```
memo = {}

def fibonacci_memo(n):
    if n in memo:
        return memo[n]
    if n == 0 or n == 1:
        resultat = n
    else:
        resultat = (fibonacci_memo(n - 1)
                    + fibonacci_memo(n - 2))
    memo[n] = resultat
    return resultat
```

```
        + fibonacci_memo(n - 2))
memo[n] = resultat
return resultat
```

Chaque valeur n'est plus calculée qu'une fois : le nombre d'appels devient proportionnel à n .

④ LIMITES

- Python limite la profondeur de la pile d'appels à environ 1000 : `factorielle(1500)` provoque l'erreur `RecursionError: maximum recursion depth exceeded`. `sys.setrecursionlimit(5000)` relève cette limite, au prix de plus de mémoire.
- Toute fonction récursive peut s'écrire avec une boucle, et inversement. On choisit la forme la plus lisible : la récursivité convient aux structures elles-mêmes récursives, comme les arbres.

À retenir.

- Une fonction récursive a au moins un cas de base et un appel récursif sur un problème plus petit.
- Chaque appel empile un contexte dans la pile d'appels, dépilé quand l'appel se termine.
- Des appels qui recalculent les mêmes valeurs rendent le calcul exponentiel ; la mémorisation les évite.
- La profondeur de récursivité est limitée à environ 1000 appels en Python.

Programmation orientée objet en Python

Fiche de synthèse

① VOCABULAIRE

La **programmation orientée objet** réunit dans un même objet des données et les opérations qui s'y appliquent.

Définition 1 (Objet et classe) Un **objet** regroupe des données, ses **attributs**, et des fonctions qui agissent sur lui, ses **méthodes**. Une **classe** décrit une sorte d'objets, avec leurs attributs et leurs méthodes, et sert à les fabriquer : chaque objet fabriqué est une **instance** de la classe.

② DÉFINIR UNE CLASSE

```
class Point:
    """Point du plan."""

    def __init__(self, abscisse, ordonnee):
        self.x = abscisse
        self.y = ordonnee

    def translater(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy
```

- Le nom d'une classe commence par une majuscule.
- Le premier paramètre de chaque méthode, nommé `self` par convention, reçoit l'objet auquel on applique la méthode.
- La méthode `__init__`, le **constructeur**, crée les attributs de chaque objet dès sa création. Sans elle, un objet n'a aucun attribut et `translater` échouerait sur une erreur `AttributeError`.

③ UTILISER DES OBJETS

```
p = Point(12, 6)          # appelle __init__
p.translater(1, 2)        # self reçoit p
print(f"{p.x} {p.y}")    # affiche 13 8
```

On appelle la classe comme une fonction pour créer une instance. La **notation pointée** donne accès aux attributs (`p.x`) et aux méthodes (`p.translater(1, 2)`). Chaque objet a ses propres valeurs : ce sont des **attributs d'instance**.

④ INTERFACE ET ENCAPSULATION

Définition 2 (Interface) L'**interface** d'une classe est l'ensemble des méthodes que l'utilisateur peut appeler, avec leur spécification. On utilise la classe à travers elle, sans lire le code des méthodes.

Définition 3 (Encapsulation) L'**encapsulation** consiste à ne lire et ne modifier les attributs qu'à travers les méthodes de l'interface, pour que l'objet reste dans un état cohérent.

```
class CompteBancaire:
    def __init__(self, titulaire):
        self.titulaire = titulaire
        self._solde = 0          # attribut interne

    def deposter(self, montant):
        assert montant > 0
        self._solde = self._solde + montant

    def retirer(self, montant):
        assert 0 < montant <= self._solde
        self._solde = self._solde - montant

    def solde(self):
        return self._solde
```

Par convention, un nom qui commence par un tiret bas, comme `_solde`, appartient à l'**implémentation** : on ne l'utilise pas hors de la classe. Python ne l'interdit pas, mais l'implémentation peut alors changer sans modifier les programmes qui utilisent la classe.

⑤ MÉTHODES SPÉCIALES

En Python, toutes les valeurs sont des objets : une liste est une instance de la classe `list`, et `L.append(4)` appelle une méthode. Les **méthodes spéciales**, encadrées par deux tirets bas, sont appelées par Python lui-même quand on utilise une fonction ou un opérateur.

On écrit	Python appelle
<code>Point(12, 6)</code>	<code>__init__(self, ...)</code>
<code>print(p)</code>	<code>__str__(self)</code>
<code>repr(p)</code>	<code>__repr__(self)</code>
<code>p == q</code>	<code>__eq__(self, other)</code>
<code>p + q</code>	<code>__add__(self, other)</code>
<code>len(p)</code>	<code>__len__(self)</code>

```
def __str__(self):
    return f"({self.x}, {self.y})"

def __add__(self, other):
    return Point(self.x + other.x,
                 self.y + other.y)

def __eq__(self, other):
    return (self.x == other.x
            and self.y == other.y)
```

Avec ces méthodes ajoutées à la classe `Point`, `print(Point(12, 6) + Point(3, 4))` affiche `(15, 10)`, et `Point(1, 2) == Point(1, 2)` vaut `True`. Sans `__eq__`, `==` vérifierait seulement s'il s'agit du même objet et donnerait `False`.

Remarque 1. `__str__` et `__repr__` renvoient une chaîne avec `return`; c'est `print` qui l'affiche. Pour afficher une liste de points, Python appelle `__repr__` sur chaque élément.

À retenir.

- Une classe décrit et fabrique des objets; un objet est une instance de sa classe.
- `__init__` crée les attributs; `self` désigne l'objet sur lequel la méthode agit.
- On accède aux attributs et aux méthodes par la notation pointée.
- L'interface est ce que l'utilisateur doit connaître; l'encapsulation protège l'état de l'objet.
- Les méthodes spéciales définissent l'affichage, les opérateurs et les comparaisons des objets.

Programmation orientée objet : pour aller plus loin

Fiche de synthèse

❗ Important. L'héritage, le polymorphisme et la généricité sont hors programme : ils sont présentés pour la culture informatique.

① LES PRINCIPES DE LA PROGRAMMATION ORIENTÉE OBJET

- **Encapsulation** : les données d'un objet lui appartiennent et ne sont modifiées que par ses méthodes, qui forment son interface.
- **Persistence** : les données d'un objet existent tant que l'objet existe, même quand aucune de ses méthodes ne s'exécute.
- **Héritage** : une classe peut reprendre automatiquement les méthodes d'une autre classe, sans les réécrire.
- **Polymorphisme** : une classe peut manipuler des données de types variés ; une pile, par exemple, peut contenir des entiers comme des chaînes de caractères.
- **Généricité** : une solution écrite pour un problème se réutilise pour des problèmes semblables.

② EN PYTHON, LES TYPES SONT DES CLASSES

La classe d'un objet détermine les méthodes qu'on peut lui appliquer : `type([1, 2])` donne `<class 'list'>`, et la classe `list` possède une méthode `append`, que la classe `tuple` ne possède pas. `(1, 2).append(3)` provoque donc une erreur `AttributeError`.

③ ENCAPSULATION NON GARANTIE

- Un nom qui commence par un tiret bas, comme `_solde`, est interne **par convention** : rien n'empêche d'écrire `compte._solde = -1000`.
- Avec deux tirets bas, `__solde`, Python renomme l'attribut en `_CompteBancaire__solde` : `compte.__solde` échoue hors de la classe, mais le nouveau nom reste accessible. L'accès est généré, pas interdit.

④ UN MODÈLE OBJET DYNAMIQUE

Python permet d'ajouter des attributs à un objet, ou des méthodes à une classe, après leur création.

```
class ObjetVide:
    pass

a = ObjetVide()
a.nom = "Objet 1"  # attribut ajouté
```

```
def affiche(self):
    print(self.nom)

ObjetVide.affiche = affiche  # méthode ajoutée
a.affiche()                 # affiche Objet 1
```

Chaque objet a son propre espace de noms : les attributs créés pour `a` n'existent pas pour un autre objet de la même classe.

⚠ Attention. Tout ce qui est possible n'est pas conseillé. On définit les attributs d'instance dans `__init__`, pour que tous les objets d'une classe aient les mêmes attributs.

⑤ ATTRIBUTS DE CLASSE ET D'INSTANCE

Définition 1 (Attributs de classe et d'instance) Un **attribut d'instance** appartient à un seul objet, comme `a.nom`. Un **attribut de classe** est défini sur la classe et partagé par toutes ses instances, comme `ObjetVide.numero`.

```
b = ObjetVide()
ObjetVide.numero = 12
ObjetVide.numero += 1
print(a.numero)  # 13 : valeur partagée
b.numero += 1    # crée un attribut d'instance
print(a.numero)  # 13
print(b.numero)  # 14
```

L'instruction `b.numero += 1` lit l'attribut de classe, puis crée pour `b` seul un attribut d'instance de même nom. Ce mélange devient vite incompréhensible : on limite les attributs de classe aux cas où ils sont nécessaires.

⑥ HÉRITAGE

Définition 2 (Héritage) Quand une classe `B` hérite d'une classe `A`, toutes les méthodes de `A` sont disponibles dans `B`, qui peut en redéfinir certaines. On écrit la classe mère entre parenthèses : `class B(A):`.

```
class Polynome:
    def __init__(self, coefs):
        self.coefs = coefs  # degré décroissant

    def eval(self, x):
        px = 0
        for c in self.coefs:
            px = px * x + c
```

```

        return px

class Affine(Polynome):
    def __init__(self, a, b):
        self.coefs = [a, b]

f = Affine(2, 4)
print(f.eval(12))    # 28, méthode héritée

```

La classe `Affine` ne redéfinit que le constructeur : la méthode `eval`, et toutes les autres méthodes de `Polynome`, s'appliquent à ses instances.

À retenir.

- En Python, les types de base sont des classes.
- L'encapsulation n'est qu'une convention : les attributs internes restent accessibles.
- Un attribut de classe est partagé par toutes les instances ; un attribut d'instance est propre à un objet et se définit dans `__init__`.
- Une classe fille hérite des méthodes de sa classe mère et peut les redéfinir.

Bases de données relationnelles

Fiche de synthèse

① BASE DE DONNÉES ET SGBD

Définition 1 (Base de données et SGBD) Une **base de données** rassemble des informations nombreuses et liées entre elles, chacune à une place définie. Le logiciel qui la gère est le **système de gestion de base de données** (SGBD) : il range les informations, répond aux **requêtes**, écrites en SQL, et applique les modifications.

Un fichier CSV ne suffit plus quand les données sont nombreuses et partagées. Le SGBD rend quatre services :

- la **persistance des données** : elles survivent à une coupure de courant, et une modification interrompue est appliquée en entier ou pas du tout ;
- la **gestion des accès concurrents** : des **verrous** font passer l'une après l'autre les modifications d'une même donnée ;
- l'**efficacité de traitement des requêtes** : des **index** retrouvent une ligne sans parcourir toute la table ;
- la **sécurisation des accès** : chaque utilisateur a des droits de lecture ou de modification.

② POURQUOI PLUSIEURS TABLES

Une table unique des appels d'un opérateur recopie l'adresse du client à chaque appel : c'est une **redondance**, source de trois anomalies.

- **Mise à jour** : un déménagement oblige à modifier toutes les lignes du client ; en oublier une rend la table incohérente.
- **Suppression** : effacer le seul appel d'un client efface ses coordonnées.
- **Insertion** : on ne peut pas enregistrer un client qui n'a encore passé aucun appel.

On découpe alors les données en tables Clients, Telephones et Appels, reliées par des valeurs communes : chaque information n'est écrite qu'une fois.

③ LE MODÈLE RELATIONNEL

Le modèle relationnel a été proposé en 1970 par Edgar Codd.

Définition 2 (Vocabulaire)

- Une **relation** est une table.

- Un **attribut** est une colonne ; il a un nom et un **domaine**, l'ensemble des valeurs qu'il peut prendre, souvent donné par un type : INT, TEXT, REAL, DATE.
- Un **n-uplet**, ou enregistrement, est une ligne : (156, 'Dupont', 'Léon', '3 rue du Marché').
- Le **schéma** d'une relation donne son nom et ses attributs.

④ LES CLÉS

Définition 3 (Clé primaire) La **clé primaire** d'une relation est un attribut, ou un groupe d'attributs, qui identifie chaque n-uplet. Elle est **unique** et jamais vide. On la souligne dans le schéma.

Deux clients peuvent s'appeler Léon Dupont : le nom ne convient pas, on ajoute un identifiant `id_client`.

Définition 4 (Clé étrangère) Une **clé étrangère** est un attribut dont chaque valeur désigne une ligne d'une autre table par sa clé primaire. Elle doit toujours désigner une ligne qui existe. On la fait précéder d'un dièse dans le schéma.

Le **schéma relationnel** de la base est l'ensemble des schémas de ses relations :

- **Clients**(id_client, nom, prenom, adresse)
- **Telephones**(numero, #id_client)
- **Appels**(id_appel, #appelant, appele, duree)

L'attribut `appelle` n'est pas une clé étrangère : le numéro appelé peut appartenir au client d'un autre opérateur.

🔍 **Lire un schéma relationnel.** Repérer les tables, puis les attributs de chaque table, puis les liens créés par les clés étrangères : ici, chaque téléphone appartient à un client et chaque appel part d'un téléphone.

⑤ LES CONTRAINTES D'INTÉGRITÉ

Le SGBD vérifie ces règles à chaque insertion, modification ou suppression, et refuse toute opération qui en viole une.

- **Contrainte de domaine** : chaque valeur appartient au domaine de son attribut ; une durée d'appel est un entier positif ou nul.
- **Contrainte de relation** : la clé primaire est unique et jamais vide ; un second client d'identifiant 156 est refusé.

- **Contrainte de référence** : chaque clé étrangère désigne une ligne existante ; un téléphone du client 999, inexistant, est refusé, de même que la suppression du client 156 tant que des téléphones y font référence.

⑥ STRUCTURE ET CONTENU

La **structure** d'une base est formée du schéma relationnel et des contraintes d'intégrité ; elle change rarement. Le **contenu** est l'ensemble des n-uplets ; il change à chaque insertion, modification ou suppression. Enregistrer un appel modifie le contenu ; ajouter un attribut `date_naissance` modifie la structure.

À retenir.

- Le SGBD assure persistance, gestion des accès concurrents, efficacité des requêtes et sécurisation des accès.
- Plusieurs tables évitent la redondance et les anomalies de mise à jour, de suppression et d'insertion.
- Relation, attribut, domaine, n-uplet, schéma : le vocabulaire du modèle relationnel.
- La clé primaire identifie chaque ligne ; une clé étrangère désigne une ligne d'une autre table.
- Contraintes de domaine, de relation et de référence protègent la cohérence des données.

Requêtes de sélection SQL

Fiche de synthèse

① LE LANGAGE SQL

SQL, langage structuré de requêtes, est utilisé par tous les SGBD relationnels : SQLite, PostgreSQL, MySQL. Il comprend :

- le **langage de définition des données** : `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`;
- le **langage de manipulation des données** : `SELECT`, `INSERT`, `UPDATE`, `DELETE`.

Une requête `SELECT` consulte les données sans les modifier. Les exemples portent sur la table **Produits**(id, nom, catégorie, prix, stock), qui contient huit produits des catégories Papeterie et Informatique.

② SÉLECTIONNER DES COLONNES

```
SELECT * FROM Produits;
SELECT nom, prix FROM Produits;
```

* désigne tous les attributs ; sinon, on nomme les colonnes à afficher.

③ FILTRER LES LIGNES

```
SELECT nom, prix
FROM Produits
WHERE categorie = 'Informatique' AND prix < 20;
```

`WHERE` ne garde que les lignes qui vérifient la condition : ici la clé USB et la souris.

Opérateur	Signification
<code>=, <></code>	égal, différent
<code><, <=, >, >=</code>	comparaisons
<code>BETWEEN a AND b</code>	entre <i>a</i> et <i>b</i> , bornes comprises
<code>LIKE 'C1%'</code>	motif : % remplace toute suite de caractères
<code>AND, OR, NOT</code>	combinent des conditions

Une chaîne de caractères s'écrit entre apostrophes : `'Informatique'`.

④ TRIER ET ÉLIMINER LES DOUBLONS

```
SELECT nom, prix
FROM Produits
WHERE prix < 5
ORDER BY prix DESC;
```

```
SELECT DISTINCT categorie
FROM Produits;
```

- `ORDER BY` trie le résultat, par ordre croissant par défaut (`ASC`), décroissant avec `DESC`.
- `DISTINCT` supprime les lignes identiques du résultat : la seconde requête donne Papeterie et Informatique une seule fois chacune.

⑤ FONCTIONS D'AGRÉGATION

Une fonction d'agrégation calcule une seule valeur à partir de toutes les lignes retenues.

Fonction	Résultat
<code>COUNT(*)</code>	nombre de lignes
<code>SUM(prix)</code>	somme des valeurs
<code>AVG(prix)</code>	moyenne des valeurs
<code>MIN(prix), MAX(prix)</code>	plus petite, plus grande valeur

```
SELECT COUNT(*)
FROM Produits
WHERE stock < 50;
```

Cette requête compte les produits dont le stock est inférieur à 50 : elle renvoie 3.

Remarque 1. *Hors programme.* `GROUP BY categorie` calcule l'agrégat pour chaque catégorie ; `AS` nomme une colonne du résultat ; `LIMIT 1` ne garde que la première ligne.

📌 Écrire une requête de sélection.

1. Choisir les colonnes à afficher, ou l'agrégat à calculer, après `SELECT`.
2. Indiquer la table après `FROM`.
3. Écrire la condition sur les lignes après `WHERE`.
4. Préciser l'ordre du résultat après `ORDER BY`.

Les clauses s'écrivent toujours dans cet ordre.

📌 À retenir.

- `SELECT` colonnes `FROM` table `WHERE` condition `ORDER BY` attribut.
- `DISTINCT` supprime les doublons du résultat.
- `COUNT`, `SUM`, `AVG`, `MIN` et `MAX` calculent une valeur sur les lignes retenues.

Modification des données et définition des tables

Fiche de synthèse

① MODIFIER LE CONTENU

```
INSERT INTO Produits
(id, nom, categorie, prix, stock)
VALUES (9, 'Règle', 'Papeterie', 1.80, 120);
```

```
UPDATE Produits
SET prix = 2.80
WHERE nom = 'Cahier A4';
```

```
DELETE FROM Produits
WHERE stock = 0;
```

- `INSERT INTO ... VALUES` ajoute une ligne; les valeurs suivent l'ordre des attributs nommés.
- `UPDATE ... SET ... WHERE` modifie les lignes qui vérifient la condition.
- `DELETE FROM ... WHERE` supprime les lignes qui vérifient la condition.

⚠ Attention. Sans clause `WHERE`, `UPDATE` modifie toutes les lignes de la table, et `DELETE` les supprime toutes : la table reste, mais vide.

Le SGBD refuse toute opération qui viole une contrainte d'intégrité : un `id` déjà utilisé, une valeur hors du domaine, ou la suppression d'une ligne à laquelle une clé étrangère fait référence.

② DÉFINIR UNE TABLE

```
CREATE TABLE Emprunts (
  id_emprunt INTEGER PRIMARY KEY,
  id_livre INT NOT NULL,
  id_adherent INT NOT NULL,
  date_emprunt TEXT NOT NULL,
  duree INT CHECK (duree > 0),
  FOREIGN KEY (id_livre)
    REFERENCES Livres(id_livre),
  FOREIGN KEY (id_adherent)
    REFERENCES Adherents(id_adherent)
);
```

`CREATE TABLE` définit la structure : le nom, le type et les contraintes de chaque attribut. Les types courants sont `INT` ou `INTEGER`, `REAL`, `TEXT` et `DATE`. `DROP TABLE Emprunts;` supprime la table et toutes ses données, de façon irréversible.

Mot-clé	Contrainte	Effet
NOT NULL	domaine	interdit la valeur vide
CHECK (...)	domaine	restreint les valeurs
UNIQUE	relation	interdit les doublons
PRIMARY KEY	relation	clé primaire : unique et non vide
FOREIGN KEY	référence	la valeur désigne une ligne existante

Remarque 1. SQLite, utilisé en TP, ne vérifie les clés étrangères qu'après `PRAGMA foreign_keys = ON;` et n'interdit la valeur vide dans une clé primaire que si elle est déclarée `INTEGER PRIMARY KEY`.

③ UTILISER SQLITE DEPUIS PYTHON

```
import sqlite3

conn = sqlite3.connect("ma_base.db")
cursor = conn.cursor()
# la table Eleves existe déjà dans la base
cursor.execute("""INSERT INTO Eleves
(id_eleve, nom, prenom, classe)
VALUES (1, 'Dupont', 'Marie', 'TA')""")
conn.commit()

cursor.execute("""SELECT nom, prenom
FROM Eleves WHERE classe = 'TA'""")
for ligne in cursor.fetchall():
    print(ligne) # ('Dupont', 'Marie')
conn.close()
```

- `connect` ouvre la base, et crée le fichier s'il n'existe pas.
- `execute` envoie une requête SQL, écrite dans une chaîne de caractères.
- `commit` enregistre les modifications dans le fichier : sans lui, elles sont perdues à la fermeture.
- `fetchall` renvoie la liste des lignes du résultat, chacune sous forme de tuple; `fetchone` renvoie la ligne suivante.
- `close` ferme la connexion sans rien enregistrer.

À retenir.

- `INSERT`, `UPDATE` et `DELETE` modifient le contenu; sans `WHERE`, toutes les lignes sont touchées.
- `CREATE TABLE` définit la structure et déclare les contraintes d'intégrité; `DROP TABLE` supprime la table.
- En Python, on enregistre les modifications avec `commit` avant de fermer la connexion.

Jointures entre tables

Fiche de synthèse

① POURQUOI UNE JOINTURE

Définition 1 (Jointure) Une **jointure** associe les lignes de deux tables qui ont la même valeur pour un attribut commun, le plus souvent une clé étrangère et la clé primaire qu'elle désigne.

Le modèle relationnel répartit les données dans plusieurs tables pour éviter la redondance : pour afficher le nom d'un élève avec le nom de sa classe, il faut relier les tables **Eleves**(id_eleve, nom, prenom, #id_classe) et **Classes**(id_classe, nom_classe, salle).

nom	prenom	id_classe
Dupont	Marie	101
Martin	Lucas	102

id_classe	nom_classe	salle
101	Terminale A	B204
102	Terminale B	C112

Marie Dupont a la valeur 101 : on l'associe à la ligne 101 de Classes, Terminale A.

② SYNTAXE JOIN ... ON

```
SELECT Eleves.nom, Eleves.prenom,
       Classes.nom_classe
FROM Eleves
JOIN Classes
  ON Eleves.id_classe = Classes.id_classe
WHERE Classes.salle = 'B204';
```

- **JOIN** Classes combine la table Eleves avec la table Classes.
- **ON** donne la condition de jointure : on associe les lignes de même id_classe.
- On préfixe un attribut par le nom de sa table, Eleves .nom, pour lever toute ambiguïté.
- **WHERE** filtre ensuite le résultat, comme dans une requête sur une seule table.

Quand l'attribut commun porte le même nom dans les deux tables, **JOIN** Classes **USING** (id_classe) sous-entend l'égalité.

③ SYNTAXE AVEC WHERE

```
SELECT Eleves.nom, Classes.nom_classe
FROM Eleves, Classes
WHERE Eleves.id_classe = Classes.id_classe;
```

Le résultat est le même. Sans la condition, le SGBD calculerait le **produit cartésien** : toutes les combinaisons d'une ligne de chaque table, soit $4 \times 2 = 8$ lignes pour 4 élèves et 2 classes, sans signification. On préfère **JOIN ... ON**, qui sépare la condition de jointure des filtres.

④ TROIS TABLES ET ALIAS

Pour relier trois tables, on enchaîne les **JOIN**. Un **alias**, déclaré par **AS**, donne un nom court à une table dans toute la requête.

```
SELECT E.nom, M.nom_matiere, M.enseignant
FROM Eleves AS E
JOIN Classes AS C
  ON E.id_classe = C.id_classe
JOIN Matieres AS M
  ON C.id_classe = M.id_classe
WHERE M.enseignant = 'M. Dupuis';
```

⑤ AGRÉGATION ET JOINTURE

```
SELECT COUNT(*)
FROM Eleves
JOIN Classes
  ON Eleves.id_classe = Classes.id_classe
WHERE Classes.nom_classe = 'Terminale A';
```

Cette requête compte les élèves de Terminale A.

📌 Écrire une jointure.

1. Repérer les tables qui contiennent les informations demandées.
2. Repérer l'attribut qui les relie : une clé étrangère et la clé primaire qu'elle désigne.
3. Écrire **FROM T1 JOIN T2 ON T1.cle = T2.cle**, puis les colonnes, les filtres et le tri.

📌 À retenir.

- Une jointure relie les lignes de deux tables qui ont la même valeur d'un attribut commun.
- On écrit **FROM T1 JOIN T2 ON T1.cle = T2.cle**.
- Sans condition de jointure, on obtient le produit cartésien.

Mise au point et débogage des programmes

Fiche de synthèse

① PRÉVENIR LES ERREURS

Un code lisible se débogue plus facilement : noms de variables significatifs, fonctions courtes, spécifiées et documentées par une docstring.

Définition 1 (Annotation de type) Une **annotation** indique le type attendu d'un paramètre et de la valeur renvoyée. Python ne la vérifie pas à l'exécution : elle documente la fonction et sert à l'éditeur et aux outils comme `mypy`.

```
def moyenne(notes: list[float]) -> float:
    return sum(notes) / len(notes)
```

② LIRE UN MESSAGE D'ERREUR

Le message donne le **type** de l'erreur et la **ligne** où elle est apparue, qui n'est pas forcément la ligne fautive : une parenthèse oubliée à la ligne précédente, par exemple, signale l'erreur plus loin.

Erreur	Cause
<code>SyntaxError</code>	mot mal écrit, symbole manquant
<code>IndexError</code>	indice hors de la liste
<code>NameError</code>	nom de variable inconnu
<code>TypeError</code>	opération sur un type qui ne la permet pas, par exemple des parenthèses après un entier
<code>ZeroDivisionError</code>	division par zéro

③ TECHNIQUES DE DÉBOGAGE

- **Affichages** : des `print` provisoires montrent la valeur des variables aux endroits critiques.
- **Assertions** : `assert condition, "message"` arrête le programme si la condition est fausse. On vérifie ainsi des **préconditions**, au début d'une fonction, et des **post-conditions**, juste avant le `return`.
- **Débogueur** : l'instruction `breakpoint()` met l'exécution en pause. La commande `p n` affiche la variable `n`, `n` exécute la ligne suivante, `s` entre dans la fonction appelée, `c` continue jusqu'au point d'arrêt suivant.

④ EXCEPTIONS

Définition 2 (Exception) Une **exception** est une erreur détectée pendant l'exécution. Si rien ne l'intercepte, le programme s'arrête et affiche son message.

```
try:
    x = int(input("Saisir x : "))
    print(f"1/x = {1 / x}")
except ValueError:
    print("Ce n'est pas un entier.")
except ZeroDivisionError:
    print("Division par zéro impossible.")
finally:
    print("Fin du calcul.")
```

- Python exécute le bloc `try` ; à la première erreur, il exécute le bloc `except` du type correspondant au lieu d'arrêter le programme.
- Le bloc `finally` s'exécute dans tous les cas, erreur ou non.
- `raise ValueError("message")` lève soi-même une exception.
- On crée une exception personnalisée par héritage : `class SuperstitionError(Exception): pass`.

⑤ CAUSES FRÉQUENTES D'ERREURS

- **Débordement de tableau** : accès à un indice qui n'existe pas. Penser à la liste vide, et tester l'indice *avant* l'accès : `i < len(lst) and lst[i] < 100`, car `and` n'évalue pas sa seconde condition si la première est fausse.
- **Typage** : `[1, 2] * 2` vaut `[1, 2, 1, 2]` et non `[2, 4]`. En C, un entier `int` peut dépasser sa capacité et devenir négatif.
- **Effet de bord** : une fonction qui modifie une variable extérieure, comme une liste globale, rend le programme difficile à suivre.
- **Flottants** : `while x != y` : peut ne jamais s'arrêter à cause des arrondis ; on écrit `while abs(x - y) > 1e-10`.
- **Conditions** : oublier un cas, ou se tromper entre inégalité stricte et large, entre `and` et `or`. Par exemple, `not a and not b` équivaut à `not (a or b)`.

⑥ TESTS

Définition 3 (Jeu de tests) Un **jeu de tests** est l'ensemble des valeurs utilisées pour tester un programme. Il comprend les cas courants et les **cas limites** : liste vide, valeurs minimale et maximale, valeur interdite.

Avec `doctest`, les exemples de la docstring servent de tests :

```
def fact(n):
    """Renvoie n! pour n entier naturel.

    >>> [fact(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    """
    assert n >= 0, "n doit être positif"
    if n == 0:
        return 1
    return n * fact(n - 1)

import doctest
doctest.testmod() # muet si tout passe
```

Avec `pytest`, chaque fonction dont le nom commence par `test` est un test, lancé par la commande `pytest test_fact.py` :

```
import pytest
from mon_fichier import fact

def test_fact_5():
    assert fact(5) == 120

def test_fact_negatif():
    with pytest.raises(AssertionError):
        fact(-2)
```

Avec `unittest`, les tests sont les méthodes d'une classe qui hérite de `unittest.TestCase` et emploient `self.assertEqual(a, b)`, `self.assertTrue(a)` ou `self.assertRaises(Erreur)`.

Le **développement piloté par les tests** consiste à écrire les tests d'abord, puis le code qui les fait passer.

⑦ PERFORMANCES

Pour mesurer la durée d'un appel à une fonction additionner :

```
import time

debut = time.perf_counter()
resultat = additionner(1_000_000)
fin = time.perf_counter()
print(f"Durée : {fin - debut} s")
```

Le **profilage**, lancé par `python3 -m cProfile fichier.py`, donne pour chaque fonction le nombre d'appels et le temps passé : on optimise d'abord la fonction qui consomme le plus de temps.

À retenir.

- Le message d'erreur donne le type de l'erreur et la ligne où elle est apparue, pas toujours celle qui est fautive.
- Une assertion vérifie une condition pendant le développement ; `try/except` traite une erreur sans arrêter le programme.
- Un jeu de tests couvre les cas courants et les cas limites ; `doctest`, `pytest` et `unittest` les automatisent.
- On ne compare jamais deux flottants avec `==` ou `!=`.
- Le profilage désigne la fonction à optimiser en priorité.

Modularité

Fiche de synthèse

① MODULES

Définition 1 (Module) Un **module** est un fichier `.py` qui regroupe des définitions de fonctions, de classes ou de variables, réutilisables dans d'autres programmes.

La **modularité** consiste à découper un programme en modules, chacun chargé d'une tâche bien définie. Le programme devient plus lisible, chaque module peut être réutilisé et testé séparément, et plusieurs personnes peuvent travailler en même temps sur des modules différents.

Le fichier `geometrie.py` étant placé dans le même répertoire que le programme :

```
import geometrie
print(geometrie.aire_cercle(10))

from geometrie import aire_rectangle
print(aire_rectangle(3, 7))
```

Avec `import`, on préfixe chaque nom par celui du module ; avec `from ... import`, on n'importe que les noms cités et on les emploie sans préfixe.

② DOCUMENTATION ET TESTS

Définition 2 (Docstring) Une **docstring** est la chaîne placée en première ligne d'un module ou d'une fonction. Elle décrit son rôle, ses paramètres et sa valeur renvoyée.

```
def aire_rectangle(longueur, largeur):
    """Renvoie l'aire d'un rectangle.

    >>> aire_rectangle(3, 4)
    12
    """
    return longueur * largeur
```

- `help(aire_rectangle)` affiche cette documentation sans ouvrir le code source.
- Les lignes `>>>` sont des exemples que le module `doctest` exécute : `doctest.testmod()` n'affiche rien si chaque résultat obtenu est celui écrit sous l'exemple, et signale chaque écart sinon.

③ EXÉCUTER OU IMPORTER

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Python donne à chaque module une variable `__name__`. Elle vaut `"__main__"` quand on lance le fichier lui-même : le bloc s'exécute et vérifie les exemples. Elle vaut le nom du module, ici `"geometrie"`, quand un autre programme l'importe : le bloc est ignoré.

④ API ET IMPLÉMENTATION

Définition 3 (API) L'**API** (interface de programmation) d'un module est l'ensemble des fonctions, classes et constantes qu'il met à disposition, avec leur spécification.

L'**implémentation** est le code qui réalise ces fonctions. On utilise un module à travers son API, sans lire son implémentation : celle-ci peut changer sans gêner les utilisateurs, tant que l'interface reste la même. Un nom qui commence par un tiret bas, comme `_carre`, désigne par convention une fonction interne, qui ne fait pas partie de l'API.

📖 **Utiliser une bibliothèque inconnue.** Lire la documentation avec `help(random.choice)`, repérer les paramètres attendus et ce qui est renvoyé (une séquence non vide, l'un de ses éléments au hasard), puis écrire un appel conforme : `random.choice(['rouge', 'vert'])`.

⑤ PAQUETS

Définition 4 (Paquet) Un **paquet** (*package*) est un répertoire qui contient plusieurs modules et un fichier `__init__.py`, qui peut être vide. On importe alors `from mon_jeu.personnage import Guerrier`.

📌 À retenir.

- Un module est un fichier `.py` ; on l'importe avec `import` ou `from ... import`.
- Chaque fonction publique a une docstring, avec des exemples vérifiés par `doctest`.
- Le bloc `if __name__ == "__main__":` ne s'exécute jamais lors d'un import.
- On utilise un module par son API, documentée par `help`, sans dépendre de son implémentation.

Programmation fonctionnelle

Fiche de synthèse

① LE PARADIGME FONCTIONNEL

Le premier langage fonctionnel, Lisp, date de 1958; OCaml et Haskell en sont des descendants. Python n'est pas un langage fonctionnel pur, mais on peut y programmer dans un **style fonctionnel**.

Définition 1 (Style fonctionnel) Un programme en **style fonctionnel** calcule son résultat avec des fonctions, la composition de fonctions, des expressions conditionnelles et la récursivité, sans variable dont la valeur change, sans boucle et sans instruction exécutée pour son effet.

```
# style impératif
def nombre_de_chiffres(n):
    chiffres = 1
    while n >= 10:
        n = n // 10
        chiffres = chiffres + 1
    return chiffres

# style fonctionnel
def nombre_de_chiffres(n):
    return (1 if n < 10 else
            1 + nombre_de_chiffres(n // 10))
```

- L'**expression conditionnelle** `v1 if condition else v2` vaut `v1` ou `v2`, alors qu'une instruction `if` ne vaut rien.
- La **composition** utilise le résultat d'une fonction comme argument d'une autre : `somme(reste(liste))`.

Impératif	Fonctionnel
enchaîne des instructions	compose des fonctions
exécute des instructions	calcule des expressions
répète avec une boucle	rappelle la fonction

② FONCTIONS PURES ET EFFETS DE BORD

Définition 2 (Fonction pure) Une fonction est **pure** quand son résultat ne dépend que de ses arguments et qu'elle ne modifie rien en dehors d'elle-même. Une modification extérieure (variable globale, liste reçue en argument, fichier, écran) s'appelle un **effet de bord**.

```
def ajouter(x, L):          # impure : modifie L
    L.append(x)
    return L
```

```
def ajouter_pur(x, L):      # pure : nouvelle liste
    return L + [x]
```

En programmation fonctionnelle, on construit une nouvelle structure plutôt que de modifier une structure existante.

③ FONCTIONS D'ORDRE SUPÉRIEUR

Définition 3 (Fonction d'ordre supérieur) Une fonction est **d'ordre supérieur** quand elle reçoit une fonction en argument ou renvoie une fonction.

Le mot-clé `lambda` crée une fonction sans nom : `double = lambda x: 2 * x`, puis `double(16)` vaut 32.

```
villes = ['Bordeaux', 'Paris', 'Lyon']
sorted(villes, key=len)
# ['Lyon', 'Paris', 'Bordeaux']
eleves = [("Inès", 14), ("Hugo", 9)]
sorted(eleves, key=lambda e: e[1])
# [('Hugo', 9), ('Inès', 14)]
```

`sorted` renvoie une nouvelle liste triée; avec `key`, elle range les éléments selon leur image par la fonction donnée.

```
list(map(lambda x: x ** 3, [3, 8, 2]))
# [27, 512, 8]
list(filter(lambda x: x > 4, [3, 7, 2, 9]))
# [7, 9]
```

- `map(f, liste)` applique `f` à chaque élément.
- `filter(predicat, liste)` garde les éléments pour lesquels le **prédicat**, fonction qui renvoie un booléen, vaut `True`.
- Les deux renvoient un **itérateur**, qui calcule les valeurs à la demande; `list` le transforme en liste.
- Une liste en compréhension donne les mêmes résultats : `[x ** 3 for x in L]` et `[x for x in L if x > 4]`.

Remarque 1. Hors programme. `reduce(f, liste, depart)`, du module `functools`, combine tous les éléments en une seule valeur : `reduce(lambda acc, x: acc + x, [3, 7, 2], 0)` calcule `0 + 3`, puis `3 + 7`, puis `10 + 2` et vaut 12.

④ TROIS PARADIGMES

Pour calculer la moyenne des notes supérieures ou égales à 10, le style **impératif** parcourt la liste avec une boucle et met à jour une somme et un compteur; le style **objet** écrit une

classe `ReleveDeNotes` munie d'une méthode `moyenne_au_dessus` ; le style **fonctionnel** compose des fonctions :

```
def moyenne_au_dessus(notes):
    garde = list(filter(lambda n: n >= 10,
                        notes))
    return (sum(garde) / len(garde) if garde
            else 0)
```

	Impératif	Fonctionnel	Objet
Brique	instruction, variable	fonction, expression	objet, méthode
Contrôle	boucles, conditions	récurtivité, composition	messages entre objets
État modifié	oui	non	oui, les attributs
Langage	C	Haskell	Smalltalk

Python est **multiparadigme** : on peut y combiner les trois styles.

À retenir.

- Le style fonctionnel compose des fonctions et répète par récursivité, sans modifier de variable.
- Une fonction pure ne dépend que de ses arguments et n'a aucun effet de bord.
- Une fonction d'ordre supérieur reçoit ou renvoie une fonction : `map`, `filter`, `sorted` avec `key`.
- `lambda` crée une fonction sans nom, souvent passée en argument.

Arbres

Fiche de synthèse

① VOCABULAIRE

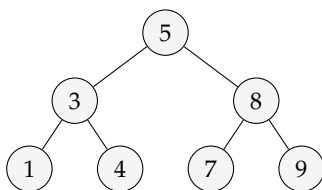
Définition 1 (Arbre) Un **arbre** est une structure hiérarchique formée de **nœuds** reliés par des **arêtes**, sans cycle : entre deux nœuds, il existe un seul chemin.

Exemples : l'arborescence des fichiers, un arbre généalogique, la structure d'une page HTML.

- La **racine** est l'unique nœud sans père ; une **feuille** est un nœud sans fils ; un **nœud interne** a au moins un fils.
- Le **père** d'un nœud est le nœud situé juste au-dessus ; ses **fils** sont les nœuds situés juste en dessous.
- Le **degré** d'un nœud est son nombre de fils.
- La **profondeur** d'un nœud est le nombre d'arêtes entre la racine et ce nœud ; la racine est à la profondeur 0.
- La **hauteur** d'un arbre est la plus grande profondeur de ses feuilles : 0 pour un arbre réduit à sa racine, -1 pour l'arbre vide.
- La **taille** d'un arbre est son nombre de nœuds.

② ARBRES BINAIRES

Définition 2 (Arbre binaire) Un **arbre binaire** est soit vide, soit formé d'une racine et de deux arbres binaires, son **sous-arbre gauche** et son **sous-arbre droit**. Chaque nœud a donc au plus deux fils.



Cet arbre a pour taille 7 et pour hauteur 2. Son interface comprend les opérations `est_vide`, `racine`, `gauche`, `droit` et `est_feuille`, qui ne s'appliquent qu'à un arbre non vide, sauf la première.

```

class Arbre:
    def __init__(self, valeur, gauche=None,
                  droit=None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

    def est_feuille(self):
        return (self.gauche == None
                and self.droit == None)
  
```

```

a = Arbre(5, Arbre(3, Arbre(1), Arbre(4)),
          Arbre(8, Arbre(7), Arbre(9)))
  
```

L'arbre vide est représenté par `None`. Ainsi, `a.droit.gauche.valeur` vaut 7.

③ ALGORITHMES RÉCURSIFS

Un arbre binaire est défini à partir de sous-arbres : on traite l'arbre vide en cas de base, puis on combine les résultats des deux sous-arbres.

```

def taille(arbre):
    if arbre == None:
        return 0
    return (1 + taille(arbre.gauche)
            + taille(arbre.droit))

def hauteur(arbre):
    if arbre == None:
        return -1
    return 1 + max(hauteur(arbre.gauche),
                   hauteur(arbre.droit))
  
```

④ PARCOURS EN PROFONDEUR

Parcours	Ordre	Sur l'arbre ci-dessus
préfixe	nœud, gauche, droit	5, 3, 1, 4, 8, 7, 9
infixe	gauche, nœud, droit	1, 3, 4, 5, 7, 8, 9
suffixe	gauche, droit, nœud	1, 4, 3, 7, 9, 8, 5

```

def prefixe(arbre):
    if arbre != None:
        print(arbre.valeur)
        prefixe(arbre.gauche)
        prefixe(arbre.droit)
  
```

Pour les parcours infixe et suffixe, on déplace l'instruction `print` entre les deux appels récursifs, ou après eux.

⑤ PARCOURS EN LARGEUR

Le parcours en largeur visite les nœuds niveau par niveau, de gauche à droite : 5, 3, 8, 1, 4, 7, 9. Il utilise une **file** : premier entré, premier sorti.

```

def largeur(arbre):
    file = [arbre]
    while file != []:
        noeud = file.pop(0)
  
```

```

if noeud != None:
    print(noeud.valeur)
    file.append(noeud.gauche)
    file.append(noeud.droit)

```

⑥ ARBRES BINAIRES DE RECHERCHE

Définition 3 (Arbre binaire de recherche) Un **arbre binaire de recherche** (ABR) est un arbre binaire dans lequel, pour chaque nœud, les valeurs du sous-arbre gauche sont strictement inférieures à la valeur du nœud, et celles du sous-arbre droit strictement supérieures.

L'arbre de cette fiche est un ABR : son parcours infixe donne les valeurs dans l'ordre croissant, ce qui est vrai pour tout ABR.

```

def rechercher(arbre, x):
    if arbre == None:
        return False
    if x == arbre.valeur:
        return True
    elif x < arbre.valeur:
        return rechercher(arbre.gauche, x)
    else:
        return rechercher(arbre.droit, x)

```

La recherche suit un seul chemin de la racine vers une feuille : son coût dépend de la hauteur de l'arbre.

- Dans un arbre **équilibré**, les deux sous-arbres de chaque nœud ont des hauteurs proches ; la hauteur est de l'ordre de $\log_2 n$ et la recherche est en $O(\log n)$.
- Dans un arbre **dégénéré**, chaque nœud a au plus un fils, comme après l'insertion de valeurs déjà triées ; l'arbre se comporte comme une liste et la recherche est en $O(n)$.

📌 À retenir.

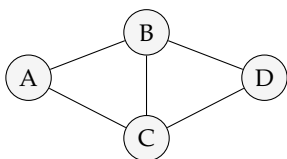
- La taille compte les nœuds ; la hauteur est la plus grande profondeur d'une feuille.
- Les algorithmes sur un arbre binaire sont récurifs : cas de l'arbre vide, puis les deux sous-arbres.
- Préfixe, infixe et suffixe sont des parcours en profondeur ; le parcours en largeur utilise une file.
- Le parcours infixe d'un ABR donne ses valeurs dans l'ordre croissant.
- La recherche dans un ABR est en $O(\log n)$ s'il est équilibré, en $O(n)$ s'il est dégénéré.

Graphes

Fiche de synthèse

① VOCABULAIRE

Définition 1 (Graphe) Un **graphe** est formé d'un ensemble de **sommets** et d'un ensemble d'**arêtes**, chaque arête reliant deux sommets.



- L'**ordre** d'un graphe est son nombre de sommets : 4 ci-dessus.
- Deux sommets reliés par une arête sont **adjacents**.
- Une **boucle** relie un sommet à lui-même ; un graphe **simple** n'a aucune boucle et au plus une arête entre deux sommets.
- Dans un graphe **orienté**, chaque arête a un sens de parcours : on parle d'**arcs**.
- Dans un graphe **complet**, deux sommets distincts quelconques sont reliés par une arête.

② DEGRÉ D'UN SOMMET

Définition 2 (Degré) Le **degré** d'un sommet est le nombre d'extrémités d'arêtes en ce sommet. Dans un graphe simple, c'est le nombre d'arêtes issues du sommet ; une boucle compte pour deux.

Théorème 1 (Somme des degrés) La somme des degrés de tous les sommets d'un graphe est égale au double du nombre d'arêtes.

Ci-dessus, les degrés de A, B, C et D sont 2, 3, 3 et 2 : leur somme vaut 10, le double des 5 arêtes. Dans un graphe complet à n sommets, chaque sommet est de degré $n - 1$.

③ CHAÎNES ET CONNEXITÉ

- Une **chaîne** est une suite d'arêtes mises bout à bout ; sa **longueur** est son nombre d'arêtes.
- Un **cycle** est une chaîne dont les arêtes sont toutes distinctes et dont les extrémités coïncident : A, B, C, A ci-dessus.
- Un graphe est **connexe** quand deux sommets quelconques sont reliés par une chaîne : il est d'un seul tenant.

Remarque 1. Hors programme. Une chaîne **eulérienne** emprunte chaque arête une seule fois. Un graphe en possède

une si, et seulement si, il est connexe et a zéro ou deux sommets de degré impair. Le graphe ci-dessus en a deux, B et C : une chaîne eulérienne va de B à C.

④ MATRICE D'ADJACENCE

Définition 3 (Matrice d'adjacence) Les sommets étant numérotés de 1 à n , la **matrice d'adjacence** est le tableau carré A de taille n dont le coefficient a_{ij} est le nombre d'arêtes reliant le sommet i au sommet j .

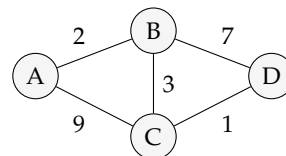
Avec A, B, C, D numérotés de 1 à 4, le graphe ci-dessus a pour matrice

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

- La matrice d'un graphe non orienté est symétrique par rapport à sa diagonale.
- Dans un graphe simple non orienté, la somme d'une ligne est le degré du sommet correspondant.

⑤ GRAPHES PONDÉRÉS

Définition 4 (Graphe pondéré) Un graphe **pondéré** associe à chaque arête un nombre positif, son **poids**. Le **poids** d'une chaîne est la somme des poids de ses arêtes.



Une **plus courte chaîne** entre deux sommets est une chaîne de poids minimal ; ce poids est la **distance** entre les deux sommets. Ici, la plus courte chaîne de A à D est A, B, C, D, de poids 6 : elle a trois arêtes, alors que A, B, D n'en a que deux mais pèse 9. On ne confond pas le poids d'une chaîne et sa longueur.

À retenir.

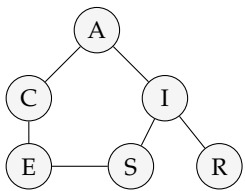
- Un graphe est formé de sommets et d'arêtes, ou d'arcs s'il est orienté.
- La somme des degrés vaut le double du nombre d'arêtes.
- Le poids d'une chaîne est la somme des poids de ses arêtes, sa longueur est son nombre d'arêtes.

Parcours de graphes

Fiche de synthèse

① REPRÉSENTER UN GRAPHE

- Une **matrice d'adjacence** est un tableau à deux dimensions : la case de la ligne i et de la colonne j indique la présence, ou le poids, de l'arc du sommet i vers le sommet j . Elle occupe n^2 cases pour n sommets.
- Une **liste d'adjacence** donne, pour chaque sommet, la liste de ses voisins. Elle convient mieux quand il y a peu d'arêtes. En Python, on l'écrit avec un dictionnaire dont les clés sont les sommets.



```
G = { "A": [ "C", "I" ],
      "C": [ "A", "E" ],
      "I": [ "A", "R", "S" ],
      "R": [ "I" ],
      "E": [ "C", "S" ],
      "S": [ "E", "I" ] }
```

Un graphe non orienté se traite comme un graphe orienté où chaque arête est un arc dans les deux sens. Pour un graphe pondéré, on range des couples (voisin, poids).

② PARCOURS EN PROFONDEUR

Définition 1 (Parcours en profondeur) Le **parcours en profondeur** avance d'un sommet à un voisin non encore visité tant que c'est possible, puis revient en arrière. Il utilise une **pile**, celle des appels récursifs ou une pile explicite.

```
def profondeur(G, S, deja_vus):
    deja_vus.append(S)          # marquer S
    print(S)                   # traiter S
    for voisin in G[S]:
        if voisin not in deja_vus:
            profondeur(G, voisin, deja_vus)

profondeur(G, "A", [])         # A C E S I R
```

La version itérative empile les voisins et marque un sommet quand elle le dépile :

```
def profondeur_iteratif(G, S):
    deja_vus = []
    pile = [S]
    while pile != []:
        S = pile.pop()
        if S not in deja_vus:
            deja_vus.append(S)
            print(S)
            for voisin in G[S]:
                if voisin not in deja_vus:
```

```
pile.append(voisin)
```

Elle affiche A I S E C R : les voisins sont dépilés dans l'ordre inverse de leur empilement, donc le dernier voisin de A est visité en premier.

③ PARCOURS EN LARGEUR

Définition 2 (Parcours en largeur) Le **parcours en largeur** visite le sommet de départ, puis tous ses voisins, puis les voisins de ses voisins, et ainsi de suite. Il utilise une **file**.

```
def largeur(G, S):
    deja_vus = [S]
    file = [S]
    while file != []:
        S = file.pop(0)
        print(S)
        for voisin in G[S]:
            if voisin not in deja_vus:
                deja_vus.append(voisin)
                file.append(voisin)
```

Depuis A, il affiche A, puis C et I, à une arête de A, puis E, R et S, à deux arêtes.

	Profondeur	Largeur
Structure	pile : dernier entré, premier sorti	file : premier entré, premier sorti
Marquage	au dépilement	à l'enfilement

En largeur, un sommet est atteint par une chaîne qui a le moins d'arêtes possible : le parcours en largeur donne un **plus court chemin en nombre d'arêtes**.

④ DÉTECTER UN CYCLE

On parcourt le graphe en profondeur sans revenir par l'arête d'arrivée. Si l'on dépile un sommet déjà marqué, il a été atteint par deux chemins différents : le graphe possède un cycle.

```
def possede_cycle(G, S):
    deja_vus = []
    pile = [S]
    while pile != []:
        S = pile.pop()
        if S in deja_vus:
            return True
```

```

    deja_vus.append(S)
    for voisin in G[S]:
        if voisin not in deja_vus:
            pile.append(voisin)
    return False

```

⑤ RECHERCHER UN CHEMIN

On parcourt le graphe en largeur en notant, dans un dictionnaire, le **prédécesseur** de chaque sommet atteint, c'est-à-dire le sommet d'où l'on vient. On reconstitue ensuite le chemin en remontant les prédécesseurs depuis l'arrivée jusqu'au départ.

```

def chemin(G, depart, arrivee):
    pred = {depart: None}
    file = [depart]
    while file != []:
        S = file.pop(0)
        for voisin in G[S]:
            if voisin not in pred:
                pred[voisin] = S
                file.append(voisin)
    if arrivee not in pred:
        return []
    R = [arrivee]
    while R[-1] != depart:
        R.append(pred[R[-1]])
    R.reverse()
    return R

```

`chemin(G, "A", "S")` renvoie ['A', 'I', 'S'].

📌 À retenir.

- Un graphe se représente par une matrice ou par un dictionnaire d'adjacence.
- Le parcours en profondeur utilise une pile, le parcours en largeur une file.
- On marque les sommets visités pour ne pas les traiter deux fois.
- Le parcours en largeur trouve un chemin qui a le moins d'arêtes possible.
- Rencontrer un sommet déjà marqué en profondeur révèle un cycle.

Algorithme de Dijkstra

Fiche de synthèse

! Important. L'algorithme de Dijkstra est hors programme : il prolonge le chapitre sur les parcours de graphes.

① PRINCIPE

Dans un graphe pondéré dont tous les poids sont positifs ou nuls, l'algorithme de Dijkstra, publié en 1959, calcule les plus courts chemins d'un sommet de départ vers tous les autres sommets. Il repose sur une propriété : un morceau d'un plus court chemin est lui-même un plus court chemin.

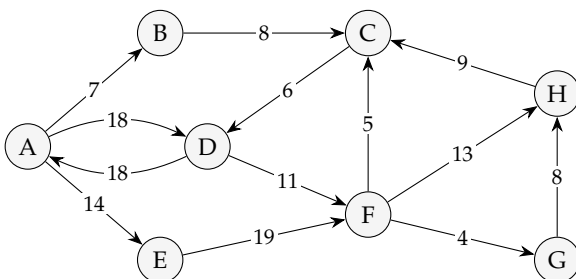
≡ Algorithme de Dijkstra.

1. Donner la distance 0 au sommet de départ et la distance $+\infty$ à tous les autres.
2. Choisir, parmi les sommets non encore traités, celui dont la distance provisoire est la plus petite.
3. Pour chacun de ses voisins, si passer par ce sommet donne une distance plus petite que celle connue, remplacer la distance du voisin et noter ce sommet comme son **prédécesseur**.
4. Recommencer à l'étape 2 jusqu'à ce que tous les sommets soient traités.

Un poids négatif fausserait les distances des sommets déjà traités : l'algorithme ne s'applique qu'aux poids positifs ou nuls. Il est utilisé par les calculateurs d'itinéraires et par le protocole de routage OSPF.

② EXEMPLE

On cherche le plus court chemin de A à H dans ce graphe orienté, dont les poids sont des distances en kilomètres.



La notation 7_A signifie « distance 7, en venant de A ». Chaque ligne indique le sommet choisi, puis les distances mises à jour.

Choisi	B	C	D	E	F	G	H
A (0)	7_A	∞	18_A	14_A	∞	∞	∞
B (7)		15_B	18_A	14_A	∞	∞	∞
E (14)		15_B	18_A		33_E	∞	∞
C (15)			18_A		33_E	∞	∞
D (18)					29_D	∞	∞
F (29)						33_F	42_F
G (33)							41_G

Le sommet H est enfin choisi avec la distance 41. En traitant D, la distance de F passe de 33 à $18 + 11 = 29$. En remontant les prédécesseurs depuis H, on obtient $H \leftarrow G \leftarrow F \leftarrow D \leftarrow A$: le plus court chemin est A, D, F, G, H, de longueur 41 km.

③ IMPLÉMENTATION EN PYTHON

Le graphe est un dictionnaire de dictionnaires : `graphe["A"]["B"]` est le poids de l'arc de A vers B.

```
graphe = {"A": {"B": 7, "E": 14, "D": 18},
          "B": {"C": 8, "C": {"D": 6}},
          "D": {"A": 18, "F": 11},
          "E": {"F": 19},
          "F": {"G": 4, "H": 13, "C": 5},
          "G": {"H": 8, "H": {"C": 9}}

def dijkstra(graphe, depart):
    dist = {s: float("inf") for s in graphe}
    dist[depart] = 0
    pred = {s: None for s in graphe}
    non_traites = list(graphe)
    while non_traites != []:
        s = min(non_traites,
                key=lambda x: dist[x])
        non_traites.remove(s)
        for voisin, poids in graphe[s].items():
            if dist[s] + poids < dist[voisin]:
                dist[voisin] = dist[s] + poids
                pred[voisin] = s
    return dist, pred

def plus_court_chemin(graphe, depart, arrivee):
    dist, pred = dijkstra(graphe, depart)
    chemin = []
    s = arrivee
    while s != None:
        chemin.append(s)
        s = pred[s]
    chemin.reverse()
    return chemin, dist[arrivee]
```

`float("inf")` est un flottant plus grand que tout nombre : il représente une distance infinie. `plus_court_chemin(graphe, "A", "H")` renvoie `(['A', 'D', 'F', 'G', 'H'], 41)`.

④ COMPLEXITÉ

Pour n sommets, chaque étape cherche le minimum en parcourant les sommets non traités : la complexité est en $O(n^2)$. Avec un tas, structure qui donne rapidement le minimum, elle descend à $O((n + m) \log n)$, où m est le nombre d'arêtes : l'algorithme traite alors des réseaux routiers de millions de sommets.

À retenir.

- Dijkstra calcule les plus courts chemins depuis un sommet, dans un graphe à poids positifs ou nuls.
- À chaque étape, il traite le sommet non traité de plus petite distance, puis met à jour ses voisins.
- On reconstitue le chemin en remontant les prédécesseurs depuis l'arrivée.
- La complexité est en $O(n^2)$ avec une recherche simple du minimum.

Diviser pour régner

Fiche de synthèse

① LE PARADIGME

Définition 1 (Diviser pour régner) Un algorithme **diviser pour régner** procède en trois étapes :

1. **diviser** le problème en sous-problèmes plus petits de même nature ;
2. **régner** : résoudre chaque sous-problème récursivement, jusqu'à des cas assez simples pour être résolus directement ;
3. **combiner** les solutions des sous-problèmes en une solution du problème initial.

L'efficacité vient de la division : couper le problème en deux moitiés à chaque étape mène à $\log_2 n$ niveaux d'appels.

② RECHERCHE DICHOTOMIQUE

Dans un tableau **trié**, on compare l'élément cherché à celui du milieu : s'ils sont égaux, on l'a trouvé ; s'il est plus petit, on cherche dans la moitié gauche ; sinon, dans la moitié droite.

```
def recherche_dichotomique(x, tab):
    gauche = 0
    droite = len(tab) - 1
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if tab[milieu] == x:
            return milieu
        elif tab[milieu] < x:
            gauche = milieu + 1
        else:
            droite = milieu - 1
    return -1
```

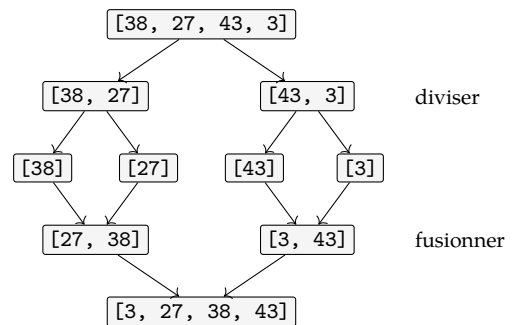
La fonction renvoie l'indice de x , ou -1 s'il est absent. La version récursive reçoit *gauche* et *droite* en paramètres et s'appelle sur la moitié retenue.

L'intervalle de recherche est divisé par deux à chaque étape : il y a au plus $\lfloor \log_2 n \rfloor + 1$ étapes, et la complexité est en $O(\log n)$. Pour un million d'éléments, 20 comparaisons suffisent, contre un million pour une recherche séquentielle.

③ TRI FUSION

1. Diviser : couper le tableau en deux moitiés.
2. Régner : trier chaque moitié par tri fusion.
3. Combiner : **fusionner** les deux moitiés triées.

Un tableau de zéro ou un élément est déjà trié : c'est le cas de base.



```
def fusion(gauche, droite):
    resultat = []
    i, j = 0, 0
    while i < len(gauche) and j < len(droite):
        if gauche[i] <= droite[j]:
            resultat.append(gauche[i])
            i += 1
        else:
            resultat.append(droite[j])
            j += 1
    resultat.extend(gauche[i:])
    resultat.extend(droite[j:])
    return resultat

def tri_fusion(tab):
    if len(tab) <= 1:
        return tab
    milieu = len(tab) // 2
    gauche = tri_fusion(tab[:milieu])
    droite = tri_fusion(tab[milieu:])
    return fusion(gauche, droite)
```

La fusion prend à chaque étape le plus petit des deux premiers éléments restants ; quand une liste est épuisée, **extend** ajoute à la fin du résultat les éléments restants de l'autre.

Il y a $\log_2 n$ niveaux de découpage, et la fusion parcourt les n éléments à chaque niveau : la complexité est en $O(n \log n)$ dans tous les cas, contre $O(n^2)$ pour les tris par sélection et par insertion. Le tri fusion n'est pas **en place** : il crée de nouvelles listes, ce qui demande de la mémoire supplémentaire.

④ EXPONENTIATION RAPIDE

$$a^n = \begin{cases} 1 & \text{si } n = 0, \\ (a^{n/2})^2 & \text{si } n \text{ est pair,} \\ a \times (a^{(n-1)/2})^2 & \text{si } n \text{ est impair.} \end{cases}$$

```
def puissance(a, n):
    if n == 0:
        return 1
```

```

demi = puissance(a, n // 2)
if n % 2 == 0:
    return demi * demi
else:
    return a * demi * demi

```

Pour n impair, $n // 2$ vaut $(n - 1)/2$. Le calcul demande $O(\log n)$ multiplications au lieu de $n - 1$.

	Diviser	Régner	Combiner
Dichotomie	couper l'intervalle en deux	chercher dans une moitié	renvoyer l'indice
Tri fusion	couper le tableau en deux	trier chaque moitié	fusionner
Puissance	diviser l'exposant par deux	calculer la puissance réduite	élever au carré, multiplier par a

À retenir.

- Diviser pour régner : diviser, résoudre récursivement, combiner.
- La recherche dichotomique, dans un tableau trié, est en $O(\log n)$.
- Le tri fusion est en $O(n \log n)$ dans tous les cas, mais n'est pas en place.
- L'exponentiation rapide calcule a^n en $O(\log n)$ multiplications.

Tri rapide

Fiche de synthèse

❗ **Important.** Le tri rapide est hors programme : il illustre le paradigme diviser pour régner et se compare au tri fusion, qui est au programme.

① PRINCIPE

☰ Tri rapide.

1. Choisir un élément du tableau, le **pivot**.
2. **Partitionner** : placer à gauche les éléments plus petits que le pivot, à droite les plus grands.
3. Trier récursivement la partie gauche et la partie droite.

Avec le tableau $[7, 2, 1, 6, 8, 5, 3, 4]$ et le pivot 4, on obtient les éléments plus petits $[2, 1, 3]$, le pivot $[4]$ et les éléments plus grands $[7, 6, 8, 5]$; on trie ensuite récursivement $[2, 1, 3]$ et $[7, 6, 8, 5]$.

```
def tri_rapide(tab):
    if len(tab) <= 1:
        return tab
    pivot = tab[0]
    petits = [x for x in tab[1:] if x < pivot]
    egaux = [x for x in tab if x == pivot]
    grands = [x for x in tab[1:] if x > pivot]
    return (tri_rapide(petits) + egaux
            + tri_rapide(grands))
```

② DIVISER POUR RÉGNER

Le tri rapide divise le problème en deux sous-problèmes, les résout récursivement, puis combine les résultats par une simple concaténation. Le travail principal se fait lors de la **division**, par la partition. Dans le tri fusion, au contraire, la division est immédiate (couper au milieu) et le travail principal se fait lors de la **combinaison**, par la fusion.

③ COMPLEXITÉ

- **Meilleur cas** : le pivot partage toujours le tableau en deux parties de même taille. Il y a $\log_2 n$ niveaux d'appels, et les partitions d'un même niveau parcourent n éléments en tout : la complexité est en $O(n \log n)$, comme le tri fusion.
- **Pire cas** : le pivot est toujours le plus petit ou le plus grand élément, par exemple en prenant le premier élément d'un tableau déjà trié. Chaque partition ne retire que le pivot : il y a n niveaux, et la complexité est en $O(n^2)$.
- **Cas moyen** : $O(n \log n)$.

Choisir le pivot au hasard, ou la médiane de trois éléments, rend le pire cas très improbable.

④ COMPARAISON AVEC LE TRI FUSION

Critère	Tri fusion	Tri rapide
Meilleur cas	$O(n \log n)$	$O(n \log n)$
Cas moyen	$O(n \log n)$	$O(n \log n)$
Pire cas	$O(n \log n)$	$O(n^2)$
Mémoire	nouvelles listes	en place possible
Stable	oui	non en version en place

Définition 1 (Tri stable) Un tri est **stable** quand deux éléments égaux restent dans l'ordre où ils étaient avant le tri.

Le tri rapide écrit avec des listes en compréhension crée de nouvelles listes ; sa version en place, qui échange les éléments dans le tableau lui-même, n'utilise presque pas de mémoire supplémentaire, mais n'est pas stable. En pratique, le tri rapide est souvent plus rapide que le tri fusion. On préfère le tri fusion quand il faut garantir $O(n \log n)$ ou un tri stable.

📌 À retenir.

- Le tri rapide partitionne autour d'un pivot, puis trie récursivement les deux parties.
- Il est en $O(n \log n)$ en moyenne et en $O(n^2)$ dans le pire cas.
- Son travail se fait à la division, celui du tri fusion à la combinaison.

Programmation dynamique

Fiche de synthèse

① PRINCIPE

Définition 1 (Programmation dynamique) La **programmation dynamique** résout un problème à partir de sous-problèmes plus petits qui **se répètent** : chaque sous-problème est résolu une seule fois, et son résultat est conservé pour être réutilisé.

Avec diviser pour régner, les sous-problèmes sont **indépendants** : le tri fusion trie deux moitiés sans élément commun, et rien n'est à conserver. Pour la suite de Fibonacci, au contraire, le calcul de F_{40} demande F_{38} deux fois, F_{37} trois fois, et ainsi de suite.

② LA SUITE DE FIBONACCI

$F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$. La version récursive directe recalcule sans cesse les mêmes termes : `fibonacci(40)` prend plusieurs secondes.

Mémoïsation, ou approche **descendante** : on garde la fonction récursive et on range chaque terme calculé dans un tableau.

```
def fibo_memo(n, tableau):
    if n == 0 or n == 1:
        return n
    if tableau[n] == 0:
        a = fibo_memo(n - 1, tableau)
        b = fibo_memo(n - 2, tableau)
        tableau[n] = a + b
    return tableau[n]

print(fibo_memo(40, [0] * 41))
```

Approche **ascendante** : on remplit le tableau en partant des plus petits cas.

```
def fibo_dyn(n):
    if n == 0:
        return 0
    tableau = [0] * (n + 1)
    tableau[1] = 1
    for i in range(2, n + 1):
        tableau[i] = (tableau[i - 1]
                     + tableau[i - 2])
    return tableau[n]
```

Chaque terme est calculé une seule fois : le calcul de F_n demande un nombre d'opérations proportionnel à n .

③ LE PROBLÈME DU SAC À DOS

Chaque objet a une valeur et un poids ; le sac supporte au plus un poids w . On cherche les objets qui donnent la plus grande valeur totale sans dépasser w . La force brute essaie les 2^n choix possibles ; un algorithme glouton est rapide mais ne garantit pas le meilleur choix.

☞ **Sac à dos par programmation dynamique.** On note $T[i][p]$ la meilleure valeur obtenue avec les i premiers objets et un sac de capacité p . Pour le i -ième objet, de valeur v et de poids q :

- on le **laisse** : la valeur est $T[i - 1][p]$;
- on le **prend**, si $q \leq p$: la valeur est $v + T[i - 1][p - q]$.

$T[i][p]$ est la plus grande de ces deux valeurs. Chaque ligne du tableau se calcule à partir de la précédente.

Pour les objets (3, 2), (8, 10), (2, 2), (8, 1), (4, 6) et (6, 6), au format (valeur, poids), et $w = 10$:

$i \backslash p$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3	3
2	0	0	3	3	3	3	3	3	3	3	8
3	0	0	3	3	5	5	5	5	5	5	8
4	0	8	8	11	11	13	13	13	13	13	13
5	0	8	8	11	11	13	13	13	13	15	15
6	0	8	8	11	11	13	13	14	14	17	17

La meilleure valeur est 17. On retrouve les objets en remontant le tableau : quand une valeur diffère de celle de la ligne du dessus, l'objet de cette ligne est pris, et l'on retire son poids à la capacité. On obtient les objets 1, 4 et 6, de poids total 9 kg.

```
def sac_a_dos(objets, w):
    n = len(objets)
    T = [[0] * (w + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        v, q = objets[i - 1]
        for p in range(w + 1):
            T[i][p] = T[i - 1][p]
            if q <= p:
                avec = v + T[i - 1][p - q]
                T[i][p] = max(T[i][p], avec)
    return T[n][w]
```

④ DISTANCE D'ÉDITION

Définition 2 (Distance d'édition) La **distance d'édition** entre deux chaînes est le plus petit nombre de modifications, **substitution**, **insertion** ou **suppression** d'un caractère, qui transforme la première en la seconde.

La distance de "CHAT" à "CHATS" vaut 1 ; celle de "NICHE" à "CHIEN" vaut 4. On compare ainsi des séquences d'ADN.

On note $d(i, j)$ la distance entre les i premiers caractères de A et les j premiers de B ; c vaut 0 si $A[i - 1] = B[j - 1]$ et 1 sinon. Alors $d(0, j) = j$, $d(i, 0) = i$ et

$d(i, j) = \min(d(i - 1, j) + 1, d(i, j - 1) + 1, d(i - 1, j - 1) + c)$.

```
def distance_edition(a, b):
    n, m = len(a), len(b)
    t = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        t[i][0] = i
    for j in range(m + 1):
        t[0][j] = j
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            c = 0 if a[i - 1] == b[j - 1] else 1
            t[i][j] = min(t[i - 1][j] + 1,
                          t[i][j - 1] + 1,
                          t[i - 1][j - 1] + c)
    return t[n][m]
```

La version récursive directe déclenche trois appels à chaque étape et devient exponentielle ; le tableau rempli ligne par ligne demande un temps et une mémoire en $O(n \times m)$.

📌 À retenir.

- La programmation dynamique s'applique quand les sous-problèmes se répètent ; diviser pour régner, quand ils sont indépendants.
- La mémoïsation garde la récursivité et range les résultats ; l'approche ascendante remplit un tableau depuis les plus petits cas.
- Sac à dos : on compare, pour chaque objet et chaque capacité, le choix de le laisser et celui de le prendre.
- La distance d'édition se calcule dans un tableau en $O(n \times m)$.

Recherche textuelle

Fiche de synthèse

① LE PROBLÈME

On cherche toutes les positions d'un **motif** de longueur m dans un **texte** de longueur n , avec $m \leq n$. Le motif 'bra' apparaît dans 'abracadabra' aux positions 1 et 8. Ce problème intervient dans les éditeurs de texte, les moteurs de recherche, la commande `grep` et l'analyse de l'ADN.

② ALGORITHME NAÏF

On essaie chaque position i de 0 à $n - m$ et l'on compare le motif au texte, caractère par caractère, de gauche à droite; au premier caractère différent, on passe à la position suivante.

```
def algo_naif(texte, motif):
    n = len(texte)
    m = len(motif)
    indices = []
    for i in range(n - m + 1):
        trouve = True
        for j in range(m):
            if texte[i + j] != motif[j]:
                trouve = False
                break
        if trouve:
            indices.append(i)
    return indices
```

La borne est $n - m + 1$ car $n - m$ est la dernière position où le motif tient dans le texte. Il y a $n - m + 1$ positions et au plus m comparaisons pour chacune : la complexité dans le pire cas est en $O(n \times m)$, quadratique quand m est de l'ordre de n . Chercher 'aab' dans 'aaaaaaaa' demande presque trois comparaisons à chaque position.

③ ALGORITHME DE BOYER-MOORE-HORSPOOL

- On compare le motif au texte **de droite à gauche**, en commençant par la dernière lettre du motif.
- Après un échec, on décale le motif d'après le caractère du texte aligné avec la **fin** du motif : on l'aligne avec l'occurrence la plus à droite de ce caractère dans le motif, ou l'on décale de m si le motif ne le contient pas.

Recherche de 'dab' dans 'abracadabra' ; la dernière lettre du motif, encadrée, est comparée en premier à la lettre du texte située au-dessus :

0	1	2	3	4	5	6	7	8	9	10
a	b	r	a	c	a	d	a	b	r	a
d a b										
r ≠ b, r absent : +3										
d a b										
a ≠ b : +1										
d a b										
d ≠ b : +2										
d a b										
trouvé en 6										

On teste 4 positions au lieu de 7 pour trouver cette occurrence.

Définition 1 (Table de décalages) Pour chaque caractère c du motif, sauf le dernier, le **décalage** vaut $m - 1 - i$, où i est la position la plus à droite de c dans le motif, dernière position exclue. Un caractère absent du motif donne le décalage m .

Pour 'dab', la table est {'d': 2, 'a': 1}; pour 'maman', {'m': 2, 'a': 1}. On exclut le dernier caractère pour ne jamais obtenir un décalage nul, qui ferait boucler l'algorithme.

```
def pre_traitement(motif):
    m = len(motif)
    decalages = {}
    for i in range(m - 1):
        decalages[motif[i]] = m - 1 - i
    return decalages

def boyer_moore(texte, motif):
    n = len(texte)
    m = len(motif)
    decalages = pre_traitement(motif)
    indices = []
    i = 0
    while i <= n - m:
        j = m - 1
        while (j >= 0
              and texte[i + j] == motif[j]):
            j -= 1
        if j < 0:
            indices.append(i)
            i += 1
        else:
            c = texte[i + m - 1]
            if c in decalages:
                i += decalages[c]
            else:
                i += m
    return indices
```

Parcourir le motif de gauche à droite garde, pour chaque caractère, sa position la plus à droite : chaque nouvelle occurrence remplace la précédente.

④ COMPLEXITÉ

- **Meilleur cas** : les caractères du texte sont absents du motif, on décale de m à chaque fois. Il y a environ n/m comparaisons : la complexité est **sous-linéaire**, on trouve le motif sans lire tout le texte.
- **Pire cas** : $O(n \times m)$, comme l'algorithme naïf ; chercher 'baa' dans 'aaaaaa' décale toujours d'une position.
- **En pratique**, sur un texte en français, les décalages sont grands : l'algorithme est souvent trois à cinq fois plus rapide que l'algorithme naïf, d'autant plus que le motif est long.

À retenir.

- L'algorithme naïf teste chaque position, de gauche à droite : $O(n \times m)$ dans le pire cas.
- Boyer-Moore-Horspool compare de droite à gauche et saute des positions grâce à une table de décalages calculée à l'avance.
- Le décalage dépend du caractère du texte aligné avec la fin du motif.
- Dans le meilleur cas, la recherche demande environ n/m comparaisons.

Protocoles de routage

Fiche de synthèse

① RAPPELS DE PREMIÈRE

- Une adresse IPv4 compte 32 bits, écrits en quatre octets : 192.168.1.42. Le **masque** sépare la partie réseau de la partie hôte : /24, ou 255.255.255.0, réserve 24 bits au réseau.
- L'**adresse réseau** est le ET logique bit à bit de l'adresse et du masque : 172.16.5.130/16 appartient au réseau 172.16.0.0/16. Deux machines sont sur le même réseau local si elles ont la même adresse réseau.
- Le modèle TCP/IP a quatre couches : accès réseau (adresses MAC, trames), internet (IP, paquets), transport (TCP ou UDP, segments), application (HTTP, DNS). À l'envoi, chaque couche ajoute un en-tête : c'est l'**encapsulation**.
- Un **switch** relie les machines d'un réseau local par leurs adresses MAC ; un **routeur** relie plusieurs réseaux par les adresses IP, avec une interface dans chacun.

② TABLES DE ROUTAGE

Définition 1 (Table de routage) La **table de routage** d'un routeur associe à chaque réseau de destination le moyen de l'atteindre, interface de sortie ou routeur suivant appelé **passerelle**, et une **métrique**, qui mesure le coût de la route.

Réseau	Moyen de l'atteindre	Métrique
172.16.0.0/16	interface eth0	0
192.168.7.0/24	interface eth1	0
10.0.0.0/8	passerelle 192.168.7.2	1

Un réseau directement relié a la métrique 0. La **route par défaut** 0.0.0.0/0 reçoit toute destination absente de la table. Le **routage statique** est saisi à la main par l'administrateur ; le **routage dynamique** est calculé par des protocoles qui échangent des informations entre routeurs.

③ LE PROTOCOLE RIP

- RIP est un protocole à **vecteur de distances** : un vecteur de distance est un couple (destination, distance).
- Sa métrique est le **nombre de sauts**, c'est-à-dire de routeurs à traverser.
- Au départ, chaque routeur ne connaît que ses réseaux directs, à la distance 0. Toutes les 30 secondes, il envoie sa table complète à ses voisins directs.
- À la réception d'une table, il ajoute les réseaux inconnus avec la distance reçue plus 1, garde un chemin plus court,

et accepte un chemin plus long s'il vient du voisin déjà utilisé, car le réseau a changé. Une route non confirmée pendant 180 secondes est retirée.

- Après quelques échanges, les tables ne changent plus : le réseau a **convergé**. L'algorithme sous-jacent est celui de **Bellman-Ford**.

Chaque routeur ne connaît que ses voisins : il n'a jamais de vision globale du réseau. La distance est limitée à 15 sauts, 16 signifiant « inaccessible » ; les envois périodiques chargent le réseau ; la convergence est lente ; la qualité des liaisons est ignorée.

④ LE PROTOCOLE OSPF

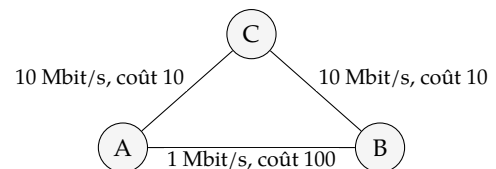
- OSPF est un protocole à **état de liens** : chaque routeur découvre ses voisins par des messages *hello*, toutes les 10 secondes, puis diffuse l'état de ses liaisons à **tous** les routeurs.
- Chaque routeur dispose ainsi d'une carte complète du réseau, un graphe pondéré, et calcule les routes de coût minimal avec l'algorithme de **Dijkstra**.
- Les mises à jour ne sont envoyées qu'en cas de changement.

Définition 2 (Coût d'une liaison OSPF)

$$\text{coût} = \frac{10^8}{\text{débit en bit/s}}$$

Le coût d'une route est la somme des coûts de ses liaisons.

Une liaison à 10 Mbit/s coûte $10^8/10^7 = 10$; à 100 Mbit/s, elle coûte 1.



Pour aller de A à B, RIP choisit la liaison directe, un seul saut, de coût 100. OSPF choisit A, C, B, de coût 10 + 10 = 20, malgré un saut de plus.

	RIP	OSPF
Métrique	nombre de sauts	coût fondé sur le débit
Vision	locale, les voisins	globale, tout le réseau
Algorithme	Bellman-Ford	Dijkstra
Mises à jour	toutes les 30 s	lors d'un changement
Limite	15 sauts	aucune limite de sauts
Réseaux	petits	moyens et grands

À retenir.

- Une table de routage donne, pour chaque réseau, la passerelle et la métrique.
- RIP compte les sauts et n'a qu'une vision locale ; il est limité à 15 sauts.
- OSPF calcule un coût égal à 10^8 divisé par le débit, et applique Dijkstra sur une carte complète du réseau.

Commandes réseau sous Linux

Fiche de synthèse

① VUE D'ENSEMBLE

Les commandes modernes, `ip` et `ss`, remplacent peu à peu les commandes historiques `ifconfig`, `netstat` et `route`, encore fréquentes dans les sujets.

Commande	Couche	Rôle
<code>ip a</code>	accès réseau	configuration des interfaces
<code>ping</code>	internet (ICMP)	tester la connectivité
<code>traceroute</code>	internet	chemin d'un paquet
<code>nslookup</code> , <code>dig</code>	application (DNS)	résoudre un nom de domaine
<code>ss</code>	transport	connexions et ports
<code>ip neigh</code> , <code>arp</code>	accès réseau (ARP)	table ARP

② CONFIGURATION : IP A

`ip a` affiche chaque interface avec son adresse MAC (après `link/ether`), son adresse IP et son masque (`inet 192.168.1.42/24`) et son adresse de diffusion (`brd 192.168.1.255`).

- `lo` est la boucle locale, d'adresse `127.0.0.1` : la machine communique avec elle-même.
- L'adresse de diffusion a tous les bits de la partie hôte à 1 ; un paquet qui lui est envoyé atteint toutes les machines du réseau local. L'adresse réseau a tous ces bits à 0.
- Ces deux adresses ne sont attribuées à aucune machine : avec h bits de partie hôte, un réseau accueille au plus $2^h - 2$ machines, soit 254 pour un masque `/24`.

③ CONNECTIVITÉ : PING

`ping -c 4 192.168.1.1` envoie quatre paquets ICMP et attend les réponses. Pour chacune, il affiche `icmp_seq`, numéro du paquet, `ttl`, nombre de routeurs que le paquet peut encore traverser, et `time`, temps d'aller-retour en millisecondes ; puis le taux de perte.

Message	Interprétation
Destination Host Unreachable	la machine ne répond pas sur son réseau local : éteinte, débranchée ou inexistante
Network is unreachable	aucune route vers la destination
100% packet loss	un pare-feu bloque, ou la machine distante est injoignable
perte partielle	liaison instable ou saturée

④ CHEMIN : TRACEROUTE

Chaque routeur diminue de 1 le TTL d'un paquet ; à 0, il le détruit et renvoie un message ICMP *Time Exceeded*. `traceroute` envoie un paquet avec un TTL de 1, détruit par le premier routeur, qui se fait connaître ; puis un TTL de 2, et ainsi de suite jusqu'à la destination. Chaque ligne affichée donne un routeur et trois temps d'aller-retour ; * * * signale un routeur qui ne répond pas.

⑤ NOMS DE DOMAINE : NSLOOKUP ET DIG

Le DNS traduit un nom de domaine en adresse IP, par le port UDP 53. `nslookup www.python.org` indique le serveur DNS qui a répondu et l'adresse obtenue ; *non-authoritative answer* signifie que la réponse vient du cache d'un serveur. `dig www.python.org +short` n'affiche que l'adresse ; sans `+short`, il détaille la réponse, dont le type d'enregistrement, `A` pour une adresse IPv4.

⑥ CONNEXIONS : SS

`ss -tln` liste les ports TCP en écoute, avec des numéros plutôt que des noms : par exemple 22 pour SSH, 80 pour HTTP, 631 pour l'impression.

⑦ TABLE ARP : IP NEIGH

Le protocole ARP trouve l'adresse MAC qui correspond à une adresse IP du réseau local. `ip neigh`, ou `arp -a`, affiche les correspondances mémorisées : *REACHABLE* si elle est récente et confirmée, *STALE* si elle est ancienne, *INCOMPLETE* si la requête ARP est restée sans réponse.

À retenir.

- `ip a` montre la configuration ; `ping` teste la connectivité ; `traceroute` exploite le TTL pour révéler les routeurs traversés.
- `nslookup` et `dig` interrogent le DNS ; `ss` liste les ports en écoute ; `ip neigh` affiche la table ARP.

Sécurisation des communications

Fiche de synthèse

① TROIS OBJECTIFS

Une communication sécurisée garantit la **confidentialité**, seuls l'émetteur et le destinataire lisent le message; l'**intégrité**, le message n'est pas modifié en chemin; l'**authentification**, chacun est sûr de l'identité de l'autre.

② CHIFFREMENT SYMÉTRIQUE

Définition 1 (Chiffrement symétrique) Dans un **chiffrement symétrique**, la même clé, partagée par l'émetteur et le destinataire, sert à chiffrer et à déchiffrer.

Il est rapide, mais il faut d'abord se transmettre la clé sans qu'un tiers l'intercepte.

```
def chiffre_cesar(message, decalage):
    resultat = ""
    for lettre in message:
        rang = ord(lettre) - ord("A")
        nouveau = (rang + decalage) % 26
        code = ord("A") + nouveau
        resultat = resultat + chr(code)
    return resultat

chiffre_cesar("NSI", 3)    # 'QVL'
```

Le chiffrement de César décale chaque lettre d'un même nombre de rangs; on déchiffre avec le décalage opposé. Il n'est pas sûr : il suffit d'essayer les 25 décalages possibles.

③ CHIFFREMENT ASYMÉTRIQUE

Définition 2 (Chiffrement asymétrique) Dans un **chiffrement asymétrique**, chaque personne possède une paire de clés : une **clé publique**, diffusée à tous, qui sert à chiffrer, et une **clé privée**, gardée secrète, qui seule permet de déchiffrer.

Pour écrire à Jeanne, Serge chiffre avec la clé publique de Jeanne; seule Jeanne, qui détient la clé privée, peut déchiffrer. Aucune clé secrète n'a besoin de circuler.

L'algorithme **RSA** repose sur une difficulté : multiplier deux grands nombres premiers p et q est facile, mais retrouver p et q à partir de leur produit n est extrêmement long. La clé publique est le couple (n, e) ; la clé privée, un nombre d , ne se calcule qu'à partir de p et q .

Il est beaucoup plus lent que le chiffrement symétrique et ne prouve pas l'identité de l'émetteur, puisque tout le monde connaît la clé publique.

④ SIGNATURE ÉLECTRONIQUE

≡ Signer et vérifier un message.

1. Serge calcule l'**empreinte** du message avec une **fonction de hachage** : un condensé du message, qui ne permet pas de le retrouver.
2. Il chiffre cette empreinte avec sa **clé privée** : c'est la signature, qu'il envoie avec le message.
3. Jeanne déchiffre la signature avec la clé publique de Serge, calcule elle-même l'empreinte du message reçu et compare les deux.

Si les empreintes sont égales, le message vient bien de Serge, seul détenteur de sa clé privée, et n'a pas été modifié : la signature garantit l'authentification et l'intégrité.

⑤ CERTIFICATS

Un tiers pourrait publier une clé publique au nom d'une autre personne et se faire passer pour elle : c'est l'**attaque de l'homme du milieu**. Une **autorité de certification** vérifie l'identité du propriétaire d'une clé publique, puis délivre un **certificat électronique** qu'elle signe avec sa propre clé privée. Les autorités et leurs certificats forment une **infrastructure à clés publiques**.

⑥ LE PROTOCOLE TLS

TLS, anciennement SSL, est un protocole **hybride** : il utilise le chiffrement asymétrique pour échanger une clé, puis le chiffrement symétrique, plus rapide, pour les données. HTTPS est HTTP sécurisé par TLS.

1. Le client demande une connexion sécurisée au serveur.
2. Le serveur envoie son certificat, qui contient sa clé publique.
3. Le client vérifie le certificat auprès de l'autorité de certification.
4. Le client crée une **clé de session** au hasard, la chiffre avec la clé publique du serveur et l'envoie.
5. Le serveur la déchiffre avec sa clé privée. Dans TLS 1.3, elle est plutôt calculée de part et d'autre, sans circuler.
6. Les échanges sont ensuite chiffrés de façon symétrique avec la clé de session.

Un attaquant qui intercepte l'étape 4 ne peut pas lire la clé de session sans la clé privée du serveur.

📌 À retenir.

- Symétrique : une clé partagée. Asymétrique : on chiffre avec la clé publique du destinataire, il déchiffre avec sa clé privée.

Systèmes d'exploitation et processus

Fiche de synthèse

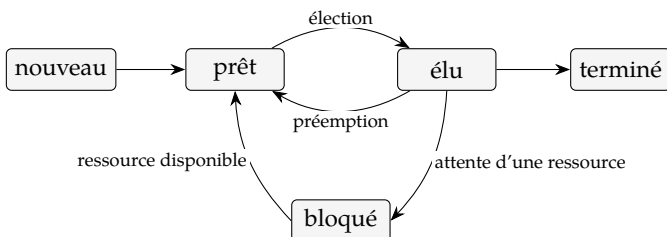
① PROGRAMME ET PROCESSUS

Le **système d'exploitation** fait le lien entre le matériel et les programmes : il gère les processus, les fichiers, la mémoire, les périphériques et la sécurité.

Définition 1 (Processus) Un **programme** est un fichier d'instructions, statique. Un **processus** est un programme en cours d'exécution, avec ses données, ses ressources et son état d'avancement. Lancer trois fois le même programme crée trois processus.

- Chaque processus a un **PID**, numéro unique, un **PPID**, le PID du processus père qui l'a créé, et un **UID**, l'identifiant de l'utilisateur qui l'a lancé.
- Son espace mémoire contient son code, ses variables globales, sa pile et son **contexte d'exécution**, copie des registres du processeur.
- Pour passer d'un processus à un autre, le système sauvegarde puis restaure les contextes en quelques microsecondes : les programmes semblent s'exécuter en même temps, c'est le **pseudo-parallélisme**.

② LES ÉTATS D'UN PROCESSUS



- Un processus **prêt** attend le processeur ; un processus **élu** s'exécute.
- Un processus élu qui attend une ressource, comme une saisie au clavier, passe à l'état **bloqué** ; quand la ressource est disponible, il redevient prêt, car un autre processus occupe le processeur.
- Un processus ne se termine que depuis l'état élu.

③ COMMANDES LINUX

Commande	Rôle
<code>ps -ef</code>	liste les processus, avec PID et PPID
<code>pstree</code>	affiche l'arbre des processus
<code>top</code>	suit les processus en temps réel
<code>kill PID</code>	demande l'arrêt d'un processus
<code>kill -9 PID</code>	force l'arrêt

④ ORDONNANCEMENT

L'**ordonnanceur** choisit quel processus prêt obtient le processeur.

- **FIFO** : premier arrivé, premier servi ; un processus long fait attendre tous les autres.
- **SJF** : le plus court d'abord ; il minimise le temps d'attente moyen mais exige de connaître les durées à l'avance.
- **Priorités** : les processus de faible priorité risquent de ne jamais être élus, c'est la **famine**.
- **Tourniquet** : chaque processus reçoit à son tour un **quantum** de temps ; s'il n'a pas fini, il repart en fin de file.

Définition 2 (Temps de séjour et temps d'attente) Le **temps de séjour** d'un processus est son temps de terminaison moins son temps d'arrivée ; son **temps d'attente** est son temps de séjour moins sa durée.

Cinq processus de durées 3, 6, 4, 2 et 1 arrivent aux instants 0, 1, 4, 6 et 7. Avec le tourniquet et un quantum de 2 :

P1	P2	P1	P3	P2	P4	P5	P3	P2	
0	2	4	5	7	9	11	12	14	16

Les temps d'attente valent 2, 9, 6, 3 et 4 : la moyenne est 4,8 quanta, contre 3,2 avec SJF. Le tourniquet est plus équitable et évite la famine. Linux combine quantum et priorités.

⑤ ACCÈS CONCURRENT

Deux processus testent qu'il reste un pion, puis le prennent. Si le premier est interrompu entre le test et la prise, le second prend le même pion : c'est un **accès concurrent** mal protégé. On protège la **section critique** par un **verrou** : un seul processus à la fois peut l'exécuter.

```

import threading

compteur = 0
verrou = threading.Lock()

def incrementer():
    global compteur
    for _ in range(10_000):
        with verrou:
            compteur = compteur + 1

threads = [threading.Thread(target=incrementer)
            for _ in range(10)]
for t in threads:
    t.start()
  
```

```
for t in threads:
    t.join()
print(compteur)    # 100000
```

Un **thread** est un fil d'exécution à l'intérieur d'un processus ; les threads d'un même processus partagent sa mémoire, d'où le besoin de verrous. `start` lance un thread, `join` attend sa fin.

⑥ INTERBLOCAGE

Définition 3 (Interblocage) Il y a **interblocage** quand des processus sont bloqués, chacun attendant une ressource détenue par un autre : P1 détient R1 et attend R2, P2 détient R2 et attend R1.

Selon Coffman (1971), un interblocage exige quatre conditions à la fois :

1. **exclusion mutuelle** : une ressource est libre ou attribuée à un seul processus ;
2. **détention et attente** : un processus qui détient une ressource en demande une autre ;
3. **non-préemption** : seul le processus qui détient une ressource peut la libérer ;
4. **attente circulaire** : un cycle de processus s'attendent les uns les autres.

Supprimer une seule condition rend l'interblocage impossible : par exemple, numéroté les ressources et imposer de les demander dans l'ordre croissant empêche toute attente circulaire.

📌 À retenir.

- Un processus est un programme en cours d'exécution, identifié par son PID.
- États : nouveau, prêt, élu, bloqué, terminé.
- Temps de séjour : terminaison moins arrivée ; temps d'attente : séjour moins durée.
- Un verrou protège une section critique contre les accès concurrents.
- L'interblocage exige les quatre conditions de Coffman.

Composants intégrés d'un système sur puce

Fiche de synthèse

① LE CIRCUIT INTÉGRÉ

Définition 1 (Circuit intégré) Un **circuit intégré**, ou puce électronique, regroupe des milliers à des milliards de composants électroniques, surtout des transistors, sur une plaquette de silicium de quelques millimètres carrés.

- Jack Kilby invente le premier circuit intégré en 1958, chez Texas Instruments.
- La **finesse de gravure** , en nanomètres, est la taille des plus petits transistors : plus elle est faible, plus la puce contient de transistors. Elle est passée de 10 μm en 1971 à 3 nm aujourd'hui.
- La **loi de Moore** (1965) constate que le nombre de transistors d'une puce double environ tous les deux ans. C'est une loi empirique, issue de l'observation, non une loi physique : elle se heurte aujourd'hui aux limites de la matière à l'échelle atomique.

De l'Intel 4004 (1971, 2 300 transistors) à l'Apple M4 (2024, 28 milliards), le nombre de transistors a été multiplié par plus de dix millions.

② TROIS NIVEAUX D'INTÉGRATION

	CPU	MCU	SoC
Processeur	oui	oui	oui
Mémoire	non	oui	oui
Entrées et sorties	non	oui	oui
GPU, radio, capteurs	non	non	oui
Consommation	élevée	faible	faible

Définition 2 (Microprocesseur) Un **microprocesseur** (CPU) contient le cœur de calcul d'un ordinateur : l'**unité arithmétique et logique** (UAL), qui calcule, l'**unité de contrôle**, qui dirige l'exécution des instructions, et des registres. Il a besoin d'une mémoire et de circuits d'entrée et de sortie extérieurs, reliés par la carte mère.

Ses caractéristiques : son **jeu d'instructions**, ensemble des opérations élémentaires qu'il exécute ; la **largeur du bus de données**, nombre de bits traités à la fois, 64 aujourd'hui ; sa **fréquence d'horloge**, 3 GHz pour trois milliards de cycles par seconde ; son **nombre de cœurs**, qui calculent en parallèle. Exemples : Intel Core, AMD Ryzen.

Définition 3 (Microcontrôleur) Un **microcontrôleur** (MCU) rassemble sur une puce un processeur peu puissant, de la mémoire (RAM pour les données, mémoire Flash pour le programme) et des périphériques d'entrée et de sortie : c'est un petit ordinateur autonome, dédié à une tâche.

On le trouve dans une machine à laver, le système ABS d'une voiture, une télécommande ou une carte Arduino.

Définition 4 (Système sur puce) Un **système sur puce** (SoC) intègre sur une seule puce un ou plusieurs processeurs, un processeur graphique (GPU), de la mémoire, des contrôleurs de communication (Wi-Fi, Bluetooth, 4G ou 5G) et des coprocesseurs spécialisés, par exemple pour l'intelligence artificielle.

Le SoC équipe les smartphones, les tablettes et les montres connectées : Apple M4, Snapdragon.

③ AVANTAGES ET LIMITES DE L'INTÉGRATION

- **Vitesse** : les composants sont à quelques millimètres les uns des autres, au lieu de plusieurs centimètres de pistes sur une carte mère ; les signaux arrivent plus vite et la **latence** diminue.
- **Consommation** : des connexions courtes demandent moins d'énergie, ce qui est essentiel pour un appareil sur batterie.
- **Dissipation thermique** : des milliards de transistors actifs sur quelques millimètres carrés produisent une chaleur difficile à évacuer.
- **Coût de conception** : plusieurs milliards de dollars et des années de développement.
- **Non-évolutivité** : on ne peut remplacer aucun composant seul ; il faut changer la puce entière, voire l'appareil.
- **Limites physiques** : à quelques nanomètres, des effets quantiques comme l'effet tunnel perturbent les transistors.

④ SYSTÈMES EMBARQUÉS

Définition 5 (Système embarqué) Un **système embarqué** est un système informatique autonome, intégré dans l'appareil qu'il contrôle et dédié à une tâche précise.

- Ses **ressources sont limitées** : mémoire, puissance de calcul, taille, énergie.

- Sa **fonction est spécialisée** : freinage ABS, régulation d'un thermostat, pilote automatique.
- Il doit être **fiable**, souvent en **temps réel** : un système de freinage réagit en quelques millisecondes.
- Son logiciel, le **micrologiciel** (*firmware*), est enregistré dans une mémoire non volatile et s'exécute dès la mise sous tension.

Il repose sur un microcontrôleur pour les systèmes simples et économes, sur un SoC pour les systèmes complexes comme un smartphone ou un drone.

À retenir.

- La loi de Moore décrit le doublement du nombre de transistors environ tous les deux ans.
- Le CPU contient l'UAL, l'unité de contrôle et les registres; le MCU y ajoute mémoire et entrées et sorties; le SoC y ajoute GPU, communications et co-processeurs.
- L'intégration apporte vitesse et faible consommation, au prix de la chaleur et de la non-évolutivité.
- Un système embarqué exécute une tâche spécialisée avec des ressources limitées, souvent en temps réel.

Calculabilité et décidabilité

Fiche de synthèse

① UN PROGRAMME EST UNE DONNÉE

Le code d'un programme est un texte : il peut être lu, transmis, analysé ou exécuté par un autre programme. L'interpréteur `python3` exécute le fichier `moyenne.py` ; un éditeur colore ses mots-clés ; un antivirus examine les fichiers exécutables ; le correcteur de Cypitale exécute un programme sur des tests.

```
def carre(x):
    return x * x

def verifier(fonction, entree, attendu):
    return fonction(entree) == attendu

verifier(carre, 4, 16)    # True
```

La fonction `carre` est transmise à `verifier`, qui l'exécute. Un programme peut même recevoir son propre code comme donnée.

② LE PROBLÈME DE L'ARRÊT

Certains programmes ne s'arrêtent pas : avec `dette = 10` et un remboursement de 3 par mois, la condition `dette != 0` reste toujours vraie. Pour d'autres, personne ne sait : on ignore si la boucle de la suite de Syracuse s'arrête pour tout entier de départ.

Théorème 1 (Arrêt, Turing, 1936) Il n'existe aucun programme capable de déterminer, pour tout programme et toute donnée, si ce programme s'arrête sur cette donnée. Le problème de l'arrêt est **indécidable**.

✂ **Démonstration par l'absurde.** Supposons qu'il existe une fonction `arret(programme, donnee)` qui renvoie `True` si l'appel `programme(donnee)` s'arrête, `False` sinon. On écrit :

```
def contraire(programme):
    if arret(programme, programme):
        while True:
            pass
    else:
        return "terminé"
```

Étudions l'appel `contraire(contraire)`.

- Si `arret` répond `True`, l'appel entre dans une boucle infinie : il ne s'arrête pas.
- Si `arret` répond `False`, l'appel renvoie `"terminé"` : il s'arrête.

Dans les deux cas, `arret` se trompe : la fonction `arret` ne peut pas exister.

Le théorème n'interdit pas de savoir si un programme particulier s'arrête : il affirme qu'aucune méthode automatique unique ne répond correctement pour tous les programmes. Il répond au problème de la décision posé par Hilbert en 1928.

③ LE THÉORÈME DE RICE

Théorème 2 (Rice, 1953) Toute propriété non triviale du **comportement** d'un programme est indécidable.

Une propriété porte sur le comportement quand elle concerne ce que fait le programme à l'exécution ; elle est non triviale quand certains programmes la vérifient et d'autres non. Aucun programme ne peut donc décider dans tous les cas si une fonction de tri trie toute liste, si un fichier est un virus, ou si deux fonctions renvoient toujours le même résultat. Une question sur le **texte** du programme reste décidable : « ce code contient-il une boucle `while` ? ». Les analyseurs de code et les antivirus ne donnent que des approximations.

④ CALCULABILITÉ

Définition 1 (Fonction calculable) Une fonction f est **calculable** s'il existe un algorithme qui, pour chaque entrée x , s'arrête après un nombre fini d'opérations élémentaires en donnant $f(x)$.

Le PGCD, le n -ième nombre premier ou le tri d'une liste sont calculables ; la fonction qui répond au problème de l'arrêt ne l'est pas. Il y a bien plus de fonctions que de programmes : la plupart des fonctions de $\mathbb{N}$ dans $\mathbb{N}$ ne sont pas calculables.

Définition 2 (Machine de Turing) Une **machine de Turing** comporte un ruban infini découpé en cases, une tête qui lit et écrit un symbole puis se déplace d'une case, un état courant pris dans un ensemble fini, et une table de transitions : pour chaque état et chaque symbole lu, elle donne le symbole à écrire, le déplacement et le nouvel état.

0	1	1	1	-	-
---	---	---	---	---	---

↑
tête, état q_0

La machine qui inverse les bits remplace chaque 0 par 1 et chaque 1 par 0 en avançant vers la droite, et s'arrête sur la première case vide. Partie du ruban 1011, elle a déjà inversé deux cases; elle finira sur 0100.

- Une **machine universelle** lit sur son ruban la description d'une autre machine et de sa donnée, puis la simule; un ordinateur exécute de même des programmes rangés en mémoire.
- **Thèse de Church-Turing** : les fonctions calculables par un algorithme sont exactement celles que calcule une machine de Turing. C'est une thèse, non un théorème : la notion d'algorithme n'a pas de définition mathématique.
- Un langage est **Turing-complet** s'il peut simuler toute machine de Turing : Python, C, JavaScript ou Scratch le sont, HTML seul ne l'est pas. Tous ces langages calculent les mêmes fonctions.

⑤ P, NP ET NP-COMPLET (HORS PROGRAMME)

- La classe **P** regroupe les problèmes résolus par un algorithme de complexité polynomiale, majorée par n^k pour un k fixé : trier une liste, chercher dans une liste triée, plus court chemin avec Dijkstra, test de primalité.
- La classe **NP**, pour non déterministe polynomial, regroupe les problèmes dont une solution proposée se **vérifie** en temps polynomial. NP ne signifie pas « non polynomial ».
- Exemples de NP : trouver les facteurs de 3 127 demande des essais, vérifier que $53 \times 59 = 3\,127$ une seule multiplication; colorier un graphe, organiser une tournée, remplir un sac à dos 0/1.
- $P \subset NP$: si l'on sait résoudre un problème, on vérifie une solution en le résolvant.
- Un problème **NP-complet** est un problème de NP auquel tout problème de NP se ramène en temps polynomial : SAT, sac à dos 0/1, voyageur de commerce, coloration avec trois couleurs. En résoudre un seul en temps polynomial prouverait que $P = NP$.

La question « $P = NP$? » est l'un des sept problèmes du prix du millénaire, doté d'un million de dollars.

📌 À retenir.

- Un programme est une donnée que d'autres programmes peuvent lire et exécuter.
- Le problème de l'arrêt est indécidable; la preuve par l'absurde utilise `contraire(contraire)`.
- Théorème de Rice : toute propriété non triviale du comportement d'un programme est indécidable.
- Une fonction est calculable si un algorithme la calcule en un nombre fini d'étapes; tous les langages Turing-complets calculent les mêmes fonctions.