MATHÉMATIQUES

25

Algorithmique appliquée

BTS | Mathématiques | Groupement D2

Objectifs du chapitre

- Analyser et écrire des algorithmes de recherche (séquentielle, dichotomique) et en comprendre la complexité.
- Implémenter et comparer les principaux algorithmes de tri (bulles, insertion, sélection, rapide).
- Maîtriser les structures de données linéaires : pile (LIFO), file (FIFO), liste chaînée.
- Comprendre la récursivité et l'appliquer à des problèmes classiques (factorielle, Fibonacci, Hanoï).
- Traiter des données : calcul de statistiques (moyenne, médiane, variance), recherche et filtrage.
- Écrire des fonctions Python documentées pour tous ces algorithmes.
- Analyser la complexité algorithmique avec la notation Big-O.

Situation professionnelle — Développeur en informatique industrielle

Contexte : Karim est technicien informatique dans une unité de production automobile. Sa mission : développer un programme de traitement des données de mesure collectées par les capteurs de contrôle qualité sur la ligne d'assemblage.

Chaque heure, 500 mesures de cotes de pièces (en mm) sont enregistrées dans un fichier CSV. Karim doit :

- Chercher si une pièce donnée dépasse la tolérance (+/- 0,5 mm) : algorithme de **recherche**.
- Trier les mesures pour calculer la médiane rapidement : algorithme de **tri**.
- Gérer une file d'attente des alertes à traiter : **structure de données**.
- Calculer la moyenne, l'écart-type et détecter les anomalies : **traitement statistique**.
- Optimiser le programme pour qu'il tourne en moins de 2 secondes : analyse de **complexité**.

Ce chapitre fournit les algorithmes et leur implémentation Python pour résoudre ces problèmes.

1. Rappels algorithmiques

Ce chapitre s'appuie sur les notions du chapitre 17 (variables, structures de contrôle, fonctions). Rappels essentiels en Python :

```
# Variables et types de base
n = 10           # entier
x = 3.14         # flottant
nom = "Alice"    # chaîne
tableau = [5, 3, 8, 1, 9, 2]  # liste (mutable, indexée à 0)

# Boucle for
for i in range(5):
    print(i)          # affiche 0, 1, 2, 3, 4

# Boucle while
```

```python
i = 0
while i < 5:
    i += 1

# Fonction avec docstring
def maxi(tableau):
    """Retourne le maximum d'une liste."""
    m = tableau[0]
    for x in tableau:
        if x > m:
            m = x
    return m
```

2. Algorithmes de recherche

2.1 Recherche séquentielle

Algorithme — Recherche séquentielle Parcourt le tableau de gauche à droite et compare chaque élément avec la valeur cible. Fonctionne sur tout tableau (trié ou non).

```python
def recherche_sequentielle(tableau, cible):
    """Retourne l'indice de cible, ou -1 si absent."""
    for i in range(len(tableau)):
        if tableau[i] == cible:
            return i
    return -1

# Exemple d'utilisation
t = [5, 3, 8, 1, 9, 2]
print(recherche_sequentielle(t, 8))  # → 2
print(recherche_sequentielle(t, 7))  # → -1
```

Complexité : $O(n)$ dans le pire cas (élément absent ou en fin de tableau).

2.2 Recherche dichotomique

Algorithme — Recherche dichotomique (binaire) **Prérequis :** le tableau doit être *trié*.
Principe : à chaque étape, on compare la cible avec l'élément médian et on restreint la recherche à la moitié restante pertinente.

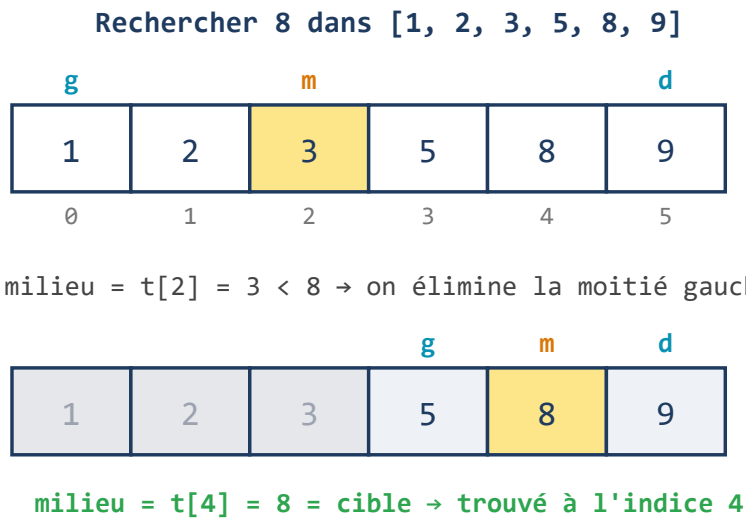
```python
def recherche_dichotomique(tableau, cible):
    """Retourne l'indice de cible dans un tableau trié, ou -1."""
    gauche, droite = 0, len(tableau) - 1
    while gauche <= droite:
        milieu = (gauche + droite) // 2
        if tableau[milieu] == cible:
            return milieu
        elif tableau[milieu] < cible:
            gauche = milieu + 1   # chercher à droite
        else:
            droite = milieu - 1  # chercher à gauche
    return -1

# Exemple : tableau trié
t = [1, 2, 3, 5, 8, 9]
print(recherche_dichotomique(t, 8))  # → 4
print(recherche_dichotomique(t, 7))  # → -1
```

Complexité : $O(\log_2 n)$. À chaque comparaison, la taille du problème est divisée par 2.

Comparaison chiffrée :

Pour un tableau de $n = 1\,000\,000$ éléments :

- Recherche séquentielle : jusqu'à 1 000 000 comparaisons
- Recherche dichotomique : $\lceil \log_2(1\,000\,000) \rceil = 20$ comparaisons au maximum



À chaque étape, on ne conserve que la moitié du tableau contenant la cible : la zone de recherche est divisée par 2.

3. Algorithmes de tri

3.1 Tri à bulles (Bubble Sort)

Algorithme — Tri à bulles Principe : parcourir le tableau en comparant des éléments adjacents et en les échangeant si nécessaire. Répéter jusqu'à ce qu'aucun échange ne soit effectué.

```
def tri_bulles(t):
    """Tri à bulles (modifie t en place)."""
    n = len(t)
    for i in range(n - 1):
        echange = False
        for j in range(n - 1 - i):
            if t[j] > t[j+1]:
                t[j], t[j+1] = t[j+1], t[j]
                echange = True
        if not echange:   # déjà trié → arrêt anticipé
            break
```

Complexité : $O(n^2)$ (pire et cas moyen), $O(n)$ (meilleur cas : déjà trié).

Trace d'exécution sur $[5, 3, 1, 4, 2]$ — passe 1 :

Étape	Tableau	Comparaison	Échange
1	[5, 3, 1, 4, 2]	5 > 3 ?	**Oui**
2	[3, 5, 1, 4, 2]	5 > 1 ?	**Oui**
3	[3, 1, 5, 4, 2]	5 > 4 ?	**Oui**
4	[3, 1, 4, 5, 2]	5 > 2 ?	**Oui**
Fin passe 1	[3, 1, 4, 2, 5]	—	Le 5 est en place

3.2 Tri par insertion

Algorithme — Tri par insertion Pour chaque élément $t[i]$, on l'insère à sa bonne position dans la partie déjà triée $t[0..i-1]$.

```
def tri_insertion(t):
    """Tri par insertion (modifie t en place)."""
    for i in range(1, len(t)):
        cle = t[i]
        j = i - 1
        while j >= 0 and t[j] > cle:
            t[j+1] = t[j]   # décaler vers la droite
            j -= 1
        t[j+1] = cle    # insérer à la bonne position
```

Complexité : $O(n^2)$ pire cas, $O(n)$ si tableau quasi-trié. Efficace pour les *petits tableaux* et les insertions en temps réel.

3.3 Tri par sélection

Algorithme — Tri par sélection À chaque tour, on sélectionne le minimum de la partie non triée et on l'échange avec le premier élément non trié.

```
def tri_selection(t):
    """Tri par sélection (modifie t en place)."""
    n = len(t)
    for i in range(n - 1):
        i_min = i
        for j in range(i + 1, n):
            if t[j] < t[i_min]:
                i_min = j
        t[i], t[i_min] = t[i_min], t[i]  # échange
```

Complexité : $O(n^2)$ dans tous les cas. Nombre d'échanges minimal : $O(n)$ — utile si les échanges sont coûteux.

3.4 Tri rapide (Quicksort)

Algorithme — Tri rapide Principe *diviser pour régner* :

1. Choisir un **pivot**.
2. Partitionner : éléments < pivot à gauche, éléments > pivot à droite.
3. Récursivement trier les deux partitions.

```
def tri_rapide(t, gauche=0, droite=None):
    """Tri rapide récursif (modifie t en place)."""
    if droite is None:
        droite = len(t) - 1
    if gauche < droite:
        pivot_idx = partition(t, gauche, droite)
        tri_rapide(t, gauche, pivot_idx - 1)
        tri_rapide(t, pivot_idx + 1, droite)

def partition(t, gauche, droite):
    """Partition autour du dernier élément (pivot)."""
    pivot = t[droite]
    i = gauche - 1
    for j in range(gauche, droite):
        if t[j] <= pivot:
            i += 1
            t[i], t[j] = t[j], t[i]
    t[i+1], t[droite] = t[droite], t[i+1]
    return i + 1
```

Complexité : $O(n \log n)$ en moyenne, $O(n^2)$ pire cas (tableau déjà trié avec mauvais pivot).

3.5 Comparaison des algorithmes de tri

Algorithme	Meilleur	Moyen	Pire	Stable ?	Utilisation typique
Tri à bulles	$O(n)$	$O(n^2)$	$O(n^2)$	Oui	Pédagogie, petits tableaux
Tri par insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Oui	Tableaux quasi-triés, petits n
Tri par sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$	Non	Peu d'échanges nécessaires
Tri rapide	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Non	Usage général, très efficace
Tri fusion	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Oui	Données externes, garantie
Python `sort()`	$O(n)$	$O(n \log n)$	$O(n \log n)$	Oui	Usage général (Timsort)

4. Structures de données linéaires

4.1 Pile (LIFO — Last In First Out)

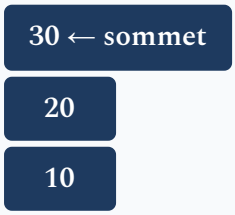
Définition — Pile Une **pile** est une structure de données où le dernier élément inséré est le premier extrait (principe LIFO).
Opérations : push (empiler), pop (dépiler), peek (lire le sommet).

```
# En Python, une liste s'utilise efficacement comme pile
pile = []
pile.append(10)    # push → pile = [10]
pile.append(20)    # push → pile = [10, 20]
pile.append(30)    # push → pile = [10, 20, 30]
sommet = pile.pop()  # pop  → sommet = 30, pile = [10, 20]

# Implémentation sous forme de classe
class Pile:
    def __init__(self):
        self._data = []
    def push(self, val): self._data.append(val)
    def pop(self):
        if self._data:
            return self._data.pop()
    def peek(self):
        return self._data[-1] if self._data else None
    def est_vide(self): return len(self._data) == 0
```

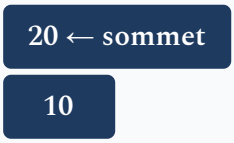
Applications : appels de fonctions (pile d'exécution), annulation (Ctrl+Z), vérification des parenthèses, évaluation d'expressions.

Visualisation d'une pile :

Après push(10), push(20), push(30) :



Après pop() (retourne 30) :



4.2 File (FIFO — First In First Out)

Définition — File Une **file** est une structure où le premier élément inséré est le premier extrait (FIFO). Opérations : enqueue (enfiler par l'arrière), dequeue (défiler par l'avant).

```
from collections import deque

file = deque()            # file vide
file.append("alerte_1")   # enqueue
file.append("alerte_2")   # enqueue
file.append("alerte_3")   # enqueue
premier = file.popleft()  # dequeue → "alerte_1"

# deque est O(1) pour popleft()
# list.pop(0) serait O(n) — à éviter pour les grandes files
```

Applications : gestion de files d'attente (serveurs, impressions), BFS, tampon E/S.

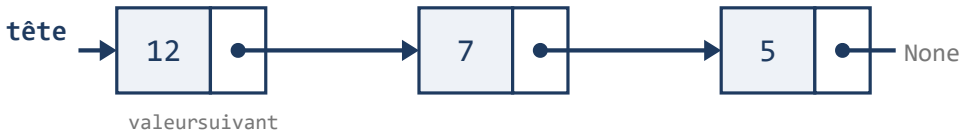
4.3 Liste chaînée

Définition — Liste chaînée Chaque **nœud** contient une valeur et un pointeur vers le nœud suivant. Contrairement à un tableau, les éléments ne sont pas contigus en mémoire.

```
class Noeud:
    def __init__(self, valeur):
        self.valeur = valeur
        self.suivant = None

class ListeChainee:
    def __init__(self): self.tete = None

    def inserer_debut(self, val):
        nouveau = Noeud(val)
        nouveau.suivant = self.tete
        self.tete = nouveau

    def parcourir(self):
        courant = self.tete
        while courant:
            print(courant.valeur, end=" -> ")
            courant = courant.suivant
        print("None")
```

Opération	Tableau (list)	Liste chaînée
Accès par indice	$O(1)$	$O(n)$
Insertion en tête	$O(n)$	$O(1)$
Insertion en fin	$O(1)^*$	$O(n)$ ou $O(1)$ avec pointeur fin
Suppression	$O(n)$	$O(1)$ si nœud connu

*amorti



5. Algorithmes récursifs

5.1 Principe de la récursivité

Définition — Récursivité Une fonction est **récursive** si elle s'appelle elle-même. Tout algorithme récursif comporte obligatoirement :

- Un ou plusieurs **cas de base** (condition d'arrêt) — sans appel récursif.
- Un ou plusieurs **appels récursifs** sur un problème de taille strictement plus petite.

Sans cas de base (ou si le cas de base n'est jamais atteint), on obtient une *récursion infinie* (équivalent d'une boucle infinie), qui provoque un dépassement de pile (*stack overflow*).

5.2 Factorielle

```
def factorielle(n):
    """Calcule n! récursivement. n doit être un entier >= 0."""
    if n == 0:          # cas de base : 0! = 1
        return 1
    return n * factorielle(n - 1)  # appel récursif

# Arbre d'appels pour factorielle(4) :
#  factorielle(4) = 4 × factorielle(3)
#                       = 3 × factorielle(2)
#                             = 2 × factorielle(1)
#                                   = 1 × factorielle(0) = 1
```

5.3 Suite de Fibonacci

```python
def fibonacci_recursif(n):
    """F(n) récursif — complexité O(2^n), à éviter pour grands n."""
    if n <= 1: return n
    return fibonacci_recursif(n-1) + fibonacci_recursif(n-2)

def fibonacci_iteratif(n):
    """F(n) itératif — complexité O(n), recommandé."""
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

def fibonacci_memo(n, cache={}):
    """F(n) avec mémoïsation — complexité O(n)."""
    if n in cache: return cache[n]
    if n <= 1: return n
    cache[n] = fibonacci_memo(n-1) + fibonacci_memo(n-2)
    return cache[n]
```

> **Attention — Fibonacci récursif naïf** La version récursive naïve recalcule les mêmes valeurs un très grand nombre de fois : $T(n) \approx 2^n$ opérations. Pour $n = 50$, c'est plus de 10^{15} appels. Toujours préférer la version itérative ou avec mémoïsation.

5.4 Tours de Hanoï

```python
def hanoi(n, source, cible, intermediaire):
    """Déplace n disques de source vers cible via intermediaire."""
    if n == 1:
        print(f"Déplacer disque 1 : {source} → {cible}")
        return
    hanoi(n-1, source, intermediaire, cible)
    print(f"Déplacer disque {n} : {source} → {cible}")
    hanoi(n-1, intermediaire, cible, source)

# hanoi(3, 'A', 'C', 'B')  → 7 déplacements
# Nombre de déplacements pour n disques = 2^n - 1
```

5.5 Récursion vs itération

Critère	Récursion	Itération
Lisibilité	Souvent plus claire	Variable
Mémoire	Pile d'appels ($O(n)$ minimum)	Constante si possible
Performance	Surcoût des appels	Généralement plus rapide
Risque	Stack overflow si trop profond	Boucle infinie
Idéal pour	Arbres, graphes, diviser-pour-régner	Boucles simples

6. Algorithmes de traitement de données

```
def statistiques(data):
    """Calcule moyenne, médiane, variance, min, max d'une liste."""
    n = len(data)
    if n == 0: return None

    # Minimum et maximum
    minimum = maximum = data[0]
    for x in data:
        if x < minimum: minimum = x
        if x > maximum: maximum = x

    # Moyenne
    somme = sum(data)
    moyenne = somme / n

    # Variance et écart-type
    variance = sum((x - moyenne)**2 for x in data) / n
    ecart_type = variance ** 0.5

    # Médiane (nécessite un tableau trié)
    trie = sorted(data)
    if n % 2 == 1:
        mediane = trie[n // 2]
    else:
```

```
        mediane = (trie[n//2 - 1] + trie[n//2]) / 2

    return {
        "n": n, "min": minimum, "max": maximum,
        "moyenne": moyenne, "médiane": mediane,
        "variance": variance, "écart_type": ecart_type
    }

def filtrer(data, predicate):
    """Retourne les éléments vérifiant predicate."""
    return [x for x in data if predicate(x)]

def compter(data, predicate):
    """Compte les éléments vérifiant predicate."""
    return sum(1 for x in data if predicate(x))
```

7. Complexité algorithmique

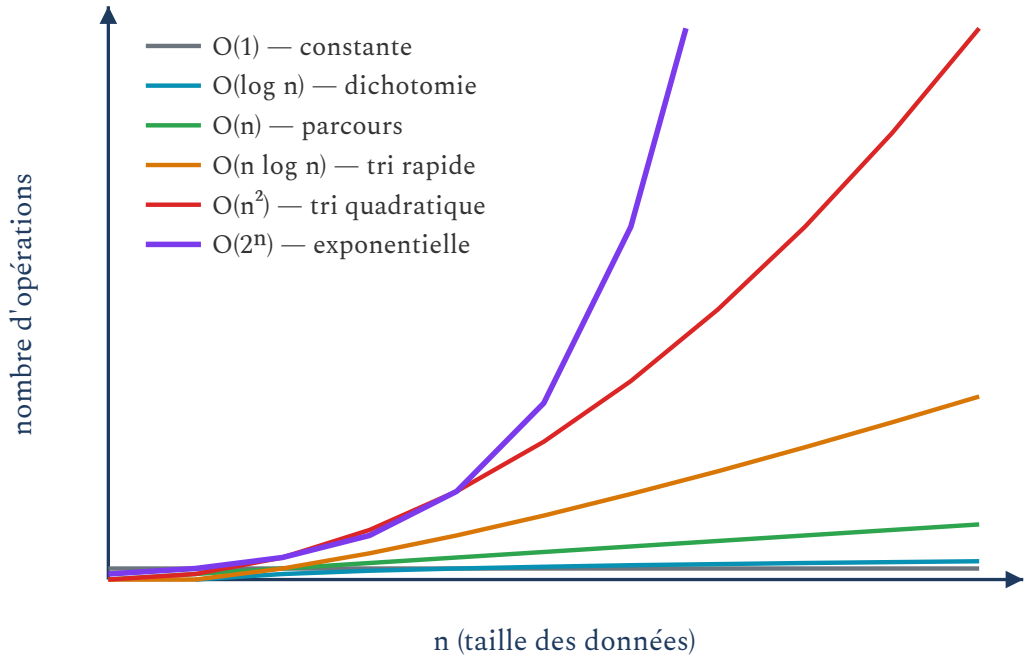
7.1 Notation Big-O

> Définition — Notation Big-O La **notation** $O(f(n))$ (Big-O) décrit la croissance asymptotique du nombre d'opérations d'un algorithme en fonction de la taille n de l'entrée.
>
> On dit que l'algorithme est $O(f(n))$ s'il existe des constantes $c > 0$ et n_0 telles que le nombre d'opérations $T(n) \leq c \cdot f(n)$ pour tout $n \geq n_0$.
>
> On ne garde que le terme *dominant* et on ignore les constantes : $3n^2 + 5n + 2 \in O(n^2)$.

$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$
Constante accès tableau par indice	Logarithmique dichotomie	Linéaire parcours, recherche séquentielle	Linéarithmique tri rapide, tri fusion

$O(n^2)$	$O(2^n)$

Quadratique	Exponentielle
tri à bulles, sélection	Fibonacci naïf, backtracking



Pour de grandes tailles n, l'écart entre les classes de complexité devient énorme : $O(2^n)$ et $O(n^2)$ explosent, tandis que $O(\log n)$ reste quasi plate.

7.2 Complexité temporelle et spatiale

Deux types de complexité

- **Complexité temporelle :** nombre d'opérations élémentaires en fonction de n. C'est la plus souvent analysée.
- **Complexité spatiale :** quantité de mémoire supplémentaire utilisée. Exemple : tri fusion $O(n)$ en espace supplémentaire ; tri en place $O(1)$.

On distingue le pire cas (O), le cas moyen (Θ) et le meilleur cas (Ω). En pratique, le *pire cas* est le plus utile pour garantir les performances.

Analyse de la recherche dichotomique :
À chaque étape, la taille du sous-tableau est divisée par 2. Après k étapes, la taille est $\frac{n}{2^k}$. On s'arrête quand $\frac{n}{2^k} \leq 1$, soit $k \geq \log_2 n$. Donc au maximum $\lceil \log_2 n \rceil$ comparaisons. Pour $n = 10^6$: $\lceil \log_2(10^6) \rceil = 20$ comparaisons. Bien plus efficace que $O(n)$.

8. Application — Traitement de données industrielles

Programme complet de traitement des données de cotes de pièces pour Karim. Il lit les mesures, les trie, calcule les statistiques et génère un rapport d'anomalies.

```
#!/usr/bin/env python3
"""
Contrôle qualité automatique — Analyse des cotes de fabrication.
Utilisation : python3 controle_qualite.py
"""
from collections import deque
import math

TOLERANCE = 0.5    # mm
COTE_NOMINALE = 100.0  # mm

def charger_mesures(fichier):
    """Simule le chargement de mesures depuis un fichier CSV."""
    # En production : with open(fichier) as f: ...
    import random
    random.seed(42)
    return [100.0 + random.gauss(0, 0.3) for _ in range(500)]

def detecter_anomalies(mesures, cote_nom, tol):
    """Retourne les indices et valeurs des mesures hors tolérance."""
    return [(i, v) for i, v in enumerate(mesures)
            if abs(v - cote_nom) > tol]

def statistiques(data):
    """Calcule les statistiques descriptives."""
    n = len(data)
    moy = sum(data) / n
    var = sum((x - moy)**2 for x in data) / n
```

```
    trie = sorted(data)
    med = trie[n//2] if n%2 else (trie[n//2-1]+trie[n//2])/2
    return {"n":n, "min":min(data), "max":max(data),
            "moy":moy, "med":med, "ecart":math.sqrt(var)}

def recherche_dichotomique_trie(trie, cible, tol):
    """Vérifie rapidement si une valeur proche de cible existe."""
    g, d = 0, len(trie)-1
    while g <= d:
        m = (g+d)//2
        if abs(trie[m]-cible) <= tol: return True
        elif trie[m] < cible: g = m+1
        else: d = m-1
    return False

def main():
    # 1. Chargement
    mesures = charger_mesures("mesures.csv")
    print(f"Mesures chargées : {len(mesures)} valeurs")

    # 2. Statistiques
    stats = statistiques(mesures)
    print(f"Moyenne : {stats['moy']:.4f} mm | Écart-type : {stats['ecart']:.4f} mm")
    print(f"Médiane : {stats['med']:.4f} mm | Min : {stats['min']:.4f} | Max : {stats[

    # 3. Détection des anomalies
    anomalies = detecter_anomalies(mesures, COTE_NOMINALE, TOLERANCE)
    print(f"Pièces hors tolérance : {len(anomalies)} / {len(mesures)}")

    # 4. File d'alertes FIFO
    alertes = deque()
    for idx, val in anomalies:
        alertes.append({"piece": idx+1, "cote": val,
                        "ecart": val-COTE_NOMINALE})

    print("\n=== Rapport d'alertes (file FIFO) ===")
    while alertes:
        a = alertes.popleft()
        print(f"  Pièce #{a['piece']:4d} : {a['cote']:.4f} mm (écart {a['ecart']:+.4f}

if __name__ == "__main__":
    main()
```

Méthode — Analyser et écrire un algorithme

1. **Comprendre le problème :** données d'entrée, résultat attendu, contraintes.
2. **Choisir la structure de données :** accès rapide → tableau ; insertion/suppression fréquentes → liste ou deque ; LIFO → pile ; FIFO → file.
3. **Choisir l'algorithme :** si données triées → dichotomique ; si petit tableau → tri insertion ; si grand tableau → tri rapide ou `sorted()`.
4. **Écrire le pseudo-code** avant de coder (préconditions, postconditions).
5. **Implémenter en Python** avec docstring et noms de variables explicites.
6. **Tester sur des cas limites :** tableau vide, un élément, éléments identiques, déjà trié, trié à l'envers.
7. **Analyser la complexité :** déterminer $O(\cdot)$ et décider si c'est acceptable pour le volume de données.

À retenir — Algorithmique appliquée

Recherche et tri

- Séquentielle : $O(n)$, tout tableau
- Dichotomique : $O(\log n)$, tableau trié
- Tri quadratique : bulles, insertion, sélection — $O(n^2)$
- Tri rapide (quicksort) : $O(n \log n)$ moyen
- Python `sorted()` / `.sort()` : Timsort $O(n \log n)$

Structures et récursivité

- Pile (LIFO) : `list.append()` / `list.pop()`
- File (FIFO) : `deque.append()` / `deque.popleft()`
- Récursivité : cas de base obligatoire + réduction du problème
- Fibonacci : naïf $O(2^n)$, mémoïsation/itératif $O(n)$
- Complexité : $O(1) \ll O(\log n) \ll O(n) \ll O(n \log n) \ll O(n^2) \ll O(2^n)$

Règle d'or : ne jamais utiliser `list.pop(0)` dans une boucle (c'est $O(n)$ par appel, donc $O(n^2)$ au total) — utiliser `collections.deque` à la place.

MATHÉMATIQUES

25

Algorithmique appliquée

Algorithmique appliquée | BTS | Mathématiques

Rappels : recherche séquentielle (jusqu'à n comparaisons) ; recherche dichotomique sur un tableau trié (environ $\log_2 n$ comparaisons). Complexité : un parcours simple est en $O(n)$, une double boucle en $O(n^2)$.

Exercice 1 — Recherche séquentielle

On cherche une valeur dans un tableau de 50 éléments par recherche séquentielle.

1. Combien de comparaisons au maximum (pire cas) ?

2. En moyenne (valeur présente, position quelconque) ?

Exercice 2 — Recherche dichotomique

Un tableau **trié** contient 1024 éléments.

Combien de comparaisons au maximum avec une recherche dichotomique ?

Exercice 3 — Tri par sélection

Applique le tri par sélection (on place le minimum en tête à chaque étape) au tableau $[5,\ 2,\ 4,\ 1]$. Donne les états successifs.

Exercice 4 — Récursivité

On définit la factorielle par $f(0) = 1$ et $f(n) = n \times f(n-1)$.

Déroule le calcul de $f(4)$.

Exercice 5 — Complexité (type BTS)

Un algorithme parcourt un tableau de n éléments avec deux boucles imbriquées (pour chaque élément, on reparcourt tout le tableau).

1. Combien d'opérations en fonction de n ? **2.** Quelle est sa complexité ?

MATHÉMATIQUES

25

Algorithmique appliquée

Algorithmique appliquée | BTS | Mathématiques

🕐 **Durée :** 1 heure 🧮 **Calculatrice :** autorisée ✍️ **Barème :** 20 points

📄 **Documents :** non autorisés

Exercice 1 — Recherches *(8 points)*

1. Dans un tableau de 200 éléments, combien de comparaisons au pire pour une recherche séquentielle ? *(2 pts)*

2. Le tableau est trié et contient 256 éléments : combien au pire pour une recherche dichotomique ? *(3 pts)*

3. Pourquoi la dichotomie exige-t-elle un tableau trié ? *(3 pts)*

Exercice 2 — Récursivité *(6 points)*

On définit $s(0) = 0$ et $s(n) = n + s(n-1)$ (somme des entiers de 1 à n).

1. Calcule $s(5)$ en déroulant. *(3 pts)*

2. Donne une formule directe de $s(n)$. *(3 pts)*

Exercice 3 — Tri et complexité *(6 points)*

1. Trie $[4,\ 1,\ 3,\ 2]$ par sélection (donne le tableau final). *(3 pts)*

2. Quelle est la complexité d'un tri par sélection sur n éléments ? *(3 pts)*