

ALGORITHMIQUE ET PROGRAMMATION

Notion d'algorithme et de pseudo-code — Classe de Seconde générale

Résumé de cours rigoureux, suivi d'exercices corrigés classés par niveau de difficulté (simple, moyen, difficile), avec justification des choix et rédaction détaillée des corrections. Conforme au programme officiel de mathématiques de Seconde générale (algorithmique et programmation).

I. Résumé de cours

1. Qu'est-ce qu'un algorithme ?
2. Variables et affectation
3. Instruction conditionnelle
4. Boucles : Pour et Tant que
5. Fonctions
6. Correspondance pseudo-code / Python

II. Exercices — Niveau simple (n°1 à 3)

III. Exercices — Niveau moyen (n°4 à 6)

IV. Exercices — Niveau difficile (n°7 à 9)

I. RÉSUMÉ DE COURS

1. Qu'est-ce qu'un algorithme ?

Définition. Un *algorithme* est une suite finie et non ambiguë d'instructions permettant de résoudre un problème ou d'obtenir un résultat, à partir de données d'entrée, en un nombre fini d'étapes.

Un algorithme se distingue d'un programme : l'algorithme décrit la **méthode** de résolution de façon indépendante d'un langage informatique particulier, tandis que le programme est sa **traduction** dans un langage exécutable par une machine (ici, principalement Python). Le **pseudo-code** est une écriture intermédiaire, structurée mais non formelle, qui permet de décrire un algorithme de façon lisible par un humain avant sa traduction en code.

Trois exigences caractérisent un bon algorithme : la **correction** (il doit produire le résultat attendu pour toute entrée valide), la **terminaison** (il doit s'arrêter après un nombre fini d'étapes) et la **lisibilité** (sa structure doit être compréhensible).

2. Variables et affectation

Définition. Une *variable* est un identifiant associé à un emplacement mémoire contenant une valeur, susceptible d'évoluer au cours de l'exécution de l'algorithme. L'*affectation*, notée $\text{variable} \leftarrow \text{expression}$, consiste à calculer la valeur de l'expression puis à la stocker dans la variable.

En Seconde, on manipule essentiellement trois types de données : les **nombres** (entiers ou flottants), les **booléens** (Vrai / Faux, issus d'un test de comparaison) et les **chaînes de caractères**. On utilise également des **listes** pour stocker une collection ordonnée de valeurs.

Remarque essentielle sur l'affectation

Le symbole $\leftarrow$ n'est **pas** un signe d'égalité mathématique : l'instruction $a \leftarrow a + 1$ n'a pas de sens en algèbre (aucun nombre ne vérifie $a = a + 1$), mais est parfaitement licite en algorithmique : on calcule d'abord la valeur actuelle de a , on lui ajoute 1, puis on range le résultat dans a , qui « écrase » l'ancienne valeur.

```
Algorithme : Échange de deux variables
Entrée : deux nombres a et b
Début
    temp  $\leftarrow$  a
    a  $\leftarrow$  b
    b  $\leftarrow$  temp
Fin
Sortie : a et b (valeurs échangées)
```

Justification : la variable auxiliaire `temp` est indispensable : si l'on écrivait directement $a \leftarrow b$ puis $b \leftarrow a$, la valeur initiale de a serait perdue avant d'être recopiée dans b .

3. Instruction conditionnelle

Syntaxe. L'instruction conditionnelle permet d'exécuter des instructions différentes selon qu'une *condition* (expression booléenne) est vraie ou fausse.

```
Si condition Alors
    instructions si condition vraie
Sinon
    instructions si condition fausse
Fin Si
```

La branche `Sinon` est facultative. On peut enchaîner plusieurs conditions avec `Sinon Si` pour traiter plus de deux cas. Les opérateurs de comparaison usuels sont : $=$, $\neq$, $<$, $\leq$, $>$, $\geq$; les opérateurs logiques ET, OU, NON permettent de combiner plusieurs conditions.

4. Boucles : Pour et Tant que

On distingue deux structures de répétition, dont le choix dépend de la nature du problème.

a) Boucle « Pour » (nombre d'itérations connu à l'avance)

```
Pour i allant de a à b (pas de 1)
    instructions
Fin Pour
```

Le compteur *i* prend successivement les valeurs *a*, *a*+1, ..., *b* : la boucle s'exécute exactement *b* – *a* + 1 fois. On utilise cette structure lorsque le nombre de répétitions est connu avant l'exécution (par exemple, parcourir tous les éléments d'une liste de taille fixée).

b) Boucle « Tant que » (nombre d'itérations inconnu a priori)

```
Tant que condition Faire
    instructions
Fin Tant que
```

La condition est évaluée **avant** chaque itération ; tant qu'elle est vraie, le bloc d'instructions est exécuté. On l'utilise lorsque le nombre de répétitions dépend d'un résultat calculé en cours d'exécution (recherche d'un seuil, saisie utilisateur, etc.).

Point de vigilance (terminaison). Dans une boucle « Tant que », il est impératif qu'au moins une variable intervenant dans la condition soit modifiée à chaque itération, afin de garantir que la condition finira par devenir fausse. Dans le cas contraire, l'algorithme ne termine jamais : on parle de **boucle infinie**, ce qui constitue une erreur grave de conception.

5. Fonctions

Une **fonction** est un sous-algorithme nommé, qui prend éventuellement des **paramètres** en entrée et peut renvoyer une valeur grâce à l'instruction **Retourner**. Elle permet de découper un problème complexe en sous-problèmes réutilisables (principe de modularité).

```
Fonction carre(x)
    Retourner x * x
Fin Fonction

Variable resultat
resultat ← carre(5) # resultat vaut 25
```

6. Correspondance pseudo-code / Python

Pseudo-code	Python
variable ← valeur	variable = valeur
Si ... Alors ... Sinon ... Fin Si	if ... : \n ... \nelse : \n ...
Pour i de a à b Faire ... Fin Pour	for i in range(a, b+1): \n ...
Tant que condition Faire ... Fin Tant que	while condition : \n ...
Fonction f(x) ... Retourner y Fin Fonction	def f(x): \n ... \n return y
ET / OU / NON	and / or / not
Afficher(x)	print(x)

Convention adoptée dans ce document : tous les algorithmes sont rédigés en pseudo-code, puis une traduction Python est proposée dans les corrections lorsqu'elle apporte un éclairage supplémentaire.

II. EXERCICES — NIVEAU SIMPLE

Exercice 1. Parité d'un nombre entier

SIMPLE

Écrire en pseudo-code un algorithme qui demande à l'utilisateur un nombre entier n , puis affiche « Pair » si n est pair et « Impair » sinon.

CORRECTION DÉTAILLÉE

Analyse préalable. Un entier n est pair si et seulement si le reste de la division euclidienne de n par 2 est nul, c'est-à-dire $n \bmod 2 = 0$. On dispose ainsi d'un test booléen direct, qui ne nécessite ni boucle ni fonction : une simple instruction conditionnelle suffit.

```
Algorithme : Parité
Entrée : un entier n
Début
    Si  $n \bmod 2 = 0$  Alors
        Afficher("Pair")
    Sinon
        Afficher("Impair")
    Fin Si
Fin
```

Justification de la structure choisie. La condition $n \bmod 2 = 0$ est une expression booléenne à deux issues exclusives (vraie ou fausse) ; l'instruction « Si ... Sinon » est donc exactement adaptée, car exactement une des deux branches doit s'exécuter, quel que soit n .

Traduction Python :

```
n = int(input("Entrez un entier : "))
if n % 2 == 0:
    print("Pair")
else:
    print("Impair")
```

Vérification sur un exemple : pour $n = 7$, on calcule $7 \bmod 2 = 1 \neq 0$, donc la condition est fausse et l'algorithme affiche « Impair », ce qui est conforme au résultat attendu.

Exercice 2. Somme des n premiers entiers

SIMPLE

Écrire un algorithme qui, étant donné un entier naturel n , calcule et affiche la somme $S = 1 + 2 + \dots + n$ à l'aide d'une boucle.

CORRECTION DÉTAILLÉE

Analyse préalable. Le nombre de termes à additionner (n) est connu avant l'exécution : on doit donc privilégier une boucle « Pour », qui parcourt exactement les entiers de 1 à n . On initialise un accumulateur S à 0 avant la boucle, car une somme vide doit valoir 0 (élément neutre de l'addition).

```
Algorithme : Somme des n premiers entiers
Entrée : un entier naturel n
Variables : S (somme), i (compteur)
Début
     $S \leftarrow 0$ 
    Pour i allant de 1 à n Faire
         $S \leftarrow S + i$ 
    Fin Pour
    Afficher(S)
Fin
```

Justification de l'initialisation. Si l'on omettait l'instruction $S \leftarrow 0$, la variable S contiendrait une valeur indéterminée (ou celle d'une exécution précédente), et le résultat final serait faux : l'initialisation de

l'accumulateur est une étape non négociable de ce type d'algorithme.

Vérification pour $n = 4$ (déroulé pas à pas) :

i	S avant	S après ($S \leftarrow S + i$)
1	0	1
2	1	3
3	3	6
4	6	10

On obtient $S = 10$, ce qui coïncide avec la formule connue $S = n(n+1)/2 = 4 \times 5 / 2 = 10$: le résultat de l'algorithme est ainsi validé par un calcul indépendant.

Traduction Python :

```
n = int(input("Entrez n : "))
S = 0
for i in range(1, n + 1):
    S = S + i
print(S)
```

Exercice 3. Maximum de deux nombres

SIMPLE

Écrire un algorithme qui demande deux nombres a et b et affiche le plus grand des deux (on suppose $a \neq b$ pour cet exercice).

CORRECTION DÉTAILLÉE

Analyse préalable. Il s'agit à nouveau d'un test à deux issues, portant cette fois sur une comparaison entre deux variables plutôt que sur une propriété d'une seule variable.

```
Algorithme : Maximum de deux nombres
Entrée : deux nombres a et b ( $a \neq b$ )
Début
    Si  $a > b$  Alors
        Afficher(a)
    Sinon
        Afficher(b)
    Fin Si
Fin
```

Justification. L'hypothèse $a \neq b$ garantit que les deux branches sont bien exclusives et exhaustives : soit $a > b$, soit $b > a$, il n'existe pas de troisième cas. Si l'on autorisait $a = b$, l'algorithme resterait correct (il afficherait b , valeur commune), mais cette précision mérite d'être mentionnée pour la rigueur de la preuve.

III. EXERCICES — NIVEAU MOYEN

Exercice 4. Recherche du maximum d'une liste

MOYEN

On considère une liste L de n nombres ($n \geq 1$). Écrire un algorithme qui détermine et affiche la plus grande valeur de la liste.

CORRECTION DÉTAILLÉE

Analyse préalable. Le nombre d'éléments à parcourir est connu (c'est la taille de la liste) : une boucle « Pour » est donc adaptée. La méthode classique consiste à maintenir un accumulateur maxi contenant le plus grand élément rencontré jusqu'ici, et à le mettre à jour à chaque comparaison. Il est essentiel d'initialiser maxi avec le **premier élément de la liste** (et non avec 0, par exemple), car si tous les éléments de L étaient négatifs, une initialisation à 0 produirait un résultat faux.

```
Algorithme : Maximum d'une liste
Entrée : une liste L de n nombres (n ≥ 1), indexée de 0 à n-1
Variables : maxi, i
Début
    maxi ← L[0]
    Pour i allant de 1 à n-1 Faire
        Si L[i] > maxi Alors
            maxi ← L[i]
        Fin Si
    Fin Pour
    Afficher(maxi)
Fin
```

Justification de la borne de la boucle. La boucle démarre à $i = 1$ (et non $i = 0$) car l'élément d'indice 0 a déjà été pris en compte lors de l'initialisation de maxi ; le comparer à nouveau à lui-même serait inutile (bien que sans conséquence sur le résultat).

Vérification pour $L = [3, 7, 2, 9, 5]$ ($n = 5$) :

i	L[i]	L[i] > maxi ?	maxi après
init	—	—	3
1	7	7 > 3 : Vrai	7
2	2	2 > 7 : Faux	7
3	9	9 > 7 : Vrai	9
4	5	5 > 9 : Faux	9

On obtient bien $\text{maxi} = 9$, qui est effectivement la plus grande valeur de L .

Traduction Python :

```
L = [3, 7, 2, 9, 5]
maxi = L[0]
for i in range(1, len(L)):
    if L[i] > maxi:
        maxi = L[i]
print(maxi)
```

Exercice 5. Table de multiplication conditionnelle

MOYEN

Écrire un algorithme qui affiche, pour i allant de 1 à 10, le produit $i \times 7$, mais uniquement lorsque ce produit est un nombre pair.

CORRECTION DÉTAILLÉE

Analyse préalable. Cet exercice combine deux structures étudiées séparément dans les exercices précédents : une boucle « Pour » (nombre d'itérations fixé à 10) imbriquant une instruction conditionnelle (test de parité). L'ordre d'imbrication est imposé par l'énoncé : le test de parité doit être réévalué à chaque itération, il se trouve donc **à l'intérieur** de la boucle.

```
Algorithme : Produits pairs de la table de 7
Variables : i, p (produit)
Début
    Pour i allant de 1 à 10 Faire
        p ← i * 7
        Si p mod 2 = 0 Alors
            Afficher(p)
        Fin Si
    Fin Pour
Fin
```

Justification. Comme 7 est impair, le produit $i \times 7$ a la même parité que i (produit d'un nombre pair par un nombre impair = nombre pair ; produit de deux nombres impairs = nombre impair). L'algorithme affichera donc les produits correspondant aux valeurs paires de i , soit $i \in \{2, 4, 6, 8, 10\}$, c'est-à-dire 14, 28, 42, 56, 70. Cette prédiction théorique permet de valider la sortie du programme sans avoir à l'exécuter.

Traduction Python :

```
for i in range(1, 11):
    p = i * 7
    if p % 2 == 0:
        print(p)
```

Exercice 6. Calcul d'une racine carrée entière par tâtonnement

MOYEN

Écrire un algorithme qui, pour un entier naturel n donné, détermine le plus grand entier k tel que $k^2 \leq n$, sans utiliser de fonction racine carrée.

CORRECTION DÉTAILLÉE

Analyse préalable. Le nombre d'itérations nécessaires n'est pas connu à l'avance : il dépend de n (il faudra plus d'étapes pour un grand n). On utilise donc une boucle « Tant que », qui incrémente un compteur k tant que $(k+1)^2$ reste inférieur ou égal à n . Il faut vérifier la **terminaison** : la variable k est strictement croissante et bornée par n , donc la condition finit nécessairement par devenir fausse.

```
Algorithme : Racine carrée entière (partie entière de  $\sqrt{n}$ )
Entrée : un entier naturel n
Variables : k
Début
    k ← 0
    Tant que (k + 1) * (k + 1) ≤ n Faire
        k ← k + 1
    Fin Tant que
    Afficher(k)
Fin
```

Justification de la condition d'arrêt. On souhaite le plus grand k tel que $k^2 \leq n$. On teste donc si le *candidat suivant*, $k + 1$, vérifie encore $(k+1)^2 \leq n$: si oui, on peut avancer ; si non, k est déjà le plus grand entier convenable et la boucle s'arrête. Cette formulation évite de « dépasser » la valeur cherchée.

Vérification pour $n = 20$: on cherche le plus grand k avec $k^2 \leq 20$. Déroulé : $k=0 \rightarrow (1)^2=1 \leq 20$ vrai $\rightarrow k=1$; $(2)^2=4 \leq 20$ vrai $\rightarrow k=2$; $(3)^2=9 \leq 20$ vrai $\rightarrow k=3$; $(4)^2=16 \leq 20$ vrai $\rightarrow k=4$; $(5)^2=25 \leq 20$ faux $\rightarrow$ arrêt. On obtient $k = 4$, et l'on vérifie $4^2 = 16 \leq 20 < 25 = 5^2$: le résultat est correct.

Traduction Python :

```
n = int(input("Entrez n : "))
k = 0
while (k + 1) * (k + 1) <= n:
    k = k + 1
print(k)
```

IV. EXERCICES — NIVEAU DIFFICILE

Exercice 7. Test de primalité

DIFFICILE

Écrire un algorithme qui teste si un entier $n \geq 2$ est premier, en limitant le nombre de divisions effectuées de façon pertinente (on ne demande pas la méthode la plus optimisée, mais une méthode justifiée).

CORRECTION DÉTAILLÉE

Analyse préalable. Par définition, n est premier si ses seuls diviseurs positifs sont 1 et n . Une méthode naïve consisterait à tester tous les diviseurs d de 2 à $n - 1$. On peut toutefois restreindre la recherche : si n possède un diviseur d avec $2 \leq d \leq n - 1$, alors n s'écrit $n = d \times d'$ avec $d' = n/d$. Or, on ne peut pas avoir simultanément $d > \sqrt{n}$ et $d' > \sqrt{n}$ (sinon $d \times d' > n$). Il existe donc nécessairement un diviseur de n inférieur ou égal à $\sqrt{n}$. **Il suffit donc de tester les diviseurs de 2 à $\sqrt{n}$ pour conclure.**

On utilise une variable booléenne `est_premier` initialisée à Vrai, que l'on passe à Faux dès qu'un diviseur est trouvé. On parcourt les diviseurs candidats avec une boucle « Pour » (borne connue à l'avance : de 2 à $\sqrt{n}$).

```
Algorithme : Test de primalité
Entrée : un entier  $n \geq 2$ 
Variables : est_premier (booléen), d, borne
Début
    Si  $n = 2$  Alors
        est_premier  $\leftarrow$  Vrai
    Sinon
        est_premier  $\leftarrow$  Vrai
        borne  $\leftarrow$  partie_entiere(racine_carree(n))
        Pour d allant de 2 à borne Faire
            Si  $n \bmod d = 0$  Alors
                est_premier  $\leftarrow$  Faux
            Fin Si
        Fin Pour
    Fin Si
    Si est_premier Alors
        Afficher("premier")
    Sinon
        Afficher("non premier")
    Fin Si
Fin
```

Justification du cas particulier $n = 2$. Pour $n = 2$, on a $\sqrt{2} = 1$, ce qui rendrait la boucle « Pour d de 2 à 1 » vide (borne inférieure supérieure à la borne supérieure) : l'algorithme conclurait correctement que 2 est premier même sans ce cas particulier, mais on l'isole ici par clarté pédagogique et pour éviter toute ambiguïté sur le comportement d'une boucle à intervalle vide.

Remarque sur l'optimisation. Cette borne $\sqrt{n}$ est une amélioration essentielle par rapport à la méthode naïve (borne $n - 1$) : pour $n = 1\,000\,000$, on passe de 999 998 tests potentiels à 1 000 seulement, ce qui illustre concrètement l'intérêt d'une analyse mathématique préalable pour concevoir un algorithme efficace.

Vérification pour $n = 37$: $\sqrt{37} = 6$ (car $6^2 = 36 \leq 37 < 49 = 7^2$). On teste $d = 2, 3, 4, 5, 6$: aucun ne divise 37 (37 est impair, non multiple de 3, ne se termine pas par 0 ou 5, $37/6$ n'est pas entier). Donc `est_premier` reste Vrai : 37 est bien premier, ce que l'on peut vérifier indépendamment.

Traduction Python :

```
import math

n = int(input("Entrez n (n >= 2) : "))
```

```

if n == 2:
    est_premier = True
else:
    est_premier = True
    borne = int(math.sqrt(n))
    for d in range(2, borne + 1):
        if n % d == 0:
            est_premier = False

print("premier" if est_premier else "non premier")

```

Exercice 8. Tri par sélection

DIFFICILE

Écrire un algorithme qui trie une liste L de n nombres par ordre croissant, en utilisant la méthode du *tri par sélection* : à chaque étape, on recherche le minimum parmi les éléments non encore triés et on l'échange avec le premier élément de cette zone restante.

CORRECTION DÉTAILLÉE

Analyse préalable. Le tri par sélection procède en $n - 1$ étapes. À l'étape i (i variant de 0 à $n - 2$), on cherche l'indice du minimum parmi les éléments $L[i]$, $L[i+1]$, ..., $L[n-1]$, puis on échange $L[i]$ avec ce minimum. Après l'étape i , les $i+1$ premiers éléments de la liste sont définitivement triés et sont les plus petits éléments de L . On imbrique donc **deux boucles « Pour »** : la boucle externe fixe la position i à traiter, la boucle interne recherche le minimum sur la zone restante — cette recherche de minimum reprend directement la méthode de l'exercice 4.

```

Algorithme : Tri par sélection
Entrée : une liste L de n nombres, indexée de 0 à n-1
Variables : i, j, indice_min, temp
Début
    Pour i allant de 0 à n-2 Faire
        indice_min ← i
        Pour j allant de i+1 à n-1 Faire
            Si L[j] < L[indice_min] Alors
                indice_min ← j
            Fin Si
        Fin Pour
        # Échange de L[i] et L[indice_min]
        temp ← L[i]
        L[i] ← L[indice_min]
        L[indice_min] ← temp
    Fin Pour
Fin
Sortie : la liste L, triée par ordre croissant

```

Justification de la borne « n-2 » de la boucle externe. Lorsque les $n - 1$ premiers éléments sont à leur place définitive, le dernier élément l'est nécessairement aussi (c'est le seul restant) : il est donc inutile de traiter $i = n - 1$, ce qui justifie l'arrêt à $n - 2$.

Justification de l'échange conditionnel : si $\text{indice_min} = i$ (le minimum est déjà en première position de la zone restante), l'échange $L[i] \leftrightarrow L[\text{indice_min}]$ est un échange d'un élément avec lui-même, qui ne modifie pas la liste : l'algorithme reste correct même dans ce cas, sans qu'il soit nécessaire d'ajouter un test supplémentaire.

Vérification pour $L = [5, 2, 4, 1]$ ($n = 4$) :

Étape i	Zone examinée	indice_min trouvé	L après échange
-----------	---------------	-------------------	-----------------

0	[5,2,4,1]	3 (valeur 1)	[1, 2, 4, 5]
1	[2,4,5]	0 → indice 1 (valeur 2)	[1, 2, 4, 5]
2	[4,5]	indice 2 (valeur 4)	[1, 2, 4, 5]

La liste finale [1, 2, 4, 5] est bien triée par ordre croissant, ce qui valide l'algorithme sur cet exemple.

Traduction Python :

```
L = [5, 2, 4, 1]
n = len(L)
for i in range(n - 1):
    indice_min = i
    for j in range(i + 1, n):
        if L[j] < L[indice_min]:
            indice_min = j
    L[i], L[indice_min] = L[indice_min], L[i]
print(L)
```

Exercice 9. Recherche dichotomique

DIFFICILE

On considère une liste L de n nombres **triée par ordre croissant**. Écrire un algorithme qui détermine si une valeur v appartient à L, en utilisant la méthode de la *recherche dichotomique* (recherche par dichotomie), et affiche « Trouvé » ou « Absent ».

CORRECTION DÉTAILLÉE

Analyse préalable. Le caractère trié de la liste permet, à chaque étape, d'éliminer environ la moitié des éléments restants : on compare v à l'élément central de la zone de recherche [gauche ; droite]. Si v est égal à cet élément, la recherche est terminée avec succès. Si v est strictement inférieur, tous les éléments à partir du centre (inclus) peuvent être écartés car ils sont trop grands (liste croissante) : on poursuit la recherche à gauche. Symétriquement, si v est strictement supérieur, on poursuit à droite. Le nombre d'itérations n'est pas connu à l'avance (il dépend de la position de v) : on emploie donc une boucle « Tant que », avec une condition d'arrêt combinant deux critères : la zone de recherche n'est pas vide (gauche ≤ droite), **et** la valeur n'a pas encore été trouvée.

```
Algorithme : Recherche dichotomique
Entrée : une liste L triée par ordre croissant (indices 0 à n-1), une valeur v
Variables : gauche, droite, milieu, trouve (booléen)
Début
    gauche ← 0
    droite ← n - 1
    trouve ← Faux
    Tant que gauche ≤ droite ET NON trouve Faire
        milieu ← partie_entiere((gauche + droite) / 2)
        Si L[milieu] = v Alors
            trouve ← Vrai
        Sinon
            Si L[milieu] < v Alors
                gauche ← milieu + 1
            Sinon
                droite ← milieu - 1
        Fin Si
    Fin Tant que
    Si trouve Alors
        Afficher("Trouvé")
    Sinon
        Afficher("Absent")
```

Fin Si
Fin

Justification de la terminaison. À chaque itération où v n'est pas trouvé, soit gauche augmente strictement (cas $L[\text{milieu}] < v$, car gauche devient $\text{milieu} + 1 > \text{milieu} \geq \text{gauche}$), soit droite diminue strictement (cas symétrique). La quantité (droite – gauche) est donc strictement décroissante à chaque itération sans succès, et l'algorithme s'arrête nécessairement dès que gauche > droite si la valeur n'est pas présente, ou plus tôt si elle l'est. La terminaison est ainsi garantie dans tous les cas.

Justification de l'exigence « liste triée ». C'est cette hypothèse, et elle seule, qui autorise à éliminer toute une moitié de la liste sans l'examiner : sur une liste non triée, rien ne permettrait d'affirmer que les éléments écartés ne contiennent pas la valeur v . L'énoncé impose donc cette hypothèse comme condition **nécessaire** à la validité de l'algorithme.

Vérification pour $L = [1, 3, 4, 7, 9, 12, 15]$ ($n = 7$) et $v = 9$:

Itération	gauche	droite	milieu	$L[\text{milieu}]$	Décision
1	0	6	3	7	$7 < 9 \rightarrow \text{gauche} \leftarrow 4$
2	4	6	5	12	$12 > 9 \rightarrow \text{droite} \leftarrow 4$
3	4	4	4	9	égal à $v \rightarrow$ trouvé

La valeur $v = 9$ est trouvée en 3 itérations seulement (contre, potentiellement, 5 comparaisons pour une recherche linéaire), ce qui illustre l'efficacité de la méthode dichotomique : le nombre d'itérations nécessaires est de l'ordre de $\log_2(n)$, à comparer à n pour une recherche élément par élément.

Traduction Python :

```
L = [1, 3, 4, 7, 9, 12, 15]
v = 9
gauche, droite = 0, len(L) - 1
trouve = False
while gauche <= droite and not trouve:
    milieu = (gauche + droite) // 2
    if L[milieu] == v:
        trouve = True
    elif L[milieu] < v:
        gauche = milieu + 1
    else:
        droite = milieu - 1

print("Trouvé" if trouve else "Absent")
```