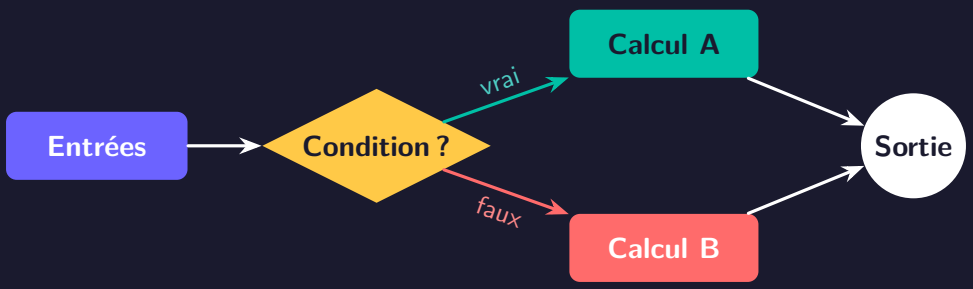
FICHE 16

Algorithmique et Python

Comprendre, écrire et vérifier

Variables ▪ Boucles ▪ Fonctions



Seconde générale ▪ Mathématiques ▪ Programme officiel

Cours complet | 15 exercices | Problème défi | Corrigés

Table des matières

« Un programme juste est un raisonnement rendu exécutable. »

1 Pourquoi étudier l'algorithmique ?

Un ordinateur calcule très vite, mais il ne devine rien. Pour lui faire résoudre un problème, il faut décrire une suite d'instructions sans ambiguïté. Cette description s'appelle un **algorithme**. Python est un langage qui permet de la rendre exécutable.

Intuition | Un algorithme est une recette qui doit fonctionner pour toutes les entrées prévu

Une recette de cuisine peut dire « ajoute un peu d'eau ». Un algorithme ne peut pas se contenter de « un peu » : il doit préciser la valeur, l'ordre des opérations et les cas particuliers. Programmer apprend donc à **raisonner avec précision**.

Définition | Algorithme et programme

Un **algorithme** est une méthode finie et non ambiguë qui transforme des **entrées** en **sorties**. Un **programme** est la traduction de cet algorithme dans un langage compris par une machine.

Exemple | Calculer le prix après une réduction

Entrées : un prix p et un taux t en pourcentage. Instruction : calculer $p(1 - t/100)$. Sortie : le prix réduit. Pour $p = 80$ et $t = 15$, on obtient $80(1 - 0{,}15) = 68$.

Attention | Le programme ne remplace pas l'explication

Un résultat affiché n'est pas une démonstration. Tu dois savoir ce que calcule le programme, pourquoi il le calcule et dans quelles conditions le résultat est valable.

2 Du langage naturel à Python

2.1 Lire un programme de haut en bas

Sauf instruction particulière, Python exécute les lignes dans l'ordre. Les espaces au début d'une ligne, appelés **indentation**, indiquent les blocs appartenant à une condition, une boucle ou une fonction.

Exemple | Un premier programme

```
prix = 80
taux = 15
prix_reduit = prix * (1 - taux / 100)
print(prix_reduit)
```

La sortie est `68.0`. Le point indique que Python manipule ici un nombre décimal de type `float`.

Méthode | Traduire une consigne

1. repère les données d'entrée et donne-leur des noms explicites ;
2. écris les calculs dans leur ordre logique ;
3. prévois les choix avec `if` et les répétitions avec une boucle ;
4. choisis ce que le programme doit afficher ou renvoyer ;
5. teste un cas simple dont tu connais déjà la réponse.

2.2 Afficher n'est pas renvoyer

Définition | Deux actions différentes

`print(...)` **affiche** une information à l'écran. `return ...` **renvoie** une valeur depuis une fonction afin qu'elle puisse être utilisée dans un autre calcul.

Exemple | Pourquoi la différence compte

```
def carre(x):
    return x ** 2

y = carre(5) + 1
print(y)
```

La fonction renvoie 25, puis le programme calcule $25 + 1$ et affiche 26.

3 Variables, types et affectations

3.1 Une variable informatique est une boite nommée

Définition | Affectation

L'instruction `x = expression` calcule d'abord l'expression située à droite, puis range le résultat dans la variable `x`. En langage naturel, on peut écrire $x \leftarrow$ expression.

Intuition | Le signe = n'est pas une égalité mathématique

En mathématiques, $x = x + 1$ est impossible. En programmation, `x = x + 1` signifie : « prends l'ancienne valeur de `x`, ajoute 1, puis remplace l'ancienne valeur par le résultat ».

Exemple | Table de trace

```
x = 4
y = 3 * x
x = x + 2
y = y - x
```

Après la ligne	x	y
1	4	non définie
2	4	12
3	6	12
4	6	6

Une table de trace permet de suivre exactement l'état des variables.

Démonstration | Pourquoi l'ordre des lignes change le résultat

Dans l'exemple, la ligne `y = 3 * x` utilise $x = 4$, donc elle mémorise 12. Lorsque x devient ensuite 6, Python ne recalcule pas automatiquement y. Une affectation conserve le résultat obtenu **au moment où la ligne est exécutée**.

3.2 Les quatre types essentiels

Type Python	Nom	Exemple	Valeur
`int`	entier	`n = 12`	12
`float`	flottant	`x = 2.5`	approximation décimale
`str`	chaine de caractères	`mot = "lycee"`	texte
`bool`	booléen	`ok = x > 0`	`True` ou `False`

Méthode | Connaitre le type

La fonction `type(x)` indique le type de `x`. Les conversions usuelles sont `int(x)`, `float(x)` et `str(x)`. Une conversion n'est possible que si le contenu s'y prête.

Attention | Les flottants sont des approximations

De nombreux décimaux n'ont pas d'écriture binaire finie. Python peut donc produire, par exemple, une valeur très proche de 0,3 sans être exactement 0,3. Pour comparer deux résultats calculés, on teste souvent `abs(a - b) < 1e-9` plutôt que `a == b`.

4 Calculs, comparaisons et booléens

4.1 Opérateurs numériques

Python	Mathématiques	Exemple
`+`, `-`, `*`, `/`	$+, -, \times, \div$	`7 / 2` donne `3.5`
`**`	puissance	`3 ** 2` donne `9`
`//`	quotient entier	`17 // 5` donne `3`
`%`	reste	`17 % 5` donne `2`

Propriété | Division euclidienne

Pour des entiers $a \geqslant 0$ et $b > 0$, si

$$q = \texttt{a // b} \qquad \text{et} \qquad r = \texttt{a \% b},$$

alors $a = bq + r$ et $0 \leqslant r < b$.

Démonstration | Vérification sur $a = 47$ et $b = 6$

Python donne $q = 47 \,//\, 6 = 7$ et $r = 47 \,\%\, 6 = 5$. On vérifie

$$47 = 6 \times 7 + 5 \qquad \text{et} \qquad 0 \leqslant 5 < 6.$$

Le test `n % d == 0` permet donc de savoir si d divise n.

4.2 Comparaisons et connecteurs logiques

Python	Sens	Python	Sens
`==`	est égal à	`!=`	est différent de
`<`, `<=`	inférieur, inférieur ou égal	`>`, `>=`	supérieur, supérieur ou égal
`and`	et : deux conditions vraies	`or`	ou : au moins une vraie
`not`	négation		

Exemple | Appartenance à un intervalle

La condition mathématique $2 \leqslant x < 7$ s'écrit directement

```
2 <= x < 7
```

La condition « n est un multiple positif de 3 inférieur à 20 » s'écrit

```
n > 0 and n < 20 and n % 3 == 0
```

Propriété | Négation d'une condition

La négation de `x >= 2 and x < 7` est `x < 2 or x >= 7`. Le mot `and` devient `or` et chaque comparaison est niée.

Démonstration | Pourquoi

Être en dehors de l'intervalle $[2;7[$ signifie être soit strictement avant 2, soit à partir de 7. Ainsi

$$\text{non}(2 \leqslant x < 7) \text{ si et seulement si } x < 2 \text{ ou } x \geqslant 7.$$

Cette traduction empêche les erreurs de frontière.

5 Choisir avec `if`, `elif`, `else`

Définition | Instruction conditionnelle

Une condition permet d'exécuter un bloc seulement lorsqu'une proposition est vraie.

```
if condition:
    instructions_si_vraie
else:
    instructions_si_fausse
```

Les deux-points et l'indentation sont obligatoires.

Exemple | Tarif selon l'âge

```
def tarif(age):
    if age < 0:
        return "age impossible"
    elif age < 12:
        return 5
    elif age < 18:
        return 8
    else:
        return 12
```

Pour `age = 15`, le premier test est faux, le deuxième est faux, le troisième est vrai : la fonction renvoie 8 et Python ne lit pas le dernier `else`.

Méthode | Étudier les frontières

Pour vérifier un tarif par tranches, teste une valeur au milieu de chaque tranche et toutes les frontières : ici $-1, 0, 11, 12, 17, 18$. Les erreurs se cachent souvent dans un `<` qui aurait dû être `<=`.

Démonstration | Pourquoi les cas couvrent toutes les possibilités

Après avoir écarté $age < 0$, les intervalles $[0;12[$, $[12;18[$ et $[18;+\infty[$ sont disjoints et leur réunion est $[0;+\infty[$. Une entrée admissible appartient donc à une unique tranche : une seule valeur est renvoyée.

Attention | Deux `if` ne sont pas toujours un `if/elif`

Deux blocs `if` séparés sont testés tous les deux. Dans une chaine `if/elif/else`, Python s'arrête au premier test vrai. Choisis selon que plusieurs actions peuvent ou non se produire.

6 Répéter un nombre connu de fois : la boucle `for`

6.1 Comprendre `range`

Définition | Boucle bornée

```
for i in range(debut, fin, pas):
    instructions
```

La variable `i` prend les valeurs de `debut` à `fin` **exclue**, en avançant de `pas`. Si le début vaut 0 et le pas vaut 1, on peut les omettre.

Instruction	Valeurs prises par `i`	Nombre de tours
`range(5)`	0, 1, 2, 3, 4	5
`range(2, 6)`	2, 3, 4, 5	4
`range(1, 10, 2)`	1, 3, 5, 7, 9	5
`range(6, 1, -1)`	6, 5, 4, 3, 2	5

Attention | La borne finale est exclue

Pour faire varier i de 1 à n inclus, écris `range(1, n + 1)`. C'est l'une des erreurs les plus fréquentes.

6.2 Accumuler une somme

Exemple | Somme des entiers de 1 à n

```
def somme_entiers(n):
    s = 0
    for k in range(1, n + 1):
        s = s + k
    return s
```

Pour $n = 4$, la variable `s` prend successivement les valeurs 0, 1, 3, 6, 10.

Démonstration | Pourquoi cet algorithme est correct

Avant la boucle, $s = 0$. Après le tour correspondant à k, la variable s contient

$$1 + 2 + \cdots + k.$$

En effet, si elle contient $1+\cdots+(k-1)$ avant le tour, l'affectation `s = s + k` lui ajoute précisément k. Au dernier tour, $k = n$: la fonction renvoie donc $1 + 2 + \cdots + n$. Cette propriété conservée au fil des tours s'appelle un **invariant de boucle**.

6.3 Compter des valeurs qui vérifient une condition

Exemple | Nombre de multiples de 7 entre 1 et n

```
def compte_multiples_7(n):
    compteur = 0
    for k in range(1, n + 1):
        if k % 7 == 0:
            compteur = compteur + 1
    return compteur
```

L'accumulateur `s` additionne des valeurs ; le compteur ajoute seulement 1 lorsqu'une condition est satisfaite.

7 Répéter jusqu'à un seuil : la boucle `while`

Définition | Boucle non bornée

```
while condition:
    instructions
```

Tant que la condition est vraie, le bloc est répété. Le nombre de tours n'est pas fixé à l'avance.

Exemple | Premier entier dont le carré dépasse un seuil

```
def premier_carre_superieur(seuil):
    n = 0
    while n ** 2 <= seuil:
        n = n + 1
    return n
```

Pour `seuil = 30`, la boucle s'arrête à $n = 6$, car $5^2 \leqslant 30$ mais $6^2 > 30$.

Démonstration | Correction et arrêt

Pendant la boucle, n augmente de 1 : il finira donc par avoir un carré supérieur au seuil si celui-ci est positif. Au moment où la boucle s'arrête, la condition `n ** 2 <= seuil` est fausse, donc $n^2 > \textit{seuil}$.

Au tour précédent, on avait encore $(n-1)^2 \leqslant \textit{seuil}$. Ainsi, n est bien le **premier** entier qui convient.

Méthode | Les trois questions d'une boucle `while`

1. Quelle condition doit rester vraie pour continuer ?
2. Quelle variable change à chaque tour ?
3. Pourquoi cette évolution rend-elle finalement la condition fausse ?

Si tu ne peux pas répondre à la troisième question, la boucle risque d'être infinie.

Attention | Ne pas confondre continuer et s'arrêter

Si l'on cherche le premier n tel que $u_n \geqslant 100$, la boucle doit continuer tant que $u_n < 100$. La condition du `while` décrit ce qui est encore insuffisant, pas le résultat final.

8 Construire des fonctions réutilisables

Définition | Fonction Python

Une fonction possède un nom, zéro ou plusieurs **paramètres**, un bloc d'instructions et, le plus souvent, une valeur renvoyée par `return`.

```
def nom(parametre1, parametre2):
    instructions
    return resultat
```

Exemple | Distance entre deux nombres

```
def distance(a, b):
    return abs(a - b)
```

L'appel `distance(3, 8)` renvoie 5. Les nombres 3 et 8 sont les **arguments** transmis aux paramètres `a` et `b`.

Propriété | Une seule tâche clairement nommée

Une bonne fonction accomplit une tâche précise, porte un nom explicite et renvoie une valeur au type attendu. Il est alors possible de la tester séparément et de la réutiliser.

Exemple | Une fonction qui appelle une autre fonction

```
def aire_disque(r):
    return 3.141592653589793 * r ** 2

def aire_couronne(r1, r2):
    return aire_disque(r2) - aire_disque(r1)
```

Pour $0 \leqslant r_1 \leqslant r_2$, on soustrait l'aire du petit disque à celle du grand.

Attention | Après `return`, la fonction est terminée

Dès qu'un `return` est exécuté, Python quitte la fonction. Une ligne placée ensuite dans le même bloc ne sera jamais lue.

9 Simuler une expérience aléatoire

9.1 Une réalisation n'est pas une probabilité

Définition | Fonctions aléatoires utiles

Après `from random import random, randint` :

— `random()` renvoie un flottant pseudo-aléatoire dans $[0; 1[$;

— `randint(a, b)` renvoie un entier pseudo-aléatoire entre a et b, bornes incluses.

Exemple | Simuler un succès de probabilité p

```
from random import random

def bernoulli(p):
    if random() < p:
        return 1
    return 0
```

Comme la longueur de l'intervalle $[0; p[$ est p, la fonction renvoie 1 avec probabilité p.

Démonstration | Pourquoi le test `random()` < `p` modélise la probabilité p

On suppose les valeurs de `random()` uniformément réparties dans $[0; 1[$. L'évènement « la valeur est dans $[0; p[$ » occupe une longueur p sur une longueur totale 1. Sa probabilité vaut donc $p/1 = p$.

9.2 Répéter pour obtenir une fréquence

Exemple | Estimer une probabilité

```
def frequence_succes(p, n):
    succes = 0
    for k in range(n):
        succes = succes + bernoulli(p)
    return succes / n
```

Pour un grand n, le résultat est généralement proche de p, mais il varie d'une exécution à l'autre.

Attention | Une simulation ne prouve pas une valeur exacte

Elle aide à conjecturer, à vérifier l'ordre de grandeur d'un calcul ou à étudier un phénomène difficile. Elle ne remplace pas un raisonnement lorsqu'un calcul exact est possible.

10 Lire une fonction statistique

Le programme demande de savoir lire une fonction renvoyant une moyenne ou un écart type, sans exiger la connaissance des listes. Il faut donc comprendre le rôle des étapes, même si certaines écritures seront fournies.

Exemple | Moyenne d'une série

```
def moyenne(valeurs):
    somme = 0
    for x in valeurs:
        somme = somme + x
    return somme / len(valeurs)
```

`len(valeurs)` est le nombre de données. Si `valeurs = [8, 11, 15, 10]`, la fonction renvoie $(8+11+15+10)/4 = 11$.

Démonstration | Formule traduite ligne par ligne

Pour n valeurs $x_1, \dots, x_n$, la boucle construit $S = x_1 + \cdots + x_n$. La dernière ligne renvoie donc

$$\frac{S}{n} = \frac{x_1 + \cdots + x_n}{n},$$

qui est exactement la moyenne.

Exemple | Écart type : comprendre sans mémoriser le code

```
from math import sqrt

def ecart_type(valeurs):
    m = moyenne(valeurs)
    somme = 0
    for x in valeurs:
        somme = somme + (x - m) ** 2
    return sqrt(somme / len(valeurs))
```

Le programme calcule la moyenne, les écarts à la moyenne, leurs carrés, leur moyenne, puis la racine carrée.

11 Tester, expliquer et corriger un programme

Méthode | Les quatre familles d'erreurs

— **Syntaxe :** Python ne comprend pas le texte, par exemple un deux-points oublié.

— **Exécution :** une opération impossible survient, par exemple une division par zéro.

— **Logique :** le programme s'exécute mais répond faux.

— **Modèle :** le code correspond au calcul prévu, mais le calcul ne décrit pas correctement la situation.

Exemple | Une erreur de logique

On veut tester si n est pair.

```
def est_pair(n):
    return n % 2 == 1
```

Le code s'exécute, mais renvoie `True` pour les impairs. Il faut remplacer `== 1` par `== 0`.

Méthode | Un plan de tests efficace

1. un cas ordinaire calculé à la main ;
2. chaque valeur frontière ;
3. une valeur minimale, nulle ou négative si elle est autorisée ;
4. un cas où la réponse doit être fausse ;
5. un cas assez grand pour détecter une boucle mal contrôlée.

« Un programme juste est un raisonnement rendu exécutable. »

Propriété | Un test révèle une erreur, il ne prouve pas l'absence d'erreur

Même cent tests réussis ne couvrent pas nécessairement toutes les entrées. Pour établir qu'un programme est correct, on explique aussi pourquoi ses instructions réalisent le raisonnement attendu.

12 Pour aller plus loin : raisonner sur les algorithmes

Intuition | Approfondissement ouvert à toutes et tous

Cette section dépasse le socle minimal, mais chaque idée est reconstruite pas à pas. Elle ne demande aucun talent caché : seulement de suivre l'état des variables et de vérifier ce qui reste vrai.

12.1 L'invariant : une preuve qui accompagne la boucle

Définition | Invariant

Un **invariant de boucle** est une propriété vraie avant la boucle et qui reste vraie après chaque tour. À la sortie, il permet de relier l'état final au résultat recherché.

Exemple | Produit des entiers de 1 à n

```
def produit_entiers(n):
    p = 1
    for k in range(1, n + 1):
        p = p * k
    return p
```

Après le tour d'indice k, l'invariant est $p = 1 \times 2 \times \cdots \times k$. L'initialisation à 1 est indispensable : partir de 0 rendrait tous les produits nuls.

« Un programme juste est un raisonnement rendu exécutable. »

12.2 La dichotomie : éliminer la moitié à chaque étape

On cherche une valeur approchée de $\sqrt{2}$, donc un nombre x tel que $x^2 = 2$. On sait que $1^2 < 2 < 2^2$.

Méthode | Principe

On conserve un encadrement $a^2 \leqslant 2 \leqslant b^2$. On calcule le milieu $m = (a+b)/2$:

— si $m^2 < 2$, on remplace a par m ;

— sinon, on remplace b par m.

La longueur de l'intervalle est divisée par 2 à chaque tour.

```
def racine2(epsilon):
    a = 1.0
    b = 2.0
    while b - a > epsilon:
        m = (a + b) / 2
        if m ** 2 < 2:
            a = m
        else:
            b = m
    return (a + b) / 2
```

Démonstration | Pourquoi l'erreur est contrôlée

L'invariant $a \leqslant \sqrt{2} \leqslant b$ est conservé par le choix de la moitié qui contient $\sqrt{2}$. À la sortie, $b - a \leqslant \varepsilon$. Le milieu est à une distance d'au plus $(b-a)/2 \leqslant \varepsilon/2$ de tout point de $[a ; b]$, donc en particulier de $\sqrt{2}$.

12.3 Euclide : un algorithme vieux de plus de deux mille ans

```
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

Démonstration | L'idée mathématique

Si $a = bq + r$, les diviseurs communs de a et b sont exactement les diviseurs communs de b et r :

$$d \mid a \text{ et } d \mid b \text{ si et seulement si } d \mid b \text{ et } d \mid (a - bq) = r.$$

Remplacer (a, b) par (b, r) ne change donc pas le PGCD. Les restes diminuent jusqu'à 0 ; le dernier reste non nul est le PGCD.

13 Boite à outils Python de Seconde

Méthode | Choisir la structure

— choix : `if/elif/else` ;
— répétitions connues : `for` ;
— seuil ou arrêt inconnu : `while` ;
— calcul réutilisable : `def` et `return` ;
— hasard : `random()` ou `randint()`.

Exemple | Patron accumulateur

```
s = 0
for k in range(n):
    s = s + valeur(k)
```

Exemple | Patron compteur

```
c = 0
for k in range(n):
    if condition(k):
        c = c + 1
```

Méthode | Réflexe de vérification

1. nommer les entrées et la sortie ;
2. faire une table de trace ;
3. tester les frontières ;
4. vérifier l'arrêt d'un `while` ;
5. expliquer l'invariant ;
6. comparer à un calcul manuel.

Exemple | Patron seuil

```
n = 0
valeur = valeur_initiale
while valeur < seuil:
    valeur = mise_a_jour(
        valeur)
    n = n + 1
```

Attention | Les six pièges

Confondre = et == ; oublier les deux-points ; mal indenter ; inclure ou exclure la mauvaise borne ; oublier de modifier la variable d'un `while` ; utiliser `print` à la place de `return`.

« Un programme juste est un raisonnement rendu exécutable. »

14 Exercices progressifs

⭐☆☆ Prise en main ⭐⭐☆ Raisonnement ⭐⭐⭐ Maitrise complète

Pour chaque programme, commence par prévoir le résultat sans ordinateur. Une table de trace est toujours autorisée et souvent recommandée.

Exercice 1 ⭐☆☆ : Types et affectations

On exécute :

```
a = 7
b = 2.5
nom = "Ada"
test = a > 5
a = a + 3
b = a / 4
```

1. Donne le type initial de chacune des variables `a`, `b`, `nom` et `test`.
2. Donne les valeurs finales de `a`, `b`, `nom` et `test`.
3. Explique pourquoi la dernière valeur de `b` est un flottant.

Exercice 2 ⭐☆☆ : Construire une table de trace

```
x = 5
y = 2
x = x + y
y = 2 * x - y
x = y - x
print(x, y)
```

Construis une table donnant les valeurs de `x` et `y` après chaque affectation, puis indique l'affichage final.

Exercice 3 ⭐☆☆ : Traduire une formule

1. Écris une fonction `prix_ttc(prix_ht, taux)` qui renvoie le prix TTC pour un taux exprimé en pourcentage.
2. Calcule à la main puis avec ta fonction le prix TTC d'un objet à 60 euros soumis à un taux de 20 %.
3. Écris une fonction `distance(vitesse, duree)` qui renvoie la distance parcourue à vitesse constante.

Exercice 4 ⭐☆☆ : Trois cas

On considère la fonction :

```
def signe(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1
```

« Un programme juste est un raisonnement rendu exécutable. »

1. Que renvoient `signe(-4)`, `signe(0)` et `signe(2.7)` ?
2. Pourquoi le dernier `else` correspond-il nécessairement à $x < 0$?
3. Écris une fonction `minimum(a, b)` qui renvoie le plus petit des deux nombres.

Exercice 5 ⭐⭐☆ : Traduire un ensemble par une condition

On veut tester si un réel x appartient à

$$E = [-3; 2[\cup]5; +\infty[.$$

1. Écris une expression booléenne Python qui vaut `True` exactement lorsque $x \in E$.
2. Teste mentalement $x = -3$, $x = 2$, $x = 5$ et $x = 5{,}1$.
3. Écris une expression équivalente à `not(x in E)`, sans utiliser `not`.

Exercice 6 ⭐☆☆ : Maitriser `range`

Donne, sans ordinateur, toutes les valeurs successives de `k` et le nombre de tours pour :

1. `range(4)` ;
2. `range(3, 8)` ;
3. `range(2, 13, 3)` ;
4. `range(10, 3, -2)`.

Puis indique ce qu'affiche :

```
for k in range(1, 6):
    print(2 * k)
```

Exercice 7 ⭐⭐☆ : La somme des nombres impairs

```
def somme_impairs(n):
    s = 0
    for k in range(n):
        s = s + (2 * k + 1)
    return s
```

1. Fais une table de trace pour $n = 4$.
2. Conjecture une formule simple pour le résultat en fonction de n.
3. Montre que $(k+1)^2 - k^2 = 2k + 1$.
4. Explique pourquoi cette égalité prouve ta conjecture.

Exercice 8 ⭐⭐☆ : Compter les diviseurs

Complète la fonction suivante afin qu'elle renvoie le nombre de diviseurs positifs de n :

```
def nombre_diviseurs(n):
    compteur = ...
    for d in range(..., ...):
        if ...:
            compteur = ...
    return compteur
```

Utilise-la ensuite pour écrire `est_premier(n)`, qui renvoie `True` exactement lorsque n est premier. Pense au cas $n < 2$.

Exercice 9 ⭐⭐☆ : Une épargne atteint un seuil

Un capital de 500 euros augmente de 1,5 % par mois. On exécute :

```
capital = 500
mois = 0
while capital < 600:
    capital = capital * 1.015
    mois = mois + 1
print(mois, capital)
```

1. Explique le rôle de chaque variable et la condition de la boucle.
2. Pourquoi la boucle finit-elle par s'arrêter ?
3. Montre qu'après n mois, le capital vaut $500 \times 1{,}015^n$.
4. Détermine la valeur affichée pour `mois` et encadre le capital final au centime.

Exercice 10 ⭐⭐☆ : Composer des fonctions

Écris les fonctions suivantes :

1. `carre(x)`, qui renvoie x^2 ;
2. `distance_origine(x, y)`, qui renvoie $\sqrt{x^2 + y^2}$ en appelant `carre` ;
3. `dans_disque(x, y, r)`, qui renvoie un booléen indiquant si le point $(x ; y)$ appartient au disque de centre l'origine et de rayon r.

Teste les points $(3 ; 4)$ avec $r = 5$ puis $(4 ; 4)$ avec $r = 5$.

Exercice 11 ⭐⭐☆ : Déboguer sans deviner

La fonction suivante est censée renvoyer la moyenne des entiers de 1 à n :

```
def moyenne_entiers(n):
    somme = 0
    for k in range(1, n):
        somme = somme + k
    return somme / n + 1
```

1. Montre sur $n = 3$ que le résultat est faux.
2. Repère les deux erreurs logiques indépendantes.
3. Propose une version corrigée.
4. Démontre que le résultat renvoyé vaut $(n+1)/2$.

Exercice 12 ⭐⭐☆ : Deux dés simulés

On lance deux dés équilibrés et on s'intéresse à l'évènement « la somme est au moins 10 ».

1. Complète la fonction de simulation.

```
from random import randint

def frequence_somme_10(n):
    succes = 0
    for k in range(n):
        d1 = randint(..., ...)
        d2 = randint(..., ...)
        if ...:
            succes = ...
    return ...
```

2. Calcule exactement la probabilité recherchée en dénombrant les 36 couples équiprobables.
3. Quelle valeur attends-tu approximativement pour $n = 100000$? Pourquoi ne sera-t-elle presque jamais exacte ?

Exercice 13 ⭐⭐☆ : Lire une fonction d'écart type

On reprend la fonction du cours.

```
def ecart_type(valeurs):
    m = moyenne(valeurs)
    somme = 0
    for x in valeurs:
        somme = somme + (x - m) ** 2
    return sqrt(somme / len(valeurs))
```

1. Explique chaque ligne en français.
2. Calcule à la main le résultat pour `valeurs = [2, 2, 6, 6]`.
3. Que renvoie la fonction si toutes les valeurs sont égales ? Justifie sans exécuter le code.

« Un programme juste est un raisonnement rendu exécutable. »

Exercice 14 ⭐⭐⭐ : Approcher $\sqrt{3}$ par dichotomie

Complète les quatre zones :

```
def racine3(epsilon):
    a = 1.0
    b = 2.0
    while ...:
        m = ...
        if ...:
            a = m
        else:
            ...
    return (a + b) / 2
```

1. Effectue deux tours à la main à partir de $[1; 2]$.
2. Montre que $a \leqslant \sqrt{3} \leqslant b$ reste vrai après chaque tour.
3. Si la fonction s'arrête lorsque $b - a \leqslant 10^{-3}$, majore l'erreur sur la valeur renvoyée.

Exercice 15 ⭐⭐⭐ : Comprendre l'algorithme d'Euclide

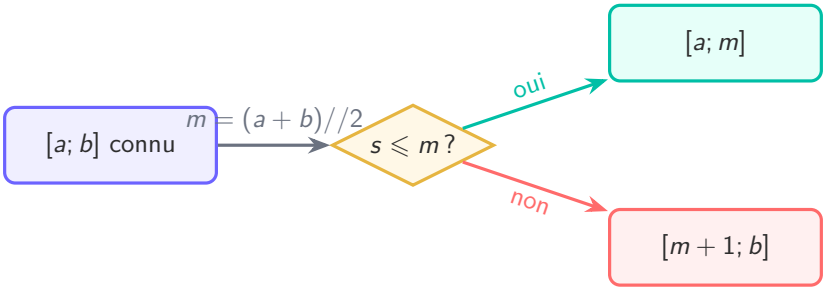
On considère :

```
def pgcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

1. Construis la table de trace pour `pgcd(252, 105)`.
2. Vérifie que la fonction renvoie 21 et que 21 divise les deux entiers initiaux.
3. Si $a = bq + r$, montre qu'un entier divise a et b si et seulement s'il divise b et r.
4. Explique pourquoi le deuxième nombre diminue strictement tant qu'il n'est pas nul. Déduis-en que la boucle s'arrête.
5. Conclus que la valeur renvoyée est bien le PGCD.

15 Problème défi : le coffre à code

Un coffre choisit secrètement un entier s entre 1 et 100. À chaque question, on propose un entier m et le coffre répond si $s \leqslant m$. On veut retrouver le secret en posant le moins de questions possible. Toutes les parties sont guidées ; les résultats précédents peuvent être réutilisés.



Partie A : découvrir la stratégie

On part de $[a ; b] = [1 ; 100]$ et on choisit à chaque étape $m = (a + b)//2$.

1. Si $m = 50$ et si la réponse est « non », quel nouvel intervalle contient s ?
2. Si la réponse est « oui », quel nouvel intervalle contient s ?
3. Explique pourquoi le nombre de candidats est approximativement divisé par 2 à chaque question.
4. Retrouve $s = 73$ en indiquant successivement les intervalles et les milieux.

Partie B : lire et justifier le programme

```
def nombre_questions(secret, maximum):
    a = 1
    b = maximum
    questions = 0
    while a < b:
        m = (a + b) // 2
        if secret <= m:
            b = m
        else:
            a = m + 1
        questions = questions + 1
    return questions
```

5. Pourquoi la condition est-elle `a < b` et non `a <= b` ?
6. Montre que l'invariant $a \leqslant s \leqslant b$ est vrai au départ et conservé par les deux branches.
7. Montre que la longueur $b - a + 1$ de l'intervalle diminue strictement si $a < b$.
8. Déduis des deux questions précédentes qu'à la sortie on a $a = b = s$.
9. Vérifie que `nombre_questions(73, 100)` renvoie 7.

« Un programme juste est un raisonnement rendu exécutable. »

Partie C : garantir un nombre maximal de questions

Après k questions, le nombre de candidats est au plus

$$\left\lceil \frac{100}{2^k} \right\rceil .$$

10. Explique cette formule à partir du partage en deux moitiés.
11. Compare 2^6, 2^7 et 100. Prouve que sept questions suffisent toujours, alors que six ne peuvent pas suffire dans tous les cas.
12. Complète le programme qui vérifie les cent secrets et renvoie le pire nombre de questions.

```
def pire_cas(maximum):
    pire = 0
    for secret in range(..., ...):
        q = ...
        if ...:
            pire = ...
    return pire
```

Partie D : secret aléatoire et nombre moyen de questions

13. Complète la simulation suivante.

```
from random import randint

def moyenne_questions(n):
    total = 0
    for k in range(n):
        secret = randint(..., ...)
        total = total + ...
    return ...
```

14. Une exécution avec un grand n donne environ 6,72. Explique pourquoi ce résultat varie légèrement et pourquoi il reste inférieur à 7.

Partie E : quand une réponse peut être fausse

Chaque réponse du coffre a une probabilité 0,05 d'être erronée. Pour sécuriser une question, on la pose trois fois et on garde la réponse majoritaire. On suppose les trois réponses indépendantes.

15. La majorité est fausse s'il y a exactement deux ou exactement trois erreurs. Montre que cette probabilité vaut
$$3 \times 0{,}05^2 \times 0{,}95 + 0{,}05^3 = 0{,}00725.$$

16. Complète la fonction qui renvoie `True` lorsque la majorité des trois réponses est correcte.

```
from random import random

def reponse_majoritaire_correcte():
    correctes = 0
    for k in range(3):
        if random() >= ...:
            correctes = correctes + 1
    return correctes >= ...
```

17. Compare 0,00725 à 0,05 et explique l'intérêt, mais aussi le cout, de la répétition.
18. Une erreur au début peut-elle être réparée par la suite par l'algorithme de dichotomie ? Explique pourquoi il faut sécuriser chaque réponse.

16 Corrigés détaillés des exercices

Corrigé 1

Démonstration | Types et affectations

1. Au départ :

`a` : `int`, `b` : `float`, `nom` : `str`, `test` : `bool`.

En effet, 7 est entier, 2.5 est un flottant, `"Ada"` est du texte et une comparaison renvoie un booléen.

2. La comparaison `a > 5` est vraie, donc `test` vaut `True`. Puis `a = a + 3` remplace 7 par 10. Enfin `b = a / 4` donne $10/4 = 2{,}5$. Les valeurs finales sont donc

`a = 10, b = 2.5, nom = "Ada", test = True.`

3. En Python 3, l'opérateur `/` réalise une division réelle et renvoie un `float`, même lorsque le quotient est entier.

Corrigé 2

Démonstration | Table de trace

On utilise toujours les valeurs disponibles **avant** l'affectation de la ligne.

Ligne	x	y	Calcul
1	5	non définie	initialisation de x
2	5	2	initialisation de y
3	7	2	$x \leftarrow 5+2$
4	7	12	$y \leftarrow 2\times 7-2$
5	5	12	$x \leftarrow 12-7$

Le programme affiche donc `5 12`.

Corrigé 3

Démonstration | Traduire une formule

1. Augmenter le prix HT du taux t revient à multiplier par $1 + t/100$.

```
def prix_ttc(prix_ht, taux):
    return prix_ht * (1 + taux / 100)
```

2. À la main,

$$60\left(1+\frac{20}{100}\right) = 60\times 1{,}2 = \boxed{72 \text{ euros}}.$$

L'appel `prix_ttc(60, 20)` renvoie `72.0`.

3. À vitesse constante, $d = v \times t$:

```
def distance(vitesse, duree):
    return vitesse * duree
```

Les unités doivent être compatibles : par exemple km/h et h donnent des km.

« Un programme juste est un raisonnement rendu exécutable. »

Corrigé 4

Démonstration | Trois cas

1. On obtient

`signe(-4) = -1`, `signe(0) = 0`, `signe(2.7) = 1`.

2. Lorsque Python atteint `else`, il sait que `x > 0` est faux et que `x == 0` est faux. Le nombre n'est ni positif ni nul ; il est donc strictement négatif.

3.

```
def minimum(a, b):
    if a <= b:
        return a
    else:
        return b
```

Le cas $a = b$ renvoie a, qui est bien le minimum commun.

Corrigé 5

Démonstration | Ensemble et booléen

1. La première partie impose $-3 \leqslant x < 2$ et la seconde $x > 5$:

```
(-3 <= x < 2) or (x > 5)
```

2.

— -3 appartient à E : la borne est incluse ;

— 2 n'appartient pas à E : la borne est exclue ;

— 5 n'appartient pas à E ;

— 5,1 appartient à E.

3. Le complémentaire de E est

$$]-\infty;-3[\cup [2;5].$$

Une expression sans `not` est donc

```
(x < -3) or (2 <= x <= 5)
```

On retrouve bien le changement de statut de toutes les frontières.

Corrigé 6

Démonstration | Les bornes de `range`

La borne finale n'est jamais prise.

Instruction	Valeurs	Tours
`range(4)`	$0, 1, 2, 3$	4
`range(3, 8)`	$3, 4, 5, 6, 7$	5
`range(2, 13, 3)`	$2, 5, 8, 11$	4
`range(10, 3, -2)`	$10, 8, 6, 4$	4

Dans la dernière boucle, k prend les valeurs $1, 2, 3, 4, 5$. Le programme affiche, sur cinq lignes,

$$\boxed{2,\ 4,\ 6,\ 8,\ 10}.$$

Corrigé 7

Démonstration | Somme des impairs

1. Pour $n = 4$:

k	nombre ajouté $2k+1$	s après le tour
avant la boucle	aucun	0
0	1	1
1	3	4
2	5	9
3	7	16

2. On conjecture $\boxed{\texttt{somme\_impairs(n)} = n^2}$.

3. En développant,

$$(k+1)^2 - k^2 = k^2 + 2k + 1 - k^2 = 2k + 1.$$

4. Le programme additionne

$$1 + 3 + \cdots + (2n-1) = \sum_{k=0}^{n-1} (2k+1).$$

Or $2k + 1 = (k+1)^2 - k^2$. Les termes intermédiaires s'annulent :

$$(1^2 - 0^2) + (2^2 - 1^2) + \cdots + (n^2 - (n-1)^2) = n^2.$$

La conjecture est donc démontrée.

Corrigé 8

Démonstration | Compter les diviseurs

On teste tous les entiers d de 1 à n inclus et on incrémente le compteur lorsque le reste est nul.

```
def nombre_diviseurs(n):
    compteur = 0
    for d in range(1, n + 1):
        if n % d == 0:
            compteur = compteur + 1
    return compteur
```

Un entier est premier s'il est au moins égal à 2 et possède exactement deux diviseurs positifs.

```
def est_premier(n):
    if n < 2:
        return False
    return nombre_diviseurs(n) == 2
```

Par exemple, 1 possède un seul diviseur mais n'est pas premier ; le test $n < 2$ est donc indispensable.

Corrigé 9

Démonstration | Capital et seuil

1. `capital` contient la somme disponible et `mois` compte les augmentations déjà appliquées. La boucle continue tant que le capital reste strictement inférieur à 600 euros.
2. À chaque tour, le capital positif est multiplié par $1{,}015 > 1$: il augmente. Les puissances de 1,015 finissent par dépasser $600/500 = 1{,}2$, donc la boucle s'arrête.
3. Au départ, pour $n = 0$, le capital vaut $500 = 500 \times 1{,}015^0$. S'il vaut $500 \times 1{,}015^n$, le tour suivant le multiplie par 1,015 et donne $500 \times 1{,}015^{n+1}$. La formule est donc vraie à chaque mois.
4. On trouve

$$500 \times 1{,}015^{12} \approx 597{,}81 < 600, \qquad 500 \times 1{,}015^{13} \approx 606{,}78 \geqslant 600.$$

Le programme affiche donc $\boxed{13}$ mois et un capital final d'environ $\boxed{606{,}78 \text{ euros}}$.

Corrigé 10

Démonstration | Composer des fonctions

```python
from math import sqrt

def carre(x):
    return x ** 2

def distance_origine(x, y):
    return sqrt(carre(x) + carre(y))

def dans_disque(x, y, r):
    return distance_origine(x, y) <= r
```

Pour $(3;4)$, la distance vaut $\sqrt{9+16} = 5$: le point appartient au disque fermé de rayon 5. Pour $(4;4)$, elle vaut $\sqrt{32} = 4\sqrt{2} \approx 5{,}66 > 5$: le point n'y appartient pas. Ainsi,

`dans_disque(3, 4, 5) = True`, `dans_disque(4, 4, 5) = False`.

Corrigé 11

Démonstration | Débogage raisonné

1. Pour $n = 3$, la boucle utilise seulement $k = 1, 2$, donc `somme` vaut 3. La dernière ligne renvoie $3/3 + 1 = 2$, alors que la moyenne de $1, 2, 3$ est également 2. Ce cas ne révèle donc pas l'erreur : il montre précisément qu'un test isolé peut réussir par compensation de deux erreurs. Pour $n = 4$, le programme renvoie

$$\frac{1+2+3}{4} + 1 = 2{,}5,$$

alors que la vraie moyenne vaut $(1+2+3+4)/4 = 2{,}5$: la compensation persiste pour tout n. Pour révéler les erreurs séparément, il faut lire le code, pas seulement tester sa sortie.

2. `range(1, n)` oublie n. Ensuite `somme / n + 1` ajoute 1 après la division, au lieu de simplement diviser la somme complète par n.

3.

```
def moyenne_entiers(n):
    somme = 0
    for k in range(1, n + 1):
        somme = somme + k
    return somme / n
```

4. Pour $n \geqslant 1$,

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

La fonction renvoie donc

$$\frac{n(n+1)/2}{n} = \boxed{\frac{n+1}{2}}.$$

Attention | Une subtilité importante

La version fausse renvoie par hasard la bonne réponse pour tout $n \geqslant 1$, car

$$\frac{1 + \cdots + (n-1)}{n} + 1 = \frac{n(n-1)/2}{n} + 1 = \frac{n+1}{2}.$$

Les deux erreurs se compensent exactement. Un programme peut donc donner de bonnes sorties pour une mauvaise raison ; expliquer le code reste indispensable.

Corrigé 12

Démonstration | Deux dés

1.

```python
from random import randint

def frequence_somme_10(n):
    succes = 0
    for k in range(n):
        d1 = randint(1, 6)
        d2 = randint(1, 6)
        if d1 + d2 >= 10:
            succes = succes + 1
    return succes / n
```

2. Les couples favorables sont

$$(4,6),(5,5),(5,6),(6,4),(6,5),(6,6).$$

Il y en a 6 parmi 36 couples équiprobables, donc

$$P(\text{somme} \geqslant 10) = \frac{6}{36} = \boxed{\frac{1}{6}}.$$

3. Pour $n = 100000$, on attend une fréquence proche de $1/6 \approx 0{,}1667$. Les tirages sont aléatoires : le nombre de succès fluctue, donc la fréquence n'est presque jamais exactement égale à $1/6$.

Corrigé 13

Démonstration | Écart type

1. La fonction calcule d'abord la moyenne m, initialise une somme, parcourt chaque valeur x, ajoute le carré de l'écart $x - m$, divise par le nombre de valeurs et prend la racine carrée.
2. Pour $2, 2, 6, 6$, la moyenne vaut 4. Les écarts sont $-2, -2, 2, 2$, donc les carrés valent tous 4. Ainsi

$$\sigma = \sqrt{\frac{4+4+4+4}{4}} = \sqrt{4} = \boxed{2}.$$

3. Si toutes les valeurs sont égales à c, leur moyenne est c. Chaque différence $x - m$ vaut 0, donc la somme et l'écart type valent $\boxed{0}$.

Corrigé 14

Démonstration | Dichotomie pour $\sqrt{3}$

Le programme complété est :

```
def racine3(epsilon):
    a = 1.0
    b = 2.0
    while b - a > epsilon:
        m = (a + b) / 2
        if m ** 2 < 3:
            a = m
        else:
            b = m
    return (a + b) / 2
```

1. Premier milieu : $m = 1{,}5$ et $m^2 = 2{,}25 < 3$, donc le nouvel intervalle est $[1{,}5; 2]$. Deuxième milieu : $m = 1{,}75$ et $m^2 = 3{,}0625 > 3$, donc le nouvel intervalle est $[1{,}5; 1{,}75]$.
2. Supposons $a \leqslant \sqrt{3} \leqslant b$. Si $m^2 < 3$, alors $m < \sqrt{3}$ et remplacer a par m conserve l'encadrement. Sinon $m^2 \geqslant 3$, donc $m \geqslant \sqrt{3}$ et remplacer b par m conserve encore l'encadrement.
3. À la sortie, $b - a \leqslant 10^{-3}$. Le milieu est à une distance d'au plus $(b-a)/2$ de $\sqrt{3}$, donc

$$\boxed{\left|\frac{a+b}{2} - \sqrt{3}\right| \leqslant 5 \times 10^{-4}}.$$

Corrigé 15

Démonstration | Algorithme d'Euclide

1.

Étape	a	b
départ	252	105
1	105	$252 \,\%\, 105 = 42$
2	42	$105 \,\%\, 42 = 21$
3	21	$42 \,\%\, 21 = 0$

2. La fonction renvoie $\boxed{21}$. On vérifie $252 = 21 \times 12$ et $105 = 21 \times 5$.
3. Si d divise a et b, alors il divise toute combinaison $a - bq = r$. Réciproquement, s'il divise b et r, il divise $bq + r = a$. Ainsi

$$d \mid a \text{ et } d \mid b \text{ si et seulement si } d \mid b \text{ et } d \mid r.$$

4. Le reste $r = a \,\%\, b$ vérifie $0 \leqslant r < b$. Tant que $b \neq 0$, le nouveau deuxième nombre est donc un entier positif ou nul strictement plus petit que l'ancien. Une suite strictement décroissante d'entiers naturels ne peut pas continuer indéfiniment : la boucle s'arrête.
5. Chaque remplacement conserve exactement les diviseurs communs, donc conserve le PGCD. À la fin, le couple est $(a, 0)$; ses diviseurs communs sont les diviseurs de a, et le plus grand est a. La valeur renvoyée est bien le PGCD initial.

17 Corrigé détaillé du problème

Partie A : découvrir la stratégie

Démonstration | Questions 1 à 4

1. Si la réponse à $s \leqslant 50$ est « non », alors $s > 50$ et le nouvel intervalle est $\boxed{[51;100]}$.

2. Si la réponse est « oui », on conserve $\boxed{[1;50]}$.

3. Le milieu sépare l'intervalle en deux parties dont les effectifs diffèrent d'au plus 1. Une réponse permet donc d'éliminer environ la moitié des candidats.

4. Pour $s = 73$:

Question	Intervalle avant	Milieu m	Intervalle après
1	$[1;100]$	50	$[51;100]$
2	$[51;100]$	75	$[51;75]$
3	$[51;75]$	63	$[64;75]$
4	$[64;75]$	69	$[70;75]$
5	$[70;75]$	72	$[73;75]$
6	$[73;75]$	74	$[73;74]$
7	$[73;74]$	73	$[73;73]$

Le secret est isolé après sept questions.

Partie B : lire et justifier le programme

Démonstration | Questions 5 à 9

5. Lorsque $a = b$, l'intervalle ne contient plus qu'un entier : le secret est trouvé. Avec `a <= b`, la boucle ferait un tour inutile et pourrait ne plus réduire correctement un intervalle déjà réduit à un point.

6. Au départ, l'entrée garantit $1 \leqslant s \leqslant \mathit{maximum}$, donc l'invariant est vrai. Supposons $a \leqslant s \leqslant b$.

— Si $s \leqslant m$, on remplace b par m : alors $a \leqslant s \leqslant m$.

— Sinon $s > m$; comme s est entier, $s \geqslant m+1$. On remplace a par $m+1$ et $m+1 \leqslant s \leqslant b$.

L'invariant est donc conservé.

7. Si $a < b$, le milieu $m = (a+b)//2$ vérifie $a \leqslant m < b$. Dans la branche gauche, la nouvelle longueur est $m - a + 1 < b - a + 1$. Dans la branche droite, elle vaut $b - m < b - a + 1$. Elle diminue donc strictement.

8. La longueur est un entier positif qui diminue : elle finit par valoir 1, donc $a = b$. L'invariant donne alors $a \leqslant s \leqslant a$, d'où $\boxed{a = b = s}$.

9. La trace de la partie A compte sept réductions, donc

$$\boxed{\texttt{nombre\_questions(73, 100) = 7}}.$$

Partie C : garantir le pire cas

Démonstration | Questions 10 à 12

10. Une question remplace un ensemble de N candidats par une moitié contenant au plus $\lceil N/2 \rceil$ candidats. Après deux questions, il en reste au plus $\lceil 100/2^2 \rceil$, et après k questions au plus $\lceil 100/2^k \rceil$.

11. On a

$$2^6 = 64 < 100 \leqslant 128 = 2^7.$$

Sept réponses binaires permettent de distinguer jusqu'à 128 possibilités, donc sept questions suffisent. Six réponses ne produisent que 64 suites possibles de réponses : elles ne peuvent pas identifier à coup sûr l'un des 100 secrets.

12.

```
def pire_cas(maximum):
    pire = 0
    for secret in range(1, maximum + 1):
        q = nombre_questions(secret, maximum)
        if q > pire:
            pire = q
    return pire
```

L'appel `pire_cas(100)` renvoie $\boxed{7}$, ce qui confirme la garantie démontrée.

Partie D : moyenne pour un secret aléatoire

Démonstration | Questions 13 et 14

13.

```
from random import randint

def moyenne_questions(n):
    total = 0
    for k in range(n):
        secret = randint(1, 100)
        total = total + nombre_questions(secret, 100)
    return total / n
```

14. Les secrets sont tirés aléatoirement, donc leurs effectifs varient légèrement d'une exécution à l'autre. La moyenne fluctue autour de la valeur théorique. Elle reste proche de 6,72 et inférieure à 7, car chaque recherche demande soit six, soit sept questions. Plus précisément, 28 secrets demandent six questions et 72 en demandent sept, donc

$$\frac{28 \times 6 + 72 \times 7}{100} = \boxed{6{,}72}.$$

Partie E : réponses imparfaites

Démonstration | Questions 15 à 18

15. Pour exactement deux erreurs, il y a trois choix pour la seule réponse correcte. Par indépendance, chaque ordre a une probabilité $0{,}05^2 \times 0{,}95$. Pour trois erreurs, la probabilité est $0{,}05^3$. Ainsi

$$P(\text{majorité fausse}) = 3 \times 0{,}05^2 \times 0{,}95 + 0{,}05^3 = \boxed{0{,}00725}.$$

16.

```python
from random import random

def reponse_majoritaire_correcte():
    correctes = 0
    for k in range(3):
        if random() >= 0.05:
            correctes = correctes + 1
    return correctes >= 2
```

Une réponse est modélisée correcte lorsque le nombre tiré appartient à $[0{,}05\,;1[$, de longueur 0,95.

17. Le risque passe de 5 % à 0,725 %, soit environ sept fois moins. En contrepartie, chaque étape demande trois réponses au lieu d'une : le temps et le nombre total de questions sont triplés.

18. Une mauvaise réponse choisit la moitié qui ne contient pas le secret. Le secret sort alors définitivement de l'intervalle $[a\,;b]$; les étapes suivantes ne cherchent que dans le mauvais intervalle et ne peuvent pas le récupérer. Il faut donc sécuriser chaque décision, surtout les premières.

18 Fiche-mémoire

Prévoir, exécuter, vérifier, expliquer

Définition | Affecter

`x = expression` calcule la droite puis remplace la valeur de `x`.

Méthode | Choisir

```
if condition:
    action_A
else:
    action_B
```

Méthode | Répéter *n* fois

```
for k in range(n):
    action
```

Méthode | Répéter jusqu'au seuil

```
while valeur < seuil:
    valeur = mise_a_jour(
        valeur)
```

Définition | Fonction

```
def calcul(x):
    resultat = ...
    return resultat
```

Méthode | Accumuler ou compter

Somme : partir de 0 et ajouter une valeur. Produit : partir de 1 et multiplier. Compteur : partir de 0 et ajouter 1 si une condition est vraie.

Méthode | Prouver

Identifier un invariant, montrer qu'il est vrai au départ, conservé à chaque tour, puis l'exploiter à la sortie.

Méthode | Tester

Cas simple, frontières, cas faux, cas extrême. Faire une table de trace avant d'exécuter.

Attention | Les réflexes qui évitent presque toutes les erreurs

= affecte et == compare ; la fin de `range` est exclue ; l'indentation définit les blocs ; un `while` doit modifier ce qui contrôle son arrêt ; `return` renvoie une valeur alors que `print` l'affiche.
Réflexe final : avant de lancer le programme, annonce en une phrase ce qu'il doit renvoyer et pourquoi.

« Un programme juste est un raisonnement rendu exécutable. »