

Introduction

1.1 Compétences exigées

— Objectifs —

Les capacités évaluées dans cette partie de la formation sont :

- comprendre un algorithme et expliquer ce qu'il fait,
- modifier un algorithme existant pour obtenir un résultat différent,
- concevoir un algorithme répondant à un problème précisément posé,
- expliquer le fonctionnement d'un algorithme,
- écrire des instructions.

1.2 Qu'est-ce qu'un algorithme ?

— Algorithme —

Un algorithme est une suite d'instructions, qui une fois exécutée correctement, conduit à un résultat donné.

1.3 Les algorithmes dans la vie courante

Les manuels d'utilisation des appareils actuels de la vie courante sont essentiellement des recueils d'algorithmes : des instructions sont données afin de faire fonctionner une fonction quelconque.

Les exemples de la vie courante ne manquent pas : automobiles, appareils divers, ...

Une recette de cuisine est également un algorithme simple.

Celui-ci comporte grossièrement trois étapes :

1. Réunir les ingrédients
2. Préparer
3. Déguster

La préparation consiste à exécuter une suite d'instructions : par exemple, plonger les tomates dans une casserole d'eau bouillante pendant quelques instants avant de les peler. On ne sait pas pourquoi il faut procéder de la sorte et d'ailleurs, ça n'a aucune importance : la recette a été écrite par quelqu'un qui sait. Elle marche.

En comparant avec les algorithmes de mathématiques, on pourrait dire que les ingrédients de la recette sont les **entrées** du processus auxquelles on applique le **traitement** (la préparation) pour obtenir, en **sortie**, un plat que l'on dégustera.

1.4 Construction d'un algorithme

En langage naturel, un algorithme se présente en général sous la forme suivante :

- Déclaration des variables :
On décrit dans le détail les éléments que l'on va utiliser dans l'algorithme.
- Initialisation et / ou Entrée des données :
On récupère les données et/ou on les initialise.
- Traitement des données :
On effectue les opérations nécessaires pour répondre au problème posé.
- Sortie :
On affiche le résultat.

On pourra alors mettre l'algorithme sous la forme suivante :

```

1: VARIABLES
2: Les variables (entrées et sorties entre autres) ainsi que leur type
3: ENTRÉES
4: LIRE les entrées
5: SORTIES
6: AFFICHER les sorties
7: DEBUT_ALGORITHME
8: FIN ALGORITHME

```

Algorithme 1 : Algorithme : construction

Vous pourrez, pour vous entraîner, télécharger le logiciel libre *AlgoBox* à l'adresse suivante :

<http://www.xm1math.net/algobox/download.html>

Il est disponible pour tous les systèmes d'exploitation.

1.5 Différents langages

Il existe une quantité de langages de programmation et de logiciels permettant de définir des algorithmes. Cette année, nous serons amenés à utiliser les outils suivants :

- Langage de programmation **Python**
- Logiciel **Scilab**

Considérons un algorithme historiquement célèbre, l'algorithme d'Euclide, qui sert à calculer le plus grand commun diviseur (pgcd) :

Étant donnés deux entiers, retrancher le plus petit au plus grand et recommencer jusqu'à ce que les deux nombres soient égaux. La valeur obtenue est le plus grand diviseur commun. L'idée de départ est de prendre



les 2 nombres et tant qu'ils ne sont pas égaux, de retirer le plus petit au plus grand.
L'algorithme se présente sous la forme suivante :

```
1: VARIABLES
2: a : int # "int" signifie entier comme integer
3: b : int
4: DEBUT_ALGORITHME
5:   TANT_QUE a et b sont différents FAIRE
6:     DEBUT_TANT_QUE
7:       SI a est le plus grand ALORS
8:         DEBUT_SI
9:           remplacer a par a - b
10:        FIN_SI
11:       SI b est le plus grand ALORS
12:         DEBUT_SI
13:           remplacer b par b - a
14:        FIN_SI
15:     FIN_TANT_QUE
16:   AFFICHER "le pgcd est" a
17: FIN ALGORITHME
```

Algorithme 2 : Algorithme d'Euclide



Voici cet algorithme présenté dans différents langages de programmation :

```
# CODE PYTHON
def pgcd(a,b):
    while a!=b # ou while a
        <>b:
            if a>b : a=a-b
            else : b=b-a
    return a
```

```
# CODE JAVA
public static int pgcd(int a,
int b) {
    while (a!=b){
        if (a>b) a=a-b
        else b=b-a
    }
    return a;
}
```

```
(* Code OCaml *)
let rec pgcd(a, b) =
    if a = b then a else
    if a > b then
        pgcd(a-b, b)
    else pgcd(a, b-a)
;;
```

```
Rem Code OOBASIC
Function Pgdc(ByVal a As Integer
,
ByVal b
As
Integer
) As
Integer

Do While a<>b
    If a>b Then
        a=a-b
    Else
        b=b-a
    EndIf

Loop
Pgdc=a
End Function
```

```
# CODE OCTAVE
function r = pgcd(a,b)
    while (a ~=b)
        if (a>b) : a=a-b
        ;
        else : b=b-a;
    end
    r=a;
```

```
/* CODE C */
int pgdc(int a,int b)
{
    while (a != b) {
        if (a>b) a=a-b;
        else b=b-a; }
    return a;
}
```

```
# CODE RUBY
def pgcd(a,b)
    while a!=b
        if a>b then a=a-
        b
        else b=b-a end
    end
    a
end
```

```
; Code Scheme
(define (pgdc a b)
    (cond
        ((< a b) (pgdc a
        (- b a) ))
        ((> a b) (pgdc
        (- a b) b ))
        (else a)
    )
)
```

```
# CODE PERL
sub pgcd {
    my ($a, $b) = @_;
    while ($a != $b) {
        if ($a > $b) {
            $a = $a - $b;
        } else {
            $b = $b - $a;
        }
    }
    return $a;
}
```

Action !

2.1 Un premier exemple complet

Le but de ce premier algorithme consiste à déterminer la distance entre deux points connaissant leurs coordonnées dans un repère orthonormé.

On considère les points $A \begin{pmatrix} x_A \\ y_A \end{pmatrix}$ et $B \begin{pmatrix} x_B \\ y_B \end{pmatrix}$ définis dans un repère orthonormal (O, \vec{i}, \vec{j}) .

Construire un algorithme permettant de calculer la longueur AB .

2.1.1 Langage naturel

Le langage naturel, pour nous, est le français. Nous utiliserons seulement des mots simples, le texte doit être clair et bien structuré.

On sait que la longueur d'un segment AB est définie par : $AB = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$.

On peut construire l'algorithme suivant :

Variables :

x_A est l'abscisse de A
 y_A est l'ordonnée de A
 x_B est l'abscisse de B
 y_B est l'ordonnée de B
 D est la distance entre A et B

Initialisation, entrées :

Saisir x_A
 Saisir y_A
 Saisir x_B
 Saisir y_B

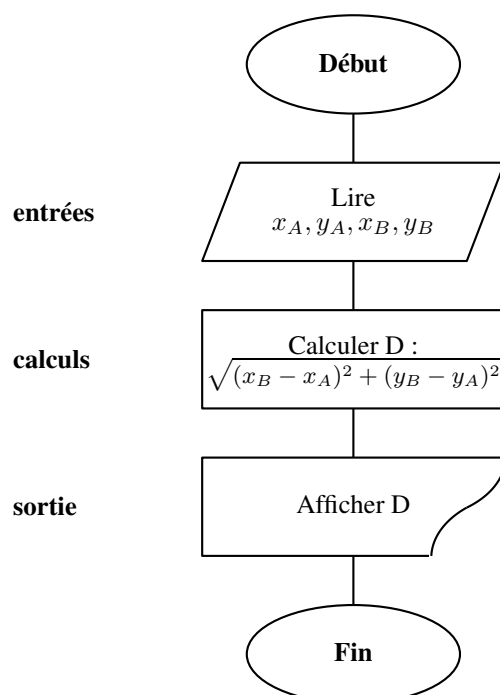
Traitement :

D prend la valeur $\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$

Sortie :

Afficher la valeur de D

Ou sous forme d'organigramme :



2.1.2 Avec Python

Python est un **langage de programmation** facile à utiliser et puissant. Il offre des structures de données de haut niveau et une approche simple mais réelle de la programmation.

Il est téléchargeable à l'adresse :

<http://www.python.org/download/>

L'algorithme précédent peut s'écrire en python de la façon suivante :

distance1.py :

```

1  # -*- coding: utf-8 -*-
2
3  from math import sqrt
4  # ou from math import *
5
6  # commentaire : entrée des données
7  print ("Entrez l'abscisse de A")
8  x_A = float(input())
9  print ("Entrez l'ordonnée de A")
10 y_A = float(input())
11 print ("Entrez l'abscisse de B")
12 x_B = float(input())
13 print ("Entrez l'ordonnée de B")
14 y_B = float(input())
15 # calcul de la distance (commentaires)
16 d = sqrt((x_B-x_A)**2+(y_B-y_A)**2)
17 # affichage du résultat
18 print ("La distance entre A et B est :")
19 print (d)
20 # affichage du résultat arrondi
21 print("Avec 2 décimales, cela donne : %.2f" %d)

```

2.2 À vous de jouer !

2.2.1 Milieu de 2 points

⇒ **Activité 2.1**

Construisez un algorithme en langage naturel qui permet de déterminer les coordonnées du milieu I d'un segment $[AB]$ connaissant les coordonnées de A et de B dans un repère quelconque.

2.2.2 Résolution mathématique

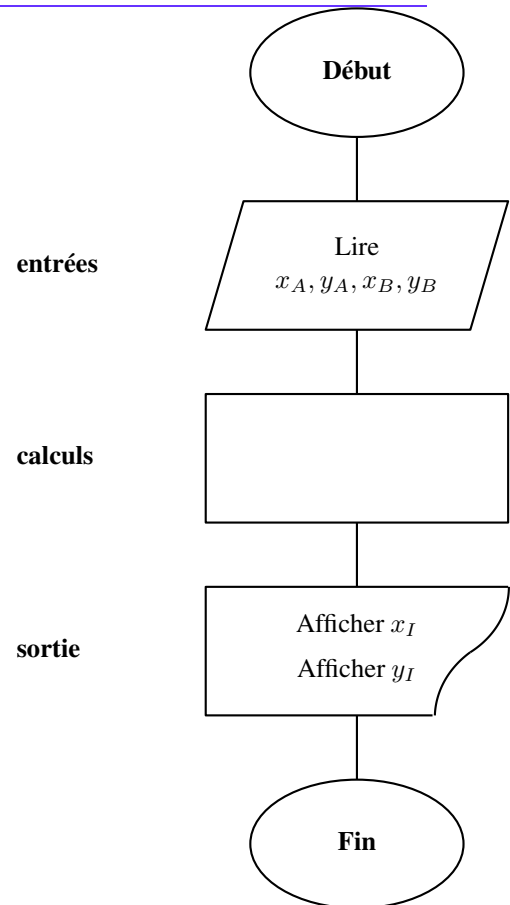
2.2.3 En langage naturel

Variables :

x_A est l'abscisse de A
 y_A est l'ordonnée de A
 x_B est l'abscisse de B
 y_B est l'ordonnée de B
 x_I est l'abscisse du milieu I du segment $[AB]$
 y_I est l'ordonnée du milieu I du segment $[AB]$

Initialisation, entrées :

Saisir x_A
 Saisir y_A
 Saisir x_B
 Saisir y_B



2.2.4 Avec Python, si vous avez déjà quelques notions

milieu1.py :

```

1  # -*- coding: utf-8 -*-
2
3  from math import *
4
5  # commentaire : entrée des données
6  print ("Entrez l'abscisse de A")
7  x_A = float(input())
8  print ("Entrez l'ordonnée de A")
9  y_A = float(input())
10 print ("Entrez l'abscisse de B")
11 x_B = float(input())
12 print ("Entrez l'ordonnée de B")
13 y_B = float(input())
14
15 # calcul des coordonnées du milieu
16 x_I = (x_A+x_B)/2
17 y_I = (y_A+y_B)/2
18
19 # affichage du résultat
20 print ("Le milieu a pour coordonnées :")
21 print (x_I, y_I)
22 # affichage du résultat arrondi
23 print("Avec 2 décimales, cela donne \
24 %.2f pour l'abscisse et \
25 %.2f pour l'ordonnée" %(x_I,y_I))
26 # \ permet de passer à la ligne
  
```

2.3 Algorithmique : techniques de base

2.3.1 Variables et affectations

Les variables en algorithmique

- Les variables algorithmiques peuvent servir à stocker des données de différents types.
- La valeur d'une variable peut changer au fil des instructions de l'algorithme.
- Les opérations sur les variables s'effectuent ligne après ligne et les unes après les autres.
- Quand l'ordinateur exécute une ligne du type `mavARIABLE PREND_LA_VALEUR un calcul`, il effectue d'abord le calcul et stocke ensuite le résultat dans `mavARIABLE`.



— Remarque —

Les commentaires seront placés après le symbole # comme dans python pour faciliter la lecture.

⇒ Activité 2.2

On considère l'algorithme suivant :

```

1: VARIABLES
2: x : int # "int" signifie entier comme integer
3: y : int
4: z : int
5: DEBUT_ALGORITHME
6:   x ← 2 # x prend la valeur 2
7:   y ← 3
8:   z ← x + y
9: FIN_ALGORITHME

```

Algorithme 3 : Valeurs de x, y z

Après exécution de l'algorithme, la variable x contient la valeur , la variable y contient la valeur et la variable z contient la valeur .

⇒ Activité 2.3

On considère l'algorithme suivant :

```

1: VARIABLES
2: x : int
3: DEBUT_ALGORITHME
4:   x ← 2
5:   x ← x + 1
6: FIN_ALGORITHME

```

Algorithme 4 : Valeur de x

Après exécution de l'algorithme, la variable x contient la valeur : .

⇒ Activité 2.4

Ajoutons la ligne « $x \leftarrow 4 * x$ » à la fin du code précédent. x contient alors la valeur .

⇒ **Activité 2.5**

On considère l'algorithme suivant :

```

1: VARIABLES
2: y : int
3: DEBUT_ALGORITHME
4:   y ← 2
5:   y ← y + 1
6:   y ← 4 * y
7: FIN_ALGORITHME

```

Algorithme 5 : Valeur de y

Après exécution de l'algorithme, la variable y contient la valeur : .

⇒ **Activité 2.6**

On considère l'algorithme suivant :

```

1: VARIABLES
2: a : int
3: b : int
4: c : int
5: DEBUT_ALGORITHME
6:   a ← 5
7:   b ← 3
8:   c ← a + b
9:   b ← b + a
10:  a ← c
11: FIN_ALGORITHME

```

Algorithme 6 : Valeurs de a, b, c

Après exécution de l'algorithme, la variable a contient la valeur , la variable b contient la valeur et la variable c contient la valeur .

⇒ **Activité 2.7**

On considère l'algorithme suivant :

```

1: VARIABLES
2: x : int
3: y : int
4: z : int
5: ENTRÉES
6: LIRE x
7: SORTIES
8: AFFICHER z
9: DEBUT_ALGORITHME
10:  y ← x - 2
11:  z ← -3 * y - 4
12:  AFFICHER z
13: FIN_ALGORITHME

```

Algorithme 7 : Afficher z

On cherche maintenant à obtenir un algorithme équivalent sans utiliser la variable y . Complétez la ligne 9 dans l'algorithme ci-dessous pour qu'il réponde au problème.

```

1: VARIABLES
2: x : int
3: z : int
4: ENTRÉES
5: LIRE x
6: SORTIES
7: AFFICHER z
8: DEBUT_ALGORITHME
9:  z ← .....
10:  AFFICHER z
11: FIN_ALGORITHME

```

Algorithme 8 : Afficher z simplifié

⇒ **Activité 2.10**

On considère l'algorithme suivant :

```

1: VARIABLES
2: a : int
3: b : int
4: DEBUT_ALGORITHME
5:   a ← 1
6:   b ← 3
7:   SI (a > 0) ALORS
8:     DEBUT_SI
9:     a ← a + 1
10:    FIN_SI
11:   SI (b > 4) ALORS
12:     DEBUT_SI
13:     b ← b - 1
14:     FIN_SI
15: FIN_ALGORITHME

```

Algorithme 11 : Valeurs de a et b

Après exécution de l'algorithme,

- la variable a contient la valeur : ,
- La variable b contient la valeur : .

⇒ **Activité 2.11**

On cherche à concevoir un algorithme correspondant au problème suivant :

- on demande à l'utilisateur d'entrer un nombre (représenté par la variable x)
- si le nombre entré est différent de 1, l'algorithme doit stocker dans une variable y la valeur de $\frac{1}{x-1}$ et afficher la valeur de y
(note : la condition x différent de 1 s'exprime avec le code $x! = 1$). On ne demande pas de traiter le cas contraire.

Complétez l'algorithme ci-dessous pour qu'il réponde au problème.

```

1: VARIABLES
2: x : int
3: y : int
4: ENTRÉES
5: LIRE .....
6: SORTIES
7: AFFICHER .....
8: DEBUT_ALGORITHME
9:   SI (.....) ALORS
10:    DEBUT_SI
11:    ..... ← .....
12:    AFFICHER .....
13:    FIN_SI
14: FIN_ALGORITHME

```

Algorithme 12 : Inverse

2.3.3 Boucles

Boucles POUR...DE...A

- Les boucles permettent de répéter des instructions autant de fois que l'on souhaite.
- Lorsqu'on connaît par avance le nombre de fois que l'on veut répéter les instructions, on utilise une boucle du type POUR...DE...A... dont la structure est la suivante :

```
POUR...ALLANT_DE...A...
  DEBUT_POUR
  ...
  FIN_POUR
```

- Exemple : l'algorithme ci-dessous permet d'afficher la racine carrée de tous les entiers de 1 jusqu'à 50. La variable n est appelée « compteur de la boucle ».

```
1: VARIABLES
2: n : int
3: racine : float
4: SORTIES
5: AFFICHER racine
6: DEBUT_ALGORITHME
7:   POUR n ALLANT_DE 1 A 50
8:     DEBUT_POUR
9:     racine ← sqrt(n)
10:    AFFICHER racine
11:   FIN_POUR
12: FIN_ALGORITHME
```

Algorithme 13 : Racine carrée 2



— Remarques —

- La variable servant de compteur pour la boucle doit être du type NOMBRE et doit être déclarée préalablement (comme toutes les variables).
- Sauf précision, cette variable est automatiquement augmentée de 1 à chaque fois.
- On peut utiliser la valeur du compteur pour faire des calculs à l'intérieur de la boucle, mais les instructions comprises entre DEBUT_POUR et FIN_POUR ne doivent pas modifier la valeur de la variable qui sert de compteur.

⇒ Activité 2.12

On cherche à concevoir un algorithme qui affiche, grâce à une boucle POUR...DE...A, les résultats des calculs suivants : $8 * 1, 8 * 2, 8 * 3, 8 * 4, \dots$ jusqu'à $8 * 10$.

La variable n sert de compteur à la boucle et la variable *produit* sert à stocker et afficher les résultats. Complétez les lignes 7 et 9 dans l'algorithme ci-dessous pour qu'il réponde au problème :

```
1: VARIABLES
2: n : int
3: produit : int
4: SORTIES
5: AFFICHER produit
6: DEBUT_ALGORITHME
7:   POUR n ALLANT_DE ..... A .....
8:     DEBUT_POUR
9:     produit ← .....
10:    AFFICHER produit
11:   FIN_POUR
12: FIN_ALGORITHME
```

Algorithme 14 : Table de 8

⇒ Activité 2.13

On considère l'algorithme suivant :

```

1: VARIABLES
2: n : int
3: somme : int
4: SORTIES
5: AFFICHER somme
6: INITIALISATION
7: somme ← 0
8: DEBUT_ALGORITHME
9:   POUR n ALLANT_DE 1 A 100
10:    DEBUT_POUR
11:    somme ← somme + n
12:    FIN_POUR
13:  AFFICHER somme
14: FIN_ALGORITHME

```

Algorithme 15 : Somme 100

Complétez les phrases suivantes :

- Après exécution de la ligne 7, la variable *somme* contient la valeur : .
- Lorsque le compteur *n* de la boucle vaut 1 et après exécution du calcul de la ligne 11, la variable *somme* vaut : .
- Lorsque le compteur *n* de la boucle vaut 2 et après exécution du calcul de la ligne 11, la variable *somme* vaut : .
- Lorsque le compteur *n* de la boucle vaut 3 et après exécution du calcul de la ligne 11, la variable *somme* vaut : .

Que permet de calculer cet algorithme ?

⇒ **Activité 2.14**

Complétez les lignes 9 et 11 de l'algorithme ci-dessous pour qu'il permette de calculer la somme $5^2 + 6^2 + 7^2 + \dots + 24^2 + 25^2$, c'est-à-dire la somme $\sum_5^{25} n^2$.

```

1: VARIABLES
2: n : int
3: somme : int
4: SORTIES
5: AFFICHER somme
6: INITIALISATION
7: somme ← 0
8: DEBUT_ALGORITHME
9:   POUR n ALLANT_DE .... A .....
10:    DEBUT_POUR
11:    somme ← somme+.....
12:    FIN_POUR
13:  AFFICHER somme
14: FIN_ALGORITHME

```

Algorithme 16 : Somme des carrés

Boucles TANT QUE...

- Il n'est pas toujours possible de connaître par avance le nombre de répétitions nécessaires à un calcul. Dans ce cas là, il est possible d'avoir recours à la structure TANT QUE... qui se présente de la façon suivante :

```

TANT_QUE...FAIRE
  DEBUT_TANT_QUE
  ...
  FIN_TANT_QUE

```

Cette structure de boucle permet de répéter une série d'instructions (comprises entre DE-

BUT_TANT_QUE et FIN_TANT_QUE) tant qu'une certaine condition est vérifiée.

- Exemple : Comment savoir ce qu'il reste si on enlève 25 autant de fois que l'on peut au nombre 583 ? Pour cela on utilise une variable n , qui contient 583 au début, à laquelle on enlève 25 tant que c'est possible, c'est à dire tant que n est supérieur ou égal à 25. Voici ci-dessous un exemple d'algorithme avec une structure de boucle.

```

1: VARIABLES
2: n : int
3: SORTIES
4: AFFICHER n
5: INITIALISATION
6: n ← 583
7: DEBUT_ALGORITHME
8:   TANT_QUE n >= 25 FAIRE
9:     DEBUT_TANT_QUE
10:    n ← n - 25
11:   FIN_TANT_QUE
12:   AFFICHER n
13: FIN_ALGORITHME

```

Algorithme 17 : Tant que



— Remarques —

- Si la condition du TANT QUE est fausse dès le début, les instructions entre DEBUT_TANT_QUE et FIN_TANT_QUE ne sont jamais exécutées (la structure TANT QUE ne sert alors strictement à rien).
- Il est indispensable de s'assurer que la condition du TANT QUE finisse par être vérifiée (le code entre DEBUT_TANT_QUE et FIN_TANT_QUE doit rendre vraie la condition tôt ou tard), sans quoi l'algorithme ne pourra pas fonctionner.

⇒ Activité 2.15

On cherche à connaître le plus petit entier N tel que 2^N soit supérieur ou égal à 10000. Pour résoudre ce problème de façon algorithmique :

- On utilise une variable N à laquelle on donne au début la valeur 1.
- On augmente de 1 la valeur de N tant que 2^N n'est pas supérieur ou égal à 10000.

Une structure TANT QUE est particulièrement adaptée à ce genre de problème car on ne sait pas a priori combien de calculs seront nécessaires.

Compléter les lignes 8 et 10 de l'algorithme ci-dessous pour qu'il réponde au problème :

```

1: VARIABLES
2: N : int
3: SORTIES
4: AFFICHER N
5: INITIALISATION
6: N ← 1
7: DEBUT_ALGORITHME
8:   TANT_QUE 2N ..... FAIRE
9:     DEBUT_TANT_QUE
10:    N ← .....
11:   FIN_TANT_QUE
12:   AFFICHER N
13: FIN_ALGORITHME

```

Algorithme 18 : 2 puissance N

⇒ Activité 2.16

On considère le problème suivant :

- On lance une balle d'une hauteur initiale de 3 m.
- On suppose qu'à chaque rebond, la balle perd 10% de sa hauteur (la hauteur est donc multipliée par 0,9 à chaque rebond).
- On cherche à savoir le nombre de rebonds nécessaire pour que la hauteur de la balle soit inférieure ou égale à 10 cm.

Complétez les lignes 10 et 13 de l'algorithme ci-dessous pour qu'il réponde au problème.

```

1: VARIABLES
2: nombre_rebonds : int
3: hauteur : float
4: SORTIES
5: AFFICHER nombre_rebonds
6: INITIALISATION
7: nombre_rebonds ← 0
8: DEBUT_ALGORITHME
9:   hauteur ← 300
10:  TANT_QUE (hauteur > 0) FAIRE
11:   DEBUT_TANT_QUE
12:   nombre_rebonds ← nombre_rebonds + 1
13:   hauteur ← hauteur / 2
14:   FIN_TANT_QUE
15: AFFICHER nombre_rebonds
16: FIN_ALGORITHME

```

Algorithme 19 : Nombre de rebonds

⇒ **Activité 2.17**

Écrivez une fonction qui prend en entrée un entier naturel a et retourne (c'est-à-dire renvoie) cet entier écrit à l'envers. Par exemple, si $a = 1234$, la fonction devra retourner $a = 4321$. Vous pourrez utiliser les fonctions $\text{quotient}(n, p)$ et $\text{reste}(n, p)$ qui donnent le quotient et le reste de la division de n par p .



— Remarque —

La fonction *return* signifie "retourner", c'est-à-dire "envoyer en retour" ou "renvoyer"

L'idée est que si l'on prend le reste de a dans la division par 10, on récupère le dernier chiffre, et si on prend le quotient, on récupère a privé de son dernier chiffre. Il suffit d'itérer le procédé, en décalant à chaque fois le résultat provisoire vers la gauche.

```

1: VARIABLES
2:
3:
4: ENTRÉES
5: LIRE a
6: SORTIES
7: AFFICHER b "inverse" de a
8: INITIALISATION
9:
10: DEBUT_ALGORITHME
11:  TANT_QUE (a > 0) FAIRE
12:   DEBUT_TANT_QUE
13:   b ← a % 10
14:   a ← a / 10
15:   b ← b * 10 + a % 10
16:   FIN_TANT_QUE
17: AFFICHER b
18: FIN_ALGORITHME

```

Algorithme 20 : Renversement