Chapitre 1

Introduction à l'algorithmique

1.1 Algorithme

Un **algorithme** est une suite finie d'instructions à appliquer dans un ordre déterminé à un nombre fini de données pour arriver, en un nombre fini d'étapes, à un certain résultat.

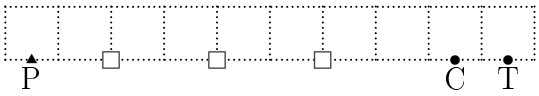
Exemple : Un pirate P doit récupérer le trésor T. Pour cela, il doit avoir la clé C de celui-ci. Le pirate se déplace sur le quadrillage ; les carrés blancs représentent des obstacles qui doivent être contournés. Ce pirate n'est capable que d'effectuer les instructions suivantes :

Avancer : avancer d'une case ;

Sauter : sauter par dessus un obstacle jusqu'à la cas suivante ;

Ramasser : ramasser un objet sur la case occupée ;

Ouvrir : Ouvrir le coffre (nécessite la clé).



Les instructions ci-dessous constituent un algorithme permettant au pirate de récupérer le trésor.

1. Avancer
2. Sauter
3. Avancer
4. Sauter
5. Avancer
6. Sauter
7. Avancer
8. Avancer
9. Ramasser
10. Avancer
11. Ouvrir

Exemples : Un algorithme n'est qu'une suite d'instructions à effectuer. On en rencontre quotidiennement par exemple lorsque l'on :

— applique une recette de cuisine ;

— suit les instructions d'un GPS ;

— applique une procédure d'évacuation d'urgence.

Un algorithme est initialement écrit en langage humain : on parle de **pseudo-code**. Le langage humain n'est pas directement, voire pas du tout compris par les machines. Il doit être traduit dans un langage que celles-ci peuvent comprendre.

Un **programme** est la traduction d'un algorithme dans un langage compréhensible par la machine. On appelle ce langage un **langage de programmation**.

Remarques :

— Tous les langages informatiques ne sont pas forcément des langages de programmations. On peut en effet considérer deux autres catégories : les langages descriptifs comme le HTML et le CSS qui servent à faire des pages Web ; les langages de requêtage dédiés à la gestion et la consultation des bases de données comme le SQL.

— Il n'a pas fallu attendre le début de l'informatique pour intégrer la notion d'algorithmes. En 1801, le français Joseph Marie Jacquard mis au point un métier à tisser, appelé métier Jacquard, qui utilisait des cartes perforées pour représenter les mouvements du bras du métier à tisser, et ainsi générer automatiquement des motifs décoratifs.

1.2 Variables

1.2.1 Variable et opérations élémentaires

Une **variable** représente une zone de la mémoire d'un ordinateur ou d'une machine. Elle peut servir à stocker un résultat (d'une opération par exemple), une valeur ou une donnée. Pour pouvoir accéder à son contenu, on lui donne un nom.

Une variable peut être vue comme une boîte dans laquelle on stockerait du contenu.

Boîte / Variable

Informatiquement, il est possible d'effectuer plusieurs opérations sur une variable, notamment

la création : on réserve une zone de la mémoire (on crée de la boîte) ;

l'affectation : on stocke un contenu dans la variable (on place d'un contenu dans la boîte) ;

la lecture : on accède au contenu de la variable (on regarde dans la boîte) ;

la modification : on change le contenu de la variable (de la boîte).

La création peut se faire sans affectation dans certain langage. L'affectation est quant à elle symbolisée par un « = ». Pour modifier une variable, il suffit de lui réaffecter une nouvelle valeur.

Algorithmiquement, la création et l'affection se font simultanément. Ces deux opérations et la modification d'une variable X sont symbolisées par :

$$X \leftarrow x$$

où x est la valeur attribuée à la variable X ; on lit « X prend la valeur x », ou « la valeur x est affectée à X ».

Pseudo-code	**Langage de programmation**	**Valeur de la variable X**
$X \leftarrow 10$	`X = 10`	10
$X \leftarrow X + 5$	`X = X + 5`	15
$X \leftarrow 6$	`X = 6`	6

1.2.2 Type d'une variable

On peut énumérer quatre types de base de variables, communs à la plupart des langages de programmation :

— les **entiers** (nombres entiers relatifs) ;

— les **flottants** (nombres à virgule) ;

— les **chaînes de caractères** : suite ordonnée de caractères (lettres, symboles, chiffres...) ;

— les **booléens** : variables qui ne prennent que deux valeurs : **Vrai** ou **Faux** (sa valeur est en général donnée par un test).

Remarques :

— Une variable ne peut avoir qu'un seul type.

— Les tests – notamment en Python – sont notés à l'aide parenthèses. Par exemples, `(1 < 2)` est un test pour savoir si 1 est plus petit que 2, sa valeur est `Vrai` ; `(0 == 1)` est un test pour savoir si 0 est égal à 1, sa valeur est `Faux`.

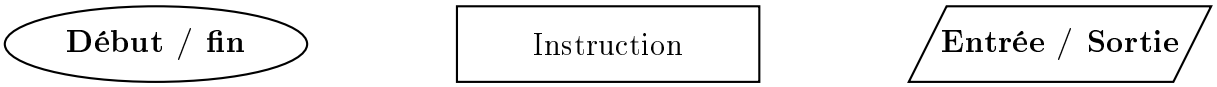
— Il existe de nombreux autres types de variables mais ils ne sont pas forcément communs à tous les langages et il s'agit généralement de types plus évolués que les quatre de base ci-dessus.

Exemples :

1. La variable `prénom`, destinée à enregistrer votre prénom est de type chaîne de caractère.
2. La variable `age`, destinée à enregistrer votre âge est de type entier.
3. On appelle `taille`, destinée à enregistrer votre taille en mètre est de type flottant.
4. La variable `majeur` qui contient le test `(age > 18)` est de type booléen, elle teste si vous êtes majeurs ou non.

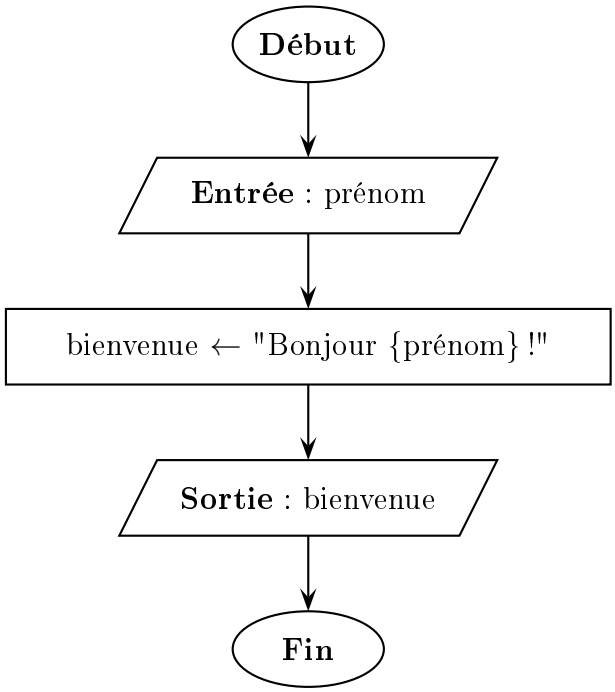
1.3 Logigramme

Un logigramme est un schéma représentant un algorithme à l'aide de symboles prédéfinis.



Les différents symboles du logigramme sont reliés entre eux par des flèches. Il n'y a qu'un seul début mais il peut y avoir plusieurs fins. Nous verrons dans les prochains cours un autre symbole correspondant à une dernière situation.

Exemple : le logigramme suivant correspond à un algorithme demandant à une personne son prénom puis lui souhaitant la bienvenue.



Demande et affectation d'une valeur pour la variable *prénom*.

Le contenu de *prénom* est inséré dans la chaîne de caractères qui est affectée à la variable *bienvenue*.

Le contenu de la variable *bienvenue* est affiché.

1.4 Instructions conditionnelles

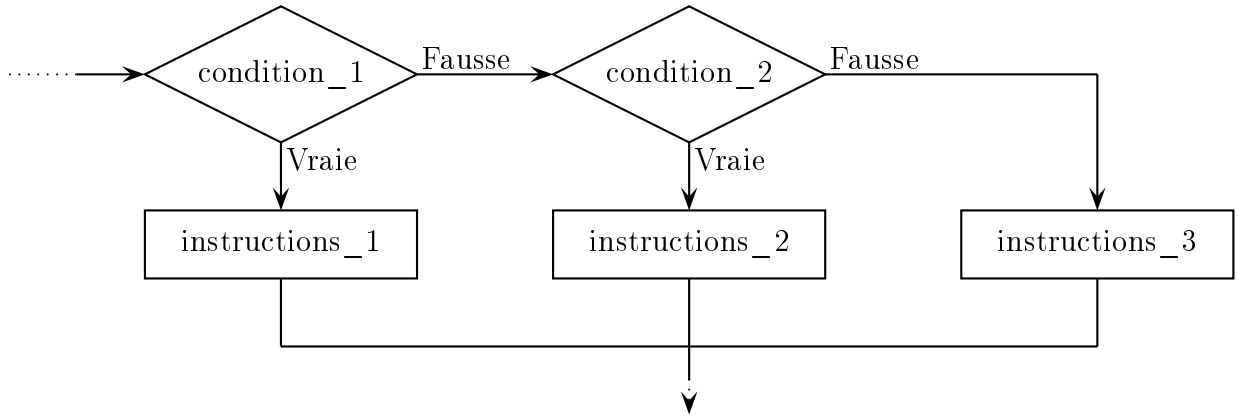
Dans un algorithme ou un programme, une séquence d'instructions conditionnelles est une suite d'instructions qui sont réalisées si certaines conditions sont vérifiées. La vérification est effectuée lors d'un test.

Une séquence d'instructions conditionnelles aura toujours la structure ci-dessous même si tous les éléments ne sont pas forcément présents.

Algorithme 1 : Structure

```
1 Si condition_1  :
2     instructions_1 # si condition_1 vérifiée
3 Sinon Si condition_2  :
4     instructions_2 # si condition_2 vérifiée mais pas la condition_1
5 Sinon :
6     instructions_3 # si aucune des conditions précédentes n'est vérifiée
```

Cela peut se représenter sous la forme du logigramme ci-dessous où les tests des conditions sont symbolisés par des losanges.

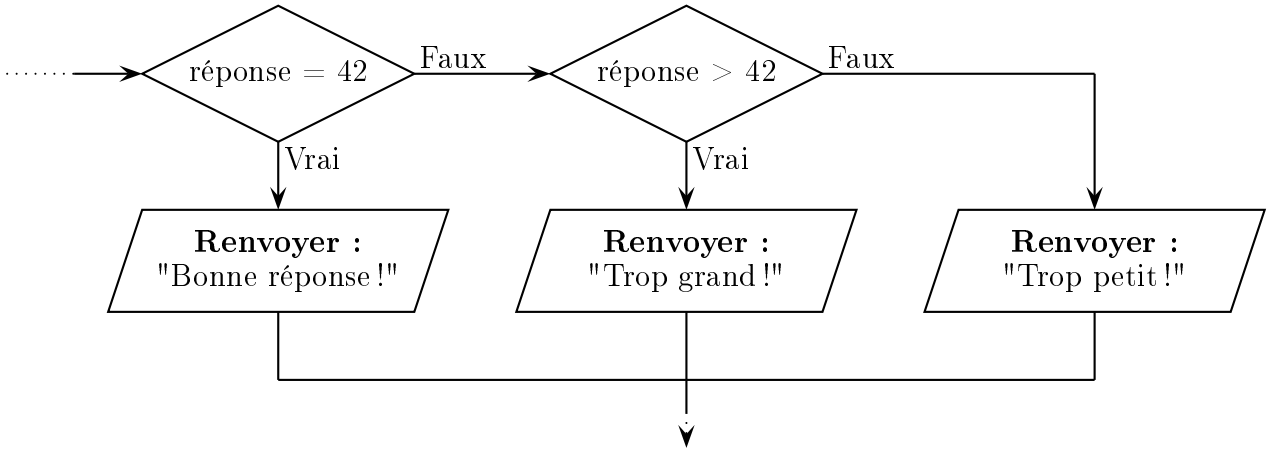


Exemples :

1. Structure avec **Si**, **SinonSi** et **Sinon**.

Algorithme 2 : Exemple 1

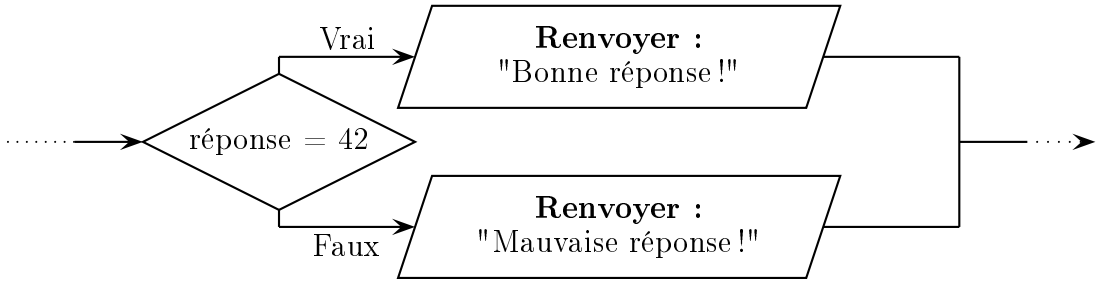
```
1 Si réponse = 42 :
2     Renvoyer : "Bonne réponse !"
3 Sinon Si réponse > 42 :
4     Renvoyer : "Trop grand !"
5 Sinon :
6     Renvoyer : "Trop petit !"
```



2. Structure avec **Si** et **Sinon**.

Algorithme 3 : Exemple 2

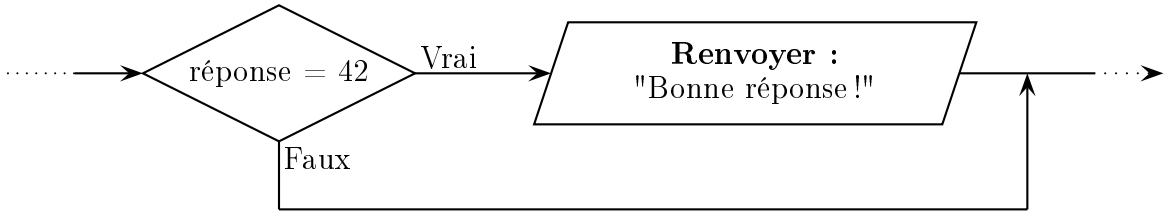
```
1 Si réponse = 42 :
2     Renvoyer : "Bonne réponse !"
3 Sinon :
4     Renvoyer : "Mauvaise réponse !"
```



Algorithme 4 : Exemple 3

```
1 Si réponse = 42 :
2     Renvoyer : "Bonne réponse !"
```

3. Structure avec seulement un **Si**.



1.5 Boucles

Si on reprend l'exemple initial de notre pirate, on remarque que l'on enchaîne la séquence « Avancer, Sauter » plusieurs fois de suite. Il serait plus concis et clair écrire cela sous la forme

1. Répéter 3 fois :
   (a) Avancer
   (b) Sauter
2. Répéter 2 fois :
   (a) Avancer
3. Ramasser
4. Avancer
5. Ouvrir

La répétition d'un bloc d'instructions déterminés est permise par les **boucles**. Elles permettent d'éviter d'avoir à écrire plusieurs lignes de code identiques. Il existe deux types de boucles :

— les **boucles conditionnelles** qui s'exécutent **tant qu'une** certaine condition est vérifiée ;

— les **boucles non conditionnelles** qui s'exécutent **pour** des valeurs prédéfinies avant le démarrage de la boucle.

1.5.1 Boucles conditionnelles

Boucles conditionnelles

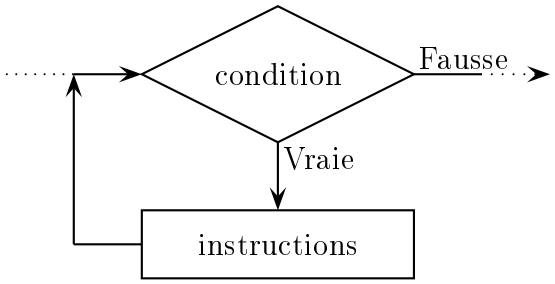
Les boucles conditionnelles s'exécutent **tant qu'une** certaine condition est vérifiée ; elles sont donc plus communément appelées boucles **Tant que** ou boucle *while* en anglais. À l'instar des instructions conditionnelles, un test est effectué pour déterminer si la condition de la boucle est vérifiée ou non.

En pseudo-code, on retrouvera les boucles **Tant que** de la façon suivante.

Algorithme 5 : Structure boucle **Tant que**

```
1 Tant que condition :
2 |  instructions
```

Cela peut également se représenter sous la forme du logigramme ci-dessous sur lequel on voit bel et bien une boucle.



Exemple : Dans l'algorithme ci-dessous, tant que la valeur de N est plus petite que 10, on la multiplie par deux et on arrête dès qu'elle devient plus grande que 10. On peut résumer l'évolution de la boucle par le tableau ci-dessous.

Algorithme 6 : Exemple **Tant que**

```
1 N ← 1
2 Tant que N < 10 :
3 |  N ← 2 × N
4 Renvoyer : N
```

N	1	2	4	8	16
Condition	V	V	V	V	F

À $N = 16$, la condition n'est plus vérifiée et la boucle s'arrête, l'algorithme affiche en sortie la dernière valeur de N : 16.

Début et fin d'une boucle inconditionnelle

Lorsqu'on manipule des boucles **Tant que**, il faut être vigilant à deux choses.

1. Que la boucle puisse bien démarrer. En effet, si la condition de la boucle n'est pas vérifiée au départ, elle ne démarre pas et est donc ignorée : on passe à la suite du programme. Par exemple :

Algorithme 7 : Boucle ignorée

```
1 N ← 1
2 Tant que N > 10 :
3 |  N ← 2 × N
4 Renvoyer : N
```

Ici, N n'est pas plus grand que 10 et donc la condition de la boucle étant fausse, elle ne démarre pas. On passe alors directement à l'instruction de la ligne 4 pour afficher la valeur de N, i.e. 1.

2. Que le boucle ait une fin. En effet, si la condition est toujours vérifiée, la boucle ne s'arrête jamais : elle est infinie. Par exemple :

Algorithme 8 : Boucle infinie

```
1 N ← 1
2 Tant que N < 10 :
3 |   N ← N − 1
4 Renvoyer : N
```

Ici, N est toujours plus petit que 10 ($1, 0, -1, \dots$) et donc la condition de la boucle étant toujours vraie, elle continue indéfiniment.

Remarque : pour qu'une boucle **Tant que** ne soit pas infinie, il est important que la condition la définissant cesse d'être Vraie pour devenir Fausse à un moment voulu par le programmateur.

1.5.2 Boucles non conditionnelles

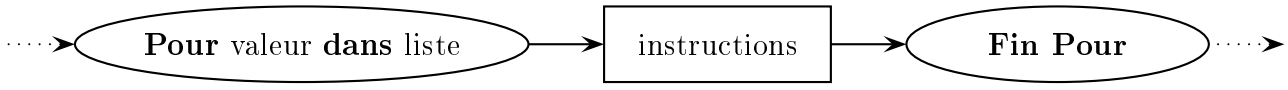
Une boucle inconditionnelle est une boucle dans laquelle sont prédéterminées les valeurs **pour** lesquelles va se réaliser le bloc d'instructions qu'elle contient. Ces boucles sont communément appelées boucles **Pour**. Les valeurs à parcourir sont précisées généralement sous forme de listes ou en donnant la première et la dernière. Comme les valeurs sont définies avant le début de la boucle, il n'y a pas de problèmes de début et de fin à considérer contrairement aux boucles **Tant que**.

En pseudo-code, on retrouvera les boucles **Tant que** de la façon suivante.

Algorithme 9 : Structure boucle **Pour**

```
1 Pour valeur dans liste :
2 |   instructions
```

Remarque : une boucle **Pour** peut toujours se ramener à une Boucle **Tant que**, elle n'a donc pas schéma de logigramme ; on peut utiliser celui de la boucle **Tant que** ou ceux de début et de fin de séquences pour la démarquer comme ci-dessous (on perd cependant l'aspect visuel de la boucle).



Exemple : Le programme ci-dessous affiche successivement "Jour 1", "Jour 2" et "Jour 3".

Algorithme 10 : Exemple **Pour** 1

```
1 Pour numéro_jour Allant de 1 à 3 :
2 |   Afficher("Jour {numéro_jour}")
```

Remarque : si la variable du Pour n'apparaît pas dans les instructions répétées, alors on retrouve le même résultat à chaque étape. C'est ce que l'on peut voir dans l'exemple ci-dessous qui une traduction en pseudo-code de l'enchaînement des trois avancées et sauts que le pirate doit effectuer dans l'exemple initial. La variable « nombre » est complètement muette, elle ne sert qu'à parcourir l'ensemble de valeurs {1, 2, 3}.

Algorithme 11 : Exemple **Pour** 2

```
1 Pour nombre Allant de 1 à 3 :
2     Avancer()
3     Sauter()
```

1.6 Attendus et savoir-faire

— Identifier une variable et son type.

— Affecter une valeur à une variable, en pseudo-code. Modifier cette valeur.

— Écrire, compléter, modifier un algorithme.

— Lire et exécuter une séquence d'instructions conditionnelles, une boucle sous forme d'algorithme ou de logigramme.

— Compléter, écrire une séquence d'instructions conditionnelles, une boucle sous forme d'algorithme ou de logigramme.

— Donner des exemples de paramètres permettant d'obtenir les différentes issues d'un test.

— Déterminer si une boucle **Tant que** est bien définie : si elle a un début et une fin.

1.7 Exercices

1.7.1 Démarrage

Exercice 1.1. Pourquoi les algorithmes sont-ils écrits sous forme de pseudo-code et non directement dans un langage de programmation ?

Exercice 1.2. [Pokédex]

1. Si l'on devait programmer un pokédex, quelles seraient les informations à enregistrer pour chaque pokémon ?
2. Comment nommer les variables pour les stocker et quels seraient leurs types ?
3. Donner un exemple d'affectation en pseudo-code pour chacune de ces variables.

Exercice 1.3. Dans un jeu, vous devez jeter un dé 10 pour déterminer si votre personnage réussit ou non son attaque sur un adversaire en fonction du résultat du dé. Le résultat est donné par l'algorithme ci-dessous.

Algorithme 12 : Attaque

```
1 Si dé ≤ 2 :
2 |  votre_santé ← votre_santé / 2
3 Sinon Si 3 ≤ dé ≤ 8 :
4 |  santé_adversaire ← santé_adversaire - puissance_attaque
5 Sinon :
6 |  santé_adversaire ← 0
```

1. Donner le logigramme correspondant à cet algorithme.
2. On considère que l'on a

   — $votre\_santé = 100$; — $santé\_adversaire = 200$; — $puissance\_attaque = 120$.

   Donner pour chacune des valeurs de la variable *dé* suivantes le résultat de l'algorithme *Attaque*.

   (a) $dé = 8$; (b) $dé = 10$; (c) $dé = 2$.

Exercice 1.4. [Starter] Compléter l'algorithme ci-dessous afin qu'il donne le pokémon choisi par le rival du joueur au début du jeu en fonction du pokémon choisi par ce dernier. On se limitera à Bulbizarre, Carapuce et Salamèche.

Algorithme 13 : Starter

```
1 Si pokémon_choisi = ..... :
2 |  pokemon_rival ← .....
3 Sinon Si ..... :
4 |  .....
5 Sinon :
6 |  .....
```

Exercice 1.5. Qu'affichent les algorithmes ci-dessous en sortie ? Dessiner leurs logigrammes.

Algorithme 14 :

```
1 k ← 1
2 Tant que k < 10 :
3 |  k ← 3 × k
4 Renvoyer : k
```

Algorithme 15 :

```
1 k ← 16
2 Tant que k > 1 :
3 |  k ← k/4
4 Renvoyer : k
```

Exercice 1.6. Déterminer si la condition de l'instruction **Tant que** des algorithmes suivants est bien définie ou non (boucle infinie ou ne démarrant pas).

Algorithme 16 :

```
1 k ← 5
2 Tant que k < 10 :
3 |   Renvoyer : k
4 |   k ← (k + 1)/2
```

Algorithme 17 :

```
1 k ← 10
2 Tant que k ≤ 1 :
3 |   k ← k + 1
4 Renvoyer : k
```

Algorithme 18 :

```
1 k ← 0
2 Tant que k < 10 :
3 |   Renvoyer : k
4 |   k ← k + 1
```

Algorithme 19 :

```
1 T ←"Ah"
2 Tant que longueur(T) ≤ 10 :
3 |   T ← T + T
4 Renvoyer : T
```

1.7.2 Approfondissement

Exercice 1.7. Vous programmez un logiciel médical dans lequel un médecin pourra enregistrer des informations sur ses patients. Les médecins souhaitent pouvoir enregistrer trois types d'informations :

— d'identité ; — de contact ; — médicales.

1. Pour chacune de ces catégories, quelles sont les informations essentielles à avoir ?
2. Comment nommer les variables pour les stocker et quels seront leurs types ?
3. Donner un exemple d'affectation en pseudo-code pour chacune de ces variables.

Exercice 1.8.

1. Donner le logigramme de l'algorithme ci-dessous.
2. Que fait l'algorithme ? Donner trois exemples de couples de valeurs pour les variables *identifiant* et *authentifiant* qui permettent d'obtenir les trois résultats possibles de ce test.

Algorithme 20 : Connexion au compte de Dark Vador

```
1 Si identifiant ≠ dark.vador@empire.gouv :
2 |   Renvoyer : "Erreur : identifiant erroné."
3 Sinon Si authentifiant ≠ *c0t3Ob$cUr* :
4 |   Renvoyer : "Erreur : mot de passe erroné."
5 Sinon :
6 |   Renvoyer : "Que la Force soit avec vous Seigneur Vador."
```

Exercice 1.9. [Des pokémons] On considère que la variable *pokedex* est une liste de tous les pokémons et que ***nom()*** et ***type()*** sont des fonctions permettant d'accéder au nom et au type du pokémon donné en entrée. Que fait l'algorithme suivant ?

Algorithme 21 :

```
1 Pour pokémon dans pokédex :
2     Si type(pokémon) = "feu" :
          Renvoyer : nom(pokémon)
```

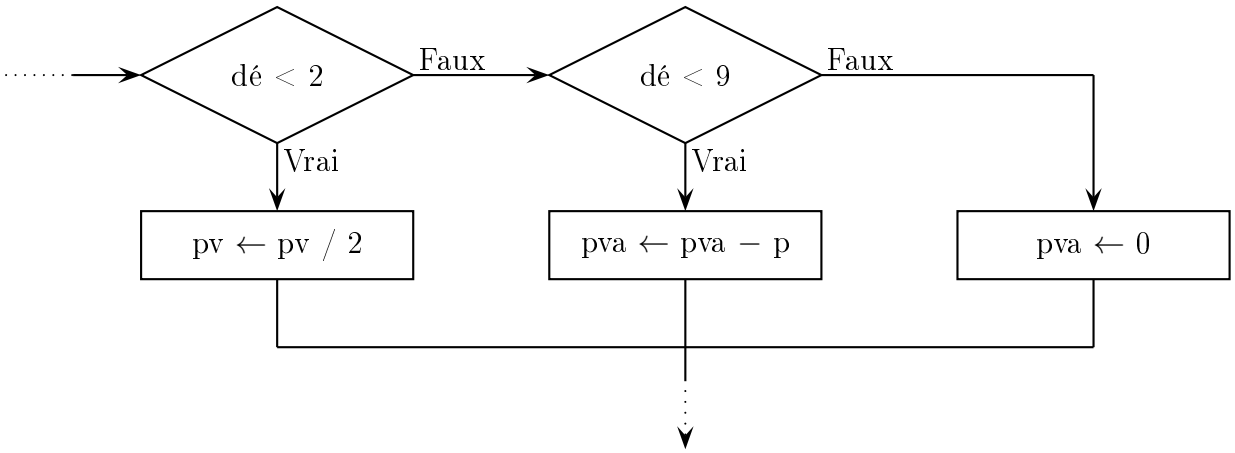
Exercice 1.10. [Encore des pokémons] Dans la lignée de l'exercice précédent, compléter l'algorithme ci-dessous afin qu"il affiche en sortie le nom des pokémons de type foudre ou dragon.

Algorithme 22 :

```
1 Pour ..... :
2     Si ..... :
          Renvoyer : .....
```

1.7.3 Entraînement

Exercice 1.11. Dans un jeu, vous devez jeter un dé 10 pour déterminer si votre personnage réussit ou non son attaque sur un adversaire en fonction du résultat du dé. Le résultat est donné par l'algorithme ci-dessous.



1. Donner l'algorithme correspondant à ce logigramme.
2. On considère que l'on a

   Vos points de vie : pv = 100 ;

   Les points de vie de l'adversaire : pva = 150 ;

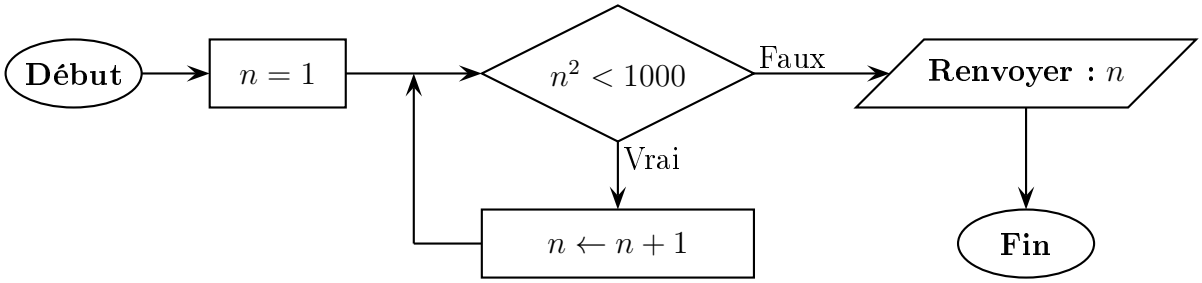
   Votre puissance : p = 120.

   Donner pour chacune des valeurs de la variable *dé* suivantes le résultat de l'algorithme *Attaque*.

(a) $dé = 8$; (b) $dé = 10$; (c) $dé = 2$.

Exercice 1.12. [Starter 2] Écrire un algorithme ou un logigramme afin qu'il donne le pokémon choisi par le rival du joueur au début du jeu en fonction du pokémon choisi par ce dernier avec les starters de la deuxième génération : Germignon, Héricendre et Kaiminus.

Exercice 1.13. Le logigramme suivant contient-il une boucle ? Si oui, laquelle ? Que fait l'algorithme correspondant ?



Exercice 1.14. Déterminer si la condition de l'instruction **Tant que** des algorithmes suivants est bien définie ou non (boucle infinie ou ne démarrant pas).

Algorithme 23 :

```
1 k ← 5
2 Tant que k > 10 :
3 |   Renvoyer : k
4 |   k ← k/2
```

Algorithme 24 :

```
1 k ← 10
2 Tant que k ≥ 1 :
3 |   k ← k²
4 Renvoyer : k
```

Exercice 1.15. [Toujours des pokémons] On suppose l'on a accès aux fonctions de l'exercice 1.9 et en plus que ***taille()*** et ***poids()*** sont des fonctions permettant d'accéder à la taille et au poids du pokémon donné en entrée.

1. Écrire un algorithme renvoyant le nom des pokémons eau mesurant plus de 1m.
2. Écrire un algorithme renvoyant le nom des pokémons pesant entre 50 et 100kg.
3. Écrire un algorithme renvoyant le nom des pokémons de type plante et, pesant moins de 20kg ou plus 30kg.