Algorithmes et programmes

Attribution - Partage dans les Mêmes Conditions :
http://creativecommons.org/licenses/by-sa/3.0/fr/

Table des matières

Introduction 3

I - Algorithmes 4

II - Exercice 6

III - Validité d'un algorithme 7

IV - Exercice 9

V - Calculabilité 10

VI - Exercice 13

VII - Optimisation d'un algorithme 14

VIII - Exercice 17

IX - Programmes 18

X - Exercice 20

XI - Quiz 21

XII - Exercice : Défi final 24

Conclusion 27

Solutions des exercices 28

Crédits des ressources 35

Contenus annexes 36

Introduction

La programmation consiste en la traduction d'un algorithme en un langage compréhensible par un ordinateur pour permettre la réalisation de cet algorithme. De ce fait il est nécessaire de comprendre ce qu'est un algorithme et comment l'analyser avant d'apprendre la programmation. La **conception** d'un algorithme et la **validation** de son fonctionnement sont des activités classiques pour un développeur.

I Algorithmes

Objectif

- Comprendre le concept d'algorithme.

Mise en situation

Aujourd'hui, il est courant d'entendre parler d'algorithmes, et ce n'est plus un terme réservé aux milieux des mathématiques et de l'informatique. Dans le même temps, ce terme a dérivé de sa signification première, et il est fréquent que sa définition ne soit pas bien comprise. En effet, le grand public parle souvent d'un algorithme pour désigner le fonctionnement obscur d'une application, ou à l'inverse une sorte de solution miracle à tous les maux. En réalité, la définition d'un algorithme est plus simple et neutre que ce que l'on peut croire.

Étymologie

Le mot *algorithme* vient du nom du mathématicien perse Al-Khwârizmî. Il n'a pas inventé la notion d'algorithme mais a eu une grande influence sur les mathématiques de son époque, par exemple en classifiant les algorithmes existants ou en diffusant l'utilisation des nombres arabes grâce à son *Traité du système de numération des Indiens*.

Définition simple Az Définition

Un algorithme est une suite finie et non ambiguë d'opérations ou d'instructions permettant de résoudre un problème. Un algorithme prend des paramètres d'entrée et retourne un résultat.

Faire tourner un algorithme Az Définition

« Faire tourner un algorithme » consiste à réaliser manuellement, une à une, les instructions et opérations de l'algorithme.

Finitude

Il est indispensable que l'algorithme soit une **suite finie d'instructions**, sinon il serait simplement impossible de résoudre le problème. En effet, il est impossible de réaliser une suite infinie d'instructions par définition.

Non-ambiguïté

Les instructions (ou les opérations) doivent être très **simples** afin de s'assurer qu'il est bien possible de réaliser l'algorithme. En d'autres termes, le résultat d'un algorithme doit être **indépendant** de l'**interprétation** de la personne le faisant tourner. Sinon, deux personnes différentes pourraient avoir deux résultats différents. De plus, une ambiguïté pourrait masquer un problème sous-jacent, potentiellement insoluble.

Instructions simples

Exemple

Voici des exemples d'instructions simples et répandues :

- Tant que x > 0 alors : ...
- Pour i de 1 à 30 faire : ...
- Si x > 0 alors : ...
- Soit x de valeur initiale 21
- x prend la valeur 84
- Incrémenter la valeur de x
- x prend la valeur de x + y

Algorithme simple

Exemple

Voici un algorithme qui prend un nombre en entrée et écrit sa table de multiplication jusqu'à 10.

```
Entrées:
    x nombre

Pour i de 1 à 9 alors:
    résultat prend la valeur i * x
    Afficher i * x = résultat
FinPour
```

Cet algorithme peut être **implémenté** en JavaScript, par exemple à travers le code suivant :

```
const x = Number(prompt("Enter the value of x :"))

for (let i = 1; i < 10; i++) {
  const result = x * i
  console.log(i, "*", x, "=", result)
}

```

Analogie avec une recette de cuisine

Pour introduire la notion d'algorithme, il est commun de le comparer à une recette de cuisine. Celle-ci correspond parfaitement à la définition ci-dessus car ses instructions sont bien finies et non ambiguës. Si ce n'est pas le cas, la recette sera impossible à réaliser.

À retenir

- Un algorithme est une suite d'instructions précisément définies permettant de réaliser une tâche.

II Exercice

Question

[solution n°1 p. 28]

On veut développer un algorithme qui calcule le prix TTC avec une réduction donnée, à partir du prix HT et du pourcentage de la réduction. On applique la réduction sur le prix TTC.

Voici le squelette de l'algorithme, il faut le compléter :

```
Entrées:
   prix_ht, un nombre à virgule
   promotion, un nombre entre 0 et 1

prix_ttc prend la valeur ? * (1.2)
prix_a_payer prend la valeur ?

Retourner ?
```

Indice :

Si un prix est réduit de 10%, le prix réduit est `prix * (1 - 0.1)`.

III Validité d'un algorithme

Objectif

- Comprendre la notion de validité d'un algorithme.

Mise en situation

L'usage des algorithmes est devenu massif et il est primordial de toujours s'assurer, voire de **prouver** que son algorithme est valide afin que son utilisation ne pose aucun problème.

Prouver un algorithme

Il est possible de fournir deux preuves mathématiques montrant qu'un algorithme est correct : une preuve d'**arrêt** et une preuve de **validité**.

La preuve d'arrêt assure que l'algorithme s'arrêtera forcément à un moment, c'est-à-dire qu'il n'y a pas de cas où il entrerait dans une boucle infinie.

La preuve de validité assure que le résultat est bien celui visé, que le but de l'algorithme est atteint quelles que soient les données d'entrée.

Plus facile à dire qu'à faire... Conseil

Ces preuves sont souvent très complexes et ne sont pas nécessaire dans un usage quotidien. Un développeur utilise la plupart du temps des algorithmes simples ou de légères adaptations d'algorithmes connus et déjà prouvés. Ces preuves sont nécessaires essentiellement dans le milieu de la recherche en informatique ou pour des applications très critiques, comme dans un logiciel de contrôle de fusée.

Comment prouver un algorithme Complément

La logique de Hoare est le formalisme le plus répandu pour réaliser les preuves de validité et d'arrêt.

Validité Az Définition

Un algorithme est considéré comme valide si le résultat renvoyé est le bon pour toute entrée possible. Il est donc important de bien spécifier le **domaine de définition** des entrées. Si une entrée est un nombre, il faut préciser si c'est un entier, un entier positif, un réel, etc. Un algorithme donné pourrait fonctionner avec des entiers seulement et il est important que cela soit précisé afin que l'algorithme soit considéré comme valide.

Vérifier empiriquement la validité

Méthode

Bien qu'une preuve formelle ne soit pas nécessaire, il est possible très simplement de vérifier la validité. Pour ce faire, il faut vérifier la validité pour quelques valeurs classiques et des valeurs dites extrêmes.

Exemple de vérification empirique

Exemple

L'algorithme ci-dessous récite le poème donné à l'envers. Les valeurs classiques seront des chaînes de caractères non vides. La valeur extrême à tester est une chaîne de caractères vide.

Après avoir vérifié ces cas, le développeur est assuré assez fortement de la validité de son algorithme.

```
Entrée:
    poème, chaine de caractère

n = longueur(poème)
Pour i de 0 à n-1 alors:
    Afficher poème[n-i]
FinPour
```

```
const poem = prompt("Entrer a poem")
const n = poem.length
for (let i = 1; i <= n; i++) {
  console.log(poem[n - i])
}

```

Validité et tests unitaires

Complément

Dans un contexte professionnel, un développeur peut mettre en place des **tests unitaires** automatisés qui permettent de vérifier empiriquement la validité de l'algorithme à chaque modification du code.

À retenir

- Un algorithme est valide s'il retourne toujours le résultat attendu pour toutes les entrées possibles.
- Il est possible de vérifier empiriquement la validité en exécutant l'algorithme pour quelques valeurs précises notamment les valeurs extrêmes.

IV Exercice

Nous mettons en place un système de notation pour des évaluations à distance. Pour améliorer ce système, nous souhaitons ajouter un système de mentions et nous devons développer l'algorithme qui retourne la mention en fonction de la note qu'on lui fournit. La notation est sur un barème de 0 à 20.

Question 1

[solution n°2 p. 28]

D'un point de vue informatique, la note sera vue comme un entier sans plus de précision. Ainsi, un utilisateur pourra entrer n'importe quel entier. Que devra faire l'algorithme pour être sûr que le domaine de validité de ses entrées est respecté ?

Indice :

L'utilisateur peut entrer une note invalide, que doit donc faire l'algorithme avant tout calcul ?

Voici l'algorithme :

```
Entrée:
    note, un entier

Si note < 0 ou note > 20 alors:
    Retourner "Erreur: note invalide"
FinSi

Si note < 10 alors:
    Retourner "Non admis"
FinSi

Si note entre 10 et 12 exclu alors:
    Retourner "Admis"
FinSi

Si note entre 12 et 14 exclu alors:
    Retourner "Assez bien"
FinSi

Si note entre 14 et 16 exclus alors:
    Retourner "Bien"
FinSi

Si note entre 16 et 20 exclus alors:
    Retourner "Très bien"
FinSi

Si note = 20 alors:
    Retourner "Parfait!"
FinSi
```

Question 2

[solution n°3 p. 28]

Donner les valeurs extrêmes à vérifier.

Question 3

[solution n°4 p. 28]

Donner 3 valeurs « classiques ».

V Calculabilité

Objectif

- Découvrir le concept de calculabilité.

Mise en situation

L'apparition de la théorie de l'Informatique est très étroitement liée à la théorie de la Calculabilité.

Fonction calculable Fondamental

Une fonction *f* est calculable si, pour tout argument *x*, il est possible d'obtenir *f(x)* en un nombre fini d'étapes.

Algorithme et fonction calculable

On comprend très vite qu'un algorithme représente une fonction calculable car on peut faire un parallèle entre le nombre fini d'étape de la fonction et la suite finie d'instructions de l'algorithme. Ainsi, le travail de l'algorithme est assimilable à un calcul.

Les algorithmes et leurs problèmes

Un algorithme a toujours pour but de résoudre un problème donné, que ce dernier soit compliqué (plus court chemin dans un graphe) ou très simple (une division euclidienne). La théorie de la Calculabilité est essentielle pour caractériser la nature des problèmes.

Indécidabilité

Il existe des choses impossibles à calculer, c'est-à-dire des problèmes insolubles. Ainsi, il existe des problèmes dont il est possible de calculer la solution et d'autres non. L'exemple le plus classique de problème indécidable est le problème de l'arrêt : il n'existe pas d'algorithme déterminant si n'importe quel algorithme donné se termine.

Classe de problèmes

Les problèmes sont classiquement classés en fonction des algorithmes qui les résolvent :

- **P** : il existe un algorithme qui résout le problème et se termine en temps polynomial, c'est-à-dire qui n'explose pas en fonction de la taille des données d'entrées.
- **NP** : il existe un algorithme qui vérifie la solution d'un problème en temps polynomial mais il faudra un temps exponentiel pour trouver une solution.
- **Indécidables** : Il n'existe pas d'algorithme pour résoudre le problème.

Les problèmes NP

Les problèmes **NP** sont considérés comme impossibles à résoudre dans un temps humain.

Pour une taille importante de données d'entrées, l'exécution de l'algorithme par un ordinateur moderne pourra prendre des années, des siècles, voire plus... Il existe de nombreux problèmes **NP** et il est nécessaire d'adopter des optimisations pour trouver une solution « sub-optimale » en un temps polynomial. Une solution « sub-optimale » est une solution qui s'approche de la solution idéale.

Faut-il déplacer ma reine ou mon fou ?

Exemple

Un problème **NP** très connu est celui du calcul du meilleur coup dans le jeu d'échecs.

Pour chaque position aux échecs, il existe un nombre extrêmement grand de positions possibles. Ainsi, il est possible théoriquement de calculer toutes ces positions. En revanche, l'algorithme mettra des siècles à se terminer. Les intelligences artificielles telles que Deep Blue calculent toutes les positions possibles sur les *x* prochains tours et choisiront celui qui offre la meilleure position dans *x* tours. Ce qui rend les intelligences artificielles meilleures que les humains est leur capacité à se projeter plus loin que les humains. Les joueurs humains calculent également les potentielles positions futures mais peuvent calculer beaucoup moins de tours que les ordinateurs modernes.

Résoudre un Rubik's Cube

Exemple

Trouver la solution d'un Rubik's Cube est un problème **P**. Il est très simple de le résoudre, en témoignent les records du monde de vitesse. Même avec des cubes un peu plus grands, le nombre de possibilités croît beaucoup moins que le nombre de coups aux échecs.

Trouver son chemin

Exemple

Les applications GPS se sont largement démocratisées et résolvent toutes un même problème : trouver le plus court chemin entre A et B. Bien qu'il existe une infinité de chemins entre A et B, ce problème est **P** car il n'est pas nécessaire d'essayer tous les chemins contrairement aux échecs où il faut calculer l'ensemble des coups possibles. Il existe des algorithmes qui permettent d'exclure rapidement les solutions impossibles et d'arriver au plus court chemin en temps polynomial. On comprend donc que la grande différence entre **P** et **NP** réside dans la nature exponentielle du nombre de calculs à effectuer pour arriver à la solution.

P = NP ?

Complément

Il existe un problème mathématique considéré comme un des sept problèmes du millénaire par les mathématiciens : *« **P=NP ?** »* Ce problème vise à affirmer ou infirmer formellement que les problèmes **NP** sont, ou pas, des problèmes **P**. La preuve de l'égalité impliquerait qu'il existe un algorithme en temps polynomial pour résoudre des problèmes tels que le cassage des chiffrements modernes sur lesquels est fondé le système de transaction bancaire.

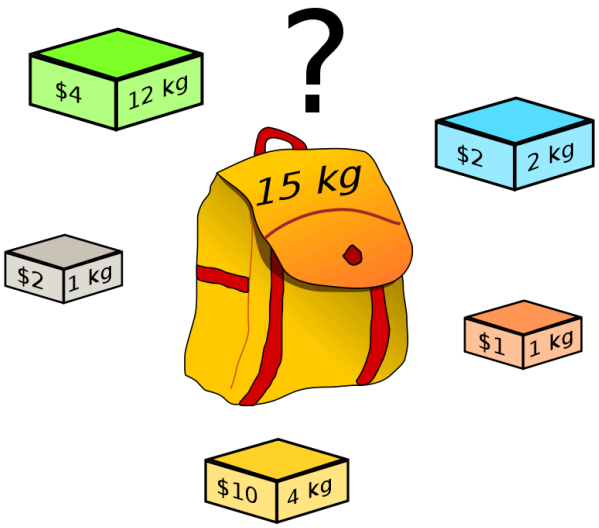
À retenir

- Un algorithme est assimilable à un calcul.
- Il existe des problèmes impossibles à résoudre mais il existe également des problèmes possibles à résoudre en théorie, mais en pratique (pas en un temps limité).

VI Exercice

Voici le problème du sac à dos : nous avons un sac à dos pouvant supporter un poids maximum, nous possédons une série d'objet ayant chacun un poids. Le problème à résoudre est alors : *quels objets choisir pour transporter le sac le plus lourd possible sans dépasser son maximum ?*.

Dans l'exemple ci-dessous, le poids maximal que peut supporter le sac est 15 kg.



Le problème du sac à dos

Question 1

[solution n°5 p. 28]

Dans le cas de l'exemple ci-dessus, quels objets faudrait-il prendre pour avoir le sac le plus lourd possible sans dépasser 15 kg ?

Indice :

Il y a deux solutions possibles.

Question 2

[solution n°6 p. 28]

Quel est la classe de ce problème ?

Indice :

Pour résoudre ce problème, il faut tester toutes les combinaisons d'objets une à une. Une fois toutes les combinaisons calculées il faudra conserver celle qui permet d'atteindre le poids le plus haut sans dépasser le poids maximum supporté par le sac.

Indice :

Imaginer que nous ayons pas 5 objets possibles mais 10, 20, 100 ou même 1000. La croissance du nombre de possibilités à vérifier serait-elle exponentielle ?

VII Optimisation d'un algorithme

Objectif

- Découvrir l'optimisation algorithmique.

Mise en situation

La complexité algorithmique est une notion importante pour qui veut développer des applications capables de **passer à l'échelle**. Lors des étapes de développements, il est courant de tester les algorithmes sur de petits jeu de données. Mais si l'algorithme s'avère trop complexe, l'utilisation sur un plus grand volume de données peut s'avérer beaucoup plus lente ou gourmande en ressources. Il convient donc de réaliser une analyse de la complexité d'un algorithme avant de l'utiliser, et éventuellement de l'adapter et l'optimiser pour permettre un fonctionnement conforme à ce qui est attendu.

Complexité algorithmique Rappel

La ***complexité algorithmique*** *(cf. p.36)* est une estimation du nombre d'opérations élémentaires pour résoudre un problème en fonction de la taille des données d'entrée, notée ***n***. On note **O() l'ordre de grandeur** de la complexité d'un algorithme. Par exemple, **O(*n*)** signifie que le nombre d'opérations est de l'ordre de *n* (si on a 1000 données en entrée, on s'attend à environ 1000 opérations).

Complexité en temps

La complexité en temps est celle qui vient en premier dans l'esprit des développeurs. Elle donne un ordre d'idée du temps d'exécution d'un algorithme. L'estimer permet de chercher des optimisations qui permettront de réduire le temps de calcul. Il est parfois très simple de diviser par 2 le temps de calcul, ce qui se ressent du point de vue de l'utilisateur.

Complexité en espace

La complexité en espace est moins connue mais toute aussi importante. Elle donne un ordre de grandeur du nombre de « case mémoire » qui seront nécessaires au calcul. Réduire au maximum cette complexité évitera que le programme ne dysfonctionne car il n'y a pas plus assez de mémoire disponible.

Exemple

Soit deux algorithmes qui calculent la somme des entiers de 1 à ***n***.

- **n** est donné en entrée de l'algorithme.
- Il existe une formule mathématique permettant d'obtenir la somme des n premiers entiers (**n * (n - 1) / 2**).

Le premier algorithme dit naïf est en **O(n)**, le second utilisant la formule mathématique est en **O(1)**. Ainsi, il faudra préférer le second algorithme car pour une complexité en espace identique, il possède une complexité en temps bien plus faible.

```
Algo Somme1aN:

Entrée:
    n, un entier positif

somme prend la valeur 0
Pour i de 1 à n alors:
    somme prend la valeur somme + i
FinPour

Retourner somme
```

```
Algo Somme1aN:

Entrée:
    n, un entier positif

somme prend la valeur n*(n+1)/2

Retourner somme
```

Méthode

- La suppression de boucles est une bonne solution pour réduire la complexité d'un algorithme.
- Il est également possible de réduire la complexité en supprimant des instructions simples. Par exemple, on parle de factorisation lorsqu'on condense plusieurs conditions en une seule instruction. Le nombre de calculs de comparaisons est réduit.

Ainsi, pour optimiser un algorithme, il faut commencer par supprimer les boucles superflues puis, dans un second temps, factoriser le code qui peut l'être.

Exemple

Dans l'exemple ci-dessous, le second algorithme réduit la complexité en temps car il fusionne les deux conditions en une seule. De plus, la complexité en espace est également réduite car il n'utilise plus de variable pour stocker le résultat.

```
Algo PeutConduire1:

Entrées:
    age, un entier
    permis, un booléen

résultat prend la valeur Vrai

Si age < 18 alors:
    resultat prend la valeur Faux
FinSi

Si age > 18 et non permis alors
    resultat prend la valeur Faux
FinSi

Retourner resultat
```

```
Algo PeutConduire2:

Entrées:
    age, un entier
    permis, un booléen

Si (age < 18) ou (age > 18 et non permis) alors:
    Retourner Faux
Sinon
    Retourner Vrai
FinSi
```

Choisir entre espace et temps Complément

Il arrive de devoir choisir entre un algorithme qui minimise le temps et un autre l'espace. Le choix devra être orienté par le domaine d'application. En informatique embarquée, l'espace est très limité donc le temps pourrait être sacrifié au profit d'un espace mémoire moindre. En Big Data, l'espace mémoire à disposition est gigantesque et l'accélération des calculs sera souhaitée.

À retenir

- Il est possible d'estimer le temps d'exécution d'un algorithme ainsi que l'espace mémoire dont il a besoin grâce à la complexité algorithmique.
- Un développeur doit chercher un algorithme minimisant le temps et l'espace.
- Selon son domaine d'application, il pourra privilégier le temps ou l'espace.

VIII Exercice

Voici deux algorithmes qui cherchent l'indice du maximum d'une liste :

```
Algo IndMax1:

Entrée:
    l, liste d'entiers

max prend la valeur l[1]
ind_max prend la valeur 1
Pour i de 2 à longueur(l) alors:
    Si max < l[i] alors:
        max prend la valeur l[i]
        ind_max prend la valeur i
    FinSi
FinPour

Retourner ind_max
```

```
Algo IndMax2:

Entrée:
    l, liste d'entiers

max prend la valeur l[1]
Pour i de 2 à longueur(l) alors:
    Si max < l[i] alors:
        max prend la valeur l[i]
    FinSi
FinPour

Pour i de 1 à longueur(l) alors:
    Si l[i] == max alors:
        Retourner i
    FinSi
FinPour
```

Question 1

[solution n°7 p. 28]

Quel algorithme a la plus petite complexité en temps ?

Question 2

[solution n°8 p. 29]

Quel algorithme à la plus petite complexité en espace ?

Question 3

[solution n°9 p. 29]

Combiner ces deux algorithmes pour trouver un algorithme avec une seule boucle et une seule variable.

IX Programmes

Objectif

- Comprendre la différence entre un algorithme et un programme.

Mise en situation

L'algorithme et le programme sont deux choses distinctes. En effet les algorithmes existent depuis des milliers d'années, et sont simplement des suites d'opérations qui permettent de résoudre un problème, exactement comme une recette de cuisine. Les programmes, eux, sont beaucoup plus récents car ce sont les implémentations informatiques d'algorithme, pour permettre leur résolution par un ordinateur. Selon le paradigme ou le langage utilisé, les implémentations d'un même algorithme peuvent être différentes et offrir des performances qui ne sont pas les mêmes. C'est donc un choix qu'il est important de faire avant de se lancer dans l'écriture d'un programme.

Programme Az Définition

Un **programme** est la traduction en code exécutable d'un algorithme. On parle également d'**implémentation** d'un algorithme. Ainsi pour un algorithme donné, il existe plusieurs implémentations. Les choix d'implémentation peuvent différer d'un langage à un autre ou au sein d'un même langage.

Niveau de traduction

Un programme peut être vu à *deux niveaux* (cf. p.39) : le code du programmeur et le langage machine. En effet, le code du programmeur sera traduit par un compilateur ou un interpréteur qui rendront les instructions exécutables par la machine. Le code est dit « haut-niveau » car il est proche de l'écriture littérale de l'instruction mais incompréhensible en tant que tel pour la machine. Le langage machine est bien moins compréhensible pour un être humain mais la machine pourra l'exécuter car ce sont des instructions en binaire.

Paradigme de programmation

Les langages de programmation peuvent être classés en plusieurs catégories nommées paradigmes. Ces catégories dépendent des caractéristiques du langage et de sa manière d'aborder la résolution d'un problème. Les paradigmes les plus connus sont : impératif, fonctionnel, logique, etc.

Langages impératifs Exemple

Les langages impératifs sont les plus connus et les plus répandus car ils ont un fonctionnement très proche de l'algorithmie classique. On peut citer :

- C/C++
- Python
- Javascript
- Java

- PHP

Comparaison de deux implémentations

Exemple

Voici l'implémentation du calcul de la suite de Fibonacci en Python puis en JavaScript. On voit que la syntaxe diffère mais également certaines instructions. Par exemple, Python parcourt les éléments d'une liste un à un, alors que JavaScript passe par un indice représentant le numéro de case.

```python
"""Python : Suite de Fibonacci. """
n = int(input("Entrer un entier supérieur à 1:"))

fibo = [0]*(n+1)
fibo[0] = 0
fibo[1] = 1

for i in range(2,n+1):
  fibo[i] = fibo[i-1] + fibo[i-2]

for number in fibo:
  print(number)
```

```javascript
/** JavaScript : Suite de Fibonacci. */
const n = Number(prompt('Entrer un entier supérieur à 1:'))

var fibo = new Array(n + 1)
fibo[0] = 0
fibo[1] = 1

for (let i = 2; i < n + 1; i++) {
  fibo[i] = fibo[i - 1] + fibo[i - 2]
}

for (let i = 0; i < n + 1; i++) {
  console.log(fibo[i])
}

```

Choisir un langage et un paradigme

Complément

Les paradigmes et les langages ayant chacun leurs spécificités, il faut choisir un langage non pas parce qu'il est connu du développeur mais parce qu'il permettra de répondre au mieux au problème posé.

- Par exemple pour de l'informatique embarquée, il faut favoriser le C/C++ car ce sont des langages très rapides et demandant moins de puissances.
- En revanche pour du Machine Learning, le Python rend l'utilisation des algorithmes très simples pour le programmeur (au détriment d'une légère baisse de performance par rapport au C/C++).

À retenir

- Un programme est la traduction d'un algorithme à l'aide d'un langage de programmation.
- Selon le langage et son paradigme, la traduction ne sera pas toujours identique.

X Exercice

Le chiffre de César est un chiffrement très simple qui décale chaque lettre d'un nombre constant. Par exemple, pour un chiffre de César de rang 3, un ***a*** devient un ***d***, un ***b*** devient un ***e***, un ***c*** devient un ***f***, etc.

Voici l'algorithme de déchiffrement pour un chiffre de César de rang 3 :

```
Algo César:

Entrée:
    texte_chiffré, chaîne de caractère

Pour tout i de 1 à longueur(texte_chiffré) faire:
    Si texte_chiffré[i] = " " alors:
        Afficher " "
    Sinon:
        lettre_chiffrée prend la valeur texte_chiffré[i]
        lettre_déchiffrée prend la valeur lettre_chiffrée - 3 rangs
        Affiche lettre_déchiffrée
    FinSi
FinPour
```

Question 1

[solution n°10 p. 29]

Voici les morceaux de JavaScript. Les mettre dans le bon ordre pour obtenir le programme permettant de déchiffrer le chiffre de César.

```
const cipher = (prompt('Entrer a cipher')).toLowerCase()
---
if (cipher[i] === ' ') {[CODE]} else {[CODE]}
---
for (let i = 0; i < cipher.length; i++){[CODE]}
---
console.log(" ")
---
const encryptedLetter = cipher.charCodeAt(i)
const realLetter = String.fromCharCode(((encryptedLetter - 97 + 26 - 3 ) % 26) + 97)
console.log(realLetter)
```

Question 2

[solution n°11 p. 29]

Déchiffrer le texte suivant : « yrxv srxyhc dffhghu dx txlcc ghvrupdlv ».

XI Quiz

Exercice 1 : Quiz - Culture

[solution n°12 p. 29]

Exercice

Quelles sont les caractéristiques d'un algorithme ?

- **A** Suite finie d'instructions
- **B** Instructions non ambiguës
- **C** Instructions en binaire
- **D** Instructions sous la forme d'opérations arithmétiques (additions, multiplication, etc.)

Exercice

Laquelle de ces instructions n'est pas une instruction simple et non ambiguë ?

- **A** Tant que *x* est différent de 0 faire : ....
- **B** Affecter à *x* la valeur 3
- **C** Pour *i* de 1 à 100 faire : ...
- **D** Doubler la liste *(3, 12, 1)*
- **E** *x* prend la valeur *max(3, 12, 1)*

Exercice

Quelle est la classe du problème suivant : trouver le coup optimal dans une partie du jeu de Go ?

- **A** P
- **B** NP
- **C** Indécidable

Exercice

Quelle est la classe du problème suivant : trouver le plus court chemin pour se rendre à un endroit donné ?

- **A** P
- **B** NP
- **C** Indécidable

Exercice

Quel est le paradigme principal du langage Python ?

- **A** Logique
- **B** Déclaratif
- **C** Dynamique
- **D** Impératif

Exercice 7 : Quiz - Méthode

[solution n°13 p. 31]

Exercice

J'ai développé un algorithme qui calcule la somme des **n** premiers entiers positifs. Quelle est la valeur limite à tester ?

Exercice

Si mon algorithme est valide pour les valeurs limites, est-il nécessaire de tester d'autres valeurs ?

- **A** Oui
- **B** Non

Exercice

Tester un algorithme pour des valeurs classiques et des valeurs extrêmes représente-il une preuve ?

- **A** Oui
- **B** Non

Exercice

Quelle est la complexité en temps de l'algorithme suivant ?

```
const str = prompt('Enter a word')
let palindrom = true
for (var i=0; i < (Math.floor(str.length / 2) + 1); i++) {
  const opposed = str.length - 1 - i
  if (str[i] != str[opposed]) {
    palindrom = false
    break
  }
}
if (palindrom) {
  console.log('Palindrom')
} else {
  console.log('Not a palindrom')
}
```

Exercice 12 : Quiz - Code

[solution n°14 p. 32]

Exercice

En quel langage de programmation est écrit ce bout de code ?

```
for element in liste:
    print(element)
```

- **A** C
- **B** Python
- **C** JavaScript
- **D** PHP

Exercice

Traduire l'instruction algorithmique suivante en JavaScript :

```
Si x est supérieur à 0
```

XII Exercice : Défi final

On veut écrire un algorithme qui indique si on pourra sortir en vélo demain selon la météo du jour (niveau du soleil S, température T et niveau de pluie P) et si la personne est déjà sortie aujourd'hui. Les niveaux sont entre 0 et 100.

L'algorithme est en deux parties. D'abord il décide s'il fera beau demain puis utilise cette information pour décider de la sortie ou non le lendemain.

Question 1

[solution n°15 p. 32]

Par défaut, il ne fera pas beau demain. Voici les conditions (il faut juste que l'une d'entre elles soit vraie) pour qu'il fasse beau :

- P > 70
- S > 50 et P < 30
- P < 70 et T > 20
- S + P < 50
- T > 30

En revanche, si la somme des niveaux de soleil et de pluie est supérieure à 150, il ne fera pas beau le lendemain même si l'une des conditions ci-dessus est vraie.

Enfin, s'il fait beau demain, l'algorithme affiche « *Sortie demain* ». S'il ne fait pas beau, l'algorithme affiche « *Sortie demain* » seulement si la personne n'est pas sortie aujourd'hui. Sinon, il affiche « *Pas sortie demain* ».

Écrire un algorithme permettant de décider si la personne doit sortir le lendemain.

Indice :

L'algorithme doit préciser les entrées qu'il reçoit, ainsi que les conditions sur ces entrées. Par exemple, une entrée nommée n qui doit être supérieure à 0 sera notée :

```
Entrées:
    n, un entier supérieur à 0
```

Indice :

L'algorithme devra tester chacune des conditions une par une. Vous pouvez utiliser la structure suivante pour évaluer les conditions :

```
Si <condition> alors:
    ...
FinSi
```

Indice :

Il est utile d'utiliser une variable `beauTempsDemain` qui stocke l'information du temps du lendemain et qui est modifiée par la série de conditions, avec la syntaxe suivante :

```
Attribuer à beauTempsDemain la valeur Vrai
```

ou

```
Attribuer à beauTempsDemain la valeur Faux
```

Indice :

Si la somme des niveaux de soleil et de pluie est supérieur à 150, il ne fera pas beau le lendemain quoi qu'il arrive. Cette condition est donc à tester en dernier.

Question 2

[solution n°16 p. 33]

Voici le squelette de l'algorithme en JavaScript.

Compléter le code JavaScript de la première partie de l'algorithme.

```
const S = Number(prompt('Entrer le niveau de soleil actuel'))
const T = Number(prompt('Entrer la température actuelle'))
const P = Number(prompt('Entrer le niveau de pluie actuel'))
const sortieAujourdhui = prompt('Êtes-vous sorti aujourd\'hui? (O ou N)') === 'O'

// Première partie
let beauTempsDemain = false

if (P > 70) {
  beauTempsDemain = true
}
if (...) {
   ...
}
if (...) {
   ...
}
if (...) {
   ...
}
if (...) {
   ...
}

if (...) {
   ...
}


// Deuxième partie de l'algorithme
...
```

Indice :

L'opérateur « *et* » en Javascript s'écrit &&.

Question 3

[solution n°17 p. 33]

Peut-on réduire la complexité en temps de cette première partie ?

Indice :

Est-il nécessaire d'avoir un **if** pour chaque condition ?

Question 4

[solution n°18 p. 34]

Traduire la seconde partie de l'algorithme.

Indice :

```
1 if (...) {...} else {...}
```

Indice :

Pour afficher un texte on peut utiliser l'instruction :

```
1 console.log("toto")
```

Question 5

[solution n°19 p. 34]

Voici les conditions météorologiques d'aujourd'hui : S = 50, T = 17, P = 38. Sachant que je suis sorti aujourd'hui, est-ce que je devrais sortir demain ?

Conclusion

Les algorithmes font aujourd'hui pleinement partie de nos sociétés. Il est donc nécessaire de bien les comprendre et de savoir les analyser, pour pouvoir correctement les implémenter en programmes. Pour cela on utilise différentes méthodes qui permettent de s'assurer de la validité d'un algorithme, de sa capacité à être résolu et de donner un ordre de grandeur du temps et des ressources nécessaires. Tous ces outils d'analyse permettent ensuite de développer un programme tout en s'assurant de son bon fonctionnement.

Solutions des exercices

Solution n°1

[exercice p. 6]

```
Entrées:
   prix_ht, un nombre à virgule
   promotion, un nombre entre 0 et 1

prix_ttc prend la valeur prix_ht * (1.2)
prix_a_payer prend la valeur prix_ttc * (1 - promotion)

Retourner prix_a_payer
```

Solution n°2

[exercice p. 9]

L'algorithme doit d'abord vérifier que la note fournie est bien comprise entre 0 et 20.

Solution n°3

[exercice p. 9]

Il faudrait vérifier :

- Un nombre inférieur à 0. Par exemple *« -1 »*.
- Un nombre supérieur à 20. Par exemple *« 30 »*.
- Chaque nombre à partir duquel il y a un changement de comportement : 0, 10, 12, 14, 16, 20.

Solution n°4

[exercice p. 9]

Il s'agirait de valeurs intermédiaires qui pourraient être 5, 13, 18 ou n'importe quel nombre qui ne rentre pas dans les catégories présentées comme extrêmes à la question précédente.

Solution n°5

[exercice p. 13]

Il faut prendre soit les objets vert, rouge et bleu soit les vert, gris et bleu. Dans les deux cas, on atteint 15 kg.

Solution n°6

[exercice p. 13]

C'est un problème NP. En effet, le temps pour résoudre sera exponentiel en fonction du nombre de poids. Pour résoudre ce problème, il faut calculer le poids de toutes les combinaisons possibles d'objets. Or, le nombre de combinaisons est exponentiel en fonction du nombre d'objets.

Solution n°7

[exercice p. 17]

`IndMax1` car il ne possède qu'une boucle.

Solution n°8

[exercice p. 17]

`IndMax2` car il ne possède une variable de moins

Solution n°9

[exercice p. 17]

```
Algo IndMax3:

Entrée:
    l, liste d'entiers

ind_max prend la valeur 1
Pour i de 2 à longueur(l) alors:
    Si l[ind_max] < l[i] alors:
        ind_max prend la valeur i
    FinSi
FinPour

Retourner ind_max
```

Solution n°10

[exercice p. 20]

```
const cipher = (prompt('Entrer a cipher')).toLowerCase()

for (let i = 0; i < cipher.length; i++) {
  if (cipher[i] === ' ') {
    console.log(' ')
  } else {
    const encryptedLetter = cipher.charCodeAt(i)
    const realLetter = String.fromCharCode(((encryptedLetter - 97 + 26 - 3) % 26) + 97)
    console.log(realLetter)
  }
}

```

Solution n°11

[exercice p. 20]

Ce qui donne : « vous pouvez acceder au quizz desormais ».

Solution n°12

[exercice p. 21]

Exercice

Quelles sont les caractéristiques d'un algorithme ?

- **A** Suite finie d'instructions
- **B** Instructions non ambiguës
- **C** Instructions en binaire

D Instructions sous la forme d'opérations arithmétiques (additions, multiplication, etc.)

Exercice

Laquelle de ces instructions n'est pas une instruction simple et non ambiguë ?

A Tant que *x* est différent de 0 faire : ....

B Affecter à *x* la valeur 3

C Pour *i* de 1 à 100 faire : ...

D

◯ Doubler la liste *(3, 12, 1)* — Cette instruction n'est pas claire : faut-il concaténer la liste deux fois ou multiplier chacun de ses membres par deux ?

E

x prend la valeur *max(3, 12, 1)* — Cette instruction peut être considérée comme non ambiguë car il existe un algorithme très simple et répandu permettant de réaliser l'opération.

Exercice

Quelle est la classe du problème suivant : trouver le coup optimal dans une partie du jeu de Go ?

A P

◉ **B** NP — A l'instar des échecs, ce problème est NP.

C Indécidable

Exercice

Quelle est la classe du problème suivant : trouver le plus court chemin pour se rendre à un endroit donné ?

A

◯ P — Ce problème est P (polynomial). L'algorithme de Djikstra permet de résoudre ce problème.

B NP

C Indécidable

Exercice

Quel est le paradigme principal du langage Python ?

A Logique

B Déclaratif

C Dynamique

D Impératif

Solution n°13

[exercice p. 22]

Exercice

J'ai développé un algorithme qui calcule la somme des **n** premiers entiers positifs. Quelle est la valeur limite à tester ?

0

La somme des 0 premiers entiers n'est pas instinctive : l'algorithme devra faire un choix (comme par exemple renvoyer 0). Comme c'est un cas particulier, il est nécessaire de le tester. Pour toutes les autres valeurs positives, le fonctionnement est en revanche similaire.

Exercice

Si mon algorithme est valide pour les valeurs limites, est-il nécessaire de tester d'autres valeurs ?

A

Oui Il reste impératif de tester des valeurs classiques. Les valeurs limites visent à identifier si les cas particuliers sont bien traités. En revanche, il faut toujours s'assurer que les cas classiques le sont également.

B Non

Exercice

Tester un algorithme pour des valeurs classiques et des valeurs extrêmes représente-il une preuve ?

A Oui

B

Non La seule preuve ayant une valeur d'un point de vue formelle est la preuve mathématique. En revanche, un développeur pourra se suffire de la vérification empirique car, la plupart du temps, il ne fait qu'adapter des algorithmes répandus et déjà prouvés.

Exercice

Quelle est la complexité en temps de l'algorithme suivant ?

```
const str = prompt('Enter a word')
let palindrom = true
for (var i=0; i < (Math.floor(str.length / 2) + 1); i++) {
  const opposed = str.length - 1 - i
  if (str[i] != str[opposed]) {
    palindrom = false
    break
  }
}
if (palindrom) {
  console.log('Palindrom')
} else {
  console.log('Not a palindrom')
}
```

O(n)

La boucle principale va être exécutée un nombre de fois égal à la moitié des caractères de la chaîne d'entrée. Le corps de la boucle ne contient que des opérations élémentaires, la complexité est donc égale à **O(*n*/2)**. Cette complexité est équivalente à **O(*n*)**.

Solution n°14

[exercice p. 23]

Exercice

En quel langage de programmation est écrit ce bout de code ?

```
for element in liste:
    print(element)
```

A C

B Python

C JavaScript

D PHP

Exercice

Traduire l'instruction algorithmique suivante en JavaScript :

```
Si x est supérieur à 0
```

if (x > 0)

Solution n°15

[exercice p. 24]

```
Algo sortie_demain

Entrées:
    S, un entier entre 0 et 100
    T, un entier supérieur à -273
    P, un entier entre 0 et 100
    SortieAujourdhui, un booléen
```

```
Attribuer à beauTempsDemain la valeur Faux

Si P > 70 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si S > 50 et T > 15 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si P < 70 et T > 20 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si S + P < 50 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si T > 30 alors:
    Attribuer à beauTempsDemain la valeur Vrai
FinSi
Si S + P > 150 alors:
    Attribuer à beauTempsDemain la valeur Faux
FinSi

Si beauTempsDemain alors:
    Afficher "Sortie demain"
Sinon
    Si non sortieAujourdhui alors:
        Afficher "Sortie demain"
    Sinon:
        Afficher "Pas sortie demain"
```

Solution n°16

[exercice p. 25]

```
let beauTempsDemain = false

if (P > 70) {
  beauTempsDemain = true
}
if (S > 50 && T > 15) {
  beauTempsDemain = true
}
if (P < 70 && T > 20) {
  beauTempsDemain = true
}
if (S + P  < 50) {
  beauTempsDemain = true
}
if (T > 30) {
  beauTempsDemain = true
}

if (S + P > 150) {
  beauTempsDemain = false
}
```

Solution n°17

[exercice p. 25]

On pourrait factoriser les conditions avec des « ou » (`||` en JS) afin de n'avoir qu'un seul `if`.

Solution n°18

[exercice p. 26]

```
if (beauTempsDemain){
  console.log('Sortie demain')
} else {
  if (sortieAujourdhui) {
      console.log('Pas sortie demain')
  } else {
      console.log('Sortie demain')
  }
}
```

Solution n°19

[exercice p. 26]

Pas de sortie demain.

Crédits des ressources

Le problème du sac à dos p. 13
Universel - Transfert dans le Domaine Public - Dake

Contenus annexes

1. Complexité algorithmique

Objectif

- Découvrir la notion de complexité algorithmique.

Mise en situation

Lorsqu'un développeur écrit un programme, il lui est nécessaire d'estimer si ce programme est coûteux. Coûteux cela signifie par exemple qu'il nécessite beaucoup de calculs de la part du processeur. Cela peut aussi signifier qu'il nécessite beaucoup d'espace mémoire.

Savoir calculer ce coût permet de comparer plusieurs solutions alternatives entre elles. Cela permet aussi de savoir si un programme peut « passer à l'échelle » c'est à dire continuer de fonctionner correctement si on le confronte à la réalité.

Par exemple si un développeur a écrit un programme permettant de vérifier une empreinte de carte de paiement, mais que cette vérification prend plusieurs minutes ou nécessite un ordinateur doté d'une mémoire très importante, il ne sera pas possible de l'utiliser pour du paiement en ligne.

Compter le nombre d'opérations pour résoudre un problème — Exemple

Pour créer la table de 2 des **n** premiers entiers, il faut réaliser **n** calculs de type multiplication.

De manière générale, on cherche à estimer de manière théorique le nombre d'opérations à effectuer pour résoudre un problème : c'est ici que l'on parle de **complexité** d'un problème.

Complexité algorithmique — Définition

La **complexité algorithmique** est une estimation du nombre d'opérations élémentaires pour résoudre un problème en fonction de la taille des données d'entrée, notée **n**.

O() — Syntaxe

On note O() **l'ordre de grandeur** de la complexité d'un algorithme.

- O(n) signifie que le nombre d'opérations est de l'ordre de n (si on a 1000 données en entrée, on s'attend à environ 1000 opérations).
- O(n^2) signifie que le nombre d'opérations est de l'ordre de n au carré (si on a 1000 données en entrée, on s'attend à environ 1 000 000 d'opérations).

Exemple simple

Exemple

Ce code fait la somme d'une liste d'entiers.

```
function somme(number_list) {
  let res = 0;
  for (var i = 0; i < number_list.length; i++) {
      res += number_list[i];
  }
  return res;
}
```

Étant donné **n** la taille de la liste, il serait nécessaire d'effectuer **n** opérations d'addition pour obtenir la somme finale. Ici, on dira que la complexité de l'algorithme est de l'ordre de **n** et on la notera **O(n)**.

Pourquoi utiliser la complexité algorithmique ?

Remarque

La complexité algorithmique peut se voir comme l'estimation du "temps d'exécution" d'un algorithme en fonction de la taille des données d'entrée.

On suppose pour ce faire que les opérations de base (affectation, addition, etc.) ont le même temps d'exécution : estimer le temps d'exécution revient alors à estimer le nombre d'opérations de base.

L'exécution reste proportionnelle à la taille des données

Attention

La complexité ne change pas en fonction de la taille des données. En revanche, l'exécution sera évidemment plus longue si la taille des données augmente. Ainsi, pour 100 données, un algorithme en **O(n)** effectuera de l'ordre de 100 opérations. Pour 1000 données il en effectuera de l'ordre de 1000.

Estimer rapidement la complexité

Méthode

La complexité permet d'avoir une idée grossière de la durée d'un algorithme. Par exemple, si on détermine que pour une liste de **n** éléments, il faut effectuer **2×n** opérations, on simplifiera en disant que la complexité est **de l'ordre** de **O(n)**.

La raison principale est que les constantes ne changent pas fondamentalement l'ordre de grandeur de la complexité. Par exemple, **$O(n^2)$** et **$O(2 \times n^2)$** sont très proches, et c'est surtout la valeur de **n** qui influencera la durée finale.

Complexités usuelles des algorithmes

Fondamental

On retrouve très régulièrement les complexités suivantes :

- **O(1)** : si le nombre d'opérations ne dépend pas de la taille des données. Par exemple, afficher le premier élément d'une liste.
- **O(n)** : cette complexité se retrouve souvent quand on a besoin de parcourir les éléments d'une liste, par exemple pour calculer la somme des éléments.

- **O(n²)** : par exemple, l'algorithme naïf pour trier une liste de nombre nécessite, pour chaque élément de la liste, de parcourir l'ensemble de la liste.
- **O(n×m)** : par exemple, pour trouver les nombres communs de deux listes de taille **n** et **m**, il faut pour chaque nombre de la première liste parcourir l'ensemble de la deuxième liste pour vérifier s'il existe.

Complément

La complexité algorithmique permet de dire, qu'à tailles de données égales, un algorithme en **O(n)** terminera bien avant un second algorithme dont la complexité est **O(n²)**.

Ainsi, pour 100 données, le premier algorithme effectuera environ 100 opérations de bases et le second 10 000 opérations. On comprend que, si la taille des données se compte en millions (ou même en milliards), le second algorithme sera beaucoup plus coûteux que le premier.

n	O(n)	O(n²)
10	10	100
100	100	10 000
1 000	1 000	1 000 000
10 000	10 000	100 000 000

Comparaison des complexités

Complexité algorithmique en espace

Complément

Le temps de calcul est important mais l'espace nécessaire au calcul l'est aussi. La mémoire d'un ordinateur n'est pas infinie : il est utile d'estimer l'espace mémoire nécessaire pour exécuter un algorithme.

La complexité décrite jusqu'ici se nomme complexité en **temps**. Ici, on parle complexité algorithmique en **espace**. La notation sera toujours la même : **O(1)**, **O(n)**, **O(n²)**, etc.

Complexité et machine de Turing

Complément

Le concept de complexité algorithmique est étroitement lié aux machines de Turing, une des bases théorique des ordinateurs. Elles ont introduit une nouvelle définition de la notion de calcul qui permet également de théoriser sa complexité. Ainsi, aujourd'hui, on peut classer les problèmes de calcul selon la complexité des algorithmes pouvant les résoudre.

On peut distinguer, très schématiquement, trois grands types de problèmes :

- Problème dont l'algorithme finira en un temps polynomial : **O(n)**, **O(n²)**, etc.
- Problème dont l'algorithme finira en un temps exponentiel : **O(2^n)**, **O(n!)**, etc. On évite que ce type d'algorithme car il suffit que **n** augmente un tout petit peu pour que le temps d'exécution se compte en mois, en années voire en siècles.
- Problème dont il n'existe pas de solution (dits indécidables).

À retenir

- Il est possible d'attribuer une complexité à un algorithme.
- Estimer la complexité d'un programme permet d'avoir un ordre de grandeur de son temps d'exécution.
- Comparer la complexité de deux algorithmes permet de déterminer lequel est le plus optimisé pour répondre à un problème donné.

2. Langage machine et assembleur

Objectifs

- Découvrir le langage machine ;
- Découvrir les bases de l'assembleur.

Mise en situation

Les développeurs écrivent des programmes dans des langages compréhensibles par les humains. Mais l'ordinateur travaille à un niveau différent, il manipule des séquences binaires.

Le langage le plus proche de la machine se nomme assembleur. C'est un langage qui permet des instructions très basiques. On l'utilise assez rarement car il est beaucoup plus efficace pour un être humain d'écrire dans un langage de haut niveau comme le C ou le JavaScript. D'autant qu'il existe des compilateurs ou des interpréteurs qui transforment ce code de haut niveau que l'on écrit en langage machine.

Mais il est néanmoins utile de comprendre les mécaniques de l'assembleur. Cela aide à comprendre certains concepts comme l'adressage mémoire. Et c'est utile pour programmer plus efficacement.

Codage binaire

Fondamental

Un processeur ne comprend pas les langages de programmation. Il ne peut traiter que des instructions sous forme binaire, c'est-à-dire constituées de 0 et de 1. Un processeur ne traitera pas le nombre 42 mais la suite `00101010`, qui est l'équivalent de 42 en codage binaire.

Le langage assembleur

Az Définition

L'assembleur est un langage compréhensible par le processeur. Il ne comporte que quelques instructions très simples (déplacer une donnée, multiplier, additionner, etc.), là où les langages de programmation classiques comportent des instructions plus complexes, correspondant à une combinaisons d'instructions en assembleur.

Les registres

Les registres sont des petites zones mémoire dans le processeur où sont stockées les variables d'entrée ou de sortie d'un calcul. Chaque processeur n'a que quelques registres nommés AX, BX, CX, DX, etc.

Leur taille étant extrêmement limitée, leur tâche est de stocker une donnée **uniquement** le temps du calcul (quelques fractions de secondes) et d'envoyer le résultat en mémoire vive par la suite, quitte à récupérer le résultat dans la mémoire vive pour un calcul ultérieur.

Quelques instructions de base

Exemple

Chaque instruction assembleur a une équivalence binaire direct. Le processeur peut ainsi comprendre ces instructions.

Voici quelques exemples d'instructions de base pour le processeur Intel 8086 :

- `MOV destination source` : déplace une valeur fixe ou celle d'un registre dans un autre,
- `INC registre` : additionne 1 à la valeur d'un registre,
- `NEG registre` : change le signe de la valeur stockée dans un registre,
- `IMUL destination source` : multiplie les valeurs et stocke le résultat dans le premier registre.

Traduire un langage de programmation en langage machine

Attention

L'écriture d'un programme informatique ne se fait la plupart du temps **pas** en assembleur.

On utilise des langages dits de plus haut niveau qui sont eux-mêmes traduits automatiquement en assembleur par un autre programme :

- le compilateur pour les langages compilés (comme C ou C++),
- ou l'interpréteur pour les langages interprétés (comme JavaScript ou Python).

Toutes les machines ne parlent pas exactement le même langage

Complément

Tous les processeurs comprennent un langage binaire. Cependant, chaque type de processeur a son propre jeu d'instructions.

Par exemple, la plupart des ordinateurs ont des processeurs utilisant un jeu d'instructions **étendu** (dit CISC) alors que les smartphones ont des processeurs utilisant un jeu d'instructions **réduit** (RISC). Ainsi, un code dans un langage de programmation donné sera traduit différemment en fonction du processeur cible.

À retenir

- Un processeur ne traite que des instructions binaires.
- L'assembleur est un langage composé d'instructions basiques et facile à traduire en binaire.
- Une instruction dans un langage de programmation donné correspond à de multiples instructions en assembleur.