Chapitre I

Introduction

Sommaire

I Informatique, information . . . 2
II Connaissances . . . 2
III Codage . . . 2
IV Algorithmes . . . 3
A D'abord, le mot ! . . . 3
B Oui, mais en pratique ? . . . 4
C Quelques exemples d'algorithmes . . . 4
D Et l'ordinateur dans tout cela ? . . . 5
V Les qualités essentielles d'un bon algorithme . . . 6
A Entrées . . . 6
B Sorties . . . 6
C Finitude . . . 7
D Définition . . . 7
E Efficacité . . . 7
VI En résumé . . . 8
Annexe : quelques grands noms de l'informatique . . . 9
TD 1 – Une introduction en douceur à l'algorithmique avec Guido . . . 10

I. Informatique, information

Vous avez choisi un BTS informatique, il est donc bon que vous ayez une idée assez précise de ce qu'est l'informatique ! Voici la définition qu'en donne le Larousse :

> **informatique** : Science du traitement automatique et rationnel de l'information en tant que support des connaissances et des communications.

et la définition de l'information (dans son acceptation informatique) :

> **information** : Élément de connaissance susceptible d'être codé pour être conservé, traité ou communiqué.

Il ressort de ces deux définitions trois concepts importants :

- la notion de connaissance,
- la notion de codage,
- et la notion de traitement.

Nous allons détailler dans la suite de cette introduction ces trois notions.

II. Connaissances

Pour commencer, qu'est-ce que la connaissance ? De quelle manière appréhendons nous le monde qui nous entoure ? En y réfléchissant, On peut distinguer trois types de connaissances :

- La *connaissance déclarative* : elle concerne le "quoi" ; elle donne des définitions et des propriétés caractéristiques de la notion étudiée. Par exemple, on peut définir la racine carrée d'un réel positif ou nul x comme étant l'unique réel y positif ou nul dont le carré est x. Cette définition permet de *tester* si un réel est bien la racine carrée d'un autre, mais elle ne donne pas de méthode pour *déterminer* la racine carrée d'un réel.
- La *connaissance impérative* : elle concerne le "comment" ; elle donne des procédés permettant de construire les objets étudiés. Par exemple, vous avez peut-être rencontré dans vos études au lycée la *méthode de Héron*, permettant d'obtenir des valeurs approchées de plus en plus précises de la racine carrée d'un réel x :

  H1. partir d'une estimation $e > 0$ de la racine cherchée ;

  H2. si $e^2 \approx x$, s'arrêter en retournant e ;

  H3. sinon, remplacer e par $\frac{1}{2}\left(e + \frac{x}{e}\right)$, et retourner à [H2].

  Les mathématiques montrent qu'un tel procédé *converge*, ce qui répond bien à nos préoccupations.
- La *connaissance conditionnelle* : elle s'occupe du "quand" ; elle donne les conditions dans lesquelles une connaissance doit être utilisée. Par exemple, on peut donner des contextes dans lesquels il peut être intéressant d'utiliser la racine carrée d'un réel, quand on connaît des propriétés de son carré.

III. Codage

Pour manipuler une connaissance, nous devons la *coder*, de manière, comme indiqué par la définition du Larousse, à pouvoir la stocker, la traiter ou la transmettre. Pour cela, nous utilisons des *symboles* : lettres, chiffres, signaux (lumineux, sonores, électromagnétiques...), pictogrammes, etc.

Voici quelques exemples :

information	codage	nature
“les voitures peuvent passer”	feu vert	signal lumineux
“un appel téléphonique arrive”	sonnerie	signal sonore
“Danger : radioactivité”	☢	pictogramme
“l’année 2011”	MMXI	suite de chiffres romains
“le caractère ’+’”	2B	codage ASCII en héxadécimal
“S O S”	... — ...	codage en morse

Le plus souvent, ce n’est pas un symbole qui est utilisé pour coder une information, mais un ensemble de symboles, appartenant à un certain *alphabet*, assemblés selon un certain nombre de “règles”. L’association d’un alphabet et de ces règles s’appelle un *langage*.

La *syntaxe* est l’ensemble des règles d’assemblage des symboles. Elle indique quelles sont les “phrases” effectivement constructibles dans le langage donné, en général en fournissant une *grammaire*. Par exemple, “$1\ 2\ =\ +\ 4$” est un phrase non valide du langage des expressions arithmétiques. Il est en général relativement simple de tester si une phrase est syntaxiquement correcte, et il existe de nombreux outils, en particulier dans le monde de l’informatique, permettant d’accomplir cette tâche de façon automatique.

La *sémantique* d’un langage exprime la signification associée aux groupes de symboles, c’est elle qui établit la correspondance entre le codage et les éléments de connaissance. Malheureusement, cette correspondance est souvent floue. Une phrase peut très bien être syntaxiquement correcte mais ne pas exprimer une connaissance valide, comme la phrase “$1\ +\ 2\ =\ 4$”. D’autre part, il est souvent très compliqué de vérifier si une phrase signifie effectivement quelque chose. Ce sera le travail du locuteur que de vérifier que ses phrases ont un sens.

Il faut bien comprendre qu’une même connaissance peut être codée de différentes façons, selon le langage qu’on emploie. Par exemple, l’entier 71 peut être codé :

- 71 (codage décimal),
- “soixante et onze” (codage en langue française),
- “seventy one” (codage en anglais)
- 1000111 (codage en binaire)

Réciproquement, le groupe de symboles 71 peut représenter :

- une quantité (les dossiers de 71 candidats ont été retenus par une première sélection),
- une étiquette attribuée à une classe d’objet (la ligne de bus 71),
- le code téléphonique du département de Haute-Loire,
- la notation décimale du code ASCII du caractère G,
- la notation octale du code ASCII du caractère 9,
- la notation hexadécimale du code ASCII du caractère q
- ...

Le “contexte” permet en général (mais pas toujours, ce qui conduit parfois à des catastrophes[1]) à un humain de coder ou décoder sans ambiguïté un message, mais c’est un domaine dans lequel les ordinateurs ont encore de gros progrès à faire !

IV. ALGORITHMES

A. D’abord, le mot !

Je ne crois pas avoir lu un seul cours d’algorithmique qui ne commence par l’origine du mot. Ne coupons pas à la tradition !

[1] Communiqué de CNN le 30 septembre 1999 : “NASA lost a 125 million Mars orbiter because one engineering team used metric units while another used English units for a key spacecraft operation, according to a review finding released Thursday.”

Le mot “algorithme” est une latinisation de la ville d'origine de Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî[2], mathématicien (entre autres) musulman perse, dont l'ouvrage le plus célèbre, *Kitab al jabr w'al muqabala*, a permis l'introduction de l'algèbre en Europe.

Voici la définition qu'en donne le Petit Robert :

> **algorithme** : Suite finie séquentielle de règles que l'on applique à un nombre fini de données, permettant de résoudre une classe de problèmes semblables.

Nous verrons dans la section consacrée aux qualités essentielles d'un bon algorithme l'importance de chacun des mots de cette définition.

B. Oui, mais en pratique ?

Comme on l'a vu dans la deuxième partie, un théorème mathématique affirmant que tout nombre positif a une racine carrée est fascinant du point de vue théorique, mais ne nous intéresse que très peu si l'on a effectivement besoin de la valeur de cette racine carrée. Dans ce cas, on a besoin d'une méthode permettant de calculer cette racine, ou bien, si cela est impossible, d'en obtenir des valeurs approchées aussi précises que possible.

L'algorithmique est la science qui étudie les problèmes du point de vue impératif, concevant des méthodes pour les résoudre, construisant leurs solutions, et étudiant les qualités et les défauts de ces méthodes. C'est elle qui construit les *traitements* des informations connues afin d'obtenir d'autres informations.

Cette notion de traitement s'articule autour de trois concepts :

description : la méthode de passage des données aux résultats est décrite par un texte (en soi, c'est aussi une information, susceptible d'être codée !),

exécution : une réalisation effective du traitement est mise en œuvre sur des données spécifiques,

agent exécutant : c'est l'entité effectuant une exécution ; cette entité est donc capable de mettre en œuvre la méthode.

Dans le contexte de l'informatique, la description sera souvent exprimée à l'aide d'un *algorithme*, l'exécutant sera un *processeur* et une exécution sera appelée un *processus*.

Mais il y a bien d'autres contextes dans lesquels on fait du traitement de l'information. Par exemple, dans une cuisine, une description sera nommée “recette”, l'exécutant sera le cuisinier, et une exécution de la recette permet de passer des ingrédients à l'objet de la recette.

Il ne faut pas confondre la description, l'exécution et l'agent : la recette, en général écrite sur une feuille de papier, n'a pas vraiment de valeur nutritionnelle, et à moins qu'on ait des mœurs spéciales, l'agent ne doit pas être consommé. Seule une exécution particulière de la recette par le cuisinier va fournir un résultat comestible.

On confond souvent, en particulier, description et exécution. Il est pourtant facile de voir que la méthode permettant de multiplier entre eux deux entiers ne doit pas être confondue avec l'exécution du produit 46×17 : la méthode ne donne pas le résultat de cette opération particulière, et l'exécution ne dit pas comment multiplier 45 et 18 !

C. Quelques exemples d'algorithmes

On a déjà rencontré quelques exemples d'algorithme :

- une recette est un algorithme permettant (si tout se passe bien !) de passer des ingrédients isolés au plat alléchant qu'on voit sur la photo ;
- lorsque votre instituteur (ou votre institutrice) vous a appris à multiplier ou diviser deux entiers, les méthodes qu'il ou elle vous a donné sont des algorithmes, que nous implémenterons lorsque nous travaillerons en TP avec les *entiers longs*.

[2] Littéralement : “Père de Ja'far, Mohammed, fils de de Moses, natif de Khowârizm”, ville maintenant nommée Khiva, qui se trouve maintenant en Ouzbékistan

L'un des premiers algorithmes mathématiques connus est le célèbre *algorithme d'Euclide*, permettant de calculer le pgcd de deux entiers (par exemple pour simplifier des fractions). Voici comment on peut l'énoncer de façon moderne[3] :

> **Algorithme d'Euclide** : étant donnés deux entiers positifs non nuls m et n, trouver leur plus grand diviseur commun, c'est à dire le plus grand entier positif les divisant tous les deux.
>
> **E1.** [Calcul du reste] : effectuer la division euclidienne de m par n, soit r le reste de cette division *(ainsi,* $0 \leqslant r < n$*)*.
>
> **E2.** [Est-il nul ?] : si $r = 0$, l'algorithme termine, n est le résultat.
>
> **E3.** [Échange] : faire $m \leftarrow n$, $n \leftarrow r$, et retourner à l'étape E1.

Nous allons, dans la prochaine partie, expliquer les qualités exemplaires de cet algorithme, mais avant cela, un petit exercice qui va vous permettre de vous exercer à l'art subtil de la création d'un algorithme :

EXERCICES :

1) Voici un algorithme, appelé *méthode de multiplication russe*, permettant de calculer le produit de deux entiers a et b :
   - si $a = 0$, le résultat est 0 ;
   - si $a \neq 0$, alors diviser[4] a par 2, multiplier b par 2, *calculer le produit* des deux nombres résultant de ces opération, et, si a est impair, ajouter b au résultat.

   a) Que doit savoir faire l'agent exécutant pour mettre en œuvre cet algorithme ?

   b) Pourquoi est-il bien adapté à un traitement informatique ?

   c) Remarquez que pour calculer le produit de a et b, il faut savoir calculer le produit de deux nouveaux entiers (deuxième ligne) ; cette description est-elle valide ? Que doit-on faire pour la rendre valide si ce n'est pas le cas ?

   d) Utiliser cet algorithme pour calculer le produit de $a = 171$ et $b = 28$ (en base 10), puis le produit de $a = 10101011_2$ et $b = 11100_2$ (en base 2).

2) Vous disposez d'une pile de crêpes[5] de diamètres différents, et vous voudriez les ranger dans l'ordre de taille de manière à ce que la plus grande soit en dessous et la plus petite au dessus.

   Pour cela, vous ne disposez que d'une "manipulation" : vous pouvez insérer votre spatule entre deux crêpes, et retourner en bloc toute la pile de crêpes au dessus de la spatule.

   Le but de cet exercice est de trouver un algorithme efficace pour atteindre l'objectif.

   Pour ceux qui voudrait un challenge un peu plus costaud : on s'est aperçu que chaque crêpe avait un coté plus brûlé que l'autre. Comment s'assurer que la face moins brûlée soit systématiquement sur le dessus ?

D. Et l'ordinateur dans tout cela ?

Un algorithme étant connu, on peut le mettre en œuvre de manière automatique en concevant un mécanisme adapté. C'est ce que fit Pascal lorsqu'il inventa sa "Pascaline" pour effectuer les quatre opérations de base (à l'aide de roues dentées). On peut ainsi concevoir une machine pour chaque tâche qu'on souhaite automatiser, réalisant ainsi le rêve des mathématiciens depuis des siècles. Ces machines existent, et s'appellent des machines à programme fixe. Votre calculette en est un bon (et sophistiqué) exemple.

Mais allons encore plus loin. Imaginons une machine qui prendrait en entrée un algorithme, et se modifierait de manière à appliquer cet algorithme. Une telle machine pourrait donc réaliser n'importe laquelle des tâches pour lesquelles on possède un algorithme. Une telle machine existe :

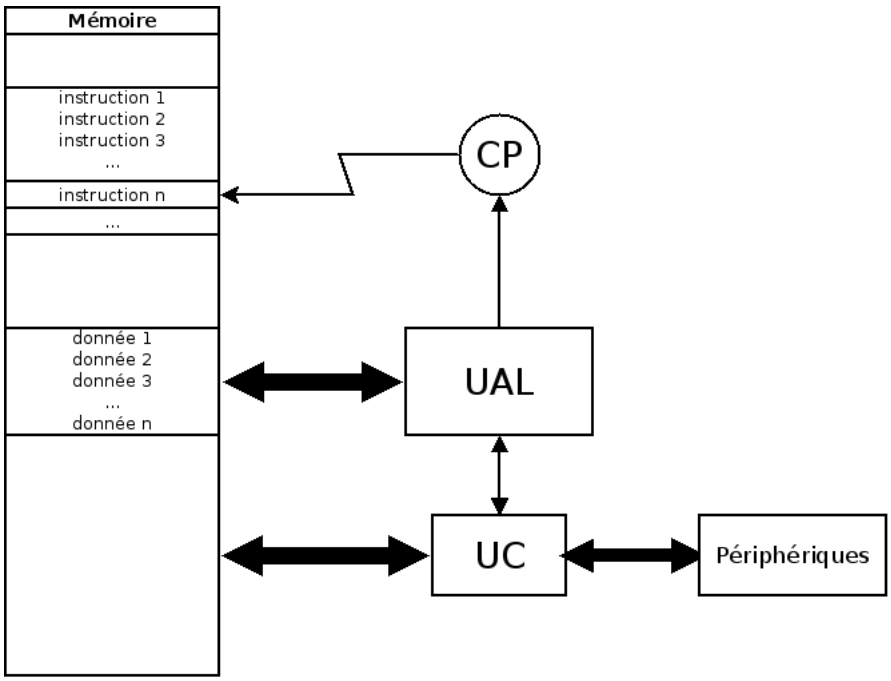
[3] tel qu'il est décrit dans le volume 1 de l'extraordinaire ouvrage du non moins extraordinaire Donald E. Knuth : "The Art Of Computer Programming".

[4] c'est une division entière !

[5] Cet exercice est tiré de l'excellent article "genèse d'un algorithme" du site Interstices (`http://interstices.info/`), site à consulter absolument !

c'est l'ordinateur. Ainsi, tout ce que l'on a à faire pour faire faire un nouveau calcul à un ordinateur est de concevoir l'algorithme le réalisant, l'implémenter à l'aide d'un langage de programmation, et à fournir les entrées.

Voici le schéma d'un ordinateur moderne :



L'UAL, ou *unité arithmétique et logique* (ALU en anglais) est le centre de calcul. Le compteur programme CP point sur une adresse de la mémoire, contenant une instruction à faire exécuter par l'unité de contrôle UC. Celle-ci peut charger des données de la mémoire dans l'UAL, faire exécuter à celle-ci des opérations, transférer les résultats en mémoire, ou bien les transférer vers les périphériques (écran, carte réseau, clavier, souris, imprimante...).

On voit, et c'est un élément fondamental apporté par John Von Neumann, que la mémoire contient à la fois les données manipulées *et* le code du programme à exécuter. Ainsi, il est possible d'envisager des instructions qui modifie le code lui-même !

V. Les qualités essentielles d'un bon algorithme

Voici de quelle façon D.E. Knuth décrit la notion d'algorithme : en plus d'être une suite finie de règles donnant une séquence d'opérations permettant de résoudre un type spécifique de problème, un algorithme a les spécificités suivantes :

A. Entrées

Un algorithme peut avoir des entrées, qui sont des données sur lesquelles il va travailler. Par exemple, les deux entrées de l'algorithme d'Euclide sont les deux entiers m et n. Dans l'exercice proposé, l'entrée est la pile de crêpes.

Ces entrées sont des objets d'un certain ensemble aux propriétés spécifiées, et il ne faut pas s'étonner si on donne à un algorithme des données qui ne respectent pas ces spécificités.

B. Sorties

Un algorithme renvoie une ou plusieurs sorties, qui sont en relation avec les entrées. Par exemple, la sortie de l'algorithme d'Euclide est l'entier n de l'étape E2, qui est le pgcd des entrées m et n. La sortie de l'"algorithme des crêpes" est la version triée de la pile de crêpes initiale. La ou les règles permettant de relier les sorties aux entrées sont les *spécifications* de l'algorithme.

C. Finitude

On a rencontré ce mot dans les définitions données, parfois plusieurs fois. C'est un aspect essentiel de la notion d'algorithme : un algorithme doit toujours se terminer[6], après avoir exécuté un nombre *fini* d'opérations (qui peut quand même être parfois monstrueusement grand !).

Dans l'exemple de l'algorithme d'Euclide, il n'est pas évident de prouver que le procédé s'arrête. Il faut constater qu'à chaque étape, on remplace n par le reste de la division euclidienne de m par n, qui est par définition strictement inférieur à n. Ainsi, la suite des valeurs stockées dans la variable n est une suite d'entiers strictement décroissante, qui doit nécessairement prendre la valeur 0 après au plus n passages dans l'étape E1 de l'algorithme.

À titre d'exercice, vous pouvez essayer de prouver que l'algorithme de retournement des crêpes que vous avez inventé a bien un caractère fini. Ce n'est pas très compliqué.

Il faut bien faire attention au fait qu'il ne suffit pas qu'une méthode s'exprime à l'aide un nombre fini de mots pour mériter le titre d'algorithme : la phrase "avance d'un pas, tourne à droite, et recommence" est un exemple simple d'une description finie (elle comporte seulement 9 mots !) d'un processus infini, qu'on appelle souvent "boucle de la mort" en informatique : le processeur répète indéfiniment la même opération sans rien pouvoir faire d'autre.

Un procédé qui a toutes les qualités d'un algorithme, sauf celle-ci, est appelé *méthode calculatoire*. Un exemple est le calcul d'un réel défini comme limite d'une suite. Ces méthodes ont bien entendu de l'intérêt, mais le travail permettant d'en tirer des algorithmes nécessite entre autres de régler le problème de l'arrêt du calcul.

D. Définition

Chaque étape de l'algorithme doit être précisément et de façon non ambiguë définie. Ce concept nécessite de prendre en compte l'entité qui effectuera les opérations : le *processeur*.

Ainsi, on ne pourra pas énoncer l'algorithme d'Euclide de la façon dont on l'a énoncé à une personne qui ne sait pas ce qu'est une division euclidienne. De la même façon, on ne décrira pas une recette de cuisine de la même façon à un chef ayant 35 ans d'expérience et à un parfait débutant qui n'a jamais monté des blancs en neige ! Qu'est-ce qu'une "pincée" de sel pour quelqu'un qui n'a jamais fait la cuisine ? Ou une "béarnaise"[7] ?

De la même façon, vous ne pourrez pas concevoir des programmes corects sans prendre en compte les "connaissances" du langage que vous utiliserez. Dans le cadre de ce cours, nous prendrons un sous-ensemble minimal suffisant du langage Pascal, dont un résumé vous sera donné en temps voulu. Ce langage connaît la division euclidienne, mais pas le pgcd !

E. Efficacité

Les algorithmes que nous voulons concevoir ne devront pas seulement effectuer un nombre fini d'opérations pour accomplir leur tâche. Nous aimerions que ce nombre d'opérations soit *raisonnablement fini* ! Rien ne sert d'avoir un algorithme permettant de trouver la "réponse à la grande question de l'univers[8]" si cet algorithme demande un temps de calcul supérieur à l'âge de l'univers (de l'ordre de 30 milliards d'années d'après les estimations actuelles).

On ne peut malheureusement en général pas connaître le nombre exact d'opérations effectuées par un calcul. Ou plutôt, ce nombre dépend des entrées, et plus précisément de leur taille. Pour des entiers, ce sera leur grandeur, pour un tableau, le nombre de ses cellules... Nous passerons donc pas mal de temps pendant l'année à examiner le fonctionnement des algorithmes que nous concevrons, pour déterminer le plus précisément possible (ou au moins *majorer* au plus près) le nombre d'opérations nécessaires pour réaliser un calcul. Comme chaque étape de ce calcul nécessite en général plusieurs opérations, nous aurons à évaluer les opérations les plus significatives, ou bien les plus coûteuses pour le matériel.

[6] ce qui n'a en général rien d'évident, demandez à Alan Turing !

[7] Remarquons à ce sujet qu'un livre de cuisine exhaustif contient aussi la recette de la béarnaise, ainsi que la définition d'une pincée de sel, et les équivalence température-thermostat permettant de reproduire exactement les conditions dans lesquelles la recette doit être exécutées. Ces définitions et recettes peuvent être considérées comme des sous-programmes, qu'on utilise souvent, et qu'on ne veut pas réécrire à chaque fois.

[8] Consulter "Le guide du routard galactique" du regretté Douglas Adams pour plus de détails.

Par exemple, on a vu que dans l'algorithme d'Euclide, à chaque étape du calcul, on effectue une division euclidienne, suivie obligatoirement d'une comparaison du reste à 0, et d'un échange de variables (sauf à la dernière division). Comme la division est l'opération la plus compliquée effectuée à chaque étape, on utilisera le nombre de divisions effectuées comme indicateur du temps nécessaire pour le calcul.

Ainsi, si les entrées sont les entiers m et n, on a vu qu'on effectue au plus n divisions euclidiennes dans chaque étape E1. On peut ainsi *majorer* le nombre d'opérations nécessaires au calcul du pgcd de deux entiers. Mais si n est de l'ordre du milliard, cela donne un nombre très important d'opérations !

En fait, ici, l'estimation est franchement pessimiste, puisqu'on peut constater qu'il faut beaucoup moins de n divisions quelques soient les valeurs de m et n choisies. Déterminer le nombre exact de divisions est ici impossible, mais on peut démontrer qu'il est majoré par $\ln n$, $\ln$ étant le logarithme népérien (voir le cours de mathématiques).

Un autre exemple : pour trier un tableau comportant n entrées, les algorithmes peu efficaces ont besoin d'effectuer de l'ordre de n^2 opérations. Le tri d'un tableau de 10 entrées se fait donc de façon quasi instantanée. Par contre, que se passerait-il si l'on souhaitait trier un tableau comportant une soixantaine de millions d'entrées (par exemple, un tableau recensant la population d'un pays comme la France) ? Il faudrait alors de l'ordre de 4 millions de milliards d'opérations. En supposant que l'ordinateur utilisé effectue un milliard de comparaisons par seconde (ce qui est *très* optimiste !), il faudrait environ 4 millions de secondes pour trier ce tableau, soit 45 jours !

Certains algorithmes que nous rencontrerons nécessitent encore plus d'opérations, et ne sont donc en pratique utilisables que pour de petites tailles des données.

D'autre part, il n'y a pas que le temps qui soit déterminant dans le fonctionnement d'un algorithme. De la même façon, la mémoire (qu'elle soit vive ou de masse) dont nous disposons est limitée. Il ne faudrait pas que pour effectuer un calcul, il faille plus de bits de mémoire que le nombre d'atomes de l'univers (de l'ordre de 10^{100} à quelques dizaines de zéros près) !

Dans le cas de l'algorithme d'Euclide, il est assez facile de se convaincre que trois cases mémoire suffisent pour l'ensemble du calcul. En effet, une fois r déterminé dans l'étape E1, on peut se passer de la valeur de m, et donc y ranger l'ancienne valeur de n, puis ranger la valeur de r dans la case n ainsi libérée.

VI. En résumé

En pratique, les questions essentielles auquel il faut répondre lorsqu'on conçoit un algorithme sont donc :

1) l'algorithme se termine-t-il (et ce pour n'importe quelle entrée, et non pas seulement les quelques-unes que l'on a testé) ?

2) l'algorithme est-il correct, c'est-à-dire renvoie-t-il le bon résultat dans tous les cas ?

3) l'algorithme est-il performant (ce qui revient à trouver un lien entre le temps et/ou l'espace mémoire nécessaire et la "taille" des entrées) ?

Ces questions sont absolument fondamentales de nos jours. En effet, la plupart des systèmes numériques sont maintenant "embarqués" dans des outils dont notre tranquillité, notre sécurité, voire notre vie, dépend. Peut-on par exemple imaginer un système de gestion du freinage d'une voiture qui ne fonctionne pas lorsqu'un capteur est encrassé, ou bien qui réagit dans certaines conditions trois ou quatre secondes trop tard ?

On peut souvent associer les deux premiers points dans la recherche d'une *preuve*, et souvent, l'analyse de cette preuve permet d'obtenir une estimation du nombre de fois où chaque instruction de l'algorithme est effectuée, permettant de répondre au troisième point.

ANNEXE : QUELQUES GRANDS NOMS DE L'INFORMATIQUE

- **Blaise Pascal** (France, 1623-1662) a construit en 1642 l'un des premiers calculateurs mécaniques, la célèbre *Pascaline*, pour aider son père à calculer les taxes de la région de Haute-Normandie.
- C'est **Charles Babbage** (Grande Bretagne, 1791-1871) qui a le premier conçu les plans d'une machine à calculer mécanique "programmable" (à l'aide de carte perforées utilisées à l'époque pour les machines à tisser), la *machine analytique*. On y trouve tous les "ingrédients" des ordinateurs modernes. Malheureusement, cette machine n'a pas été concrétisée à l'époque (mais des scientifiques modernes se sont amusés à en construire une pour prouver que le bestiau était réalisable).
- **Ada Lovelace** (Grande Bretagne, 1815-1852) a travaillé sur le modèle théorique de Babbage et peut être considérée comme la première "programmeuse". On a d'ailleurs donné son prénom à un langage de programmation.
- Les travaux en logique de **George Boole** (Grande Bretagne, 1815-1864) et **Gottlob Frege** (Allemagne, 1848-1925) ont largement contribué à la science des ordinateurs.
- Les travaux théoriques de **Kurt Gödel** (Allemagne, 1906-1978), et des américains **Alonzo Church** (1903-1995), **Stephen Cole Kleene** (1909-1994), **Emil L. Post** (1897-1954) et **Claude Shannon** (1916-2001), ont largement contribué à faire avancer la science des ordinateurs.
- **Alan Turing** (Grande Bretagne, 1912-1954) a travaillé sur le concept de calcul, et l'a concrétisé en une machine théorique qui sert encore aujourd'hui de référence. Cette *machine de Turing*, extrêmement rudimentaire, permet de définir précisément ce qu'est un calcul réalisable par une machine.
- Les américains **J. Presper Eckert** (1919-1995) et **John Mauchly** (1907-1980) ont construit le premier ordinateur digital électronique, l'ENIAC, en 1945.
- **John von Neumann** (Hongrie, 1903-1957) a entre autres décrit l'architecture des ordinateurs, qui est toujours utilisée de nos jours.
- **Grace Hopper** (États-Unis, 1906-1992) conceptualisa la notion de langage de programmation indépendant de la machine. On peut en quelque sorte la considérer comme la mère de tous les langages informatiques modernes !
- Quelques langages célèbres, avec leur(s) inventeur(s) : COBOL[9] (collectif), FORTRAN (**John Backus**), LISP (**John McCarthy**), C (**Dennis Ritchie** et **Ken Thompson**, aussi auteurs du système d'exploitation UNIX), PASCAL (**Niklaus Wirth**), JAVA (**James Gosling** et **Patrick Naughton**, PYTHON (**Guido van Rossum**)... que les nombreux autres qui ne sont pas cités ici me pardonnent !
- Une mention spéciale à **Donald E. Knuth** (États-Unis, 1938-) et **Edsger W. Dijkstra** (Pays-Bas, 1930-2002), qui ont joué un grand rôle dans la science qui nous intéresse tout particulièrement, l'algorithmique. En particulier, "The Art Of Computer Programming", l'une des œuvres majeures de D. Knuth, est une somme sans équivalent sur les algorithmes et les structures de données. À noter que D. Knuth est aussi l'auteur de TEX, dont est dérivé LATEX, langage de composition de documents utilisé par l'ensemble de la communauté scientifique. Il est remarquable de constater qu'après 34 ans de bons et loyaux services, on a toujours pas trouvé mieux pour écrire des textes scientifiques ! Une telle longévité est rarissime dans le monde de l'informatique.

  À titre d'exercice, vous pouvez chercher comment prononcer les noms de ces deux scientifiques, et la signification des initiales E, pour Knuth, et W, pour Dijkstra !

[9] pour faire plaisir à D. Gallot !