Tangente

Bibliothèque Tangente

L'aventure mathématique

Les algorithmes

Au cœur du raisonnement

HS n° 37

ISSN 0987-0806

Bibliothèque Tangente

L'aventure mathématique

Tangente Hors-série n° 37

Les algorithmes

Au cœur du raisonnement

© Éditions POLE - Paris 2009

Toute représentation, traduction, adaptation, publication ou reproduction, même partielle, par tous procédés et sur tous supports, en tous pays, faite sans autorisation préalable, est illicite et exposerait le contrevenant à des poursuites judiciaires. Référence : loi du 11 mars 1957.

ISBN : 9782848841069 ISSN : 0987-0806 Commission paritaire : 1011 K 80883

Bibliothèque
Tangente
L'aventure mathématique

Prochainement
dans la Bibliothèque Tangente

Mathématiques et Philosophie

ÉDITIONS
POLE

Les algorithmes

Sommaire

DOSSIER Les algorithmes dans l'histoire 5

Les algorithmes ne sont pas nés avec l'informatique. Par exemple, l'algorithme d'Euclide est vieux de plus de 2 000 ans ! On trouve également des descriptions précises d'algorithmes dans la Chine ancienne. Avant d'être des programmes informatiques, les algorithmes sont des objets mathématiques.

Mohammed al-Khwarizmi et son temps 6
Aux racines de l'algorithme 10
Les algorithmes du secret : la cryptographie 14
Alan Turing et sa machine 16
Lady Augusta Ada King, comtesse de Lovelace 20

DOSSIER Algorithmes élémentaires et programmation 23

On rencontre de nombreux algorithmes dans l'histoire des mathématiques. De nos jours, les opérations manuelles auxquelles les algorithmes donnaient lieu ont laissé la place aux programmes informatiques.

De l'algorithme au langage de programmation 24
Les bases de la programmation 28
Les tests de primalité 34
Calcul de racines carrées : l'algorithme de Babylone 38
Des algorithmes pour créer le hasard 44
Les fractions égyptiennes 48
Les mariages stables existent 50
Sous l'ordinateur, les booléens 54
N'abusons pas des organigrammes ! 60
Programmer l'algorithme d'Euclide 64

(suite du sommaire au verso)

DOSSIER Algorithmes classiques et jeux 69

Les mathématiques ont donné naissance à une multitude d'algorithmes, dans tous les domaines : théorie des nombres, topologie, mathématiques financières, recherche opérationnelle, théorie des graphes, mathématiques récréatives...

Équations récurrentes en finance 70
La programmation fonctionnelle 76
Gagner au jeu grâce au noyau d'un graphe 80
Le pivot de Gauss 86
L'algorithme du simplexe 90
La tour d'Hanoï 96
Comment explorer un labyrinthe ? 100

DOSSIER Limites et performances 107

Ce n'est pas tout de savoir structurer un raisonnement répétitif à l'aide d'un algorithme. Il faut aussi prouver son efficacité et chercher à minimiser le nombre d'opérations qui vont intervenir dans son exécution, même si c'est l'ordinateur qui doit les effectuer.

Complexité et temps d'exécution 108
Veni, divisi, vici 112
Les algorithmes de tri 116
La programmation structurée 122
La magie de la récursivité 126
Itération et point fixe 130
La gloutonnerie appliquée à la compression 136
Codes correcteurs d'erreurs 142
La multiplication rapide 150

Problèmes 154
Solutions 158

En bref 9, 22, 27, 32, 33, 53, 149

Des algorithmes à la folie
59, 63, 68, 74, 75, 83, 84, 85, 88, 89, 95, 98, 99, 102, 103, 104, 105, 106, 111, 121, 125, 129, 135, 141

Mohammed al-Khwarizmi et son temps p. 6
Aux racines de l'algorithme p. 10
Histoire de la cryptographie p. 14
Alan Turing et sa machine p. 16
Lady Augusta Ada King, comtesse de Lovelace p. 20

LES ALGORITHMES DANS L'HISTOIRE

Les algorithmes ne sont pas nés avec l'informatique. Par exemple, l'algorithme d'Euclide est vieux de plus de 2 000 ans ! On trouve également des descriptions précises d'algorithmes dans la Chine ancienne. Avant d'être des programmes informatiques, les algorithmes sont des objets mathématiques.

Mohammed al-Khwarizmi et son temps

Le mot français « algorithme » provient du nom d'un savant arabe du IXe siècle. On lui doit plusieurs méthodes pour le calcul effectif de racines d'une équation du second degré. C'est également grâce à lui que se diffuseront les chiffres arabes en Occident.

Couverture d'une copie du premier livre d'al-Khwarizmi.

Cinquante ans après son installation à Bagdad (capitale de l'actuel Irak) en 762, le califat abbasside était au summum de sa splendeur. Haroun al-Rachîd, lettré éclairé, fut très occupé à mater les oppositions intérieures et extérieures qui menaçaient la stabilité du califat.

La Maison de la sagesse

Après un début de règne agité, son fils et successeur al-Mamûn, après avoir fait disparaître son frère, se consacra au développement des arts et de la culture, en particulier scientifique. Il ouvrit la bibliothèque personnelle de son père pour créer la Bayt al-Hikma, c'est-à-dire la Maison de la sagesse. Les savants s'y retrouvaient pour consulter des ouvrages. C'est là que furent traduits en arabe de nombreux textes scientifiques grecs, permettant ainsi la conservation d'un grand nombre d'entre eux.

C'est à cette époque que Muhammad ibn Musa, né en 788 probablement à Khiva, vint se fixer à Bagdad. Dans la capitale abbasside, il est connu sous le nom de Muhammad al-Khwarizmi, d'après le nom de sa contrée natale, le Khwarezm, correspondant à l'ouest de l'Ouzbékistan actuel. On sait peu de choses de sa vie sinon qu'il fréquenta assidûment la Bayt al-Hikma où la

convergence des cultures grecques et indiennes lui permit de construire une œuvre fondamentale en mathématiques, présentée en deux ouvrages écrits vers 830.

La résolution des équations

Le premier ouvrage *al-Kitâb al-mukhtasar fâ hisâb al-jabr w'al-muqâbala*, le *Livre de l'explication du calcul de la remise en place et de la simplification*, a donné son nom à l'algèbre. Contrairement aux mathématicien grecs, al-Khwarizmi détaille des méthodes effectives de résolution d'équations. Il les traite avec des exemples tirés d'expériences pratiques. Cependant, sachant son lecteur peu familiarisé aux nombres négatifs, il transfert toute quantité négative de l'autre côté du signe égal pour le rendre positif : c'est *al-jabr*, *la remise en place*. Par exemple, l'équation en notations actuelles $x^2 - 23 = 16 - 10x$ devient $x^2 + 10x = 16 + 23$.

Al-muqâbala est la simplification ; ainsi, l'équation précédente devient alors $x^2 + 10x = 39$ (voir l'encadré).

Al-Khwarizmi nous présente une exposition complète de la résolution des équations du premier et du second degré. L'inconnue, que nous notons x dans cet article, s'appelle la *racine* et, comme il a éliminé tout nombre négatif, il distingue six cas et les traite sur des exemples qui se généralisent sans difficulté pour toute équation de même type. Il considère ainsi :

- Carrés égaux aux racines, c'est-à-dire de la forme $ax^2 = bx$;
- Carrés égaux à un nombre, soit $ax^2 = c$;
- Racines égales à un nombre, soit $bx = c$;
- Carrés et racines égaux à un nombre, soit $ax^2 + bx = c$;

D'al-Khwarizmi à algorithme

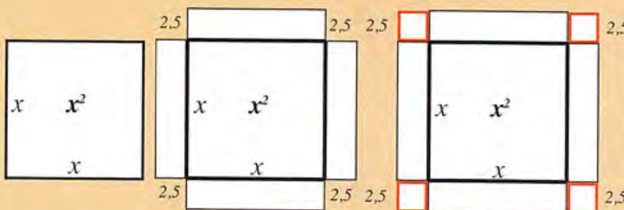
Le deuxième ouvrage n'est connu que par sa traduction latine, effectuée probablement par Adélard de Bath dans la première moitié du XIIe siècle. Le manuscrit était alors connu sous le nom de *dixit Algorizmi – ainsi disait al-Khwarizmi–* ou de *Algoritmi de numero Indorum – al-Khwarizmi, au sujet des nombres indiens*. Comme son nom l'indique, celui-ci présentait le maniement des chiffres arabes, en fait introduits par les Indiens au VIIe siècle de notre ère. *Algoritmi* est une latinisation du nom du mathématicien arabe. À partir du XIIIe siècle, on appelle *algorisme* le calcul utilisant les chiffres arabes et *algoristes* ceux qui s'y adonnent. Influencé par le grec *arithmos* qui signifie *nombre*, ce mot devient *algorithme* et semble alors presque le jumeau du mot *logarithme*, créé sur le grec par John Neper au début du XVIIe siècle. Il ne prend son sens actuel qu'au XIXe siècle, et devient d'utilisation courante grâce au développement de l'informatique depuis la Seconde Guerre mondiale.

Résoudre $x^2 + 10x = 39$ avec al-Khwarizmi

Un carré et 10 racines sont égaux à 39 unités, que vaut la racine ? Rappel : la racine correspond à ce que nous appelons l'inconnue.

Méthode algébrique :
On prend la moitié des racines (c'est-à-dire 5) ; on la met au carré, soit 25, que l'on additionne à 39, soit 64. Prenons alors la racine carrée de ce nombre, soit 8, et ôtons-lui la moitié des racines : la solution est donc $8 - 5 = 3$. C'est magique, direz-vous ! Pas plus que la méthode classique avec le discriminant ; c'est un algorithme efficace pour toutes les équations du type $x^2 + bx = c$, où b et c sont des réels positifs.

Méthode géométrique :
Tracer un carré de côté x unités de mesure. Répartir les $10x$ en quatre rectangles de côtés x et $10/4$ apposés aux quatre côtés du rectangle. La surface totale de la figure est donc $x^2 + 10x$, soit 39. Compléter cette figure en un carré en y ajoutant aux quatre coins des carrés de côté $5/2$. Le nouveau carré obtenu a donc pour surface $39 + 4.(5/2)^2 = 64 = 8^2$. Or le côté du carré initial est x alors que celui du nouveau carré est 8, donc $x = 8 - 2 . 5/2 = 8 - 5 = 3$.



Méthode actuelle :
$x^2 + 10x = (x + 5)^2 - 5^2 = 39$, donc $(x + 5)^2 = 39 + 25 = 64 = 8^2$, donc $x + 5 = 8$ ou $x + 5 = -8$, soit $x = 3$ ou $x = -13$. Reprenez la méthode algébrique : vous voyez bien pourquoi on prend la moitié des racines ! Bien sûr, à son époque, al-Khwarizmi ne pouvait considérer que les racines positives (il ne trouvait pas -13). De nos jours, on effectue l'algorithme suivant : $\Delta = 10^2 - 4.(-39) = 256 = 16^2$, donc l'équation admet deux solutions qui sont :
$x_1 = (-10 + 16)/2 = 3$ et $x_2 = (-10 - 16)/2 = -13$.

- Carrés et nombre égaux aux racines, soit $ax^2 + c = bx$;
- Racines et nombres égaux aux carrés, soit $bx + c = ax^2$; où a, b et c désignent des nombres positifs.

L'apport des chiffres arabes

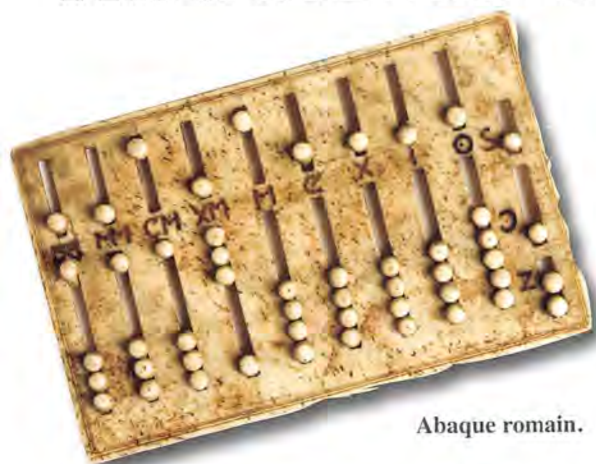
Le deuxième ouvrage ne nous est connu que par sa traduction latine *Algoritmi de numero Indorum*. Celle-ci date seulement du XIIe siècle, soit trois siècles après qu'il ne soit écrit, et semble provenir d'un texte modifié et incomplet. Le premier mot du titre est une déformation du nom du mathématicien arabe (voir encadré page précédente). Al-Khwarizmi y présente le maniement des chiffres arabes, en latin *numero Indorum*, c'est-à-dire *nombres indiens*. Les méthodes pour les additionner et les multiplier sont expliquées avec précision. Le mathématicien arabe reprend dans cet ouvrage les grandes lignes d'un traité donné au calife vers 770 par un de ses hôtes indiens. Son livre joua un rôle fondamental pour la diffusion en Europe de ce nouveau mode d'écriture des nombres, mais il fallut encore attendre trois siècles pour que leur emploi se généralise en Occident.

Al-Khwarizmi a eu le grand mérite de développer des techniques mathématiques nouvelles. Bien que son œuvre ne soit pas tellement novatrice, elle sait, avec talent, être l'héritière à la fois des mathématiques grecques et indiennes. L'importance d'al-Khwarizmi est avant tout due à l'impact de ses ouvrages en Occident pour l'introduction de la numération de position. Ceci explique que son nom, et qu'un mot du titre d'un de ses livres, soient devenus, un peu déformés, des mots du langage mathématique, les rendant ainsi immortels.

B. H.

par Bertrand Hauchecorne

Abacistes et algoristes



Abaque romain.

La diffusion des chiffres arabes

Cités pour la première fois en Occident par Gerbert d'Aurillac, mathématicien et pape de l'an mil, les chiffres arabes mettent de nombreux siècles à s'imposer. La traduction des ouvrages d'al-Khwarizmi au XIIe siècle, puis les écrits de Fibonacci cent ans plus tard, permettent leur diffusion. Il faut cependant attendre la Renaissance pour qu'ils s'imposent. Ils prouvent alors leur efficience lors du développement du commerce en Europe à la fin du Moyen Âge, grâce à la simplification des calculs. De grands mathématiciens comme Nicolas Chuquet (vers 1445–vers 1500) en France ou Adam Riese (1492–1559) en Allemagne prennent leur plume et écrivent en langue vernaculaire des manuels pour expliquer leur maniement à ceux qui font du commerce. La résistance au changement est grande et nombreux sont ceux qui continuent à calculer sur un boulier. On les connaît sous le nom d'*abaquistes* – *abaque* désignant une tablette à calculs – qui s'opposent alors aux *algoristes*. Les premiers disparaîtront petit à petit et désormais :

Nous sommes tous des algoristes.

Vous avez dit al-jabr ?

Le premier ouvrage, *Kitâb al-mukhtasar fî hisâb al-jabr w'al muqâbala*, traite des méthodes de résolution des équations de degré 1 et 2. Il fut traduit en latin par Robert de Chester en 1145 sous le titre de *Liber algebrae et almucabala* et aussi par Gérard de Crémone. *Al-jabr* signifie « la remise en place » et consiste à modifier l'équation, de telle sorte qu'elle ne contienne aucun élément négatif. Ce mot fut utilisé à la Renaissance pour désigner une extension des méthodes de calcul à des nombres négatifs, et pour la recherche de solutions d'équations. Le terme évolua par la suite pour aboutir à son sens actuel.

Algèbre et algorithme

Les deux premières lettres de ces deux mots trahissent leur origine commune, la langue arabe. Dans cet idiome, en effet, l'unique article, *al*, s'accole au mot qui le suit. On en trouve d'autres exemples dans notre langue comme *alcool*, *alcôve* ou *algarade*. La particularité des deux termes qui nous intéressent est leur provenance commune : le mathématicien arabe Mohammed al-Khwarizmi (voir l'article en page 6) et ses deux ouvrages, écrits tous deux vers 830.

par François Lavallou

Aux racines de l'algorithme

Emprunte de pragmatisme, la pensée chinoise a développé au long des siècles des procédures pour la pratique concrète des calculs. Ces procédures comportent déjà des caractéristiques qui seront fondamentales dans nos algorithmes modernes.

Nous prendrons pour illustration de nos propos le calcul de la racine carrée, objet classique de l'histoire des sciences, traité ponctuellement par les Babyloniens (voir l'article en page 38). Les Grecs démontrèrent l'irrationalité de $\sqrt{2}$ et les Indiens établirent des estimations générales des racines. Nous ferons le choix de nous intéresser aux méthodes chinoises développées à partir du IIe siècle car elles peuvent être considérées comme les premiers exemples de structures algorithmiques modernes.

Petite largeur

Alors que la philosophie occidentale se passionne pour le Vrai, la pensée chinoise est séduite par le Beau. Très pragmatique et politique, elle se préoccupe plus volontiers de résoudre les problèmes concrets de la société que de définir des concepts abstraits. Les mathématiques chinoises de l'Antiquité devaient ainsi permettre aux lettrés–fonctionnaires de pouvoir traiter les affaires de l'état dont ils avaient la charge. Les rares écrits qui nous soient parvenus confirment cette orientation vers le calcul concret en nous livrant un grand nombre de méthodes pratiques de résolution de problèmes de natures administratives : partage, imposition, cadastre, *etc*. Néanmoins, l'abstraction n'est pas totalement absente de la science chinoise, mais elle se présente sous une autre forme que celle développée en Grèce. Certaines de ces méthodes (ou procédures) de calcul qui sont introduites à l'aide de *paradigmes* firent l'objet de tentatives de généralisation au cours des siècles. Nous allons illustrer cette tendance en étudiant un outil universel de l'histoire des sciences, aux technicités multiples et élaborées : l'extraction d'une racine carrée.

Les *Neuf chapitres sur les procédures mathématiques*, ou *Neuf chapitres*, un classique de la dynastie Han qui sera

Paradigme

Dans le contexte de l'article, exercice de référence dont la structure est représentative de toute une classe de problèmes et dont le traitement définit la procédure ou méthode à utiliser pour leur résolution : c'est un modèle.

notre principale référence, est organisé en chapitres présentant une unité thématique. Nous nous intéresserons particulièrement au chapitre IV intitulé *Petite largeur*. Ce chapitre est sans doute le plus abstrait de tous. Il traite principalement de la division et de l'extraction de racines carrées ou cubiques à l'aide de problèmes consistant à déterminer les côtés d'une figure dont on connaît la surface, ou le volume, et une dimension caractéristique. Le lien entre ces divers problèmes est l'unité structurelle des procédures utilisées.

Surface à calculer

Avant d'étudier la pratique du calcul, nous avons à présenter un élément central des mathématiques chinoises : la surface à calculer. Son rôle, non totalement explicité dans les *Neuf chapitres*, va être précisé au cours des siècles : il s'agit simplement d'une surface plane quelconque qui reçoit les baguettes à calculer avec lesquelles on représente les nombres (voir encadré). Mais les textes mathématiques imposent une mise en page de cette surface en stipulant, par écrit et dessin, la position du flot de calculs à venir le long des lignes horizontales et verticales. Nous explicitons en encadré cette structure calculatoire en détaillant, étape par étape comme dans un livre du XIIIe siècle, la procédure fondamentale associée à la division. Notons bien que les termes employés – quotient, dividende, diviseur – dénomment les emplacements des valeurs utilisées au cours du calcul et non les valeurs elles-mêmes. Nous sommes en présence d'une assignation de variables dont la nature va apparaître clairement dans la procédure d'extraction d'une racine carrée.

Procédure de division

Nous allons illustrer la méthode associée à la division en figurant la surface à calculer à chaque étape du calcul. Soit à diviser 1 312 par 23. L'algorithme demande de placer le nombre à diviser, le dividende (dvd), dans la zone médiane de la surface à calculer et, dans un premier temps (*a*), le diviseur (dvs) en dessous de lui. La première étape consiste à décaler au maximum le diviseur vers la gauche tout en restant sous le dividende. Puisque 23 est supérieur à 13, on le décale d'un cran vers la droite pour obtenir la position (*b*). La division de 131 par 23 donne 5, qui doit être placé sur la ligne supérieure (*c*), le quotient (Q). On retranche ensuite du dividende le produit de Q par le diviseur en soustrayant dans un premier temps 5 fois 20 (*d*), puis 5 fois 3 (*e*). Après cette première phase, on décale le diviseur vers la droite (*f*) pour pouvoir réitérer le processus : la division de 162 par 23 nous donne 7, placé au quotient (*f*), qui amène à ôter 7 fois 20 (*g*), puis 7 fois 3 (*h*) au diviseur. Le résultat final est donc 1 312 / 23 = 57 + 1 / 23.

(a)						(e)		
				Q				5
1	3	1	2	*dvd*		1	6	2
		2	3	*dvs*		2	3	

(b)						(f)		
				Q			5	7
1	3	1	2	*dvd*		1	6	2
	2	3	←	*dvs*	→		2	3

(c)						(g)		
		5		*Q*			5	7
1	3	1	2	*dvd*			2	2
	2	3		*dvs*			2	3

(d)						(h)		
		5		*Q*			5	7
	3	1	2	*dvd*				1
	2	3		*dvs*			2	3

Extraction de racines

Calculons la racine carrée de A = 55 225. Le texte demande de placer A comme dividende, ce qui correspond à la ligne centrale de la surface à calculer pour l'algorithme de division. Une « baguette empruntée » est utilisée sur une ligne

auxiliaire (deux pour la racine cubique) pour marquer les ordres de grandeurs. On la déplace le plus loin possible de la position des unités par saut de 10^2 vers la gauche, tout en restant sous le dividende.

	Q
5 5 2 2 5	*dvd*
	dvs
1 ← ←	*auxiliaire*

De ces $n = 2$ sauts, on déduit que le premier chiffre a du quotient (racine) se place dans la $n + 1$ soit la troisième colonne de la ligne supérieure de la surface à calculer (Q). La façon de positionner les résultats intermédiaires du calcul relève d'une volonté délibérée d'utiliser la structure de la division. Pour retrancher de A le carré $(a10^n)^2$ du premier ordre de grandeur de la racine, correspondant au carré jaune *Jia* de la figure, on forme le diviseur en multipliant le quotient par 10^n et on applique la procédure de base de la division : retirer au dividende le produit du quotient et du diviseur.

2	*Q*
1 5 2 2 5	*dvd*
2	*dvs*

Si le second chiffre est b à l'ordre 10^{n-1}, il faut soustraire à l'aire restante la valeur $(2a10^n + b10^{n-1}).b10^{n-1}$ qui correspond à la surface des deux rectangles vermillon et du carré jaune *Yi*. Ce calcul est exécuté comme une division. La première étape est de produire un ordre de grandeur de la somme des longueurs des deux rectangles vermillon, ce qui nécessite deux opérations : doubler le diviseur et rétrograder d'un cran vers la droite.

Il s'agit maintenant de déterminer $b10^{2(n-1)}$ pour éliminer l'aire du carré jaune *Yi*. On recourt de nouveau au marquage de position à l'aide de la baguette empruntée qui est placée en $10^{2(n-1)}$.

2	*Q*
1 5 2 2 5	*dvd*
→ 4	*dvs*
1	*auxiliaire*

Son produit par b, placé au quotient en 10^{n-1}, est rangé en ligne auxiliaire et ajouté à la ligne du dessus : le diviseur est constitué.

2 3	
1 5 2 2 5	
4 3	
3	

L'application de la procédure de division produit alors la configuration suivante :

2 3	
2 3 2 5	
4 3	
3	

S'il fallait continuer, nous aurions à évaluer la surface des deux rectangles bleu-vert et du petit carré jaune de la figure. La somme des longueurs des rectangles bleu-vert, $2(a10^n + b10^{n-1})$, s'obtient en vidant la ligne auxiliaire dans le diviseur, et le processus de division continue pour obtenir *in fine* $Q = 235$. Cet exemple détaillé permet de comprendre pourquoi cette procédure est souvent dénommée « diviser par extraction de racine carrée ». Il confirme de plus que les termes diviseur, dividende et quotient renvoient à une *position* plus qu'à une *fonction*. Cette *assignation* de variables permet d'établir des *itérations* en établissant les mêmes calculs sur des quantités différentes : après ajustement de la

Carré d'aire A.

Références

Les neuf chapitres. Karine Chemla et Guo Schuchun, Dunod, 1 140 pages, 2004.
Histoire des mathématiques, de l'Antiquité à l'an mil. Bibliothèque Tangente numéro 30, 156 pages, 2007.

valeur du diviseur, on effectue les mêmes opérations que la division. Après dix siècles d'évolution, les algorithmes d'extraction de racines carrées ou cubiques, considérés comme semblables dans ce chapitre IV des *Neuf chapitres*, seront unifiés en une seule procédure qui traite de la division et des racines énièmes !

Algos à gogo

L'utilisation de baguettes semble avoir été un élément fondamental de l'émergence de telles procédures. Cette représentation matérielle des nombres permet en effet la modification rapide de quantités par une suite de calculs élémentaires sans en changer la place. La surface à calculer sur laquelle nous avons placé des quantités à certains emplacements peut être mise en analogie avec une *mémoire* qui contient des valeurs à certaines adresses. Puisque les procédures des *Neuf chapitres* utilisent les notions d'assignation, d'itération et de tests, il est raisonnable, d'après les critères de Knuth (voir encadré) de trouver dans les exemples de procédures des mathématiques chinoises de l'Antiquité des analogies structurelles avec les algorithmes actuels. Pour compléter, signalons deux algorithmes, exceptionnels pour l'époque : une procédure d'interpolation linéaire, qui ne sera connue en Europe qu'avec Fibonacci sous le nom de *règle de la fausse position*, et la procédure du Fangcheng, la plus élaborée des *Neuf chapitres*, qui résout un système linéaire selon une méthode en tout point identique à celle du pivot de Gauss. Ces algorithmes élaborés nous permettent d'apprécier pleinement la technique des Chinois dans les maths.

F. L.

Numération chinoise

Les Chinois développèrent un système de numération décimal et positionnel dès 1600 avant Jésus-Christ. Les nombres étaient représentés par des baguettes en bambou, disposées verticalement dans les colonnes de rang impair et horizontalement dans celles de rang pair. Deux lots de baguettes parallèles permettent ainsi, le plus souvent, de déterminer les espaces vides (les zéros). Cette méthode de notation, dite « du Wei », apparaît près de mille ans avant son équivalente indienne ! Pour des raisons pratiques, ce système de baguettes a naturellement évolué vers le Ve siècle en différents systèmes de perles, de valeurs associées à des couleurs, précurseurs du boulier (XIIIe siècle).

Numération savante (–200).

Algorithme selon Knuth

Il n'existe pas de définition universellement admise du mot *algorithme*. Donald Knuth (né en 1938), créateur des logiciels TeX et Metafont et auteur de la bible inachevée *The Art Of Computer Programming* (TAOCP), a donné une liste de cinq propriétés largement reconnues comme prérequis d'un algorithme :

Finitude : Un algorithme doit toujours se terminer après un nombre fini d'étapes.

Précision : Chaque étape d'un algorithme doit être définie précisément ; les actions à transposer doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas.

Entrées : Quantités, prises dans un ensemble d'objets spécifié, qui sont données à l'algorithme avant qu'il ne commence.

Sorties : Quantités qui ont une relation spécifiée avec les entrées.

Rendement : Toutes les opérations que l'algorithme doit accomplir doivent être suffisamment élémentaires pour pouvoir être en principe réalisées dans une durée finie par un homme utilisant du papier et un crayon.

par Jean-Christophe Novelli

Les algorithmes du secret : la cryptographie

Depuis les simples substitutions de lettres à l'époque de César, l'art de dissimuler aux regards indiscrets le contenu de messages, notamment militaires, dépend aujourd'hui de l'arithmétique, de l'informatique et d'algorithmes puissants.

La machine *Enigma*.

Jules César est le premier a avoir compris toute l'importance de l'art de coder et de décoder les messages pour transmettre des instructions secrètes à ses officiers. En effet, un ordre militaire n'est efficace que si l'ennemi n'y a pas accès. Et comme on ne peut compter sur le sens du sacrifice du messager pour ne pas divulguer l'information critique, le premier principe de la cryptographie était né : le messager ne doit disposer ni de l'information, ni du moyen de l'obtenir. L'interception reste utile pour bloquer le transfert d'information, mais n'apprend rien.

Le codage de César

Le codage dit *de Jules César* est simple : faire un décalage de trois lettres sur toutes les lettres d'un message (dans la suite, les lettres du message en clair seront en minuscule et les lettres du message codé en majuscule). Dans l'algorithme de César, *a* devient D, *b* devient E, *etc*.

Cet algorithme, très simple, a plusieurs défauts. Déjà, si un ennemi acquiert une connaissance du procédé, il lui est très facile de décoder immédiatement tout message reçu. En outre, s'il se trouve devant un message codé et le même message en clair, il comprendra immédiatement les règles de cryptage.

La méthode qui est venue après celle de César a tenu compte de ces défauts : au lieu de coder chaque lettre par un même décalage, on transforme chaque lettre en une autre selon une règle de correspondance telle que :

b	*c*	*d*	*e*	*f*	*g*	*h*	*i*	...	*z*	*a*
C	H	A	R	L	E	S	B	...	Y	Z

On choisit une lettre de départ (le *b*, dans notre exemple) et un mot clé (CHARLES) et on ordonne les lettres par le codage décrit ci-dessus, les lettres n'appartenant pas au mot CHARLES étant classées par ordre croissant ensuite. Ce moyen semble assez efficace (nombre de combinaisons possibles important et correspondance facile à retenir). Mais cet algorithme est vulnérable : en français, les

lettres n'ont pas la même fréquence d'apparition ! Par exemple, la seule lettre *e* constitue en moyenne 15 % des lettres d'un texte ordinaire. Une simple analyse statistique du texte codé suffit alors à retranscrire les lettres les plus fréquentes, puis les autres. Mais un pas essentiel a été accompli : distinguer le *procédé de chiffrement* (ici, la substitution mono-alphabétique) de la *clé* (CHARLES). Le premier peut maintenant être connu de tous, seule la clé doit être tenue secrète.

Truquer les statistiques

Une première amélioration consiste à employer une substitution poly-alphabétique (voir l'encadré). Une deuxième solution consiste à utiliser plusieurs codages différents pour une même lettre. Par exemple, on peut coder par AR, FE, QT, YW, QT, QU, JA, LA, HB, PA, BS, BF, TH, PQ et BP la seule lettre *e* et par HG la lettre *z*. Ainsi, la lettre *e*, environ quinze fois plus fréquente que la lettre *z*, est remplacée par un mot de deux lettres parmi quinze alors que le *z* est toujours transformé en un seul même mot. Les statistiques sur les paires de lettres donneront des probabilités d'apparition qui peuvent être rendues rigoureusement identiques, rendant inopérante une attaque statistique. Mais… Mais certains mots apparaissent plus fréquemment dans les textes que d'autres ! Par exemple, les courriers du Vatican (dont les cryptanalystes ont toujours compté parmi les plus chevronnés) contenaient des mots simples, tels « Votre Seigneurie », « Votre Grâce »… Si l'on code chaque lettre des mots par le procédé précédent, un cryptanalyste reconstruira le codage de certaines lettres, puis utilisera ce début de rébus pour avancer.

La substitution poly-alphabétique

Un algorithme de substitution poly-alphabétique est le suivant : on décale les lettres d'un mot en fonction de la place des lettres, et non en fonction des lettres elles-mêmes. On se munit d'une clé, par exemple CHARLES, et on opère des codages à la Jules César avec les $(k+7)^{\text{èmes}}$ lettres du texte en utilisant la $k^{\text{ème}}$ lettre de la clé. Voici un exemple d'application.

C H A R L E S C H A R
s o u v e n t p o u r
U V U M P R L R V U I

En effet, « $s + 2 = \text{U}$ », « $o + 7 = \text{V}$ », « $u + 0 = \text{U}$ », « $v + 17 = \text{M}$ »… Une analyse statistique sur les lettres ne révèle rien. Mais si on effectue une analyse statistique sur : une lettre sur deux, puis une sur trois, puis une sur quatre, *etc.*, on finit par trouver la longueur de la clé : c'est le nombre où, par miracle, se dessine soudain la courbe des fréquences d'apparition des lettres en français !

L'époque moderne

À partir du milieu du XIXe siècle, des méthodes réalisées par des machines font leur apparition, pour culminer avec l'invention de la machine allemande *Enigma* durant la Seconde Guerre mondiale. Le principe était celui d'une substitution poly-alphabétique. Puis, à la fin des années 1970, le système RSA révolutionne la cryptographie. La clé est entièrement publique. L'algorithme de cryptage repose sur la difficulté de factoriser des très grands nombres. Aujourd'hui, la recherche est active à la fois pour trouver des algorithmes de cryptage de plus en plus efficaces, et pour casser les codes existants : cryptographie quantique, cryptographie sur les courbes elliptiques… Considérée comme élément clé de la défense nationale dans tous les pays, cette partie des mathématiques est entourée d'une surveillance comparable à celle exercée sur la physique nucléaire.

J.-C. N

Références

Cryptographie et codes secrets. Bibliothèque *Tangente* 26, 152 pages, 2006.
La guerre déchiffrée. Dossier dans *Tangente* 91, 2003.
Histoire des codes secrets. Simon Singh, JC Lattès, 430 pages, 1999.

par Hervé Lehning

Alan Turing et sa machine

Avant l'avènement du premier ordinateur, Alan Turing a conçu une machine dans le but de préciser ce qui est calculable. Il est inutile d'en chercher une dans le commerce : c'est un objet théorique, pas une réalisation industrielle.

Comment définir, de façon rigoureuse, ce qui est calculable et ce qui ne l'est pas ? Alan Turing (1912–1954) a répondu à la question en créant les machines qui portent son nom. N'en cherchez pas dans le commerce, il ne s'agit pas d'une machine au sens industriel, il s'agit d'un outil théorique.

Calculabilité et algorithmes

Dans les années 1930, Turing désire préciser ce qui peut être calculé de manière algorithmique, que ce soit par l'homme ou par une machine. Pour cela, il conçoit un modèle de calculatrice tellement primaire que l'on a du mal à imaginer qu'elle suffise pour effectuer tous les calculs imaginables. En effet, elle consiste en une machine à écrire modifiée qui, au lieu de travailler sur une feuille de papier, travaille sur un ruban illimité dans les deux sens, au moyen d'une tête de lecture / écriture. De ce point de vue, on pourrait la comparer à un magnétophone. La bande est constituée de cases, chaque case contenant un symbole. Ce symbole peut être un blanc, on dit alors que la case est vierge. La tête de lecture / écriture est capable de :

- Lire le symbole contenu dans la case où elle se trouve ;
- Y écrire un symbole (éventuellement un blanc) ;
- Se déplacer d'une case vers la gauche ou la droite.

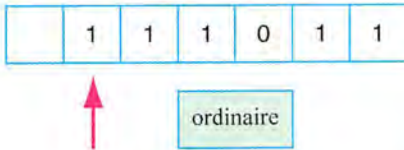
L'action s'effectue uniquement en

Tout ce qui est calculable peut l'être avec une machine de Turing.

fonction de l'état où elle se trouve au moment de la lecture.

Addition avec une machine de Turing

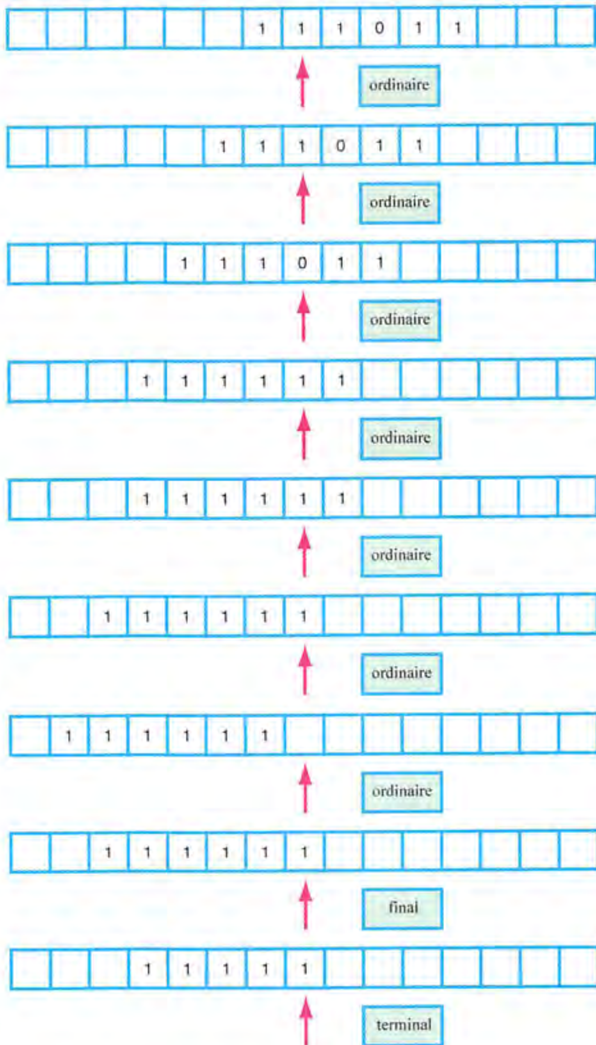
Comment une machine aussi simple peut-elle effectuer des calculs ? Voyons l'exemple de l'addition de deux nombres écrits en base 1, c'est-à-dire avec des bâtons. « 3 » s'écrit ainsi 111 ; « 2 » s'écrit 11. Pour additionner ces deux nombres, on les écrit à la suite mais séparés par un 0 sur le ruban et on positionne la tête de lecture sur le premier « 1 » à gauche :



Sur cette machine de Turing, la tête de lecture / écriture est matérialisée par une flèche rouge. L'état est indiqué dans la case verte.

L'état est noté « ordinaire ». Voici alors les règles :
Si dans cet état « ordinaire », la tête lit un 1, elle se déplace vers la droite et reste à l'état « ordinaire ». Si elle lit un 0, elle se déplace également vers la droite en restant à l'état « ordinaire » mais change la valeur 0 en 1. Si elle lit un blanc, elle se déplace vers la gauche et passe à l'état « final ». Dans cet état « final », si elle lit un 1, elle l'efface (la case devient vierge), la tête se déplace vers la gauche et passe en état « terminal » ; si elle lit un 0 ou un blanc, elle s'arrête. Dans l'état « terminal », elle s'arrête, quoi qu'elle lise. Suivons notre exemple. Les règles précédentes donnent la suite d'états ci-contre.

À ce stade, le ruban contient le résultat. Il est facile de multiplier les exemples simples de ce type. Mieux : Turing a montré que tout calcul automatisable peut être effectué avec sa machine. Bien entendu, elle est loin d'être la plus performante pour cela. Son intérêt est uniquement théorique.



Suite des états de la machine de Turing qui additionne. On obtient bien le résultat escompté.

L'origine du mot « ordinateur »

En 1955, la filiale française de la société International Business Machines Corporation (IBM France) cherchait un mot français pour remplacer le terme anglais computer. Le mot ordinateur lui a été suggéré par Jacques Perret (1906–1992), professeur de latin à la Sorbonne et théologien catholique (et qu'il ne faut pas confondre avec l'écrivain homonyme). Jacques Perret proposa ce terme car, au Moyen Âge, il désignait « Dieu qui met de l'ordre dans le monde », avant de tomber en désuétude (on lui préférera « ordonnateur »). Diviniser les computers en ordinateurs explique-t-il les fameux « C'est la faute à l'ordinateur ! » utilisés trop souvent pour expliquer les manquements des services administratifs ?

La machine universelle

Avant d'aborder l'usage que Turing a fait de ses machines, il convient de remarquer que tout ce qui peut être fait par une machine de Turing peut l'être par une seule d'entre elle : la machine de Turing universelle. Pour comprendre comment celle-ci est conçue, analysons la construction d'une machine de Turing ordinaire. Il s'agit de règles d'écritures et de changements d'états. Ces règles peuvent être entièrement décrites sur un ruban, de sorte que la machine les effectue lors de leur lecture sur une autre partie du ruban. La condition initiale du ruban peut alors être reproduite dans cette partie du ruban. On voit ainsi comment on peut concevoir une machine de Turing universelle. Cette idée de machine universelle est à la base de l'avènement des ordinateurs : les programmes sont des données particulières de la machine. Ils se différencient ainsi des anciennes tabulatrices que l'on devait programmer en changeant physiquement des connections électriques à chaque nouvelle utilisation.

Les problèmes calculables… et les autres

Turing appelle *calculable* tout nombre calculable par sa machine universelle, ce qui revient à faire l'hypothèse implicite que tout ce que peut calculer l'homme peut l'être par une machine de Turing. Comme il est difficile d'imaginer une façon de calculer qui ne serait pas de ce type, cette thèse n'a jamais été contestée, même si elle peut paraître choquante d'un point de vue philosophique. L'homme pourrait-il être égalé par sa créature ? Nous reviendrons sur ce point ; notons simplement que l'homme reste le « maître » puisqu'il conçoit les programmes de la machine, celle-ci n'invente rien. Notez d'autre part que le mot « calculable » doit être pris dans un sens très large. Par exemple, un problème d'échecs du type « Les Blancs jouent et font mat en deux coups. » est calculable. Il est en principe possible de rentrer les règles du jeu d'échec et la position des pièces sur l'échiquier dans une machine de Turing, et de lui faire produire toutes les possibilités sur deux coups. Si une voie conduit au gain dans tous les cas, nous avons la réponse au problème. Ainsi, on peut parler de problèmes calculables : ceux résolus par une machine de Turing, et donc plus généralement par un ordinateur.
La machine de Turing a ainsi permis de classifier les problèmes : les calcu-

lables et les non calculables. Il en existe du deuxième type, par exemple le *problème de l'arrêt d'une machine de Turing*. Parmi les problèmes calculables, certains le sont de façon purement théoriques : le temps de calcul est trop long pour qu'on puisse l'effectuer en un temps raisonnable (quelques années), même avec les plus puissantes machines disponibles. Arbitrairement, on convient d'accepter que seuls les problèmes dont le temps de calcul est une fonction au plus polynomiale de la taille des données sont calculables : on parle de problèmes de complexité P (« P » pour « polynôme »). Il reste beaucoup à faire dans ce domaine pour accélérer la résolution des problèmes classiques comme le tri (indispensable à la gestion des bases de données) ou celui du voyageur de commerce.

Esprit humain *vs.* machine de Turing

Cette assimilation de « calculable » à « calculable par une machine de Turing » amène une question de nature philosophique. Sommes-nous des machines de Turing ? Si l'esprit de l'homme était une machine de Turing, alors on pourrait espérer créer des machines à penser. Certains le redoutent (voir l'encadré *L'origine du mot « ordinateur »*), d'autres en doutent. Peut-on penser sans éprouver d'émotion ? Une pensée est-elle possible sans ce moteur ? Turing s'est posé cette question. C'est pourquoi il a imaginé un test permettant de décider si une machine pense. Voici l'expérience proposée par Turing : on met un testeur d'un côté, et une machine et un humain de l'autre. Ils ne peuvent communiquer que par écrit bien sûr. Si le testeur se fait pas faire la différence entre l'humain et la machine, alors la machine pense. Bien entendu, cela dépend du testeur. Certains sites de rencontre sur Internet utilisent quelques robots pour animer leurs sites. Les clients font-ils toujours la différence ? Pas tous, paraît-il...

H. L.

L'homme qui croqua la pomme

Outre ses travaux sur la calculabilité et l'invention de sa célèbre machine, Alan Turing est connu pour avoir dirigé l'équipe anglaise qui, pendant la Seconde Guerre mondiale, décrypta les messages des armées allemandes transmis par la machine Enigma.
Selon certains, il se serait donné la mort d'une façon qui rappelle le conte de Blanche-Neige : en croquant une pomme empoisonnée au cyanure. Certains prétendent que la société Apple s'en inspira pour créer son logo ; d'autres affirment que ce logo proviendrait de la pomme de Newton...

par Jean-Jacques Dupas

Lady Augusta Ada King, comtesse de Lovelace

Dans les années 1980, le langage Ada a rendu hommage à une femme, dont la légende veut qu'elle ait été « le premier programmateur » de l'histoire. Retour sur la vie fulgurante d'une mathématicienne romantique du dix-neuvième siècle.

Lady Ada King est un personnage romantique. Jeune, belle, indépendante, intelligente, unique fille légitime du poète anglais Lord Byron, elle décédera comme lui à l'âge de 36 ans. L'informatique moderne la rendra célèbre.

Ada est la fille de la mathématicienne Annabella Milbanke (1792–1860) et de Lord George Gordon Byron (1788–1824). Pour faire taire la rumeur selon laquelle il aurait une liaison scandaleuse avec sa propre demi-sœur, Lord Byron séduit Annabella, qui lui a déjà refusé sa main par deux fois. Onze mois après leur mariage, Augusta Ada naît, le 10 décembre 1815. Lord Byron n'a semble-t-il pas renoncé à voir sa demi-sœur ; Annabella décide de quitter le poète. De fait, Ada ne connaîtra jamais son père.

Annabella choisit pour sa fille une éducation musicale et scientifique, ce qui est exceptionnel à cette époque. L'idée est d'éloigner Ada de la littérature, et, par la même occasion, de son père. Parmi les professeurs de mathématiques d'Augusta Ada, on peut citer le logicien Augustus De Morgan (1806–1871) et Mary Somerville (1780–1872).

Erreur de traduction

Une biographie d'Ada de 1985, écrite par Dorothy Stein (Université de Londres), affirme que Ada Lovelace était bien trop mauvaise en mathématiques pour écrire un programme sophistiqué. Elle se fondait sur une erreur d'Ada, qui avait traduit « le cas $n = \infty$ » par « cos $n = \infty$ ». Cette traduction avait été relue par Babbage, et d'autres, qui n'avaient pas vu l'erreur...

La rencontre d'une vie

Grâce à cette dernière, le 5 juin 1833, Ada rencontre Charles Babbage (1791–1871) lors d'une réception. Elle a 17 ans, lui en a 41. Il est alors autant connu pour son activisme politique que pour ses travaux en économie (qui inspireront Karl Marx) et en mathématiques. Babbage expose les principes de sa première machine à Ada, fascinée. Il lui parle ensuite de la *machine analytique*, beaucoup plus élaborée, dont il passera le reste de sa vie à peaufiner les

plans dans l'espoir de la faire construire. Ce n'est qu'en 1992 que la « machine analytique » sera construite sur les plans originaux de son concepteur. Elle fonctionne effectivement (à la vapeur, avec des roues et engrenages mécaniques). L'architecture en est étonnement moderne : elle s'articule autour d'un magasin (la mémoire), d'un moulin (le processeur), d'un lecteur de cartes perforées (l'unité d'entrée), inventé quelques décennies auparavant par Joseph Marie Jacquard pour les métiers à tisser, ...
L'Italien Federico Luigi publie en français *Notions sur la machine analytique de Charles Babbage*. Ada lit, traduit et annote cet article, puis publie le résultat sous forme de notes consacrées à la programmation. Ce sera pendant près d'un siècle le seul article de ce genre. La dernière note décrit un programme, permettant de calculer les nombres de Bernoulli, qui utilise toutes les possibilités de la machine analytique (branchements conditionnels, boucles...). Un tel programme est bien plus complexe que ceux qu'avait envisagés Babbage. Il est considéré par beaucoup comme le premier programme de l'histoire.

Polémique

Le programme publié par Ada est polémique : certains pensent que Babbage le lui a dicté, d'autres défendent que c'est une initiative d'Ada (voir l'encadré). De fait, on ne connait l'œuvre d'Ada qu'au travers d'une biographie (écrite par Babbage !), des cahiers de Babbage, et de leur correspondance. L'étude attentive de cette dernière montre que Ada a en fait demandé à Babbage de lui fournir les formules servant de base à son programme, mais que l'idée du programme sur les nombre de Bernoulli est bien de la jeune femme. Pour être complet, précisons que l'étude des cahiers de Babbage montre que, dans les années précédentes, il avait déjà écrit de nombreux petits programmes ; c'est donc apparemment bien lui le premier programmeur de l'histoire.
À partir de 1843, Ada, très malade, décline rapidement, rongée par l'alcool et les drogues qui rendent ses douleurs supportables. Elle se ruine au jeu, appliquant une « martingale infaillible » de son crû ; elle espérait ainsi financer les travaux de Babbage. Elle décède le 27 novembre 1852, probablement d'un cancer de l'utérus. À sa demande, elle est enterrée avec son père, qu'elle n'a jamais connu.
Ada Lovelace reste un des pionniers, et un des personnages les plus attachants, de l'histoire de l'informatique.

J.-J. D.

Le langage Ada

À la fin des années 1970, le Department of Defence américain prend conscience de la « babélisation » (multiplication des langages) de l'informatique. Cette multiplication est à l'origine de nombreuses faillites de grands projets. Aussi fait-il un appel d'offre pour mettre en place un ultime langage, universel, dont la spécification serait révisée tous les dix ans. L'équipe française emmenée par Daniel Ichbiah l'emporte. Le langage Ada est donc construit à partir de spécifications. Nombre de projets d'envergure doivent leurs succès à l'utilisation de ce langage moderne (Ariane 5, le TGV, le National Ignition Facilities NIF...). Malgré tous les apports d'Ada (exceptions, généricité...), la norme Ada83 est passée à côté du paradigme « objet ». Cet oubli sera rattrapé dans la norme Ada95, qui sera même le premier langage objet normalisé. Mais la déferlante C++, Java... est déjà passée, et aujourd'hui Ada n'est plus qu'un bon souvenir. Peu de programmeurs connaissent la dernière mouture du langage Ada2005. Espérons que Lady Lovelace ne retombera pas elle-même dans l'oubli !

Références
Lady Ada et le premier ordinateur. Eugène Eric Kim, Betty Alexandra Toole, Pour la Science n°261, juillet 1999.
Augusta Ada King comtesse de Lovelace. Alain Zalmanski, Tangente n°111, juillet–août 2006.

par Jean-Jacques Dupas

Femmes et algorithmes

Barbara Liskov

Barbara Liskov (née en 1939) fut la première femme à obtenir un doctorat d'informatique. Elle reçut le prix Turing en 2008 (« Nobel » de l'informatique, doté de 250 000 $).
Elle est en poste au MIT (Massachusetts Institute of Technology) depuis 1972, où elle dirige le groupe de méthodologie de programmation du laboratoire d'informatique et d'intelligence artificielle. Ses premiers travaux servirent à la construction des langages de programmation modernes (Ada, C++, Java...). Ses recherches touchent les langages de programmations, les bases de données objet, les systèmes à tolérance de panne ou l'algorithmique répartie. Ils permirent de rendre les systèmes informatiques plus robustes et plus sûrs.
Barbara Liskov est connue des développeurs objet pour son *principe de substitution*, établi avec Jeannette Wing : Si $q(x)$ est une propriété démontrable pour tout objet x de type T, alors $q(y)$ est vraie pour tout objet y de type S avec S un sous-type de T.
Dit autrement : une fonction qui utilise un objet d'une classe mère doit pouvoir utiliser toute instance d'une classe dérivée sans avoir à le savoir.

Grace Hopper

Grace Hopper (1906–1992) obtient un doctorat de mathématiques en 1934. Elle s'engage en 1943 dans la Marine américaine pour travailler sur l'un des premiers ordinateurs, dont elle sera la première programmatrice. Elle aimait à raconter que, sur ces premières machines, elle trouva un jour que la panne était due à une mite qui s'était prise dans un relais. L'insecte (bug en anglais) fut extrait du relais et collé dans le journal de bord avec la mention: « First actual case of bug being found ». Le bug informatique était né.
Après la guerre, elle pense qu'un programme devrait pouvoir être écrit dans un langage proche du langage naturel plutôt qu'en langage machine. De cette idée naîtront les compilateurs, et le langage Cobol à la fin des années 50, grâce auquel l'informatique se répandra dans les entreprises : Grace Hopper a permis que l'ordinateur ne reste pas un instrument expérimental réservé à une poignée de mathématiciens.
En 1950, elle affirme que les logiciels finiront par coûter plus cher que le matériel (impensable à cette époque !). En 1986, elle part à la retraite avec le grade de contre-amirale. Jusqu'à son décès, elle donnera de nombreuses conférence sur les débuts de l'informatique.

De l'algorithme au langage de programmation p. 24
Les bases de la programmation p. 28
Les tests de primalité p. 34
Calcul de racines carrées : l'algorithme de Babylone p. 38
Des algorithmes pour créer le hasard p. 44
Les fractions égyptiennes p. 48
Les mariages stables existent p. 50
Sous l'ordinateur, les booléens p. 54
N'abusons pas des organigrammes ! p. 60
Programmer l'algorithme d'Euclide p. 64

ALGORITHMES ÉLÉMENTAIRES ET PROGRAMMATION

On rencontre de nombreux algorithmes dans l'histoire des mathématiques. De nos jours, les opérations manuelles auxquelles les algorithmes donnaient lieu ont laissé la place aux programmes informatiques.

par J.-A. Roddier et M. Rousselet

De l'algorithme au langage de programmation

Au départ, l'algorithmique n'était pas dévolue à l'informatique. L'algorithme d'Euclide tournait bien avant que le transistor soit sorti de terre. Pourtant, s'il est ainsi prouvé qu'elles peuvent être séparées, algorithmique et programmation vont aujourd'hui de pair.

Le mot *algorithme* vient du nom du mathématicien arabe *al-Khwarizmi* (Khiva vers 788 — vers 850 Bagdad) qui fut l'un des inventeurs de l'algèbre et du système décimal. Au Moyen Âge, ses ouvrages ont été très appréciés par les savants européens qui, invoquant l'autorité du mathématicien arabe, utilisaient volontiers la formule latine « dixit Algorizmi » (ainsi parle al-Khwarizmi). Historiquement liée au calcul, la notion d'algorithme s'est progressivement étendue à la manipulation de différents objets, des textes et des images par exemple. Un algorithme c'est simplement une méthode qui sert à résoudre un problème en un nombre fini d'étapes : chercher un mot dans le dictionnaire, classer des mots par ordre alphabétique, classer des nombres par ordre de grandeur, chercher le meilleur parcours possible sur une carte, trouver une racine carrée, construire des listes de nombres premiers, *etc.* On peut décrire un algorithme comme étant une suite d'actions à accomplir séquentiellement, dans un ordre fixé.

Un algorithme est une méthode qui sert à résoudre un problème en un nombre fini d'étapes.

C'était le cas d'algorithmes antiques, comme le plus connu, l'algorithme d'Euclide (voir page 64) qui servait à calculer le PGCD de deux nombres, mais aussi de l'algorithme connu sous le nom de « crible d'Ératosthène », qui permettait de construire la liste des nombres premiers, ou encore celui de Babylone, grâce auquel on extrayait des racines carrées (page 38). Le cryptage des textes par la méthode de César, c'est encore un algorithme. Il a donné lieu de nos jours, grâce à la puissance des ordinateurs, à des méthodes terriblement complexes, tout comme d'autres algorithmes modernes qui portent eux-mêmes sur l'informatique, comme ceux qui permettent de compresser des images.

L'algorithmique enseignée au lycée

Si elle est aujourd'hui rentrée officiellement au programme de lycées (en seconde), la notion d'algorithme n'est pas nouvelle. L'algorithme d'Euclide que nous venons de citer a toujours été enseigné quel que soit le degré de technologie disponible. Car un algorithme est avant tout quelque chose que l'on peut faire fonctionner dans sa tête, ou sur le papier, avant de le faire passer par un circuit électronique.

La *programmation*, qui accompagne cette introduction au lycée, est, quant à elle, plus nouvelle. Certes, si l'on considère que programmer c'est faire en sorte qu'un outil arrive à effectuer une série d'opérations automatiquement, on voit que lorsque les élèves construisaient une feuille de calcul sur un tableur, ils programmaient déjà. Mais tout ne peut être fait *via* un tableur, dont, malgré les progrès, la capacité et la vitesse sont limitées. Les langages informatiques offrent aujourd'hui des moyens pratiques pour programmer sans contrainte, et en tournant avec une rapidité qu'il était grand temps de faire découvrir aux élèves.

Tangente a emboîté le pas à cette réforme, offrant aux élèves de seconde, *via* un manuel interactif révolutionnaire, la possibilité de faire rentrer ces nouveautés dans leur quotidien (rendez-vous sur **www.tangente-education.com**). Quant à cet ouvrage, qui sera bien utile aux lycéens, il a décidé d'adopter, dans les exemples qui accompagnent chacun des articles, les deux principaux langages de programmation préconisés par les programmes scolaires. Nos lecteurs, même réfractaires à l'informatique, sont invités à les essayer.

Timbre à l'effigie d'al-Khwarizmi.

Mais soyez rassurés, même sans vous mettre à la programmation, vous pourrez sans aucune frustration en lire les articles. La non-lecture des programmes, que nous avons relégués en encadrés, ne gênera pas la compréhension des articles.

Les deux langages de programmation choisis : Python et Scratch

Le premier langage, Python, est un logiciel libre disponible en ligne à l'adresse suivante : **www.python.org**. Le langage utilisé est d'une syntaxe classique, l'utilisateur dispose d'un environnement propre à lui faciliter la tâche : reconnaissance des instructions

Interface graphique du logiciel Scratch.

par changement de couleur du texte dès que la bonne instruction trouvée, repérage des erreurs dans le texte du programme, description des fonctions utilisées... C'est, en plus de cela, un logiciel relativement référencé sur Internet, et l'on trouve beaucoup d'exemples d'utilisation en ligne.

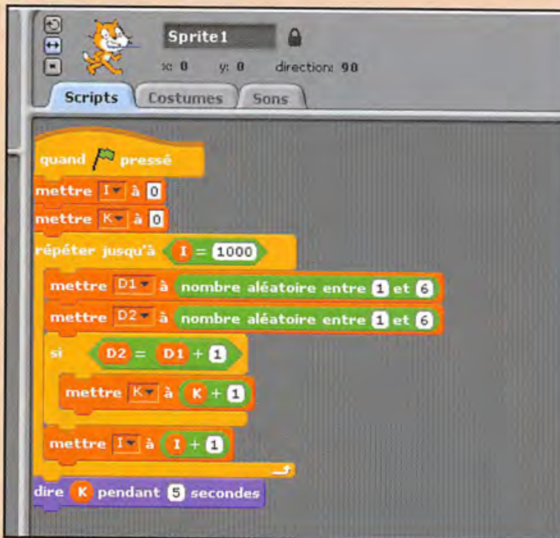
Vous pouvez trouver des rudiments de ce langage en page 27. Les lecteurs qui choisiront de souscrire au manuel interactif *Tangente* disposeront de plus d'un véritable cours complet sur Python.

Le second langage pourrait être le « nec » de demain : il permet de programmer sans retenir de syntaxe ! Vous en avez rêvé, Scratch l'a fait. Le logiciel Scratch a des instructions préconstruites qu'il vous suffit de faire glisser dans un écran actif afin de construire votre programme : vous procédez comme lors d'un *lunch* où vous allez vers le buffet prendre ce que vous voulez.

Et ne vous fiez pas à son interface enfantine ni à la publicité qui proclame qu'il est accessible à des enfants de sept ans. C'est un vrai langage de programmation. Scratch est lui aussi un logiciel libre disponible en ligne à l'adresse suivante : **http://scratch.mit.edu/**

J.-A. R. et M. R.

Un exemple de programme Scratch



On lance deux dés 1 000 fois l'un après l'autre, et l'on souhaite comptabiliser le nombre de fois où les chiffres obtenus sont consécutifs dans l'ordre croissant.

Voici le programme qu'il suffit de construire en faisant glisser les instructions de la colonne gauche de l'écran vers la colonne centrale.

Pour faire fonctionner le programme, il suffit de cliquer sur le drapeau vert.

Rudiments de programmation avec le langage Python

Les principales instructions du langage Python (instructions élémentaires, puis boucles et instructions conditionnelles) sont ici présentées en respectant la trame des commentaires du nouveau programme de la classe de seconde.

Instructions élémentaires

L'affectation

Si vous voulez placer la valeur 4 dans la variable I, il vous suffit de taper :

>>> I = 4

Même chose, si vous voulez incrémenter K de 1 :

>>> K = K + 1

Calculs

Vous pouvez réaliser tous les calculs classiques :

>>> J = 2

>>> K = racine(log(J))

Entrée

Pour demander à l'utilisateur d'entrer une valeur qui sera affectée ensuite à la variable J, il suffit de taper :

>>> J = input(« ? »)

>>> B=float(J)

>>> B

Sortie

Pour faire afficher le résultat d'un calcul, on utilise l'instruction *print* :

>>> L = 5

>>> print (L2/3)**

On utilise aussi l'instruction print pour faire apparaître du texte :

>>> print('Python')

À l'intérieur d'une fonction, on utilise l'instruction *return* :

```
>>> from random import randint
>>> def dé():
        K=randint(1,6)
        return(K)
```

Boucles et itérateurs, instructions conditionnelles

Soit à effectuer 10 000 fois le lancer de deux dés et à comptabiliser le nombre de fois où les deux dés affichent simultanément le même résultat.

>>> def doubledés():	On définit la fonction « doubledés »
I,K=0,0	On initialise les deux variables I et K
while I!=104:**	Tant que I est différent de 10^4
D1=randint(1,6)	On lance le premier dé
D2=randint(1,6)	On lance le second dé
if D1==D2:	Si D1 = D2
K=K+1	On incrémente le compteur K de 1
I=I+1	On incrémente I de 1
return I,K	On fait afficher I et K

L'exécution de la fonction « doubledés » se fait en tapant : **>>> doubledés()**

par Michel Rousselet

Les bases de la programmation

Programmer consiste à transmettre à un ordinateur, à l'aide des instructions d'un langage, l'algorithme qu'il doit appliquer pour parvenir au résultat qu'on lui demande d'établir. Voici, à l'usage des non-informaticiens, les principales caractéristiques de la programmation.

Quand on doit répéter plusieurs fois le même calcul en changeant simplement les données, on a intérêt à utiliser une calculatrice programmable ou un ordinateur. Dans les deux cas, il faudra écrire un *programme* avec un *langage de programmation*.

Il existe actuellement des dizaines de langages de programmation différents, chacun étant caractérisé par un vocabulaire et des règles syntaxiques qui lui sont propres (en quelque sorte, des règles d'orthographe et de grammaire).

Un premier programme

Pour exposer les bases de la programmation, nous utiliserons un « pseudo-langage » dont les instructions sont en français. Bien qu'on ne puisse pas le faire « tourner » sur un ordinateur, ce pseudo-langage est suffisamment proche d'un langage informatique pour qu'on puisse transcrire facilement les programmes élaborés avec lui dans un « vrai » langage de programmation.

Comme premier exemple, voici un programme tout simple qui permet de calculer le périmètre et l'aire d'un disque :

```
1 AFFICHER "Quel est le rayon du
  disque ?"
2 LIRE rayon
3 pi=3.14
4 perimetre=2*pi*rayon
5 aire=pi*rayon^2
6 AFFICHER "périmètre=",
  perimetre
7 AFFICHER "aire=", aire
```

Ce programme se présente comme un texte composé de sept lignes numérotées de 1 à 7 (les numéros ne font pas partie du programme). Il comprend trois parties : l'*entrée des données*

(lignes 1 et 2), le *traitement de ces données* (lignes 3, 4 et 5) et l'*affichage des résultats* (lignes 6 et 7). Les instructions (en rouge) y sont exécutées l'une après l'autre, dans l'ordre des numéros de ligne.

Structure du programme

- Les lignes **1** et **2** servent à l'entrée du rayon du disque qui est la seule donnée utilisée par le programme. L'instruction **AFFICHER** de la ligne **1** demande à l'utilisateur de taper la valeur du rayon sur le clavier.

Le texte **"Quel est le rayon du disque ?"** placé entre des guillemets est une *chaîne de caractères*.

L'instruction **LIRE** de la ligne **2** a un triple rôle :

- elle sert à donner le *nom* « **rayon** » à la *variable numérique* attendue ;
- elle provoque l'arrêt momentané du programme qui va « attendre » qu'une valeur numérique soit fournie par l'utilisateur ;
- elle affecte enfin cette valeur numérique au nom « **rayon** ».

- Les lignes **3**, **4** et **5** contiennent l'ensemble des calculs qui sont effectués par le programme.

La ligne **3** affecte la valeur numérique **3.14** à la *constante numérique* appelée **pi**. Les lignes **4** et **5** définissent deux nouvelles variables numériques appelées respectivement « **perimetre** » (sans accents) et « **aire** ».

Le caractère **=** que l'on trouve dans les lignes **3**, **4** et **5** est une *instruction d'affectation* qui permet d'attribuer une valeur à une variable numérique.

Les caractères * et **^** sont des *opérateurs numériques* qui servent respectivement à la multiplication et à l'élévation à une puissance. Il en existe d'autres comme : **<**, **>**, **<=**, **>=**, *etc.*

- Les lignes **6** et **7** servent à l'affichage des résultats. L'instruction **AFFICHER** qui figure dans ces lignes permet d'afficher sur l'écran de l'ordinateur aussi bien des nombres que des textes.

Pour différentes raisons, les mots utilisés dans les « vrais » langages de programmation sont le plus souvent en anglais. Ils peuvent changer avec le langage : dans un cas, **AFFICHER** et **LIRE** deviendront respectivement **WRITE** et **READ** ; dans un autre cas, ce sera **PRINT** et **INPUT()**.

Les boucles non conditionnelles

Supposons que nous voulions calculer les aires et les périmètres de cinq disques ayant respectivement pour rayons 15 cm, 20 cm, 25 cm, 30 cm et 35 cm. On pourrait évidemment utiliser cinq fois de suite le programme 1 mais il est préférable de le modifier en lui ajoutant une **boucle** qui permettra de faire les cinq calculs demandés. On obtient alors le programme 2 ci-dessous :

```
1 AFFICHER "Quel est le rayon le
   plus petit ?"
2 LIRE rayon
3 pi=3.14
4 POUR i=1, 2, ..., 5 :
5       rayon=rayon+5*(i-1)
6       perimetre=2*pi*rayon
7       aire=pi*rayon^2
8       AFFICHER "périmètre du
        disque n° ", i, "=", perimetre
9       AFFICHER "aire du disque n° ",
        i, "=", aire
10 AFFICHER "Fin du programme"
```

Les boucles conditionnelles

Quand on ne sait pas à l'avance combien de fois une boucle doit être parcourue, il faut utiliser l'instruction TANT QUE... suivie d'une condition qui décidera du moment ou la boucle sera quittée. Les actions demandées seront exécutées tant que la condition choisie reste vraie, mais elles cesseront de l'être quand cette condition ne sera plus vérifiée. On « sort » alors de la boucle et le programme reprend à la ligne suivante. Proposons-nous par exemple de traiter l'exercice suivant : en 2009, la population d'une ville s'élève à 100 000 habitants. Cette population augmente de 0,8 % par an en moyenne. Dans combien d'années aura-t-elle doublé ? C'est ce que calcule le programme 3 ci-dessous :

```
1 p=100000
2 n=1
3 TANT QUE p<200000
4     p=p*1.008
5     n=n+1
6 AFFICHER n-1, p
```

Chaque passage dans la boucle représente une année. Le nombre d'années est représenté par la variable **n**. On sort de la boucle quand, pour la première fois, la population atteint ou dépasse 200 000 habitants. On vérifie (faites le calcul) que cela arrive pour **n** = 87.

Là aussi, quelques remarques s'imposent :

1) Dans ce programme, **p** est la *variable de boucle* mais ce n'est pas une *variable locale*. C'est une *variable globale* qui doit être connue du programme avant que la boucle ne soit introduite dans le programme. On remarque que la première instruction *affecte* la valeur 100 000 à cette variable.
2) Dans les « vrais » langages de programmation, l'instruction TANT QUE est traduite par le mot anglais **WHILE**.

Les instructions SI... ALORS... et SI... ALORS... SINON...

Au rayon musique d'une grande surface, on bénéficie d'une réduction de 8 % pour tout achat d'au moins trois DVD. Tous les DVD concernés par cette promotion valent 18 euros pièce. Le programme 4 calcule le prix à payer pour l'achat d'un certain nombre de DVD.

```
1 AFFICHER "Combien de DVD ont
  été achetés ?"
2 LIRE n
3 SI n>2
4     ALORS montant=18*n*0.92
5     SINON montant=18*n
6 AFFICHER "Montant à payer :",
  montant, "euros."
7 AFFICHER "Fin du programme."
```

Les lignes **1** et **2** permettent d'entrer la valeur de **n**. Le programme exécute les instructions de la ligne **4** si la *condition* qui suit le mot SI sur la ligne **3** est vraie. Si ce n'est pas le cas, il ignore cette ligne et exécute l'instruction de la ligne **5** introduite par SINON. Ensuite, le programme reprend à la ligne **6**.

L'instruction composée SI... ALORS... SINON... est une *instruction conditionnelle*. On peut quelquefois ne pas utiliser le SINON. On peut évidemment utiliser plusieurs instructions SI... ALORS... successivement.

Remarque : en anglais, SI... ALORS... SINON... se traduit de manière variée. On trouve IF... THEN... ELSE..., IF... ELSE..., *etc*.

Les instructions ET et OU

Un libraire offre une réduction de 10 % du montant de ses achats à tout acheteur d'au moins deux romans policiers et d'au moins un livre de cuisine. Les romans policiers valent 16 euros chacun. Les livres de cuisine valent, eux, 26 euros chacun.
Le programme 5 ci-dessous calculera la somme à payer pour tout achat de livres dans les deux collections indiquées.

```
1 AFFICHER "Combien de romans
  policiers avez-vous achetés ?"
2 LIRE polar
3 AFFICHER "Combien de livres de
  cuisine avez-vous achetés ?"
4 LIRE cuisine
5 SI polar>=2 ET cuisine>=1
6     ALORS motant=
      (polar*16+cuisine*26)*0.9
7     SINON montant=
      polar*16+cuisine*26
8 AFFICHER "Fin du programme."
```

L'instruction de la ligne **6** n'est exécutée que si les deux conditions qui sont exprimées ligne **5** et qui sont liées par *l'instruction logique* ET sont vraies simultanément. Dans le cas contraire, l'instruction de la ligne **6** est ignorée et le programme reprend à la ligne **7** qui est donc utilisée dans plusieurs cas de figures (trois exactement).
Si on remplace le ET de la ligne **5** par un OU, on change évidemment les conditions d'exécution de la ligne **6**. Celle-ci n'est exécutée que si l'une des deux conditions au moins est vraie.

Des boucles de Python.

Le pseudo-langage de programmation que nous avons utilisé ne permet pas de faire « tourner » les programmes qu'il a permis d'écrire. C'est évidemment un peu frustrant ! C'est pourquoi nous invitons le lecteur intéressé à utiliser le langage de programmation Python. C'est un logiciel librement téléchargeable (voir l'adresse en page 25). Une introduction à ce langage vous est proposée ci-contre, et un cours complet de programmation dans le langage Python est disponible sur le site du manuel interactif de seconde. Pour les réfractaires à tout effort de mémoire consistant à retenir la syntaxe d'un langage, il existe un logiciel de programmation qui vous en affranchit : Scratch, également téléchargeable gratuitement (voir en page 26).

M. R.

Un jeu pour faire aimer les algorithmes

Les cartes du jeu Algorilude permettent de mettre en œuvre tous les concepts de base liés aux algorithmes : boucles, itérations, variables, affectation, compteurs, branchements, tests, arrêt (...). Le jeu consiste à construire pas à pas des organigrammes afin de totaliser le maximum de points possible. Le décompte précis du nombre de points de chaque joueur nécessite d'avoir compris toutes ces notions. Le jeu, créé par Louis Abraham en 1989, se joue de quatre à six joueurs, même si la notice précise *« trois à sept à la rigueur »* (à moins de quatre joueurs, le jeu n'est pas palpitant, et à plus de six, les cartes commencent à faire défaut). Par ailleurs, les règles du jeu sont difficiles à appréhender, mais il faut passer par cette phase d'apprentissage pour comprendre toute la richesse d'Algorilude. Enfin, deux niveaux de difficulté sont possibles (« difficile » en utilisant toutes les cartes, « facile » en n'utilisant qu'une partie des cartes).
Le jeu nous semble idéal au lycée pour un atelier encadré par un enseignant.

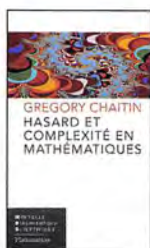
Algorilude, pour jouer comme un pro...grammeur.
Librairie d'Archimède, 41 rue Jean de Becker, 27 940 Aubevoye,
Éditions POLE, 80 boulevard Saint-Michel, 75 006 Paris,
8 € (30 € pour 5 exemplaires).

par Édouard Thomas

Références sur l'algorithmique

• Autour de la cryptographie

- Cryptographie et codes secrets. Bibliothèque *Tangente* 26, 152 pages, 2006.
- La guerre déchiffrée. Dossier dans *Tangente* 91, 2003.
- Histoire des codes secrets. Simon Singh, *JC Lattès*, 430 pages, 1999.

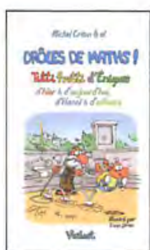


• Autour de l'arithmétique

- Logique, informatique et paradoxes. Jean-Paul Delahaye, *Pour la Science*, 158 pages, 1995.
- Le fascinant nombre pi. Jean-Paul Delahaye, *Pour la Science*, 1997.
- Merveilleux nombres premiers. Jean-Paul Delahaye, *Pour la Science*, 336 pages, 2000.
- Complexités. Jean-Paul Delahaye, *Pour la Science*, 256 pages, 2006.
- Hasard et complexité en mathématiques. Gregory Chaitin, *Flammarion*, 236 pages, 2009.

• Autour de la preuve

- À la recherche de la preuve en mathématiques. Hervé Lehning, *Pour la Science*, 128 pages, 2009.
- La vraie nature de l'intelligence. Dossier dans *Science et Vie* 1013, pp. 38–57, février 2002.



• Autour du jeu

- Marc Charnay dans *Pentamino* 1, IREM de Grenoble, 1976.
- Récréations mathématiques. Édouard Lucas, tome 3, *Blanchard* (réédition), 210 pages, 1979.
- Brian Hayes dans *Pour la Science* 79, mai 1984.
- Drôles de maths, tutti frutti d'énigmes. Michel Criton et l'Association pour le développement de la culture scientifique, *Vuibert*, 187 pages, 2008.

• Autour des graphes

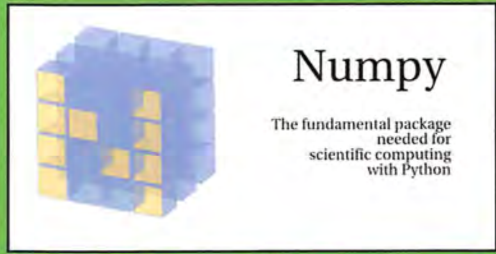
- Les jeux de Nim. Jacques Bouteloup, *Association pour le développement de la culture scientifique*, Amiens, 299 pages, 1996.

par Pierre Raybaut

Les langages dynamiques

Qu'il s'agisse de traiter des signaux ou des images, de simuler numériquement un effet physique, de piloter une expérience ou encore de visualiser des données en 2D ou 3D, l'interactivité est certainement la qualité principale recherchée par les scientifiques dans un langage de programmation. Si l'on y ajoute la rapidité de développement, on comprend l'engouement pour les langages dits *dynamiques*, c'est-à-dire ne nécessitant pas d'étape de compilation avant l'exécution d'un programme. Dans ce domaine, de nombreuses solutions existent. Cependant, aucune d'entre elles ne réunit actuellement autant d'avantages pour les scientifiques que le langage Python, un langage haut niveau, libre et gratuit doté d'une communauté d'utilisateurs étendue et active.

Page Internet de la bibliothèque SciPy. www.scipy.org



PYTHON pour taupins ? Potins...

En employant des bibliothèques adaptées aux besoins de l'utilisateur, Python devient un outil puissant dans les mains d'un chercheur ou d'un ingénieur. Il combine interactivité (s'agissant d'un langage interprété), performance de calculs (grâce à sa faculté singulière à se combiner à des langages compilés tels que C, C++ ou Fortran) et facilité d'apprentissage. Fondamentalement orienté objet, le langage Python est adapté à l'écriture du simple script à la plus complexe des applications. Les utilisateurs de langages spécialisés commerciaux, tels que Matlab ou IDL, seront séduits par des bibliothèques généralistes de qualité (interfaces graphiques, réseau, bases de données, calcul parallèle, *etc.*). Ils retrouveront également des bibliothèques scientifiques (comme NumPy pour le calcul matriciel, SciPy pour le traitement du signal et de l'image et Matplotlib pour les représentations graphiques) et des environnements de développements très aboutis. Par exemple, Spyder (Scientific Python Development Environment), qui est un environnement de travail interactif aux fonctionnalités proches de celles de Matlab. La solution la plus simple aujourd'hui pour installer l'ensemble de ces outils consiste à utiliser une distribution Python scientifique telle que Python(x,y) qui propose un choix auto-suffisant de bibliothèques, documentation et environnements de développement. Concrètement, Python(x,y) est une distribution Python scientifique libre et gratuite pour Microsoft Windows et GNU/Linux incluant Eclipse et Spyder (voir le site officiel à l'adresse www.pythonxy.com).

par M. Criton et E. Thomas

Les tests de primalité

Les nombres premiers fascinent les mathématiciens depuis toujours. Mais la détermination du caractère premier ou non premier d'un grand nombre prend beaucoup de temps. Les chercheurs se sont donc mis en quête d'algorithmes pour aller plus vite.

Longtemps, la vérification de la primalité d'un nombre s'est faite à la main. Des mathématiciens, amateurs ou professionnels, ont ainsi passé des milliers d'heures à faire des calculs, tels des divisions euclidiennes. Depuis qu'ils disposent d'ordinateurs, les limites ont été repoussées et ce ne sont plus des humains qui calculent, mais des machines. Les mathématiciens se concentrent à nouveau sur la recherche d'algorithmes toujours plus rapides et plus économes en temps de calcul, et sur la programmation de ces algorithmes.

La méthode d'Ératosthène

La méthode la plus « basique » pour établir qu'un nombre N est premier consiste à essayer de le diviser par tous les entiers strictement positifs qui lui sont inférieurs, et à constater qu'aucune de ces divisions ne tombe juste.

On gagne déjà du temps et de l'énergie en se limitant aux nombres inférieurs ou égaux à la racine carrée du nombre à tester. On peut ensuite ne pas tenter inutilement de diviser le nombre N par les entiers pairs strictement supérieurs à 2. En effet, si le nombre n'est pas divisible par 2, il ne sera divisible par aucun nombre pair. On a ainsi éliminé les nombres pairs plus grands que 2 des diviseurs possibles de N. Si le nombre n'est pas divisible par 3, on peut faire la même chose pour les multiples de 3 plus grands que 3. S'il n'est pas divisible par 5, on peut éliminer les multiples de 5 plus grands que 5, c'est-à-dire tous les entiers se terminant par un 5 (puisque les nombres pairs ont déjà été éliminés...).

Ce type de test devient vite inapplicable pour de très grands nombres. Même avec des ordinateurs très rapides, le temps de calcul croît si vite

avec le nombre de chiffres de N que cela devient rapidement impraticable à l'échelle de temps des êtres humains. Pour contourner cette limitation, plusieurs types d'algorithmes ont été développés. Les tests probabilistes et les tests conditionnels comptent parmi les plus répandus. Tous ou presque sont basés sur un outil central : le Petit Théorème de Fermat.

Le Petit Théorème de Fermat

Ce « petit » théorème (par opposition au « Grand Théorème de Fermat » qui n'a été démontré par Andrew Wiles qu'à la fin du XXe siècle) a été énoncé en 1640 dans une lettre de Pierre de Fermat (vers 1600–1665) à Bernard Frénicle de Bessy et a été démontré ensuite par Leibniz, puis par Euler.
Le théorème s'énonce ainsi :
Si p est un nombre premier et a un nombre entier non divisible par p, alors $a^{p-1} - 1$ est un multiple de p.
Ainsi, par exemple, le nombre 11 étant premier, quel que soit l'entier a supérieur ou égal à 2 et non divisible par 11, $a^{10} - 1$ sera divisible par 11.
La *contraposée* du théorème permet de déterminer si un nombre est composé. Ainsi, le fait que $2^{90} - 1$ ne soit pas divisible par 91 implique que 91 n'est pas un nombre premier ($91 = 7 \times 13$).

Malheureusement, la *réciproque* de ce théorème n'est pas vraie. Les nombres impairs composés n pour lesquels il existe un a tel que que $a^{n-1} - 1$ soit un multiple de n sont appelés nombres pseudo-premiers de Fermat.
Si $a = 2$, ce sont des nombres *de Poulet* (d'après le nom du mathématicien belge Paul Poulet (vers 1885–1946), qui a publié en 1937 une *Table des nombres composés vérifiant le théorème de Fermat pour le module 2*

Gary Lee Miller, professeur d'informatique à l'université Carnegie Mellon (Pittsburgh, États-Unis).

Les tests probabilistes et conditionnels

Depuis Lehmer, de nombreux tests de primalité ont été mis au point. Mais la plupart d'entre eux sont probabilistes, c'est-à-dire qu'ils permettent seulement d'affirmer qu'un entier à une probabilité très grande d'être un nombre premier. Citons par exemple le test probabiliste de Rabin–Miller, dû à Gary Lee Miller et Michael Rabin : tout nombre premier vérifie un ensemble d'équations diophantiennes dérivées du Petit Théorème de Fermat. L'algorithme de Rabin–Miller appliqué à un entier n donné consiste à vérifier si ce nombre « passe » les tests (c'est-à-dire est solution de toutes les équations). S'il existe une équation non vérifiée par n, alors on peut conclure avec certitude que n n'est pas premier. Par contre, si n passe tous les tests, alors on peut seulement en conclure que n est premier « avec une probabilité donnée ».
D'autres tests s'appuient sur des conjectures non encore démontrées. Leur validité dépend donc de celle de ces conjectures, et tant que celles-ci ne sont pas démontrées, ils ne permettent pas d'affirmer qu'un entier donné est premier. Nous pouvons par exemple citer le test de primalité de Solovay–Strassen, conçu par Robert Martin Solovay et Volker Strassen. Cet algorithme est basé sur une généralisation du Petit Théorème de Fermat, et a inspiré l'algorithme de Rabin–Miller. Il retourne avec certitude soit que n est premier, soit que n est composé, sous réserve que l'hypothèse de Riemann généralisée soit vraie. Cette dernière n'est toujours pas démontrée à l'heure actuelle, malgré les importantes recherches qui lui sont consacrées depuis des décennies (en particulier, un prix d'un million de dollars offert par le Clay Mathematics Institute sera offert à ceux qui en viendront à bout).

jusqu'à 100 000 000). Les premiers nombres de Poulet sont 341, 561, 645, 1 105, 1 387.

Si la propriété est vérifiée pour tout a inférieur à n et que n n'est pas premier, ce sont des nombres dits *de Carmichael* (ils doivent leur nom au mathématicien américain Robert Daniel Carmichael, 1879–1967). C'est le cas des nombres 561, 1 105, 1 729 par exemple. C'est à partir du résultat de Fermat, datant du XVIIe siècle, que les algorithmes modernes ont été développés.

Le test de Lucas-Lehmer

Ce test de primalité a été imaginé par le mathématicien français Édouard Lucas (1842–1891), puis amélioré et complètement justifié par l'Américain Derrick Henry Lehmer (1905–1991). Il concerne au départ les seuls *nombres de Mersenne*, c'est-à-dire les nombres de la forme $2^p - 1$, où p est un nombre premier.

Le test s'appuie sur la propriété suivante :

Étant donné la suite (s_n) définie par $s_1 = 4$ et, pour $n > 1$, $s_n = (s_{n-1})^2 - 2$, le nombre $2^n - 1$ est premier si, et seulement si, il divise s_{n-1}.

Les premiers termes de la suite (s_n) sont 4, 14, 194, 37 634, 1 416 317 954, 2 005 956 546 822 746 114.

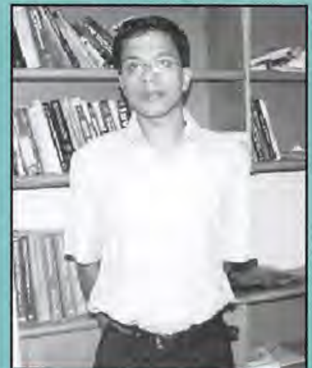
On vérifierait par exemple que $2^7 - 1$ divise s_6 (le dernier terme donné précédemment), mais que $2^{11} - 1$ ne divise pas s_{10} (à savoir le nombre gigantesque :

68 729 682 406 644 277 238 837 486 231 747 530 924 247 154 108 646 671 752 192 618 583 088 487 405 790 957 964 732 883 069 102 561 043 436 779 663 935 595 172 042 357 306 594 916 344 606 074 564 712 868 078 287 608 055 203 024 658 359 439 017 580 883 910 978 666 185 875 717 415 541 084 494 926 500 475 167 381 168 505 927 378 181 899 753 839 260 609 452 265 365 274 850 901 879 881 203 714).

On peut en conclure que 127 est premier, mais que 2 047 ne l'est pas ($2047 = 23 \times 89$).

On constate que la difficulté va être ici la croissance exponentielle du nombre de chiffres des termes de la suite (s_n).

Manindra Agrawal


Neeraj Kayal

Nitin Saxena

Références

Bibliothèque *Tangente* n° 26, Cryptographie et Codes secrets, 152 pages, 2006.

Le *blog* des projets Polymath : http://michaelnielsen.org/polymath1/index.php?title=Main_Page

Le test de primalité AKS

Une révolution a eu lieu en 2002. Cette année-là, trois Indiens ont exhibé un test de primalité déterministe et inconditionnel, fonctionnant en temps polynomial. L'algorithme AKS, c'est son nom, n'utilise que des résultats « élémentaires » de théorie des nombres. Manindra Agrawal, né en 1966, professeur au Indian Institute of Technology de Kanpur, a supervisé les recherches de Neeraj Kayal et de Nitin Saxena. Leur résultat est un test de primalité qui permet de savoir, en un temps polynomial, si un nombre donné n est premier ou non. Ce test ne dépend d'aucune conjecture, et est entièrement déterministe. C'est le premier algorithme connu qui vérifie simultanément toutes ces caractéristiques. Il a généré un extraordinaire enthousiasme dans le monde académique ; depuis, il a été considérablement amélioré. Cependant, pour les applications pratiques, on lui préfère des algorithmes probabilistes (plus rapide à l'exécution). Mais il n'est pas exclu que les chercheurs arrivent à terme à réduire la complexité de l'algorithme AKS, ce qui en ferait une méthode réellement concurrente (et déterministe) des tests probabilistes aujourd'hui utilisés. Il faut souligner que, malgré les pressions de l'institut dans lequel ces recherches ont été menées, les trois chercheurs ont tenu à rendre entièrement publiques leurs recherches afin qu'elles profitent à tous. Le test AKS s'appuie sur une extension du petit théorème de Fermat à des polynômes : Si n est le nombre à tester et a un nombre premier avec n, on compare les restes de $(x - a)^n$ et de $(x^n - a)$ dans la division par n.

M. C. et E. T

Les projets Polymath

La publication de l'algorithme AKS a ouvert des pistes nouvelles pour les théoriciens. Parmi celles-ci, le projet Polymath4, auquel tous les passionnés sont invités à participer.

Les projets Polymath sont nés en 2009 d'une idée de William Timothy Gowers (médaille Fields 1994). Il se demandait alors, sur son *blog*, s'il était possible, *via* le réseau Internet, de mener un projet de recherche en mathématique qui impliquerait des centaines (voire des milliers) de mathématiciens. L'idée est de permettre à de très nombreux chercheurs de travailler sur un même sujet (dont l'énoncé doit être simple et compréhensible par tous), et de mettre en commun leurs résultats, dans le but de progresser très rapidement. Le premier projet, Polymath1, de nature combinatoire, a permis de résoudre un problème ouvert, difficile et important. Il est apparu, au final, qu'un groupe restreint de mathématiciens a réalisé la majorité des travaux (le processus de rédaction des résultats est en cours). Depuis, sous l'impulsion de nombreux chercheurs enthousiastes, de nombreux projets Polymath ont été proposés et sont en attente de lancement. Le projet Polymath4, intitulé *Finding Primes*, a débuté durant l'été 2009. Il se donne pour objectif de mettre en œuvre le test AKS afin de trouver de grands nombres premiers ayant un nombre de chiffres déterminé. Plus précisément, il s'agit, un nombre k étant donné, de concevoir un algorithme qui, en temps polynomial, retourne un nombre premier p explicite possédant k chiffres. De nombreuses pistes originales ont d'ores et déjà été proposées...

Page Internet du projet Polymath4.

par M. Criton et M. Rousselet

Calcul de racines carrées
L'algorithme de Babylone

Si A est un entier naturel positif, $\sqrt{A}$ est le plus souvent un nombre irrationnel. Se pose alors un problème de calcul : comment trouver une valeur approchée de $\sqrt{A}$ qui soit aussi proche qu'on veut de ce nombre ? Cette question a été résolue il y a longtemps grâce à l'algorithme de Babylone, appelé aussi algorithme de Héron.

Il y a 4000 ans, les mathématiciens babyloniens utilisaient un système de numération sexagésimal (en base soixante). Après transcription en base 10, on a découvert que la tablette YBC 7289 portait la valeur $\sqrt{2} = 1,414222$.

Sur une autre tablette, vieille elle aussi d'environ 4 000 ans, on demande de trouver la diagonale d'une porte de hauteur 40 et de largeur 10. Le scribe qui pose ce problème trouve le résultat 41 et 15/60.

Beaucoup plus tard, au premier siècle de notre ère, la méthode de calcul des Babyloniens a été reprise par Héron d'Alexandrie qui l'a expliquée dans son principal ouvrage *Les Métriques*. Avec les notations qui sont les nôtres, on peut décrire de manière simple l'algorithme utilisé par aussi bien par les Babyloniens que par Héron pour calculer le nombre $\sqrt{A}$:

	A	B	C	D	E	F
1						
2			CALCUL D'UNE RACINE CARREE			
3			*METHODE DE HERON*			
4						
5						
6		Choix du			Numéro de	valeur de
7		nombre N	50		l'estimation	l'estimation
8					1	25
9					2	13,5
10					3	8,60185185
11					4	7,20727684
12					5	7,07235491
13					6	7,07106793
14					7	7,07106781
15					8	7,07106781
16					9	7,07106781
17					10	7,07106781
18					11	7,07106781
19					12	7,07106781
20					13	7,07106781
21					14	7,07106781
22					15	7,07106781
23					16	7,07106781
24						

Avec un tableur

Cet algorithme peut très facilement être porté sur un tableur tel qu'Excel. Ainsi, la feuille de calcul ci-dessus montre le calcul de $\sqrt{50}$:

On observe que la première estimation de $\sqrt{50}$ a été choisie égale à la moitié de 50, soit 25. On observe également qu'à partir de la septième estimation, les décimales des différentes valeurs

La tablette babylonienne YBC 7289.

approchées de $\sqrt{50}$ ne changent plus. On peut en déduire qu'on connaît $\sqrt{50}$ à 0,00000001 près. La justification de cette extraordinaire précision vous est donnée dans l'encadré de la page suivante.

La méthode de Héron appliquée à la racine cubique

On peut généraliser cette méthode en déterminant une racine cubique approchée par approximations successives convergeant vers la racine cubique recherchée. Ainsi, pour calculer la racine cubique de N, on considère une suite définie par un premier élément arbitraire x_1 et la relation :

$$x_{n+1} = \frac{x_n + \frac{N}{x_n^2}}{2}$$

Pour $N = 17$ et en partant par exemple de $x_1 = 1$, on obtient la suite :
1 ; 9 ; 373/81 ≈ 4,6049... ;
60 929 614 / 22 538 898 ≈ 2,7033... ;
2,5147... ; 2,6014... ; 2,5567... ; 2,578...
On parvient très rapidement à approcher :

$$\sqrt[3]{17} \approx 2,57128\,.$$

L'algorithme de Babylone

- Choisir une première estimation e de $\sqrt{A}$;
- Calculer une deuxième estimation e' en faisant la moyenne arithmétique de e et de $\frac{A}{e}$;
- Remplacer e par e' et recommencer pour avoir une troisième estimation e'' ;
- Continuer ainsi autant de fois que nécessaire pour obtenir la précision souhaitée.

Pourquoi une si bonne précision ?

Pour expliquer l'extraordinaire précision de la méthode de Héron, il nous faut faire quelques calculs qui sont peut-être un peu longs mais restent à la portée d'un lycéen.

Étape 1

Soit A un nombre strictement positif. Si x_n et x_{n+1} désignent respectivement les valeurs approchées de rangs n et $n+1$ de $\sqrt{A}$, on peut écrire :

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{A}{x_n}\right) = \frac{x_n^2 + A}{2x_n}.$$

Posons $e_n = \sqrt{A} - x_n$: ce nombre représente l'erreur commise en prenant x_n comme n^e valeur approchée de $\sqrt{A}$.

On a alors : $$e_{n+1} = \sqrt{A} - x_{n+1} = \sqrt{A} - \frac{x_n^2 + A}{2x_n} = -\frac{x_n^2 - 2x_n\sqrt{A} + A}{2x_n}$$

d'où : $$e_{n+1} = -\frac{\left(x_n - \sqrt{A}\right)^2}{2x_n} = -\frac{e_n^2}{2x_n}.$$

Comme e_n^2 et x_n sont des nombres strictement positifs, $-\frac{e_n^2}{2x_n}$ est un nombre strictement négatif. Il en résulte que, pour $n \geq 2$, e_{n+1} est toujours négatif.

À partir de la deuxième, toutes les autres valeurs approchées de $\sqrt{A}$ qui sont calculées sont donc des valeurs approchées par excès.

Étape 2

On peut également écrire $e_{n+1} = -\frac{e_n}{2x_n} e_n$ soit $e_{n+1} = \frac{\left(x_n - \sqrt{A}\right)}{2x_n} e_n$

ou encore $e_{n+1} = \left(\frac{1}{2} - \frac{\sqrt{a}}{2x_n}\right) e_n$.

Comme le nombre $\frac{\sqrt{A}}{2x_n}$ est toujours positif, on en déduit $\frac{1}{2} - \frac{\sqrt{A}}{2x_n} < \frac{1}{2}$ (1).

Par ailleurs, pour $n \geq 2$, on a vu que $x_n > \sqrt{A}$.

On en tire successivement $\frac{1}{x_n} < \frac{1}{\sqrt{A}}$ puis $\frac{\sqrt{A}}{2x_n} < \frac{\sqrt{A}}{2\sqrt{A}}$ et, finalement, $\frac{\sqrt{A}}{2x_n} < \frac{1}{2}$ d'où $0 < \frac{1}{2} - \frac{\sqrt{A}}{2x_n}$ (2).

En combinant les inégalités (1) et (2), on peut donc écrire $0 < \frac{1}{2} - \frac{\sqrt{A}}{2x_n} < \frac{1}{2}$.

Puisque $|e_{n+1}|$ peut s'écrire $\left|\left(\frac{1}{2} - \frac{\sqrt{A}}{2x_n}\right)\right| |e_n|$, on obtient finalement $|e_{n+1}| < \frac{1}{2}|e_n|$.

À chaque nouveau pas, l'erreur diminue au moins de moitié.
Ceci explique qu'on obtienne une très grande précision au bout de relativement peu d'étapes.

On peut en trouver une interprétation géométrique. Chercher la racine cubique de N revient à chercher l'arête d'un cube de volume N. On part d'un nombre a (1er terme de la suite) et d'un parallélépipède rectangle de dimensions :

$$a, \sqrt{\frac{N}{a}}, \sqrt{\frac{N}{a}}.$$

Ce parallélépipède a le même volume qu'un parallélépipède de dimensions (N/a^2, a, a). Le deuxième terme de la suite sera la moyenne arithmétique entre a et N/a^2. On itère ce calcul de moyenne de façon que le dernier parallélépipède rectangle considéré se rapproche de plus en plus d'un cube.

Deux autres algorithmes pour une racine cubique

Voici pour terminer deux autres algorithmes aboutissant eux aussi au calcul d'une racine cubique.

Le premier, dû à Newton, consiste à partir de x_1 et de la relation :

$$x_{n+1} = \frac{2x_n + \frac{N}{x_n}}{3}$$

Toujours pour $N = 17$ et $x_1 = 1$, on obtient la suite :

1 ; 19/3 ≈ 6,333 ; 4,363... ; 3,206... ; 2,68... ; 2,5712...

Cet algorithme remplace la moyenne arithmétique de la méthode de Héron par une moyenne pondérée qui accélère la convergence.

Le second est destiné aux possesseurs d'une antique calculette ne possédant pas de touche « racine cubique » ni de touche « élévation à une puissance », mais une touche permettant d'extraire une racine carrée.

Il consiste à partir de x_1 et d'appliquer la relation de récurrence :

$$x_{n+1} = \sqrt{\sqrt{N x_n}}.$$

Exemple pour $x_1 = 1$ et $N = 17$:

1 ; 2,03 ; 2,423... ; 2,533... ; 2,561... ; 2,5689... ; 2,5706...

Sur ce type de « calculette », on tape généralement sur la touche « racine carrée » après avoir tapé le radicande et non avant, contrairement à ce qui se passe sur les calculatrices scientifiques (hormis les très anciens modèles). En entrant le nombre N en mémoire, il est donc possible de calculer les valeurs approchées successives sans avoir à les saisir à nouveau à chaque étape.

On n'oubliera pas, avant de clore ce sujet, les méthodes de calcul « à la main » de racines carrées et même de racines cubiques que nos grands-pères apprenaient sur les bancs de l'école.

Nos lecteurs les plus anciens ont appris à calculer une racine carrée « à la main » dès les classes du collège, puisque cette technique a fait partie des programmes jusqu'à la fin des années 1950. Il existait aussi des techniques de calcul des racines cubiques qui ont également été enseignées, quoique à une époque plus lointaine.

L'extraction de la racine carrée « à la main »

Cette technique fut enseignée aux collégiens jusqu'au milieu des années 1960. Elle disparut ensuite des programmes, précédant de peu l'introduction des calculatrices dans les classes.

Citons la règle donnée dans un livre d'arithmétique destiné aux établissements secondaires de jeunes filles : *Leçons d'arithmétique et notions de géométrie*, par Mme Salomon (Vuibert, 1920).

« On sépare le nombre en tranches de deux chiffres à partir de la droite, la dernière tranche à gauche pouvant n'avoir qu'un chiffre ; on extrait à une unité près par défaut la racine carrée de la première tranche à gauche ; à la droite du reste, on abaisse la tranche suivante ; on en sépare le premier chiffre à droite par un point et on divise la partie séparée à gauche par le double de la racine ; on place le chiffre trouvé à côté du double de la racine et on multiplie le nombre ainsi formé par ce chiffre, en même temps qu'on retranche le produit du reste tout entier. S'il convient, on le place à côté du premier chiffre trouvé. À la droite du reste, on abaisse la tranche suivante et on opère comme précédemment. Si l'une des divisions partielles donne zéro pour quotient, on met un zéro à la racine et on abaisse la tranche suivante. Si l'un des restes est plus grand que le double de la racine, le chiffre écrit à la racine est trop faible, il faut l'augmenter d'une unité. »

Nombre	6·25	25	*Racine*
	4	45 × 5	
	22·5		
	22 5		
	0		

Nombre	9·42·62	307	*Racine*
	9	607 × 7	
	4·26·2		
	4 24 9		
Reste	1 3		

Voici la technique similaire à celle de la racine carrée, extraite d'un traité du début du XIXe siècle. L'ouvrage était destiné aux commerçants (*Traité élémentaire d'arithmétique à l'usage des commerçans de tout genre*, par M. J. Courtot, rédigé par M. F. de la Combe, professeur de mathématiques au collège de Tonnerre, 1819).

« On commence par disposer ce nombre comme pour faire une division, après quoi on le divise en tranches de trois chiffres, en allant de droite à gauche, on cherche le plus grand cube contenu dans la première tranche à gauche (101), on en extrait la racine que l'on écrit à la droite du nombre proposé (4), on élève cette racine au cube (64), et l'on retranche cette racine du nombre sur lequel on vient d'opérer, à côté du reste on abaisse la tranche suivante (847), on sépare les deux derniers chiffres, puis on divise les chiffres restant à gauche par le triple carré de la racine déjà trouvée ($3 \times 4^2 = 48$), ce qui donne le second chiffre de la racine (6) qu'on écrit à la droite du premier ; pour vérifier cette opération et connaître en même temps le reste, s'il y en a un, on élève les deux chiffres de la racine au cube ($46^3 = 97\ 336$), et l'on retranche ce cube des deux premières tranches sur lesquelles on vient d'opérer : à côté du reste on abaisse la tranche suivante (563), on sépare de même les deux derniers chiffres, puis on divise encore les chiffres restants par le triple carré de la racine déjà trouvée ($3 \times 46^2 = 6\ 348$), ce qui donne le troisième chiffre de la racine (7) qu'on écrit à la droite du second ; pour vérifier cette racine, on élève ces trois chiffres au cube ($467^3 = 101\ 847\ 563$), et l'on retranche le produit des trois premières tranches, et ainsi de suite jusqu'à

101,847,563	467
64	48
378,47	6348
97336	
45115,63	
101847563	
000000000	

Le calcul d'une racine cubique (le nombre de départ est un cube parfait).

12,000,000	2,28
8	12
40,00	1452
10648	
13520,00	
11852352	
147648	

La même méthode, appliquée à un nombre qui n'est pas un cube parfait, poussée jusqu'à la deuxième décimale.

ce que l'on ait abaissé tous les chiffres du nombre proposé.

Si après avoir abaissé la dernière tranche et fait la dernière soustraction, il ne reste rien, c'est une preuve que le nombre proposé est un cube parfait, si au contraire il y a un reste, la racine est incommensurable, à moins que ce reste ne soit égal au triple carré de la racine plus le triple de cette racine, plus l'unité, car dans ce cas, la racine serait trop faible, et il faudrait l'augmenter convenablement. »

Cette méthode, analogue à celle de l'extraction de la racine carrée, est basée sur l'identité :

$$(a + b)^3 = a^3 + 3a^2b + 3ab^2 + b^3.$$

Si le nombre n'est pas un cube parfait, on lui soustrait le plus grand cube possible, a^3, puis on divise le reste par $3a^2$ (le terme $3a^2b$ étant « grand » devant $3ab^2$ que l'on néglige, d'où la nécessité de préciser : « *à moins que ce reste ne soit égal au triple carré de la racine plus le triple de cette racine, plus l'unité, car dans ce cas, la racine serait trop faible* »).

M. C. et M. R.

par Hervé Lehning

Des algorithmes pour créer le hasard

Les algorithmes sont par nature déterministes. Pour une valeur donnée en entrée, ils fournissent toujours la même valeur en sortie. Pourtant, certains peuvent créer du hasard, ou plutôt le simuler. Un exemple très répandu est celui des générateurs de nombres pseudo-aléatoires. De tels algorithmes peuvent être utilisés pour calculer des intégrales multiples.

Pour simuler un phénomène fondé sur le hasard, on utilise des suites de nombres aléatoires c'est-à-dire fournis par le hasard. Les langages de programmation et les logiciels de calcul comme les tableurs offrent cette possibilité. Excel utilise une fonction nommée ALEA (pour *aléatoire*) et la plupart des langages de programmation une fonction nommée RAND (pour *random*).

Examinons la fonction proposée par Excel. Vous tapez =ALEA() dans une case et il renvoie un nombre entre 0 et 1 comportant huit chiffres après la virgule, ce qui revient, une fois la partie entière gommée, à un nombre compris entre 0 et 10^8. On obtient ainsi consécutivement des nombres ayant tout l'air d'être le fruit du hasard.

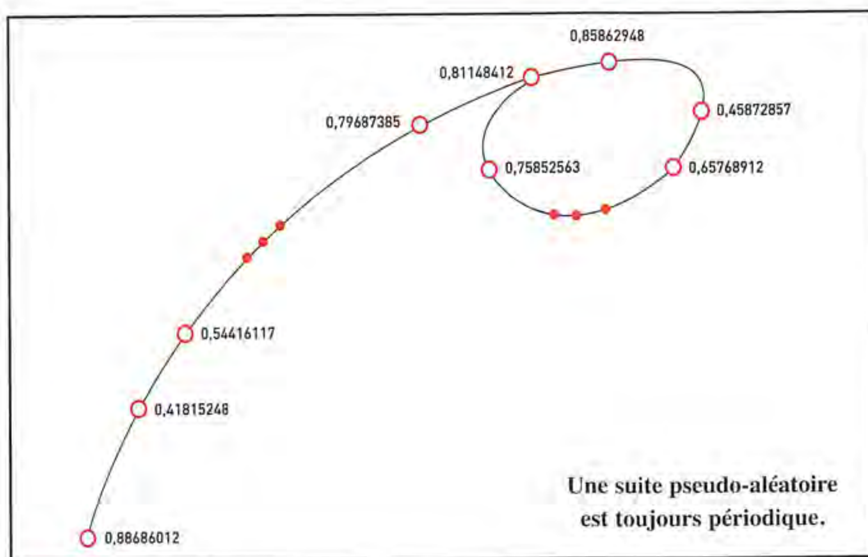
Le hasard est la mesure de notre ignorance.

Par exemple, voici une suite obtenue avec une ligne de dix instructions =ALEA() :

0,88686012	**0,41815248**
0,54416117	**0,59091058**
0,46098967	**0,73508295**
0,30939046	**0,79687385**
0,81148412	**0,85862948**

Suite de dix nombres obtenus avec la fonction ALEA.

D'où proviennent ces nombres ? Pas du hasard car rien n'est moins aléatoire que le fonctionnement d'un ordinateur. Même si l'utilisateur, et souvent même le concepteur, peut être surpris par certaines réactions de la machine, chaque opération suit un algorithme précis, son résultat est prédéterminé. Pas de hasard ici !



Une suite pseudo-aléatoire est toujours périodique.

Suite pseudo-aléatoire

Cependant, les suites produites respectent certaines règles des suites *véritablement* aléatoires. Tout d'abord, quand on augmente le nombre de termes de la suite, leur moyenne tend vers la moyenne attendue, 0,5 dans notre exemple. Bien entendu, il faut utiliser un grand nombre de termes pour la retrouver. La rencontrer trop tôt serait suspect…

Une suite *pseudo-aléatoire* est donc une suite de nombres entiers x_0, x_1, x_2, … prenant ses valeurs dans un ensemble M = {0, 1, 2, ..., $m-1$}. Le terme x_n ($n > 0$) est le résultat d'un calcul sur le terme précédent. Le premier terme x_0 est appelé le germe de la suite. Il la détermine donc entièrement ! L'ensemble M (les valeurs possibles de la suite) ayant m éléments, il est impossible que $m + 1$ termes de la suite soient distincts.

Ainsi, parmi les $m + 1$ premiers, deux sont forcément égaux. La règle de formation des termes de la suite implique qu'ils se répètent alors à l'identique ! Rien de moins hasardeux que cela ! Bien entendu, pour que ce phénomène ne soit pas gênant, on prend de très grandes valeurs de m. On a vu que le générateur d'Excel utilise $m = 10^8$. Les suites générées sont donc périodiques mais la période peut être très grande.
D'autre part, comme la suite dépend entièrement de son germe, si on ne le change pas, on retrouve toujours la même. Pour cette raison, les générateurs de nombres aléatoires introduisent à ce niveau un soupçon de vrai hasard. Par exemple, ils peuvent prendre comme germe les décimales de l'heure exacte où le programme a été démarré.

Les qualités d'un générateur idéal

Qu'attend-on d'un générateur de nombres pseudo-aléatoires ? La réponse à cette question conditionne la méthode à choisir car, dans ce domaine comme ailleurs, tout est affaire de compromis. Le jeu de dés utilise un générateur de nombres aléatoires, la simulation d'une centrale nucléaire

Simulation : la marche de l'ivrogne

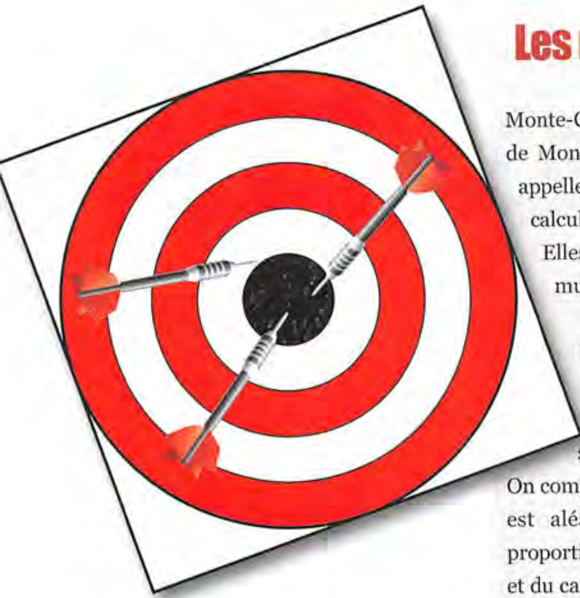
Pour déterminer la probabilité d'un événement, une technique est de le simuler, ce qui nécessite d'utiliser un générateur de nombres pseudo-aléatoires.

Prenons l'exemple de la marche de l'ivrogne :
Un ivrogne sort de son bar préféré pour rentrer chez lui à trente mètres. Il n'a qu'un long trottoir rectiligne à suivre mais il est dans un tel état qu'il oscille aléatoirement à droite et à gauche à chaque pas sauf quand il est contre le mur. Il fait des pas d'un mètre de long et oscille de cinquante centimètres sur le côté à chaque pas. Le trottoir fait trois mètres de large.

Quelle est la probabilité qu'il tombe dans le caniveau ?
Le problème peut se résoudre par la théorie. Vous pouvez également simuler la marche de l'ivrogne. La difficulté est qu'il n'est pas facile de faire quoi que ce soit vraiment au hasard ! Si vous réalisez l'expérience physiquement, ne vous saoulez pas. D'abord, c'est mauvais pour la santé, ensuite il est plus simple d'utiliser une pièce de monnaie : pile, vous oscillez à droite, face, vous le faites à gauche. Vous la réalisez un nombre N de fois jusqu'à arriver chez vous ou tomber dans le caniveau. Vous comptez le nombre M de fois où vous finissez par terre. D'après la loi des grands nombres, si N est grand, la probabilité cherchée est proche de M/N. Il est fastidieux de réaliser toutes ces expériences physiquement même en utilisant un grand nombre d'ivrognes. Il est possible de réaliser une telle simulation en utilisant un tableur comme Excel. On rentre dans la première cellule : **=MAX(0;SI(ALEA()<0,5;-1;1))** puis dans la deuxième : **=MAX(0;SOMME(A1;SI(ALEA()<0,5;-1;1)))** ce que l'on recopie ensuite jusqu'à la cellule A30. Cela correspond à la marche d'un ivrogne. Il tombe dans le caniveau si le nombre 6 figure dans une des cellules A1 à A30 ce que l'on teste en écrivant dans la cellule A31 : **=SI(NB.SI(A1:A30;6)>0;1;0)**
Plus exactement, si notre ivrogne virtuel tombe dans le caniveau, la cellule A31 contient le nombre 1, sinon elle contient le nombre 0. Il nous reste à utiliser un grand nombre d'ivrognes virtuels en recopiant cette première colonne dans les suivantes. La moyenne des dernières lignes donne une approximation de la probabilité de tomber dans le caniveau. Pour affiner cette approximation, il est nécessaire de faire recalculer plusieurs fois la feuille utilisée. On trouve finalement que l'ivrogne à un peu moins d'une chance sur deux de finir dans le caniveau.Il est ainsi possible de simuler tout phénomène reposant sur le hasard.

aussi, mais les contraintes ne sont pas les mêmes. En général on privilégie trois critères.
La vitesse : le calcul du nombre pseudo-aléatoire suivant doit être rapide. La simulation demande de générer parfois des millions de nombres.
La simplicité : une méthode très compliquée est très difficile à programmer et à tester.

La fiabilité de production des nombres : le programme ne doit pas « planter » quand il arrive sur certains nombres !
Les nombres produits ne doivent pas faire apparaître de suites logiques. Chaque nombre doit avoir à tout moment et quel que soit l'*historique* autant chance de sortir que les autres. Plus généralement, les théorèmes connus de probabilité doivent être vérifiés.
Dans la mesure où la suite proposée respecte toutes les lois connues des probabilités, on peut espérer qu'elle se comporte correctement dans les applications de simulation. Il ne s'agit pas d'obtenir vraiment du hasard. La question relève d'ailleurs plus de la philosophie que des mathématiques.



Les méthodes de Monte-Carlo

Monte-Carlo est connu pour l'utilisation que fait la principauté de Monaco du hasard pour s'enrichir. En son hommage, on appelle méthodes de Monte-Carlo toutes les méthodes de calcul fondées sur l'utilisation du hasard.

Elles sont couramment utilisées pour calculer des intégrales multiples.

Pour en donner une idée, voyons comment calculer π à l'aide... du hasard !

On tire de manière aléatoire des fléchettes (ou autres) sur une cible carrée dans laquelle on a inscrit un cercle. On compte ensuite le nombre d'impacts dans le disque. Si le tir est aléatoire, le nombre de coups dans le disque est proportionnel au rapport des surface du disque et du carré c'est-à-dire à $\pi / 4$.

Cette méthode est difficile à appliquer avec un tireur humain mais on peut l'utiliser en simulant le tir à l'aide d'un générateur de nombres aléatoires. C'est moins dangereux et plus sûr ! L'expérience montre que la convergence est assez lente.

Connaissant les théorèmes qui doivent être vérifiés, on en déduit une batterie de tests qu'une suite doit passer pour être considérée comme aléatoire ou plutôt pseudo-aléatoire puisque le hasard véritable est impossible à recréer. D'ailleurs, existe-t-il ?

Un exemple de générateur « historique »

Bien qu'on utilise de nos jours des suites plus sophistiquées, on a longtemps utilisé $m = 2\,147\,483\,647$ ($= 2^{32} - 1$) et après avoir choisi un germe x en utilisant l'horloge de l'ordinateur, on calculait le reste de $16\,807\,x$ dans la division par 2 147 483 647 pour obtenir le suivant puis on recommençait.

Une telle méthode est très facile à programmer et fournit des suites du type :

321 635 411,	**510 013 178,**
1 184 247 469,	**768 771 087,**
1 474 038 857,	**799 717 807,**
1 904 519 323,	**1 012 503 126,**
479 619 854,	**146 4758 987...**

Cette suite passe tous les tests usuels et peut donc être considérée comme acceptable. Nous vous la recommandons si vous voulez fabriquer un générateur vous-même. Vous pouvez ainsi voir avec cet exemple que le hasard programmé n'a rien en commun avec le hasard. Si vous l'utilisez pour jouer aux dés avec des amis, vous pouvez toujours trouver un nombre à partir du précédent à condition d'être capable de calculer très vite et de tête. Mais ne dit-on pas que le hasard est la mesure de notre ignorance ? Après tout, le lancer de dés est aussi contrôlé par des équations connues. Si vous connaissez les conditions exactes du lancer, vous devez connaître le résultat.

H. L.

par Hervé Lehning

Les fractions égyptiennes

Les scribes égyptiens n'utilisaient que des fractions dont le numérateur est égal à 1. Cette contrainte est à l'origine de jolis problèmes d'arithmétique et d'un algorithme ingénieux.

Au temps des pyramides, les anciens Égyptiens utilisaient des fractions telles que 1/2, 1/3, 1/4, *etc.* Toutes représentaient le partage de l'unité en plusieurs parties égales. Pour les autres, ils additionnaient les fractions de numérateurs égaux à 1, par exemple : $5/6 = 1/2 + 1/3$. De nos jours, on appelle *fraction égyptienne* toute fraction de numérateur égal à 1. On s'intéresse aux décompositions des nombres rationnels p/q comme sommes de telles fractions où les dénominateurs sont des entiers naturels tous distincts.

L'art de décomposer

Si le dénominateur d'un nombre est pratique, nous pouvons le décomposer en fractions égyptiennes.

Commençons par essayer de développer 1 à la mode égyptienne. La décomposition la plus immédiate est $1 = 1/2 + 1/2$. Malheureusement, ce développement ne convient pas, puisqu'il contient deux fois le même dénominateur. Une idée simple permet de poursuivre : $1/2 = 1/3 + 1/(2.3)$, donc $1 = 1/2 + 1/3 + 1/6$.

Cette idée est générale. Par exemple pour 3/7, on remarque que :

$3/7 = 1/7 + 1/7 + 1/7$.

Puis : $1/7 = 1/8 + 1/(7.8)$.

Donc : $3/7 = 1/7 + 1/8 + 1/56 + 1/8 + 1/56$.

En recommençant, nous obtenons :

$$3/7 = 1/7 + 1/8 + 1/56 + 1/9 + 1/72 + 1/57 + 1/3192.$$

Nous pouvons ordonner les fractions égyptiennes suivant les dénominateurs croissants :

$$3/7 = 1/7 + 1/8 + 1/9 + 1/56 + 1/57 + 1/72 + 1/3192.$$

Tout nombre rationnel peut être décomposé ainsi. Cette méthode est simple, mais a le défaut de ne pas économiser le nombre de fractions égyptiennes utilisées. Ainsi, nous pouvons trouver une décomposition de 3/7 en trois fractions seulement :

$3/7 = 1/3 + 1/15 + 1/35$.

Un algorithme efficace

Un algorithme plus efficace pour décomposer un nombre rationnel x consiste à considérer le plus petit entier

n supérieur à $1/x$, puis à recommencer sur la différence $x - 1/n$.
Sur notre exemple, cela donne successivement :

- le plus petit entier supérieur à 7/3 est 3, **3/7 – 1/3 = 2/21**,
- le plus petit entier supérieur à 21/2 est 11, **2/21 – 1/11 = 1/231**.

Nous obtenons donc la décomposition :
3/7 = 1/3 + 1/11 + 1/231.
Cet algorithme permet de décomposer un nombre rationnel p/q en au plus p fractions égyptiennes (voir l'encadré *Une descente infinie*).

Une descente infinie

Considérons un nombre x. Posons $x = x_1$ et n_1 le plus petit entier supérieur à $1/x_1$. Définissons ensuite x_2 par $x = x_2 + 1/n_1$, et n_2 à partir de x_2, *etc.* De façon générale, nous obtenons : $x = x_p + 1/n_1 + 1/n_2 + ... + 1/n_{p-1}$. Écrivons x_p sous la forme d'une fraction irréductible : $x_p = r_p/s_p$. Nous avons donc : $r_{p+1}/s_{p+1} = r_p/s_p - 1/n_p = (r_p n_p - s_p)/s_p n_p$.
L'inégalité : $n_{p-1} < 1/x_p \leq n_p$ montre que $0 \leq r_p n_p - s_p < r_p$.
La fraction étant irréductible, nous en déduisons que $r_p > r_{p+1}$. Comme ces nombres sont entiers, il existe p inférieur au numérateur de x tel que $r_{p+1} = 1$ donc : $x = 1/n_1 + 1/n_2 + ... + 1/n_p$.

Des nombres pratiques

Lorsque le numérateur p est la somme de diviseurs distincts du dénominateur q, le nombre p/q se décompose en fractions égyptiennes de dénominateurs inférieurs à q. Par exemple **9 = 5 + 4**, donc :
9/20 = (5 + 4)/20 = 1/4 + 1/5.
Il est en fait facile de décomposer en fractions égyptiennes les rationnels dont le dénominateur q vérifie la propriété suivante :
tout $p < q$ est somme de diviseurs distincts de q.
La liste de ces nombres, appelés « nombres pratiques », commence ainsi : 1, 2, 4, 6, 8, 12, 16, 18, 20, *etc.*
Si le dénominateur d'un nombre est pratique, nous pouvons le décomposer en fractions égyptiennes en écrivant son numérateur comme somme de diviseurs distincts de son dénominateur.
Par exemple, pour décomposer 17/144, cherchons les diviseurs de 144 : 1, 2, 3, 4, 6, 9, 12, 16, 18, 24, 36, 48, 72, 144.
Le nombre 17 s'écrit 16 + 1, donc :
17/144 = (16 + 1)/144 = 1/9 + 1/144.

Des multiples pratiques

Si le dénominateur d'un nombre rationnel n'est pas pratique, nous pouvons nous y ramener s'il possède un multiple pratique. Reprenons le cas de 3/7 dans cette optique. Le nombre 28 étant pratique, on cherche à écrire 12 comme somme des diviseurs de 28 (1, 2, 4, 7, 14, 28). On trouve **12 = 7 + 4 + 1**, d'où :
3/7 = 12/28 = (7 + 4 + 1)/28 = 1/4 + 1/7 + 1/28.
Pour construire de tels multiples pratiques, nous disposons du théorème suivant :
Si q est un nombre et n un nombre pratique premier avec q tel que $q < 2n$, alors qn est pratique.
Considérons par exemple le nombre rationnel 5/23. Le dénominateur 23 est premier avec 12 qui est pratique, donc 23.12 est pratique. Les diviseurs de 276 sont 1, 2, 3, 4, 6, 12, 23, 46, 69, 92, 138 et 276, et **60 = 46 + 12 + 2**, donc :
5/23 = (5.12)/(23.12) = (46 + 12 + 2)/276
= 1/6 + 1/23 + 1/138.

H. L.

Paul Erdös.

Décomposition en trois

Les exemples étudiés montrent des décompositions en *au plus* trois fractions égyptiennes. D'après l'algorithme décrit dans *Un algorithme efficace*, c'est le cas pour les nombres rationnels de la forme p/q où $p = 1$, 2 ou 3. Cela semble également le cas si où $p = 4$ ou 5. Cependant, personne n'a su prouver ces conjectures. Elles sont dues à Erdös pour la première, et à Sierpinski pour la seconde.

par Hervé Lehning

Les mariages stables existent

La métaphore des mariages stables recouvre tous les problèmes d'affectations ou d'allocations de ressources. Un algorithme permet de réaliser de tels appariements. Sa simplicité fait qu'il est utilisé pour l'affectation des étudiants en classe préparatoire.

Le *problème des mariages stables* désigne de façon imagée le problème des allocations de ressources, ou des affectations de personnels. Par exemple, comment affecter des étudiants dans des établissements ? Des travailleurs sur des postes ? Des clients à des fournisseurs ? Ou, dans un réseau de communication, des émetteurs à des receveurs ?

La métaphore des mariages stables

Utilisons la métaphore du mariage pour examiner le problème de l'affectation d'étudiants dans des établissements. Les établissements y jouent le rôle des « hommes » (chacun en représentant autant qu'il a de places disponibles), et les candidats, celui des « femmes ». Le but est de réaliser des « mariages stables » dans le sens qu'on ne puisse trouver un homme et une femme qui se préfèrent l'un l'autre à leurs conjoints respectifs. Cette contrainte, légitime dans le cas de mariages, l'est également du point de vue de notre problème d'affectation. Elle implique qu'hommes et femmes désignent au préalable les personnes du sexe opposé qu'ils (elles) jugent acceptables et les classent, sans *ex-aequo*. Dans ces conditions, David Gale (1921–2008) et Lloyd Shapley (né en 1923) ont montré qu'il existe au moins une manière stable de les marier. Bien sûr, il peut exister des « laissés pour compte », mais on démontre que ce sont les mêmes dans toutes les solutions stables.

Les femmes ne pourraient pas être plus mal servies !

L'algorithme de Gale et Shapley

Pour construire une solution stable,

Lloyd Shapley a proposé un algorithme fondé sur une idée traditionnelle : « L'homme propose, la femme dispose. » Il applique cette logique en procédant par étapes, chacune se scindant en deux parties. Dans la première, les hommes font des propositions de mariage. Les femmes les rejettent ou les acceptent. À la première étape, chaque homme propose le mariage à la femme qu'il préfère sans tenir compte d'éventuels concurrents. À ce moment, chaque femme accepte la proposition qu'elle préfère parmi celles qu'elle a reçues. Elle se « fiance » alors à celui-ci en rompant éventuellement ses fiançailles précédentes. Les femmes ne recevant aucune proposition de mariage attendent l'étape suivante. Quand elle arrive, les hommes déjà fiancés ne font rien, les autres font une proposition aux femmes qui ne les ont pas déjà refusés. Le processus se poursuit ainsi tant que cela est possible. Cette simple description montre que, dans le pire des cas, son temps est proportionnel au produit du nombre des hommes par celui des femmes. L'algorithme produit des mariages stables car, si un homme préfère une autre femme à son épouse, celle-ci l'a rejeté au cours de la procédure. Voyons un exemple. Dans le tableau suivant, les majuscules représentent les hommes, et les minuscules, les femmes.

Les propriétés de l'algorithme de Gale–Shapley

L'algorithme de Gale–Shapley s'achève après un certain nombre d'étapes car, à chacune d'entre elles, au moins un homme libre disparaît. Nous obtenons donc un certain nombre de mariages. Supposons qu'il existe alors un homme *H* préférant une femme *F* à son épouse. Celle-ci l'a forcément rejeté au cours de l'algorithme puisqu'il a invité toutes les femmes dans l'ordre décroissant de ses préférences. Elle l'a rejeté au profit d'un autre homme, soit *H* ' qu'elle préfère donc à *H*. Son époux à l'issu de l'algorithme est soit *H* ', soit un autre qu'elle lui préfère encore car les prétendants qui se sont succédé auprès d'elle allaient dans l'ordre croissant de ses préférences. Nonobstant la royale sagesse de François Ier : *« Souvent femme varie, bien fol qui s'y fie »*, les auteurs de l'algorithme en déduisent que la situation est impossible et donc que les mariages sont tous stables.
Bien entendu, il ne s'agit pas de la seule série de mariages stables possibles mais, pour les hommes, elle est optimale dans le sens que, avec cet algorithme, tout homme est marié avec la femme qu'il préfère parmi toutes les séries de mariages stables envisageables. À l'inverse, les femmes ne peuvent être plus mal loties. Il est logique que l'on ne puisse satisfaire tout le monde !

A	*cbad*	*a*	ABC
B	*edb*	*b*	DB
C	*bace*	*c*	CAD
D	*dceba*	*d*	ABDC
		e	ADCB

Tableaux des préférences des hommes et des femmes

Avec ces données, l'algorithme de Gale et Shapley donne la solution stable : A*c*, B*e*, C*a*, D*d*. En inversant les rôles des hommes et des femmes, il donne : A*a*, B*d*, C*c*, D*e*. Les solutions sont différentes. Dans ce cas particulier, il existe également deux autres solutions

Internet dans le TGV

Examinons le problème de l'accès Internet à haut débit dans un train à grande vitesse. Chaque point d'accès couvre un disque de cent mètres de rayon, environ. Pour un déplacement lent, on peut imaginer de les répartir le long des voies et en changer régulièrement. Avec un train à grande vitesse, cette solution est irréalisable car, lancé à trois cents kilomètres heure, il traverse la zone couverte en moins de trois secondes. En gardant ces relais, une idée est de découper les fichiers en petits paquets et d'apparier les relais aux antennes pour leur transfert. Cette allocation correspond à un problème de mariages stables.

stables : A*a*, B*e*, C*c*, D*d* et A*c*, B*d*, C*a*, D*e*. Dans tous les cas, *b* est laissée pour compte.

De même, chaque homme épouse la femme qu'il préfère parmi toutes les femmes possibles, dans des configurations stables. Les hommes ne peuvent donc être mieux servis. On démontre également que les femmes ne peuvent l'être plus mal ! En donnant l'initiative à ces dernières, nous obtenons d'ailleurs d'autres mariages, optimaux pour elles cette fois-ci. Notons pour finir que mentir sur leurs préférences peut leur être utile. En voici un exemple :

A	*dbac*		*a*	CBDA
B	*dacb*		*b*	BADC
C	*dacb*		*c*	BCAD
D	*acdb*		*d*	DCBD → DABC

Une manipulation réussie : d transforme ses préférences pour obtenir un meilleur résultat.

Avec les premières préférences, la solution est A*b*, B*a*, C*d*, D*c* mais, en transformant ses préférences, *d* obtient A*b*, B*c*, C*a*, D*d*.

Utilisation effective

En France, cet algorithme est utilisé pour l'affectation des étudiants en classes préparatoires et pour celle des professeurs en universités. Il est difficile d'envisager une stratégie de manipulation applicable par un étudiant dans le cadre de l'affectation en classe préparatoire. Pour cela, il faudrait au moins qu'il connaisse tous les autres classements. En revanche, une telle stratégie est envisageable pour les candidats à des postes de professeurs en universités. En effet, celles-ci publient leurs préférences avant que les candidats donnent les leurs. Des ententes entre certains candidats et des manipulations sont ainsi possibles. Pour éviter cet aspect, il suffirait d'inverser l'algorithme en commençant par les choix des « femmes ».

H. L.

Quand la programmation prend le relais du tableur

Jusqu'à présent, la production d'échantillons de simulation se faisait généralement avec des tableurs, en particulier dans le cadre scolaire. Mais, bien que parfaitement efficace, l'usage d'un tableur limitait la taille de l'échantillonnage à des proportions raisonnablement représentables. La programmation permet d'accéder plus rapidement à des échantillons de taille importante. La loi des grands nombres aidant, les fréquences obtenues seront d'autant plus proches des probabilités cherchées. Grâce à l'intégration de la programmation dans le cours de mathématiques, les lycéens aussi peuvent désormais recréer le hasard à l'aide d'un programme.

Pour illustrer cette différence, voici deux approches de la simulation du lancer d'un dé équilibré avec pour objectif le relevé du nombre d'occurrences du 6. Dans le premier cas, la simulation est réalisée sur tableur avec un échantillonnage de 50 000 lancers, dans le second cas à l'aide d'un programme Python et 10 000 000 de lancers.

Obtention d'échantillons de taille 50 000

Simulation de l'expérience avec un tableur.

Obtention d'échantillons de taille 10^7 (programme Python)

```
>>> from random import randrange
>>> def tirage():
        i=0
        k=0
        while i!=10**7:
            if randrange(1,7)==6:
                k=k+1
            i=i+1
    return i,k
```

Exécution du programme :

```
>>> tirage()
(10000000, 1666544)
```

Bob Frankston et Dan Bricklin

Robert Frankston (informaticien américain né en 1949) et Daniel Bricklin (informaticien américain né en 1951) sont les pères fondateurs du tableur. Un *tableur* (*spreadsheet* en anglais) n'est rien d'autre qu'un programme informatique capable de gérer les feuilles de calculs. De telles feuilles de calculs sont simplement des tables contenant des informations (mathématiquement, on peut les modéliser par des matrices). Elles sont employées essentiellement en comptabilité. La légende raconte que Dan Bricklin a eu l'idée du tableur en assistant à un cours, à l'université. L'enseignant, après avoir dessiné au tableau une grande table de calculs, y trouve une erreur et doit effacer et recalculer une grande partie des cases. Or, ce processus peut être automatisé à l'aide d'un ordinateur…

par Hervé Lehning

Sous l'ordinateur, les booléens

Comment calculent les ordinateurs ? À partir de circuits logiques. De nos jours, ils sont réalisés avec du matériel électronique mais on pourrait les concevoir autrement. L'essentiel est dans les fonctions logiques.

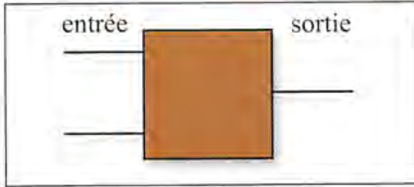
Avant vant de nous lancer dans la conception d'un ordinateur, dénombrons les ingrédients nécessaires à sa construction. À la base, il faut pouvoir véhiculer deux informations : « faux » et « vrai », « 0 » ou « 1 », « ouvert » ou « fermé » (…). L'électricité nous fournit une solution : un courant passe ou ne passe pas. En pratique, on utilise un courant de 0 (respectivement 5 volts) pour représenter le « 0 » (respectivement le « 1 ») logique. Notez que l'on pourrait faire l'inverse sans modification fondamentale. On pourrait également imaginer un autre modèle utilisant, par exemple, un tuyau où de l'eau passerait ou non. Cela dit, pour réaliser un ordinateur, nous devons pouvoir réaliser toutes les fonctions logiques imaginables à partir d'un petit nombre de « briques » élémentaires. Pourquoi ? Tout simplement parce que, grâce à elles, nous pouvons calculer en base deux, ce qui est non seulement nécessaire, mais aussi presque suffisant pour construire un processeur d'ordinateur. Il suffit en plus de savoir recopier des cases mémoire.

Pourquoi pas un ordinateur à eau ?

Les fonctions logiques

Qu'appelons-nous fonction logique ? Les fonctions logiques à une seule variable sont les fonctions prenant en entrée une information logique (« vrai » ou « faux », le courant passe ou non, 1 ou 0…) et donnant en sortie une autre information logique. Une fonction logique à deux variables prendra de même deux informations logiques en entrée et en donnera une seule en sortie, *etc*.

De façon pratique, on représente une telle fonction logique f de la façon suivante :



Fonction logique à deux variables.

Les fonctions logiques les plus simples correspondent aux connecteurs « et » et « ou » que l'on peut représenter par les tables :

1er argument / 2nd argument	Vrai	Faux
Vrai	Vrai	Vrai
Faux	Vrai	Faux

Connecteur « ou » : un seul argument « vrai » provoque « vrai » comme résultat.

1er argument / 2nd argument	Vrai	Faux
Vrai	Vrai	Faux
Faux	Faux	Faux

Connecteur « et » : un seul argument « faux » provoque « faux » comme résultat.

De façon générale, toute fonction logique correspond à une telle table.

Intérêt des fonctions logiques

Pour montrer l'intérêt de considérer des fonctions logiques, supposons que nous voulions additionner deux nombres à 16 chiffres binaires (on dit *à 16 bits* en informatique). Pour cela, il suffit de considérer les fonctions logiques à 32 variables (les bits des deux nombres) donnant chacune un des chiffres du résultat. Il s'agit bien d'une fonction logique : à 32 entrées valant 0 ou 1, elle fait correspondre une valeur du même type. Le problème est le même pour la multiplication ou toute autre opération. Ainsi, en théorie, nous pouvons fabriquer un calculateur binaire si nous pouvons réaliser toutes les fonctions logiques. Leur intérêt réside dans cette propriété.

En guise d'exemple, considérons la fonction logique de deux variables f représentée par :

(0 ; 0)	(0 ; 1)	(1 ; 0)	(1 ; 1)
↓	↓	↓	↓
1	0	0	1

ce qui signifie qu'à (0 ; 0), f associe 1, et ainsi de suite. En l'interprétant avec « vrai » pour 1 et « faux » pour 0 et en le présentant sous forme de tableau, on obtient :

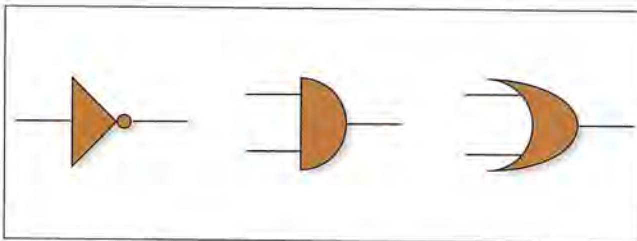
1er argument / 2nd argument	Vrai	Faux
Vrai	Vrai	Faux
Faux	Faux	Vrai

On remarque qu'il s'agit de la table de vérité de la proposition logique :
(p et q) ou ((non p) et (non q)).
Pour le démontrer, il suffit d'essayer les quatre cas : (p vrai ; q vrai), (p vrai ; q faux), (p faux ; q vrai) et (p faux ; q faux). Par exemple, dans le premier cas : (p et q) est vrai ; ((non p) et (non q)) est faux, donc ((p et q) ou ((non p) et (non q))) est vrai.
Ce résultat est vrai pour toute fonction logique. Ainsi, toutes les fonctions logiques peuvent être représentées grâce aux trois connecteurs logiques « non », « ou » et « et ». Pour créer un ordinateur, il suffit donc de produire ces trois connecteurs. On pourrait même se contenter de deux connecteurs (« et » et « non » par exemple) puisque :
p ou q équivaut à
non ((non p) et (non q)).
On peut donner une démonstration générale et simple de ce résultat sur

les fonctions logiques (voir l'encadré Fonctions et propositions logiques). Sur l'exemple ci-dessus, l'inconvénient est de multiplier les connecteurs « non », ce qui a des inconvénients pratiques. L'usage est donc d'utiliser trois connecteurs, et même quelques-uns de plus.

Les portes logiques

Les réalisations électroniques des connecteurs logiques sont appelées « portes logiques ». Nous ne rentrerons pas ici dans le détail de leur fabrication. Nous en donnons seulement les symboles utilisés aux États-Unis, les symboles européens n'étant pratiquement plus utilisés, même en France.



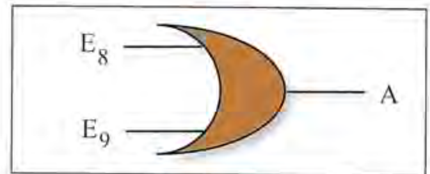
Symboles des portes logiques « non », « et » et « ou ».

On utilise aussi couramment les portes logiques « non et », « non ou » et « ou exclusif » mais nous les éviterons dans cet article.

La représentation des fonctions logiques par des propositions logiques permet de construire des circuits les représentant grâce aux trois portes logiques précédentes. Voyons par exemple comment coder un *chiffre* décimal en binaire. Le circuit que nous nous proposons de définir comporte dix variables d'entrée : E_0, E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8 et E_9 et quatre variables de sortie : A, B, C et D. Un chiffre i est représenté en mettant l'entrée E_i à 1 et les autres à 0. La sortie correspond à l'écriture binaire de ce chiffre. Le tableau suivant donne les quatre fonctions logiques demandées :

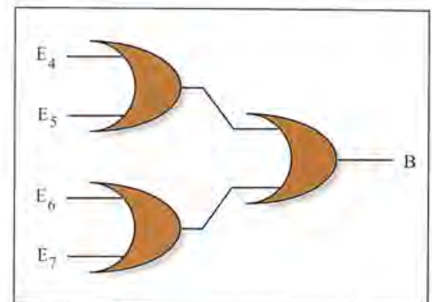
Entrée	A	B	C	D
E_0	0	0	0	0
E_1	0	0	0	1
E_2	0	0	1	0
E_3	0	0	1	1
E_4	0	1	0	0
E_5	0	1	0	1
E_6	0	1	1	0
E_7	0	1	1	1
E_8	1	0	0	0
E_9	1	0	0	1

où la première colonne signifie $E_0 = 1$, $E_1 = 1$, ... ou $E_9 = 1$ sachant que, quand l'un de ces signaux vaut 1, tous les autres valent 0. Nous pouvons donc fabriquer quatre circuits logiques A, B, C et D réalisant ces quatre fonctions. Pour le premier, A = 1 si et seulement si $E_8 = 1$ ou $E_9 = 1$. Il suffit donc d'utiliser une porte logique « ou » :



Circuit logique représentant A.

Le second fait intervenir E_4, E_5, E_6 et E_7 de la même façon, d'où :



Circuit logique représentant B.

On obtient de même les deux autres. On remarque que E_0 n'intervient pas, ce qui est assez logique puisqu'il s'agit d'une information redondante

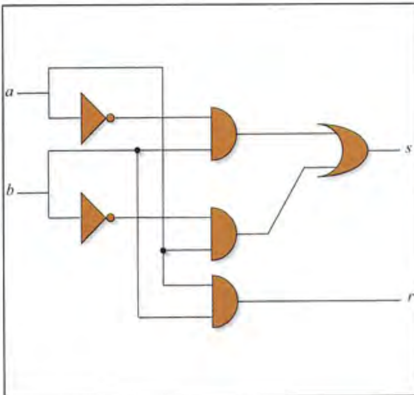
servant à un autre niveau. En réunissant les quatre circuits trouvés, on obtient un circuit logique permettant le codage en binaire.

Circuits arithmétiques

De même, nous pouvons construire des circuits permettant d'additionner et de multiplier des nombres écrits en binaire. Le plus simple à réaliser est l'additionneur à un bit. Notre circuit doit ainsi comporter deux entrées (les bits à additionner a et b) et deux sorties (le résultat s, et une retenue r). Les deux fonctions à écrire sont données par le tableau :

a	b	r	s
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

La retenue correspond donc à un « et » logique (a et b) alors que la somme elle-même est un peu plus compliquée, il s'agit en fait d'un « ou » exclusif qui peut s'écrire ((non a) et b) ou (a et (non b)). Vous pouvez vérifier cette égalité en dressant la table de vérité de cette proposition et en la comparant avec la table ci-dessus. On en déduit le circuit ci-dessous.



Additionneur un bit (avec retenue). Les croisements entre les fils correspondant à des connections réelles sont marqués d'un point.

Fonctions et propositions logiques à une variable

Toute fonction logique peut être représentée par une proposition logique. Pour préciser ce que cela signifie, considérons une fonction logique d'une variable f. À une valeur de {0 ; 1}, elle en associe une autre (qui peut être la même, bien sûr !). La fonction logique f peut donc être représentée par un tableau. En fait, il en existe quatre possibles en tout, les voici :

0 1	**0 1**	**0 1**	**0 1**	**Les quatre tableaux**
↓ ↓	↓ ↓	↓ ↓	↓ ↓	**des fonctions logiques**
0 0	**0 1**	**1 0**	**1 1**	**d'une variable.**

« Quatre » provient du nombre de possibilités de disposer 0 et 1 comme résultat, ce qui donne $2^2 = 4$ possibilités. Si on interprète 0 comme « faux » et 1 comme « vrai », ces quatre tableaux ressemblent à des tables de vérité :

F V	**F V**	**F V**	**F V**
↓ ↓	↓ ↓	↓ ↓	↓ ↓
F F	**F V**	**V F**	**V V**

La première et la dernière table retournent des constantes, la seconde est l'identité (les valeurs « vrai » et « faux » sont inchangées) et la troisième est le « non » logique (les valeurs « vrai » et « faux » sont inversées). On peut éliminer les constantes si on le désire en utilisant les deux identités suivantes :

p et (non p) = Faux   et p ou (non p) = Vrai

valables pour toute variable logique p. En utilisant les symboles $\vee$ pour « ou », $\wedge$ pour « et » et $\neg$ pour « non », nous obtenons :

$$p \wedge (\neg p) = \text{Faux} \qquad \text{et} \qquad p \vee (\neg p) = \text{Vrai.}$$

Ainsi, toute fonction logique f à une variable peut être représentée par une proposition logique P portant sur une variable p. Plus précisément, les quatre fonctions ci-dessus sont représentées respectivement par les quatre propositions $p \wedge (\neg p)$, p, $\neg p$ et $p \vee (\neg p)$. Le tableau de f coïncide avec la table de vérité de P. Plus précisément, si p a la valeur de vérité « vrai », $P(p)$ vaut f(« vrai »), et si p a la valeur de vérité « faux », $P(p)$ vaut f(« faux »).

Fonctions et propositions logiques à plusieurs variables

Examinons le cas des fonctions logiques à deux variables. Nous pouvons les représenter sous forme de tableaux (voir l'encadré précédent). Il y en a seize :

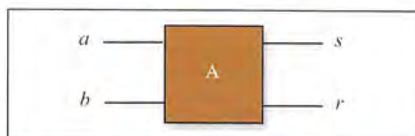
(0 ; 0)	(0 ; 1)	(1 ; 0)	(1 ; 1)	(0 ; 0)	(0 ; 1)	(1 ; 0)	(1 ; 1)
↓	↓	↓	↓	↓	↓	↓	↓
0	0	0	0	0	0	0	1

et ainsi de suite. On peut également associer seize tables de vérité à ces seize tableaux. Comme précédemment, nous pourrions les passer toutes en revue pour montrer que chacune est associée à une proposition logique portant sur deux variables p et q. Il est plus simple de raisonner autrement.

Soit f une fonction logique à deux variables. Nous associons à ces deux variables deux variables propositionnelles p et q et fixons la seconde (qui correspond à q) à « vrai ». Il nous reste une fonction logique à une seule variable. D'après ce qui précède, elle peut être représentée par une proposition logique de la variable p, soit P. Nous pouvons reprendre le raisonnement en fixant q à « faux ». Nous obtenons à nouveau une proposition logique de la variable p, soit Q. La fonction logique f est alors représentée par la proposition logique suivante des deux variables p et q : $(P \wedge q) \vee (Q \wedge (\neg q))$. En effet, si q a la valeur de vérité « vrai », f a les mêmes valeurs que P (suivant les valeurs de vérité de p) et donc que $P \wedge q$ et, de même, si q a la valeur de vérité « faux », f a les mêmes valeurs que Q et donc que $Q \wedge (\neg q)$. Dans tous les cas, f a les mêmes valeurs de vérité que la proposition $(P \wedge q) \vee (Q \wedge (\neg q))$.

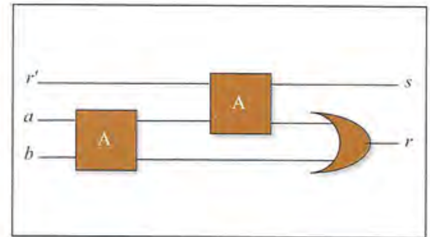
Le raisonnement que nous venons de faire s'itère, et donc le résultat est vrai quel que soit le nombre de variables.

À partir de cet additionneur un bit, on peut construire un additionneur n bits. Il faut cependant tenir compte de la retenue. L'additionneur un bit élémentaire à utiliser a donc une entrée supplémentaire (pour pouvoir prendre en compte une éventuelle retenue). Il est possible de le fabriquer à partir de ce premier (que nous notons d'un carré) :

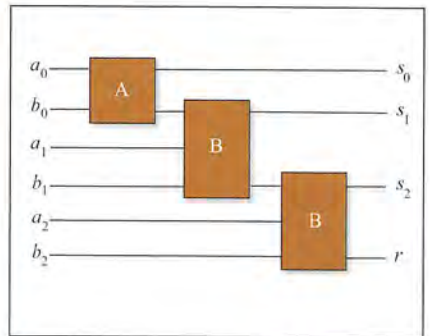


Symbole de l'additionneur un bit (sans retenue).

Nous pouvons construire un additionneur admettant une retenue r' en entrée :



Vous pouvez vérifier le fonctionnement de ce circuit en en dressant la table de vérité. Nous pouvons alors facilement construire un additionneur n bits. Il suffit pour cela de composer n additionneurs de ce type (noté B) :



Additionneur à trois bits. Il est facile de le généraliser à n bits.

De la même façon, nous pouvons créer un circuit multiplicateur et ainsi un processeur d'ordinateur. Nous voyons que l'essentiel est dans *la logique* et non dans *l'électronique*. On pourrait très bien imaginer des portes logiques fondées sur un autre phénomène. Même l'eau est utilisable, il suffit de fabriquer les trois portes logiques « non », « et » et « ou ». Alors, pourquoi pas un ordinateur à eau ? La raison, c'est le temps. Un tel ordinateur serait très lent et peu efficace.

H. L.

L'algorithme de Stern–Brocot

Moritz Abraham Stern.

Cancres et suites de Farey

La *suite de Farey* F_n d'ordre n est construite de la manière suivante : on range, dans l'ordre croissant, toutes les fractions irréductibles comprises entre 0 et 1 dont le dénominateur est inférieur ou égal à n. Ainsi, $\mathbf{F_1 = \{0\ ;\ 1\}}$, $\mathbf{F_2 = \{0\ ;\ ½\ ;\ 1\}}$, $\mathbf{F_3 = \{0\ ;\ 1/3\ ;\ ½\ ;\ 2/3\ ;\ 1\}}$...
Les suites de Farey jouissent de nombreuses propriétés mathématiques. En voici deux.
Si p/q et p'/q' sont deux fractions consécutives de F_n, alors $\mathbf{qp' - qp' = 1}$.
Si p/q, p''/q'' et p'/q' sont trois fractions successives d'une suite de Farey, alors :

$$p''/q'' = (p + p')/(q + q').$$

Cette dernière relation est liée à « l'addition des cancres », définie par :

$$a/b \oplus c/d = (a + c)/(b + d).$$

Une propriété de ⊕ est que $a/b \oplus c/d$ est compris entre a/b et c/d dès que $|\ bc - ad\ |$ est égal à 1. Avec cette propriété, on démontre de nombreux résultats sur les suites de Farey. Les cancres, en utilisant sans le savoir les suites de Farey, peuvent parfois avoir la main heureuse...

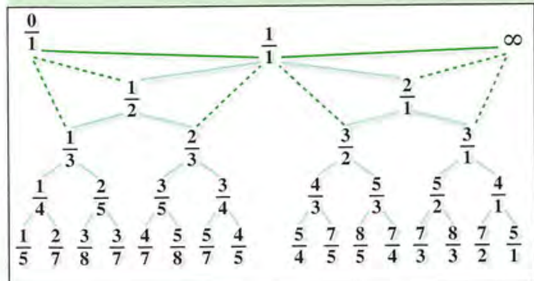
Avec une calculatrice

Calculons 3/7 à l'aide de notre calculatrice. Nous obtenons 0,42857142857...
Une question simple se pose : comment pouvons-nous retrouver rapidement 3/7 à l'aide du seul développement décimal 0,42857142857... ?
Pour cela, nous pouvons mettre en œuvre l'algorithme de Stern–Brocot, découvert indépendamment par le mathématicien allemand Moritz Stern (1807–1894) en 1858 et par l'horloger français Achille Brocot (1817–1878) en 1861.

Description de l'algorithme

Donnons-nous un nombre x que nous allons encadrer par des fractions « simples » r et s. Si, à une certaine étape, nous avons $r \leq x \leq s$, alors soit $\mathbf{r \leq x < r \oplus s}$, soit $\mathbf{x = r + s}$, soit $\mathbf{r \oplus s < x \leq s}$.
Dans le premier cas, on remplace, à l'étape suivante, s par $r \oplus s$. Dans le deuxième cas, le processus s'arrête. Dans le troisième cas, on remplace, à l'étape suivante, r par $r \oplus s$. Et, pour initialiser l'algorithme, nous posons $\mathbf{r = «0/1»}$ et $s = \infty$.

Pour x rationnel positif, on démontre que le processus s'arrête toujours ! Et ce principe de construction permet, à partir des deux données initiales « 0/1 » et ∞, de dessiner un arbre infini (l'arbre de Stern–Brocot). Tous les nombres rationnels figurent, une fois et une seule et sous forme irréductible, dans cet arbre.



Étape 1. $r =$ « 0/1 », $s = \infty$, $r \oplus s = 1/1$;
$x < r + s$, $s := 1/1$.

Étape 2. $r =$ « 0/1 », $s = 1/1$, $r \oplus s = 1/2$;
$x < r + s$, $s := 1/2$.

Étape 3. $r =$ « 0/1 », $s = 1/2$, $r \oplus s = 1/3$;
$r + s < x$, $r := 1/3$.

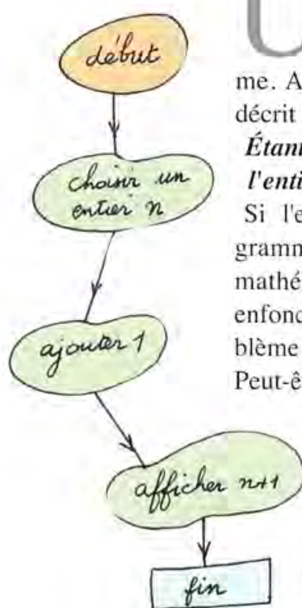
Étape 4. $r = 1/3$, $s = 1/2$, $r \oplus s = 2/5$;
$r + s < x$, $r := 2/5$.

Étape 5. $r = 2/5$, $s = 1/2$, $r \oplus s = 3/7$;
$\mathbf{x = 3/7}$.

par Michel Rousselet

N'abusons pas des organigrammes !

Un organigramme est une représentation graphique conventionnelle d'un algorithme. Très « mode » il y a une trentaine d'années, ils sont aujourd'hui remis en question. Les organigrammes ont-ils encore une utilité ?



Un organigramme est une représentation graphique conventionnelle d'un algorithme. Ainsi, l'organigramme ci-contre décrit l'algorithme suivant :

Étant donné un entier n, on obtient l'entier suivant en lui ajoutant 1.

Si l'effet décoratif d'un tel organigramme est discutable, son intérêt mathématique est quasi nul ! On enfonce des portes ouvertes car le problème posé est vraiment trop simple.

Peut-être aurons-nous plus de chances avec le deuxième organigramme qui est extrait d'un livre écrit en 1969 par le mathématicien Jean Kuntzmann (1912–1992) et consacré à divers algorithmes. Il constitue une solution à l'exercice suivant, posé par l'auteur :

« Décrire, par un organigramme, un algorithme permettant de nommer la tranche des mille d'un nombre entier écrit en décimal. »

Cette fois, c'est à cause de sa complexité qu'il est à craindre que l'intérêt de cet organigramme soit lui aussi très limité, sauf peut-être pour un linguiste professionnel.

Trop simple, un organigramme ne sert à rien. Trop compliqué, il ne sert à rien non plus. Alors à quoi peut bien servir un organigramme ? En fait, un organigramme peut présenter un réel intérêt pédagogique à condition de s'en servir pour expliquer à des débutants en algorithmique que tout algorithme repose sur trois « piliers » fondamentaux, qui sont la *séquence*, la *répétition* et l'*alternative*. Dans ce cas encore, on vérifie l'adage de Napoléon : *« Un petit dessin vaut mieux qu'un long discours ! »*

Babylone sur organigramme

L'article en page 38 explique comment l'algorithme de Babylone (ou de Héron) permet de calculer des racines carrées de façon approchée.

Voici cet algorithme, appliqué au calcul de $\sqrt{50}$, expliqué avec des « mots » :

1re étape
On choisit librement une première valeur approchée de $\sqrt{50}$, par exemple la moitié de 50. On pourrait tout aussi bien choisir 2 ou 3, *etc.*

2e étape
En divisant 50 par 25, on trouve 2 ; $\sqrt{50}$ est donc compris entre 2 et 25. La deuxième valeur approchée de $\sqrt{50}$ sera la moyenne arithmétique de 2 et de 25 soit 13,5.

3e étape
En divisant 50 par 13,5 on trouve 3,7037 ; $\sqrt{50}$ est donc compris entre 3,7037 et 13,5. La troisième valeur approchée de $\sqrt{50}$ sera la moyenne arithmétique de 3,7037 et de 13,5 soit 8,6018.

Étapes suivantes
On continue autant de fois qu'il est nécessaire en *répétant* le procédé. On obtient ainsi des valeurs approchées de plus en plus précises de $\sqrt{50}$. La méthode se généralise à tout nombre positif.
Au lieu d'utiliser le langage courant pour décrire cet algorithme, on peut utiliser un *organigramme* qui le présente de manière plus visuelle et surtout plus synthétique.
Cependant, avant de vous livrer cet organigramme, nous ferons trois remarques qui permettront de mieux comprendre comment il a été construit.

1°) La première approximation mise à part, on peut observer que toutes les autres sont calculées de la même façon : si v est l'approximation calculée immédiatement après l'approximation u, on a $v = \frac{1}{2}\left(u + \frac{50}{u}\right)$. Les valeurs changent mais le même calcul est *répété* plusieurs fois.
2°) Contrairement à ce qu'on pouvait penser *a priori*, la formule $v = \frac{1}{2}\left(u + \frac{50}{u}\right)$ montre que deux *variables* seulement suffisent pour décrire la suite des valeurs approchées de $\sqrt{50}$.
3°) Il existe différents moyens *d'arrêter les calculs*. On peut fixer à l'avance le nombre des approximations qui seront calculées, 20 ou 30 par exemple. On peut aussi décider de stopper les calculs lorsque une certaine précision, fixée elle aussi à l'avance, est atteinte. Pour cela, au cours des calculs, on peut **tester** la valeur absolue de la différence de deux valeurs approchées consécutives u et v. On décidera par exemple

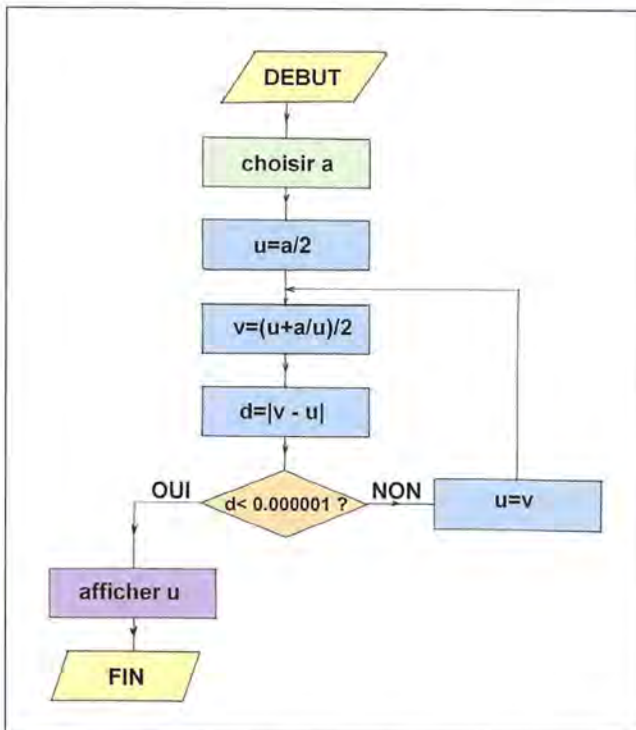
Organigramme de J. Kuntzmann, in *Méthodes numériques*, Hermann, collection Enseignement des sciences.

La programmation de l'algorithme de Babylone

Avec le langage de programmation Python, on obtient le programme suivant :

```
a=input("Choisir a :")
u=a/2.0
d=abs(a-u)
while d>0.000001 :
        v=(u+a/u)/2.0
        d=abs(u-v)
        u=v
print u
```

Même si les instructions sont exprimées en anglais, on reconnaît la démarche utilisée dans l'organigramme. L'entrée du nombre a se fait à l'aide de l'instruction **input()** : c'est un mot anglais d'origine latine qui signifie « introduire, mettre dedans ». L'instruction **print** (qui signifie afficher) est une instruction de sortie. L'instruction **while** (mot anglais qui signifie **tant que...**), sert à définir une **boucle conditionnelle**.



de cesser les calculs dès qu'on aura $|u - v| < 10^{-6}$.

Vous êtes maintenant invités à consulter l'organigramme annoncé ; il figure ci-contre, dans la marge.

Cet organigramme a un début et une fin. Il comprend trois *variables numériques* (u, v et d) ainsi qu'une *constante numérique* (c'est a). Les *instructions* qu'il comporte sont exécutées *une à une*, de manière *séquentielle*. Une « *boucle* » répète le même calcul tant que c'est nécessaire mais naturellement avec des valeurs différentes. Le nombre de répétitions dépend du résultat du *test* « $d > 0{,}000001$? » placé dans la case rose. Selon la réponse à ce test, les calculs sont poursuivis ou sont stoppés.

L'organigramme comporte plusieurs types d'instructions. On trouve en particulier :

- une *instruction d'entrée* (case verte) : on choisit de calculer non pas seulement $\sqrt{50}$ mais la racine carrée d'un nombre a quelconque mais positif ; comme la valeur de a ne change jamais dans la suite des calculs, on dit que c'est une *constante numérique* ;
- des *instructions d'affectation* (cases bleues), représentées par le signe = ;
- une *instruction de sortie* (case violette) ;
- une *instruction conditionnelle* (case rose).

L'organigramme qui schématise l'algorithme de Babylone peut être transcrit presque immédiatement dans un *langage de programmation*. C'est ce qui vous est proposé en encadré. Nous insistons néanmoins sur le fait qu'on a présenté une situation d'apprentissage. À notre avis, il est faux de croire que la réalisation d'un organigramme doit précéder la mise au point d'un programme.

M. R.

par Bernard Novelli

Algorithme de résolution des sudokus

La résolution des sudokus peut se faire à l'aide d'algorithmes simples. Il faut disposer de deux tableaux : l'un pour les cases du sudoku, l'autre pour noter les chiffres qu'il est possible de placer dans une case donnée de la grille.

Le premier algorithme correspond à l'entrée d'un nombre dans la grille (***a* div *b*** est le quotient de la division euclidienne de *a* par *b*, ***a* mod *b*** est le reste) : on place le nombre *nb* dans la colonne *x* et la ligne *y* et l'on met à jour les possibles.

```
PlaceElement(x, y, nb)
   Sudoku[x, y] = nb
   Pour i de 0 à 8 faire
   Début
        Si i différent de x, possible[i, y, nb] = faux,
        Si i différent de y, possible[y, i, nb] = faux,
        Si 3.(x div 3) + (i mod 3) différent de x ou
   si 3.(y div 3) + (i div 3) différent de y, alors :
             possible[3.(x div 3) + (i mod 3),
             3.(y div 3)+(i div 3), nb] = faux
        Fin
Trouve = vrai
```

Deuxième algorithme : si, dans une case, il ne reste qu'une seule possibilité, on place le nombre correspondant.

```
VerifCase(x, y)
Compteur = 0
Pour n de 1 à 9 faire
Début
   Si possible[x, y, n], alors
   Debut
        Incrémenter Compteur,
        Candidat = n
   Fin
   Si Compteur = 1, alors PlaceElement(x, y, Candidat)
Fin
```

Troisième algorithme : si, dans une ligne, un nombre ne peut se placer qu'à un seul endroit, on le place. On construira de même les algorithmes **VerifColonne(*x*, *n*)** et **VerifBloc(*bl*, *n*)**.

```
VerifLigne(y, n)
Compteur = 0
Pour x de 0 à 8, faire
Début
   Si possible(x, y, n), alors
   Début
        Incrémenter Compteur
        Candidat = x
Fin
```

En fusionnant tous ces algorithmes, nous pouvons résoudre 90 % des sudokus du marché !

```
ResoudSudoku
Trouve = faux
Répéter
Pour x de 0 à 8 faire
   Pour y de 0 à 8 faire
        VerifCase(x, y),
   Pour y de 0 à 8 faire
        Pour n de 1 à 9 faire
             VerifLigne(y, n)
Pour x de 0 à 8 faire
   Pour n de 1 à 9 faire
        VerifLigne(x, n)
        Pour bloc de 0 à 8
             Pour n de 1 à 9 faire
                  VerifBloc(bloc, n)
Jusqu'à non Trouve
```

Cet algorithme ne permet cependant pas de résoudre tous les sudokus, puisque seules les méthodes de base ont été implémentées.

par Michel Rousselet

Programmer l'algorithme d'Euclide

L'algorithme d'Euclide est l'un des plus anciens que l'on connaisse. Il se prête très bien à l'écriture de programmes. Nous proposons ici de l'interpréter avec un langage évolué tel que Python 2.6.1.

Deux entiers non nuls a et b ont au moins un diviseur commun : c'est le nombre 1. S'ils ont d'autres diviseurs communs, l'un d'eux est le plus grand : c'est alors leur PGCD (plus grand commun diviseur). Ainsi, les diviseurs de 28 sont 1, 2, 4, 7, 14 et 28 ; ceux de 21 sont 1, 3, 7 et 21. Le PGCD de ces deux entiers est donc 7. On écrira PGCD(28 ; 21) = 7.
Si le PGCD de deux entiers vaut 1, les deux entiers sont dits *premiers entre eux*. C'est le cas par exemple des nombres 24 et 35. Cette notion de PGCD s'est révélée très importante pour l'étude des propriétés des nombres entiers.

Notes

Voilà des siècles que cet algorithme fonctionne. Pour une démonstration, nous renvoyons les lecteurs aux ouvrages qui la détaillent, comme, par exemple, le HS6 de *Tangente*. Toutefois, il serait bon de rappeler, en particulier aux élèves de lycée qui liront cet article, que l'écriture d'un programme ne dispense pas d'une démonstration, même si le programme « tourne » bien.

Recherche « brutale » du PGCD de deux entiers

Soient a et b deux entiers naturels non nuls, avec $b < a$. Pour trouver leur PGCD, il existe une méthode très simple : on divise systématiquement les deux nombres par tous les entiers inférieurs à b ; le nombre cherché sera le plus grand des diviseurs communs trouvés.

Illustration des *Éléments* d'Euclide, vers 1309—1316.

On peut utiliser cette méthode de recherche dans un programme écrit avec Python 2.6.1 (voir encadré page suivante), un langage de programmation très puissant et qui, de surcroît, est libre.
Cette force brutale n'est pas la plus

économique. Ainsi, avec le programme ci-contre, on obtient PGCD(76084 ; 63020) = 92 au bout de... 63020 divisions ! Mais, rassurez-vous, cela se fait tout de même assez vite.

Le rôle d'Euclide

Au IIIe siècle avant J.-C., en écrivant ses *Éléments*, Euclide a voulu exposer la totalité des mathématiques de son temps. Bien qu'il ne l'appelle pas ainsi, il traite du PGCD dans le livre VII. Il commence par définir deux entiers qui sont premiers entre eux : ce sont des entiers dont « *la plus grande commune mesure [(le PGCD)] vaut 1* ». Dans une proposition, il expose la façon de rechercher cette commune mesure :

« Deux nombres inégaux étant proposés, le plus petit étant toujours retranché du plus grand, si le reste ne mesure celui qui est avant lui que lorsque l'on a pris l'unité, les nombres proposés seront premiers entre eux. »

Il affirme ensuite que si les deux entiers ne sont pas premiers entre eux, ils ont alors une commune mesure (un PGCD) supérieure à 1.

Première version de l'algorithme d'Euclide

La méthode préconisée par Euclide porte aujourd'hui le nom d'*algorithme d'Euclide*. Selon l'historien des mathématiques Jean Itard (1902–1979), l'énoncé en est un peu obscur :

« Voici ce qu'il signifie. On a deux nombres A et B. Le plus grand est A. On retranche B de A. Le reste est C. On retranche le plus petit des deux nombres B et C du grand. On continue ainsi jusqu'à ce que les deux nombres soient égaux ou jusqu'à ce que la dernière soustraction laisse un reste nul. »

Le programme « brutal »

```
# Recherche du PGCD de deux entiers

a=input("Choissez l'entier a :")
b=input("Choisissez l'entier b :")
if b>a :
    a,b=b,a   # on échange a et b si a<b
d=1
pgcd=d
while d<=b :        # on essaye le diviseur d tant
                      qu'il reste inférieur à b
    d=d+1
    r1=a%d   # reste de la division euclidienne
               de a par d
    r2=b%d   # reste de la division euclidienne
               de b par d
if r1==0 and r2==0 :
            # dans ce cas, d divise a et b à la fois
    pgcd=d
print "Le pgcd de", a, "et de",b, "est", pgcd
```

Appliquons cet algorithme aux deux nombres A = 50 et B = 30 en faisant le tableau suivant :

A	B	D = A - B
50	30	20
30	20	10
20	10	10
10	10	0

Le PGCD cherché est 10.

Appliquons-le ensuite aux deux nombres A = 120 et B = 72 :

A	B	D = A - B
120	72	48
72	48	24
48	24	24
24	24	0

Référence
Jean Itard, *Mathématiques et Mathématiciens*, Magnard, Paris, 1959.

Le programme PGCD par la méthode des divisions

```
# Recherche d'un PGCD par la méthode des divisions

a=input("Choissez l'entier a :")
b=input("Choisissez l'entier b :")
D=max(a,b) # on appelle D le plus grand des
             deux nombres choisis
d=min(a,b)  # on appelle d le plus petit des deux
             nombres choisis
             (Python distingue D et d)
r=D%d       # on calcule le reste de la division
             euclidienne de D par d
while r !=o : # tant que r est différent de o
    D=d
    d=r
    r=D%d   # on calcule le reste de la division
             euclidien de D par d
print "Le pgcd de", a, "et de",b, "vaut",d
```

Pour trouver PGCD(76 084 ; 63 020) = 92, ce programme effectue 7 divisions seulement !

Création d'une fonction PGCD

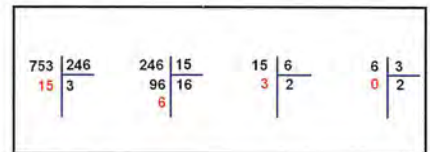
Si un programme est appelé à être souvent utilisé, il peut être intéressant de l'ajouter à la liste des fonctions reconnues par le langage de programmation. Ainsi, la fonction PGCD définie par le programme ci-dessous, une fois placée dans une des bibliothèques de fonctions de Python, sera reconnue aussi facilement que les fonctions élémentaires.

```
# Fonction PGCD de deux entiers
def pgcd(a,b):
if b>a :
    a,b = b,a
    D=a
    d=b
    r=D%d
    while r!=o :
    D=d
    d=r
    r=D%d
return d
```

Ces deux exemples nous suggèrent un programme de calcul très simple qui devra comprendre une *boucle* puisqu'on doit répéter plusieurs fois de suite le même calcul. La sortie de boucle se fera lorsque la différence calculée vaudra 0. L'encadré ci-dessous détaille la construction du programme comme elle pourrait se faire dans la réalité, avec des corrections successives qui permettent finalement la réalisation d'un programme correct.

Deuxième version de l'algorithme d'Euclide

Dans la recherche de la plus grande commune mesure de deux entiers, Euclide remplace les soustractions par des divisions. Pour obtenir par exemple le PGCD de 753 et de 246, il faut effectuer les quatre divisions suivantes :



Le PGCD cherché est 3 : c'est le dernier reste non nul de cette série de divisions. Le nouveau programme de calcul du PGCD de deux nombres écrit avec Python est disponible en encadré. On peut modifier légèrement le programme précédent pour en faire une fonction appelée naturellement PGCD(a,b). Cette fonction (voir encadré) pourra être placée dans une bibliothèque de fonctions qu'on appellera au fur et à mesure des besoins. L'algorithme d'Euclide n'a pas fini de faire parler de lui dans cet ouvrage. Vous trouverez en particulier un autre programme dans les pages consacrées à la récursivité.

M. R.

Construire un programme à petits pas

```
# Recherche d'un PGCD par la méthode
des soustractions (version 1)

A=input("Choisissez l'entier A :")
# on entre la valeur de A
B=input("Choisissez l'entier B :")
# on entre la valeur de B
D=A-B
while D!=0 :
# tant que D est différent de 0
    A=B
    B=D
    D=A-B
print "Le pgcd cherché vaut", B
```

Avec A = 50 et B = 30 comme avec A = 120 et B = 72, tout semble aller bien : on obtient effectivement 10 puis 24. Mais prenons maintenant A = 55 et B = 24. Rien ne va plus, le programme semble ne plus vouloir s'arrêter ! Stoppons-le et ajoutons quelques lignes pour voir ce qui se passe :

```
# Recherche d'un PGCD par la méthode
des soustractions (version 2)

A=input("Choisissez l'entier A :")
B=input("Choisissez l'entier B :")
D=A-B
while D!=0 :
    A=B
    print "A=",A,
    B=D
    print "B=",B,
    D=A-B
    print "D=",D
print "Le pgcd cherché vaut", B
```

Voici les premières lignes qui sont imprimées :

```
Choisissez l'entier A : 55
Choisissez l'entier B : 24
A= 24     B= 31     D= -7
A= 31     B= -7     D= 38
A= -7     B= 38     D= -45
A= 38     B= -45    D= 83
A= -45    B= 83     D= -128
A= 83     B= -128   D= 21
A= -128   B= 211    D= -339
A= 211    B= -339   D= 550
A= -339   B= 550    D= -889
A= 550    B= -889   D= 1439
```

On constate que, contrairement aux indications d'Euclide, on a D > B au départ, ce qui produit A > B quand on fait ensuite A = B et B = D. Tout le reste s'ensuit. Supprimons alors les lignes supplémentaires puis ajoutons deux nouvelles lignes pour échanger les valeurs de B et de D si D > B :

```
# Recherche d'un PGCD par la méthode
des soustractions (version 3)

A=input("Choisissez l'entier A :")
B=input("Choisissez l'entier B :")
D=A-B
while D!=0 :
    if B<D :    # si B>D on échange B et D
    B,D=D,B
    A=B
print "A=",A,
B=D
print "B=",B,
D=A-B
print "D=",D
print "Le pgcd cherché vaut", B
```

Les lignes en rouge ont été insérées pour visualiser les étapes intermédiaires. Cette fois tout va bien !
Pour trouver l'égalité :
PGCD(76 084 ; 63 020)=92, ce programme effectue 21 soustractions.

par M. Criton et A. Zalmanski

La fonction d'Ackermann

Le mathématicien allemand Wilhelm Ackermann (1896–1962) fut élève de David Hilbert. Ses travaux ont porté principalement sur la notion de récursivité et sur la non-contradiction de l'arithmétique.
En 1928, il publie un article intitulé *Zum Hilbertschen Aufbau des reellen Zahlen*, dans lequel il donne un exemple de fonction récursive sans être récursive primitive (une fonction récursive primitive est construite à partir de la fonction successeur et de la composition des fonctions, et doit en outre satisfaire à certains axiomes).
La fonction imaginée par Ackermann était initialement une fonction de trois variables. La fonction connue aujourd'hui sous le nom de *fonction d'Ackermann* est une fonction de deux variables entières, et à valeurs dans les entiers. Elle est définie récursivement. La définition qui suit a été proposée par les mathématiciens Rózsa Péter (1905–1977) et Raphael Robinson (1911–1995).

La fonction d'Ackermann est définie par les relations suivantes :
$\text{Ack}(0 ; n) = n + 1$,
$\text{Ack}(m + 1 ; 0) = \text{Ack}(m ; 1)$,
$\text{Ack}(m + 1 ; n + 1) = \text{Ack}(m ; \text{Ack}(m + 1 ; n))$.

La particularité de cette fonction est sa croissance extrêmement rapide lorsque l'on fait croître m et n. Ainsi, $\text{Ack}(0 ; 0) = 1$, $\text{Ack}(0 ; 1) = 2$, $\text{Ack}(1 ; 0) = 2$, $\text{Ack}(1 ; 1) = 3$; $\text{Ack}(4 ; 0) = 13$, $\text{Ack}(4 ; 1) = 65\,533$, $\text{Ack}(4 ; 2) = 2^{65\,536} - 3$.

L'algorithme de Flavius Josèphe

Pour éviter l'esclavage, Flavius et ses 39 compagnons décidèrent de se suicider après leur défaite contre les Romains en 70. Il se placèrent sur un cercle, se numérotèrent de 1 à 40 et, à tour de rôle le septième fut tué, jusqu'à ce qu'il n'en reste plus qu'un. Flavius choisit le numéro qui lui permettait d'être ce dernier et... ne se suicida pas ! Quelle place avait-il choisie ?

Il est possible de généraliser à n personnes placées sur un cercle, numérotées de 1 à n. On enlève la $p^{ème}$ personne, p étant premier avec n. Quelle est la place de la personne qui subsistera *in fine* ?

Équations récurrentes en finance p. 70
La programmation fonctionnelle p. 76
Le noyau d'un graphe p. 80
Le pivot de Gauss p. 86
L'algorithme du simplexe p. 90
La tour d'Hanoï p. 96
Comment explorer un labyrinthe ? p. 100

ALGORITHMES CLASSIQUES ET JEUX

Les mathématiques ont donné naissance à une multitude d'algorithmes, dans tous les domaines : théorie des nombres, topologie, mathématiques financières, recherche opérationnelle, théorie des graphes, mathématiques récréatives…

par Daniel Justens

Équations récurrentes en finance

Le modèle d'intérêt simple est inefficace à court terme, et incohérent à long terme. L'utilisation de l'intérêt composé conduit à des équations polynomiales de degré égal à la durée de l'investissement. Une bonne façon de déterminer « la » solution de l'équation trouvée, celle qui a un sens économique, est l'algorithme de l'échéance moyenne.

Face à un problème concret, le mathématicien a plusieurs options. Il peut en donner une représentation très simplifiée. Dans ce cas, le problème se réduira souvent à la résolution d'une ou plusieurs équation(s) élémentaire(s) livrant la solution sous forme d'une *formule*. Cette représentation du réel sera généralement jugée insuffisante et le matheux devra fournir une description plus réaliste. Lorsque le modèle plus complexe est traduit en équations, il est fréquent que ces dernières ne livrent leur solution qu'au moyen d'un *algorithme* : succession de démarches et de calculs intermédiaires aboutissant à la longue à la solution. Si la représentation du réel est toujours jugée trop caricaturale, le malheureux mathématicien devra ajuster une fois encore son modèle aux demandes de son employeur et renoncer à l'obtention de solutions exactes pour se contenter d'*heuristiques*.

Seule la solution qui a un sens économique nous intéresse.

Nous allons illustrer ces propos au moyen d'un problème financier : le calcul de la rentabilité d'un investissement. Considérons un employeur (sans aucun sens social) investissant dans un matériel (coût V = 100 000 €) lui permettant d'éviter pendant 5 ans (durée de vie du matériel) l'engagement d'un ouvrier supplémentaire (coût a = 25 000 € par an). Cette opération est-elle économiquement rentable ?

Mise en équation linéaire

L'étalement des économies réalisées nous contraint à recourir à des formules de mathématique financière donnant la valeur d'un capital à un autre instant que sa date d'échéance, en fonction d'un paramètre économique : le taux d'intérêt noté i. La façon la plus élémentaire de travailler est *l'intérêt simple*, une description linéaire. Soit un capital C échéant à un instant que nous baptisons *origine des temps* ($t = 0$). La valeur de ce capital en un instant t quelconque est donnée par :

$$C(t) = C\,(1 + it).$$

En équilibrant notre investissement et la somme des économies *actualisées* (calculées au moment de l'investissement), on arrive à l'égalité :

$$V = a(1 - i) + a(1 - 2i) + a(1 - 3i) + a(1 - 4i) + a(1 - 5i),$$

dans laquelle les valeurs négatives du temps traduisent des moments de calcul antérieurs à l'échéance du capital, ce qui est le cas des cinq « économies » successives réalisées. La résolution générale (à un horizon n) de l'équation fait intervenir les sommes de termes en suite arithmétique et livre une première idée du rendement de l'investissement :

$$i = [a - V/n]\,/\,[a(n + 1)/2].$$

Dans notre problème ($V = 100\,000$, $a = 25\,000$ et $n = 5$), on obtient :

$$i = 1/15 = 0{,}0666\ldots$$

On constate que la formule conduit parfois à des absurdités : avec ce taux (1/15), des économies réellement réalisées, dans 16 ans ou plus, auront une valeur actuelle négative !

Mise en équation exponentielle

Un modèle plus élaboré de l'évolution d'un capital en fonction du temps est l'*intérêt composé*. Avec les mêmes notations que plus haut :

$$C(t) = C\,(1 + i)^t.$$

Cette façon de faire est la seule description en univers stable, différentiable, pour laquelle les accroissements de capital sont proportionnels au capital, ce qui est économiquement incontournable. L'équation d'équilibre devient :

$$V = a(1 + i)^{-1} + a(1 + i)^{-2} + a(1 + i)^{-3} + a(1 + i)^{-4} + a(1 + i)^{-5}$$

qui peut s'écrire formellement (horizon n) : $a(x + x^2 + \ldots + x^n) - V = 0$. Nous savons que les équations polynomiales de degré supérieur ou égal à 5 n'admettent pas de solution générale exprimable au moyen d'une formule. Notre équation particulière admet 5 (c'est-à-dire n) solutions réelles ou complexes conjuguées, dont une seule nous intéresse : celle qui a un sens économique.

L'équation comprend la somme de 5 (soit n) termes en suite géométrique de raison $(1 + i)^{-1}$. En calculant cette somme, on arrive après quelques manipulations algébriques élémentaires à :

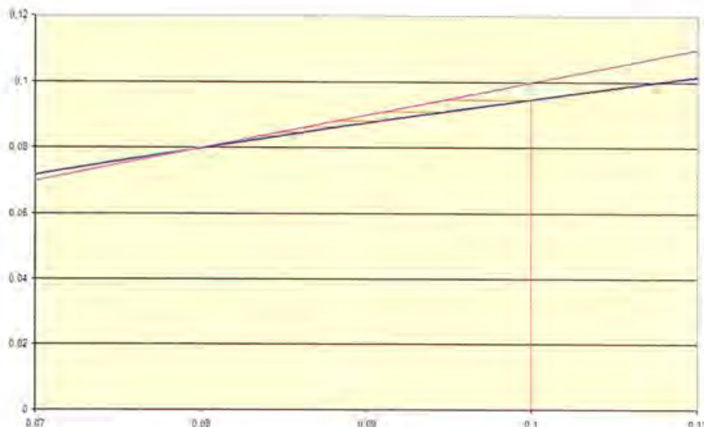
$$f(i) = V - a\,/\,i\,[1 - (1 + i)^{-n}] = 0.$$

On peut transformer cette équation en une autre, équivalente, prenant une forme dite *récurrente* $i = \varphi(i)$, en

isolant le i du dénominateur dans le membre de droite :

$$i = a / V [1 - (1 + i)^{-n}] = \varphi(i).$$

En partant d'une valeur arbitraire i_0, on construit une suite de valeurs i_k par application répétée de $i_{k+1} = \varphi(i_k)$. Lorsque la dérivée en valeur absolue de $\varphi(i)$ est bornée par un réel strictement inférieur à 1, la suite ainsi construite converge. En choisissant $i_0 > 0$ raisonnablement proche de zéro (c'est un taux d'intérêt), on obtient une suite de valeurs qui va tendre vers la solution économiquement raisonnable de l'équation. Avec nos données (V = 100 000, a = 25 000, n = 5 et i_0 = 0,1), on arrive après 65 itérations à la valeur i = 0,07930826 qui équilibre l'équation de départ au centième près. On peut vérifier que $\varphi'(i)$ est « bien bornée » et visualiser la convergence vers la solution de manière géométrique. Sur le graphique qui suit, la fonction $\varphi(i)$ est représentée en bleu, la fonction identité i en rose et la construction des valeurs successives par l'*escalier rouge* qui *descend* vers la solution $i = \varphi(i)$:



La méthode est simple mais peu efficace. Elle cesse d'être applicable dès que l'on introduit une variation dans les économies futures. Or c'est souvent le cas : l'ouvrier non engagé aurait eu un salaire en croissance régulière. L'équation d'équilibre devient :

$$V = a_1(1 + i)^{-1} + a_2(1 + i)^{-2} + a_3(1 + i)^{-3} + a_4(1 + i)^{-4} + a_5(1 + i)^{-5} = F_1(i).$$

Dans notre cas, on peut tabler par exemple (en milliers d'euros) sur $a_1 = 25, a_2 = 26, a_3 = 27, a_4 = 28, a_5 = 29$. Une méthode élégante proposée par Christian Jaumain, né en 1939, et généralisable aux dates non entières, est celle de *l'échéance moyenne*. Jaumain propose de remplacer la fonction $F_1(i) = a_1(1 + i)^{-1} + a_2(1 + i)^{-2} + \ldots + a_n(1 + i)^{-n}$ par une fonction plus simple du type $F_2(i) = K(1 + i)^{-q}$ (q symbolise l'échéance moyenne) en veillant à ce que les deux fonctions soient presque identiques au voisinage de la solution économiquement réaliste (proche de 0). En fait, l'algorithme de Jaumain consiste à remplacer une moyenne pondérée d'exponentielles par une exponentielle moyenne.

On calcule K pour avoir $F_1(0) = F_2(0)$, ce qui livre $K = a_1 + a_2 + \ldots + a_n$. Afin d'assurer un comportement presque identique des deux fonctions dans un voisinage immédiat, on veille à vérifier également $F'_1(0) = F'_2(0)$, ce qui offre une première échéance moyenne :

$$q_1 = (a_1 + 2a_2 + 3a_3 + \ldots + na_n) / (a_1 + a_2 + \ldots + a_n).$$

Au lieu de résoudre $V = F_1(i)$, on traite $V = F_2(i)$, qui se réduit à la *formule* :

$$i_1 = (K / V)^{(1/q_1)} - 1.$$

La solution obtenue ne vérifie pas $F_1(i_1) = F_2(i_1)$. On convient alors de modifier l'échéance moyenne dans F_2 pour obtenir cette égalité, ce qui donne :

$$q_2 = [\ln(K / F_1(i_1)] / [\ln(1 + i_1)].$$

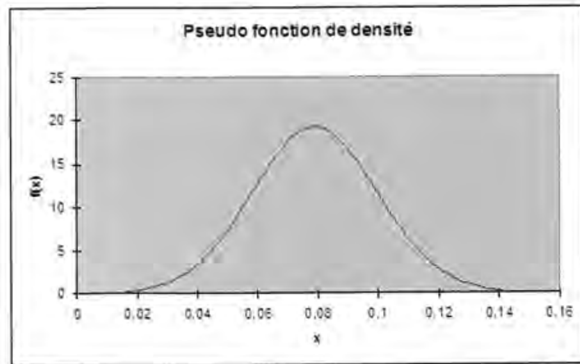
La mécanique étant lancée, on calcule successivement i_2 en remplaçant q_1 par q_2 dans la formule précédente, puis q_3, i_3... Avec les économies constantes de 25000 €, on arrive à la solution après 5 itérations seulement. Les flux variables conduisent à $i = 0{,}10620218$ sans accroître la longueur du processus. Cerise sur le gâteau, la démonstration de la convergence fait apparaître des propriétés intéressantes des moyennes pondérées de puissances.

Le passage en milieux aléatoires

Une des critiques principales aux modèles qui viennent d'être présentés est leur contexte déterministe. En univers réel, rien ne garantit à l'investisseur le montant des économies futures : l'ouvrier engagé peut être mis en chômage économique, on peut occasionnellement devoir engager deux ouvriers, et la liste est longue des aleas altérant la valeur des coûts évités. Dans un modèle réaliste, les a_k prennent la forme de variables aléatoires pour lesquelles, au mieux, on connaît la distribution de probabilité. Dans l'équation :

$$V = a_1(1 + i)^{-1} + a_2(1 + i)^{-2} + a_3(1 + i)^{-3} + a_4(1 + i)^{-4} + a_5(1 + i)^{-5}$$

lorsque les a_k sont des variables aléatoires, chaque ensemble de réalisations de ces variables (que l'on appelle un *futur possible*) détermine une valeur du rendement i qui devient alors une variable aléatoire implicite dont on désire connaître la distribution de probabilité. Pour contourner la complexité du problème, on a recours à la méthode de Monte-Carlo, qui consiste à simuler un grand nombre de futurs possibles et à travailler sur la statistique des rendements calculés. Dans le cas de notre exemple, avec l'hypothèse de 5 variables aléatoires a_k, normales, de moyennes 25 000 € et d'écarts-types 3000 €, on va obtenir, après 1 000 simulations de futurs possibles, une distribution des rendements observés ressemblant à :



Quel est au final le rendement de cet investissement ? Nous pouvons écarter la première solution fondée sur une description linéaire absurde. Si la description exponentielle est justifiable économiquement, elle n'intègre pas les modifications ultérieures non prévisibles et le rendement en apparence suffisant de 7,9 % (ou de plus de 10 % en tablant sur des salaires croissants) se traduit en fait par l'appartenance probable à une fourchette nettement moins confortable oscillant entre 2 % et 14 %. Faut-il alors vraiment renoncer à engager un ouvrier supplémentaire ?

D. J.

par Alain Zalmanski

La suite de Fibonacci

Un problème de lapins

« Un homme place un couple de lapins dans une très grande cage. Combien de couples de lapins obtiendra-t-on en douze mois si chaque couple engendre tous les mois un et un seul nouveau couple à compter du second mois de son existence ? » Voici l'énigme posée en 1202 par Leonardo Fibonacci, ou Léonard de Pise (vers 1175–1240), dans la section III de son *Liber Abaci* (*Livre des abaques*). Ce problème est à l'origine de la suite dont le énième terme correspond au nombre de paires de lapins au énième mois. Dans cette population (idéale), on suppose que :

- le premier mois, il y a juste une couple de lapins qui ne sont pubères qu'à partir du deuxième mois ;
- chaque mois, toute paire susceptible de procréer engendre effectivement un nouveau couple de lapins.

Par ailleurs, on suppose que les lapins ne meurent jamais.

Propriétés des nombres de Fibonacci

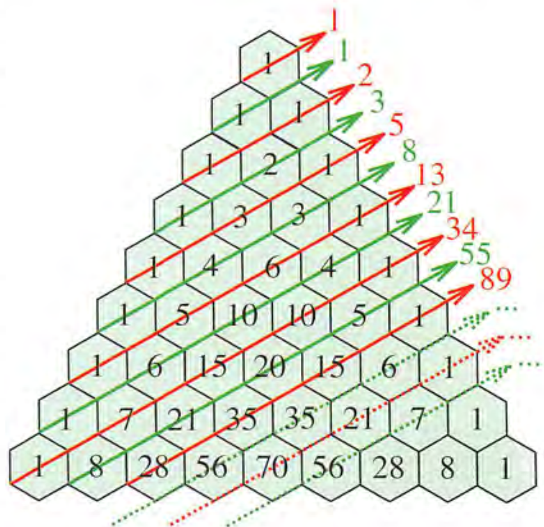
Les nombres de Fibonacci F_n vérifient une pléthore de propriétés : le rapport de F_{n+1} / F_n tend vers le nombre d'or $\left(1+\sqrt{5}\right)/2$, soit environ 1,618 ; F_n peut être explicitement déterminé, et l'on a la *formule de Binet* :

$$F_n = \frac{\left(1+\sqrt{5}\right)^n - \left(1-\sqrt{5}\right)^n}{2^n\sqrt{5}};$$

seuls F_1, F_2 et F_6 sont des cube ; seuls F_1, F_2 et F_{12} sont des carrés ; à l'exception de F_4, si F_n est premier, alors n est premier… Par contre, on ignore toujours si une infinité d'entre eux sont premiers ! Une propriété moins connue des nombres de Fibonacci est la suivante : la somme des nombres de la énième diagonale du triangle de Pascal est égale à F_n. Un petit dessin vaut toute les explications du monde…

Algorithme de comptage

Notons F_n le nombre de couples de lapins au mois n. Jusqu'à la fin du deuxième mois, la population se limite à un couple (ce qu'on note : $F_1 = F_2 = 1$). Dès le début du troisième mois, le couple de lapins a deux mois et il engendre un autre couple de lapins. On note alors $F_3 = 2$. Plaçons-nous maintenant au mois n et cherchons à exprimer ce qu'il en sera deux mois plus tard : F_{n+2} désigne la somme des couples de lapins au mois $n + 2$ et des couples nouvellement engendrés. Or, n'engendrent au mois $n + 2$ que les couples pubères, c'est-à-dire ceux qui existent deux mois auparavant. On a donc : $F_{n+2} = F_{n+1} + F_n$. Cette suite récurrente est dite *de Fibonacci* : chaque terme est la somme des deux termes précédents. Les termes de cette suite sont appelés *nombres de Fibonacci* et la réponse à la question initiale du nombre de paires de lapins après 12 mois est donc 233.



par Alain Zalmanski

Comment faire de la monnaie ?

De combien de façons peut-on faire de la monnaie sur un ancien mark allemand (DM) (ou payer une somme de 100 Pf) avec des pièces de 1, 2, 5, 10 et 50 pfennigs ? On rappelle que 100 pfennigs valent 1 DM. Il n'existe *a priori* pas vraiment de manière « élégante » de résoudre ce problème. On remarque que les réponses sont solutions de l'équation diophantienne **100 = X + 2Y + 5Z + 10U + 50W**, avec X, Y, Z, U, W, respectivement le nombre de pièces de 1, 2, 5, 10 et 50 pfennigs. Une recherche brutale et méthodique conduit à 2 498 façons. En fait, cela revient à réaliser pas à pas un petit algorithme, aisément compréhensible, tel qu'il aurait pu être écrit en Basic il y a 30 ans (<> signifie « différent de ») :

```
10 FOR V = 0 TO 2
20   FOR U = 0 to 10 – 5*V
30     FOR Z = 0 TO 20 – 10*V – 2*U
40       FOR Y = 0 TO 50 – 25*V – 5*U
50         FOR X = 0 TO 100 – 50*V – 10*U – 5*Z – 2*Y
60           IF X*2 + 5*Z + 10*U + 50*V <>100 THEN 80
70           K = K + 1
80         NEXT X
90       NEXT Y
100      NEXT Z
```

Ce problème de partition de monnaie peut se poser pour notre (ancien) francs français (FF) qui disposait de pièces de 1, 2, 5, 10, 20 et 50 centimes (100 centimes = 1 FF). Ou encore pour le dollar, qui dispose de pièces de 1, 5, 10, 25 et 50 cents (100 cents = 1 $). Le franc français pouvait se décomposait de 4 562 façons. Le dollar, ayant moins de sous-multiples, n'a « que » 292 manières de se décomposer. On notera que le problème général du partage d'un entier en somme d'entiers est un problème résolu, mais qui devient vite très complexe dès lors que l'on met des restrictions sur les partages.

Calcul de l'impôt

Le calcul de votre impôt se résume à la question suivante : *« Combien vais-je payer au fisc ? »* Le calcul est basé sur un algorithme simple, mais dont les données de calculs (revenus imposables, plafonnements spécifiques, déductions fiscales et nombre de parts) sont parfois complexes. Voici comment procède l'administration fiscale.

1. Le revenu net global imposable est divisé par le nombre de parts.
2. Au revenu correspondant à une part entière est appliqué le tarif progressif fixé par « tranches » de revenu.

Le barème, tel qu'il résulte de la loi de finances pour 2009, correspondant à une part, est le suivant :

Fraction du revenu imposable (une part)	**Taux** (en %)
N'excédant pas 5 852 €	0
De 5 852 € à 11 673 €	5,50
De 11 673 € à 25 926 €	14,00
De 25 926 € à 69 505 €	30,00
Au-delà de 69 505 €	40,00

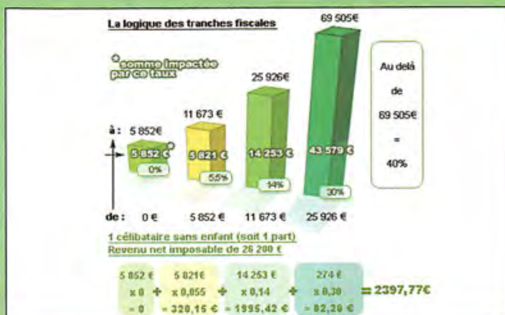
3. L'impôt ainsi obtenu est multiplié par le nombre de parts dont bénéficie le contribuable. Ce produit constitue l'impôt brut correspondant au revenu net global imposable.

Bien entendu on peut traduire en formule ce calcul direct de l'impôt dû des revenus de 2008 :

$R/N \leq 5852$	$I = 0$
$5852 < R/N \leq 11673$	$I = (R \times 0{,}055) - (321{,}86 \times N)$
$11673 < R/N \leq 25926$	$I = (R \times 0{,}14) - (1\,314{,}07 \times N)$
$25926 < R/N \leq 69505$	$I = (R \times 0{,}30) - (5\,462{,}23 \times N)$
$R/N > 69505$	$I = (R \times 0{,}40) - (12\,412{,}73 \times N)$

Le montant des droits simples (I) est donné par application de l'une des formules suivantes, selon le quotient du revenu imposable (R) par le nombre de parts (N).

On remarquera que, contrairement à une idée reçue, une augmentation de revenu fait prendre le risque de « changer de tranche d'imposition » l'année suivante, et d'augmenter ainsi brutalement l'impôt. En fait, seule la partie supérieure de ses revenus subira le taux le plus élevé d'imposition. Ainsi, par exemple, un passage de 5 euros sur la dernière tranche n'affectera que cette somme du taux de 40 %.



par Jean-Jacques Dupas

La programmation fonctionnelle

Un programme est une suite d'instructions qui modifient des données dans la mémoire de l'ordinateur jusqu'à ce que cette mémoire contienne les résultats escomptés. Plusieurs manières d'exécuter un programme existent, par lesquelles la programmation fonctionnelle.

La définition d'un programme (suite d'instructions qui modifient des données dans la mémoire de l'ordinateur jusqu'à ce que cette mémoire contienne les résultats escomptés) est très proche du fonctionnement intime d'un ordinateur. En effet, on imagine aisément le processeur allant chercher des données en mémoire, exécutant des opérations élémentaires sur ces données, puis modifiant la mémoire. Cette approche, dite programmation impérative, est si commune que vous ne vous êtes peut-être jamais demandé s'il y avait une autre possibilité de faire. Une alternative existe pourtant : la programmation fonctionnelle. Elle repose sur une théorie mathématique développée dans les années 1930 par Alonzo Church (1903–1995), le lambda-calcul. Et dès les années 1950, un langage de programmation, le Lisp, s'appuie sur les paradigmes fonctionnels. Mais, au fait… qu'est-ce que la programmation fonctionnelle ?

Alonzo Church.

En programmation fonctionnelle, les fonctions ne font que retourner une valeur.

Un monde de fonctions

Comme son nom l'indique, en programmation fonctionnelle, on ne fait que composer des fonctions. Des fonctions au sens mathématique du terme. C'est-à-dire une correspondance entre des valeurs d'entrée (éléments d'un ensemble de départ) et des valeurs de sortie (éléments d'un ensemble d'arrivée). Vous pensez que la plupart des langages de programmation fournissent des « fonctions » qui retournent une valeur ? Certes, mais la différence essentielle est qu'une fonction mathématique (une *application*) retourne toujours la même valeur pour une valeur d'entrée donnée. Par exemple,

la fonction mathématique « racine carrée » donne toujours 2 si vous lui fournissez 4. Alors qu'il y a peu de chance que la fonction « lire le prochain caractère du fichier » en programmation impérative retourne le même résultat…
Nous touchons là une propriété fondamentale de la programmation fonctionnelle : les fonctions, comme en mathématiques, ne font que *retourner une valeur*, elles ne font rien d'autre. En particulier, elles ne modifient pas les variables. D'ailleurs, en programmation fonctionnelle, il n'y a pas de variables…
Voici la plus grande surprise pour ceux qui découvrent la programmation fonctionnelle : il n'y a pas de variables ! Autre disparition : il n'y a pas de boucles ! De toute façon, à quoi servirait une boucle, puisque l'on ne peut pas modifier de variables d'une itération à l'autre ?
Mais alors, à quoi peuvent bien servirent des fonctions qui ne changent rien, et des langages sans variables ni boucles ? Ces langages sont-ils répandus dans l'industrie ? Apparemment, on ne peut rien en faire…

Récursivité

La solution réside dans la récursivité. La *récursivité* (faculté d'une fonction de s'appeler elle-même) remplace les boucles. La récursivité n'a de sens que si :

- les paramètres de la fonction changent entre les différents appels,
- il existe des arguments pour lesquels la fonction ne s'appelle pas elle-même.

Prenons un exemple de la vie quotidienne. Un enseignant fait l'appel dans une classe.

Les logiciels libres

Un logiciel *libre* (ou *open source*, ou encore *à code source ouvert*) est un logiciel qui respecte notamment les critères suivants : possibilité de libre redistribution, et possibilité d'accéder au code source (pour le lire, le modifier...). Les logiciels libres mentionnés dans cet article sont :

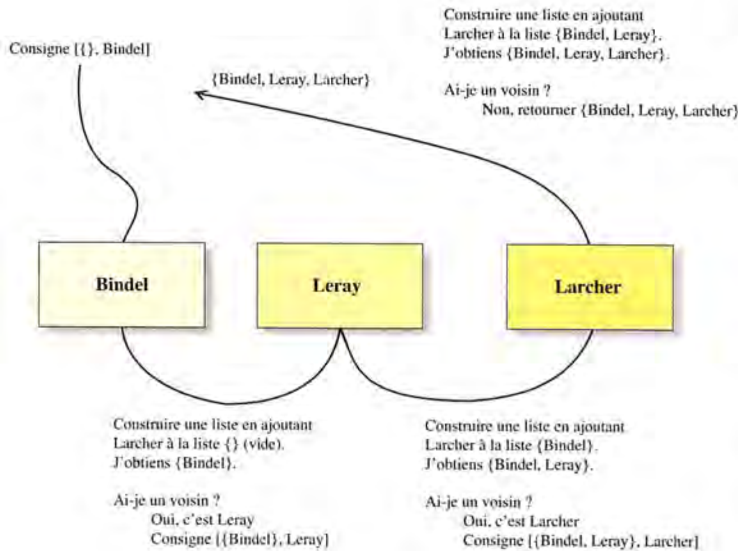
Ruby **www.ruby-lang.org/fr/**,
Python **www.python.org**,
OCaml **http://caml.inria.fr/index.fr.html**.

Page Internet du site du logiciel Ruby.

Première solution (de nature impérative) :
La liste des élèves présents est vide.
Boucle sur les élèves.
    Si l'élève répond présent, alors son nom est ajouté à la liste
Fin de la boucle

Deuxième solution (de nature récursive) :
La consigne est la suivante.
    Je reçois une liste d'élèves.
    Construire une nouvelle liste en ajoutant mon nom à la liste reçue.
    Si j'ai un voisin, alors je lui donne la consigne avec la liste que je viens de construire.
    Sinon, je retourne la liste que je viens de construire.
Donner la consigne au premier élève avec une liste d'élèves vide.

Dans cet exemple se nichent plusieurs fonctions cachées : fonction qui permet d'obtenir le voisin, fonction qui permet d'obtenir le premier élève, fonction qui permet de construire une liste en ajoutant un élément à une liste. Notre fonction récursive renvoie une liste d'élèves et possède deux paramètres : une liste initiale, et un élève. Imaginons une classe (idéale ?) de trois élèves : Bindel, Leray, Larcher. Regardons sur la figure ci-dessous comment fonctionnent la procédure récursive.



Cet exemple montre bien que la consigne est toujours appelée avec des paramètres différents :

Consigne[{}, Bindel]
Consigne[{Bindel}, Leray]
Consigne[{Bindel, Leray}, Larcher]
Consigne[{Bindel, Leray, Larcher}, Larcher]

Et il existe bien une valeur des paramètres (le cas « sans voisin ») pour laquelle la fonction ne s'appelle pas elle-même. Cela permet de retourner le résultat. Et nous voyons également que nous n'avons pas besoin de variables !

Variables ou pas variables ?

Prenons à nouveau une image de la vie quotidienne : les variables sont... comme les tiroirs d'un bureau ! Si vous êtes plusieurs dans un bureau, il y a toutes les chances que, le jour où vous ouvrez un tiroir en espérant y trouver une agrafeuse, vous y trouviez à peu près n'importe quoi sauf une agrafeuse. C'est l'effet de bord classique. On peut objecter l'argument suivant : *« Moi, je suis tout seul dans mon bureau, ce qui correspond à définir une portée pour une variable. Mon tiroir ferme à clef, ce qui correspond à définir une variable privée. »* Mais êtes-vous sûr que vous trouverez votre agrafeuse le moment venu ?

Ne pas remettre instantanément la main sur son agrafeuse n'est pas vital (en général). Par contre, si le programme d'inversion de poussée des réacteurs d'un avion volant à 30 000 pieds d'altitude ne trouve pas la bonne valeur dans une variable, vous comprenez tout de suite le problème. La programmation fonctionnelle permet de construire des programmes plus robustes et plus fiables.

Peut-être entrapercevez-vous le fait que les variables et les boucles ne sont pas l'essentiel de la programmation ? Bien sûr, les « vrais » programmes doivent à un moment ou à un autre « faire quelque chose » et donc interagir avec la mémoire. Cependant, on peut limiter ces interactions au strict nécessaire, et ainsi limiter la possibilité d'effets de bord.

Abstraction

La programmation fonctionnelle possède une autre vertu : elle permet une montée en abstraction algorithmique. En effet, les fonctions peuvent être

elles-mêmes des paramètres d'appel et de retour d'autres fonctions. Notez le parallèle avec les opérateurs mathématiques : l'opérateur de dérivation prend en entrée une première fonction, et retourne en sortie une seconde fonction (sa dérivée).

Voici un exemple simple. Imaginez que vous écriviez un algorithme qui additionne les coordonnées d'un vecteur. En fait, cet algorithme est rigoureusement le même que celui qui multiplie les coordonnées d'un vecteur : il suffit de remplacer la fonction « plus » par la fonction « multiplier ». Donc vous avez tout intérêt à écrire une fonction plus générale, dont l'un des paramètres est une fonction (« plus », « multiplier »...), l'autre paramètre un vecteur, et qui applique cette fonction, passée en paramètre, aux coordonnées du vecteur.

Page Internet du site du logiciel Caml.

Action

Bien. La programmation fonctionnelle permet d'écrire des algorithmes plus proches des mathématiques. Ils sont plus fiables, plus robustes. Mais en pratique, comment faire ? Quel langage choisir ? Nous pouvons faire plusieurs propositions, sans être exhaustif ni (espérons-le) trop directif.

La première option concerne ceux à qui cette nouvelle façon de penser fait vraiment trop peur. Si c'est votre cas, et que par ailleurs vous connaissez la programmation impérative, partez de langages classiques libres (comme Ruby ou Python) qui embarquent aussi les paradigmes fonctionnels. Vous pourrez ainsi vous familiariser avec les concepts sans trop vous éloigner de ce que vous connaissez.

Si vous êtes plus décidé(e) à sauter le pas vers de vrais langages fonctionnels, vous pouvez directement passer à OCaml (Objective Categorical Abstract Machine Language), logiciel libre dirigé et maintenu principalement par l'Inria (Institut national de recherche en informatique et en automatique). Ou bien au logiciel, commercial cette fois ci, F# (prononcer « F sharp »), de Microsoft. Un autre logiciel commercial, Mathematica (de Wolfram Research), s'appuie lui aussi sur les paradigmes fonctionnels.

Actuellement, la programmation fonctionnelle est très utilisée dans le monde universitaire et académique. Elle commence à se répandre dans le monde industriel. Et l'expérience montre que la programmation fonctionnelle n'est pas plus ardue à apprendre que la programmation impérative. Cela vaut donc la peine de s'y intéresser !

J.-J. D.

Référence

The Promises of Functional Programming. Konrad Hissen, *Computing in Science and Engineering*, pp. 86–90, juillet–août 2009.

par Jacques Bair

Gagner au jeu

grâce au noyau d'un graphe

Lorsque les itérées successives d'une fonction convergent, la limite obtenue en est un point fixe. Cette remarque est à l'origine des méthodes d'approximations successives, depuis Newton jusqu'à aujourd'hui.

Pour résoudre de multiples problèmes, tant théoriques que pratiques, il s'avère souvent utile de tracer sur une feuille des points représentant des objets, des personnes, des localités, des tâches…, ainsi que des lignes continues (orientées ou non) reliant certaines paires de ces points et traduisant une relation, une préférence, une possibilité d'accès, une condition d'antériorité…

Chemins et circuits

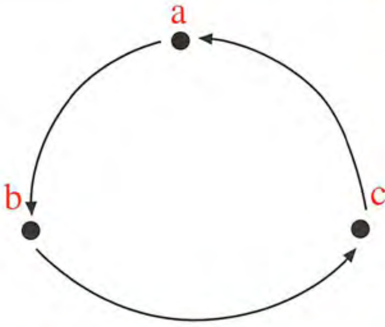
Dans un graphe G, on appelle *chemin* une succession d'arcs telle que l'extrémité terminale de chaque arc (excepté le dernier) coïncide avec l'extrémité initiale du suivant ; un *circuit* est un chemin fermé, c'est-à-dire un chemin tel que le sommet initial du premier arc coïncide avec le sommet terminal du dernier arc.

Telle est l'idée de départ de la théorie des graphes (voir *Tangente* hors série n° 12).

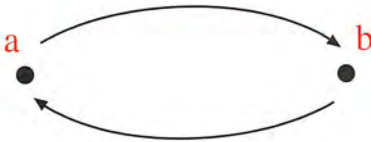
Nous allons ici nous intéresser au concept de *noyau* d'un graphe que nous supposerons *orienté* (les lignes continues entre deux points étant donc des flèches dont le sens doit être respecté). Considérons un tel graphe G : formellement, il s'agit d'un couple $(S\,;\,R)$ composé d'un ensemble fini S dont les éléments sont des *sommets* (matérialisés par des points). R est un sous-ensemble du produit cartésien dont les éléments, qui sont des couples de sommets, sont appelés des *arcs* et représentés par des flèches. Le premier élément d'un tel couple est apelé *origine*, et le second *image*. Nous admettrons de plus que G ne possède aucune *boucle* (une boucle étant un arc dont les deux extrémités coïncident).

Détermination d'un noyau

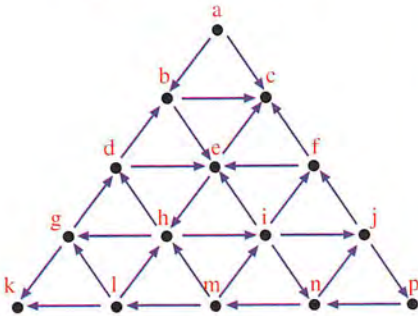
Explicitons une procédure pour construire, s'il existe, le noyau d'un graphe. Avec cette démarche, on montre que G3 ne possède pas de noyau. Pour cela, raisonnons par l'absurde en supposant qu'un tel noyau N existe.
Déjà, clairement, c est dans N car c ne possède aucune image. On en déduit que a, b, e et f ne sont pas dans N. Ainsi, d est nécessairement dans le noyau (car ses seules images sont b et e). Il vient que ni g ni h ne peuvent appartenir à N. Donc k (seule image de g) est nécessairement dans le noyau, et donc l n'y est pas. Maintenant, de deux choses, l'une. Ou bien i est dans le noyau. Alors j et n n'y sont pas, donc m appartient à N (m est la seule image de n). Mais m n'appartient pas à N (car l'une de ses images est i), contradiction. Ou bien i n'est pas dans le noyau. Alors m y est nécessairement, donc pas n, ce qui entraîne que j appartient à N. Il s'ensuit que p est dans le noyau (car son image est n) et qu'il n'y est pas (car il est l'image de j). C'est une contradiction, et G3 ne possède aucun noyau.



Dans ce graphe orienté G1, b est l'image de a, qui est l'image de c, lui-même image de b.



Le graphe orienté G2.



Le graphe G3, introduit par Aviezri Fraenkel (mathématicien israëlien né en 1929) pour représenter un jeu de Nim de son invention.

Par définition, un sous-ensemble N de sommets est appelé *noyau* du graphe G s'il satisfait aux deux conditions suivantes :
a) Si le sommet x appartient au noyau N de G, aucune de ses images (s'il en a) n'appartient à N ;
b) Si x n'appartient pas au noyau N, alors au moins une image de x appartient à N.

Pour illustrer cette définition, considérons un graphe défini par une relation du type « est dominé par » au sein d'un ensemble d'individus. Toute personne située en dehors du noyau est dominée par un élément du noyau (condition *b)*), tandis que, pour deux éléments quelconques appartenant au noyau, l'un des deux ne domine jamais l'autre (condition *a)*).
Sur les exemples introduits précédemment, on vérifie que : G_1 ne possède pas de noyau ; G_2 possède deux noyaux distincts (le premier étant réduit à l'un des deux sommets, l'autre noyau étant réduit au second sommet) ; et G_3

Algorithme pour la construction du noyau dans un graphe sans circuits

I. Encadrer tous les sommets desquels ne partent aucun arc.
II. Encercler tous les sommets *s* non encadrés dont une image, au moins, est un sommet encadré.
III. Supprimer tous les arcs arrivant ou partant des sommets encerclés et encadrés.
IV. Recommencer la procédure sur le sous-graphe composé des sommets non encadrés et non encerclés, jusqu'à ce que tous les sommets soient soit encadrés, soit encerclés. Le noyau est constitué des sommets encadrés.

ne comporte pas de noyau (voir encadré ci-contre).

On démontre qu'un graphe fini sans boucle ni circuit possède un et un seul noyau (voir *Chemins et Circuits* pour les définitions) ; celui-ci peut alors être construit en adoptant la procédure décrite en encadré.

Les jeux de Nim

Le concept de noyau s'avère particulièrement intéressant pour certains jeux de stratégie, notamment pour des jeux à deux joueurs qui choisissent chacun à tour de rôle une option. C'est le cas, par exemple, pour les jeux qui consistent à retirer, à tour de rôle, des éléments de différents tas d'objets selon des règles précises (ce sont les *jeux de Nim*).
À pareil jeu peut être associé un graphe dont les sommets et les arcs représentent respectivement les positions des joueurs et les déplacements possibles. Ce graphe contient un sommet de départ qui correspond à la situation initiale et au moins un sommet d'arrivée indiquant la fin du jeu. Est déclaré vainqueur, par exemple, celui qui atteint le premier un sommet d'arrivée. Tous les sommets d'arrivée font nécessairement partie du noyau. Pour fixer les idées, nous appellerons J le joueur qui débute. Si le sommet de départ est hors du noyau, J peut, à chaque fois qu'il joue, choisir une option le ramenant dans le noyau, ce qui obligera son adversaire à en sortir : J est dès lors certain de gagner la partie à condition de « bien » jouer (c'est-à-dire de revenir à chaque fois dans le noyau). Au contraire, si le sommet initial est dans le noyau, J devra en sortir, de sorte que son adversaire pourra toujours y rentrer : sauf mauvais choix de son adversaire, J ne peut pas espérer gagner le jeu. Connaître le noyau du graphe d'un jeu de Nim, c'est être assuré de gagner !

En guise d'exemple, le lecteur cherchera à déterminer une stratégie gagnante pour chacun de ces deux jeux :
a) Deux joueurs se trouvent en présence de deux tas d'allumettes ; ceux-ci contiennent respectivement 1 et 3 allumette(s). À tour de rôle, les joueurs choisissent un tas non vide et y prélèvent un nombre arbitraire non nul d'allumettes. Le joueur gagnant est celui qui enlève la dernière allumette.
b) On pose 10 billes sur une table et deux joueurs sont en présence. Chacun de ceux-ci peut, à tour de rôle, retirer 1, 2 ou 3 bille(s). Gagne le jeu celui qui enlève la dernière bille.
(Réponse : le joueur qui commence le jeu peut adopter une stratégie gagnante).

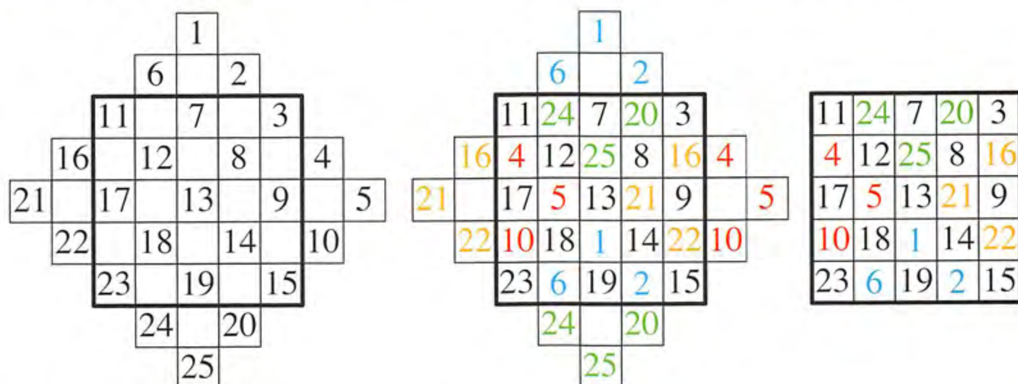
J. B.

RÉFÉRENCE

Les jeux de Nim, Jacques Bouteloup, édité par l'Association pour le développement de la culture scientifique (ADCS), Amiens, 299 pages, 1996.

Construction d'un carré magique d'ordre impair

On rappelle qu'un *carré magique* (*normal*) *d'ordre n* est un carré de n^2 cases comportant les entiers de 1 à n^2, de telle sorte que la somme des nombres sur chaque ligne, chaque colonne et chacune des deux diagonales soit constante. La constante magique est égale à $n(n^2 + 1) / 2$. Une règle de construction des carrés magiques d'ordre impair porte le nom de *méthode de Bachet* : le mathématicien et grammairien Bachet de Méziriac (1581–1638) fut le premier à l'avoir exposée dans un ouvrage en français. Cette méthode était en fait connue depuis plusieurs siècles des mathématiciens persans et arabes et avait été diffusée en Occident par un moine byzantin ayant vécu au XIVe siècle, Manuel Moschopoulos. La méthode, applicable à tous les carrés d'ordre impair, est illustrée par la figure ci-dessous.



11	24	7	20	3
4	12	25	8	16
17	5	13	21	9
10	18	1	14	22
23	6	19	2	15

Jean Tricot, précurseur de la vulgarisation algorithmique

Aujourd'hui, où plus de 60 % des foyers français sont équipés d'un ordinateur, personne ne s'étonnera qu'il existe quelques dizaines de magazines dédiés à l'informatique individuelle.

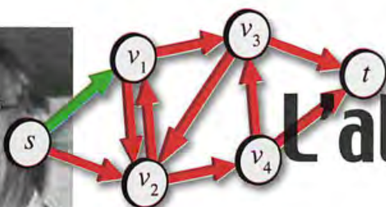
Revenons 35 ans en arrière. L'informatique personnelle en était encore à ses balbutiements, aux États-Unis pour l'essentiel. Saluons alors Jean Tricot qui, dès 1975, tout d'abord dans *Science et Vie* puis dans *Jeux et Stratégies*, consacra une rubrique d'informatique ludique destinée à un très large public.

Les thèmes abordés par Jean Tricot dans ses rubriques comprennent notamment : la machine de Turing, la démonstration du théorème des quatre couleurs, les automates cellulaires, les jeux de Nim, les stratégies gagnantes. On y trouve une mine d'idées utiles aux amateurs d'algorithmes ludiques.

JEUX ET INFORMATIQUE

LA MACHINE DE TURING, ORDINATEUR EN PAPIER

par F. Picard et E. Thomas



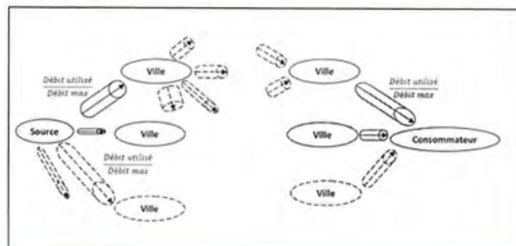
Delbert Ray Fulkerson.

L'algorithme de Ford et Fulkerson

Le problème du débit dans un réseau

Une question souvent difficile est de trouver quel est le débit maximum possible entre deux points éloignés d'un réseau. Ce réseau peut être un ensemble d'ordinateurs reliés entre eux, un système de distribution d'eau ou bien le câblage électrique d'une ville. L'objet circulant dans le système est, selon le type du réseau, une quantité de données, un volume d'eau ou bien une intensité électrique. Le débit doit passer par un certain nombre de liens qui ne peuvent pas dépasser leurs limites. Par exemple, le débit maximum d'un tuyau est fonction de son diamètre, et cette contrainte sur le diamètre ne peut être modifiée.

Dans le cas d'un réseau de distribution d'eau, on pourra avoir le schéma suivant :



Algorithme de calcul du débit maximum

Afin de déterminer le débit maximum d'eau, on applique la procédure suivante :

- Chercher un chemin entre la source et le consommateur,
- Pour chaque canalisation du chemin, si le flux sur la canalisation est dans le sens de la flèche, on ajoute le débit maximum du chemin au débit utilisé sur la canalisation,
- Pour chaque canalisation du chemin, si le flux sur la canalisation *n'est pas* dans le sens de la flèche, on enlève au débit utilisé de la canalisation le débit maximum du chemin,
- On note le débit du chemin que l'on vient d'emprunter,
- S'il reste des chemins à explorer, on retourne à la première étape, sinon on passe à l'étape suivante,
- Afin de trouver le débit maximum du circuit, nous faisons la somme des débits de chacun des chemins. Par superposition, cela nous donne le débit maximum possible de la source au consommateur.

C'est simple, mais il fallait y penser…

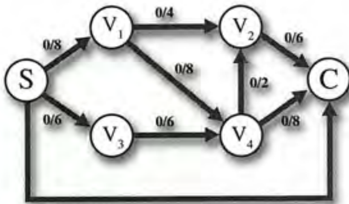
L. R. Ford et D. R. Fulkerson

Lester Randolph Ford Senior (1886–1967) était un mathématicien américain. Il reste connu pour ses *cercles de Ford* (voir Bibliothèque *Tangente* 36) et pour avoir été président de la Mathematical Association of America. Mais son nom reste attaché au Lester R. Ford Award, prix qui récompense annuellement les articles de qualité publiés dans les American Mathematical Monthly ou Mathematics Magazine. C'est à son fils **Lester Randolph Ford Junior**, né en 1927, que l'on doit l'article (en 1956, avec Fulkerson) qui introduit l'algorithme présenté ci-dessus. **Delbert Ray Fulkerson** (1924–1976) était lui aussi un mathématicien américain. Il laisse également son nom à un prix, le Fulkerson Prize, qui récompense tous les trois ans des articles d'excellence dans le domaine des mathématiques discrètes.

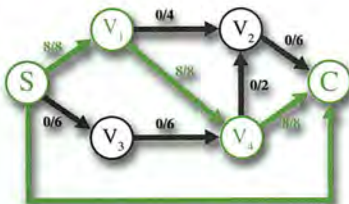
Réseaux de distribution d'eau

Les réseaux de distribution d'eau potable sont des systèmes effroyablement complexes. Des villes (V_1, V_2...) sont reliées les unes aux autres par des tuyaux de diamètres très différents. Le problème est de savoir quel débit, au maximum, on peut envoyer depuis la source jusqu'à l'arrivée.
Ce débit n'est pas limité seulement par les tuyaux placés juste avant le consommateur, mais également par le réseau complexe de tuyaux en amont. Pour savoir quel débit D arrivera au consommateur, nous pouvons employer l'algorithme de Ford et Fulkerson présenté dans la page précédente.

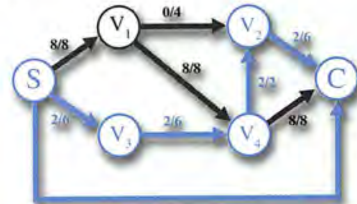
Considérons le réseau de distribution suivant. La source est notée S, le consommateur, C. Les chemins sont représentés par des flèches sur lesquelles est noté le ratio entre le débit utilisé et le débit maximum de la canalisation. Au début, les débits utilisés sont nuls. Par exemple, le débit maximum que l'on peut envoyer de la source aux villes 1 et 3 sont respectivement 8 et 6 unités.



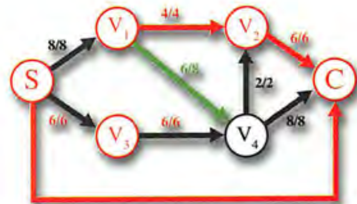
1er chemin : Nous faisons passer un débit de 8 vers le consommateur. Nous saturons les liens (S, V_1), (V_1, V_4) et (V_4, S). Nous ne pourrons plus utiliser ces liens dans le sens la flèche à la prochaine itération.
D = 8.



2e chemin : Nous faisons passer un débit de 2 vers le consommateur. Nous saturons le chemin (V_4, V_2). Nous ne pourrons plus l'utiliser dans le sens conventionnel.
D = 8 + 2.



3e chemin : Il nous reste la possibilité d'un troisième chemin, mais depuis la ville 4 tous les tuyaux sont au maximum de leur capacité. Il suffit alors de penser que les tuyaux ont deux sens ! Si faire passer de l'eau dans le sens de la flèche utilise des ressources de capacité de débit, faire passer de l'eau dans l'autre sens libère ces capacités. Nous saturons (S, V_3), (V_3, V_4), (V_1, V_2) et (V_2, C), et nous réduisons la charge de (V_1, V_4). Nous faisons arriver un débit de 4.
D = 8 + 2 + 4.



Il ne reste plus de chemin de la source au consommateur augmentant le débit. L'algorithme est fini. Le débit maximum D que peut espérer le consommateur est de 14 unités.

Par Hervé Lehning

Le pivot de Gauss

La méthode de Gauss est bien adaptée pour la résolution de petits systèmes d'équations. Elle a en outre l'avantage d'être simple à concevoir et à programmer.

Un grand nombre de problèmes se résolvent au moyen d'un système d'équations linéaires. Voyons-en un exemple :

$$\begin{cases} x - y + 2z = 2, \\ 2x + y + 3z = 0, \\ 5x - y + 6z = 6, \end{cases}$$

et il s'agit de trouver toutes les valeurs des triplets $(x ; y ; z)$ vérifiant simultanément ces trois équations.

Calculs « à la main »

Quand on calcule « à la main », on procède par analyse et synthèse, c'est-à-dire que l'on suppose qu'une solution existe. On la note $(x ; y ; z)$ et on combine les trois équations données pour en déduire les valeurs de x, y et z. Par exemple, en ajoutant les deux premières équations puis en retranchant la première à la troisième, nous obtenons les deux équations suivantes en x et z :

$$\begin{cases} 3x + 5z = 2, \\ x + z = 1. \end{cases}$$

En retranchant alors trois fois la première équation à la seconde, nous obtenons l'équation suivante : $2z = -1$, de laquelle on déduit $z = -1 / 2$. En reportant dans la première équation ci-dessus, on tire la valeur de x : $x = 3 / 2$. L'utilisation de la première équation du système donné initialement fournit finalement : $y = -3 / 2$.

Ainsi, le système ne peut avoir qu'une solution : $x = 3/2$, $y = -3/2$ et $z = -1/2$. Une réciproque est nécessaire pour affirmer que ces valeurs fournissent bien une solution du système initial. Celle-ci consiste simplement à s'assurer que les valeurs obtenues vérifient bien toutes les équations du système de départ. On s'en convainc ici aisément.

La méthode du pivot a surtout un intérêt historique.

Systématisation

La méthode que nous avons exposée n'est pas programmable telle quelle. Pour cela, il est nécessaire de la rendre plus systématique. Voyons comment procéder sur un exemple. Les inconnues sont notées x, y, z... Échangeons l'ordre des équations, de façon à placer en tête une équation comportant la première inconnue, x. S'il n'en existe pas, l'inconnue peut tout simplement être supprimée puisqu'elle n'intervient pas dans le système.
Le coefficient de x dans la première équation est alors utilisé comme « pivot » pour « détruire » les coefficients de x dans les autres équations, au moyen de combinaisons linéaires. Sur l'exemple précédent, cela revient à retrancher deux fois la première à la seconde et cinq fois à la troisième, ce qui donne :

$$\begin{cases} x - y + 2z = 2, \\ 3y - z = -4, \\ 4y - 4z = -4. \end{cases}$$

Nous considérons alors les équations à partir de la deuxième. Elles ne contiennent pas de termes en x. On recommence sur elles, avec la deuxième inconnue. Ici, cela donne :

$$\begin{cases} x - y + 2z = 2, \\ 3y - z = -4, \\ -8z = 4. \end{cases}$$

Ce système linéaire triangulaire se résout en commençant par la dernière équation, ce qui donne les mêmes valeurs que précédemment. On démontre de plus que le système obtenu est toujours équivalent au système de départ, ce qui dispense de la réciproque précédente.

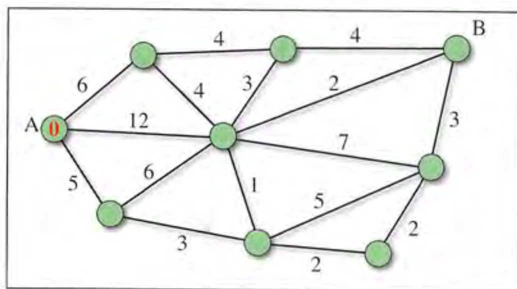
Méthodes modernes

Cette méthode, due à Karl Friedrich Gauss (1777–1855), n'a plus qu'un intérêt historique. Elle n'est pas la plus pratique dans un usage « à la main » car elle peut introduire des coefficients plus compliqués que la méthode usuelle par combinaisons linéaires, vue précédemment. Elle est même dangereuse si le système comporte des paramètres, car elle introduit des cas particuliers factices.
En ce qui concerne l'utilisation d'un ordinateur, elle n'est efficace que pour des systèmes d'au plus cent équations à cent inconnues. Ceci peut sembler énorme, mais les applications des mathématiques (comme en météorologie) conduisent à des systèmes de plusieurs milliers (voire millions) d'équations. On préfère alors des méthode itératives du type « point fixe » comme la méthode de Gauss–Seidel.

H.L.

L'algorithme de Moore–Dijkstra

L'algorithme de Moore–Dijkstra a pour but de déterminer le plus court chemin entre deux sommets d'un graphe valué, c'est-à-dire d'un graphe dont les arêtes ont toutes une longueur (ou une masse) positive. Prenons un exemple pour voir comment fonctionne cet algorithme.

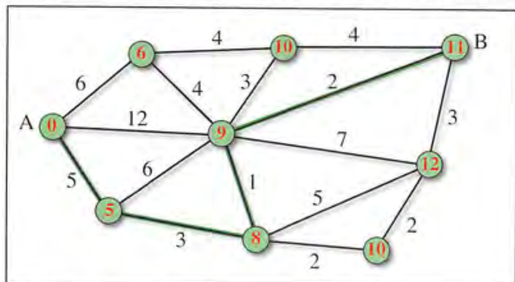


Il s'agit de déterminer le plus court chemin pour aller de A à B dans ce graphe où les temps de parcours sont indiqués sur chaque arête.

L'algorithme fonctionne comme suit :

- On inscrit la valeur 0 en A et la valeur $+\infty$ en chacun des autres sommets ;
- En chaque sommet adjacent à A, on inscrit la valeur de l'arête qui relie ce sommet à A ;
- On choisit un sommet S de valeur minimale ;
- En chacun des sommets S' adjacents à S, on inscrit le minimum entre la valeur qui y est déjà inscrite et la somme minimale obtenue en ajoutant, à la valeur inscrite dans un sommet adjacent à S', la valeur de l'arête qui relie ce sommet à S' ;
- Les deux dernières opérations sont itérées jusqu'à ce que tous les sommets du graphe aient une valeur finie.

On obtient alors le graphe suivant pour notre exemple.



Dans ce graphe, le nombre écrit en chaque sommet est sa distance minimale au point A. Le trajet de longueur minimale entre A et B a été représenté en vert.

E. W. Dijkstra.

J. S. Moore.

L'algorithme de Moore–Dijkstra est attribué conjointement au mathématicien néerlandais **Edsger Wybe Dijkstra** (1930–2002) et au mathématicien anglo-saxon **J. Strother Moore** (université de Texas, Austin). Moore est spécialisé en logique informatique (notion de preuve). Dijkstra était préoccupé par les problèmes de partage des ressources en informatique système (c'est à lui que l'on doit le célèbre problème du dîner des philosophes). On doit à Dijkstra de nombreux aphorismes croustillants concernant l'informatique :

« Tester un programme peut démontrer la présence de bugs, *jamais leur absence. »*

« Se demander si un ordinateur peut penser est aussi intéressant que de se demander si un sous-marin peut nager. »

« La programmation objet est une idée exceptionnellement mauvaise qui ne pouvait naître qu'en Californie. »

« Les progrès ne seront possibles que si nous pouvons réfléchir sur les programmes sans les imaginer comme des morceaux de code exécutable. »

« Autrefois, les physiciens répétaient les expériences de leurs collègues pour se rassurer. Aujourd'hui ils adhèrent à FORTRAN et s'échangent leurs programmes, bugs *inclus. »*

« [À propos des langages] Il est impossible de tailler un crayon avec une hache émoussée. Il est vain d'essayer, à la place, de le faire avec dix haches émoussées. »

« L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des télescopes. »

« Le plus court chemin d'un graphe n'est jamais celui que l'on croit ; il peut surgir de nulle part, et la plupart du temps il n'existe pas. »

Quel jour êtes vous né ?

La détermination du jour de la semaine d'une date donnée se pose (assez) fréquemment.
Voici un algorithme utilisable entre 1900 et 2099. Avec un peu de pratique, on y arrive aisément de tête, au grand étonnement de ses interlocuteurs ! On apprend par cœur le code suivant des mois : 033 614 625 035, avec janvier = 0, février = 3, mars = 3, *etc*. Chaque chiffre représente en fait le décalage, modulo 7, du premier jour de la semaine d'un mois par rapport au même jour du mois de janvier, en partant de 0 pour janvier (par exemple, le 1er février arrive trois jours plus tard dans la semaine que le même jour au mois de janvier).
On additionne :

- les deux derniers chiffres de l'année,
- le quart de ces deux derniers chiffres (tronqué à la virgule si ce n'est pas un entier),
- la date du jour (donc un entier entre 1 et 31),
- le code du mois.

On divise par 7, et le reste donne le jour de la semaine (0 = dimanche, 1 = lundi, *etc*.)
Deux petits ajouts à cet algorithme :

- si la date est après 2000, on soustrait 1 au résultat de l'addition,
- si l'année est bissextile et si la date est avant le 1er mars, on soustrait 1.

Exemples : 10 septembre 2009.
09 + 02 + 10 + 5 = 26, 26 – 1 = 25
(car 2009 > 2000) et 25 mod 7 donne 4, ce qui correspond à un jeudi.
20 février 2008.
08 + 02 + 20 + 3 = 33, 33 – 1 – 1 = 31
(car avant le 1er mars d'une année bissextile et aussi après 2000), 31 mod 7 = 3, ce qui correspond à un mercredi).
19 septembre 1991.
91 + 22 + 19 + 5 = 137, et on conclut rapidement à un jeudi.
Il existe de nombreux algorithmes effectuant ce calcul. Lewis Carroll, Maurice Kraitchik comme l'abbé Christian Zeller (1822–1899) ont proposé leurs formules. C'est celle de ce dernier que nous reproduisons ici (E désigne la fonction partie entière) :
$W = T + E(2{,}6M - 0{,}2) + J + E(J/4) + E(H/4) - 2C$,
où C désigne la centaine d'année (donc le siècle diminué de 1), J l'année dans le siècle considéré, M le mois avec la numération romaine (mars = 1, avril = 2, … février = 12), et T le quantième du mois.

Alors W modulo 7 = W – 7 E(W/7), avec la correspondance 0 pour dimanche, 1 pour lundi…

Combien ai-je de jours ?

Le calcul du nombre de jours entre deux dates est courant dans les comptes bancaires, car il détermine le montant des agios ou des intérêts.
Des tableurs comme Excel attribuent à chaque date un numéro d'ordre depuis 1 pour le 1er janvier 1900.
On a donc la correspondance :

1	**pour dimanche 1 janvier 1900,**
366	**pour dimanche 30 décembre 1900,**
3 650	**pour mardi 28 décembre 1909,**
36 500	**pour lundi 6 décembre 1999,**
40 070	**pour samedi 12 septembre 2009.**

Pour les écarts postérieurs à 1 900, c'est donc facile : il s'agit de la différence des numéros d'ordre des dates considérées. Pour un calcul plus général du nombre de jours entre deux dates, le point délicat se trouve dans le changement de calendrier à la fin du XVIe siècle. Mais il faut aussi faire attention aux « fausses années bissextiles » (1700, 1800, 1900, 2100, 2200, 2300, 2500). Mais il ne s'agit en fait que d'un simple calcul…

Référence
La saga des calendriers ou le frisson millénariste. Jean Lefort, *Belin*, 191 pages, 1999.

par J. Bair et V. Henry

L'algorithme du simplexe

De nombreux problèmes de production consistent à maximiser une fonctionnelle linéaire soumise à des contraintes elles aussi linéaires. Ce cas, très fréquent dans l'industrie, se résoud simplement à l'aide de l'algorithme du simplexe.

De nombreux problèmes concrets se traduisent mathématiquement par la recherche d'une valeur qui rend optimale une fonction dont les variables sont assujetties à des contraintes. Les relations qui lient entre elles les variables peuvent être linéaires : c'est le cas, par exemple, pour des quantités fabriquées qui sont proportionnelles aux quantités de matières premières utilisées. C'est aussi le cas pour les productions de plusieurs ateliers (fabriquant un même article) qui s'ajoutent les unes aux autres. Si, de plus, la fonction à optimiser est elle-même linéaire, c'est-à-dire du premier degré en toutes ses variables, on est en présence d'un problème de *programmation linéaire*. Il s'agit donc de rechercher le maximum (ou le minimum) d'une fonction linéaire, appelée l'*objectif*, sur un ensemble défini par des inéquations linéaires et appelé l'ensemble des *programmes réalisables*.

On a trouvé le plan optimal de production qui assure un profit global maximal.

Le mathématicien George Dantzig (1914–2005) a mis au point l'*algorithme du simplexe* pour résoudre pareils problèmes. L'idée essentielle de cette méthode consiste à observer que l'ensemble des programmes réalisables est un polyèdre convexe (voir l'encadré) : l'algorithme consiste alors à partir d'un sommet de ce polyèdre, puis à passer en un sommet adjacent pour améliorer l'objectif, et à recommencer le processus jusqu'à ce que l'objectif ne puisse plus être amélioré.

Présentons l'algorithme du simplexe en résolvant le problème suivant :

Un industriel souhaite fabriquer deux produits P_1 et P_2 à l'aide de deux matières premières M_1 et M_2. Par expérience, il sait que, pour produire une unité de P_1, il doit exploiter une unité de M_1 et trois unités de M_2. De même, pour produire une unité de P_2,

il doit utiliser trois unités de M_1 et deux unités de M_2. Par ailleurs, il dispose de stocks limités à 12 unités de M_1 et 15 unités de M_2. Sachant qu'il peut réaliser des profits unitaires de 6 unités monétaires sur P_1 et de 5 unités monétaires sur P_2, déterminer son plan optimal de production.

Les inconnues du problème sont les quantités de produits finis à fabriquer : appelons respectivement x_1 et x_2 les nombres d'unités de P_1 et de P_2 à produire. L'objectif consiste à maximiser le profit global z donné par :

$$z = f(x_1\,;\,x_2) = 6x_1 + 5x_2$$

sur l'ensemble E des programmes réalisables défini par les *conditions de non-négativité* $x_1 \geq 0$, $x_2 \geq 0$ et les *vraies contraintes* dues aux limitations des stocks en M_1 et M_2, à savoir respectivement :

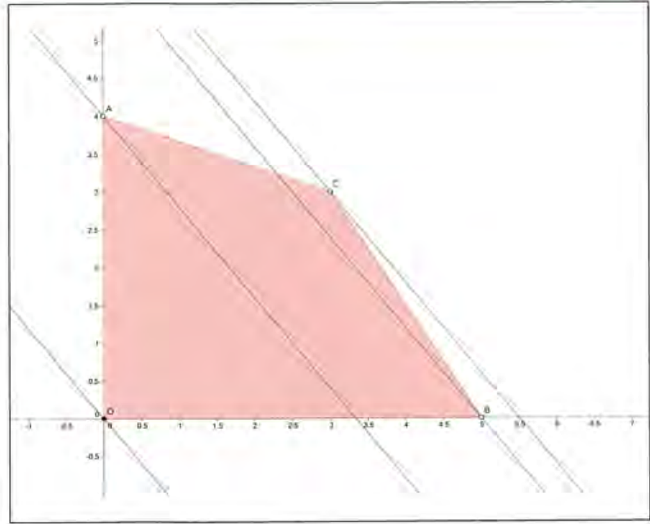
$$x_1 + 3x_2 \leq 12,\ 3x_1 + 2x_2 \leq 15.$$

L'ensemble E est, dans le plan, le polygone convexe dont les sommets sont l'origine O(0 ; 0) ainsi que les points A(0 ; 4), B(5 ; 0) et C(3 ; 3).

Il est facile de voir que le maximum de z est atteint au point C (voir l'article Recherche opérationnelle et optimisation, par Yves Crama, dans le dossier « La recherche opérationnelle » paru dans *Tangente* 126).

La méthode de Dantzig introduit deux nouvelles variables qui représentent l'écart entre les stocks disponibles de matières premières et les quantités de celles-ci réellement utilisées ; elles sont de ce fait appelées *variables d'écart* et sont ici données par :

$$x_3 = 12 - x_1 - 3x_2 \text{ et } x_4 = 15 - 3x_1 - 2x_2.$$



En travaillant avec les quatre variables x_1, x_2, x_3 et x_4, le problème étudié peut se mettre sous la forme *standard* suivante :

maximiser $z = 6x_1 + 5x_2 + 0x_3 + 0x_4$ sous les conditions de non négativité $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$, $x_4 \geq 0$ et les contraintes $x_1 + 3x_2 + x_3 = 12$ et $3x_1 + 2x_2 + x_4 = 15$.

Remarquons que les conditions de non-négativité portent aussi sur les variables d'écart (en raison des inégalités dans les vraies contraintes), et que les vraies contraintes sont devenues des égalités (plus simples à traiter que des inégalités).

On est ainsi confronté au système linéaire suivant :

$$\begin{bmatrix} 1 & 3 & 1 & 0 \\ 3 & 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 12 \\ 15 \end{bmatrix},$$

qui est doublement indéterminé. Les deux variables x_3 et x_4 sont dites *de base*, car elles correspondent aux

vecteurs unitaires dans la matrice des coefficients du système ; elles peuvent s'exprimer en fonction de x_1 et x_2 et si l'on annule ces deux dernières, on obtient une solution, dite *de base*, à savoir $x_1 = x_2 = 0$, $x_3 = 12$, $x_4 = 15$ qui correspond au sommet O de l'ensemble E des programmes réalisables et où l'objectif vaut 0.

Passons à présent du sommet O de E à un sommet adjacent, soit A soit B, de manière à augmenter le plus possible l'objectif. Algébriquement, cela revient à changer une des deux variables de base x_3 ou x_4 en la remplaçant par une variable hors base, x_1 ou x_2. Comme, dans l'objectif z, le coefficient 6 de x_1 est supérieur au coefficient 5 de x_2, on va faire entrer x_1 dans la base. En effet, une augmentation (à partir de la solution de base décrite par le sommet O) d'une unité de P_1 donne un meilleur profit global z qu'une augmentation d'une unité de P_2. x_1 va remplacer dans la base x_4, parce que la restriction $x_1 \leq 5$ résultant des inégalités $x_1 \leq x_1 + 3x_2 \leq 12$ et $3x_1 \leq 3x_1 + 2x_2 \leq 15$ provient de la vraie contrainte ayant introduit la variable d'écart x_4. En sélectionnant donc x_1 et x_3 comme variables de base, le système formé par les contraintes peut s'écrire :

$$x_3 = 7 - \frac{7}{3}x_2 + \frac{1}{3}x_4\,;$$

$$x_1 = 5 - \frac{2}{3}x_2 - \frac{2}{3}x_4,$$

avec pour objectif :
$z = 0x_1 + x_2 + 0x_3 - 2x_4 + 30$.

La solution de base du système, toujours obtenue en annulant les variables hors base (x_2 et x_4 cette fois), est donnée par : $x_1 = 5$, $x_2 = 0$, $x_3 = 7$, $x_4 = 0$. Cela correspond au sommet B du polygone E où l'objectif z vaut 30 (remarquons que la valeur de z en l'autre sommet A adjacent de O vaut 20 et est donc inférieure à celle en B).

Recommençons le processus en retenant cette fois x_1 et x_2 comme variables de base, x_2 remplaçant x_3 dans la base car le coefficient 1 de x_2 dans z est positif. Les contraintes et l'objectif s'écrivent alors respectivement :

$$x_2 = 3 - \frac{3}{7}x_3 + \frac{1}{7}x_4,$$

$$x_1 = 3 + \frac{2}{7}x_3 - \frac{3}{7}x_4 \text{ et}$$

$$z = 0x_1 + 0x_2 - \frac{3}{7}x_3 - \frac{13}{7}x_4 + 33.$$

La solution de base est : $x_1 = x_2 = 3$, $x_3 = x_4 = 0$, ce qui correspond au sommet C où l'objectif z vaut 33.

Le maximum de z sur E est alors atteint car tous les coefficients des variables dans z sont négatifs ou nuls, ou encore parce que le sommet A de E qui est adjacent à C (et non encore choisi) ne peut pas être retenu, puisque l'objectif y prend une valeur inférieure à 33 (en fait, le sommet B avait déjà été préféré à A dans une étape précédente).

Tous ces calculs et ce raisonnement peuvent être réalisés systématiquement dans une disposition en tableaux en partant de l'énoncé sous sa forme standard. Un premier tableau se présente comme suit : une ligne supérieure comporte toutes les variables considérées (x_1, x_2, x_3 et x_4) ainsi que les termes indépendants b_i ; une colonne marginale comprend les variables de base (pour le premier tableau, x_3 et x_4) ainsi que l'objectif z, tandis que le corps même du tableau se compose des coefficients des différentes variables intervenant dans les vraies

contraintes et dans l'objectif (avec, dans la dernière colonne et sur la dernière ligne, le nombre 0 exprimant que l'objectif ne comporte pas de terme indépendant des variables). Cela se présente comme suit :

	x_1	x_2	x_3	x_4	b_i
x_3	1	3	1	0	12 (12/1)
x_4	3	2	0	1	**15 (15/3)**
z	**6**	5	0	0	0

On applique alors la procédure suivante qui consiste à choisir les variables entrant dans et sortant de la base, d'après des critères donnés par Dantzig, puis à effectuer une classique opération de « pivotage » (consistant à changer un seul vecteur de base par substitution) et à recommencer l'opération tant que cela est possible.

Détaillons ces différents points de manière générale, en les illustrant par notre exemple.

1) *Critère d'entrée* : on sélectionne sur la ligne z le terme positif le plus grand : la variable correspondante, soit x_j, va entrer dans la base et sera donc qualifiée d'*entrante*. Sur notre exemple, il s'agit de la variable x_1 (car **6 > 5**).

2) *Critère de sortie* : on divise chaque élément de la dernière colonne des b_i par l'élément de la même ligne et figurant dans la colonne de la variable entrante, puis on retient le plus petit quotient positif ; la variable correspondante, soit x_i, va sortir de la base (et est donc qualifiée de *sortante*) et l'on remplace x_i par x_j dans la première colonne marginale. Sur notre exemple, la variable sortante est x_4, car :

$$\frac{15}{3} < \frac{12}{1}.$$

3) Détermination du *pivot* : l'élément $a_{i\,;\,j}$ qui se trouve à l'intersection de la ligne de la variable sortante x_i et de la colonne de la variable entrante x_j va jouer le rôle de *pivot* pour les calculs ultérieurs : il est encadré. Pour nos données, le pivot est le nombre 3.

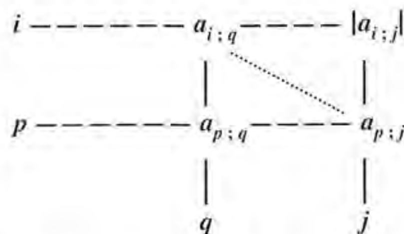
4) Modification de la ligne correspondant à la variable sortante : tous les éléments de cette ligne sont divisés par le pivot.

5) Modification de la colonne correspondant à la variable entrante : les éléments autres que le pivot de cette colonne, y compris le terme sur la ligne des coefficients de z, deviennent nuls.

6) Modification des autres éléments du tableau : tout élément $a_{p\,:\,q}$ du tableau qui n'a pas encore été modifié est remplacé par :

$$a_{p;q} - \frac{a_{p;j}\,a_{i;q}}{a_{i;j}}\,;$$

graphiquement, cette formule revient à utiliser l'*algorithme du quadrille* qui est schématisé par la figure suivante :



Une application des points précédents à notre exemple livre le tableau suivant qui correspond au sommet B du polygone E et où l'objectif z est égal à 30 (notons que cette dernière valeur est égale, au signe près, au nombre figurant sur la dernière ligne et la dernière colonne du tableau) :

Programmation linéaire

Un problème de programmation linéaire se présente, sous forme « canonique », comme suit : Maximiser

$$f(x_1, x_2, \dots, x_n) = c_1x_1 + c_2x_2 + \dots + c_nx_n$$

sous les n conditions de non-négativité :

$$x_1 \geq 0, x_2 \geq 0, \dots, x_n \geq 0$$

et les m vraies contraintes :

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i \text{ pour } i = 1\dots m.$$

L'ensemble E des programmes réalisables, défini par les n conditions de non-négativité et les m vraies contraintes, est en fait l'intersection de $n + m$ demi-espaces fermés de l'espace à n dimensions considéré : il s'agit donc d'un polyèdre convexe (un polygone convexe du plan lorsque $n = 2$) qui possède un nombre fini de sommets. Les coordonnées de chacun des sommets peuvent être calculées en résolvant un système linéaire formé d'équations obtenues en transformant en égalités les inégalités des contraintes (y compris des conditions de non-négativité). Les sommets sont donc en nombre fini. L'enveloppe convexe de tous les sommets redonne E, lorsque cet ensemble est borné (c'est-à-dire ne contient aucune demi-droite) ; l'enveloppe convexe de $n + 1$ sommets affinement indépendants fournit un *simplexe*. Deux sommets sont dits *adjacents* lorsqu'ils sont reliés par une arête.

	x_1	x_2	x_3	X_4	b_i
x_3	0	7/3	1	–1/3	**7 [7/(7/3)]**
x_1	1	2/3	0	1/3	5 [5/(2/3)]
z	0	**1**	0	–2	–30

7) Une nouvelle application des points 1) à 6) à ce nouveau tableau conduit à celui-ci :

	x_1	x_2	x_3	x_4	b_i
x_2	0	1	3/7	–1/7	3
x_1	1	0	–2/7	3/7	3
z	0	0	–3/7	–13/7	–33

8) ***Critère d'arrêt*** **:** il est impossible d'appliquer les critères (d'entrée et de sortie) de Dantzig à ce dernier tableau, puisque tous les nombres situés sur la dernière ligne sont négatifs ou nuls. On a ainsi trouvé le plan optimal de production consistant à fabriquer trois unités de chacun des deux produits (la valeur 3 pour x_2 et pour x_1 pouvant être trouvée sur la dernière colonne du dernier tableau, en première ou deuxième ligne respectivement), ce qui assure le profit global maximal de 33 unités monétaires (le nombre 33 étant l'opposé de l'élément situé dans le coin sud-est du dernier tableau).

Cette méthode itérative peut évidemment être appliquée à d'autres données, éventuellement avec plus de variables et plus de vraies contraintes.

J. B. et V. H.

L'algorithme de Sluse

René de Sluse

Le Belge **René-François Walter de Sluse** (1622–1685) étudie en Italie la théologie, la philosophie, l'histoire, les mathématiques, la physique, l'astronomie ainsi que diverses langues, et rencontre des savants réputés de l'époque (Cavalieri, Torricelli...). Il exerce ensuite diverses fonctions au sein de l'Église de la principauté de Liège. Il entretient une correspondance soutenue, avec le Français Pascal, le Hollandais Huygens, l'Italien Ricci, les Anglais Wallis et Oldenbourg (secrétaire de la *Royal Society* de Londres, dont Sluse devient membre en 1674)... C'est grâce à cette correspondance, publiée en 1884 par le mathématicien liégeois Constantin Le Paige (1882–1929), que ses travaux scientifiques nous sont connus.

Les perles de Sluse

Les courbes définies par une équation du type

$$y^n = k(a - x)^p x^m,$$

où a et k désignent des constantes réelles et n, p et m des entiers positifs, sont appelées des *perles de Sluse*. Sluse se pencha sur le problème des tangentes à ces courbes. Il semble être le premier mathématicien à fournir un algorithme général pour trouver la tangente à de telles courbes.

Sa méthode fournit en fait la valeur de la *sous-tangente* en un point P d'une courbe C : dans un repère orthonormal, si la tangente t à C en $P(x\,;y)$ et la droite verticale menée par ce même point P coupent l'axe des abscisses aux points T et M respectivement, la sous-tangente v est définie par la mesure algébrique du segment $[TM]$.

La connaissance de v permet de déterminer T, et donc de caractériser t par deux de ses points (P et T).

Prenons l'exemple d'une quartique piriforme d'équation $y^2 = x^3(1 - x)$. L'algorithme peut se décomposer en cinq étapes.

1. Poser $f(y) = y^2$ et $g(x) = x^3(1 - x) = x^3 - x^4$.

2. Construire l'expression N en partant de $f(y)$ et en y multipliant chaque terme par l'exposant correspondant de y ; ici $N = 2y^2$.

3. Construire l'expression D en partant de $g(x)$ et en y multipliant chaque terme par l'exposant correspondant de x, puis en le divisant par x ; ici
$$D = (3x^3 - 4x^4)/x = 3x^2 - 4x^3.$$

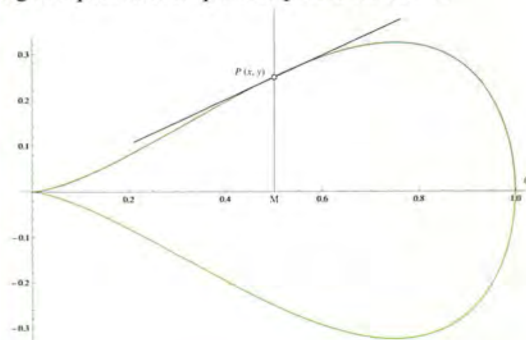
4. Diviser N par D et obtenir ainsi la sous-tangente v ; pour notre quartique,
$$v = N/D = 2y^2/(3x^2 - 4x^3).$$

5. Le coefficient directeur m de la tangente t au point $P(x\,;y)$ est obtenu en divisant y par v ; pour notre « poire »,
$$m = y/[2y^2/(3x^2 - 4x^3)] = (3x^2 - 4x^3)/2y.$$

Cette égalité est uniquement valable lorsque :

$$0 < x < 1.$$

Cet algorithme, créé bien avant l'arrivée du calcul différentiel, peut être adapté pour des courbes algébriques autres que des perles de Sluse.



par Michel Criton

La tour d'Hanoï

La tour d'Hanoï est un casse-tête inventé et popularisé par le mathématicien français Édouard Lucas (1842–1891) dans le but de présenter la numération binaire à un large public. C'est aujourd'hui l'occasion d'expliquer un algorithme de résolution d'un problème ludique.

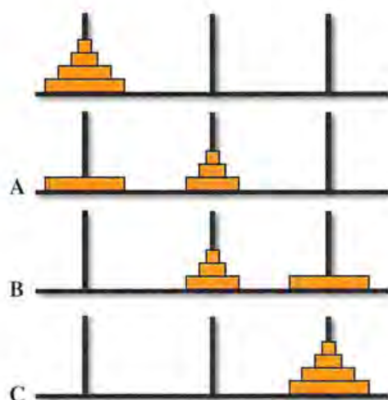
Édouard Lucas.

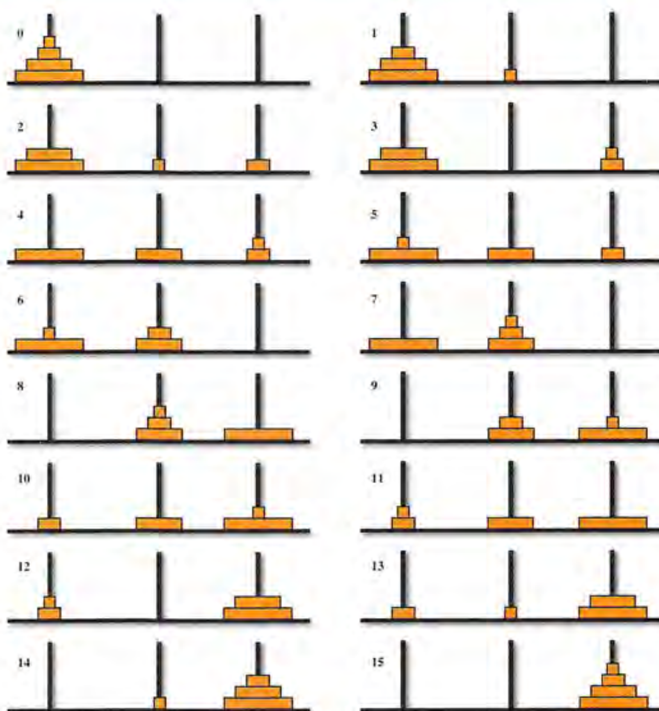
Lorsque Édouard Lucas fait la présentation de son jeu, il écrit : *« Un de nos amis, le professeur N. Claus (de Siam) mandarin du collège de Li-Sou-Stian, a publié, à la fin de l'année dernière, un jeu inédit qu'il a appelé la Tour d'Hanoï, véritable casse-tête annamite qu'il n'a pas rapporté du Tonkin, quoi qu'en dise le prospectus. Cette tour se compose d'étages superposés et décroissants, en nombre variable, représentés par huit pions en bois percés à leur centre, enfilés dans l'un des trois clous fixés sur une tablette. Le jeu consiste à déplacer la tour en enfilant les pions sur un des deux autres clous et en ne déplaçant qu'un seul étage à la fois, mais avec défense expresse de poser un étage sur un étage plus petit. Le jeu est toujours possible et demande deux fois plus de temps chaque fois que l'on ajoute un étage à la tour… »*

Le nom prétendu de l'inventeur du jeu, *« N. Claus de Siam, mandarin de Li-Sou-Stian »*, est tout simplement l'anagramme de « Lucas d'Amiens, professeur au lycée Saint-Louis ». Lucas aimait agrémenter ses récréations de pointes d'humour et, à l'époque, un casse-tête, s'il n'était chinois, ne pouvait être qu'asiatique. Selon Lucas, N. Claus de Siam préparait la publication des écrits du mandarin *Fer-Fer Tam-Tam* (Lucas avait fondé le projet de publier les œuvres de Fermat). Il rapporte également la légende d'une tour d'Hanoï située à Bénarès et comportant soixante-quatre disques. Édouard Lucas écrit que,

lorsque les $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$ mouvements nécessaires au transport des soixante-quatre disques auront été effectués, « *les brahmes tomberont et ce sera la fin du monde !* ».

La tour d'Hanoï est longtemps restée un divertissement exotique destiné à montrer l'utilité du système binaire. Mais ce jeu a connu une nouvelle jeunesse avec l'avènement de l'informatique. Tous les programmeurs en herbe ont rencontré ce casse-tête comme un exemple typique de situation où l'on peut faire appel à la récursivité.





Résolution complète d'une tour d'Hanoï de quatre disques. Cette résolution nécessite $2^4 - 1$ soit 15 mouvements. Elle passe par la réalisation d'une tour d'un disque en un mouvement (étape 1), de deux disques en trois mouvements (étape 3) et de trois disques en sept mouvements (étape 7).

En effet, pour résoudre le jeu avec n disques, on commence par le résoudre avec $n - 1$ disques, en faisant comme si le plus grand disque n'existait pas (étape A de la figure). On transporte ensuite le grand disque sur la tige libre (étape B). Il ne reste plus qu'à résoudre à nouveau le casse-tête avec $n - 1$ disques en transportant la tour de hauteur $n - 1$ sur le plus grand disque (étape C). D'où le doublement, à un près, du nombre d'étapes nécessaires lorsque l'on passe de $n - 1$ disques à n disques.

Mais pour résoudre le jeu avec $n - 1$ disques, il faut d'abord le résoudre avec $n - 2$ disques en ignorant le plus grand des $n - 1$ disques... En continuant ce raisonnement, on est amené finalement à résoudre en premier le jeu à un seul disque, ce que l'on sait faire, bien évidemment !

M. C.

Références

Récréations Mathématiques. Édouard Lucas, tome 3, réédition Blanchard, 210 pages, 1979.

Drôles de Maths, Tutti frutti d'énigmes. Michel Criton et l'Association pour le développement de la culture scientifique, Vuibert, 187 pages, 2008.

 par Édouard Thomas

Le problème de Syracuse

Un algorithme célèbre circule depuis soixante ans dans les milieux mathématiques. Il est à l'origine de la *conjecture de Collatz*, du nom du mathématicien allemand Lothar Collatz (1910–1990), encore appelée *problème* (ou *conjecture*) *de Syracuse*, du nom d'une université américaine qui a contribué à le faire connaître, *problème 3x + 1*, ou *conjecture d'Ulam*, du nom de Stanislas Ulam (1909–1984), qui l'a longtemps étudiée.

– Prendre un entier naturel quelconque (non nul). S'il est pair, le diviser par deux. S'il est impair, le tripler, puis ajouter un.
– Recommencer avec le nombre obtenu.
– Répéter ce procédé. Qu'observe-t-on ?

Voyons sur un exemple : 18 → 9 → 28 → 14 → 7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1 → 4 ...

Nous tombons dans le cycle 4 → 2 → 1, qui se répète indéfiniment.

À l'aide de puissants ordinateurs, tous les entiers jusqu'à plus de 5.10^{18} ont été testés. Tous vérifient cette propriété. De nombreuses variantes de l'algorithme ont été étudiées, parfois avec succès (remplacer 3 par un autre nombre premier, ajouter ou soustraire un autre nombre que 1…) mais personne n'a jamais pu démontrer que, en partant d'un entier quelconque dans le problème de Syracuse, on tombe dans le cycle 4 → 2 → 1.

Les suites aliquotes

Une *suite aliquote* est définie par un algorithme récursif. Le premier terme est un entier naturel non nul. Si un terme est différent de 1, alors la somme de ses diviseurs propres constitue le terme suivant. Comme 1 ne possède aucun diviseur propre, la suite s'arrête dès que l'un de ses termes est égal à 1.
Par exemple, en partant de 14 dont les diviseurs propres sont 1, 2 et 7, on obtient successivement : 14 → 10 → 8 → 7 → 1.

On remarque que l'image d'un nombre premier est toujours égale à 1, et donc mène à l'arrêt de l'algorithme. À l'inverse, un nombre tel que 6 (ou 28), qui est égal à la somme de ses diviseurs propres, est invariant. Un tel nombre est *parfait*. On ignore s'il existe une infinité de nombres parfaits ; on ignore s'il existe un nombre parfait impair. De manière plus générale, la suite rentre dans une boucle (devient périodique à partir d'un certain rang) si l'un de ses termes est un nombre amical ou sociable. Ainsi, la somme des diviseurs propres de 220 est 284, dont la somme des diviseurs propres est 220 (nombres *amicaux*). De même, 12 496 conduit au cycle 14 288, 15 472, 14 536, 14 264 et 12 496 (nombre *sociable* d'ordre 5).

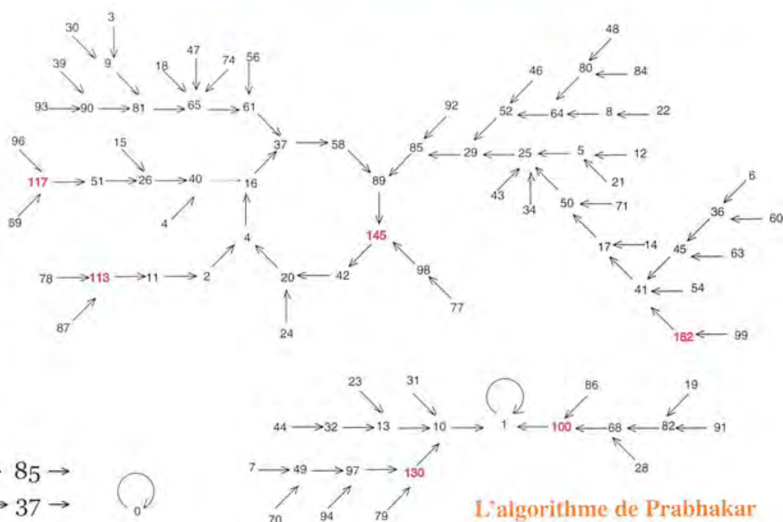
Le comportement général des suites aliquotes (arrêt, suite constante à partir d'un certain rang, cycle) n'est toujours pas connu à ce jour. Comment évolue la suite aliquote de 276 ? Est-il possible qu'il existe des suites aliquotes de longueur infinie ?

Pour plus d'information : Jean-Paul Delahaye, *Les inattendus mathématiques*, Bibliothèque Pour la Science, Belin, Paris, 256 pages, 2004.

par Michel Criton

L'algorithme de Prabhakar

– Prendre un nombre quelconque, par exemple 162.
– Calculer la somme des carrés de ses chiffres : $1^2 + 6^2 + 2^2 = 41$.
– Recommencer avec le nouveau nombre obtenu : $4^2 + 1^2 = 17$.
– Répéter la procédure.
On obtient la suite suivante :
162 → 41 → 17 → 50 → 25 → 29 → 85 → **89** → 145 → 42 → 20 → 4 → 16 → 37 → 58 → **89** → 145.
Nous venons de découvrir un cycle.



L'algorithme de Prabhakar pour tous les nombres à deux chiffres.

Le diagramme ci-contre représente les images de tous les nombres de 0 à 100. Les nombres de trois chiffres qui apparaissent comme images de nombres à deux chiffres ont été portés en rouge. On constate qu'outre le cycle décrit plus haut, il existe deux puits : 0 et 1. On peut en fait montrer qu'à partir de tout nombre non nul, on finit par aboutir soit au puits 1, soit au cycle 16 → 37 → 58 → 89 → 145 → 42 → 20 → 4.

Variantes à étudier :
L'algorithme de Prabhakar cubique : à chaque nombre, on associe la somme des cubes de ses chiffres. Trouver les puits de cet algorithme sera facile pour les familiers du Championnat des jeux mathématiques et logiques, puisque cela constituait un des problèmes du second Championnat (tricubes).
Considérer la somme des puissances énièmes des chiffres (n fixé).
Considérer le carré de la somme des chiffres, ou, plus généralement, la puissance énième de la somme des chiffres.
Considérer le produit des chiffres, augmenté de la somme des chiffres.

L'algorithme de Kaprekar

L'algorithme de Kaprekar doit son nom au mathématicien indien Dattatreya Ramachandra Kaprekar (1905–1988), qui le publia en 1949.
– Prendre un nombre en base 10 (par exemple $n = 4\,742$).
– Réordonner ses chiffres du plus grand au plus petit ($n_1 = 7\,442$), puis du plus petit au plus grand ($n_2 = 2\,447$). Dans ce dernier cas, l'écriture d'un nombre peut commencer par un ou plusieurs zéros. Soustraire n_2 de n_1 : 7 442 – 2 447 = 4 995.
– Recommencer ce procédé à partir du nouveau nombre obtenu :
9 954 – 4 599 = 5 355.
– Répéter le procédé.
On obtient ici la suite 5 355 → 4 742 → 4 995 → 5 355 → 1 998 → 8 082 → 8 532 → 6 174.

Le nombre qui suit 6 174 sera encore 6 174, « puits » dont l'algorithme ne peut plus sortir. En base 10, on démontre que l'algorithme de Kaprekar aboutit soit à 0 (si les chiffres du nombre choisi sont tous identiques), soit à un nombre invariant (un *nombre de Kaprekar*, tel 6 174, 495 ou encore 549 945), soit à un cycle (tel 53 955 → 59 994 → 53 955…).

Comment explorer un labyrinthe ?

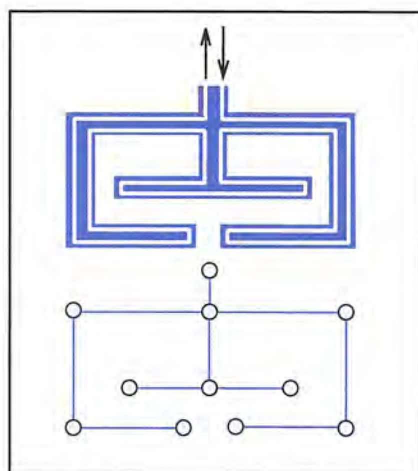
L'exploration systématique d'un labyrinthe, nécessaire pour en trouver une issue, nécessite d'utiliser une méthode rigoureuse. Différents algorithmes d'exploration ont été imaginés au cours des siècles.

Gaston Tarry.

Dans la jungle des labyrinthes se trouve le labyrinthe bidimensionnel possédant une et une seule entrée. Pour ne pas vous y perdre, il est recommandé d'explorer les différentes façons qui vous permettront d'en sortir.

Une première méthode d'exploration fonctionne lorsque l'ensemble des murs est connexe (d'un seul tenant), mais seulement dans ces cas-là. Elle consiste à se déplacer en longeant constamment le mur de gauche (ou celui de droite) dont on vérifie la continuité à l'aide de la main. Le graphe associé à un tel labyrinthe est alors un arbre, c'est-à-dire un graphe ne comportant aucun circuit. Hélas, cette méthode ne permet pas d'explorer des parties entières de labyrinthes dont l'ensemble des murs comporte des « îlots ».

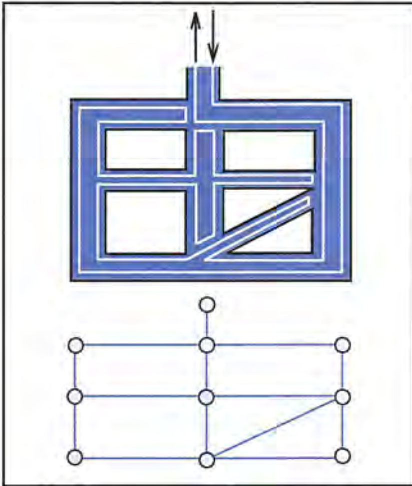
Les labyrinthes présentant des îlots, dont l'ensemble des murs est non-connexe, nécessitent des méthodes d'exploration plus sophistiquées, qui ont été mises au point vers la fin du XIXe siècle.



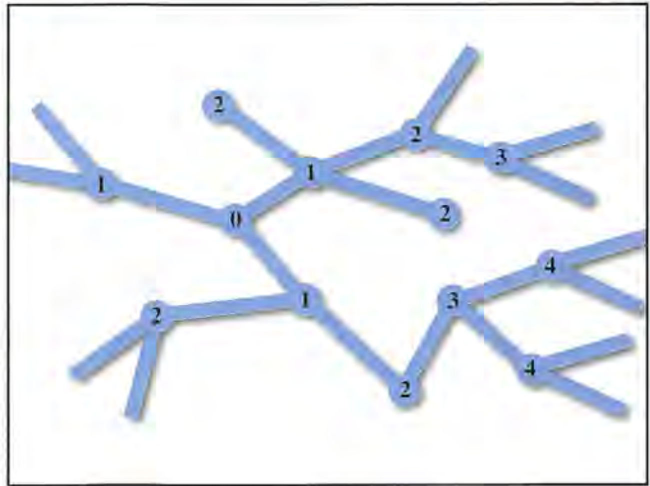
En haut, un labyrinthe sans îlot. En bas, son graphe est un arbre.

La méthode de Gaston Tarry

À la même époque, M. Trémaux, ingénieur des télégraphes, avait communiqué au mathématicien français Édouard Lucas une méthode similaire,



En haut, un labyrinthe avec des îlots.
En bas, son graphe comporte des circuits.



Mise en œuvre de la méthode d'Oystein Ore.

mais plus précise. Cette méthode s'expose en trois règles :

- on emprunte une voie quelconque. Si on aboutit à une impasse, on revient sur ses pas. Si on aboutit à un carrefour, on emprunte une voie quelconque non encore explorée ;
- si on arrive à un carrefour déjà exploré, par une voie nouvelle, on revient sur ses pas, ce qui revient à condamner cette voie ;
- si on arrive à un carrefour déjà exploré par une voie déjà parcourue dans l'autre sens, on choisit en priorité une voie nouvelle, ou sinon une voie parcourue dans un seul sens.

Les méthodes de Tarry et de Trémaux permettent de parcourir un labyrinthe à partir de son point d'entrée. Elles pourraient évidemment être appliquées à partir de n'importe quel point du labyrinthe.

La méthode de Oystein Ore

Une dernière méthode due au mathématicien norvégien Oystein Ore (1899–1968) permet de trouver la sortie d'un labyrinthe dans lequel on est perdu, en connaissant à tout moment la distance entre l'endroit où l'on se trouve et le point de départ. Elle consiste à marquer les carrefours atteints à partir du point de départ, supposé être un carrefour auquel on a attribué le numéro 0. On explore chacune des galeries partant de ce carrefour 0, jusqu'à tomber sur un cul de sac ou un autre carrefour qui sera alors numéroté 1. On rebrousse ensuite chemin en marquant la galerie aux deux extrémités, et en ajoutant une croix lorsqu'il s'agit d'une impasse. L'étape suivant consiste à explorer tous les couloirs partant de chacun des carrefours 1, excepté celui relié au carrefour 0, jusqu'à atteindre chacun des carrefours 2, situés à deux couloirs du carrefour 0, puis on revient au carrefour 0, toujours en marquant chaque couloir deux fois, à l'entrée et à la sortie...

Le labyrinthe de la cathédrale de Chartres (XIIe siècle).

Cette méthode, très longue à mettre en œuvre, permet d'explorer l'intégralité du labyrinthe, en s'éloignant régulièrement du carrefour 0, et en sachant à chaque instant à quelle « distance » on se trouve par rapport à ce carrefour de départ.

M. C.

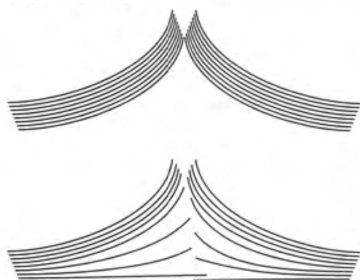
Références
D. Souder et M. Criton : Le fil d'Ariane, *Tangente* n° 57, 1997
Tangente Hors-Série n°12 Les graphes.

par M. Criton et A. Zalmanski

Tours de cartes

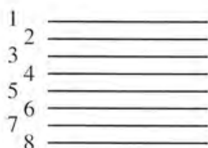
Battre les cartes

Dans un tour de cartes automatique, on n'a en fait aucune liberté, hormis celle d'appliquer des règles déterminées à l'avance. L'opération « battre un jeu de cartes », qui consiste à mélanger les cartes du jeu de telle sorte que, en les prenant ensuite une par une, elles apparaissent comme rangées de manière aléatoire, peut être complètement déterminée à l'avance, en suivant un algorithme, et avec un peu de pratique. Il existe différentes manières de battre les cartes, qui reposent presque toujours sur une « coupe » préalable du jeu en deux tas, que l'on réarrange ensuite. Un tel « réarrangement » peut ne pas faire intervenir le hasard, comme dans l'exemple représenté ci-dessous, où chaque coupe partage le jeu en deux tas de même effectif.

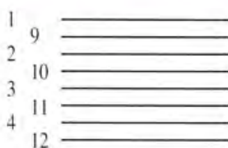


Un mélange des cartes.

La première carte à partir du bas vient du paquet de droite, puis les cartes viennent alternativement d'un paquet et de l'autre.



Effet du mélange appliqué deux fois : l'ordre 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 devient 1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16, puis 1 5 9 13 2 6 10 14 3 7 11 15 4 8 12 16.



Un joli tour de cartes

Prendre 21 cartes d'un jeu quelconque. Les distribuer en trois colonnes de 7 cartes. Penser – ou faire penser – à une carte. Noter simplement la colonne dans laquelle elle se trouve (exemple : colonne 1). Rassembler en gardant l'ordre des cartes chaque colonne et placer la colonne contenant la carte pensée en sandwich entre les deux autres colonnes (ici, mettre la colonne 1 entre la colonne 2 et la colonne 3). Recommencer l'opération deux fois.
Ces trois itérations amènent à l'invariance de la onzième carte, ce qui signifie que la carte pensée (par vous ou par votre « victime ») se situe exactement au milieu (à la quatrième rangée) de la deuxième colonne. Pour plus de clarté, il est utile de réaliser effectivement l'expérience avec un vrai jeu de cartes.
Peut-on obtenir un tel schéma avec $3n$ cartes, n étant supérieur à 7 ?

Solutions en page 149.

par Alain Zalmanski

Royaux problèmes

Le problème des huit reines

Il s'agit de placer sur un échiquier classique huit dames (ou reines) telles qu'aucune ne puisse être prise par l'une des sept autres. Il doit donc y avoir au plus une dame dans chaque ligne, dans chaque colonne et dans chaque diagonale. Combien existe-t-il de solutions ?
Ce problème a été posé pour la première fois en 1848 dans un journal d'échecs par Max Bezzel (1824–1871), puis dans l'hebdomadaire illustré Illustrierten Zeitung du 1er juin 1850 par Franz Nauck.

Cette publication donna lieu à un engouement extraordinaire. Dans le même hebdomadaire, le 21 septembre 1850, le Dr. Nauck donna les 92 solutions. Carl Friedrich Gauss n'en trouva « que » 72.

La solution proposée ci-contre est représentée par la permutation (3 ; 5 ; 2 ; 8 ; 1 ; 7 ; 4 ; 6). C'est sous cette forme qu'un ordinateur déterminera toutes les solutions.

8								
7								
6								
5								
4								
3								
2								
1								
	1	2	3	4	5	6	7	8

Échiquiers de taille supérieure

Le *problème des n reines* (ou *dames*) consiste à placer n dames n'étant pas en prise mutuelle sur un échiquier de taille n^2. Le problème admet des solutions dès que n est supérieur à 4. Voici un algorithme pour trouver rapidement une solution :

- Calculer $r = n$ modulo 12.
- Écrire les nombres pairs par ordre croissant de 2 à n.
- Si $r = 3$ ou 9, mettre le 2 à la fin de la liste.
- Écrire ensuite, par ordre croissant, les nombres impairs de 1 à n, mais si $r = 8$, alors permuter les nombres impairs deux par deux.
- Si $r = 2$, permuter les places de 1 et 3, puis mettre 5 à la fin de la liste.
- Si $r = 3$ ou 9, mettre 1 puis 3 en fin de liste.
- La liste obtenue est une solution au problème des n dames.

Exemple avec $n = 15$ ($r = 3$) : une solution est (4 ; 6 ; 8 ; 10 ; 12 ; 14 ; 2 ; 5 ; 7 ; 9 ; 11 ; 13 ; 15 ; 1 ; 3).

Détermination des solutions

Il faut qu'il y ait une dame et une seule dans chaque ligne et dans chaque colonne.
Positionnons donc les dames colonne par colonne, en commençant par la première colonne. Nous y plaçons une première reine. Dans la deuxième colonne, nous faisons glisser la deuxième dame du bas vers le haut jusqu'à la première case libre de toute menace. Puis la troisième dame est placée selon le même critère dans la troisième colonne et ainsi de suite. Si une dame ne peut pas être placée de cette manière parce qu'aucune case de sa colonne n'est libre, on revient à la dame précédente, que l'on remonte jusqu'à une autre case libre de toute menace des dames la précédant, *etc.*
Chaque solution est notée, puis la dernière dame est enlevée de l'échiquier. L'avant-dernière reine est remontée jusqu'à la prochaine case libre de toute menace. Un peu de réflexion montre qu'on obtient ainsi toutes les solutions (même la première dame va parcourir toute sa colonne par une suite de conséquences rétrogrades).
Il y aura lieu de supprimer les solutions qui se déduisent d'une solution déjà trouvée par rotation ou par symétrie. Le lemme de Burnside permet d'établir qu'il n'existe en fait que 12 solutions de base au problème. Ainsi, avec les notations introduites, on visualisera aisément les huit solutions équivalentes suivantes :
(1 ; 5 ; 8 ; 6 ; 3 ; 7 ; 2 ; 4) = (1 ; 7 ; 5 ; 8 ; 2 ; 4 ; 6 ; 3) = (8 ; 2 ; 4 ; 1 ; 7 ; 5 ; 3 ; 6) = (8 ; 4 ; 1 ; 3 ; 6 ; 2 ; 7 ; 5) = (5 ; 7 ; 2 ; 6 ; 3 ; 1 ; 4 ; 8) = (3 ; 6 ; 4 ; 2 ; 8 ; 5 ; 7 ; 1) = (4 ; 2 ; 7 ; 3 ; 6 ; 8 ; 5 ; 1) = (6 ; 3 ; 5 ; 7 ; 1 ; 4 ; 2 ; 8).
C'est ainsi que dans un algorithme optimisé de recherche on restreint la dame de la première colonne aux seules positions 1, 2 et 3, les autres solutions découlant par symétrie ou rotation.

 par Michel Criton

Le jeu de la vie

John Horton Conway (mathématicien britannique né en 1937), croqué par l'un de ses collègues.

Les règles du jeu

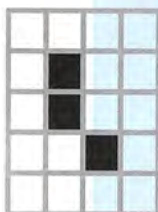
Le jeu de la vie est un jeu déterministe qui fonctionne sur un quadrillage régulier infini formé de carrés unitaires pavant le plan. Des créatures vont naître, vivre et mourir dans ces cellules. Le temps dans lequel vivent ces créatures est *discret* (on passe de l'instant 0 à l'instant 1, puis à l'instant 2, et ceci sans aucune transition).

Le quadrillage est constitué de cellules qui peuvent être habitées par des objets (à raison d'au plus un de ces objets dans une cellule). Chaque cellule est entourée par exactement huit cellules voisines. Une cellule habitée est dite *vivante* (elle est coloriée en noir). Une cellule vide est dite *morte* (elle n'est pas coloriée).

Si, à l'instant n, une cellule est vivante, elle le reste à l'instant $n + 1$ si et seulement si deux ou trois des cellules voisines sont vivantes. Dans tous les autres cas, la cellule meurt à l'instant suivant.

Si, à l'instant n, une cellule n'est pas vivante, elle s'anime (ou elle naît) si et seulement si exactement trois des cellules voisines sont vivantes. Dans les autres cas, la cellule demeure vide.

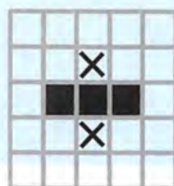
Quelques exemples

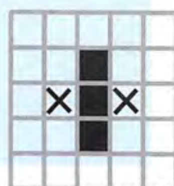


Prenons quelques exemples d'évolution de populations dans le jeu de la vie.
Dans le premier exemple, on part d'une population de trois cellules.

Deux de ces cellules ont un seul voisin, la troisième en a deux. Toutes les trois vont donc mourir. Mais une cellule vide possède trois voisins (avant leur mort). Cette cellule va donc voir une naissance. On notera que les morts et les naissances se produisent simultanément, et que, juste avant de mourir, une cellule peut participer à la naissance d'une autre. Une cellule seule ne peut survivre. À l'instant 2, notre population sera donc éteinte.

Prenons maintenant l'exemple de trois cellules formant un rectangle. On constate que le rectangle va être alternativement horizontal puis vertical. Nous avons affaire à une transformation de période 2 : la population va prendre deux états. On l'appelle parfois un *pulsar*.





Des mots parfaits

Sur une idée d'Éric Angelini.
Avec l'aimable concours de Jean-Marc Falcoz.
On affecte à chaque lettre un code correspondant à sa place dans l'alphabet : A vaut 1, B vaut 2… et Z vaut 26. Pour un mot donné, on affiche d'une part son code alphabétique concaténé en (a), d'autre part son code additionné, ou *gématrie*, en (b).

On cherche, dans un dictionnaire donné, les mots MOTS tels que MOTS(a) soit divisible par MOTS(b) dans un dictionnaire donné. Par un rapprochement naturel avec les nombres divisibles par la somme de leurs diviseurs propres, on appellera *mots parfaits* les mots dont les codes concaténés sont divisibles par l'addition de ces mêmes codes.

Voyons quelques exemples :
ALAIN(a) = 1121914 = 1 121 914.
ALAIN(b) = 1 + 12 + 1 + 9 + 14 = 37.
1 121 914 est divisible par 37 : ALAIN est parfait.

LYON(a) = 12251514 = 12 251 514.
LYON(b) = 66.
12 251 514 est divisible par 66 : LYON est parfait.

De même, FOIX est parfait, ARIEGE est parfait. C'est le seul couple (préfecture ; département) ayant cette propriété !
On imagine aisément les extensions (voir page 98) : mots « amicaux », « sociables »…

Des trous noirs troublants

Nous vous proposons quelques petits exercices (pas toujours faciles) sur des algorithmes très simples.

1) Choisir un nombre entier positif non nul. Écrire la liste de ses diviseurs, y compris lui-même et l'unité. Additionner tous les *chiffres* qui figurent dans la liste. Réitérer avec la somme ainsi obtenue.

Exemple : les diviseurs de 18 sont 18, 9, 6, 3, 2 et 1, ce qui fournit 1 + 8 + 9 + 6 + 3 + 2 + 1, soit 30, nouveau nombre à traiter.

Existe-t-il un *trou noir*, c'est-à-dire un nombre vers lequel « plongent » tous les autres ?

2) Écrire littéralement un nombre. Compter son nombre de lettres, incluant les espaces et les traits d'union. Recommencer avec le nombre obtenu.

Exemple : 421 s'écrit « quatre cent vingt-et-un », composé de 23 lettres, ce qui fournit « vingt-trois », de 11 lettres, *etc.*
Là encore, existe-t-il un trou noir ?
À votre avis, la prise en compte conventionnelle des espaces et traits d'union change-t-elle la donne ?
Que peut-il se passer pour d'autres langues ?

Jeux algorithmiques

Exemples de problèmes

Voici déjà deux questions sur le jeu de la vie : comment les deux tétraminos ci-dessous vont-ils évoluer ? (Solutions en page 149.)

Voici maintenant, sur l'exemple du *glisseur*, un autre type de population : le motif revient périodiquement à sa forme initiale, mais en se déplaçant (en glissant) selon un vecteur bien défini.

Premier tétramino A

Second tétramino B

L'étude du jeu de la vie de Conway peut conduire à de multiples questions, parfois difficiles :

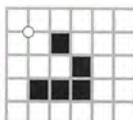
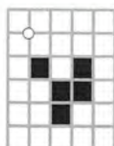
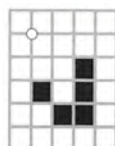
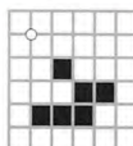
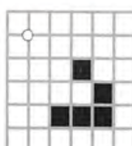
Trouver des *jardins d'Éden*, ou populations sans antécédent (c'est-à-dire des populations qui ne peuvent découler d'aucune population antérieure).

Trouver des populations de période 2 (comme le pulsar vu précédemment), de période 3, 4, 5…

Explorer d'autres règles de « survie », de « mort » et de « naissance » des cellules.

Explorer d'autres type de pavage du plan : triangulaire, hexagonal.

Imaginer une généralisation du jeu de la vie à l'espace tridimensionnel.

Autour des palindromes et de la convergence

Vous trouverez ci-dessous quelques petits problèmes sur des variantes d'algorithmes présentés dans ce numéro.

• On prend un nombre de trois chiffres non identiques (exemple : 592). On le « renverse » (le palindrome de 592 est 295), et on soustrait du plus grand le plus petit, qui peut commencer par 0 (ici : 592 − 295 = 297). Le résultat $100a + 10b + c$ est « renversé » (ici 792) et on effectue l'addition du résultat et de son palindrome (ici 792 + 297 = 1 089).

La succession de ces deux opérations conduit-elle toujours à 1089 ?

Peut-on généraliser avec un nombre de départ de quatre ou cinq chiffres, tous différents ?

Aboutira-t-on à une constante, et en combien d'opérations élémentaires (additions et soustractions) ?

• On prend un nombre de 4 chiffres et on lui ajoute son palindrome. Si le résultat est un palindrome, c'est gagné. Sinon, on recommence avec ce nouveau nombre. Exemple : 5 214 a pour palindrome 4 125. La somme des deux, 9 339, est un palindrome.

Obtient-on toujours en fin de compte un palindrome ?

• On considère les entiers naturels comme des chaînes de caractères dont on note, côte à côte et dans cet ordre, le nombre de chiffres pairs, le nombre de chiffres impairs et le total du nombre de chiffres.

« 9288759111111 » conduit par exemple à 3, 10 et 13 qui donnera comme nouvelle chaîne à traiter « 31013 ».

À quel(s) nombre(s) ou chiffre(s) conduit cette série d'itérations, appelée parfois *chaîne de Sisyphe* ?

Trouverez-vous une justification mathématique de cette étonnante convergence ?

Que peut-il se passer pour d'autres langues ?

Complexité et temps d'exécution p. 108
Veni, divisi, vici p. 112
Les algorithmes de tri p. 116
La programmation structurée p. 122
La magie de la récursivité p. 126
Itération et point fixe p. 130
La gloutonnerie appliquée à la compression p. 136
Codes correcteurs d'erreurs p. 142
La multiplication rapide p. 150

LIMITES ET PERFORMANCES

Ce n'est pas tout de savoir structurer un raisonnement répétitif à l'aide d'un algorithme. Il faut aussi prouver son efficacité et chercher à minimiser le nombre d'opérations qui vont intervenir dans son exécution, même si c'est l'ordinateur qui doit les effectuer.

par Hervé Lehning

Complexité et temps d'exécution

L'exécution d'un algorithme prend un certain temps, et occupe de l'espace. L'expérience nous le montre au quotidien. La théorie le prouve. On parle de complexité, un modèle mathématique du temps de calcul.

Certains algorithmes sont efficaces, d'autres non. Comment mesurer cette efficacité ? Quels sont les problèmes pour lesquels il existe une méthode efficace ? L'expérience est utile pour répondre à ces questions *a posteriori*. Mais un modèle mathématique du temps de calcul est indispensable pour y répondre *a priori*.

Complexité des algorithmes

Pour tester si un jeu de n cartes est trié, on peut les prendre l'une après l'autre et vérifier que la nouvelle carte est de valeur supérieure à la précédente. Cet algorithme a un temps d'exécution proportionnel au nombre de cartes n. Arrêtons-nous un instant sur l'expression *temps d'exécution*. En toute rigueur, elle est incorrecte et dépend de notre vitesse. Dans le cas où nous utilisons un ordinateur, le « temps d'exécution » dépend de ce dernier, et des systèmes d'exploitation disponibles. Parmi ces derniers, certains sont particulièrement inefficaces et quelques surprises sont possibles ! En fait, le « temps d'exécution » se rapporte ici à un modèle mathématique. Un tel modèle porte le nom de *complexité*, même si ce terme, il est vrai, évoque mal ce qu'il recouvre.

La complexité est un modèle du temps d'exécution.

Comparaison des complexités

La procédure décrite précédemment permet de concevoir un algorithme de tri du jeu de cartes. Il suffit de créer toutes les permutations possibles des cartes, puis de les tester l'une après l'autre. La complexité de cet algorithme est au moins proportionnelle au

nombre de permutations, c'est-à-dire à la factorielle de n, ce qui est énorme. Même s'il fonctionne bien pour de petites valeurs de n, il ne peut trier un jeu de 32 cartes. Cela prendrait trop de temps. Pourquoi peut-on l'affirmer *a priori* ? Imaginez qu'une permutation soit créée en une nanoseconde. Les créer toutes prendrait 32! nanosecondes. Une nanoseconde vaut 10^{-9} seconde, donc environ 10^{-9} divisé par 60 . 60 . 24 . 365,25 année. Comme 32! vaut approximativement $2{,}6 \cdot 10^{35}$, l'algorithme s'exécuterait en $8{,}2 \cdot 10^{18}$ années, soit des milliards de milliards d'années.

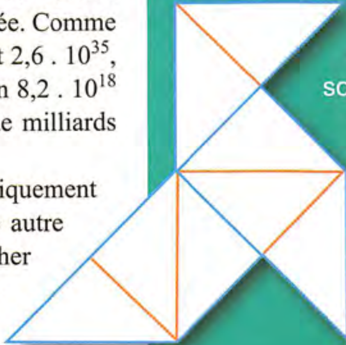
L'algorithme, bien que logiquement correct, est inefficace. Une autre méthode de tri est de chercher la plus faible carte en parcourant le jeu en entier (on effectue ainsi n opérations). On met cette carte de côté et on recommence avec le reste du jeu, et ainsi de suite. La complexité de ce nouvel algorithme est proportionnelle à la somme des entiers de 1 à n, donc au carré de n, ce qui est raisonnable. Cet algorithme de tri est efficace.

Complexité polynomiale

De façon générale, un algorithme fournit un résultat à partir de données dont on mesure la taille par un entier n. Dans le cas du tri, n est le nombre d'éléments à trier. Dans le cas de la factorisation d'un nombre entier N, n est son nombre de chiffres. Dans celui du problème du voyageur de commerce qui veut préparer son voyage, il s'agit du nombre de villes à visiter. Nous considérons alors, pour un algorithme donné, la complexité maximale T(n) pour les données de taille n. L'algorithme est dit *de complexité*

Aplatissement des origamis

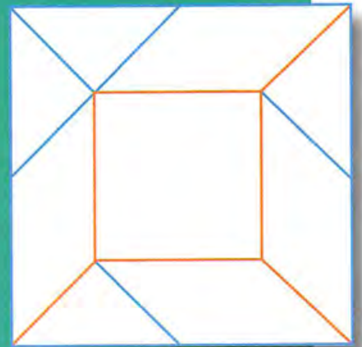
Fabriquer une cocotte en papier n'est pas aussi facile que les personnes non versées dans cet art pourraient le croire. Considérons par exemple la cocotte ci-dessous.



La cocotte traditionnelle en France.

Si nous la déplions, nous obtenons les lignes de pliage ci-dessous. Chaque ligne s'inscrit en bosse ou en creux, ce que nous avons indiqué sur la figure en traçant les lignes en bleu (bosse) ou en gris (creux). Si vous n'avez jamais fabriqué de cocotte en papier, vous aurez bien du mal à en reconstituer une à partir de ces indications (il faut au moins trouver dans quel ordre effectuer les pliages).

L'art de plier des carrés de papier nous viendrait du Japon, où les objets ainsi obtenus portent le nom d'origamis (ori : plier, gami : papier). Ces sculptures sont de deux types : celles que l'on peut aplatir entre les pages d'un livre sans les endommager (comme notre cocotte), et les autres (comme les bombes à eaux).



Dépliage de la cocotte en papier.

Des scientifiques se sont intéressés à l'important problème suivant : comment prédire, à partir de ses lignes de pliage, qu'un origami est « aplatissable » ? En 1995, deux jeunes informaticiens américains, Marshall Bern et Barry Hayes, ont montré que ce problème est NP-complet.

La complexité spatiale

La complexité envisagée dans cet article concerne le temps de calcul. On parle de complexité temporelle. L'étude de la place mémoire nécessaire à l'exécution d'un algorithme se nomme la *complexité spatiale*. Après avoir eu une grande importance dans les débuts de l'informatique, elle est de nos jours négligée, car la mémoire est devenue bon marché.

polynomiale s'il existe un nombre entier positif k et un nombre positif A tel que $T(n) \leq A\,n^k$. Il est admis que seuls ces algorithmes sont utilisables dans la pratique.

Un problème pour lequel on peut trouver un algorithme en temps polynomial est dit P. Remarquez que la question n'est pas de savoir si un tel algorithme existe effectivement à l'heure actuelle, mais de savoir s'il est théoriquement possible d'en écrire un. Le tri est donc un problème P.

Un exemple classique de problème où l'on ne connaît pas d'algorithme en temps polynomial est celui du voyageur de commerce. Celui-ci doit visiter n villes différentes reliées par un certain nombre de routes. Le problème est de minimiser la distance parcourue. Si chaque couple de villes est relié par une route, cela fait $n!$ possibilités. L'algorithme naïf qui consiste à dresser la liste de tous les trajets possibles n'est donc pas de complexité polynomiale. En existe-t-il un ? Nul ne le sait à l'heure actuelle.

Stephen Cook, né en 1939, père des problèmes NP-complets.

Problèmes NP-complets

Le type de machine que nous avons utilisé pour notre tri est dit *déterministe* : après chaque action en vient une autre bien déterminée. C'est le modèle des machines utilisées en informatique. Un problème est NP s'il existe une machine non déterministe le résolvant en temps polynomial. Qu'est-ce qu'une *machine non déterministe* ? N'en cherchez pas dans votre supermarché : il s'agit d'un outil théorique. Décrire une telle machine est abstrait. Il est cependant possible de comprendre pourquoi le problème du voyageur de commerce est NP sans rentrer dans ces détails : imaginez disposer d'une infinité d'ordinateurs identiques. À chacun, nous faisons calculer la distance d'une route possible. Chaque résultat est obtenu en temps polynomial. Si un superordinateur peut alors trouver le plus petit de ces nombres instantanément, le temps total reste polynomial. En première approximation, on peut se représenter notre machine non déterministe comme l'assemblage impossible décrit ci-dessus. L'intérêt de ces machines imaginaires est que l'on sait montrer qu'un grand nombre de problèmes (difficiles) sont NP alors qu'on ne sait pas s'ils sont P. La question fondamentale est de savoir si P = NP. On a de bonnes raisons d'en douter. Cependant le problème reste ouvert, et doté d'un prix d'un million de dollars par la fondation Clay.

Les *problèmes NP-complets* sont ceux qui ne peuvent être résolus en temps polynomial que si P = NP. Le problème du voyageur de commerce est un problème NP-complet mais il en existe un grand nombre d'autres, comme celui de l'aplatissement des origamis (voir l'encadré « aplatissement des origamis »). Trouver un algorithme polynomial pour un problème NP-complet reviendrait donc à établir que P = NP.

H. L.

Pierre Bézier (1910–1999)

Après des études au Conservatoire national des arts et métiers, **Pierre Bézier** fit toute sa carrière à la Régie Renault, jusqu'au poste de directeur des méthodes mécaniques.

À la fin des années 1950, les carrosseries de voitures ne correspondaient plus à des courbes mathématiques « classiques ». Mais pour usiner une pièce, il fallait rentrer un programme dans une machine-outil sous la forme d'un codage utilisant des arcs de cercle, des morceaux d'ellipses et de bouts de paraboles. Pierre Bézier inventa alors un moyen de manipuler les courbes et les surfaces gauches au moyen de points caractéristiques.

Pour fabriquer des pièces telles qu'une carrosserie de voiture, on traçait à l'époque des schémas à main levée, sans modélisation mathématique des contours. Les approximations et les erreurs étaient nombreuses… Pierre Bézier imagina alors une façon de traduire mathématiquement les propriétés de courbes, puis de surfaces, utilisées chez Renault. Le résultat devait être exploitable par un non mathématicien à l'aide des premières machines à commande numérique.

Les courbes de Bézier sont un moyen élégant de joindre des points. Les concepteurs peuvent maintenant oser toutes les courbes possibles, il leur suffit de placer plusieurs points, puis de les relier avec une courbe de Bézier. Ces travaux ont donné naissance au système de codage de document PDF, développé par la société Adobe.

Voir *Tangente* 125, page 24, Les lobes de Bézier, pour plus d'informations.

Les courbes de Bézier sont reliées aux polynômes de Bernstein.

Un *polynôme de Bernstein* est une fonction de la forme suivante (n est un entier fixé, l'entier i varie entre 0 et n) :

$$B_n^i(t) = C_n^i t^i (1-t)^{n-i}, \text{ avec } C_n^i = \frac{n!}{i!(n-i)!}.$$

Par exemple, $B_3^0 = (1-t)^3$, $B_3^1 = 3t(1-t)^2$, $B_3^2 = 3t^2(1-t)^1$, $B_3^3 = t^3$.

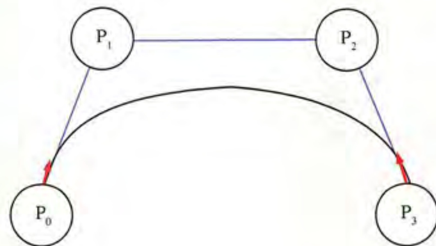
On considère maintenant un réel t entre 0 et 1 et $n + 1$ points du plan $P_0(x_0 ; y_0)… P_n(x_n ; y_n)$. La *courbe de Bézier* associée à ces points vérifie

$$\begin{cases} x(t) = \sum_{i=0}^{n} B_n^i(t) x_i, \\ y(t) = \sum_{i=0}^{n} B_n^i(t) y_i. \end{cases}$$

On écrit simplement $P(t) = \sum_0^n B_n^i(t) P_i$.

Les points P_i sont les *points de contrôle*.

Par exemple, une courbe de Bézier avec quatre points de contrôle correspond à la figure suivante.



par Hervé Lehning

Veni, divisi, vici

Parmi les stratégies usuelles pour concevoir des algorithmes, la méthode la plus troublante, fondée sur l'idée de récursivité, fait penser à celle qu'employa le général Bonaparte pour détruire les armées adverses pendant la campagne d'Italie.

Imaginez ! Je vous tends un jeu de cartes constitué d'une seule carte et vous demande : « Pouvez-vous trier ce paquet de cartes ? » Sans doute penserez-vous que je suis dérangé. Le jeu est déjà trié ! Pourtant cette idée est à la base de plusieurs méthodes de tri, détaillées dans l'article page 36.

Admettons que nous sachions trier un jeu de N cartes, ce qui est vrai si $N = 1$. Comment trier un jeu de $N + 1$ cartes ? Prenez votre jeu de $N + 1$ cartes, écartez la dernière. Il vous reste un jeu de N cartes, que vous savez trier. *Triez-le* ! Vous obtenez un jeu trié de N cartes d'un côté, et une carte isolée de l'autre. L'idée pouvait sembler perturbante, elle débouche sur une méthode concrète, le *tri par insertion*. En divisant le jeu en deux, on en obtient une autre, nommée *tri-fusion*.

« Lorsque, avec de moindres forces, j'étais en présence d'une grande armée, groupant avec rapidité la mienne, je tombais comme la foudre sur l'une de ses ailes et je la culbutais. Je profitais ainsi du désordre que cette manière ne manquait jamais de mettre dans l'armée ennemie, pour l'attaquer dans une autre partie, toujours avec mes forces. Je la battais ainsi en détail, et la victoire qui en était le résultat était toujours, comme vous le voyez, le triomphe du grand nombre sur le petit.»

Napoléon Bonaparte,
Lettres au Directoire pendant la campagne d'Italie.

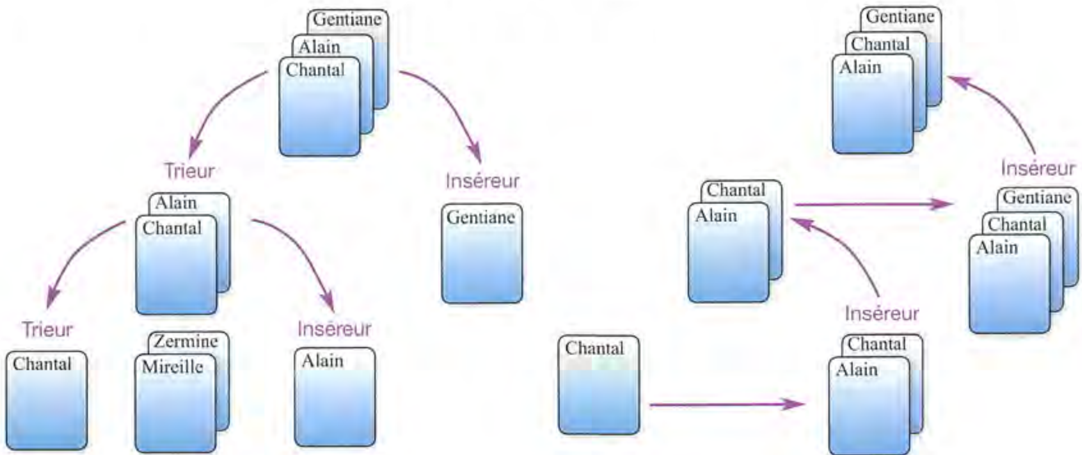
Cette stratégie de création d'algorithmes porte le nom de *divide and conquer* aux États-Unis ou *Veni, divisi, vici*. En français, on la traduit souvent par *Diviser pour régner*, ce qui ne rend pas le sens de conquête d'une solution. L'idée essentielle est la division en deux parties pour lesquelles on suppose le problème résolu. De tels algorithmes sont *récursifs* : leur écriture fait appel à eux-mêmes.

Délégation puis exécution des taches

La méthode (du tri par insertion, par exemple) vous semble factice ? Imaginez que chaque fonction soit accomplie par une personne. Il faut un trieur et plusieurs inséreurs. Suivons l'algorithme pas à pas sur la figure ci-dessous, où nous trions seulement trois cartes.

Les méthodes itératives peuvent sembler plus naturelles, mais les méthodes récursives sont plus faciles à prouver.

Le trieur reçoit trois cartes (Chantal ; Alain ; Gentiane). Il donne la troisième (Gentiane) au premier inséreur, qui ne fait rien car il n'a pas de jeu où l'insérer. Le trieur n'a plus que deux cartes (Chantal ; Alain). Il donne la seconde (Alain) au deuxième inséreur. Il ne lui reste plus qu'une carte (Chantal), seul cas où il sache trier un jeu. Il le donne donc tel quel au deuxième inséreur, qui y insère la carte qu'il a reçue (Alain) et rend le jeu trié (Alain ; Chantal) au trieur. Celui-ci le donne au premier inséreur, qui y insère la carte qu'il a reçue (Gentiane) et rend le jeu trié (Alain ; Chantal ; Gentiane) au trieur. Le jeu est ainsi trié !

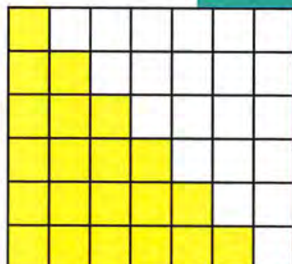


À gauche, série de délégations des tâches pour trier un jeu de cartes. Le travail ne commence effectivement qu'au moment où le trieur n'a rien à faire, c'est-à-dire a un jeu d'une seule carte dans les mains.
À droite, l'exécution de ces tâches est à lire de bas en haut, en suivant les flèches.

La division par deux donne l'idée d'une méthode de tri plus rapide que le tri par insertion, le tri par fusion. Prenez un jeu de cartes, divisez-le en deux. Supposez que chacun est trié. Ne protestez pas ! L'idée des méthodes *Veni, divisi, vici* est là ! Vous avez donc deux jeux triés, comment en faire un jeu trié ? Vous pre-

Une mesure de complexité

Pour mesurer *a priori* le temps de calcul du tri par insertion, on compte le nombre de comparaisons de cartes effectuées dans le cas le plus défavorable. Pour l'insertion d'une carte parmi *N* cartes, on est amené à la comparer à *N* cartes dans le pire des cas.



La complexité de l'insertion est donc proportionnelle à *N*. Pour le tri, on opère une insertion parmi une, deux, *etc.* jusqu'à *N* cartes. On obtient donc un nombre proportionnel à la somme des *n* premiers entiers. Ce nombre peut être vu comme un escalier, en jaune sur la figure.

En retournant cet escalier sur lui-même (en bleu), on obtient un rectangle de largeur *N* + 1 et de hauteur *n*. Ce nombre est donc égal à la moitié de *N* (*N* + 1). La complexité de l'algorithme du tri par insertion est quadratique, puisque pour *N* assez grand, *N* (*N* + 1) est pratiquement égal au carré de *N*.

nez les deux jeux et choisissez la plus petite carte parmi celles qui sont au-dessus. Vous la retirez et la mettez de côté, puis vous recommencez. La nouvelle plus petite va derrière la première sélectionnée, et ainsi de suite. Au terme de ce processus de fusion, le jeu total est trié.

Indispensables la preuve et le calcul de complexité

De même que le tri par insertion, le tri-fusion se justifie par récurrence. Pourquoi prouver qu'un algorithme fonctionne ? Pour éviter les erreurs bien entendu ! Lors du premier lancement de la fusée Ariane V, une faute de programmation a coûté plus d'un milliard d'euros. À ce prix, on pourrait penser à une erreur d'une grande subtilité. Pas du tout, on avait simplement réutilisé un programme fonctionnant très bien sur Ariane IV, en l'adaptant un peu. L'ennui est que la plage des valeurs possibles d'un certain paramètre était modifiée. Tout fonctionnait en dessous d'un certain seuil, respecté dans le cas d'Ariane IV mais pas dans celui d'Ariane V. Au-dessus, rien n'allait plus et la fusée a explosé pour cette raison. L'aéronautique n'est pas le seul domaine où les logiciels doivent être absolument dépourvus d'erreur. C'est le cas également dans les questions touchant la sécurité dans les centrales nucléaires, les avions, les trains ou même le métro. Le secteur bancaire a également besoin d'une sécurité maximale. Que dire d'une carte bancaire qui pourrait parfois se tromper ?

Pour obtenir des logiciels exempts d'erreur, il ne suffit pas de les tester. Des cas particuliers inattendus sont toujours possibles. Il est nécessaire de prouver que le logiciel fonctionne dans tous les cas. Pour cela, une preuve de type mathématique est indispensable.

Une fois que l'on a prouvé un programme, il est nécessaire d'étudier son temps de calcul, sa *complexité*. Le mot est mal choisi, mais il s'agit d'un modèle mathématique du temps de calcul (voir l'article sur la question en page 108). On ne peut se contenter d'utiliser un chronomètre !

Supposons, par exemple, qu'en mesurant le temps nécessaire pour trier un jeu de mille cartes, nous trouvions une milliseconde.
Combien de temps faut-il pour en trier cent mille ou un million ?

La réponse dépend de la complexité. Si l'algorithme est linéaire, c'est-à-dire si sa complexité est proportionnelle à la taille des données, la réponse est du même ordre, soit un dixième de seconde et une seconde.
La complexité de l'algorithme du tri par insertion est quadratique, c'est-à-dire proportionnelle au carré de la taille des données. Les réponses sont donc dix mille ou un million de fois plus longtemps : dix secondes, et environ trois heures pour chacun des deux cas.
La complexité de la fusion, elle, est linéaire, du même type que celle de la multiplication rapide (voir l'article page 150), car il suffit de passer en revue les deux jeux triés.

H. L.

Calcul d'une puissance

On peut appliquer la stratégie *Veni, divisi, vici* pour écrire une procédure calculant une puissance, que ce soit celle d'un nombre ou d'une matrice. Le but est donc de calculer X^N où N est un nombre entier naturel et X un « objet algébrique » passible de multiplications.
Si $N = 0$, on définit $X^N = 1$ sous réserve que X soit différent de 0.
Si N est pair, il s'écrit $2\,Q$ (avec $Q < N$) et $X^N = (X^Q)^2$.
Si N est impair, il s'écrit $2\,Q + 1$ (avec $Q < N$) et $X^N = (X^Q)^2 \,.\, X$.

Cette idée fournit la procédure récursive suivante :

Fonction Puissance, arguments X, N
Si $N = 0$ alors Puiss := 1 sinon
    Diviser N par 2 (quotient Q, reste R), Calculer S := Puissance(Q)
    Si $R = 0$ alors Puiss := S^2 sinon Puiss := $S^2 \,.\, X$.
Résultat : Puiss

Cette procédure se prouve aisément par récurrence. Quant à sa complexité, elle est logarithmique.

par Hervé Lehning

Les algorithmes de tri

En algorithmique, un problème classique consiste à trier des listes de valeurs, par exemple dans l'ordre croissant. Mais on n'imagine pas la richesse et la diversité des méthodes qui conduisent à ce résultat. Un point commun néanmoins : l'utilisation du principe de récurrence.

Partons d'une liste dont les éléments sont rangés n'importe comment. On veut obtenir, à la fin, une liste dont les éléments sont rangés du plus petit au plus grand. Comment procéder ? Le premier algorithme auquel nous allons nous intéresser est un processus très simple, que nous appellerons *le tri par sélection*.

Le tri par sélection

Par souci d'efficacité, nous allons présenter cet algorithme sur un exemple. Soit à trier la liste suivante : **8 3 10 6 5**.

Les meilleures façons de trier reposent sur le principe de récurrence : pour trier n + 1 cartes, il suffit de savoir en trier n et de recommencer !

On recherche dans la liste le plus petit élément : 3.

On l'échange avec le premier élément de la liste, on obtient la liste : **3 8 10 6 5**.

On considère la liste à partir du 2e élément, son plus petit élément est : 5.

On l'échange avec le deuxième élément de la liste, on obtient la liste : **3 5 10 6 8**.

On considère la liste à partir du 3e élément, son plus petit élément est : 6.

On l'échange avec le troisième élément de la liste, on obtient la liste : **3 5 6 10 8**.

On considère la liste à partir du 4e élément, son plus petit élément est : 8.

On l'échange avec le quatrième élément de la liste, on obtient la liste : **3 5 6 8 10**.

C'est fini.

On peut s'entrainer sur d'autres exemples à faire fonctionner l'algorithme. Pour cela, utilisons des tableaux où sont coloriées en bleu les cases au fur et à mesure que les valeurs de la liste occupent leur place définitive.

14	8	10
8	14	10
8	10	14

7	9	-3	-1
-3	9	7	-1
-3	-1	7	9

Lorsque la liste à trier est de grande dimension (c'est-à-dire lorsqu'elle contient par exemple 1 000 valeurs), le traitement manuel de la liste avec cet algorithme peut vite se révéler rébarbatif (et c'est un euphémisme). Il peut en être de même si l'on doit traiter plusieurs listes.
Pour y remédier, la programmation informatique prendra le relais (voir en encadré le programme réalisé avec le logiciel Python) mais, même au stade de l'ordinateur, l'économie en place mémoire et en temps d'exécution sera peu probante. D'autres méthodes s'imposent !

Le tri par insertion

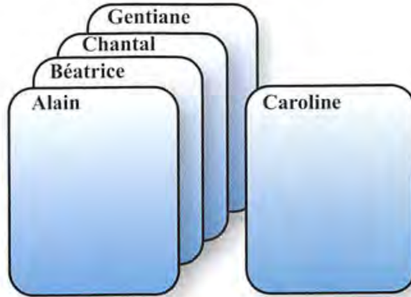
Sans le dire ouvertement, l'algorithme précédent utilisait déjà un principe de récurrence.
Les méthodes suivantes vont formaliser cette approche qui s'avère simplificatrice.
On partait en effet du problème suivant :

Prenez un jeu de cartes, mélangez-le puis essayez de le trier. Comment faire ?

A priori, la question est complexe. Le principe de récurrence permet de la simplifier considérablement en la posant autrement :

Imaginez que vous sachiez trier n cartes, comment en trier $n + 1$?

Plus prosaïquement, vous savez trier un jeu de quatre cartes. On vous en donne une de plus, que faites-vous ?



Pour trier cinq cartes, il suffit de trier les quatre premières, puis d'insérer la cinquième dans le jeu trié.

La preuve de l'algorithme

Le tri par insertion donne le bon résultat si le jeu T est réduit à une seule carte. Supposons qu'il fonctionne correctement pour des jeux d'au plus n cartes et donnons-nous un jeu T de $n + 1$ cartes. Exécutons l'algorithme de tri par insertion. Il nous demande de trier T ', ce qui s'effectue correctement d'après l'hypothèse de récurrence car T ' contient n cartes. L'insertion étant ensuite correcte, le résultat de Tri(T) est correct pour un jeu de $n + 1$ cartes. Par récurrence, nous avons prouvé que le résultat de Tri(T) est correct pour tout jeu. On peut opérer de la même façon pour prouver la correction du tri par fusion.

Tri par sélection : le programme Python

par J.-A. R.

```
>>> def tri(liste):
        n=len(liste)
        i=0
        while i!=n:
            j=i+1
            while j!=n:
                if liste[j]<liste[i]:
                    m=liste[i]
                    liste[i]=liste[j]
                    liste[j]=m
                j=j+1
            i=i+1
        return liste
```

On définit la fonction tri appliquée à une liste
L'entier n est la longueur de la liste
Au départ, l'entier i vaut 0
Tant que l'on n'a pas atteint la fin de liste
On définit l'entier j
Tant que j est différent de n
Si l'élément de rang j est < à celui de rang i, on permute les deux éléments
On incrémente j de 1
On incrémente i de 1
Le programme donne la nouvelle liste

```
Execution du programme :
>>> tri([35,-6,12,0,10,34,67])
[-6, 0, 10, 12, 34, 35, 67]
```

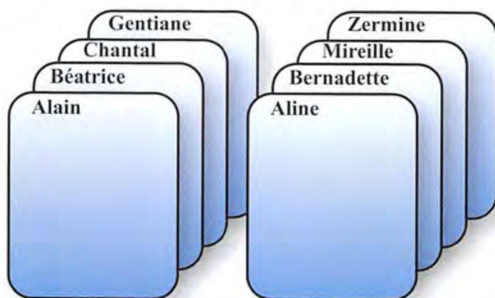
On applique la fonction tri à la liste [35,-6,12,0,10,34,67]
On obtient le résultat escompté (heureusement !).

Bien entendu, vous triez les quatre premières puis insérez la cinquième à sa place. Pour cela, il suffit de la comparer successivement avec les cartes déjà triées en commençant par la première. Ici, la nouvelle carte s'insère après la deuxième. Cette méthode repose exactement sur l'idée de récurrence. Pour vous en convaincre, nous la formalisons dans le style des démonstrations par récurrence.

Voici donc une méthode qui, pour la donnée d'un jeu de cartes T, renvoie le même jeu trié :

Tri (T) :
Si le nombre n de cartes est égal à 1, renvoyer T (car le jeu est trié), sinon priver le jeu T de son dernier élément x de façon à obtenir un jeu T' et Insérer (x, Tri (T'))
où Insérer (x, Tri (T')) insère une carte dans un jeu trié.

Comment être sûr que cet algorithme fonctionne quel que soit le jeu de cartes donné ? La réponse tient encore dans le



Comment trier un jeu composé de deux parties triées ?

principe de récurrence (voir l'encadré La preuve de l'algorithme).

Diviser pour se reposer : le tri par fusion

Améliorons notre algorithme avec un principe souvent adopté en informatique : diviser par deux. Pour résoudre un problème pour n, on suppose qu'on sait le résoudre pour n / 2 (où cette expression désigne le quotient entier de la division de n par 2, ainsi, « 5 / 2 = 2 »).
Prenons un exemple. Imaginons que vous vouliez classer huit fiches comportant chacune un nom. Une idée est de couper le jeu en deux jeux de quatre fiches et de donner ces deux jeux à classer à deux amis. Admettons qu'ils vous renvoient les deux jeux triés ci-contre.
Le travail vous est fort simplifié. Il suffit de commencer par prendre la première carte des deux jeux, ici celle du premier jeu puis de recommencer. Cette fois-ci ce sera le tour de la première du second jeu et ainsi de suite. Après huit comparaisons, les deux jeux sont fusionnés en un seul jeu trié. Bien entendu, notre méthode repose sur une hypothèse : nos deux amis renvoient les deux jeux triés. Comment peuvent-ils faire ? La réponse semble absurde : de la même façon, chacun avec deux amis. Ceux-ci vont se trouver avec deux cartes dans les mains. Quoi de plus simple à trier ? Ou bien, elles sont dans l'ordre. Ou bien, on les permute pour les y mettre.
Notre méthode repose encore sur l'idée de récurrence. Pour vous en convaincre, nous la formalisons dans le style des démonstrations par récurrence.
Voici donc l'algorithme qui, pour la donnée d'un jeu de cartes T, renvoie le même jeu trié :

Tri(T) :
Si le nombre n de cartes est égal à 1,
renvoyer T (car le jeu est trié), sinon
diviser le jeu en deux jeux T_1 et T_2 (le premier a n / 2 cartes)
renvoyer Fusion(Tri(T_1), Tri(T_2))

où Fusion(T_1, T_2) fusionne deux jeux triés en un jeu trié suivant la méthode que nous avons décrite auparavant.

Le principe de récurrence permet encore de prouver que cet algorithme fonctionne quelque soit le jeu de cartes donné.

H. L.

Un autre algorithme : le tri par dénombrement

Pour trier un jeu de cartes constitué de nombres dont on sait qu'ils sont compris entre 1 et n, on peut utiliser n cases mémoires mises à zéro au préalable. On dénombre alors les occurrences de chaque nombre en passant le jeu en revue. Il suffit ensuite de le réécrire en utilisant le dénombrement trouvé. Le temps d'exécution est proportionnel à n, la mémoire nécessaire aussi.

Insérer, c'est longuet, diviser, c'est gagner

La méthode de fusion présente une économie de fatigue pour la personne qui l'utilise effectivement comme nous l'avons présentée. Elle se repose en fait sur ses deux amis.
Qu'en est-il quand la méthode est utilisée par un ordinateur ? Peut-on espérer un gain quelconque ? Nous allons examiner le problème du point de vue du temps, en comparant les temps d'exécution des deux algorithmes précédents.

Par insertion

Supposons que, pour trier n cartes, le temps d'exécution du tri par insertion soit au plus $t(n)$. Pour en trier $n + 1$, le temps est alors $t(n)$ plus le temps de l'insertion. Celle-ci demande de comparer l'élément à insérer avec les éléments du reste du jeu jusqu'à trouver l'endroit d'insertion. Dans le pire des cas, il faut donc n comparaisons. Si le temps d'une comparaison est c, nous avons donc la relation :
$t(n + 1) = t(n) + n\,c$.
D'autre part, $t(1) = 0$. On démontre assez facilement que :
$$t(n) = \frac{n(n-1)}{2}c\ .$$
Autrement dit, dans le pire des cas, le temps nécessaire pour trier n cartes est proportionnel au carré de n. Toutes les méthodes *naïves* de tri se comportent de même et fournissent des temps proportionnels au carré de n.

Par fusion

Reprenons notre calcul de temps dans le cas du tri par fusion en suivant le même raisonnement. La fusion des deux jeux prends le temps $n\,c$ d'où la relation :

$$t(n) = 2\,t(n\,/\,2) + n\,c.$$

On en déduit, par récurrence, que :

$$0{,}5\ \ n\,(\log_2 n - 1)\,c \leq t(n) \leq 2\,n\,(\log_2 n + 1)\,c$$

où $\log_2$ désigne le logarithme de base 2. Le temps $t(n)$ est donc proportionnel à $n \log_2 n$ ce qui est peu supérieur à n et beaucoup plus petit que n^2. Le gain de temps est considérable.

De façon générale, on démontre que ce temps en $n \log_2 n$ est minimal pour les tris fondés sur des comparaisons des cartes deux à deux. Il est possible de le minimiser en procédant autrement. Il s'agit du *tri par dénombrement.* L'inconvénient est alors un coût en mémoire prohibitif.

par Florent Picard

L'algorithme de Casteljau

Paul de Casteljau

Paul de Faget de Casteljau est un mathématicien français né en 1930. Il était chercheur dans l'entreprise Citroën, à l'époque où existait encore le département de Détermination mathématique des carrosseries. Son algorithme permet de placer d'une façon simple les points d'une courbe de Bézier selon leur paramètre t à partir de seulement un polygone caractéristique de la courbe (voir en page 111). L'idée est qu'une partie de la courbe de Bézier est aussi une courbe de Bézier.

L'algorithme

Soit P une courbe de Bézier d'équation

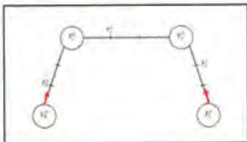
$$P(t) = \sum_{i=0}^{n} B_n^i(t) P_i$$

où $P_0 \ldots P_n$ représentent les $n + 1$ points de contrôle (ou *pôles* de la courbe). Pour t entre 0 et 1, on détermine P(t) par :

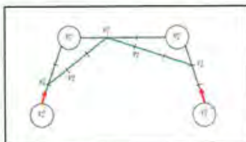
Pour $j = 1 \ldots n$ Faire
    Pour $i = 0 \ldots (n - j)$ Faire
        $P_i^j(t) = (1-t)P_i^{j-1} + tP_{i+1}^{j-1}$

(Le point $P_i^j(t)$ est barycentre de $P_i^{j-1}(t)$ affecté du poids $1 - t$ et du point $P_{i+1}^{j-1}(t)$ affecté du poids t.)
    Fin Pour
Fin Pour

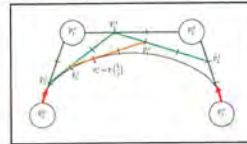
Voyons un exemple avec $n = 3$ et $t = 1/3$:



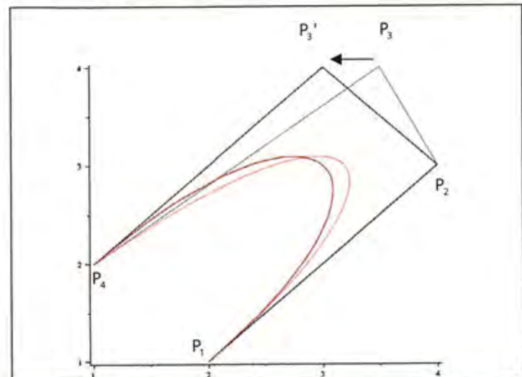
On partage les droites reliant les points caractéristiques en trois (car $t = 1/3$). On place P_0^1 à la distance t de P_0^0 et à la distance $(1 - t)$ de P_1^0. On continue avec les point P_1^1 et P_2^1.



On trace les droites qui relient les points, puis on place P_0^2 et P_1^2.



Ainsi on place le point P_0^3, la courbe est tangente à la droite $(P_0^2 ; P_1^2)$ en ce point. À partir de ce point et des tangentes au polygone caractéristique aux points P_0^0 et P_3^0, on peut tracer l'allure de la courbe de Bézier correspondante. Il reste toutefois un problème lié aux courbes de Bézier : si *un seul* point se déplace, c'est l'intégralité de la courbe qui est à redessiner ! Ce problème sera résolu plus tard par la découverte des courbes B-splines par Cox et de Boor en 1971 (voir en page 125).



par Hervé Lehning

La programmation structurée

Comment localiser une erreur dans un programme informatique ? Question complexe s'il en est, spécialement lorsque vous n'avez pas écrit ledit programme ! La programmation structurée fournit une réponse.

Dès les années 1960, les applications de gestion font apparaître une des difficultés de la programmation : il peut être facile d'écrire un programme en suivant pas à pas son cahier des charges, et très difficile de relire le résultat. Cette tâche est pourtant indispensable pour contrôler le programme ou, en cas d'erreur, le modifier.

Le GOTO et les paquets de nouilles

Dans cet esprit, la pire des instructions est le branchement inconditionnel, noté GOTO dans la plupart des langages (*go to* : aller à). En effet, en l'utilisant, une suite d'instructions peut s'étendre sur plusieurs feuilles différentes. Il est alors difficile d'en suivre la logique, à l'exemple de ce magnifique circuit électrique aperçu

Utilité des GOTO

Les branchements inconditionnels (GOTO) ont malgré tout une utilité : celle de traiter les exceptions, essentiellement les cas où une erreur inattendue s'est produite et où il est nécessaire de sortir du programme. Par exemple, une procédure de division de deux nombres peut comporter un branchement inconditionnel vers une procédure d'arrêt du programme dans le cas où le dénominateur est nul.

Résolution d'une équation du second degré

Arguments en entrée : a, b et c (réels)
Arguments en sortie : Nb, x et y (réels)
Si $a = 0$ alors Arrêt sur erreur (voir encadré *Utilité des GOTO*)
Calcul du discriminant : $\Delta := b^2 - 4\ ac$ (voir l'encadré *Affectation et Test d'égalité* ; il s'agit ici d'une affectation)
Calcul du nombre de racines : $Nb :=$ signe(Δ) + 1
Si $Nb \geq 1$ alors $\delta := \sqrt{\Delta}$, $x := (-b + \delta)/(2a)$, $y := (-b - \delta)/(2a)$.

dans une rue de Katmandou. Comment vérifier un tel réseau ? Comment intervenir dessus ? Comment le contrôler, le maintenir ?

Une solution pour éviter une pareille confusion est de structurer les branchements. Le problème est de même nature en informatique, où l'on parle volontiers de « paquets de nouilles » pour décrire ce type d'amas (dans lesquels on ne sait pas où mène le fil que l'on vient de saisir).

Structuration des programmes

Pour éviter ce genre de difficultés, une idée est de décomposer chaque programme en tâches intermédiaires, que l'on nomme souvent *routines*, *procédures* ou *fonctions*. Bien entendu, il existe une nuance entre chacun de ces objets. Par exemple, les fonctions renvoient une valeur, ce que ne font pas routines et procédures. Dans tous les cas, l'idée est de concevoir des parties de programme courtes (une quinzaine de lignes au maximum) admettant un certain nombre de données en entrées et de résultats en sortie. Chacune de ces procédures peut alors être testée ou modifiée séparément. Certaines peuvent être réutilisées dans d'autres contextes. Par exemple, une procédure de calculs de nombres en grande précision est utilisable dans bien des situations. On a donc intérêt à la paramétrer : taille des nombres, base de numération…

Branchements électriques à Katmandou. © Hervé Lehning.

Affectation et test d'égalité

Le symbole mathématique de l'égalité (symbole =) se dédouble dans les algorithmes en *affectation* et en *test d'égalité*. La première permet de changer la valeur d'une variable. Ainsi, $x := a$ met dans la case mémoire de nom x la valeur a. On dit également que l'on *affecte* a à x. Certains préfèrent noter cette opération avec une flèche : $x \leftarrow a$ (ce qui suggère un déplacement dans la mémoire de l'ordinateur).
Le second symbole = est un test logique. Ainsi, $x = a$ donne la valeur *Vrai* (*true* en anglais) si x vaut a et *Faux* (*false* en anglais) sinon.

Analyse

Pour bien structurer un programme, il est nécessaire d'analyser le problème posé afin de définir les sous-tâches à accomplir pour le résoudre. Prenons l'exemple de la résolution (dans l'ensemble des nombres réels) d'une équation du second degré à coefficients réels. Une telle procédure comporte en entrée trois coefficients (ceux de l'équation $a x^2 + b x + c$) et en sortie un indice Nb donnant le nombre de solutions (0, 1 ou 2), ainsi que deux réels (les solutions x_1 et x_2). Dans le corps de cette procédure, nous devons calculer le discriminant : $\Delta = b^2 - 4\,ac$ et tester s'il est nul, strictement positif ou strictement négatif. Cette analyse donne naissance à une première fonction (voir l'encadré *Une fonction signe*).

Ainsi, signe(x) donne le signe de x. La procédure générale de résolution d'une équation du second degré en découle immédiatement (voir l'encadré *Résolution d'une équation du second degré*).

L'utilisation de la fonction signe peut sembler futile. Il n'en est rien, cela permet de la modifier sans changer le programme principal. Dans l'ensemble des nombres réels, l'égalité n'a qu'un sens relatif. On peut ainsi utiliser un seuil de tolérance $\varepsilon > 0$ (10^{-12} par exemple) et estimer qu'un nombre x est nul si et seulement si : $| x |$ ″ ε. Ceci donne la nouvelle fonction (voir encadré *Une fonction signe en pratique*).

Inutile de modifier la procédure Résolution, l'utilisation de cette nouvelle fonction *signe* suffit pour le faire. Cette possibilité est l'un des avantages de la programmation structurée.

H. L.

Une fonction signe

En théorie :

Argument en entrée : x (un réel)
Argument en sortie : *signe* (– 1, 0 ou 1)
Si $x < 0$ alors *signe* := –1 sinon
Si $x > 0$ alors *signe* := 1 sinon *signe* := 0.

En pratique :

Argument en entrée : x (réel)
Arguments en sortie : *signe* (– 1, 0 ou 1)
Si $x < -\varepsilon$ alors *signe* := –1 sinon
Si $x > \varepsilon$ alors *signe* := 1 sinon *signe* := 0.

par F. Picard et É. Thomas

Les courbes B-splines

De Bézier aux B-splines

Le majeur problème d'une courbe de Bézier est que, lors de la modification d'un de ses points de contrôle, elle doit être entièrement redessinée (voir en page 121). Les courbes B-splines, qui sont une généralisation des courbes de Bézier, permettent de contourner cette limitation. Elles permettent de tracer des courbes à partir de plusieurs points de contrôle. Les modifications ne s'effectuent que localement, et non plus globalement.

Les *splines* sont des fonctions définies par morceaux par des polynômes. Les plus simples représentent des polygones. Pierre Bézier déplorait l'emploi du mot anglais *spline*, qui se traduit en français par « latte » ou « baguette ». Une *B-spline* est une combinaison linéaire de plusieurs splines (« B » pour « Bézier »). Pour les courbes qui ne peuvent être représentées par des B-splines (comme un quart de cercle), il existe une généralisation : les NURBS (Non-Uniform Rational Basis Splines). Ce sont des fonctions définies par morceaux par des fractions rationnelles, plutôt que des polynômes.

Les courbes B-splines ont été développées indépendamment par M.G. Cox et par le mathématicien allemand Carl de Boor en 1971 pour offrir un plus grand confort dans le dessin assisté par ordinateur. L'atout des courbes B-splines est qu'elles ne sont pas seulement définies avec des point de contrôle, mais également à partir de vecteurs de nœuds. Une courbe B-spline est tangente à son polygone caractéristique en le premier et le dernier points de contrôle.

Karl de Boor

L'algorithme

On définit déjà une courbe B-spline de degré 3. Du degré va dépendre le « tassement » de la courbe : plus le degré est faible, et plus la courbe sera proche des points de contrôle.

Les quatre points de contrôles sont $P_0(x_0\,;y_0)\ldots P_3(x_3\,;y_3)$. Selon la formule de Cox et de Boor, la courbe aura alors huit *nœuds* N (soit le nombre de points de contrôle, plus le degré de la courbe, plus un). À chaque nœud correspond un terme u_i défini par la séquence U = [0, 0, 0, 0, 1, 1, 1, 1].

L'équation de la courbe est la suivante :

$$P(u) = \sum_{i=0}^{3} N_{i;3}(u) P_i = P_0 N_{0;3}(u) + P_1 N_{1;3}(u) + P_2 N_{2;3}(u) + P_3 N_{3;3}(u).$$

Un algorithme permet de définir, par récurrence, les termes employés :

$$N_{i;0}(u) = \begin{cases} 1 & \text{si } \; u_j \le u < u_{j+1}, \\ 0 & \text{sinon.} \end{cases}$$

$$N_{i;m}(u) = \frac{u - u_i}{u_{i+m} - u_i} N_{i;m-1}(u) + \frac{u_{i+m+1} - u}{u_{i+m+1} - u_{i+1}} N_{i-1;m-1}(u).$$

Par convention, les quantités non définies seront prises égales à 0.

Dans le tableau ci-dessous, on peut observer le schéma du mécanisme de calcul de l'équation de la courbe en fonction du paramètre u.

	$u_0=0$	$u_1=0$	$u_2=0$	$u_3=0$	$u_4=1$	$u_5=1$	$u_6=1$	$u_7=1$
$m=0$	0	0	0	$N_{3,0}(u)=1$	0	0	0	0
$m=1$	0	0	$N_{2,1}(u)=1-u$	$N_{3,1}(u)=u$	0	0	0	0
$m=2$	0	$N_{1,2}(u)=(1-u)^2$	$N_{2,2}(u)=2u(1-u)$	$N_{3,2}(u)=u^2$	0	0	0	0
$m=3$	$N_{0,3}(u)=(1-u)^3$	$N_{1,3}(u)=3u(1-u)^2$	$N_{2,3}(u)=3u^2(1-u)$	$N_{3,3}(u)=u^3$	0	0	0	0

Nous obtenons finalement l'équation de la courbe B-spline dont les points de contrôle sont P_0, P_1, P_2 et P_3 :

$$P(u) = \left.\begin{matrix} x_0 \\ y_0 \end{matrix}\right| \times (1-u)^3 + \left.\begin{matrix} x_1 \\ y_1 \end{matrix}\right| \times 3u(1-u)^2 + \left.\begin{matrix} x_2 \\ y_2 \end{matrix}\right| \times 3u^2(1-u) + \left.\begin{matrix} x_3 \\ y_3 \end{matrix}\right| \times u^3.$$

par Hervé Lehning

La magie de la récursivité

Utiliser dans son algorithme (ou son programme) la fonction qu'on est en train de définir, c'est le principe de la récursion. Et vous constaterez que c'est bien plus efficace qu'un serpent qui se mord la queue !

Pour examiner la différence entre l'itération classique, rencontrée par exemple dans les algorithmes de tri de l'article précédent, et la récursion, nous allons prendre l'exemple de la fonction factorielle, qui à un nombre N associe le produit des nombres de 1 à N.

On la note $N!$ donc : $1! = 1$, $2! = 1 . 2 = 2$, $3! = 1 . 2 . 3 = 6$, $4! = 1 . 2 . 3 . 4 = 24$.

D'après sa définition, cette fonction vérifie la relation de récurrence : $(N + 1)! = (N!) \times (N + 1)$.

Cette relation permet de la prolonger (à rebours) en $N = 0$: de l'égalité : $1! = 0! . 1$, on déduit : $0! = 1$.

Les méthodes itératives peuvent sembler plus naturelles, mais les méthodes récursives sont plus faciles à prouver.

Mais on ne peut poursuivre cette logique en $N = -1$ puisqu'elle aboutirait à :

$0! = ((-1)!) . 0 = 0$ c'est-à-dire à $1 = 0$, ce qui est absurde.

La fonction factorielle est ainsi définie sur l'ensemble des nombres entiers naturels.

Procédure itérative

La première façon de calculer une factorielle est celle que nous employons quand nous effectuons les calculs « à la main ». Avec un ordinateur, nous devrons utiliser un emplacement de la mémoire que nous nommons « variable ». Cette notion, même si elle est désignée par une lettre, diffère légèrement de la notion mathématique correspondante. Notons S cette variable. *A priori*, l'emplacement correspondant dans la mémoire ne contient aucune

valeur. Pour cela, il est nécessaire de lui en attribuer une, c'est-à-dire de l'*initialiser*. L'algorithme commence donc par l'*affectation* d'une valeur à cette variable. Ensuite, nous calculons successivement les factorielles des entiers de 1 à N, ce que nous pouvons décrire de la façon suivante :

Fonction Factorielle, argument N
Initialisation : $S := 1$
Boucle : Pour K variant de 1 à N faire : $S := S \,.\, K$.
Résultat Factorielle: S

Cet algorithme est facile à écrire dans tous les langages de programmation. Les essais montrent qu'il donne le bon résultat dans tous les cas simples.

... et procédure récursive

La seconde façon de calculer une factorielle est récursive.
L'écriture de la fonction est alors bien plus troublante puisqu'elle fait appel à elle-même. La voici :

Fonction Factorielle, argument N
Si $N = 0$ alors Fact := 1 sinon
Fact := N . Factorielle($N - 1$)

Cette écriture peut sembler absurde, car la factorielle y est définie à partir d'elle-même. On la comprend mieux en pensant à une délégation de tâche.

Voyons comment cela fonctionne pour $N = 4$:
Comme 4 n'est pas égal à 0, l'appel de Factorielle(4) provoque l'affectation de 4 . Factorielle(3), qui demande le calcul de Factorielle(3). Nous retournons au calcul d'une factorielle en suivant l'algorithme ci-dessus.
Comme 3 n'est pas égal à 0, l'appel de Factorielle(3) provoque l'affectation de 3 . Factorielle(2), qui demande le calcul de Factorielle(2), et nous recommençons.
Comme 2 n'est pas égal à 0, l'appel de Factorielle(2) provoque l'affectation de 2 . Factorielle(1), qui demande le calcul de Factorielle(1), et nous recommençons.
Comme 1 n'est pas égal à 0, l'appel de Factorielle(1) provoque l'affectation de 1 . Factorielle(0), qui demande le calcul de Factorielle(0), et nous recommençons.
Comme 0 est égal à 0, l'appel de Factorielle(0) provoque l'affectation de 1. Les calculs envisagés et laissés de côté peuvent alors être effectués.
Nous obtenons successivement :
Factorielle(1) = 1 . Factorielle(0) = 1 . 1 = 1,
Factorielle(2) = 2 . Factorielle(1) = 2 . 1 = 2,
Factorielle(3) = 3 . Factorielle(2) = 3 . 2 = 6,
Factorielle(4) = 4 . Factorielle(3) = 4 . 6 = 24.

Preuve et invariant de boucle

Pour affirmer qu'un algorithme est correct, il faut de plus le prouver.
Dans le cas de la construction par itération de la factorielle, il suffit de montrer par récurrence qu'à la fin de la ligne « Boucle » (Pour K variant de 1 à N faire : $S := S \,.\, K$), la variable S contient factorielle de K.
- Cette propriété est vraie pour $K = 1$.
- Supposons qu'elle soit vraie pour une valeur K, la définition de la factorielle fait qu'elle est vraie à l'étape suivante, pour $K + 1$.
Par récurrence, elle est donc vraie pour tout K, d'où le résultat.
Cette propriété vraie à toute étape de la boucle est appelée un *invariant de boucle*. Elle est souvent utile pour prouver qu'une boucle est correcte.

Algorithme d'Euclide et récursivité

par Jean-Alain Roddier

La plupart des algorithmes qui utilisent l'itération (et ils sont légion) peuvent s'exprimer de manière récursive. Illustration avec l'algorithme d'Euclide (voir page 64).

Rappelons que cet algorithme, appliqué à deux entiers, consiste à effectuer des divisions successives, en remplaçant à chaque fois le couple (dividende ; diviseur) par le couple (diviseur ; reste), et cela tant que le reste n'est pas nul.

936 | 204 → 120 | 4
204 | 120 → 84 | 1
120 | 84 → 36 | 1
84 | 36 → 12 | 2
36 | 12 → 0 | 3

Exemple :
On peut réaliser un programme qui effectue une boucle tant que les restes des divisions successives sont non nuls. Un tel programme, développé avec Python, vous est donné en page 67. Il ne faut pas beaucoup le modifier pour obtenir un programme récursif :

```
>>> def algorec(a,b):
        r=a%b
        if r==0:
            p = b
        else:
            p=algorec(b,r)
        return p
```

Chaque procédure récursive peut se détailler ainsi. Elle occupe une place importante en mémoire car tous les appels récursifs doivent être stockés dans un sens, avant d'être exécutés dans l'autre. Mais cette description est inutile dans la pratique, car l'un des intérêts primordiaux des fonctions récursives est qu'elles permettent de concevoir des algorithmes dont la correction est facile à établir par récurrence.

Programmer, c'est prouver

De même que les procédures itératives, les procédures récursives se prouvent par récurrence. Celle-ci est cependant plus facile à exprimer. Par exemple, ici l'hypothèse de récurrence est : « Factorielle(N) donne le bon résultat, c'est-à-dire $N!$ », ce qui est vrai pour $N = 0$ d'après le début de la procédure « Si $N = 0$ alors Fact := 1 ». Supposons donc l'hypothèse exacte pour une valeur $N - 1$ et considérons l'entier suivant, N. Comme il n'est pas égal à 0, nous tombons sur : « sinon Fact := N . Factorielle($N - 1$) ». Comme (par hypothèse de récurrence) Factorielle(N – 1) donne le bon résultat, soit $(N - 1)!$, Factorielle(N) donne $N \,.\, (N - 1)! = N!$ c'est-à-dire le bon résultat. La *correction* (qualité de ce qui est correct, exact) de l'algorithme est aisément démontrée par récurrence

H. L.

La suite de Goodstein

Construction

La suite de Goodstein fut découverte par Reuben Louis Goodstein (1912–1985) en 1942. Il utilise la notion d'écriture *en base totale* d'un nombre (ou de sa *notation héréditaire dans une base*). Écrire un nombre en base totale consiste à écrire un nombre en utilisant uniquement les chiffres contenus dans la base prédéfinie. Par exemple, pour le nombre 42, on a $42 = 2^5 + 2^3 + 2 = 2^{2^2+1} + 2^2 + 2$ (notation héréditaire en base 2). De même, on a $42 = 3^3 + 3^2 + 2.3$ (notation héréditaire en base 3). L'écriture en base totale est unique.

Pour u et v deux entiers tels que $v \geq u \geq 2$, on définit l'application $T_{u\,;v} : \mathbb{N} \to \mathbb{N}$ tel que l'entier $T_{u\,;v}(n)$ soit le développement en base totale u de n, dans lequel on aura remplacé chaque u par v.

Par exemple, $T_{2;3}(42) = T_{2;3}\left(2^{2^2+1} + 2^2 + 2\right)$
$= 3^{3^3+1} + 3^3 + 3 = 2\,287\,679\,245\,991$
(on a remplacé tous les « 2 » par des « 3 »).

La suite de Goodstein utilise T. Pour chaque entier n, on définit la suite de Goodstein comme la suite d'entiers g_2, g_3, g_4... vérifiant $g_2 = n$ puis, par induction, $g_{b+1} = T_{b;b+1}(g_b) - 1$ si g_b est non nul, et $g_{b+1} = 0$ si g_b est nul.

Voyons l'exemple de la suite de Goodstein ayant pour premier terme 8. On obtient successivement $g_2 = 8$, $g_3 = 80$, $g_4 = 553$, $g_5 = 6\,310$. On pourrait penser que cette suite « explose » et tend très rapidement vers l'infini (penser à la nature de l'opération $T_{b;b+1}$). Il n'en est rien ! En 1942, Goodstein démontre que, pour tout entier n de départ, la suite converge vers 0. C'est-à-dire qu'il existe un entier $p(n)$ tel que $g_{p(n)} = 0$.

Le lièvre et la tortue

Une cigogne propose un concours arithmétique à un lièvre et une tortue. Les animaux ont deux objectifs différents. Le lièvre doit faire son possible pour obtenir le nombre le plus grand possible, la tortue a pour objectif de rendre le nombre le plus petit possible.

La cigogne choisit le nombre 8 comme début du jeu. À chaque tour, les deux animaux ont le droit de réaliser une opération sur le nombre choisi. Pour chaque animal, l'opération sera *toujours la même* tout le long du jeu.

Le lièvre, qui se croit malin, décompose le nombre 8 dans la base totale 2 et dit qu'il incrémente à chacun de ses tours la base totale. La tortue, qui a tout son temps, dit qu'elle soustrait 1 à chacun de ses tours.

Joueur	Valeur initiale	Ecriture en base totale	Après modification	Valeur finale
Lièvre	8	2^{2+1}	3^{3+1}	81
Tortue	81	3^{3+1}	$3^{3+1} - 1$	80
Lièvre	80	$2.3^3 + 2.3^2 + 2.3 + 2$	$2.4^4 + 2.4^2 + 2.4 + 2$	554
Tortue	554	$2.4^4 + 2.4^2 + 2.4 + 2$	$2.4^4 + 2.4^2 + 2.4 + 1$	553
Lièvre	553	$2.4^4 + 2.4^2 + 2.4 + 1$	$2.5^5 + 2.5^2 + 2.5 + 1$	6 311
Tortue	6 311	$2.5^5 + 2.5^2 + 2.5 + 1$	$2.5^5 + 2.5^2 + 2.5$	6 310

On peut croire que le lièvre l'emportera haut la main. Mais, d'après le résultat de Goodstein, la tortue gagne toujours, quel que soit l'entier de départ ! À chaque tour, la tortue « grignote » doucement mais sûrement les exposants, si bien qu'au bout de (très) nombreux tours de jeu, la tortue finit par gagner.

par Hervé Lehning

Itération et point fixe

Lorsque les itérées successives d'une fonction convergent, la limite obtenue en est un point fixe. Cette remarque est à l'origine des méthodes d'approximations successives, depuis Newton jusqu'à aujourd'hui.

Considérons une fonction f d'un ensemble E dans lui-même, où E peut être un intervalle de l'ensemble des nombres réels, un domaine du plan ou même un ensemble de phrases. Itérer la fonction f consiste à prendre une valeur initiale a, souvent appelée *germe*, calculer $f(a)$, et recommencer en remplaçant le germe par cette nouvelle valeur. On continue ainsi jusqu'à ce qu'une certaine condition d'arrêt soit remplie. Celle-ci porte normalement sur une comparaison entre les deux dernières valeurs atteintes. Cet algorithme se décrit donc ainsi :

$y := a$
Faire : $x := y$, $y := f(x)$ tant que l'égalité entre x et y est décrétée fausse
Résultat : y

L'inconvénient de cette itération est de « boucler » (c'est-à-dire ne pas s'arrêter) si x et y ne sont jamais égaux. Dans l'ensemble des nombres réels, cette égalité est souvent prise dans un sens approché. On choisit une valeur $\varepsilon > 0$ (10^{-10} par exemple) et on teste si $|y - x| < \varepsilon$.

Calcul d'une racine

Voyons comment et pourquoi cette méthode fonctionne sur l'exemple de $f(x) = \frac{x}{2} + \frac{1}{x}$ avec le germe 1. Nous effectuons donc l'algorithme :

$y := 1$
Faire : $x := y$, $y := f(x)$
tant que $|y - x| \geq \varepsilon$.
Résultat : y

Nous obtenons le tableau ci-contre.

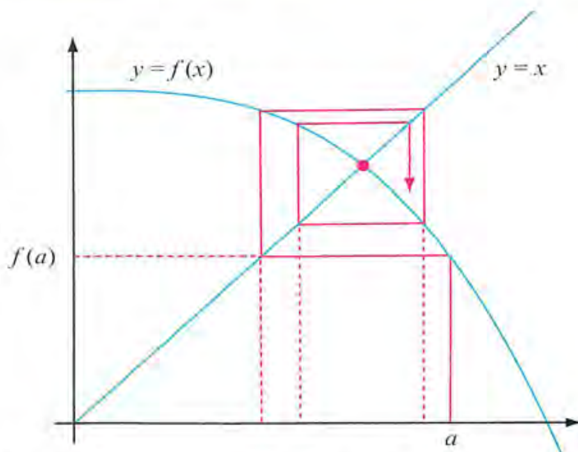
Itération	x	y	$\lvert y - x \rvert$
1	1	3/2	1/2
2	3/2	17/12	1/12
3	17/12	577/408	1/408
4	577/408	665857 / 470832	1 / 470832
5	665857 / 470832	886731088897 / 627013566048	1 / 627013566048

La dernière valeur dans la colonne de droite est inférieure à ε (10^{-10} ici). La boucle s'arrête donc. Ces premiers calculs laissent penser que c'est toujours le cas, ce que l'on peut démontrer (sous certaines hypothèses). Quelle est la limite ? Puisque $f(x) - x$ est quasiment nul après un certain nombre d'étapes, la limite correspond (intuitivement) à un point fixe de f, c'est-à-dire à un réel l tel que : $f(l) = l$. Dans notre cas, cette égalité s'écrit : $\frac{l}{2} + \frac{1}{l} = l$ soit $l^2 = 2$. Comme ce nombre est positif, on obtient : $l = \sqrt{2}$. Effectivement, les valeurs successives de x s'approchent très rapidement de $\sqrt{2}$ (voir l'encadré *Un calcul de convergence*).

Méthode du point fixe

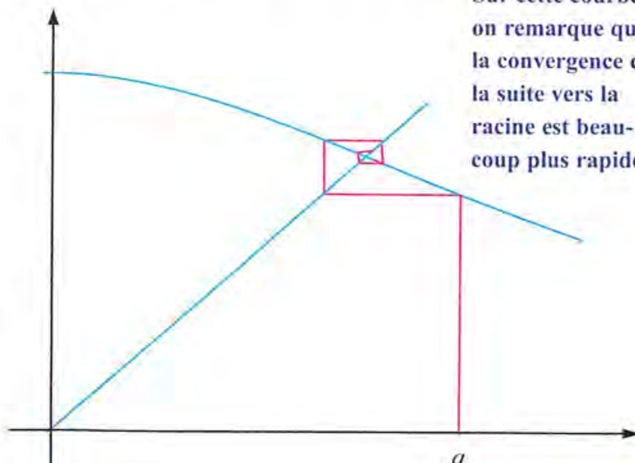
De façon générale, si l'itération d'une fonction f converge, la limite est un point fixe l de f. Nous retrouvons cette méthode pour résoudre des équations, à condition de les mettre sous la forme : $f(x) - x = 0$. Comment faire ? Examinons le problème sur le cas de l'équation $x^3 + 2x - 2 = 0$, qui possède une racine située entre 0 et 1 (puisque son premier membre change de signe entre ces deux valeurs).
Une première idée est d'écrire cette équation sous la forme $1 - \frac{x^3}{2} = x$. En prenant comme valeur initiale $a = 1$, nous obtenons une suite de valeurs dont la convergence est très lente. Le phénomène s'explique bien géométriquement.

On porte a sur l'axe des abscisses, la verticale en ce point coupe la courbe d'équation $y = f(x)$ en un point d'ordonnée $f(a)$. L'horizontale en ce point coupe la droite d'équation $y = x$ en un point d'abscisse $f(a)$, second terme de la suite. En recommençant le procédé, on obtient les suivants.



Construction géométrique de la suite des valeurs *x* à partir du germe a.

Le cinquantième terme est 0,77. Le calcul de $x^3 + 2x - 2$ pour 0,76 et 0,78 montre un changement de signe entre ces deux valeurs donc 0,77 est une valeur approchée à 10^{-2} près de la racine cherchée.
Sur la figure ci-dessus, nous voyons que la lenteur de la convergence de la suite est liée à la pente de la courbe d'équation $y = f(x)$ au point d'abscisse l. En écrivant l'équation sous la forme : $\frac{2}{x^2+2} = x$, nous obtenons une convergence beaucoup plus rapide, ce qui se visualise sur un dessin.



Sur cette courbe, on remarque que la convergence de la suite vers la racine est beaucoup plus rapide.

Maintenant, 0,77 est obtenu après seulement huit itérations. Après trente itérations, on obtient 0,770 916 997 1. De même que précédemment, on peut alors vérifier l'approximation obtenue en calculant le signe de $x^3 + 2x - 2$ pour 0,770 916 997 0 et 0,770 916 997 2. Remarquez également qu'un mauvais choix d'écriture, comme : $x^3 + 3x - 2 = x$, peut donner une suite divergente.

Choix de Newton

Une idée générale pour mettre une équation $P(x) = 0$ sous la forme $f(x) - x$ est de choisir $f(x) = x + k(x)\,P(x)$ où k est une fonction telle que la dérivée de f s'annule comme P, c'est-à-dire : $f'(x) = 0$ quand $P(x) = 0$. Pour cela, il suffit de choisir : $1 + k(x)\,P'(x) = 0$, ce qui donne formellement :

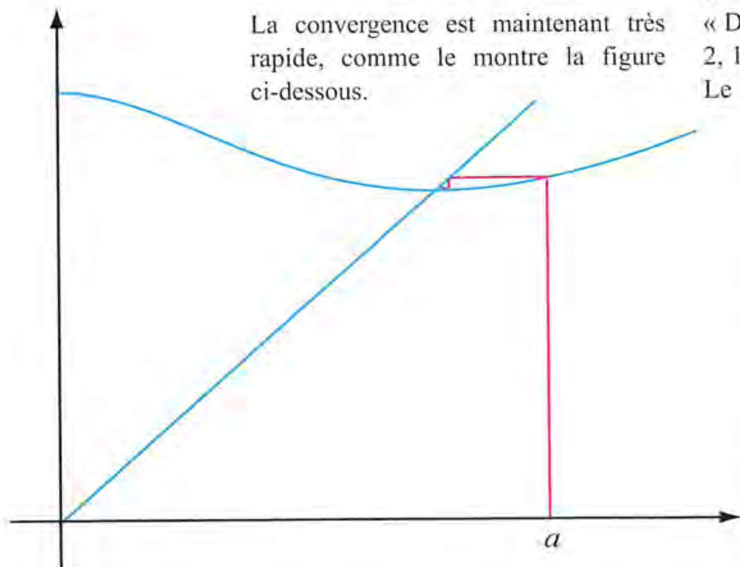
$$f(x) = x - \frac{P(x)}{P'(x)}.$$

Ce choix est celui de Newton (1643–1727). Dans notre cas particulier, nous obtenons :

$$f(x) = x - \frac{x^3 + 2x - 2}{3x^2 + 2} = 2\frac{x^3 + 1}{3x^2 + 2}.$$

La convergence est maintenant très rapide, comme le montre la figure ci-dessous.

La tangente en la limite est horizontale, ce qui donne une convergence très rapide.



Le résultat est maintenant obtenu avec 10 décimales exactes après quatre itérations seulement.

La méthode hors les murs

La méthode des approximations successives fournit des résultats surprenants dans d'autres domaines. Voici un exemple proposé par un logicien, Raphaël Robinson (1911–1995), qui le résolvait autrement :

Compléter les blancs dans la phrase suivante de sorte qu'elle exprime une assertion vraie :

« Dans cette phrase, il y a _ 0, _ 1, _ 2, _ 3, _ 4, _ 5, _ 6, _ 7, _ 8 et _ 9. »

Une idée *a priori* folle est de décrire la phrase actuelle, celle écrite ci-dessus. On compte le nombre de 0, de 1, *etc.* On trouve *un* chiffre de chaque sorte, d'où un premier résultat :

« Dans cette phrase il y a 1 0, 1 1, 1 2, 1 3, 1 4, 1 5, 1 6, 1 7, 1 8 et 1 9. »

Bien sûr, cette proposition est fausse, puisque la phrase a été modifiée. Pour une fois, persévérons dans l'erreur en décrivant cette nouvelle phrase. On trouve *un* chiffre de chaque sorte, sauf *onze* 1 :

« Dans cette phrase il y a 1 0, 11 1, 1 2, 1 3, 1 4, 1 5, 1 6, 1 7, 1 8 et 1 9. »

Le résultat est encore faux. Un peu moins toutefois que précédemment, puisque cette fois-ci nous avons *douze* 1 :

« Dans cette phrase il y a 1 0, 12 1, 1 2, 1 3, 1 4, 1 5, 1 6, 1 7, 1 8 et 1 9. »

Nous revenons ensuite à *onze* 1 mais *deux* 2 :

« Dans cette phrase il y a 1 0, 11 1, 2 2, 1 3, 1 4, 1 5, 1 6, 1 7, 1 8 et 1 9. »

Si nous recommençons, nous trouvons le même résultat. La

phrase précédente est donc vraie ! La méthode des approximations successives nous a permis de résoudre le problème de Robinson. Cette méthode a été proposée par Douglas Hofstadter (voir *Ma Thémagie*, Interéditions, 897 pages, 1988) qui s'est demandé si cette suite d'approximations se comportait toujours ainsi lorsque l'on changeait la condition initiale.

Une suite qui se raconte

Par exemple, si nous commençons par le nombre 0, nous comptons un « 0 », le terme suivant est donc : 10 (« un zéro »). De même, nous comptons un « 0 » et un « 1 », le terme suivant est donc : 10 11. Le compte est toujours de un « 0 », mais trois « 1 », d'où : 10 31. Nous continuons ainsi et obtenons successivement :

10 21 13
10 31 12 13
10 41 12 23
10 31 22 13 14
10 41 22 23 14
10 31 32 13 24
10 31 22 33 14
10 31 22 33 14

Les deux derniers termes sont égaux. La règle de formation des termes fait que le suivant est égal au précédent. La suite reste donc fixe à partir de ce terme. Est-ce toujours le cas ? Il faut être patient pour réaliser que non. Pour trouver un résultat différent, nous devons essayer la condition initiale 40 :

40
10 14
10 21 14
10 31 12 14
10 41 12 13 14
10 51 12 13 24
10 41 22 13 14 15
10 51 22 13 24 15
10 41 32 13 14 25
10 41 22 23 24 15
10 31 42 13 24 15
10 41 22 23 24 15

Le dernier terme est égal à celui qui le précède deux lignes plus haut (en gras). Le suivant est donc égal au précédent, et ainsi de suite. Autrement dit, les termes de la suite se répètent de deux en deux indéfiniment. On dit que la suite est périodique, de période 2. En essayant la valeur initiale 50, on obtient une suite périodique, de période 3. Vous pouvez essayer autant de valeurs que vous voulez, vous ne trouverez jamais d'autres cas : ces suites sont toujours périodiques, de période 1, 2 ou bien 3.

De façon surprenante, il est relativement facile de prouver ce résultat par la méthode d'exhaustion. Pour le démontrer, examinons la structure d'un terme de la suite après la condition initiale. Il s'agit d'une suite de dix coefficients, suivis éventuellement des chiffres de 0 à 9. Supposons que les coefficients aient au plus D chiffres. Ainsi chacun apparaît au plus 10 D fois dans les coefficients, plus une dans lui-même. À l'étape suivante, chaque coefficient est au plus égal à 10 D + 1. Si D > 2, ce nombre a moins de D – 1 chiffres. En itérant ce raisonnement, nous arrivons à une étape où les coefficients ont au plus deux chiffres.

Cela nous amène à essayer 99^{10} cas distincts (soit un peu moins de 10^{20}). Bien sûr, ce nombre est beaucoup trop grand pour qu'on puisse étudier tous les cas, même avec un ordinateur très puissant. (Pour vous en rendre compte, imaginez que vous soyez capable d'examiner un cas par nanoseconde, c'est-à-dire en un milliardième de seconde. Combien de temps vous faudra-t-il au total ? Le calcul est

Un calcul de convergence

La relation : $f(x) = x$ se simplifie en $x^2 = 2$ puis, comme x est positif, en $x = \sqrt{2}$. Cette remarque permet d'écrire :

$$f(x) - \sqrt{2} = \left(\frac{x}{2} + \frac{1}{x}\right) - \left(\frac{\sqrt{2}}{2} + \frac{1}{\sqrt{2}}\right),$$

$$= \left(\frac{x}{2} - \frac{\sqrt{2}}{2}\right) + \left(\frac{1}{x} - \frac{1}{\sqrt{2}}\right) = \frac{\left(x - \sqrt{2}\right)^2}{2x}.$$

Comme $x > 1$, cela implique l'inégalité :

$$f(x) - \sqrt{2} < \frac{\left(x - \sqrt{2}\right)^2}{2}.$$

Si, à une étape, la précision de n chiffres significatifs est atteinte, à la suivante on obtient deux fois plus de chiffres. Cela se voit mieux quand on écrit les termes de la suite sous forme décimale : 1 ; 1,5 ; 1,417 ; 1,414 216
1,414 213 562 375 ; 1,414 213 562 373 095 048 801 690 ;
1,414 213 562 373 095 048 801 688 724 209 698 078 56
9 671 875 377 ; *etc.*

simple : $10^{20} \times 10^{-9} = 10^{11}$ secondes. Une année moyenne contient $60 \times 60 \times 24 \times 365{,}25 = 0{,}3 \times 10^8$ secondes. Le calcul demanderait donc environ 3 000 ans. Le jeu n'en vaut pas la chandelle !) Cependant, cette suite cache un bon nombre d'invariants. Ils agissent comme des contraintes sur ses termes. Le nombre de cas à analyser en est fortement réduit, ce qui permet de prouver le résultat. La question est cependant trop technique pour donner la preuve complète ici (voir notre article Computer Aided or Analytic Proof ?, in *The College Mathematics Journal*, Mai 1990).

La méthode hors des chiffres

La méthode des approximations successives fonctionne dans des cas plus surprenants encore, où des lettres sont en jeu, comme dans ce problème :
Compléter le blanc dans la phrase suivante par un nombre écrit en toutes lettres de sorte qu'elle exprime une assertion vraie :
« Cette phrase contient _ consonnes. »
Reprenons notre méthode *a priori* stupide. Décrivons la phrase ci-dessus. Nous y comptons 18 consonnes, nous proposons donc :
« Cette phrase contient dix-huit consonnes. »
Bien entendu, cette phrase est fausse puisque le nombre de consonnes a été modifié. Nous en comptons 22 maintenant ! Nous la corrigeons donc successivement avec la même méthode :
« Cette phrase contient vingt-deux consonnes. »
« Cette phrase contient vingt-quatre consonnes. »
« Cette phrase contient vingt-cinq consonnes. »
« Cette phrase contient vingt-cinq consonnes. »
Les deux dernières phrases sont identiques. Comme la seconde décrit la première, elle se décrit elle-même. Cette assertion est vraie ! Encore une fois, nous avons résolu le problème au moyen de notre stratégie *a priori* fausse. Bien entendu, le miracle ne se produit pas toujours. Comme dans le cas de la suite qui se raconte, nous pouvons obtenir des périodes autres que 1.

H. L.

Algorithmes et ordinaux

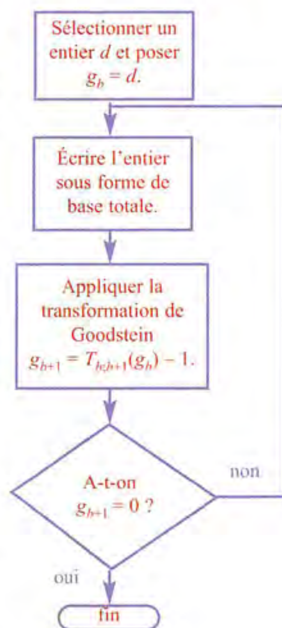
Quand les ordinaux prennent le train

Pour se familiariser avec la notion de nombre ordinal, on peut imaginer un train mathématique pouvant transporter une infinité dénombrable de passagers. Le train s'arrête dans ω_1 gares. Dans chaque gare, sauf la première, on aperçoit 5 passagers monter et 1 descendre. À la gare numéro n, le train contiendra $1 + (5 - 1).n$ passagers, soit $1 + 4n$. Nous pouvons numéroter les passagers de 1 à $5n$. Nous pouvons également numéroter de 1 à $n - 1$ les passagers descendus avant ω_1.

L'objet ω_1 est tel que, en arrivant à la gare ω_1, le train sera vide. En fait, selon l'axiome du choix, le train aura toujours plus de gares que de passagers. Le train arrivera donc toujours vide !
Une propriété des nombres ordinaux est que toute suite strictement décroissante d'ordinaux est finie. ω_1 est en fait le plus petit ordinal non dénombrable. Nous renvoyons le lecteur intéressé par les propriétés des ordinaux à l'article Cantor et les infinis, *Tangente* 129, page 10.

Utilisation des ordinaux en algorithmique

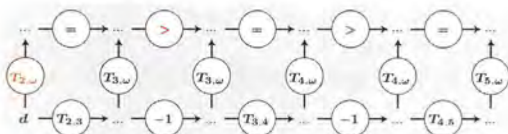
La suite de Goodstein peut être représentée de façon schématique par l'algorithme suivant (voir en page 129 pour plus d'informations).



On démontre que, quelle que soit la valeur initiale d, l'algorithme de Goodstein mène à 0. Le principe de la démonstration de cette propriété repose sur les nombres ordinaux (faire correspondre à chaque entité de la suite de Goodstein un ordinal).
Soit g_b le $b^{\text{ème}}$ terme de la suite : $g_{b+1} = T_{b;b+1}(g_b) - 1$.
Pour b un entier et ω un nombre ordinal tels que $\omega \geq b \geq 2$, on définit l'application $T_{b;\omega}$ telle que l'entier $T_{b;\omega}(n)$ soit le développement en base totale b de n dans lequel on aura remplacé chaque b par ω.
On note $T_{3;\omega}(g_3) = G_3$. Si $g_2 = 8$, alors

$$T_{3;\omega}(8) = T_{3;\omega}\left(2\times 3^3 + 2\times 3^2 + 2\times 3 + 2\right) = 2\times\omega^\omega + 2\times\omega^2 + 2\times\omega + 2.$$

On remarque que appliquer $T_{b;\omega}$ est similaire à appliquer $T_{b;b+1}$ puis $T_{b+1;\omega}$
La transformation conserve les relations d'ordre, donc nous pouvons obtenir le schéma suivant :



Si la suite de Goodstein était infinie (c'est-à-dire si elle ne s'arrêtait pas à 0), alors nous aurions une suite infinie d'ordinaux strictement décroissante. Cela est impossible.

Cette propriété générale est vraie, mais *ne peut pas* être démontrée dans l'axiomatique de Peano. Pour plus d'informations, lire *L'infini est-il nécessaire ?* par Patrick Dehornoy, Pour la Science 278, pp. 102–106, décembre 2000.

La gloutonnerie
appliquée à la compression

Tous les gloutons le savent : le meilleur moyen de s'empiffrer est de toujours choisir le plus gros morceau ! Cette stratégie s'applique également à certains algorithmes. Ils sont alors qualifiés de « gloutons ».

Pour moi, satisfaisant mes appétits gloutons,
J'ai dévoré force moutons.

Les animaux malades de la peste,
Jean de la Fontaine

Rendre la monnaie est l'un des problèmes essentiels de la programmation des distributeurs automatiques. On connaît le nombre de pièces dont on dispose dans chaque catégorie ainsi que la somme à rendre. Comment monnayer correctement cette somme en utilisant le minimum de pièces ? Le glouton répond bien à cette question. Commencez par rendre la plus grande pièce possible et recommencez avec le nouveau montant. À condition de disposer de suffisamment de pièces, vous obtenez ainsi une fonction récursive donnant une réponse optimale. Bien entendu, cet algorithme ne fonctionne pas toujours. Parfois, la gloutonnerie ne paie pas !

Pour atteindre une solution optimale, le glouton optimise chaque étape.

Imaginez que vous deviez rendre 2,10 € alors que votre caisse contient une pièce de 1 €, quatre pièces de 50 centimes et quatre de 20. Le glouton se précipite sur la pièce de 1 € puis sur deux de 50 centimes. Il est alors incapable de rendre les 10 centimes restant. Bien entendu, un bon caissier rendra une pièce de 1 €, une de 50 centimes et trois de 20 pour obtenir les 2,10 € demandés. Dans ce cas, la stratégie gloutonne ne fonctionne pas.

La compression de données

En guise d'exemple plus significatif, traitons le problème de la compression de données sans perte. Pour simplifier, supposons que nous ayons à transmettre un texte T composé seulement avec les lettres a, b, c, d et e. Les fré-

Le code ASCII

ASCII est l'acronyme de American Standard Code for Information Interchange, ou Code américain normalisé pour l'échange de l'information. C'est une norme très largement répandue de codage des caractères en informatique. Elle a été conçue en 1961 par l'informaticien américain Robert Bemer (1920–2004). La norme ASCII définit 128 caractères (de 0 à 127 en notation décimale, de 0000000 à 1111111 en notation binaire) et contient en particulier tous les caractères nécessaires pour écrire en anglais. Par exemple, l'esperluette & est codée 38 (en décimal) et 0100110 (en binaire). Toutefois, comme aujourd'hui l'unité de mesure de la quantité d'information est l'octet (huit bits), chaque caractère d'un texte en ASCII est stocké dans un octet dont le huitième bit est 0.

quences d'apparition de ces lettres dans le texte sont données par le tableau ci-dessous :

Lettre	*a*	*b*	*c*	*d*	*e*
Fréquence	5%	50%	20%	10%	15%

Fréquences d'apparition des symboles dans le texte T.

En code ASCII, chaque caractère a la taille d'un octet c'est-à-dire huit bits (un *bit* ou *binary unit* est un chiffre égal à 0 ou à 1). Un texte d'un million de caractères utilise donc un méga-octets s'il est codé en ASCII (voir encadré). Dans notre cas, nous n'avons que cinq caractères différents, nous pouvons donc les coder chacun sur trois bits selon la table de codage :

a : 000 ; b : 001 ; c : 010 ; d : 011 et e : 100.

Le codage du texte proposé comprendra trois millions de bits au lieu de huit millions, soit un gain de plus de 60 % ! Le décodage est simple en découpant le texte codé en groupes de trois bits et en lisant la table de codage à l'envers. Une idée plus subtile consiste à utiliser des codes de longueurs variables. Voici une table de codage adaptée :

a : 1101 ; b : 0 ; c : 101 ; d : 111 et e : 100.

Pour cent caractères à coder, la longueur moyenne du message codé est égale à :

$$5 \times 4 + 50 \times 1 + 20 \times 3 + 10 \times 3 + 15 \times 3 = 205 \text{ bits}.$$

Pour un million de caractères, nous obtenons 2 050 000 bits. Le gain supplémentaire est d'environ 30 %. Cette méthode suit la stratégie gloutonne puisqu'elle affecte les codes les plus courts aux lettres les plus fréquentes, et les codes les plus longs aux lettres moins fréquentes.

La règle des préfixes

La méthode de décodage précédente n'est plus applicable si les codes sont de longueur variable. Pour décoder, les bits du texte sont lus et accumulés l'un

Le premier choix qui compte

Le nombre de codages préfixes étant fini, l'un d'entre eux est optimal (la longueur du texte codé correspondant est minimale). Soit A l'arbre correspondant. On considère a et b les deux feuilles sœurs de profondeur maximale dans A. On peut supposer :

$$f(a) \leq f(b) \text{ et } f(x) \leq f(y)$$

puisqu'il ne s'agit que d'une question de notation. Comme x et y sont les lettres les moins fréquentes,

$$f(x) \leq f(a) \text{ et } f(y) \leq f(b).$$

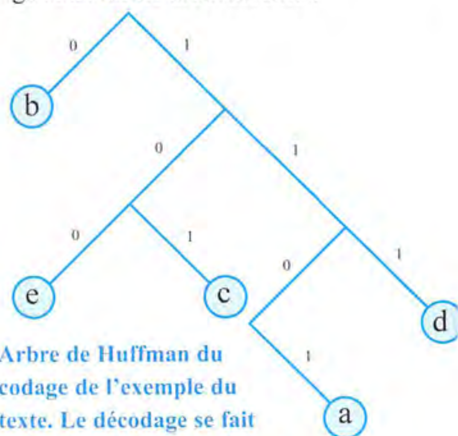
On permute alors les positions de a et de x, puis de b et de y dans l'arbre A. On obtient un nouvel arbre B. La longueur du codage du texte correspondant à B est inférieure à celle correspondant à A, puisque a et b sont plus fréquentes que x et y respectivement. La longueur du code correspondant à A étant minimale, les deux longueurs sont donc égales. L'arbre B est donc optimal. Dans ce codage, les codes de x et de y ont même longueur et ne diffèrent que par le dernier bit.

David Huffman (1925–1999) a découvert les codes qui portent son nom en 1952 alors qu'il était étudiant au M.I.T.

après l'autre jusqu'à obtenir un code valide. Par exemple, supposons qu'avec la table de codage précédente, un mot ait été codé en 11111010100. Pour le décoder, nous lisons le premier bit, 1. Il ne constitue pas un code. Nous lisons le bit suivant (1 encore), ce qui donne 11. Ce n'est pas un code non plus. Nous continuons et obtenons 111, qui est le code de d. Nous notons « d » et remettons l'accumulateur à zéro. Nous reprenons : 1, puis 11, puis 110 ne sont pas des codes, mais 1101 en est un, celui de a. Et ainsi de suite. Aucune ambiguïté n'est possible car aucun code n'est le préfixe d'un autre. Nous obtenons finalement le mot « *dabe* ».

Arbre de Huffman

Une méthode simple pour décoder le message est de noter la règle de codage sous forme arborescente :



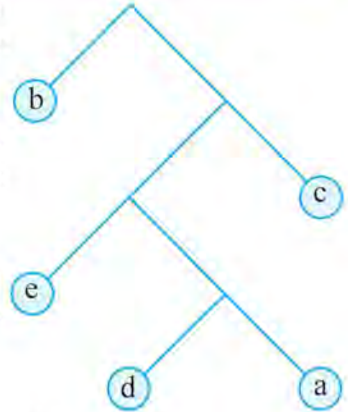
Arbre de Huffman du codage de l'exemple du texte. Le décodage se fait en le parcourant de la racine jusqu'aux feuilles.

Au départ, nous commençons à la racine de l'arbre. Celle-ci se situe en haut de l'arbre comme en généalogie. Chaque bit correspond à une descente sur une branche suivant la règle suivante : 0 implique une descente à gauche, 1 une descente à droite. Chacun nous fait donc passer d'un nœud (le premier étant la racine) à un autre. Après lecture d'un certain nombre de bits, nous parvenons à une feuille étiquetée par une lettre. Nous l'écrivons et recommençons à la racine de l'arbre avec le bit suivant. Cette notation arborescente fournit une programmation simple du décodage. De plus, elle est à l'origine d'une idée de création automatique de la table de codage.

Construction de l'arbre

David Huffman a proposé un algorithme pour construire un arbre de Huffman associé à un texte. Nous commençons par calculer la fréquence d'apparition de chaque caractère du texte à compresser. Nous prenons alors les deux lettres les moins fréquentes (a et d dans notre exemple), et nous formons un arbre les joignant, de la façon suivante :

Dans cet arbre, chacune des feuilles est étiqueté par a et d. La racine est étiquetée par $a \mid d$. Très logiquement, nous lui attribuons la somme des fréquences de a et de d, soit 15 %. Nous recommençons en remplaçant a et d dans la liste des caractères par $a \mid d$. Nous adjoignons donc e et remplaçons $a \mid d$ et e par $a \mid d \mid e$, dont la fréquence est 30 %. Notre nouvel arbre comprend une feuille de plus. À la fin, nous obtenons l'arbre de Huffman représenté ci-contre.



Arbre de Huffman construit avec l'algorithme de Huffman.

La table de codage correspondant à cet arbre est la suivante :

a : 1011 ; b : 0 ; c : 11 ;
d : 1010 et e : 100.

À présent, pour cent caractères à coder, la longueur moyenne du message codé est égale à :

$$5 \times 4 + 50 \times 1 + 2 \times 3 + 10 \times 4 + 15 \times 3 = 161 \text{ bits.}$$

Le gain est maintenant d'environ 45 % au lieu de 30 %.

Huffman est optimal

Ce phénomène est général. En d'autres termes, le codage de Huffman est optimal. Dans notre cas particulier de codage suivant la règle des préfixes, la gloutonnerie consiste à attribuer les codes les plus longs aux lettres les moins fréquentes. Ainsi, nous commençons la construction de l'arbre par ces feuilles en reliant les lettres les moins fréquentes. Étant les plus « basses » sur l'arbre, elles auront les codes les plus longs. Plus précisément, le résultat suivant montre que la construction d'un arbre optimal peut commencer par le choix glouton consistant à fusionner les deux lettres de fréquences minimales (voir l'encadré Le premier choix qui compte) :Soit T un texte où chaque caractère a possède une fréquence $f(a)$. Supposons que x et y soient les deux caractères les moins fréquents de T. Il existe un codage préfixe optimal tel que les caractères x et y aient des codes de même longueur et ne diffèrent que par le dernier bit.

L'étape suivante revient à remplacer x et y dans T par $z = \{x ; y\}$ de fréquence :

$$f(z) = f(x) + f(y)$$

Compressions d'images et de sons

Pour la compression d'images ou de sons (fichiers de suffixe JPEG, MPEG, ou MP3), on procède au préalable à une compression avec perte utilisant les faiblesses des sens humains. Du point de vue mathématique, après avoir utilisé des représentations utilisant des transformées de Fourier, on utilise actuellement des ondelettes. Dans ce domaine, la recherche reste très active. Après une première compression avec perte, on applique ensuite l'algorithme de Huffman.

et à coder le nouveau texte obtenu T' suivant le même principe. Les longueurs des deux textes codés sont les mêmes, puisque celles des codes de x et de y aussi. L'algorithme de Huffman apporte donc une solution optimale au problème posé.

Le codage de Huffman a de plus l'avantage de donner une programmation simple. Son utilisation comme méthode de compression (création de fichiers ZIP) exige de transmettre en début de message codé l'arbre de codage, ce qui réduit son intérêt dans le cas de textes courts. Il est à la base des compressions JPEG, MPEG ou MP3 (voir l'encadré Compressions d'images et de sons). D'autre part, l'élimination de cet arbre dans le message peut transformer cette méthode de compression en méthode de cryptographie. Mais ce n'est pas tout !

Allocation de ressources

Le problème d'affectation d'une ou de plusieurs ressources (imprimante, moniteur vidéo, haut-parleur…) se rencontre souvent, en informatique comme dans d'autres domaines. Pour illustrer la question de façon concrète, imaginons que nous ayons un certain nombre de demandes de location pour un seul ours en peluche (la même idée fonctionne pour les bicyclettes, les dromadaires, les automobiles et même les chars d'assaut). Chaque demande est donnée par une date de fin et une date de début de location. Le but est de satisfaire le maximum de clients possibles. Un algorithme glouton pour y parvenir consiste tout d'abord à classer les demandes par dates de fins. On obtient e_1, e_2, …, e_n. On initialise ensuite un ensemble E au vide ($E = \{\}$). Puis, pour i variant de 1 à n, on ajoute la demande e_i si elle ne chevauche pas la dernière demande appartenant à E. À la fin, on obtient un planning de location optimal.

Si le but est de maximiser la durée totale de location de l'ours en peluche, l'algorithme glouton ne donne pas l'optimum car il ne considère pas comme prioritaire une demande de location de durée très importante. Une idée est alors de classer les demandes par durées décroissantes et d'appliquer l'algorithme glouton.

Malheureusement, cet algorithme ne donne pas non plus le bon résultat. En fait, on montre que le problème de la maximisation de cette durée totale est NP-complet (voir l'article sur la question). Aussi, n'espérez pas trouver un algorithme simple et efficace. L'article sur le problème des mariages stables dans ce numéro approfondit la question.

H. L.

L'algorithme de Bruss

Un problème de dé

On vous propose le jeu suivant : on va faire rouler 20 fois, devant vous, un dé non truqué à six faces. On vous demande, au cours de ces lancers, de dire « Stop ! » si vous pensez que le chiffre affiché est exactement le dernier « 6 » de la série des 20 lancers. Une fois que vous avez dit « Stop ! », au lancer numéro j (compris entre 1 et 20), vous ne pouvez plus revenir sur votre décision : les $20 - j$ derniers lancers sont effectués, et l'on regarde si vous avez gagné.
Votre problème est de trouver une stratégie d'arrêt qui vous maximise la probabilité de vous arrêter sur le *dernier* « 6 » de la série des 20 lancers. Dans le cas du dé, vous savez que, au cours des six derniers lancers, vous devriez *en moyenne* observer un « 6 ». Votre stratégie *optimale* (mais qui ne sera peut-être pas couronnée de succès…) consiste à attendre ces six derniers lancers, et à vous arrêter (« Stop ! ») dès que vous voyez un « 6 ».
Le problème se généralise à un nombre fini d'événements à venir ($n = 20$ lancers), dont certains sont pour vous des succès (A_i : « Obtenir 6 »), et dont on peut estimer la probabilité d'occurrence p_i (1/6 ; l'équiprobabilité n'est pas exigée). Il s'agit de choisir le *dernier* succès parmi les n événements.
Si toutes les réalisations de ces événements sont indépendantes, F.T. Bruss a démontré qu'il existe toujours une stratégie de choix optimale ! Et un algorithme très simple explique comment faire. On écrit les probabilités $p_n, p_{n-1}, p_{n-2} \ldots p_2, p_1$ de la dernière à la première. De même, on écrit les *odds* $r_n, r_{n-1}, r_{n-2} \ldots r_2, r_1$ (voir ci-dessous). On calcule successivement r_n, $r_n + r_{n-1}$, $r_n + r_{n-1} + r_{n-2} \ldots$ jusqu'à ce que cette somme dépasse 1. Soit s l'indice tel que $r_n + r_{n-1} + r_{n-2} + \ldots + r_{s+1} < 1$ et $r_n + r_{n-1} + r_{n-2} + \ldots + r_s \geq 1$. Votre stratégie consiste à laisser passer les $s - 1$ premiers événements, et à choisir le premier succès (s'il s'en présente un !) à partir de l'événement s.

F.T. Bruss

Franz Thomas Bruss est professeur de mathématiques à l'Université libre de Bruxelles. Il est spécialiste des probabilités. Il publie en 2000 l'algorithme de Bruss (ou algorithme « des *odds* »). Le terme anglo-saxon *odds* désigne le ratio r entre une probabilité p (strictement comprise entre 0 et 1) et son complément à 1 : $r = p / (1 - p)$. On retrouve ce terme dans certains jeux de hasard (poker…). Il ne possède pas d'équivalent en français (ni dans bien d'autres langues d'ailleurs).
L'algorithme de Bruss est aujourd'hui employé dans de nombreux contextes décisionnels (de la vie courante au cadre industriel) : choix d'une place de parking en ville ; entretiens d'embauche ; en médecine, il pourrait aider un spécialiste à décider si un patient a intérêt à suivre un traitement lourd ou non… Récemment, il a été employé pour montrer que l'on peut améliorer la maintenance des systèmes de production complexes.

Référence

Le bon choix… raisonné. F.T. Bruss, *Pour la Science* 335, septembre 2005.

par Hervé Lehning

Codes correcteurs d'erreurs

Aucune ligne de transmission n'est à l'abri d'imperfections. La communication d'un message peut produire des erreurs indétectables. Pour remédier à cette faille, les messages doivent pouvoir les corriger eux-mêmes. C'est ce que réalise le code de Hamming.

Quand on envoie un message, on désire qu'il arrive intact même en cas d'erreurs de transmission. Par exemple, imaginons que nous envoyions le message « les fouilles sont visibles. » Si une erreur se produit à la transmission, le destinataire peut en recevoir un autre, par exemple : « des fouilles sont visibles », « les nouilles sont visibles », « les fouilles sont risibles » et encore bien d'autres possibilités ayant parfois un sens. Une erreur de transmission peut donc être indétectable. Une idée pour éviter ces confusions est que les messages possibles soient « à distances mutuelles » au moins égales à deux. Deux erreurs sont alors nécessaires pour transformer l'un en l'autre.

Codage des messages

Si nous restons dans le langage courant, ceci est difficile à réaliser. Seul un maître ès contrepets est capable d'écrire un message sans confusion possible. Il vaut mieux s'en remettre aux mathématiques, en opérant sur le codage numérique des messages. Pour donner un exemple très simple, si nous voulons coder notre alphabet sans les majuscules et les accents mais avec l'espace plus quelques signes de ponctuation, il suffit d'utiliser cinq bits (symboles 0 ou 1), à l'aide par exemple de la table ci-contre.

<Espace>	00000	*h*	01000	*p*	10000	*x*	11000
a	00001	*i*	01001	*q*	10001	*y*	11001
b	00010	*j*	01010	*r*	10010	*z*	11010
c	00011	*k*	01011	*s*	10011	,	11011
d	00100	*l*	01100	*t*	10100	.	11100
e	00101	*m*	01101	*u*	10101	!	11101
f	00110	*n*	01110	*v*	10110	?	11110
g	00111	*o*	01111	*w*	10111	:	11111

Une table de codage numérique.

Une interprétation géométrique

Prenons l'exemple de 1110 pour décrire le code de Hamming. Nous répartissons ces quatre bits dans un tableau carré, en écrivant de gauche à droite et de bas en haut (voir la figure ci-dessous). Nous ajoutons alors un bit de parité à chaque ligne et chaque colonne. On obtient un octet, ici 11010101.

1	**1**	0
1	**0**	1
0	1	

Répartition des quatre bits 1110 en tableau (en gras) et addition des bits de parité (en maigre). Nous obtenons l'octet 11010101.

Admettons que cet octet soit transmis avec une erreur : 10010101. Nous le disposons en tableau comme précédemment. Deux erreurs de parité apparaissent alors, une sur la colonne bordante à droite, l'autre sur la ligne bordante en dessous. L'erreur se trouve à l'intersection des deux, il est donc facile de rétablir l'octet transmis (voir la figure ci-après). Cela se généralise à tous les cas où il apparaît deux erreurs de parité. Si une seule a été commise, il est possible de la corriger. S'il n'apparaît qu'une erreur de parité, c'est le caractère de contrôle qui est erroné.

1	0	**0**
1	0	1
0	**1**	

Détection d'une erreur. Deux erreurs de parité apparaissent (en gras). L'erreur de transmission se trouve à l'intersection de la ligne et de la colonne concernée, ce qui permet de la corriger.

Pour coder un texte, il suffit de mettre les codes des lettres utilisées les uns à la suite des autres. Avec la table de codage précédente, le texte « les fouilles sont visibles » devient la suite suivante de 130 bits :

0110000101100110000000110011111
0101010010110001100001011001100
0001001101111011101010000000101
1001001100110100100010011000010
110011.

Une erreur de transmission donne un texte différent. Par exemple, si le quatre-vingt-treizième bit est transmis de façon erronée, le texte devient :

0110000101100110000000110011111
0101010010110001100001011001100
0001001101111011101010000000100
1001001100110100100010011000010
110011,

c'est-à-dire « les fouilles sont risibles ».

Distance linguistique

La détection des erreurs est fondée sur une notion de distance. Plus précisément, nous définissons la distance entre deux mots comme le nombre minimal de bits à modifier pour passer de l'un à l'autre. Par exemple, la

Richard Hamming (1915–1998)

Richard Hamming a découvert les codes correcteurs d'erreur qui portent aujourd'hui son nom alors qu'il travaillait aux laboratoires Bell, en 1950. Son œuvre porte également sur l'informatique théorique et sur l'analyse numérique.

distance de 0110 à 1010 est égale à 2 puisqu'il faut modifier les deux premiers bits pour passer du premier mot au second.

Pour rendre possible la détection d'une erreur, il suffit d'appliquer aux messages une transformation telle que deux textes transformés soient toujours à une distance au moins égale à 2. Ainsi, une erreur ne peut faire passer d'un texte valide à un autre. C'est le cas si la somme des bits des messages transformés est toujours paire. Une idée simple pour le réaliser est de transformer les textes en leur ajoutant le bit 0 si la somme de ses bits est paire et 1 si elle est impaire. Dans l'exemple précédent, on ajoute donc le bit 1 en queue, d'où le message :

01100001011001100000000110011111
01010100101100011000010110011 00
00010011011110111010100000000101
1001001100110100100010011000010
1100111.

Une erreur de transmission est automatiquement détectée puisque la parité n'est alors plus respectée. Bien entendu, deux erreurs de transmission peuvent se compenser et être ainsi indétectables. Pour qu'elles le soient, il faudrait que deux textes transformés soient toujours à une distance au moins égales à 3 l'un de l'autre.

Découpage des textes

Dans la pratique, les textes sont découpés en morceaux plus petits. Si nous utilisons un découpage par groupes de cinq bits, c'est-à-dire par lettres, nous pouvons ajouter à chacun un bit de parité, ce qui revient à utiliser la table de codage :

<Espace>	000000	*h*	010001	*p*	100001	*x*	110000
a	000011	*i*	010010	*q*	100010	*y*	110011
b	000101	*j*	010100	*r*	100100	*z*	110101
c	000110	*k*	010111	*s*	100111	,	110110
d	001001	*l*	011000	*t*	101000	.	111001
e	001010	*m*	011011	*u*	101011	!	111010
f	001100	*n*	011101	*v*	101101	?	111100
g	001111	*o*	011110	*w*	101110	:	111111

Table de codage numérique avec bit de parité : les codes sont à distance mutuelle au moins égale à 2.

Notre message devient :

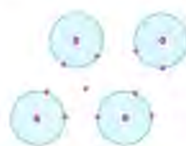
011000000101010011100000000011000
1111010101101001001100001100000
1010100111000000010011101111100111
01101000000000010110101001010011
1010010000101011000001010100111.

Pour décoder le message, il suffit de le

découper en mots de six bits puis de vérifier que la somme des bits est paire. Si une erreur (au plus) est commise dans chacun de ces mots, toutes les erreurs sont détectées. Dans ce cas, on pourra demander un nouvel envoi des mots erronés pour les corriger. Cette méthode est simple mais n'est utilisable que s'il est possible de renvoyer un message de contrôle. Parfois, c'est impossible comme dans le cas de messages stockés (musiques ou films sur CD ou DVD). C'est également le cas pour un message envoyé par une sonde spatiale. Dans ces cas, il vaut mieux pouvoir *corriger* les erreurs.

Codage auto-correcteur

Pour qu'un codage puisse corriger une erreur, il suffit que les différents codes soient à des distances mutuelles au moins égales à 3 (voir la figure *Les mots valides*).



Les mots valides sont représentés par des gros points rouges, et les mots non valides par des petits. Autour de chaque mot valide, le cercle des mots à distance égale à 1. Ces cercles sont distincts deux à deux. S'ils sont le résultat d'une et une seule erreur, les mots non valides correspondent au mot valide au centre du cercle.

Combien de bits faut-il utiliser pour cela ? Pour chaque mot comme 00000, on considère l'ensemble des mots obtenus en changeant au plus un bit. Ils sont au nombre de 6. Comme nous avons 32 mots de cinq bits au départ, il nous faut utiliser 6 x 32 = 192 mots. Huit bits peuvent donc suffire ; mais, dans ce cas, l'entourage de chaque mot en contient 9, ce qui donne 9 x 32 = 288 mots. Il nous faut donc au moins neuf bits !

Une idée est de garder les cinq premiers et de compléter par quatre bits de façon à rester à distance au moins égale à 3 des autres. Voici un exemple d'un tel code :

<Espace>	000000000	*h*	010001110	*p*	100001011	*x*	110001101
a	000010110	*i*	010011000	*q*	100010001	*y*	110010111
b	000101100	*j*	010100010	*r*	100100111	*z*	110100001
c	000111010	*k*	010110100	*s*	100111101	,	110111011
d	001001010	*l*	011000100	*t*	101000101	.	111000011
e	001011100	*m*	011010010	*u*	101011111	!	111011001
f	001100110	*n*	011101000	*v*	101101001	?	111101111
g	001110000	*o*	011111110	*w*	101110011	:	111110101

Table de codage numérique avec distances mutuelles égales à 3 au moins.

Notre message ainsi codé devient :

0110001000010111001001111010000
0000000110011001111111010101111
0100110000110001000110001000010
1110010011110100000000010011110
1011111111001110100010100010100
0000000101101001010011000100111
1010100110000001011000110001000
01011100.

Il est facile de dresser la liste des codes à une distance égale à 1 des codes ci-dessus avec la lettre correspondante.

Ce tableau sert alors au décodage. Il est trop volumineux pour être dressé ici, mais en voici le début :

000000000	<Espace>	000010110	*a*	000101100	*b*	000111010	*c*
000000001	<Espace>	000010111	*a*	000101101	*b*	000111011	*c*
000000010	<Espace>	000010100	*a*	000101110	*b*	000111000	*c*
000000100	<Espace>	000010010	*a*	000101000	*b*	000111110	*c*
000001000	<Espace>	000011110	*a*	000100100	*b*	000110010	*c*
000010000	<Espace>	000000110	*a*	000111100	*b*	000101010	*c*
000100000	<Espace>	000110110	*a*	000001100	*b*	000011010	*c*
001000000	<Espace>	001010110	*a*	001101100	*b*	001111010	*c*
010000000	<Espace>	010010110	*a*	010101100	*b*	010111010	*c*
100000000	<Espace>	100010110	*a*	100101100	*b*	100111010	*c*

Début de la table de correction et de décodage.

Quand l'algèbre s'en mêle

Rien n'oblige à découper le texte en groupe de cinq bits correspondant aux lettres, on peut le découper en trois, quatre, cinq ou six également. Un codage classique consiste à découper le texte en groupes de quatre bits. Ici, on ajoute donc deux bits à la fin (deux « 0 » par exemple), et on découpe notre texte en 33 mots de quatre bits :

0110 0001 0110 0110 0000 0011 0011 1110 1010 1001 0110 0011 0000 1011 0011 0000 0100 1101 1110 1110 1010 0000 0010 1100 1001 1001 1010 0100 0100 1100 0010 1100 1100.

Richard Hamming a proposé un codage simple à exposer géométriquement (voir l'encadré Une interprétation géométrique) mais que nous exposons ici de façon algébrique afin de montrer qu'il se généralise. Il consiste à transformer chaque groupe de quatre bits (x_1, x_2, x_3, x_4) en le mot de huit bits :

$$f(x_1, x_2, x_3, x_4) = (x_1, x_2, x_3, x_4, x_1 + x_2, x_3 + x_4, x_1 + x_3, x_2 + x_4)$$

où l'addition se fait suivant la table de Pythagore suivante :

+	0	1
0	0	1
1	1	0

Table de Pythagore de l'addition utilisée.

Le premier mot du texte est 0110, soit $x_1 = 0, x_2 = 1, x_3 = 1$ et $x_4 = 0$. Il est donc transformé en 01101111. Le mot 0010 est transformé en 00100110. Alors que les deux mots 0110 et 0010 sont à une distance égale à 1, les deux mots images 01101111 et 00100110 sont à une distance égale à 3. Ce résultat est général : deux codes distincts génèrent deux images à distance au moins égale à 3 (voir l'encadré Des démonstrations esthétiques). Notre code permet donc de corriger une erreur. On peut dresser une table de correction comme dans le cas précédent, mais on peut également donner ici une formule de correction.

Travail dans l'image

Un texte transformé vérifie :

$$(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8) = (x_1, x_2, x_3, x_4, x_1 + x_2, x_3 + x_4, x_1 + x_3, x_2 + x_4)$$

et donc les quatre relations :

$$y_1 + y_2 + y_5 = 0,\ y_3 + y_4 + y_6 = 0,$$

$$y_1 + y_3 + y_7 = 0,\ y_2 + y_4 + y_8 = 0.$$

Si un texte reçu ne vérifie pas ces équa-

Des démonstrations esthétiques

Si (x_1, x_2, x_3, x_4) est un mot de quatre bits, les autres mots de quatre bits sont tous de la forme $(x_1 + d_1, x_2 + d_2, x_3 + d_3, x_4 + d_4)$ où (d_1, d_2, d_3, d_4) est un mot de quatre bits. La distance entre les deux mots est égale au nombre de bits (d_1, d_2, d_3, d_4) égaux à 1. La linéarité des formules montre que :

$$f(x_1 + d_1, x_2 + d_2, x_3 + d_3, x_4 + d_4) = f(x_1, x_2, x_3, x_4) + f(d_1, d_2, d_3, d_4)$$

et la distance entre les deux transformés est égale au nombre de bits égaux à 1 parmi :

$$f(d_1, d_2, d_3, d_4) = (d_1, d_2, d_3, d_4, d_1 + d_2, d_3 + d_4, d_1 + d_3, d_2 + d_4).$$

Si un seul des bits (d_1, d_2, d_3, d_4) est égal à 1, par exemple d_1, trois des bits transformés sont égaux à 1 : d_1, $d_1 + d_2$ et $d_1 + d_3$. Il en est de même dans chaque autre cas puisque chaque bit apparaît trois fois.

Si deux des bits (d_1, d_2, d_3, d_4) sont égaux à 1, par exemple d_1 et d_2, quatre des bits transformés sont égaux à 1 : d_1, d_2, $d_1 + d_3$ et $d_2 + d_4$. Il en est de même dans les autres cas par symétrie.

Si trois ou quatre des bits (d_1, d_2, d_3, d_4) sont égaux à 1, la question est déjà réglée. Finalement, les mots transformés sont à des distances mutuelles au moins égales à 3.

De même, si le mot transformé valide

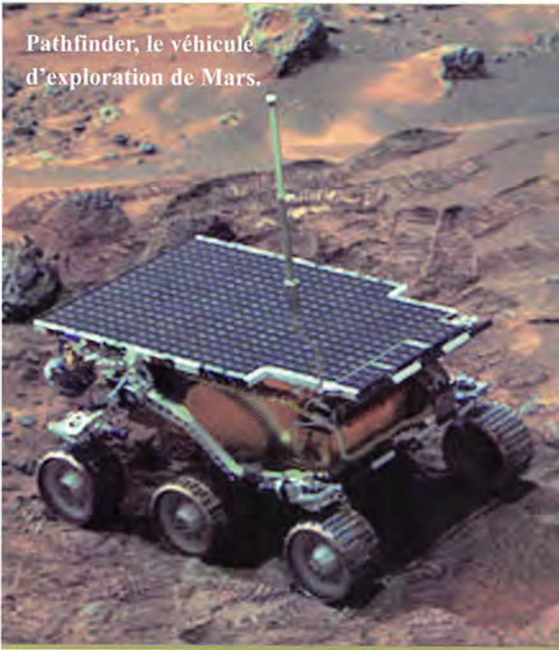
$$(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8) = (x_1, x_2, x_3, x_4, x_1 + x_2, x_3 + x_4, x_1 + x_3, x_2 + x_4)$$

est transmis avec une et une seule erreur, cela revient à lui ajouter $(d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8)$ où un seul de ces bits est égal à 1. On calcule alors :

$$g(y_1 + d_1, y_2 + d_2, y_3 + d_3, y_4 + d_4, y_5 + d_5, y_6 + d_6, y_7 + d_7, y_8 + d_8)$$

$$= g(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8) + g(d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8).$$

Or $g(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8) = 0$ puisque $(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8)$ est valide. On obtient donc $g(d_1, d_2, d_3, d_4, d_5, d_6, d_7, d_8)$, c'est-à-dire l'une des images des mots (1, 0, 0, 0, 0, 0, 0, 0), (0, 1, 0, 0, 0, 0, 0, 0), …, (0, 0, 0, 0, 0, 0, 0, 1), soit l'un des mots 1010, 1001, 0110, 0101, 1000, 0100, 0010, 0001. Comme ces huit mots sont deux à deux distincts, cela permet de rétablir le bit erroné.


Pathfinder, le véhicule d'exploration de Mars.

Exploration de l'espace et codes correcteurs

Quand une sonde martienne envoie des images sur la terre, au cas où l'une serait détériorée, il serait long de lui demander de répéter son message. Il est donc impératif que l'image arrive en bon état. Pour cela, on utilise des codes correcteurs d'erreurs.

tions, il est erroné. Nous considérons donc la transformation :

$$(z_1, z_2, z_3, z_4) = g(y_1, y_2, y_3, y_4, y_5, y_6, y_7, y_8)$$

$$= (y_1 + y_2 + y_5,\ y_3 + y_4 + y_6,\ y_1 + y_3 + y_7,\ y_2 + y_4 + y_8).$$

Si un texte reçu n'est pas transformé en 0000 par g, il comporte forcément une erreur. Prenons l'exemple du texte 0110 codé en 01101111. S'il est reçu correctement, on obtient bien 0000. Supposons qu'il soit transmis avec une erreur au premier bit, c'est-à-dire que l'on reçoive 11101111. Il est transformé en 1010. On démontre que ceci est la signature d'une erreur au bit 1. De façon générale, les différentes signatures sont :

Une démonstration de ce résultat est proposée dans l'encadré *Des démonstrations esthétiques*. Nous resterions dans l'esthétisme si ce type de techniques algébriques ne se généralisait pas. En fait, elle permet de créer des codes plus performants utilisant des découpages en mots plus grands. Ils sont utilisés aussi bien dans le codage des CD et DVD que dans les communications spatiales.

H. L.

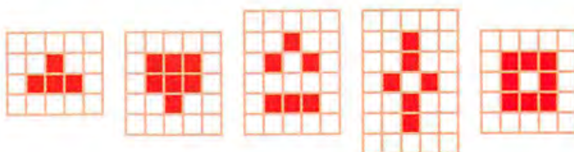
bit	1	2	3	4	5	6	7	8
signature	1010	1001	0110	0101	1000	0100	0010	0001

Solutions des jeux en pages 102 et 106.

Au bout de seulement 12 mélanges on retrouvera l'ordre initial.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
5	9	13	2	6	10	14	3	7	11	15	4	8	12	16
3	5	7	9	11	13	15	2	4	6	8	10	12	14	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
5	9	13	2	6	10	14	3	7	11	15	4	8	12	16
3	5	7	9	11	13	15	2	4	6	8	10	12	14	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
5	9	13	2	6	10	14	3	7	11	15	4	8	12	16
3	5	7	9	11	13	15	2	4	6	8	10	12	14	16
2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

La population A donne naissance à une population stable de quatre cellules disposées en carré.
La population B évolue très vite : voici les premières étapes de cette évolution.



L'algorithme PageRank

Le fonctionnement du moteur de recherche américain Google repose sur un algorithme mathématique qui a été longuement disséqué dans *Tangente*. Voilà un bel exemple d'application des maths (et en particulier des algorithmes) à la vie quotidienne de millions de personnes !

Comment Google classe les pages ? Jacques Bair, *Tangente Sup* n° 44–45, pp. 10–13, septembre–octobre 2008.

L'algorithme PageRank de Google : promenade sur la Toile. Michael Eisermann, *Tangente* n° 130, pp. 44–47, septembre–octobre 2009.

RÉFÉRENCES

Arthur Engel. *Mathématiques élémentaires d'un point de vue algorithmique*, CÉDIC, 253 pages, 1979.

Michael W.Ecker. *Mathemagical Black Holes, in: Tribute to Martin Gardner*, A.K. Peters Limited, 266 pages, 1993.

par Hervé Lehning

La multiplication rapide

Les méthodes de cryptographie en usage sur Internet demandent à effectuer des multiplications de nombres de plusieurs centaines de chiffres. La méthode apprise à l'école montre vite ses limites. Dès lors, comment exécuter ces calculs dans un temps raisonnable ?

L'écriture d'un entier A en base N ressemble à celle d'un polynôme :

$$A = a + b\,N + c\,N^2 + etc.$$

où a, b, c, *etc.* sont des nombres entiers compris entre 0 et $N - 1$. Pour multiplier deux nombres à n chiffres, en utilisant l'algorithme classique enseigné à l'école élémentaire, nous effectuons n^2 multiplications de nombres à un chiffre, puis environ n additions. On vérifie effectivement que, en utilisant un ordinateur, le temps de calcul est proportionnel à n^2. Bien entendu, ceci n'est visible que pour les nombres de plusieurs milliers de chiffres, mais si vous *doublez* le nombre de chiffres, vous multipliez le temps de calcul par *quatre*.

Une stratégie du type **veni, divisi, vici.**

Transformation de Fourier

À un nombre A de n chiffres, nous associons le n-uplet de nombres complexes T(A) :

$$A(1), A(\omega), A(\omega^2), \ldots, A(\omega^{n-1})$$

où $A(X) = a + b\,X + c\,X^2 + etc.$

et $\omega = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$

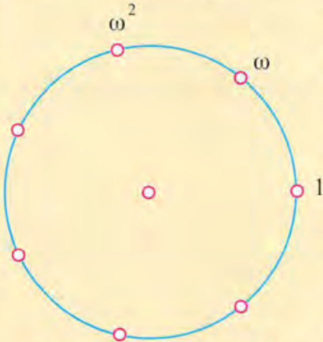
(voir l'encadré Les racines de l'unité). Ce calcul demande *a priori* n^2 multiplications, son temps de calcul est donc proportionnel à n^2. Une méthode plus subtile permet de l'effectuer dans un temps proportionnel à $n \ln n$, ce qui est beaucoup plus rapide. Par exemple, cela fait passer d'un million à sept mille, d'un milliard à trois cent mille. C'est un gain considérable ! Avant d'examiner cette méthode, voyons l'intérêt de la transformation T appelée *transformation de Fourier discrète*.

Son avantage principal est d'être

Les racines de l'unité

D'après la formule de Moivre, si $\omega = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$,

alors $\omega^k = \cos\frac{2k\pi}{n} + i\sin\frac{2k\pi}{n}$, donc 1, ω, ω^2, …, ω^{n-1} sont n racines énièmes distinctes de l'unité. De plus, l'exponentiation correspond à une rotation. Ainsi, dans le plan complexe, les nombres précédents sont aux sommets d'un polygone régulier.



Les racines septièmes de l'unité sont aux sommets d'un heptagone régulier, leur centre de gravité est donc 0.

Nous les avons ainsi toutes énumérées.

D'autre part :
$(1 - \omega)(1 + \omega + \omega^2 + \ldots + \omega^{n-1}) = 1 - \omega^n = 0$.
Autrement dit, la somme des racines énièmes de l'unité est nulle. De même si nous remplaçons ω par ω^k où k est compris entre 1 et $n - 1$.

inversible, c'est-à-dire que la connaissance de T(A) permet de reconstituer A. Le calcul consiste simplement à résoudre le système d'équations :
$a + b + c + \ldots = \mathrm{A}(1)$,
$a + b\,\omega + c\,\omega^2 + \ldots = \mathrm{A}(\omega)$,
…
$a + b\,\omega^{n-1} + c\,\omega^{2(n-1)} + \ldots = \mathrm{A}(\omega^{n-1})$,
où les inconnues sont a, b, c, *etc.* Comme la somme des racines énièmes de l'unité est nulle (voir l'encadré), il suffit d'ajouter toutes les équations pour obtenir a :

$$a = \frac{\mathrm{A}(1) + \mathrm{A}(\omega) + \mathrm{A}(\omega^2) + \cdots + \mathrm{A}(\omega^{n-1})}{n}.$$

De même, nous obtenons b en multipliant la dernière équation par ω, l'avant dernière par ω^2, *etc.* :

$$b = \frac{\mathrm{A}(1) + \omega^{n-1}\mathrm{A}(\omega) + \omega^{n-2}\mathrm{A}(\omega^2) + \cdots + \omega\,\mathrm{A}(\omega^{n-1})}{n}$$

et ainsi de suite. Le passage de T(A) à A est de même nature que celui de A à T(A), la seule différence est la division finale par n. La méthode rapide permet de l'effectuer dans un temps proportionnel à $n \ln n$.

Multiplication par transformée de Fourier

Étant donnés deux nombres A et B, nous pouvons calculer leurs transformées de Fourier respectives T(A) et T(B) dans un temps proportionnel à $n \ln n$. Nous les multiplions alors pour obtenir C(1) = A(1) B(1), C(ω) = A(ω) B (ω), *etc.*, ce qui demande n multiplications. Il est alors facile de reconstituer C en un temps proportionnel à $n \ln n$. Nous obtenons donc le produit C = A B en un temps proportionnel à $n \ln n$ au lieu de n^2. L'intérêt principal de la transformée de Fourier

Un calcul de complexité

Considérons une fonction c telle que $c(0) = c(1) = 1$ et $c(2n) = 2\ c(n) + n$ pour tout $n \geq 1$. Si nous posons $u_n = \frac{c(2^n)}{2^n}$, cette suite vérifie la relation de récurrence $u_{n+1} = u_n + \frac{1}{2}$ et $u_0 = 1$, d'où : $u_n = (n + 1)/2$, et donc $c(2^n) = 2^{n-1}(n + 1)$. On démontre alors, grâce à la relation de récurrence, que la fonction c est croissante. Donc, si $2^p - n - 2^{p+1}$, (*) alors :
$\frac{p+1}{2}2^p \leq c(n) \leq \frac{p+2}{2}2^{p+1}$.
En utilisant la croissance de la fonction logarithme et (*), nous en déduisons la double inégalité $\frac{n \ln n}{4 \ln 2} \leq c(n) \leq \frac{n(\ln n + 2 \ln 2)}{\ln 2}$, qui prouve que $c\ (n)$ est de l'ordre de grandeur de $n \ln n$.

discrète se situe là : gagner du temps dans les calculs arithmétiques. Ceux-ci sont indispensables en cryptographie, qui est d'utilisation constante, en particulier sur Internet.

Veni, divisi, vici

Soit $2n$ le nombre de chiffres de A (au besoin, on ajoute un zéro en tête), et séparons les termes d'ordre pair de ceux d'ordre impair. En considérant les polynômes associés, nous obtenons :

$$A(X) = A_0(X^2) + X\, A_1(X^2)$$

où A_0 et A_1 sont de degré n.
La formule $\omega = \cos\frac{2\pi}{2n} + i \sin\frac{2\pi}{2n}$
implique $\omega^2 = \cos\frac{2\pi}{n} + i \sin\frac{2\pi}{n}$,
donc :
$A(1) = A_0(1) + A_1(1)$,
$A(\omega) = A_0(\omega^2) + \omega\, A_1(\omega^2)$,
…
$A(\omega^{n-1}) = A_0(\omega^{2(n-1)}) + \omega^{n-1}\, A_1(\omega^{2(n-1)})$.
Ces formules montrent que, pour calculer une transformée de Fourier discrète d'un nombre à $2n$ chiffres, il suffit d'en calculer deux de nombres à n chiffres, puis d'effectuer n multiplications et n additions. Si nous notons $c(n)$ le nombre de multiplications nécessaires pour calculer la transformée de Fourier discrète d'un nombre de n chiffres, la fonction c vérifie :

$$c(2n) = 2\ c(n) + n.$$

En résolvant cette relation de récurrence, on montre que $c(n)$ est de l'ordre de grandeur de $n \ln n$ (voir l'encadré Un calcul de complexité).
Le calcul se fait donc de façon récursive : celui de T(A) appelle ceux de T(A_0) et T(A_1), et ainsi de suite jusqu'au cas évident où les nombres n'ont qu'un chiffre. Nous obtenons ainsi T(A), et en déduisons A.

H. L.

Multiplications de matrices

Les idées exposées dans l'article sont applicables à la multiplication matricielle, qui trouve des applications en météorologie par exemple. Une matrice carrée M de taille 2 est la donnée d'un tableau de quatre nombres, que l'on note $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$.

La multiplication matricielle est définie de la façon suivante :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} a\alpha + b\gamma & a\beta + b\delta \\ c\alpha + d\gamma & c\beta + d\delta \end{pmatrix}.$$

Elle demande donc *a priori* huit multiplications de nombres, ainsi que quatre additions. Les multiplications exigeant beaucoup plus de temps que les additions, nous négligeons ces dernières. Nous passons aux matrices carrées de taille 4 en remplaçant chaque nombre dans M par une matrice de taille 2. Nous obtenons ainsi la multiplication des matrices de taille 4. En itérant le procédé, nous atteignons les matrices de taille 8, 16…

Volker Strassen réduisit le nombre de multiplications de 8 à 7 en calculant :

$$m_1 = (a + d) \,.\, (\alpha + \delta),\ m_2 = (c + d) \,.\, \alpha,\ m_3 = a \,.\, (\beta - \delta),\ m_4 = d \,.\, (\gamma - \alpha),\ m_5 = (a + b) \,.\, \delta,$$
$$m_6 = (c - a) \,.\, (\alpha + \beta),\ m_7 = (b - d) \,.\, (\gamma + \delta).$$

Dans ce cas, le produit s'exprime :

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 - m_2 + m_3 + m_6 \end{pmatrix}.$$

Comme celui de la multiplication rapide, l'algorithme de Strassen est appliqué de façon récursive. Si n est la taille des matrices, nous passons d'une complexité proportionnelle à n^3 (pour l'algorithme usuel) à $n^{\log_2 7}$, soit environ $n^{2,807}$.

Le gain est intéressant pour les très grandes matrices, comme celles utilisées en météorologie ou dans l'aéronautique.

Le mathématicien allemand Volker Strassen (né en 1936).

Niveau de difficulté
- ❍ très facile
- ✓ facile
- ✓✓ pas facile
- ✓✓✓ difficile
- ✓✓✓✓ très difficile

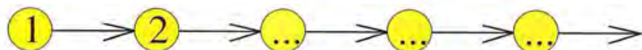
Algorithmes en folie

HS3è01 - Produit + somme ❍

Mathilde écrit un nombre à deux chiffres, par exemple 38. Elle multiplie les deux chiffres (ici, $3 \times 8 = 24$) ; les additionne (ici, $3 + 8 = 11$) ; enfin elle ajoute les deux résultats et elle écrit le nouveau nombre obtenu (ici, 35). Elle recommence les mêmes calculs à partir de ce nouveau nombre pour en obtenir un troisième qu'elle écrit (ici, 23).
Si Mathilde part du nombre 75 qu'elle écrit en premier, quel sera le 2 010ᵉ nombre écrit ?

HS3702 - Dites 43 ✓

Mathilde écrit les nombres 1 et 2, comme sur le dessin ci-dessous.



Ensuite, elle s'impose la règle suivante :
- Soit elle double le dernier nombre écrit et elle le place dans le rond suivant ;
- Soit elle additionne les deux derniers nombres écrits et note la somme dans le rond suivant.

Son but est d'arriver à écrire 43 ou, à défaut, à un nombre le plus proche possible de 43.
Mathilde s'arrête lorsqu'elle a atteint son but.
Quelle peut être la suite des nombres écrits par Mathilde ?

HS3703 - Somme des carrés des écrits ✓✓

Choisissez un nombre de départ ayant au plus trois chiffres et écrivez-le.
Additionnez les carrés de ses chiffres : vous obtiendrez un deuxième nombre que vous écrirez.
Recommencez l'opération consistant à additionner les carrés des chiffres du dernier nombre écrit, et écrivez le résultat tant que celui-ci n'est pas égal à un nombre déjà écrit.
Exemple : 409 ; puis 97 ; 130 ; 10 ; et finalement 1.
Quelle est la liste de nombres la plus longue possible ?

HS3704 - Les nombres consistants ✓✓

Le père Sistant adore jouer avec les nombres. Son jeu favori consiste à partir d'un nombre, à calculer le produit de ses chiffres, puis à recommencer avec le nombre obtenu, ceci jusqu'à

obtention d'un nombre à un seul chiffre. Voici trois exemples de jeu :
$23 \to 2 \times 3 = 6$
$54 \to 5 \times 4 = 20 \to 2 \times 0 = 0$
$999 \to 9 \times 9 \times 9 = 729 \to 7 \times 2 \times 9 = 126 \to 1 \times 2 \times 6 = 12 \to 1 \times 2 = 2$
On appelle *persistance d'un nombre* le nombre d'étapes nécessaires pour obtenir un nombre à un seul chiffre. Ainsi, la persistance de 6 est 0, celle de 23 est 1, celle de 54 est 2, et celle de 999 est 4. Le père Sistant s'intéresse particulièrement aux nombres dont la persistance est supérieure ou égale à 4, qu'il appelle des *nombres consistants*.
Quel est le plus petit des nombres consistants ?

HS3705 – Le message ✓✓

Dans le message suivant, on a utilisé un code à substitution, c'est-à-dire que chaque lettre du texte en clair a été remplacée par une autre lettre suivant un décalage. On rappelle que pour un décalage de $n = 3$, A devient D, B devient E, *etc.* et que l'on considère que la lettre A suit la lettre Z.
Ici, la valeur de n change à chaque lettre.

Le procédé utilisé est une substitution variable, inspirée de l'algorithme de Collatz ; on passe de n_1 à n_2 de la façon suivante :
- Si n_1 est pair, alors $n_2 = n_1/2$;
- Si n_1 est impair, alors $n_2 = 3\,n_1 + 1$.

Soit le message :
«**KRAWTVZGBAAMELRGWYUU**».
À vous de le décrypter.

HS3706 – Combien de termes ✓✓✓✓

On considère l'algorithme suivant :
- Étape n° 0

Vous disposez du nombre 1.
Vous choisissez deux nombres positifs différents a et b de somme 1.
- Étape n° 1

Vous multipliez le nombre 1 par la somme $(a + b)$, et vous obtenez $a + b$, qui est constituée de deux termes différents.
- Étape n° 2

Vous multipliez $(a + b)$ par $(a + b)$, ce qui vous donne, d'après une identité remarquable bien connue :
$a^2 + b^2 + ab + ab$. (On s'interdit d'écrire des coefficients numériques, tels le « 2 » de « $2ab$ »).
La nouvelle somme obtenue contenant deux termes identiques (ab et ab), on multiplie à nouveau l'un de ces deux termes par $(a + b)$, ce qui donne :
$a^2 + b^2 + ab + a^2b + ab^2$.
Nous avons finalement une somme de cinq termes tous différents (sans coefficients).
- Étape n° 3

Vous multipliez $(a^2 + b^2 + ab + a^2b + ab^2)$ par $(a + b)$, en n'utilisant pas de coefficients, puis vous faites la chasse aux termes en double en les multipliant par $(a + b)$ jusqu'à épuisement de ces doublons.
Combien obtiendrez-vous de termes tous différents (sans coefficients) à la fin de cette étape numéro 3 ?

HS3707 – Triplets équilibrés ✓✓✓✓

Un triplet de nombres positifs irrationnels $(x_1 ; x_2 ; x_3)$ tels que $x_1 + x_2 + x_3 = 1$ est dit *équilibré* si chacun des x_i est strictement inférieur à 1/2. Si un triplet n'est pas équilibré, on lui applique la « procédure d'équilibrage » suivante :
$(x_1 ; x_2 ; x_3) \mapsto (x'_1 ; x'_2 ; x'_3)$ où $x'_i = 2x_i$ si $x_i < 1/2$ et $x'_i = 2x_i - 1$ si $x_i > 1/2$.
Si le nouveau triplet n'est pas équilibré, on lui applique à nouveau la

procédure d'équilibrage.
L'application répétée de la procédure d'équilibrage conduit-elle toujours à un triplet équilibré après un nombre fini d'étapes ?

HS3708 – Un algorithme « cosinusoïdal » ✓✓✓

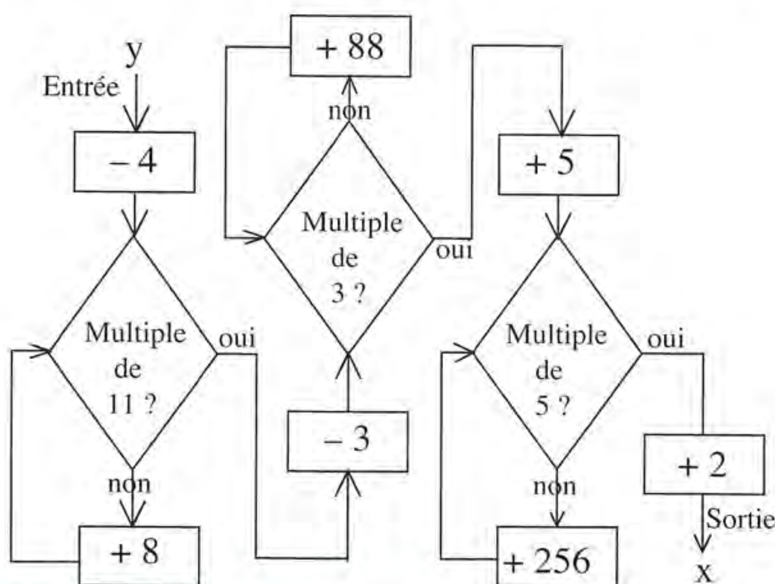
Soient m et n deux nombres entiers premiers entre eux et Θ un nombre réel. On considère l'algorithme suivant :
(1) Initialisation : $(A ; B ; C ; d ; e) = (2\cos\Theta ; 2\cos\Theta ; 2 ; n - m ; m)$.
(2) Si $d > e$, remplacer $(B ; C ; d)$ par $(AB - C ; B ; d - e)$;
sinon remplacer $(A ; C ; e)$ par $(AB - C ; A ; e - d)$.
(3) Si $e = 0$, fin de l'algorithme ;
sinon aller en (2).
Démontrer que $A = 2\cos(n\Theta)$ lorsque l'algorithme se termine.

HS3709 – Drôles de machines ✓✓✓

Mathias a apporté trois machines à fabriquer des nombres. Il les a mises en série comme sur le dessin ci-dessous.
Francis demande à Mathias : **Combien de valeurs de y vont donner 3 997 à la sortie ?**

HS3710 – La boucle infernale ✓✓✓

Prenez un nombre de quatre chiffres, par exemple 1 990.
Additionnez les premier et quatrième chiffres de ce nombre, et prenez le chiffre des unités u du résultat.
Formez ensuite un nouveau nombre de quatre chiffres en plaçant à droite des trois derniers chiffres du nombre de départ le chiffre u.
Si vous recommencez ce procédé, vous ne retrouverez le nombre 1 990 qu'après très exactement 1 560 étapes, les premières étant successivement 9 901, 9 010, 0 109, 1 099, 0 990, 9 900, 9009…
Trouvez un nombre de quatre chiffres qui ne soient pas tous pairs, ne commençant pas par un zéro, et tel qu'en itérant l'algorithme décrit ci-dessus, on retrouve le nombre de départ avant vingt étapes.



HS3711 – L'arbre généalogique ✔

a) Je veux faire mon arbre généalogique ascendant, c'est-à-dire celui qui contient mes deux parents (1re génération), mes quatre grands-parents (2e génération)… jusqu'à mes ancêtres de la cinquième génération.
Combien cet arbre comporte-t-il de personnes, moi inclus ?

b) On numérote ainsi les membres de l'arbre : je porte le numéro 1, mon père le numéro 2, ma mère le numéro 3 et les parents du numéro n sont numérotés $2n$ pour le père et $2n + 1$ pour la mère. Ainsi, le numéro 4 est le père de mon père.
Sur le même modèle et en utilisant uniquement les mots père, mère, de, la, du, mon, préciser qui porte le numéro 37.

GENEALOGICAL TREE OF THE QUEEN AND HER DECENDANTS

HS3712 – On ne s'énerve pas ! ✔

Un participant au Rallye *Mathématiques sans frontières* s'énerve : il déchire en huit morceaux sa feuille de papier (1re étape), puis il prend un des morceaux qu'il déchire de nouveau en huit (2e étape), et ainsi de suite (à chaque étape, il prend un des morceaux et il le déchire en huit).
Au bout de combien d'étapes aura-t-il obtenu 2003 morceaux (en admettant qu'il puisse déchirer des morceaux même très petits) ?

HS3713 – Répétez 98 fois ✔✔

On choisit un nombre, on le double, puis on soustrait 1. On applique cette procédure à 98 reprises (en utilisant à chaque fois le nombre obtenu à l'étape précédente). On obtient finalement $2^{100} + 1$.
Quel était le nombre choisi au départ ?
A) 1 B) 2 C) 4 D) 6
E) Aucun des nombres précédents.

HS3714 – Successeur ✔✔✔

On considère une suite infinie de nombres entiers positifs telle que le successeur x' de x est obtenu en ajoutant à x l'un de ses chiffres non nuls.
Montrer que dans cette suite apparaît nécessairement un nombre pair.

SOURCE DES PROBLÈMES

Trophée Lewis Carroll (HS3701, HS3703)
Championnat des jeux mathématiques et logiques (HS3702, HS3704)
Maths en vacances, Élisabeth Busser, PLOT, **72**, pp. 45–47 (HS3705)
Championnat des jeux mathématiques et logiques (HS3709, HS3710)
Maths en vacances, PLOT, **72**, pp. 45–47 (HS3706)
William Lowell Putnam Mathematical Competition (HS3707)
Mathematics Magazine, Mathematical Association of America (HS3708)
Rallye mathématique d'Auvergne (HS3711)
Rallye mathématiques sans frontières Midi-Pyrénées (HS3712)
Concours Kangourou (HS3713)
Tournoi des villes (HS3714)

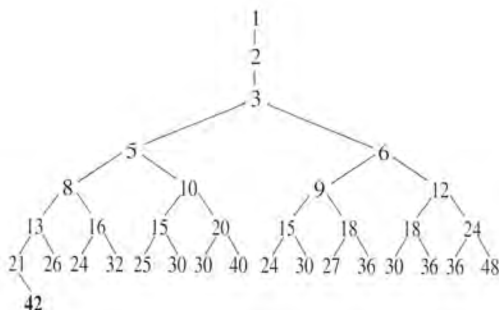
Solutions

HS3701

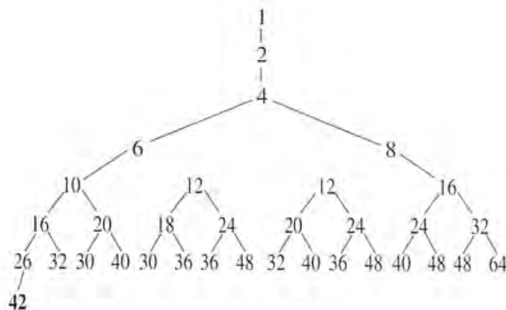
On obtient la suite 75 → 47 → 39 → 39 → 39 → ...., le nombre 39 se répétant ensuite indéfiniment. Le 2 010e nombre écrit sera donc **39.**

HS3702

On peut démarrer de deux façons : 123 ou 124. Explorons la première.



On vérifie, en explorant toutes les branches obtenues, qu'elle ne permet d'obtenir ni 43, ni 44. Par contre, on peut arriver à 42 avec la suite : 1 2 3 5 8 13 21 42.



Explorons la seconde voie 124.

On vérifie que celle-ci non plus ne permet d'atteindre ni 43, ni 44, mais qu'elle peut aboutir à 42 avec la suite 1 2 4 6 10 16 26 42.
Le problème a donc deux solutions :
1 2 3 5 8 13 21 42 ou 1 2 4 6 10 16 26 42.

HS3703

En partant d'un des nombres 799, 979 ou 997, on obtient la liste de 18 nombres suivante :
211 ; 6 ; 36; 45 ; 41; 17 ; 50 ; 25; 29 ; 85 ; 89 ; 145 ; 42 ; 20 ; 4 ; 16 ; 37 ; 58.
En partant d'un des nombres 667, 676, 766, 269, 296, 629, 692, 926 ou 962, on obtient la liste de 18 nombres suivante : **121 ; 6 ; 36; 45 ; 41; 17 ; 50 ; 25; 29 ; 85 ; 89 ; 145 ; 42 ; 20 ; 4 ; 16 ; 37 ; 58.**

HS3704

Les nombres à un chiffre ont tous pour persistance 0. Examinons ce qui se passe pour les nombres à deux chiffres. Pour cela, construisons un tableau des produits de deux chiffres (il s'agit tout simplement d'une table de multiplication, ou table de Pythagore !). Dans ce tableau, tous les nombres à un seul chiffre, en petits caractères, sont images de nombres à deux chiffres de persistance 1 (10, 20, 30, ..., 90, 11, 12, 21, 13, 31, ..., 19, 91, 22, 23, 32, 24, 42, 33). Les nombres à deux chiffres en caractères ajourés ont pour image un nombre de persistance 1 ; ils sont donc de persistance 2.
Les nombres en caractères gras ont pour image des nombres de persistance 2 ; ils sont donc de persistance 3.

Il reste alors un seul nombre dans le tableau : 49, image de 77. On vérifie que 77 est de persistance 4 : $77 \to 49 \to 36 \to 18 \to 8$.
Le plus petit nombre consistant est donc le nombre **77**, et le suivant est, bien sûr, **177**.

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

HS3705

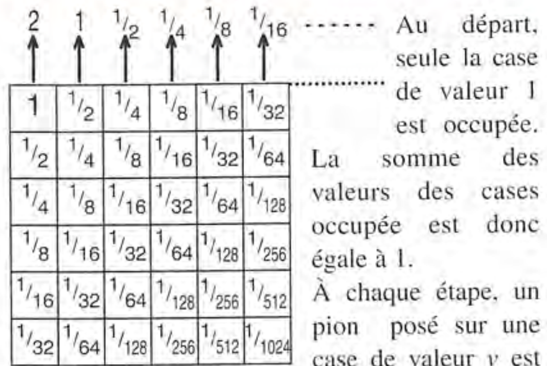
La suite obtenue à partir de 98 est la suivante :
$98 \to 49 \to 148 \to 74 \to 37 \to 112 \to 56 \to 28 \to 14 \to 7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.
Soit, modulo 26 :
$20 \to 23 \to 18 \to 22 \to 11 \to 8 \to 4 \to 2 \to 14 \to 7 \to 22 \to 11 \to 8 \to 17 \to 0 \to 0 \to 13 \to 14 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.
Le message décodé est donc : « QUIAINVENTELETELESCOPE » et la réponse est **Galilée**.

HS3706

On peut représenter tous les monômes en a et b sur un quart de plan en damier.

1	b	b^2	b^3	b^4	b^5
a	ab	ab^2	ab^3	ab^4	ab^5
a^2	a^2b	a^2b^2	a^2b^3	a^2b^4	a^2b^5
a^3	a^3b	a^3b^2	a^3b^3	a^3b^4	a^3b^5
a^4	a^4b	a^4b^2	a^4b^3	a^4b^4	a^4b^5
a^5	a^5b	a^5b^2	a^5b^3	a^5b^4	a^5b^5

Attribuons à chaque case du damier une valeur donnée par le dessin ci-dessous.



1	1/2	1/4	1/8	1/16	1/32
1/2	1/4	1/8	1/16	1/32	1/64
1/4	1/8	1/16	1/32	1/64	1/128
1/8	1/16	1/32	1/64	1/128	1/256
1/16	1/32	1/64	1/128	1/256	1/512
1/32	1/64	1/128	1/256	1/512	1/1024

Au départ, seule la case de valeur 1 est occupée. La somme des valeurs des cases occupée est donc égale à 1.
À chaque étape, un pion posé sur une case de valeur v est remplacé par deux pions posés chacun sur une case de valeur $v / 2$. D'après la règle de passage d'une étape à la suivante, la valeur totale des cases occupées ne varie donc pas : elle est toujours égale à 1.
La somme des valeurs de toutes les cases de la première colonne (colonne de gauche du damier) est égale à $1 + 1/2 + 1/4 + 1/8 + ... = 2$.
De même, celle de la seconde colonne vaut 1, celle de la troisième colonne, 1/2, celle de la quatrième, 1/4, et celle de la énième colonne, $1/2^n$.
La somme de toutes les cases du « damier–quart de plan » est donc égale à : $2 + 1 + 1/2 + 1/4 + 1/8 + ... = 4$.
L'étape 0 occupe l'unique case de valeur 1.
L'étape 1 occupe les deux cases de valeur 1/2.
L'étape 2 occupe les trois cases de valeur 1/4, et deux cases de valeur 1/8.
Ces trois étapes ont donc utilisé 8 cases dont la somme des valeurs égale 3.
Or, une case utilisée à une étape ne peut plus l'être dans les étapes suivantes (on ne se déplace que vers la droite et vers le bas).
La somme des valeurs de toutes les cases restantes du damier (une infinité de cases) est donc égale à 1. **On ne parviendra donc jamais à la fin de l'étape 3.**

HS3707

Il existe des triplets tels que l'application de la procédure ne conduit jamais à un triplet équilibré. Donnons-en un exemple où les x_i sont écrits en binaire.

$x_1 = 0{,}100\ 000\ 001\ 000\ 000\ 000\ 000\ 000\ 10...$ où les 1 occupent les seuls rangs 1, 9, 25, 49... des carrés de nombres impairs ;

$x_2 = 0{,}000\ 100\ 000\ 000\ 000\ 100\ 000\ 000\ 00...$ où les 1 occupent les seuls rangs 4, 16, 36, 64... des carrés de nombres pairs ;

$x_3 = 0{,}011\ 011\ 110\ 111\ 111\ 011\ 111\ 111\ 01...$ où les 1 occupent les seuls rangs 2, 3, 5, 6, 7, 8, 10... des nombres qui ne sont pas des carrés.

L'algorithme a pour effet de supprimer le premier chiffre après la virgule de chaque nombre et de décaler tous les autres d'un rang vers la gauche.

Le triplet est construit de telle sorte que pour chaque rang il y ait un et un seul des trois nombres tel que le chiffre de ce rang soit un « 1 ». On vérifie qu'il y a alors toujours un nombre strictement supérieur à 1/2 (celui dont le premier chiffre après la virgule est un « 1 ») et que la somme des trois nombres est égale à 1 = 0,111 111 111 11... Enfin, ces trois nombres, dont l'écriture en base deux ne comporte pas de période, sont bien des nombres irrationnels.

HS3708

Soient $(x ; y ; z)$ initialisés à (1 ; 1 ; 0). L'étape (2) de l'algorithme remplace $(y ; z)$ par $(x + y ; y)$ si $d > e$ et $(x ; z)$ par $(x + y ; x)$ sinon.

On vérifie qu'à chaque étape de l'algorithme, on a :

$xd + ye = n$;

$z = x - y$;

$A = 2\cos(x\Theta)$; $B = 2\cos(y\Theta)$; $C = 2\cos(z\Theta)$.

Les valeurs successives de d et e correspondent aux étapes de l'algorithme d'Euclide. L'algorithme s'arrête lorsque $\boldsymbol{e = 0, d = 1}$ **et** $\boldsymbol{x = n}$.

HS3709

Désignons par a, b et c les nombres de fois où il faut passer dans les trois boucles successives avant de pouvoir en sortir. On a :

$0 < a < 10$; $0 < b < 2$; $0 < c < 4$.

Lorsqu'on part de y, en suivant les opérations indiquées, on obtient le nombre : $y - 4 + 8a - 3 + 88b + 5 + 256c + 2 = y + 8a + 88b + 256c$, d'où l'équation $y = 3997 - 8a - 88b + 256c$.

De plus, écrivons les conditions de sortie des boucles :

• $y - 4 + 8a = 3993 - 88b - 256c$ doit être un multiple de 11.

On en déduit que $256c$ doit être un multiple de 11, ce qui implique que $c = 0$, étant donné que $c < 4$.

• $y - 4 + 8a - 3 + 88b = 3\,990$ doit être un multiple de 3, ce qui est toujours vrai.

• $y - 4 + 8a - 3 + 88b + 5 + 256c = 3\,995$ doit être doit être un multiple de 5, ce qui est également toujours vrai..

Les nombres cherchés sont donc les nombres de la forme :

$3\,997 - 8a - 11b = 3\,997 - 8\,(a + 11b)$.

Puisque a varie de 0 à 10 et b de 0 à 2, $a + 11b$ décrit tous les entiers de 0 à 32. Les nombres cherchés sont donc les nombres de la forme $3\,997 - 8k$, k variant de 0 à 32. Il existe donc trente-trois valeurs de y qui vont donner 3 997 à la sortie. **Ces 33 valeurs sont :**

3 741, 3 749, 3 757, 3 765, 3 773, 3 781, 3 789, 3 797, 3 805, 3 813, 3 821, 3 829, 3 837, 3 845, 3 853, 3 861, 3 869, 3 877, 3 885, 3 893, 3 901, 3 909, 3 917, 3 925, 3 933, 3 941, 3 949, 3 957, 3 965, 3 973, 3 981, 3 989, 3 997.

HS3710

Examinons d'abord la parité des chiffres successifs obtenus à partir d'un nombre de quatre chiffres comportant au moins un chiffre impair :

0	pppi	6	ipip	12	ppip
1	ppii	7	pipi	13	pipp
2	piii	8	ipii	14	ippp
3	iiii	9	piip	15	pppi
4	iiip	10	iipp		
5	iipi	11	ippi		

On constate que la séquence de départ, p p p i, ne réapparaît qu'après 15 étapes. D'autre part, toutes les séquences possibles, excepté p p p p, apparaissent. Donc le nombre d'étapes est un multiple de 15 pour tout nombre de 4 chiffres comportant au moins un chiffre impair.

Montrons que ces 15 étapes ne peuvent être atteintes qu'avec des nombres composés uniquement de 0 et de 5. Supposons que l'on parte du

nombre qui s'écrit 1 000a + 100b + 10c + d.
On obtient successivement (modulo 10) :

$a+d$ (1) $a+b+d$ (2) $a+b+c+d$ (3) $a+b+c+2d$ (4)
$2a+b+c+3d$ (5) $3a+2b+c+4d$ (6)
$4a+3b+2c+5d$ (7) $5a+4b+3c+7d$ (8)
$7a+5b+4c+10d$ (9) $10a+7b+5c+14d$ (10)
$14a+10b+7c+19d$ (11) $19a+14b+10c+26d$ (12)
$26a+19b+14c+36d$ (13) $36a+26b+19c+50d$ (14)
$50a+36b+26c+69d$ (15)

Supposons qu'à la 15e étape, on obtienne le nombre de départ. On a alors le système

$9a + 4b + 6d = a$ (mod 10),
$6a + 9b + 4c + 6d = b$ (mod 10),
$6a + 6b + 9c = c$ (mod 10),
$6b + 6c + 9c = d$ (mod 10),
$4a + 2b + 3d = 0$ (mod 5),
$3a + 4b + 2c + 3d = 0$ (mod 5),
$3a + 3b + 4c = 0$ (mod 5),
$3b + 3c + 4d = 0$ (mod 5).

Ce système a pour solutions
$a = b = c = d = 0$ (mod 5).

Sur les 16 nombres de quatre chiffres s'écrivant à l'aide des seuls 0 ou 5, 8 sont des solutions du problème :
5 555, 5 550, 5 505, 5 055, 5 500, 5 050, 5 005, 5 000.

HS3711

a) 1 + 2 + 4 + 8 + 16 + 32 = 63. L'arbre contient donc **63 personnes**.
b) 37 = 2 × 18 + 1 ; il s'agit donc de la mère de 18.
18 = 2 × 9 ; 18 est donc le père de 9.
9 = 2 × 4 + 1 ; 9 est donc la mère de 4.
Le numéro 37 est donc celui de **la mère du père de la mère du père de mon père**.

HS3712

Il aura obtenu 2003 morceaux au bout de **286 étapes**.

HS3713

E. Aucun des nombres précédents.
En effet, le nombre précédent $2^{100} + 1$ était $2^{99} + 1$, le précédent $2^{98} + 1$, ..., et le nombre de départ $2^2 + 1 = 5$.

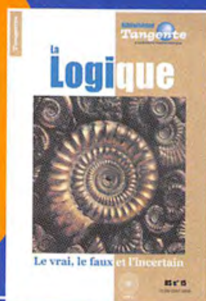
HS3714

Tout d'abord, regardons la même suite où l'on a enlevé le dernier chiffre de chaque terme. Cette suite augmente au plus de 1 à chaque étape et tend vers l'infini (car la suite de départ tend vers l'infini) donc elle parcourt tous les nombres plus grands que le nombre de départ.
À un moment, elle passera donc par un nombre constitué uniquement de chiffres impairs. Revenons à la suite originale : à ce moment-là, elle n'aura que des chiffres impairs, sauf peut-être le dernier ; soit ce dernier chiffre est pair et c'est gagné, soit il est impair **et le prochain nombre de la suite sera forcément pair**.

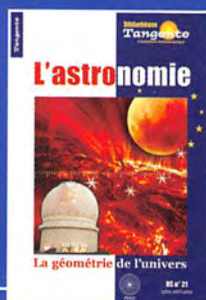
Bibliothèque

Tangente

L'aventure mathématique



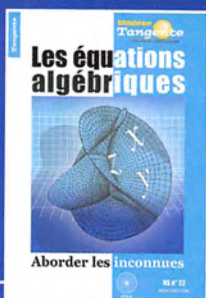
La logique



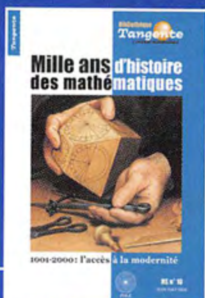
L'astronomie

Symétrie
& jeux de miroir

Hasard & probabilités

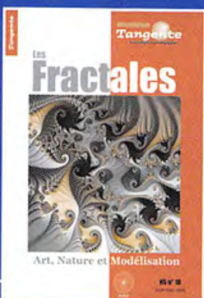


Les équations
algébriques

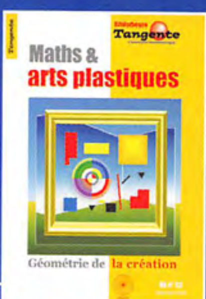


HS10
+ HS30
Toute
l'histoire
des
mathé-
matiques
en
2 tomes

1000 ans d'histoire
des mathématiques



Les fractales



Maths
& arts plastiques

Maths & musique

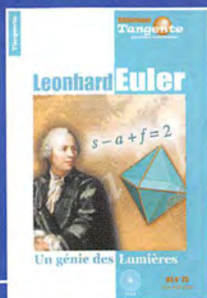
Jeux Mathématiques

Le triangle

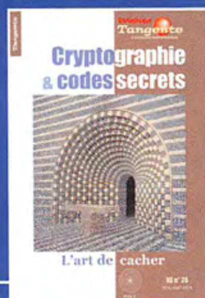
Une nouvelle façon de faire rimer mathématique avec esthétique

Ces magnifiques ouvrages en couleur de 160 pages feront l'admiration de tous les visiteurs de votre bibliothèque.

Grands mathématiciens modernes

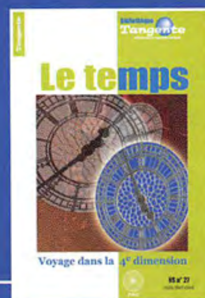


Euler

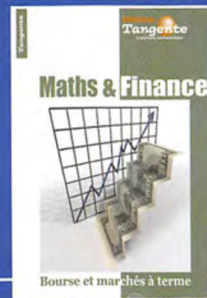


-Cryptographie & codes secrets

Histoire des mathématiques



Le temps



Maths et finance

Mathématiques & littérature

À PARAÎTRE PROCHAINEMENT :

Maths et philosophie

La combinatoire

Disponible chez votre libraire ou auprès des Éditions Pole - 80, boulevard Saint-Michel, 75006 Paris

Abonnez-vous à tangente
l'aventure mathématique

Tangente le magazine des mathématiques

Pour mieux comprendre le monde : *Tangente*
Le seul magazine au monde sur les mathématiques.
Tous les deux mois depuis 20 ans.

les hors-séries Bibliothèque Tangente

Ce sont de magnifiques ouvrages d'en moyenne 160 pages (prix unitaire 19,80 €), richement illustrés, approfondissant le sujet du dernier numéro des HS *Thématiques* de *Tangente* (HS "kiosque").
Disponibles
- chez votre libraire
- avec **l'abonnement SUPERPLUS** à un prix exceptionnel (33% de réduction).

Tangente Sup

6 numéros par an, destinés à ceux qui veulent aller plus loin. Ils traitent des dossiers du dernier *Tangente* à un niveau supérieur et initient aux grandes orientations de la science.

Les hors-séries thématiques

4 fois par an, les hors-séries Thématiques (format "kiosque") pour explorer un sujet

Ces numéros d'au moins 52 pages explorent un grand dossier de savoir ou de culture mathématiques.
Derniers parus :
Les Transformations,
Le Cercle
Les Algorithmes

Disponibles
- chez votre marchand de journaux
- ou avec **l'abonnement PLUS.**

Pour nos lecteurs les plus curieux, les articles des *Thématiques de Tangente* sont repris et complétés dans les hors-séries de la *Bibliothèque Tangente*.

Tangente Éducation

Nouvelle formule trimestrielle destinée aux enseignants
pour aborder des thèmes de pédagogie variés (les manuels scolaires, les TICE, l'erreur, les examens, les programmes......).

NOUVEAU ! Abonnement de soutien

codif : POLE **HS37**

Bulletin d'abonnement à retourner à :
Espace Tangente - 80, bld Saint-Michel - 75006 PARIS

Nom .... Prénom ....
Établissement ....
Adresse ....
Code Postal .... Ville ....
Profession .... E-mail ....

Oui, je m'abonne à	FRANCE		HORS MÉTROPOLE
	1 AN	2 ANS	
TANGENTE	❑ 32 €	❑ 60 €	❑ + 12 € par an
TANGENTE PLUS	❑ 52 €	❑ 100 €	❑ + 20 € par an
TANGENTE SUPERPLUS	❑ 82 €	❑ 160 €	❑ + 24 € par an
TANGENTE SUP*	❑ 24 €	❑ 46 €	❑ + 6 € par an
TANGENTE EDUCATION*	❑ 10 €	❑ 18 €	❑ + 2 € par an
ABONNEMENT DE SOUTIEN**	❑ 150 €	❑ 300 €	+ 0 €

* Réduction de 2 €/an en cas d'abonnement à Tangente
** Donne droit à tous les titres

Total à payer

Je joins mon paiement par (établissements scolaires, joindre bon de commande administratif) :

❑ Chèque (uniquement **payable en France**)
❑ Carte (à partir de 30 €) numéro : |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|
Date et Signature : Expiration le : ..../.........

Tangente Hors-série n° 37

Les algorithmes

Tangente

est publié par Les Éditions POLE
SAS au capital de 40 000 euros

Siège social
80 bd Saint-Michel – 75006 Paris
Commission paritaire : 1011 K 80883
Dépôt légal à parution

Directeur de Publication
et de la Rédaction
Gilles COHEN

Secrétaire de rédaction
Édouard THOMAS

Ont collaboré à ce numéro
Jacques BAIR, Michel CRITON,
Jean-Jacques DUPAS,
Bertrand HAUCHECORNE, Valérie HENRY,
Daniel JUSTENS, François LAVALLOU,
Hervé LEHNING, Bernard NOVELLI,
Jean-Christophe NOVELLI, Florent PICARD,
Jean-Alain RODDIER, Michel ROUSSELET,
Alain ZALMANSKI

Abonnements
Tél. : 01 47 07 51 15 – Fax : 01 47 07 88 13

Maquette
Guillaume GAIDOT
Claude LUCCHINI

Photos : droits réservés

Achevé d'imprimer
pour le compte des Éditions POLE
sur les presses de l'imprimerie Louis Jean
05000 GAP
Imprimé en France -
Dépôt légal 454 - Octobre 2009

Tangente

Les algorithmes

Au cœur du raisonnement

- Histoire des algorithmes
- Algorithmes élémentaires et programmation
- Performances et limites
- Au-delà de l'algorithmique

En Grèce à l'époque d'Euclide, en Chine il y a 2000 ans ou aujourd'hui à l'ère de l'informatique, les algorithmes ont vocation à expliquer, étape par étape, comment fonctionne un raisonnement. Certaines caractéristiques émergent naturellement : boucles, conditions d'arrêt, itérations, convergence, récursivité...

Les algorithmes font officiellement leur entrée dans les programmes scolaires de 2009. Le présent ouvrage couvre leurs aspects historiques, techniques et mathématiques, mais également les besoins spécifiques des enseignants (et de leurs élèves).

Pour autant, le grand public n'est pas oublié : de nombreuses questions fascinantes, en arithmétique par exemple, sont issues d'algorithmes très simples.

Prix : 19,80 €

EDITIONS POLE